Interval Based I/O: A New Approach to Providing High Performance Parallel I/O

Jeremy Logan
National Center for Computational Sciences
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
loganjs@ornl.gov

Phillip Dickens
Department of Computer Science
University of Maine
Orono, Maine, USA
dickens@umcs.maine.edu

Abstract

Providing scalable, high-performance parallel I/O for data-intensive computations is beset by a number of difficult challenges. The most often cited difficulties include the non-contiguous I/O patterns prominent in scientific codes, the lack of support for parallel I/O optimizations in POSIX, the high cost of providing strict file consistency semantics, and the cost of accessing storage devices over a network. We believe, however, that a more fundamental problem is the legacy view of a file as a linear sequence of bytes. To address this issue, we are developing a new approach to parallel I/O that is based on what we term intervals and interval files. This paper provides an overview of the interval-IO system and a set of benchmarks demonstrating the power of this new approach.

1. Introduction

Large-scale computing clusters, with thousands to tens-of-thousands of computing cores, are becoming an increasingly important component of the national computational infrastructure [1]. These large-scale computing clusters are coupled with state of the art parallel file systems such as Lustre [2], PVFS [3], and Panasas [4], which offer massive storage capabilities and are designed to provide scalable access to thousands of clients concurrently. Software systems, such as MPI (Message Passing Interface) [5], support large-scale applications executing in such extreme environments by providing sophisticated mechanisms for message passing and process management. MPI-IO is the I/O component of the MPI standard, which provides to MPI applications 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.

Taken together, these technologies have enabled an important new class of scientific applications termed data-intensive applications, which can manipulate data sets on the order of terabytes to petabytes and beyond. Such applications are capable of executing very high-resolution scientific models, completing computations that would once have been deemed intractable. This has significantly deepened our understanding of previously unexplored scientific phenomena, including, for example, climate modeling [7], earthquake modeling [6], and genomic pattern matching [8]. It has also made possible detailed animation and visualization of scientific data [9], further deepening our understanding of natural phenomena.

All of these large-scale scientific applications can manipulate massive data sets, ranging from gigabytes to terabytes and beyond, and are supported by state-of-the-art parallel file systems (e.g., Lustre [2], Panasas [4], GPFS [10], and PVFS2 [3]), which are designed to scale with increasing processor counts and data requirements. MPI-IO [5], the I/O specification of the MPI standard, further supports such data requirements by providing a powerful parallel I/O API, and high-performance implementations of the standard, which can interact with the file system to optimize access to the underlying storage.

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 capacity of even the largest, most powerful file systems available today, and are becoming a significant drag on application performance. This problem will only become worse as the number of processors required for application execution, and the size of the data sets such applications require, continue to increase. This is a critical problem on a national scale because continued scientific breakthroughs are, in many cases, dependent upon the ability to execute higher resolution scientific

models, which, in turn, requires data sets of increasing size.

There are many factors that make this problem, often referred to as the scalable I/O problem, so challenging. The most often cited issues include the I/O access patterns exhibited by scientific applications (e.g., non-contiguous I/O [11-13]), poor file system support for parallel I/O optimizations, the high cost of enforcing strict file consistency semantics [14], 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 data model, and a large component of the scalability problem is the cost of dynamically translating between the process data model and the file data model at runtime. In fact, the data model used by applications is more accurately defined as an *interval model*, where each process maintains a set of perhaps unrelated intervals. We believe that aligning these two different data models will significantly enhance the performance of parallel I/O for data-intensive scientific applications.

Over the past three years, we have been addressing the scalable I/O problem through the development of a new file model, termed *interval-files,* which are much more closely aligned with the application's access patterns than the traditional linear files. We have been developing the software infrastructure required to support this new I/O model, and have integrated it into the ROMIO implementation of the MPI-IO standard [17]. The goal of this research is to merge the power and flexibility of the MPI-IO parallel I/O interface with a more powerful *interval-based file model*. The system that supports this new file model for parallel I/O is termed the *Interval-IO system*.

Intervals are based on MPI file views, or, more precisely, the intersections of such views. Individual file views declare the file regions within which a given process will operate. The intersections of all processes' views, which is what we refer to as intervals, identify all file regions within which conflicting accesses are possible, and, by extension, those regions within which there can be no conflicts (termed shared-intervals and private-intervals respectively). This is valuable information, and can be utilized by the runtime system to increase the concurrency of file accesses and to reduce the cost of enforcing strict file consistency semantics and maintaining global cache coherence.

This paper updates and extends our earlier work presented in [26] and [38], which provided an overview of the Interval-IO system and preliminary experimental results showing the promise of this approach. Since the time of these publications, we have completed the implementation of the final two components of the system that were previously unavailable: the *distributed locking system*, which is required to support MPI-IO file consistency semantics (including Atomic Mode), and an *interval translation tool*, which is required to convert between different *interval sets*.

One important capability made possible through the addition of these two new Interval-IO components is what we term *collaborative I/O* between two applications. One example is a tile reader/writer, in which a pair of cooperating applications collaborates to visualize scientific data. In this case, there are two applications; one is a simulation that produces data to be displayed, and another that consumes the data, producing a visualization on a high-resolution display wall. Often it is the case that they are comprised of different numbers of processes reading from and writing to the same file. The applications may use different access patterns. For example, the reader may require additional "ghost cells" to manage potential overlap between the adjacent display devices. In such cases, the producer of the data writes to the file with respect to one interval set, and the consumer reads from the same file according to a different interval set.

Another important problem that the Interval I/O system addresses is the issue of producing checkpoint files in long-running scientific applications. This is a critical issue for our national laboratories and other High Performance Computing Centers, where large-scale parallel simulations can take weeks or months to complete their execution [27]. The problem is that the meantime to failure for large-scale computing systems is not measured in months, but rather in terms of hours or days. Thus long-running applications must frequently checkpoint their current state in order to be restarted in event of a failure. As noted by Bent *et al*. [27], the writing of large checkpoint files by long-running applications is placing significant stress on the parallel file systems. The magnitude of this problem will only increase as the size of the simulations, their execution time, and the supercomputing systems upon which they execute all continue to increase.

In this paper, we discuss the Interval-IO system and provide experimental results demonstrating its ability to address both of these important challenges. The rest of this paper is organized as follows. In Section 2, we provide background information relevant to the Interval I/O system. In Section 3, we provide an overview of the basic components of the system. In Section 4, we discuss the creation of interval sets. In Section 5, we provide our experimental results. We discuss related work in Section 6, and provide our conclusions in Section 7.

2. Background

2.1 MPI-IO

MPI-IO is the IO component of the MPI standard [5] that was designed to provide MPI applications with portable, high performance parallel I/O. It provides a rich and flexible API that provides to an application the ability to express complex parallel I/O access patterns in a single I/O request, and provides to the underlying implementation important opportunities to optimize access to the underlying file system. It is generally agreed that the most widely used implementation of the MPI-IO standard is ROMIO [17], [18], which was developed at Argonne National Laboratory and is included in the MPICH2 [19] distribution of the MPI standard. ROMIO provides key optimizations for enhanced performance (e.g., two-phase I/O [20] and data sieving [21]), and is implemented on a wide range of parallel architectures and file systems. The portability of ROMIO stems from an internal layer termed ADIO [17] (an Abstract Device Interface for parallel I/O) upon which ROMIO implements the MPI-IO interface. ADIO implements the file system dependent features, and is thus implemented separately for each file system.

2.2 MPI File Views

An important feature of MPI-IO is the file view [22], 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 depicts a file region in which two processes are operating, and the data for each is laid out non-contiguously on disk. The file view for Process P0 is shown, which creates a contiguous "view window" of the four data blocks it will access. Thus, the data model that P0 is using is a contiguous file region, which conflicts with the file data model.

Such non-contiguous file I/O patterns can be difficult to implement in a scalable fashion, and several techniques have been developed to address this issue (e.g., two-phase I/O [20], List I/O [35], DataType I/O [36]). The Interval-IO system, however, takes a different approach, which is to store the data on disk as it is defined in the file view. Thus each process is able to read/write its data in a single I/O operation

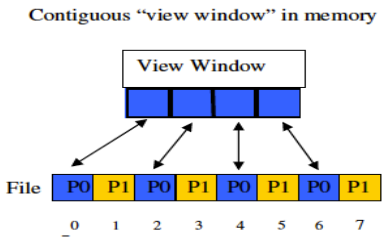


Figure 1. Example of a file view.

MPI file views play a central role in the Interval I/O system. They provide information about the file access patterns of individual processes, and, when aggregated, show exactly those file regions for which contention is possible (overlapping file views), and, by extension, those file regions in which conflicting accesses are not possible. The technique for aggregating file views is discussed in Section 4.

3. Interval I/O System

In this section, we provide a brief overview of the Interval I/O system is presented. The interval I/O system consists of five primary components: the interval integration interface, the interval cache, a distributed lock management system, an interval-based file layer, and an interval set translation tool.

3.1 The Interval Integration Interface

The interval integration interface (I^3) adapts MPI-IO calls into corresponding interval set accesses that are supported by the underlying interval-based components (cache, lock manager). Specifically, the I^3 utilizes file views set by the application to create interval sets designed to efficiently handle the application's I/O operations. It also converts file read and write operations into corresponding interval accesses based on the current interval set.

3.2 Interval Cache

The interval cache is a collaborative software cache implemented as an extension to ROMIO. The cache is designed to manage contents of the file in memory, distributed across the participating processes. The cache uses collaboration between application processes to handle MPI-IO file accesses.

Although other research has shown the potential effectiveness of a parallel software cache [23-25], this earlier work has focused on the use of block-based caches. In contrast, our system abandons the traditional block-based paradigm, a remnant of physical disk caching, in favor of an interval-based approach. The

interval cache not only provides improved performance by itself, but it also acts as an extremely fast interface to the more powerful interval-based files described below.

3.3 Distributed Locking System

We have developed a novel locking system designed to provide sequential consistency to atomic operations performed by the interval cache. The system is designed to operate as a distributed system of lock managers, each of which acts as a central manager for a specific subset of the available locks. The locks are assigned to the application processors according to the active file view (and the corresponding interval set) so that typical access patterns will require each process to interact with a relatively small number of lock managers. Our flexible design allows the number of lock managers to be determined dynamically according to the I/O pattern of the application, thus providing a mechanism to balance speed and scalability.

3.4 Interval Files

A central motivation for this research is the observation that the traditional sequential file is often not a good match for a parallel I/O environment. Therefore, a major component of this research is to provide a suitable alternative file model that eliminates the parallel I/O performance issues inherent in sequential files. Thus we introduce a virtual interval-based file layer designed to integrate seamlessly with the caching system and to provide more optimal I/O performance. The key to the design is the use of a structured, interval-based file format used to represent a given flat file by reorganizing file data to better fit the actual access pattern used by a parallel application. The organization of the interval files corresponds directly to the arrangement of cached data, with intervals from each process stored contiguously on disk. This allows the file accesses to be accomplished via large contiguous data transfers with no contention. Metadata included in the interval file allows the original flat file layout to be reconstructed when necessary. The interval files themselves are stored as flat files in an underlying file system, allowing their use regardless of the actual file system available on a particular cluster. It is important to note that a single Interval-based file is created and shared by all of the application processes.

A preliminary design of the Interval-based file format was presented in [26], although no performance data was available at that time. Recently, we have redesigned the Interval-based file format and completed its implementation. The primary difference between the two versions has to do with the placement of the data and meta-data within the file. In the previous version, all of the interval meta-data, interval data, and process meta-data were stored contiguously in the file on a process-by-process basis (i.e., all such information for Process P0 was stored contiguously in the file, followed by all such information for Process P1, and so forth). In the new design, all process meta-data is stored in the file, followed by all interval meta-data, followed by all interval data. The advantage of this approach is that all of the metadata can be read in a single I/O operation rather than one read for each process that wrote the file. This is particularly important if the file is read by a small number of processes relative to the number of writers.

3.5 Interval Translation Tool

Although the interval files provide excellent I/O performance, the files are tuned specifically to a particular file view of an application running on a particular number of processors. To achieve more general interoperability of the interval files, the final component of the Interval I/O system, called the translator, is designed to perform the migration of data from one interval layout to another. Efficient translation is accomplished by the use of an interval tree that is used to remap the intervals between source and target layouts. The translator allows a great deal of flexibility in its use; it is designed to read from or write to files, or stream data to or from an interval cache. In addition, the translator can be run on a separate set of processors (or cores) from the application performing I/O, effectively pipelining the I/O and increasing the utilization of the available hardware.

A detailed discussion of the *design* of the interval translation tool can be found in [38]. Since the time of that publication, we have implemented a prototype version of the tool, and, in Section 5.2.3, provide experimental results showing that the utilization of interval trees does, in fact, lead to highly efficient translations.

4. Interval Creation

Having provided an overview of the basic system, we now turn to a discussion of how intervals are created.

Think of a file as an integer line that extends from 0 to n – 1, where n is the number of bytes in the file. Given this representation of a file, a file view can be thought of as a set of intervals on this integer line, where each interval represents the endpoints of a file region in which the owning process will operate. These endpoints are obtained from the file views, and divide the integer line into a set of partitions termed elementary intervals [26]. Each file view can contain

multiple intervals, and as more intervals are placed on the integer line, more elementary intervals are created. Once all of the intervals (of all file views) have been added to the line, each of the resulting elementary intervals corresponds to an object.

Figure 2 depicts object creation using this technique. Figure 2.A shows three processes, their file views, and an integer line representing a 125-byte file. Figure 2.B shows the elementary intervals that are created when the endpoints of process P0, which represent the two file regions in which it will be active, are added to the integer line. In particular, it creates four elementary intervals: {0, 24}, {25, 49}, {50, 74} and {75, 125}. Figure 2.C shows the eight elementary intervals that are created when the three file views are all aggregated onto the integer line. It is these elementary intervals that are utilized by the Interval-IO system.

There are two key observations to be made about the resulting set of elementary intervals: First, the elementary intervals are non-overlapping. This allows for the development of a highly efficient interval search tree that can store and retrieve information about interval sets at runtime. It also means that any shared intervals encompass exactly the regions of the file that are shared. This means that there is no false sharing, at least with respect to the file views provided by the application.

The second key observation is that intervals can be classified as either private to a process or shared between two or more processes. This information can significantly increase performance because only shared regions require locking. It is also very important because the set of applications that can access a given shared interval is known at interval creation time. Thus each lock manager knows exactly which processes can attempt to acquire locks it controls, and the lock managers can inform such processes about where such locks are maintained. This obviates the need for a central lock manager, which can also significantly increase the performance of the locking system.

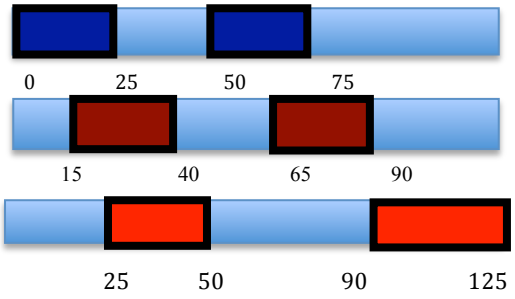


Figure 2.A. This figure depicts the file views of three processes. The rectangles represent the file regions in which the processes will be active.

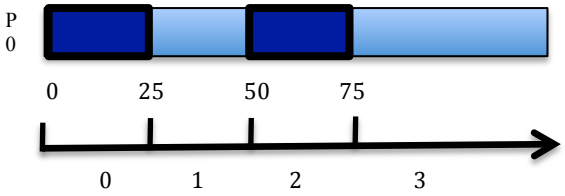
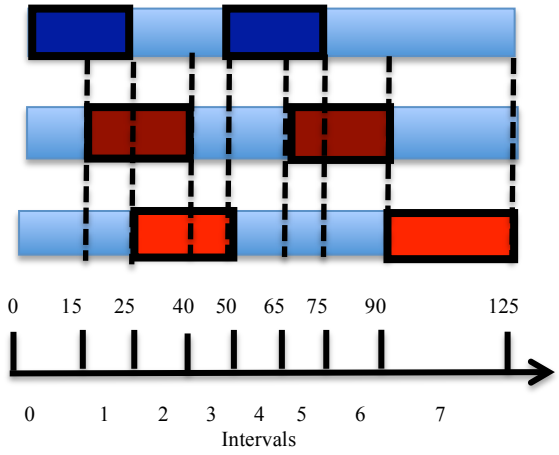


Figure 2.B. This figure depicts the elementary elements that are created when the file view of P_0 is added to the integer



5.

Figure 2.C. This figure depicts the eight intervals that are created when the file views of all three processes are added to the integer line.

Performance Studies

We now look at two important benchmarks that illustrate the performance benefits that can be obtainfrom the Interval-IO system. We first look at the FLASH-IO benchmark [16] that directly addresses the issue of writing large checkpoint files in long-running, data-intensive scientific applications. In a preliminary study [26], we showed up to a 40% improvement in performance for this benchmark (compared to native MPI-IO), which was gained simply through utilization of the interval-based caching system (Interval files were not fully implemented at that time). The improvement thus came from the fact that all writes were collected in the cache requiring only a single write to the file system when the checkpoint file was closed. In the experiments that follow, we demonstrate an *order of magnitude speedup* that is made possible by utilizing the cache *and* writing the checkpoint file as an Interval file rather than a traditional file.

We then demonstrate how the Interval-IO system supports collaborative I/O utilizing the MPI-Tile-IO benchmark [15]. We provided preliminary experimental results in [38], which demonstrated that the efficiency gained by reading from an Interval file rather than a

traditional file should outweigh the costs of the Interval translation. However, neither the distributed locking system or the Interval translation tool was implemented at that time, which significantly limited the conclusions that could be made from that earlier study. In the experimental results that follow, we demonstrate that the Interval I/O system is able fully to support such I/O collaboration because the interval translation tool is capable of efficiently converting file data from one interval set into another, and because the distributed locking system is able to maintain file consistency between the tile writer and tile reader.

5.1 FLASH I/O

The FLASH [28] simulation computes the solution of fully-compressible, reactive hydrodynamic equations, and was developed to study nuclear flashes on the surfaces of neutron stars and white dwarfs. FLASH I/O [16] is a parallel I/O benchmark that is based on the I/O kernel of the FLASH simulation. The benchmark uses identical I/O code as that used in the simulation, thus any improvements in the I/O performance of the benchmark are expected to translate directly to the FLASH simulation. The I/O workload consists of writing a checkpoint file and two plot files at each checkpoint.

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 3. 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 V0 from every block on every process are stored contiguously, followed by all of the V1's, and so forth. Thus each process writes approximately 7.5 MB per checkpoint.

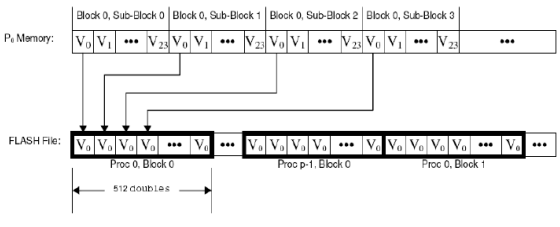


Figure 3. A simplified view of FLASH's data layout in memory and in the file. The intervals created for the FLASH simulation are represented by the dark rectangles.

Each interval contains all of the variables for a particular block on a particular process. Each interval is 4096 bytes, and the file will contain 1920 intervals for each process.

The FLASH simulation and benchmark are implemented in Fortran using the Parallel HDF5 Library for I/O [29] (Hierarchical Data Format). Parallel HDF5 is a high level I/O library that provides a structured file format that is portable across multiple file systems. While HDF5 utilizes MPI-IO as the underlying I/O mechanism, the user does not access MPI-IO directly, but rather through a higher-level API provided by the library. In order to use the interval-file format, we wrote a version of the benchmark in C that executes on top of the Interval-IO system. Our version of the FLASH I/O benchmark uses exactly the same file layout and memory layout as the original version, but writes the checkpoint data to disk as interval files.

The FLASH I/O benchmark has also been used to evaluate the performance of the PNetCDF parallel I/O library [30]. Similar to Parallel HDF5, PnetCDF provides a structured file format that is portable across multiple file systems. It also uses MPI-IO as the underlying I/O mechanism, which is only accessed by the user through a high-level interface provided by the library.

In the experiments that follow, we compare the performance of the Interval I/O system with both of these important I/O libraries.

5.1.1 Experimental Design

We performed these experiments on the Lonestar cluster housed at the Texas Advanced Computing Center. At the time these experiments were conducted, Lonestar consisted of 1300 Dell PowerEdge 1955 blades (nodes). Each node contained two Xeon Intel Duo-Core 64-bit processors running at 2.66 Ghz and had 8 GB of DDR-2 memory. The nodes were connected by an InfiniBand interconnect using a fat tree topology. Lonestar was attached to a 68 TB Lustre file system comprised of 16 Dell 1850 I/O data servers (Lustre OSSs).

In these experiments, we compared the time required to write the FLASH I/O checkpoint files using the three different file formats discussed above: Parallel HDF5 [29], PNetCDF [30], and interval-files using the Interval-IO system. We varied the number of processors between 16 and 256, with one FLASH I/O process per processor.

5.1.2 Experimental Results

The results of these experiments are shown in Figure 4. As can be seen, the Interval I/O system performed

significantly better than either Parallel HDF5 or PNetCDF. In fact, it was able to write the checkpoint file approximately nine times faster than PNtCDF, and five times faster than Parallel HDF5. Visually, it appears as though the write times were linear in the number of processors for Parallel HDF5 and PNetCDF, and logarithmic in the case of Interval I/O, at least for up to 256 processors. Unfortunately, we were unable to acquire more than 256 processors on Lonestar to observe whether this trend would continue as the problem size further increased.

There are several reasons why our Interval I/O System performs so well with the FLASH-IO benchmark. FLASH-IO benefits from caching since a number of separate MPI-IO write operations are performed, and the results of the writes can be combined in the cache, which generates fewer file system operations than would be required without caching. Furthermore, our approach avoids false sharing in the cache by using intervals as the cache unit. Another factor is that we are able to eliminate noncontiguous file system accesses because the intervals written by each process are stored together in the interval file. Finally, we avoid all locking overhead by detecting that the pattern used by FLASH-IO contains only private intervals.

These results provide strong support for our hypothesis that parallel I/O performance can be significantly improved by utilizing the Interval-based file format. In hindsight, it would have been very informative to look at the performance of the Interval-IO system when the data was written out using the traditional file format in addition to the Interval-based file format. This would help provide information as to the relative contribution of (a) collecting all writes in the cache, and (b), writing the file as an Interval file, to overall performance gains. As noted above, the experiments provided in [28], where the cache was utilized but the data was written out as a traditional file, provide some insight into this issue and suggest that utilizing the Interval-based file format is the primary contributor to increased parallel I/O performance. However, more studies are needed to draw definitive conclusions.

5.2 Collaborative I/O

The MPI-Tile-IO benchmark [15] models quite well the idea of collaborative I/O discussed above. A producer application consists of a set of processes that generate a dense two-dimensional set of pixel data that is written to a shared file (MPI-Tile-Writer), and the consuming application consists of a set of processes that read the pixel data from the shared file and display the data on a tiled wallboard (MPI-Tile-Reader). The tiled wallboard consists of a set of individual monitors that together display the entire image. Adjacent monitors (in the horizontal and vertical directions) share a column of pixel data to help blend the individual components of the image into a smoother aggregate image. Such columns of shared data are often referred to as "guard cells".

For clarity of presentation, assume that both applications consist of four processes, that the image data is displayed on a tiled wallboard with four monitors, and that there is a one-to-one mapping between the tile reader processes and the monitors on the display wall. This is shown in Figure 5. The dashed lines in the figure represent the pixel data that is shared between processes P_0 and P_1. While we do not show the interval sets that would be created for the application processes, we note that all of the intervals associated with the writer processes are private. The interval sets associated with the reader processes, however, contain both private and shared intervals.

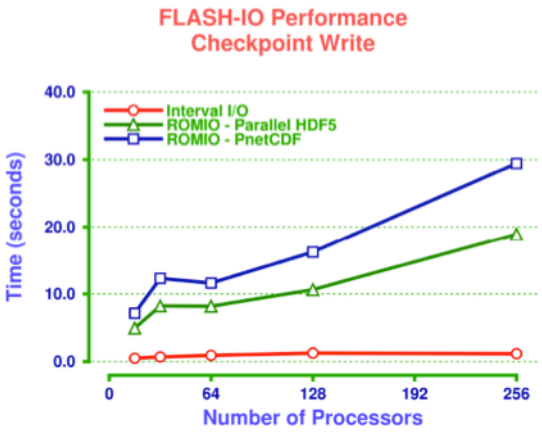


Figure 4. This graph shows the time required to produce checkpoint files as a function of the number of processors and the data file format.

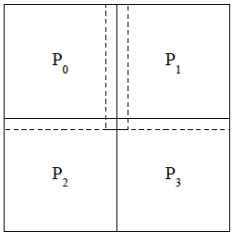


Figure 5. The figure shows four display devices, and the pixel data that is shared between processes P_0 and P_1.

These file access patterns are quite challenging for current parallel I/O implementations. There are three

reasons for such difficulties. First, there are a large number of noncontiguous file accesses, each of which requires a separate I/O operation. Second, the file regions written by producer processes overlap with the file regions written by other producer processes. This can lead to the serialization of file accesses due to false sharing. Third, complex locking is required to access the guard cells.

Utilizing the Interval-IO system to implement the producer and consumer applications is also problematic. It is expected that the I/O performed by the writer processes would be efficient because they would write their data to the file in the same way it is stored in their caches. Thus each process could write all of its data to disk in a single I/O operation.

The issue is more complex in the case of the consumer processes, however, because the data is not stored on disk according to their interval sets. In fact, the consumer processes would be unable to access the data at all because the file metadata describes the interval sets of the producer processes. Thus in order to utilize the Interval-IO system an additional translation step is required that converts the producer's interval set into one that can be efficiently read (and understood) by the consumers. The question, then, is whether the benefits of using interval-IO outweigh the costs of performing the translation. We conducted two sets of experiments to gain insight into this issue.

5.2.1 Experimental Design

The first set of experiments was designed to get a handle on the possible benefits that could be obtained by using interval files with the MPI-Tile-IO benchmark. We utilized the Interval-IO system for the tile writer application, and handcrafted a second interval file that matched the interval set of the tile reader. We then studied their performance separately, and compared the results to those obtained using unmodified ROMIO. If there were not a significant difference in performance without the overhead of the translation tool, then it would stand to reason that the Interval-IO system would not be effective in supporting cooperative I/O.

We performed these experiments on the Ranger cluster at the Texas Advanced Computing Center. At the time of these experiments, Ranger consisted of 3936 SunBlade x6420 blade nodes, each of which contained four quad-core AMD Opteron processors for a total of 62,976 cores. Each blade was running a 2.6.18.8 x86-64 Linux kernel from kernel.org. Ranger was attached to a 1.73 petabyte Lustre file system comprised of 72 Sun x4500 disk servers, each containing 48 SATA drives.

We varied the size of the display wall between a 3x3 and an 8x8 array of monitors. We also maintained a one-to-one mapping between the number of monitors and the number of readers and writers.

5.2.2 Experimental Results

The results of these experiments are shown in Figure 6. As can be seen, the read time was reduced by as much as 35% (in the 8x8 configuration) when compared to ROMIO. More significantly, however, the write time was reduced by as much as 90% in the 7x7 configuration. It should be noted that ROMIO was unable to complete write operations for the 8x8 configuration within the 1-hour allotted run time.

As noted above, the reasons for the relatively poor performance of ROMIO with respect to the tile writer include a large number of noncontiguous I/O operations and the serialization of write operations due to false sharing. In the case of the Interval I/O system, all of the writing was performed without acquiring locks since the system detected that none of the intervals were shared. Also, all of the noncontiguous writes were collected in the cache and only a single I/O operation was required to write the data to disk.

5.2.3 Translation Time

The second set of experiments investigated the costs associated with performing the translation between the interval sets of the tile writer processes and those of the tile reader processes. While the basic functionality of the translator tool has been implemented, it had not been integrated into the Interval I/O system at the time of this writing. We thus executed the translator as a stand-alone application, and recorded the time required to perform the interval set translation for the same configurations used in the first experiments. The results are shown in Figure 7.

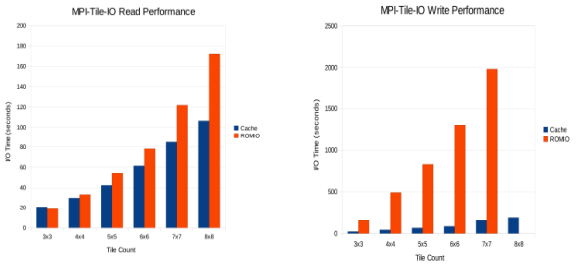


Figure 6. This graph shows the time taken to perform the I/O operations for the tile reader and writer as a function of the parallel I/O system and system configuration.

As can be seen, the amount of time required to perform the translations was quite reasonable, and was more than offset by the significant performance increases obtained through the utilization of interval files. We conclude that the Interval I/O system is a promising approach to supporting collaborative I/O.

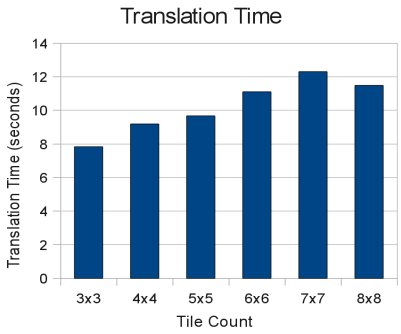


Figure 7. This graph shows the costs of performing the translation between interval sets as a function of the system configuration.

6. Related Work

The DACHe project [24] is also exploring the use of a caching system to improve the performance of parallel I/O. The DACHe system is a block-based, client-side cache designed to use remote memory access (RMA) to perform cache management across cluster nodes. The primary difference between DACHe and the interval-based cache developed in this research is that DACHe caches fixed size blocks while our system caches intervals. Because DACHe cannot distinguish between shared and private data, it must provide mutual exclusion for every data block access, while only shared intervals require locking in our system. Utilizing intervals as the basic unit of caching can also significantly reduce false sharing. Both systems can provide increased write performance by collecting writes in the cache.

View I/O [32] provides another approach to increasing parallel I/O performance based on file views. Like Interval I/O, View I/O makes use of MPI-IO file views to optimize transfers between the application and the file system. It combines smaller, noncontiguous accesses into large chunks, and uses associated metadata to allow the data to be later reorganized into linear file order.

One primary difference between View I/O and Interval I/O is that View I/O requires support from the underlying file system to reorganize file data while Interval I/O is implemented at the application layer and requires no file system support. While View I/O reorganizes data at the I/O disks, we store the data in the same format as it appears in the cache, along with additional metadata that allows the data to be reorganized (if necessary) when the file is read.

Sehrish, Wang, and Thakur [35] developed a conflict detection algorithm to minimize the locking overhead when an application is executing in Atomic Mode. This is essentially equivalent to detecting shared and private intervals in the Interval I/O system, although the techniques for doing so are different. However, their system does not provide an efficient distributed locking system in the case of shared regions, and does not provide the significant performance enhancements that come with using interval files.

Other techniques have been developed specifically to address the issue of noncontiguous file accesses, including List I/O [35] and Data Type I/O [36]. While these techniques can reduce the number of file system accesses that result from noncontiguous I/O patterns, they fail to provide a conflict detection mechanism to reduce the overhead of locking. Also, the Interval I/O system largely obviates the need for such approaches since the intervals belonging to a process are stored contiguously on disk.

The Adaptable IO System (ADIOS) project [33] is also addressing the IO performance limitations of HPC systems. ADIOS exposes a high level API that supports structured, self-describing data. It provides a variety of different IO methods, which are selectable via an XML-based IO configuration file. Similar to our use of interval files, ADIOS relies on an internal file format called BP to achieve best performance.

A software cache, which is a significant component of the Interval-IO system, is absent from ADIOS. However, ADIOS does support several staging methods (e.g., DataStager [39] and DataSpaces [40]), which provide in memory data to the application as an external component. Interval-IO differs from ADIOS most significantly in its support of the MPI-IO interface, which, as noted above, is one of the key goals of this research. This requires the Interval-IO system to support random access and the MPI-IO file consistency semantics.

The Parallel Log-structured File System (PLFS, [27]) is a virtual FUSE file system designed specifically to optimize the writing of application checkpoint data. It is a “virtual interposition layer” that sits between the parallel application and the underlying parallel file system. PLFS takes application data that the processes are writing to a shared file, and rearranges it such that each process is actually writing to its own independent file.

The goal of rearranging application data to increase parallel I/O performance is common to PLFS and the Interval-IO system. As with ADIOS, however, one of the major differences in the two systems is that

Interval-IO is implemented within MPI-IO itself and fully supports the MPI-IO interface. This includes support for MPI-IO file consistency semantics, which could be, but is not currently, implemented within PLFS. Also, the Interval-IO system can support functionality such as I/O cooperation between applications that is not a concern of PLFS. Finally, Interval-IO creates a single shared file rather than one file per process as is done in PLFS.

7. Conclusions

In this paper, we have introduced the Interval I/O system and provided experimental results showing the effectiveness of this new approach. The primary remaining task is to integrate the translation tool into the Interval I/O system. Once this is accomplished, we plan to investigate more fully the concept of cooperative I/O.

8. Acknowledgements

This work was supported in part by grant number 0702748 from the National Science Foundation.

8. References

[1] "Top 500 Supercomputing Sites."[Online]. Available: http://www.top500.org/. [Accessed: 09-Mar-2010].

[2] "Lustre File System - Overview."[Online]. Available: http://www.oracle.com/us/products/servers-storage/storage/storage-software/031855.htm. [Accessed: 09-Mar-2010].

[3] P. H. Carns, W. B. L. III, R. B. Ross, and R. Thakur, "PVFS: A Parallel File System for Linux Clusters," in *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, GA, 2000, p. 317–327.

[4] "Panasas."[Online]. Available: http://www.panasas.com.

[5] Message Passing Interface Forum, "MPI-2: Extensions to the Message-Passing Interface," *MPI-2: Extensions to the Message-Passing Interface*. [Online]. Available: http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[6] G. Hernandez, "Large scale parallel and distributed simulations and visualizations of the Olami-Feder-Christiensen earthquake model," in *Parallel and Distributed Processing Symposium., Proceedings 15th International*, 2001.

[7] J. B. Drake, P. W. Jones, and G. R. Carr, "Overview of the Software Design of the Community Climate System Model," *International Journal of High Performance Computing Applications*, vol. 19, no. 3, pp. 177-186, Aug. 2005.

[8] X. Ma and A. Choudhary, with A. Ching, W. and H., "Exploring I/O Strategies for Parallel Sequence Database Search Tools with S3aSim.," in *Proceedings of the 15th International Symposium on High Performance Distributed Computing*, 2006.

[9] S. Klasky, S. Ethier, Z. Lin, K. Martins, D. McCune, and R. Samtaney, "Grid -Based Parallel Data Streaming implemented for the Gyrokinetic Toroidal Code," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003, p. 24.

[10] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters.," in *Conference on File and Storage Technologies*, IBM Almaden Research Center, San Jose, California, 2002.

[11] P. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed, "Input/Output Characteristics of Scalable Parallel Applications," *IN PROCEEDINGS OF SUPERCOMPUTING \uc0\u8217{}95*, 1995.

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

[13] A. Ching, A. Choudhary, W.-k Liao, L. Ward, and N. Pundit, "Evaluating I/O Characteristics and Methods for Storing Structured Scientific Data," presented at the the 20th International Parallel and Distributed Processing Symposium (IPDPS), 2006.

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

[15] "Parallel I/O Benchmarking Consortium."[Online]. Available: http://www.mcs.anl.gov/research/projects/pio-benchmark/.

[16] "FLASH I/O benchmark routine -- parallel HDF 5," *FLASH I/O benchmark routine -- parallel HDF 5*. [Online]. Available: http://flash.uchicago.edu/~zingale/flash_benchmark_io/. [Accessed: 13-Feb-2010].

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

[18] 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.

[19] "MPICH2: High-performance and Widely Portable MPI."[Online]. Available: http://www.mcs.anl.gov/research/projects/mpich2/. [Accessed: 09-Mar-2010].

[20] R. Thakur and A. Choudhary, "An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays," *Scientific Programming*, vol. 5, no. 4, pp. 301-317, Winter.

[21] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO," in *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, 1999, p. 182.

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

[23] K. Coloma, A. Choudhary, L. Ward, E. Russell, and S. Tideman, with Wei-keng Liao, "Collective caching: application-aware client-side file caching," in *HPDC-14. Proceedings. 14th IEEE International Symposium on High Performance Distributed Computing, 2005.*, Research Triangle Park, NC, USA, pp. 81-90.

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

[25] K. Coloma, A. Choudhary, W.-k Liao, L. Ward, E. Russell, and N. Pundit, "Scalable High-level Caching for Parallel I/O," in *The 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004.

[26] J. Logan and P. M. Dickens, "Using Object Based Files for High Performance Parallel I/O," in *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, 4th IEEE Workshop on*, 2007, pp. 149-154.

[27] J. Bent et al., "PLFS: a checkpoint filesystem for parallel applications," in *SC Conference*, Los Alamitos, CA, USA, 2009, vol. 0, pp. 1-12.

28] B. Fryxell et al., "FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes," *The Astrophysical Journal Supplement Series*, no. 131, pp. 273-334, Nov. 2000.

[29] "Parallel HDF5."[Online]. Available: http://www.hdfgroup.org/HDF5/PHDF5/. [Accessed: 07-Feb-2009].

[30] A. Choudhary et al., with Jianwei Li and Wei-keng Liao, "Parallel netCDF: A High-Performance Scientific I/O Interface," in *Supercomputing, 2003 ACM/IEEE Conference*, 2003, p. 39.

[31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*, 2nd ed. The MIT Press, 2001.

[32] F. Isaila and W. F. Tichy, "View I/O: improving the performance of non-contiguous I/O," in *the Third IEEE International Conference on Cluster Computing*, 2003, pp. 336-343.

[33] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan, "Adaptable, metadata rich IO methods for portable high performance IO," in *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, 2009.

[34] S. Sehrish, J. Wang, and R. Thakur, "Conflict Detection Algorithm to Minimize Locking for MPI-IO Atomicity," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2009, pp. 143-153.

[35] 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*, Atlanta, Georgia, United States, 1999, pp. 23-32.

[36] A. Ching, A. Choudhary, W.-k Liao, R. Ross, and W. Gropp, "Efficient Structured Data Access in Parallel File Systems," in *the IEEE International Conference on Cluster Computing*, 2003.

[37] Logan, J. The Interval I/O System for High Performance Parallel I/O. Ph.D. Dissertation, Department of Computer Science, University of Maine.

[38] Logan, J. and P. Dickens. "Improving I/O Performance through the Dynamic Remapping of Object Sets," in IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, 21-23 September 2009, Rende (Cosenza), Italy.

[39] Abbasi, H., *et al.*, "DataStager: Scalable Data Staging Services for Petascale Applications," in HPDC'09, June 11–13, 2009, Munich, Germany.

[40] Docan, C., Prashar, M., and S. Klasky, "DataSpaces: An Interaction and Coordination Framework for Coupled Simulation Workflows," in *HPDC'10*, June 20–25, 2010, Chicago, Illinois, USA.`