2010

Improving Parallel I/O Performance Using Interval I/O

Jeremy Logan

IMPROVING PARALLEL I/O PERFORMANCE USING INTERVAL I/O

By

Jeremy Logan

B.S. University of Southern Maine, 2001

M.S. University of Maine, 2006

A THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

(in Computer Science)

The Graduate School

The University of Maine

December, 2010

IMPROVING PARALLEL I/O PERFORMANCE

USING INTERVAL I/O

By Jeremy Logan

Thesis Advisor: Dr. Phillip Dickens

An Abstract of the Thesis Presented
in Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy
(in Computer Science)
December, 2010

Today's most advanced scientific applications run on large clusters consisting of hundreds of thousands of processing cores, access state of the art parallel file systems that allow files to be distributed across hundreds of storage targets, and utilize advanced interconnections systems that allow for theoretical I/O bandwidth of hundreds of gigabytes per second. Despite these advanced technologies, these applications often fail to obtain a reasonable proportion of available I/O bandwidth. The reasons for the poor performance of application I/O include the noncontiguous I/O access patterns used for scientific computing, contention due to false sharing, and the somewhat finicky nature of parallel file system performance. We argue that a more fundamental cause of this problem is the legacy view of a file as a linear sequence of bytes. To address these issues, we introduce a novel approach for parallel I/O called Interval I/O.

Interval I/O is an innovative approach that uses application access patterns to partition a file into a series of intervals, which are used as the fundamental unit for subsequent I/O operations. Use of this approach provides superior performance for the noncontiguous access patterns which are frequently used by scientific applications. In addition, the approach reduces false contention and the unnecessary serialization it causes. Interval I/O also significantly increases the performance of atomic mode operations. Fianlly, the Interval I/O approach includes a technique for supporting parallel I/O for cooperating applications. We provide a prototype implementation of our Interval I/O system and use it to demonstrate performance improvements of as much as 1000% compared to ROMIO when using Interval I/O with several common benchmarks.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

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 [6], earthquake modeling [7], 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 [10].

The problem, however, is that despite the impressive computational and data-storage capabilities of these large clusters, 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. This problem, known as the *scalable I/O problem* [11,12], is of critical importance because continued scientific discovery is in many cases dependent upon the ability to execute more complex and higher resolution models.

There are many factors that make 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 [13-15]), poor file system support for parallel I/O optimizations [16,17], enforcing strict file consistency semantics [18], and the latency of accessing I/O devices across a network. However, it is our contention that a more fundamental problem, whose solution would go a long way toward solving all of these problems, is the legacy view of a file as a linear sequence of bytes. The problem is that application processes rarely access their data in a way that matches this file model, and a significant portion of the scalability problem is the high cost of dynamically translating between the process data model and the file data model at runtime. In fact, the data model used by application processes is more accurately described as an *object model,* where each process maintains a set of (perhaps) unrelated objects. In this new file model, each object corresponds to a file region that is itself contiguous in the file, where the set of objects belonging to a given process are not (necessarily) so.

This research is addressing the scalable I/O problem by developing this new file model and the software infrastructure required for its support. The approach is based on what we term *intervals*, which are defined in such a way as to encode critical information about an application's I/O access patterns. This information is leveraged by the runtime system to significantly increase the parallelism of file accesses, and to reduce the costs associated with enforcing strict file system semantics and maintaining global cache coherence.

This document provides details of the new approach that we term *interval I/O*, discusses the infrastructure that supports this new I/O paradigm, and provides a large collection of experimental results showing that interval-IO can outperform current, state-of-the-art techniques by over an order of magnitude.

1.1 Background

This section includes related background material that will prepare the reader for the subsequent discussion of Interval I/O.

1.1.1 POSIX and Sequential I/O

The most commonly used file system interface (API) is POSIX (Portable Operating System Interface) [19], which was designed for serial file access by a single process. It provides basic functionality for a single process (e.g., system calls for reading from and writing to a file, and seeking to a given location), but provides no specific support for parallel I/O. This makes providing high-performance I/O in a parallel environment quite difficult, and has spawned significant research activity aimed at overcoming (or providing workarounds for) the limitations of the POSIX API [19].

1.1.2 Parallel File Systems

Parallel file systems are designed to concurrently provide access to massive quantities of data for thousands to tens-of-thousands of clients. Scalability is achieved by striping large files across a number of individual disks, each of which may be accessed separately and in parallel, thus increasing the potential I/O throughput.

Figure 1.1 illustrates file striping in a parallel file system. The long rectangle at the top represents a large file divided into six equal-sized sections called stripes. The three cylinders represent storage targets (disks) to which the file stripes are assigned. The stripes are assigned to the disks in a round-robin manner. In this case, striping data across three disks provides, in the best case, three times the I/O throughput as compared to a single disk.

1.1.3 Parallel I/O

To take advantage of the additional bandwidth provided by parallel file systems, a technique known as parallel I/O has been developed. Parallel I/O involves multiple application processes collaborating on an I/O operation involving a single shared file. This collaboration allows the additional I/O capacity provided by a parallel file system to be used by a parallel application.

1.1.4 MPI

The Message Passing Interface (MPI) is a very widely used specification [5] for supporting the development of parallel applications on high-performance computing clusters. MPI offers a standard interface that allows portability between systems with different operating systems, memory models, or interconnection networks. In addition to providing management of application processes, MPI offers a rich set of communication primitives that are made available through a standard API. The latest version of the interface, MPI-2, includes additional features such as dynamic process management and support for parallel I/O, which is discussed in the next section.

There are a variety of implementations of the MPI specification. One of the more widely used of these implementations, and the one we have chosen for experimentation, is MPICH2. MPICH2 was developed at Argonne National Laboratory, and provides a portable, high performance implementation of the MPI-2 standard.

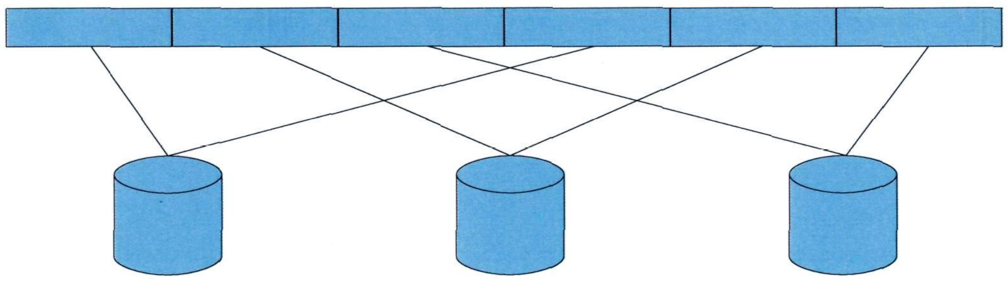

Figure 1.1. File Striping on a Parallel File System

1.1.5 MPI-IO

MPI-IO is the I/O component of the MPI standard that was designed to provide MPI applications with portable, high performance parallel I/O. It is a rich and flexible parallel I/O API that allows an application to express complex parallel I/O access patterns in a single I/O request, and provides to the underlying implementation important opportunities to optimize accesses to the underlying file system. As with MPI, the details of an MPI-IO implementation are left to the implementer; any optimizations are permitted, as long as the implementation provides the functionality described in the specification.

1.1.6 MPI File Views

An important feature of MPI-IO is the file view, which is set by each of the processes that open a file. The file view 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.

A simple example of a file view is shown in Figure 1.2. In this example, there are three processes sharing a file. For each process, the first rectangle represents the shared file and illustrates how the processes' data is laid out on the disk. The second rectangle represents the processes' file view. As can be seen, the file view maps the non-contiguous regions within which the process will operate onto a contiguous view window. Note that two or more processes access some of the file regions and that some file regions are accessed by only a single process.

It is important to note that setting a file view is a *collective operation*, which means that all processes sharing a given file must participate in the operation. Once the collective call is completed, the runtime system has information about the file access patterns of each process. When appropriately aggregated, the collection of file views shows exactly those regions in which contention is possible, and, by extension, those regions for which contention is not possible. This is very valuable information that can be utilized by the runtime system to significantly improve I/O performance. How file views are aggregated to provide such information, and how the information is utilized by the runtime system, are discussed in following sections.

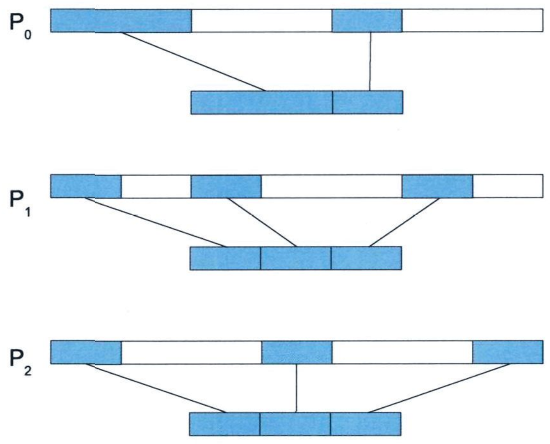

Figure 1.2. A file view example showing both shared and private regions

1.1.7 ROMIO

It is generally agreed that the most widely used implementation of MPI-IO is ROMIO [20-23], which was developed at Argonne National Laboratory and which is included in the MPICH2 [24] distribution of the MPI standard. ROMIO provides key optimizations for enhanced performance (e.g., two-phase I/O [22,25,26] and data sieving [20,22,27]), and is implemented on a wide range of parallel architectures and file systems.

The portability of ROMIO stems from an internal layer termed ADIO [21] (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.

There are several reasons that we have chosen to utilize ROMIO in this research, including:

- ROMIO is a high-performance implementation of the MPI-IO standard and includes several important optimizations for parallel I/O.
- ROMIO consists of freely available code licensed under an open source license.
- The portability of ROMIO allows flexibility in testing our work on a variety of file systems with minimal additional effort.
- Leveraging ROMIO's existing high-quality I/O infrastructure, allows us to focus on specific improvements to the system rather than building a system from scratch.

- Use of the most widely used MPI-IO implementation insures maximum impact of these results.

1.2 The Scalable I/O Problem

The massive I/O requirements of large-scale data-intensive applications represent a significant bottleneck in application performance. This is a critical problem for scientific applications, and solving the problem has been a focus of significant research activity [12,14,22,25,27-39]. As noted above, the most often cited roadblocks to achieving high-performance I/O include the I/O access patterns of scientific applications, poor file system support for parallel I/O optimizations, enforcing strict file consistency semantics, and the latency of accessing I/O devices across a network. In this section, we describe these issues, and, in subsequent sections, show how Interval-IO addresses these challenges.

1.2.1 I/O Access Patterns of Scientific Applications

It is quite common for parallel applications to perform file accesses that address a large number of noncontiguous file regions [20,13,40]. The problem is that such access patterns often result in a large number of small I/O requests, incurring the high cost of I/O on each such request. A simple example of such a noncontiguous access pattern is shown in Figure 1.3. In this example, a two-dimensional array is read from a shared file by an application consisting of four processes. As can be seen, the array is partitioned in two dimensions among all four processes. To read its array data from the file, each process must make two separate I/O requests. For instance, process 0 must perform one

I/O operation to request contiguous blocks (0-1), and a second request to access contiguous blocks (4-5). This may be a negligible cost when just two accesses are required, but noncontiguous accesses often consist of large numbers of separate file regions, with access overhead costs increasing in proportion to the number of such regions.

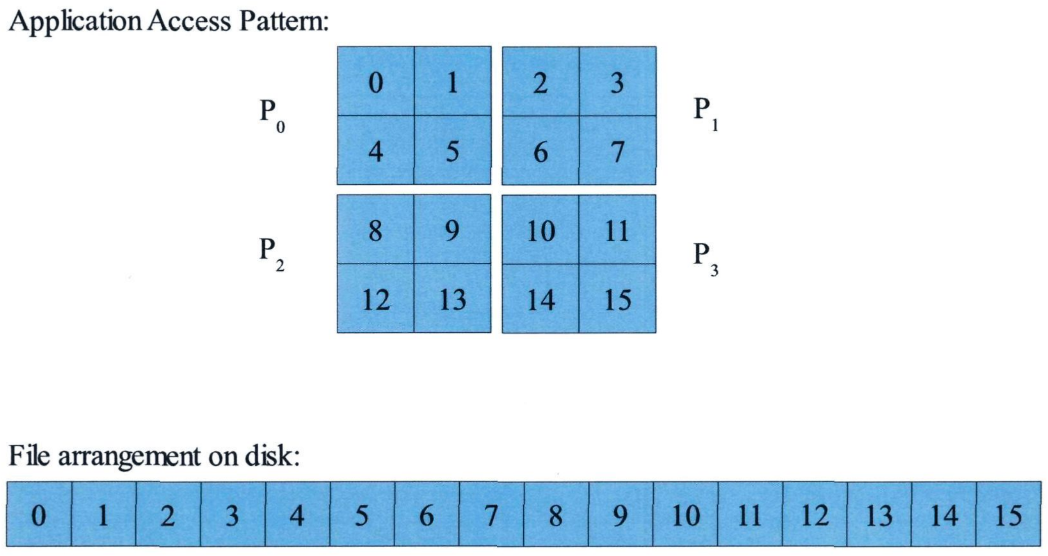

Figure 1.3. Mapping a noncontiguous access pattern onto the disk

Noncontiguous I/O operations provide one example of the richness and power of the MPI-IO interface. One advantage of using the parallel I/O API is that it can specify non-contiguous I/O requests in a single operation (through the file view mechanism). More importantly, the runtime system can, in many cases, use the information available in the file views to optimize accesses to the parallel file system. In fact, developing optimizations for non-contiguous I/O operations has been one of the most well-researched problem areas in parallel I/O.

1.2.2 Strict File Consistency Semantics

The consistency semantics associated with accessing a shared file define the outcome of multiple concurrent accesses to that file. One of the most significant challenges in providing high-performance parallel I/O is supporting strict file consistency semantics in a scalable way. MPI provides what is termed *atomic mode*, which requires *sequential consistency* of file accesses. Sequential consistency requires that the result of a set of I/O operations be as if they were performed in some serial order consistent with program order (although the particular ordering is not defined), and that each such access appears atomic. A simple example may help to clarify this idea.

Figure 1.4 provides an example with four processes, where three of the processes (P_0, P_1, and P_2) perform simultaneous writes to a shared file while using atomic mode. P_3 then reads all four file blocks. The right-hand column of the figure shows several possible outcomes of P_3's read, including three valid outcomes and three invalid results. In the first valid case, the outcome is equivalent to the result of one of two different serial orderings of the write operations, either (P_0, P_1, P_2), or (P_1, P_0, P_2). The second valid outcome is equivalent to the result of the serial ordering (P_0, P_2, P_1). The third valid outcome is equivalent to the result of the serial ordering (P_1, P_2, P_0). None of the invalid results could have been a result of any sequential ordering of the write operation.

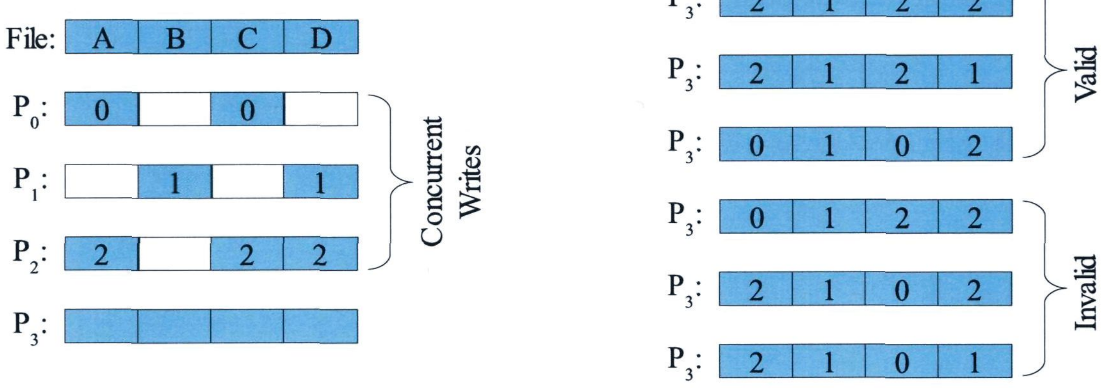

Figure 1.4. The consistency semantics of MPI-IO Atomic Mode

The reason that providing atomic mode accesses in MPI-IO is so costly is that it (generally) requires file locking for its implementation. The use of file system locks can lead to any number of bad results, including false contention arising from extent locking, complete serialization of accesses when only whole file locks are available, to a complete inability to provide atomic accesses for file systems that do not provide any locking mechanism at all.

1.2.3 Lack of File System Support for Parallel Optimizations

The most commonly used file system interface is POSIX [19], which provides access to files through a simple set of commands designed for serial access to a file by a single process. Unfortunately, the POSIX interface does not provide any specific support for underlying parallel optimizations, such as, for instance, a mechanism for expressing general noncontiguous operations.

1.2.4 A More Fundamental Problem

All of these issues stem from a more fundamental problem with parallel I/O, namely that the legacy view of a file as a linear sequence of bytes is poorly suited to file I/O in a parallel environment. The main problem is that applications don't access data in a way that matches the standard linear file model. The complex I/O access patterns utilized by parallel applications demands a more sophisticated approach.

1.3 Interval I/O

Our research directly addresses 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 *interval-based* file model. Toward this end, we are developing an *interval-based I/O system* that serves as an interface between MPI applications and interval-based files. The system is guided by MPI-IO file views [41], or, more precisely, the intersections between such views. These intersections, which we term *intervals*, 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 intervals* and *private intervals* 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.

Five major advantages are realized from viewing parallel I/O operations from this perspective:

- Increasing I/O parallelism by eliminating false sharing
- Allowing file system accesses to be performed using an optimal access pattern regardless of the access pattern used by the application
- Introducing a mechanism to support I/O cooperation between applications
- Introducing an efficient technique for distributed lock management
- Reducing lock overhead by detecting private intervals

1.4 System Design

The interval-based I/O infrastructure consists of five primary components: an interval integration interface, an interval cache, a distributed lock management system, an interval-based file layer, and an interval set transformation tool. These components are shown in Figure 1.5 in the context of the MPI software stack. We discuss each of these components in turn.

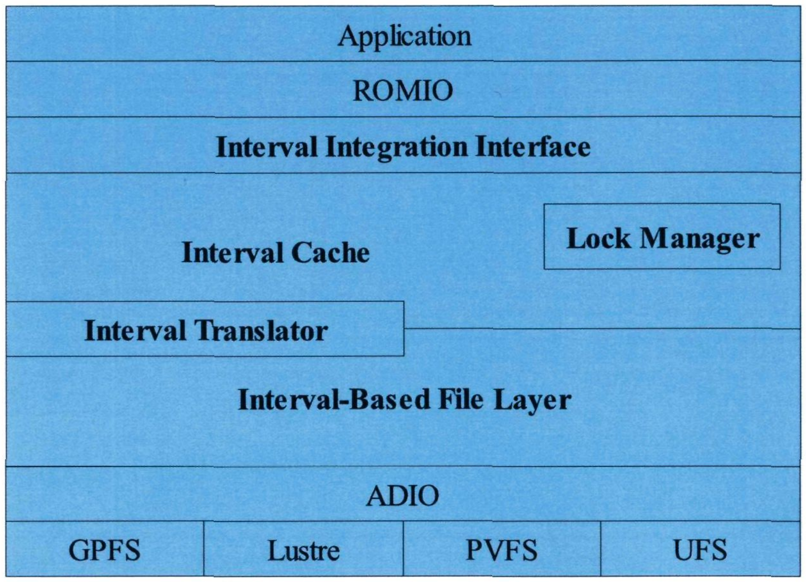

Figure 1.5. Components of the Interval I/O system

1.4.1 Interval Integration Interface

The function of the Interval Integration Interface (I^3) is to perform the translation of 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.

1.4.2 Interval-Based Cache

The interval cache is a collaborative interval-based 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 groups have shown the potential effectiveness of a parallel software cache [28-30], 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 later in this document.

1.4.3 Distributed Lock Manager

We have developed a novel locking system designed to provide sequential consistency to atomic operations performed by our interval cache. The system is designed to operate as a distributed system of lock managers, each acting 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.

1.4.4 Interval-Based File Layer

A central motivation for this research is the assertion that the traditional flat file[1] is 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.

1.4.5 Interval Set Translator

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

1 Flat file refers to a standard file viewed as a linear sequence of bytes and accessed via linear operations (i.e. read, write, seek).

migration of data from one interval layout to another. Efficient translation is accomplished by the use of an interval tree 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.

1.5 Summary of Research Results

Together, the interval cache, distributed lock manager, interval-based file layer, and the interval set translator provide a platform for researching interval-based parallel I/O. We have developed a prototype Interval I/O System, which has produced results indicating the promise of the approach. Discussions of these results have appeared in several papers, which are briefly summarized here. One study [42] demonstrates the effectiveness of interval files for performing checkpoint operations in the context of a large-scale physics simulation. A recent technical report [43] details the I/O performance improvement possible from using an interval-based approach when using pNetCDF or Parallel HDF5 in conjunction with MPI-IO. Another paper [44] illustrates the potential benefits of interval set translation for improving the performance of I/O for parallel data visualization. Finally, three papers [45,31,46] challenge widely held beliefs regarding which access patterns provide optimal performance, specifically when performing parallel I/O using the Lustre file system. Together these results offer substantial evidence of the promise of Interval I/O.

1.6 Contributions of this Research

- Examines a new paradigm for parallel I/O
- Provides a direct performance benefit to existing applications that use MPI-IO
- Allows I/O-bound applications to scale to larger problem sizes and processor counts
- Benefits the scientific modeling community by allowing models to run more quickly, or at higher resolution
- Benefits the wider community of MPI-IO users

1.7 Organization of this Document

The remainder of the document is organized as follows. The related research is discussed in Chapter 2. Chapter 3 contains a description of each of the five major components of the proposed system. Chapter 4 discusses the various advantages to using Interval I/O. In Chapter 5, we present the performance testing that has been done using the system. Chapter 6 examines the performance implications for performing parallel I/O with an object-based file system, Lustre, and Chapter 7 describes how we extended the interval file format to take advantage of Lustre's performance characteristics. Finally, Chapter 8 concludes the document and provides an overview of future directions for Interval I/O.

2 RELATED RESEARCH

A great deal of effort has been directed toward solving the scalable I/O problem. This section details the research efforts that are most closely related to this research.

2.1 MPI-IO

MPI-IO is the parallel I/O component of the Message Passing Interface 2 (MPI-2) specification [5]. MPI-IO is designed to support a variety of file access techniques including independent and collective I/O; contiguous, strided, vector, and structured accesses; and both synchronous and asynchronous operations. MPI-IO also features support for *file views*, which allow an application to declare the portions of a file which each of the application's processes will access. The ability to leverage the information contained in file views is a key element of this research.

The MPI-IO file view mechanism allows application processes to declare the file regions that they will access, and by extension, those that they will not. This is important since portions of the file that fall outside of a process's current file view cannot be accessed by that process. It is this information contained in the file views that form the basis of our approach.

2.2 ROMIO and ADIO

ROMIO is a widely used implementation of MPI-IO introduced by Thakur, *et al.* [20]. ROMIO provides a flexible, portable interface to a variety of file systems. Its flexibility is due largely to the Abstract Device Interface (ADIO) [21] upon which ROMIO implements the file-system-dependent functions of MPI-IO. This flexible architecture is illustrated in Figure 2.1. As can be seen, the application interacts with ROMIO via the MPI-IO interface. These calls are translated into corresponding ADIO calls, which in turn use a specific driver designed for the file system upon which the file is stored. ROMIO contains implementations of important parallel I/O optimizations such as data sieving and two-phase collective I/O, which are discussed below.

The Interval I/O system has been implemented as an extension to ROMIO. It is located in the ADIO layer, making it almost completely transparent to applications that use MPI-IO for parallel I/O.

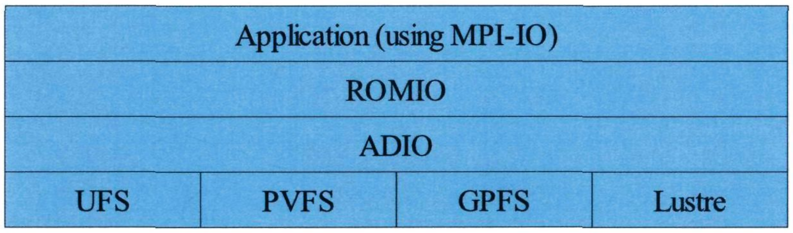

Figure 2.1. The ADIO architecture used by ROMIO

2.3 Two-Phase I/O

ROMIO implements the collective I/O operations using a technique termed two-phase I/O [25]. Consider a collective write operation. In the first phase, the processes exchange their individual I/O requests to determine the global request. The processes then use inter-process communication to re-distribute the data to a set of aggregator processes. The data is re-distributed such that each aggregator process has a large, contiguous chunk of data that can be written to the file system in a single operation. The parallelism comes from the aggregator processes performing their writes concurrently. This is successful because it is significantly more expensive to write to the file system than it is to perform inter-process communication.

A generalized version of the two-phase I/O optimization, termed multiple-phase collective I/O, was presented by Singh, et al. [47]. The technique also uses a single disk I/O phase, but allows the number of redistribution phases to vary. The authors note that the effectiveness of additional shuffling phases depends on the interconnection network technology of the target platform. Another variation of two-phase I/O known as disk-directed I/O [32] uses a set of "I/O processors" to coordinate the rearrangement of data instead of performing aggregation on the application processors.

Two-phase I/O performs well for hundreds of processors, but due to increasing communication costs, becomes untenable as the number of processors involved scales toward the thousands. Use of the Interval I/O system eliminates the need for two-phase I/O. No inter-process communication is needed to rearrange accessed data since data is not stored in linear order on the underlying file system.

2.4 Techniques Addressing Noncontiguous I/O

Much of the research related to parallel I/O aims to address the issue of noncontiguous I/O performance. In this section we discuss some of these techniques, including data sieving, view I/O, list I/O, and datatype I/O.

2.4.1 Data Sieving

The data sieving optimization [22] targets non-contiguous access patterns, which have been shown to be commonly used in scientific applications [20,13,40]. For many systems, the cost of performing several small accesses is much greater than the cost of performing a single large file access. Data sieving exploits this property by replacing a series of small accesses with a single access that spans the full extent of the non-contiguous regions.

For a data sieving read operation, it is sufficient to perform a single read encompassing the full extent of the access and then simply disregard the portions of the read that are not required. This results in a larger number of bytes to be read, but requires only a single file system access. The performance improvement achieved by data sieving depends on the number of non-contiguous file blocks being accessed, and the size of the gaps between required file blocks.

Data sieving for a write operation is similar, except that extra care must be taken to avoid overwriting portions of the file that are not part of the application's write. Typically a read-modify-write sequence is performed where (1) the full extent is read into a local buffer, (2) the data to be written is copied into the same buffer, and (3) the data from the buffer is written to the file from the modified buffer. This process requires that the gaps in the noncontiguous access pattern being written are not erroneously modified during the data sieving operation. Since the operation assumes responsibility for these additional areas of the file, it is also necessary to lock the full extent of the write to ensure that the MPI-IO consistency semantics are enforced. This additional locking leads to false contention when the extents of two or more non-conflicting writes overlap.

The Interval I/O technology eliminates the need for data sieving, since all intervals accessed by a process are stored contiguously on disk. This contrasts with traditional flat file storage, in which non-contiguous access patterns always require non-contiguous disk accesses.

2.4.2 View I/O

Isaila and Tichy [33] describe another approach to parallel I/O based on file views. Their technique, termed *View I/O* allows the multiple noncontiguous blocks of an I/O access to be combined into a single file system transfer. View I/O requires support from the file system, which rearranges the non-contiguous blocks into a linear ordering before storing them to disk.

Like Interval I/O, View I/O makes use of MPI-IO file views to optimize data transfers between the application and the file system. Both combine smaller, noncontiguous accesses into large chunks, and use associated metadata to allow the data to be later reorganized into linear file order.

View I/O, however, relies on support from the underlying file system to reorganize the file data. In contrast, our approach requires no file system modifications. Rather than reorganizing data at the disks, we store the data in the same layout as it appears in the cache, along with additional metadata that allows the data reorganization to be performed when the file is read. This approach is particularly well suited to application checkpointing, since checkpoint files only need to be read in the case of an application restart, which is relatively rare.

2.4.3 Other Techniques for Addressing Noncontiguous I/O

List I/O [20] is an I/O strategy that uses a specialized file system interface for supporting noncontiguous file system accesses. Like View I/O, List I/O allows noncontiguous data to be transferred as a single contiguous chunk, along with metadata describing its contents. A typical access would be specified with a list of (offset, size) pairs describing a block of the file being accessed. Unlike View I/O the List I/O metadata applies only to the current access, so subsequent accesses require resending the relevant metadata to the file system.

Datatype I/O [34] expands the List I/O interface to allow file patterns to be expressed more concisely than with a sequence of (offset, size) pairs. The patterns are instead represented using datatypes to take advantage of regular access patterns. For instance, the list representation {(0, 2), (4, 2), (8, 2), (12, 2), (16, 2)} could be represented as a strided type with chunk size = 2, chunk offset = 4, and count = 5. For patterns containing a large number of small chunks, the savings over the list representation of file data can be quite significant.

These approaches address the issue of reducing the large numbers of individual file system accesses that result from noncontiguous I/O patterns. Unfortunately, they fail to address a more fundamental problem, that of the rigid requirement that an application's file data be stored in linear order. Our work directly addresses the problem by removing this requirement, thus eliminating the problem that List I/O and Datatype I/O attempt to solve.

2.5 Collective Caching

There has been some research directed toward using collective caching to improve parallel I/O performance [28-30]. Collective caching uses memory available on the application processors to provide an additional layer of caching, leading to increased I/O performance. Additionally, efficient caching can result in fewer disk accesses, and consequently, in reduced power consumption [48]. The major challenge with collective caching is maintaining cache coherence without sacrificing performance.

One example of collective cache research is DAChe [28]. The DAChe system is a block-based, client-side cache designed to use remote memory access (RMA) to perform cache management across cluster nodes. To provide cache coherence, DAChe enforces the constraint that the cache may contain only a single valid copy of any particular file block. This is accomplished by requiring mutually exclusive access to file metadata, which is provided by a set of *mutex servers*. The mutex servers are drawn from available application processes, and are unable to participate in other computation while providing mutual exclusion.

The primary difference between DAChe and the system described here is that DAChe caches fixed size blocks while we cache intervals. DAChe must provide mutual exclusion for every file block access, while with our approach, we are able to distinguish between shared and private intervals, thus eliminating the need for much of the locking. Our approach also reduces false sharing, which is discussed in detail in the next chapter.

2.6 Active Buffering

Active Buffering [49] is another technique for improving parallel I/O performance. In the Active Buffering approach, data written by a parallel application is held in memory buffers on the application processors. Additional I/O processors are then used to pull data from these buffers asynchronously using one-sided communication, and write the data to disk. A variant of active buffering, described in [35], eliminates the additional I/O processors, and performs write behind of buffered data using a background thread on the compute processors.

Like our Interval I/O system, Active Buffering buffers written data in local memory to reduce the performance impact due to I/O latency. The Active Buffering approach, however, is designed to optimize the performance of write operations, however, it cannot be used for read operations.

2.7 ADIOS

ADIOS [36,50] is a framework that supports efficient parallel I/O by providing an additional level of indirection between application I/O calls and the underlying I/O technique. This allows a user to choose the I/O method that best suits the underlying hardware and the I/O pattern being used by the application. The framework allows an application to use different I/O patterns for each "grouping" of data used in a single application. Supported I/O techniques include MPI-IO (with synchronous and collective options), POSIX I/O, and DataTap. ADIOS also provides an intermediate file format, termed BT, that is used to facilitate fast file writes. The BT files are then converted to a linear file format asynchronously by the framework. To use the ADIOS framework, applications must be rewritten to use the ADIOS API.

The design of ADIOS is somewhat similar to the Interval-Based I/O stack, particularly in the use of a custom file format designed to allow the application to perform fast writes of large contiguous blocks used in conjunction with a helper application designed to reformat the hastily written data. There are several major differences between the approaches. First, the interval file format is not designed as an intermediate file format. In our system, the interval file is the standard storage format. Creation of a the logical flat file is performed only if necessary. Next, we provide explicit support for atomic mode accesses, which are not supported by ADIOS. Finally, the Interval-based I/O system directly supports MPI-IO, allowing existing applications that use MPI-IO to use the system with no modification to the source code.

2.8 Efficient Atomic Mode Accesses

One of the most significant challenges in providing high-performance parallel I/O is supporting strict file consistency semantics in a scalable way [16]. One of the main approaches to providing sequential consistency is through file system locking, although processor handshaking [37] and conflict detection [51] have been been suggested as alternatives to locking.

There are a number of approaches to locking ranging in complexity from whole-file locks to the locking provided by parallel file systems such as GPFS or Lustre. We discuss a relevant selection of locking approaches in the following sections.

2.8.1 File Locking

As the name suggests, this approach uses locks that have whole file granularity. For parallel writers, this implies that only a single process may have access to a particular file at one time, regardless of whether the writes are conflicting. Though this is sufficient to insure MPI consistency semantics, it is an extremely poor choice for parallel I/O since writes performed by multiple processes using file locking are completely serialized.

2.8.2 Extent Locking

Extent locking extends file locking by considering the full extent of the access that requires the lock. This information is used to allow locks to be granted simultaneously to processes whose access extents do not conflict. For contiguous accesses, byte range locking is sufficient to avoid false sharing. However, for the noncontiguous accesses that are typical for scientific applications, extent locking can lead to false sharing and unnecessary serialization of accesses, as discussed in Section 4.1.

2.8.3 List Locking

List locking [38] allows a process to lock a noncontiguous file region by specifying a list of contiguous byte ranges to be locked. Compared to extent locking, list locking reduces the amount of serialization by eliminating the false sharing that arises from noncontiguous access patterns.

Datatype locking extends list locking by allowing a lock list consisting of a regular pattern of byte ranges to be declared using a datatype constructor, rather than an explicit list of byte ranges. These datatypes are expressed in a manner similar to Datatype I/O, which is discussed in Section 2.4.3.

2.8.4 Distributed Locking Approaches

Supporting scalable atomic I/O for noncontiguous access patterns is a particularly challenging problem. The predominant approach is to use a distributed lock management system, where responsibility for granting locks is distributed across a number of lock managers. The primary difficulty is to maintain the efficiency of a centralized lock manager while gaining scalability by distributing the locks. Distributed lock management is often implemented in the file system. Here we describe the locking approach used by GPFS, as well as a distributed lock manager that has been built as an extension to PVFS.

GPFS [52] is a parallel file system that uses a distributed lock manager to support POSIX consistency semantics. GPFS uses a distributed locking stategy that requires a lock manager to first obtain a lock token from a central token server. The first manager to request a token receives one that covers the entire file. When another lock manager requests a token, the central token server must compare the regions being accessed. If there is no overlap, the original token is split, otherwise, the new request is queued until the first access is completed. This technique does not work particularly well for fine-grained, noncontiguous accesses, thus GPFS also provides a "data shipping" mode in which data blocks are assigned to server nodes, and read and write requests are forwarded directly to the nodes on which the data resides.

Ching, *et. al.* [53] describe a distributed lock management system that incorporates List Locking and Datatype locking into PVFS. Their system supports locking of arbitrary lists of byte regions; comparison of locks is done using interval trees. This system, like our Interval I/O system, does eliminate false sharing due to locking, however it does not incorporate conflict detection to reduce locking overhead.

2.8.5 Interval Locking

Our research introduces a novel locking technique termed interval-level locking. This system implements locking in the application layer, rather than the file system layer, and thus is portable to different systems regardless of the underlying file system. Our approach uses intervals as the fundamental unit of locking, which improves I/O performance by reducing false sharing. Interval-level locking uses a novel distributed lock management system which assigns each interval lock to one of the processes that accesses the interval. Each lock manager acts as a centralized lock manager for the subset of locks assigned to it. To reduce locking overhead, Interval-level locking uses the information in the interval set to determine whether access to a particular interval could possibly conflict with that of another process, and omit locking in cases where no conflict is possible.

2.9 Structured Data Formats

Scientific applications quite often make use of libraries that support structured data file formats. Network Common Data Format (NetCDF) [54] and Hierarchical Data Format (HDF5) [55] are the two most prevalent file formats, both of which have a long history of program support and which were originally designed to be accessed serially. Both formats now have parallel libraries available, called PnetCDF and Parallel HDF5, respectively. Each of these libraries uses MPI-IO as the underlying library to perform I/O operations.

The interval-based file format described here is similar to these structured data formats in that its use imposes additional structure on the data file. This additional structure is different in two important ways. First, the added structure imposed by NetCDF and HDF5 is more closely related to the user's concept of the data; the metadata is defined by calls made by the application to define the data being stored. In contrast, the metadata added to our interval-based files is related directly to our rearrangement of the data on the file system. Secondly, the structured data formats are typically known to the user, and explicitly supported in the application. Our interval-based file format, on the other hand, is designed to be completely transparent to the application. The use of these structured data formats is, in principle, compatible with interval I/O, provided that the data format library makes appropriate use of MPI file views.

2.10 Parallel I/O using Lustre

Lustre is a widely used distributed file system that is commonly available on large computing clusters. It has been observed that Lustre performs very poorly with MPI-IO. Thus part of our research has been directed at determining the cause of this poor performance and discovering solutions. Other researchers have also examined the particularly poor performance observed when using MPI-IO in a Lustre environment. The most closely related work is from Yu, *et al.* [56], who implemented the MPI-IO collective write operations using the Lustre file-join mechanism. In this approach, the I/O processes write separate, independent files in parallel, and then merge these files using the Lustre file-join mechanism. They showed that this approach significantly improved the performance of the collective write operation, but that the reading of a previously joined file resulted in low I/O performance. As noted by the authors, correcting this poor performance will require an optimization of the way a joined file's extent attributes are managed. The authors also provide an excellent performance study of MPI-IO on Lustre.

The approach we are pursuing does not require multiple independent writes to separate files, but does limit the number of Object Storage Targets (OST) with which a given process communicates. This maintains many of the advantages of writing to multiple independent files separately, but does not require the joining of such files. The performance analysis presented in this dissertation complements and extends the analysis performed by Yu, *et al.*

Larkin and Fahey [57] provide an excellent analysis of Lustre's performance on the Cray XT3/XT4, and, based on such analysis, provide some guidelines to maximize I/O performance on this platform. They observed, for example, that to achieve peak performance it is necessary to use large buffer sizes, to have at least as many IO processes as OSTs, and, that at very large scale (i.e., thousands of clients), only a subset of the processes should perform I/O. While our research on Lustre performance reaches some of the same conclusions on different architectural platforms, there are two primary distinctions. First, our research is focused on understanding of the poor performance of MPI-IO (or, more particularly, ROMIO) in a Lustre environment, and on implementing a new ADIO driver for object-based file systems such as Lustre. Second, our research is investigating both contiguous and non-contiguous access patterns while this related work focuses on contiguous access patterns only.

In [58], it was shown that aligning the data to be written with the basic striping pattern improves performance. They also showed that it was important to align on lock boundaries. This is consistent with our analysis, although we expand the scope of the analysis significantly to study the algorithms used by MPI-IO (ROMIO) and determine (at least some of) the reasons for sub-optimal performance.

3 THE INTERVAL I/O SYSTEM

In this chapter we discuss the design of our Interval I/O system. The system consists of five major components, which are the interval integration interface, the interval cache, the distributed lock manager, the interval-based file layer, and the interval set translator.

Interval I/O provides a variety of advantages over other approaches. We make note of specific advantages of this approach throughout the chapter but delay a detailed discussion of the advantages until the next chapter

3.1 Interval Integration Interface

The function of the Interval Integration Interface (I^3) is to perform the translation of MPI-IO calls into corresponding interval set accesses. Specifically, the I^3 utilizes file views set by the application to create interval sets, and then converts subsequent file read and write operations into equivalent interval accesses.

3.1.1 Interval Set Creation

Interval set creation takes place once all of the file views have been set, but before any file access operations have been performed. ROMIO represents a process's file view as a list of (offset, size) pairs, each of which describes a contiguous file region visible to that process.

The first step in creating intervals is for each process to share its file view with all other processes. Next, each process independently extracts a list of all boundaries. Next the boundary list is sorted, and duplicate boundaries are removed, leaving b distinct boundaries. Each interval int_i is defined by the (offset, size) pair (B_i, B_{i+1}-B_i), where B_i is the ith boundary in the sorted list. This results in a set containing b – 1 intervals. This process is shown in Figure 3.2, which shows the creation of intervals from the file view in Figure 3.1.

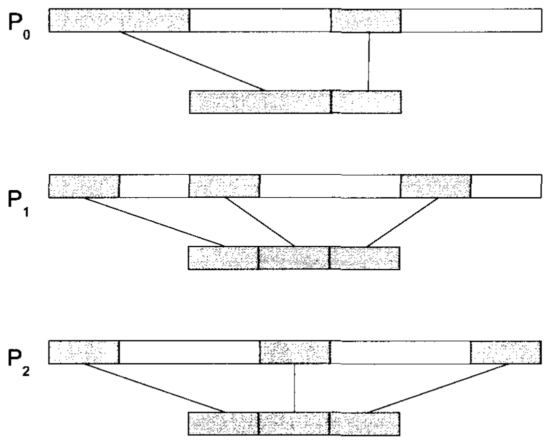

Figure 3.1. A file view example with shared and private regions

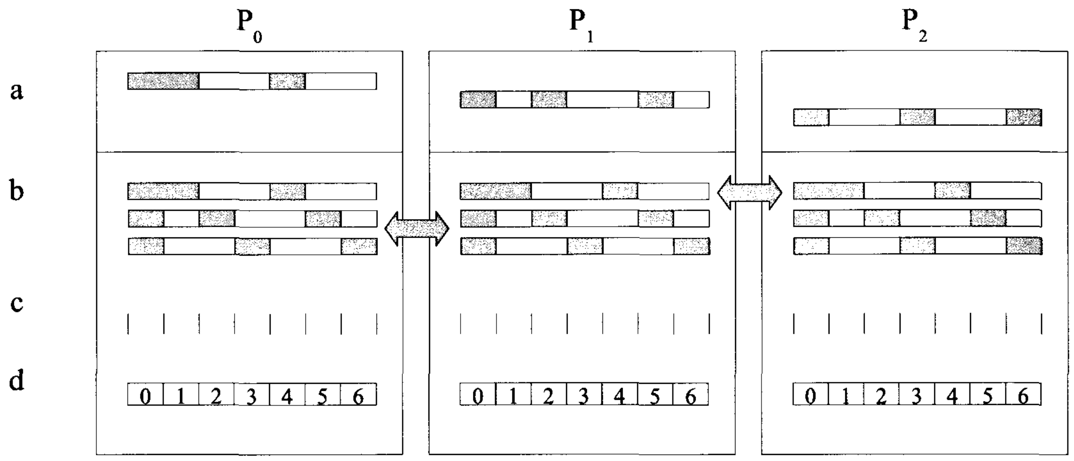

Figure 3.2. The interval creation process.

File view data from the views shown in Figure 3.1 (a) is shared among all processes (b). A list of view boundaries is extracted (c), and the boundaries are assembled into intervals (d). Note that intervals are sized according to the file view, and are not necessarily all the same size.

Once the intervals have been created, an additional step is taken to calculate the *reverse access set* for each interval, which is simply a list of every processor that has access to the given interval. The reverse access sets have two important uses. First, they show exactly which intervals are shared between processes and which are private to a process (known as *shared intervals* and *private intervals* respectively). Secondly, they help guide to which process a particular interval should be assigned. For example, private intervals are assigned to the one process that accesses it, and shared intervals are placed on one of the processes that shares the interval. Distinguishing between shared and private intervals is a significant advantage of the Interval I/O approach, an advantage which we will discuss in greater detail in Chapter 4.

3.1.2 Read and Write Translation

In addition to interval set creation, the Interval Integration Interface is responsible for converting file access parameters specified as byte ranges into equivalent parameters specified as a set of intervals. In section 3.2.1 we discuss in detail the handling of these read and write operations.

3.2 Interval-Based Cache

We have implemented a client side software cache that uses memory from all of the processes sharing a given file. The unit of caching is the interval. Global cache coherence is maintained by keeping only a single valid copy of each interval. Our interval based cache offers many of the benefits of traditional caching, including data prefetching and write behind, however, it simplifies some of the primary challenges associated with caching, which include false sharing and distributed locking, both of which are discussed in more detail in Chapter 4.

3.2.1 Cache Architecture

The cache, shown in Figure 3.3, is based on intervals and interval managers. The interval managers are required to maintain a buffer that is large enough to accommodate all of the intervals that the given process can access (we discuss relaxing this requirement in Chapter 7). Thus there will be a copy of a shared interval in the caches of all processes that share the interval. However, there is only one valid copy of a shared interval at any given time.

From an interval manager's point of view, it performs three different roles depending on the type of interval. In the case of a private interval, it simply reads/writes intervals from/to its local cache buffer, and requires no communication with other managers. In this case, there is only one copy of the interval in the global cache. In the case of a shared interval, the manager plays one of two roles. First, it can play the role of what we term the *location manager* that tracks the location of the currently valid copy of the interval. There can be only one location manager for each shared interval. Otherwise, the manager of a shared interval (that is not the location manager) must contact the location manager to determine the location of the currently valid copy of that interval.

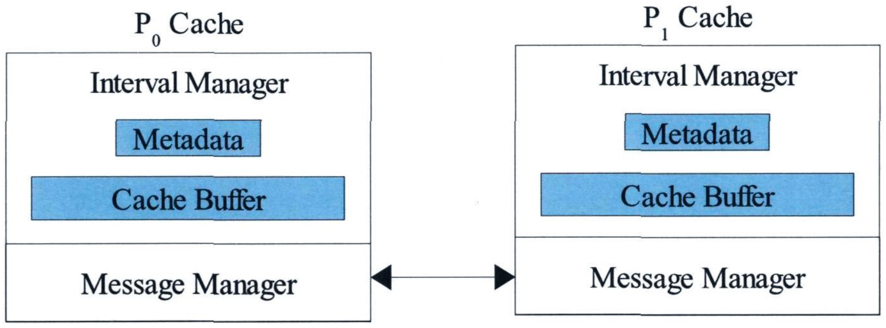

Figure 3.3. The basic architecture of the cache

Creation of the file cache is performed immediately following the interval set creation discussed in section 3.1.1. During cache creation, each interval is assigned to one of the participating I/O processes, which is responsible for managing the location of the valid copy of that interval. Once the intervals are assigned to processes, memory is allocated to store the cache data. The allocated memory consists of a single buffer on each process with enough space to store all of the intervals in that process's file view contiguously.

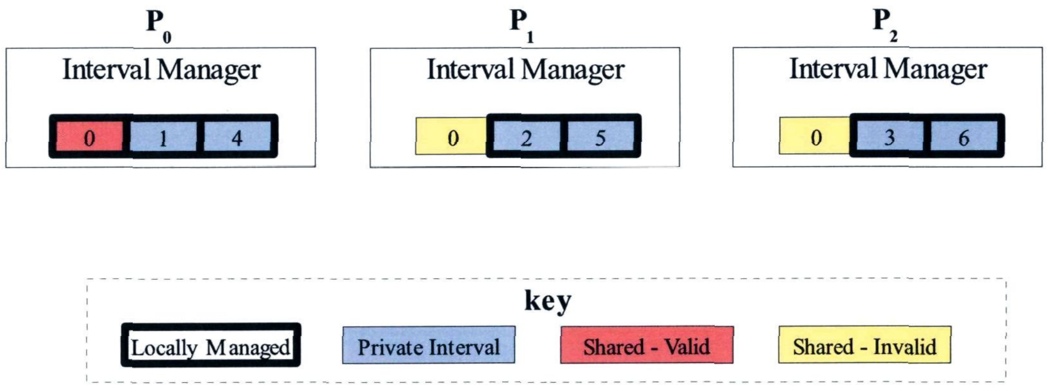

Figure 3.4. The initial cache layout resulting from the interval set created in Figure 3.2.

Figure 3.4 illustrates the interval assignments and the cache buffer layout resulting from the interval set created in Figure 3.2. In the example, Intervals 1 – 6 are private, thus each is assigned to the process that accesses it. Interval 0 is shared by all three processes, so it is assigned to one of the processes that accesses it (in this case, process 0). Since interval 0 is shared, it appears in the cache buffer of all three processes, although only one of the buffers will contain the valid copy of the interval at any particular time. In this example, interval manager 0 is assigned to be the location manager for interval 0.

For a read operation, the first step is to determine the intervals that are part of the read. Depending upon whether MPI's atomic mode is set, a set of locks must be acquired corresponding to the set of intervals being read. We discuss the details of lock acquisition in more detail in section 3.3.1. Next, for each of the intervals, the process that manages that interval is queried to determine the location of the most current data for that interval. The location of the valid copy determines how the reads are performed for each interval. If the current data for an interval is local, its data is copied directly from the local cache into the specified read buffer. Otherwise, the interval data is read remotely by sending a request to the appropriate process and blocking until the requested data is received. Once the data is copied into the read buffer, all locks are released, and control is returned to the application.

The handling of a write operation is somewhat different from that of a read. We begin in the same manner, by first determining the set of intervals involved in the write, and then acquiring a lock for each of the shared intervals. Assuming a write to an interval involves the entire interval, as will often be the case, the written data for that interval is copied directly into the local cache buffer and the relevant location manager is updated with the new location of the valid copy of that interval. If, however, the write involves only a portion of any particular interval, it is necessary to first insure that the valid data for that interval is in the local cache before performing the copy and update operations. Once all of the intervals have been written, the locks are released, and control returns to the application.

It is worth noting that it is necessary to perform the individual accesses to intervals in the cache atomically. Our distributed lock management system, discussed in Section 3.3, is used to provide the required atomicity.

To illustrate the process of reading and writing intervals in the cache, we refer the reader to Figure 3.5, which shows reads and writes involving a file distributed across two processes. In part a of the example, P_0 reads data from interval 5. Since the location manager for the interval is P_1, a request is sent to P_1 for the location of the data. Since P1 currently holds the valid copy of the interval, it sends a response containing the data for interval 5. Though a copy of the data is sent to P_0, the valid copy is not changed in response to a read operation, thus the valid copy remains on P_1.

Figure 3.5b illustrates a write to interval 1 performed by P_1. The data is simply copied into the local cache buffer, and a message is sent to the manager for that interval, P_0, indicating the new location of the valid data for interval 1.

When closing or syncing the file, the contents of the cache are written to disk. The interval file format, discussed in Section 3.4, enables the data to be written contiguously in the order it is stored in the cache.

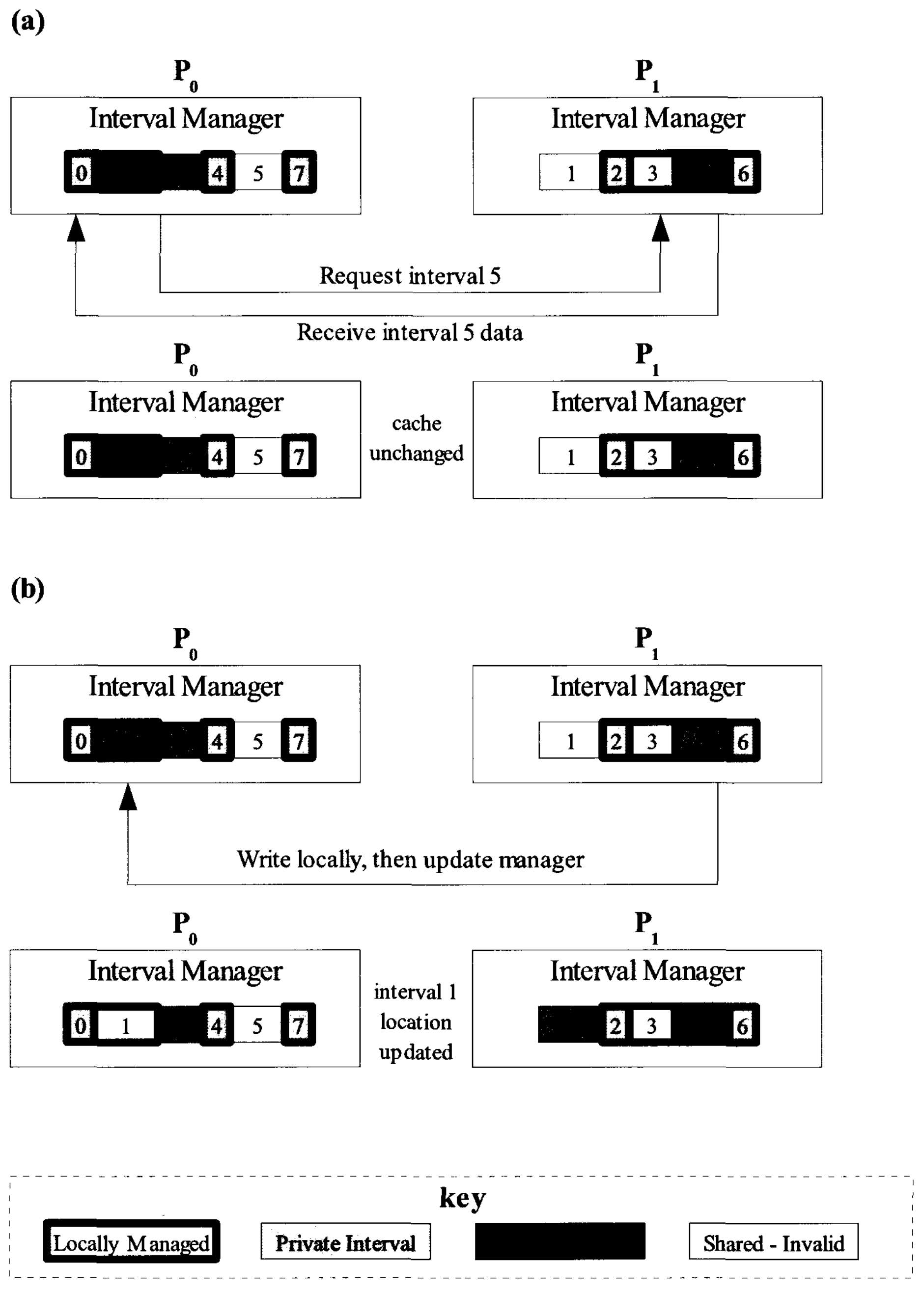

Figure 3.5. Reading and writing shared cache objects.
(a) P_0 reads interval 5
(b) P_1 writes interval 1

3.3 Distributed Lock Manager

A very important part of the Interval I/O System is the distributed lock manager. In keeping with the interval I/O philosophy, the lock manager uses the interval as the fundamental unit for locking. The locks are distributed across a collection of lock managers, with each manager acting as a centralized lock manager for the intervals assigned to it. This arrangement provides scalability, since global knowledge is not required for lock managers or lock clients. Furthermore, as discussed in the previous section, locking is required only for shared intervals, reducing (and in some cases eliminating) the cost of locking. The efficiency and scalability of this distributed locking system are major advantages to using an interval-based approach to parallel I/O. We discuss these advantages in more detail in Chapter 4.

3.3.1 Lock Manager Design

The lock manager works with the cache manager to enforce atomic mode. The mechanism is transparent to the application. The lock manager is implemented as a multithreaded library that is instantiated by the cache. TCP sockets are used to provide reliable communication between distributed lock managers running on application processes.

Each cache process has direct access to a local lock manager operating on the same processor. The cache interacts with the lock manager through a local interface that supports lock creation, lock and unlock requests, and lock destruction. Lock requests are designed to cause the calling thread to block while the communication required to acquire the requested locks is performed. That communication involves a request that is sent to each of the lock managers which possess any of the required locks. The request is sent from lock manager to lock manager in a predetermined order (shown in Figure 3.6) that is the same for all requests. A consistent order is required to prevent deadlock, which is discussed in more detail in section 3.3.2. Once all locks are obtained, the request is sent back to the lock manager on the process that initiated the request, and control is returned to the application thread.

First, we define P(I) to be the index of the process on which the lock for interval I resides.

Then, for any two intervals I_j and I_k,
I_j precedes I_k if and only if $P(I_j) < P(I_k)$ or
$P(I_j) = P(I_k)$ and $j < k$

Figure 3.6. The ordering used to grant locks

To illustrate the coordination between the distributed lock managers, a simple example of acquiring locks is shown in Figure 3.7. In the example, process one requires a set of locks for intervals 2, 3, 4, 5 and 10. The locking process proceeds as follows:

1. The process generates a request, which is sent to the local lock manager, while the process blocks waiting for a response from the lock manager.
2. The request is forwarded to the first lock manager that holds any requested lock, in this case, lock manager 0, where locks for intervals 2 and 5 are acquired in order. Although the lock for interval 3 is located on lock

manager 1, it is crucial that it is not acquired before locks 2 and 5, which preceed lock 3 in the ordering defined above.

3. The request is then forwarded to lock manager 1, where a lock for interval 3 is granted.
4. Next, the request is forwarded to lock manager 2, where the remaining locks for intervals 4 and 10 are acquired.
5. Once all requested locks are obtained, the request is returned to the lock manager on which it originated, lock manager 1
6. Lock manager 1 grants the request by responding to the original thread, allowing I/O to continue.

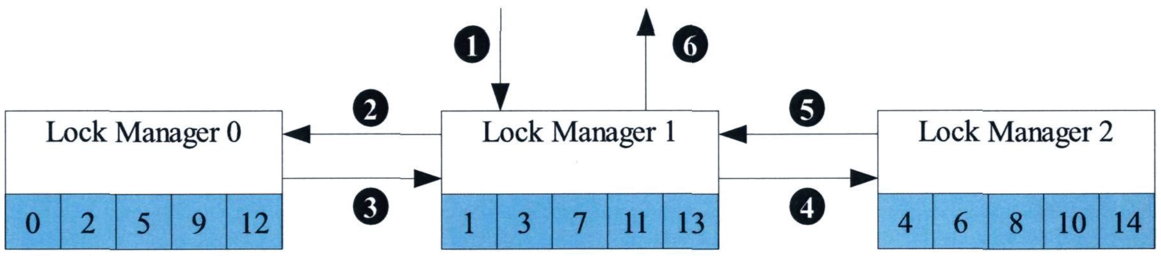

Figure 3.7. Fulfilling a request for locks 2, 3, 4, 5 and 10.

The order used to acquire locks is arbitrary, so we have chosen one that minimizes the number of network communications required to obtain a set of locks. For instance, acquiring the locks using the naïve ordering imposed by the object ids, the example shown in Figure 3.7 would require six network communications instead of the four shown in the example.

The lock manager is implemented by four additional threads running in each of the participating processes. The threads are represented as ovals in Figure 3.8. The *Send* thread and the *Receive* thread exclusively handle the sending and receiving of messages between the participating processors. The *Lock* thread is responsible for granting local locks to lock requests. The *Unlock* thread frees local locks when unlock messages are received, and notifies the next waiting lock request when an interval lock becomes available. These threads are separate from the main application thread that will request locks.

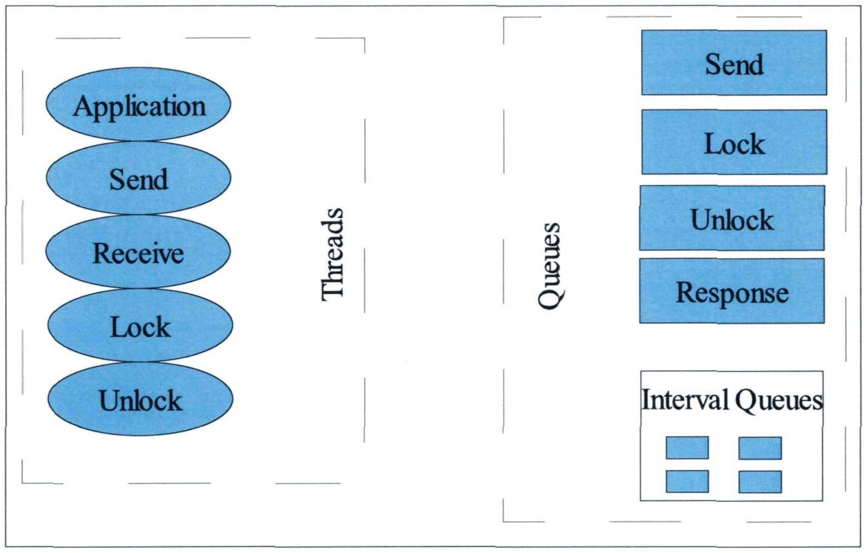

Figure 3.8. Local lock manager architecture

The manager also utilizes several queues that facilitate asynchronous communication between the threads. Each queue is shown as a rectangle in Figure 3.8. The *Send* queue holds messages waiting to be sent to one of the other processors. The

Lock queue holds incoming lock requests and lock requests restarted as a result of the release of other locks. Lock requests waiting for a specific lock are held on the *Interval* queue corresponding to the lock in question. The *Unlock* queue holds unlock messages waiting to be processed by the unlock thread. Completed lock requests are placed on the *Response* queue of the requesting process, where the application thread is blocked waiting for the lock to be granted.

3.3.2 Correctness of the Locking Protocol

The most important property of our distributed lock management system is its ability to provide atomic file access operations that conform to the consistency semantics required by MPI-IO. We have selected a variant of two-phase locking [59] that can be shown to provide the necessary semantics. Two phase locking is a locking protocol that allows locks to be acquired only during the initial "growing phase" of a transaction. The first phase is followed by a second "shrinking phase" in which locks are released, but no further locks can be acquired. In a two phase locking transaction, once a single lock is released, no further locks can be acquired.

Our locking protocol extends the basic two-phase locking protocol in two ways. First, we define a partial order on file locks, and require that locks needed to fulfill a particular lock request be acquired in order. This ensures that the acquisition of locks is deadlock free. Secondly, we constrain the transaction so that file accesses are performed only after all locks are acquired, and before any locks are released. This insures that no partial accesses are performed, and eliminates the need to provide a rollback mechanism for the handling lock revocation that would otherwise be required.

3.4 Interval-Based File Layer

The Interval I/O system includes an interval-based file layer. The key insight underlying the design of this layer is that a flat file can be represented by a set of intervals that together comprise the data contained in the flat file. We call this set of intervals, along with the metadata required to describe the objects, an *interval file.* By using interval files to represent the contents of a flat file, we gain a level of flexibility that provides significant opportunity for I/O optimization.

3.4.1 Interval-Based Files

Our Interval I/O system relies on an interval-based file format that stores file intervals in a structured manner. The layout of the interval file is chosen to coincide with the layout of cached data across the processors of a parallel application. This results in significantly increased performance since it allows each application process to access contiguous data in the interval-based file regardless of whether the accessed data would be contiguous in the corresponding flat file.

The current version of this file format is detailed in Table 1. The interval file is comprised of several sections, which include a global metadata section, one process metadata section for each participating process, and finally, sections containing the interval data.

The initial section contains the file metadata, which describes to the reader how the remainder of the file is arranged. This section is expected to be accessed by each process in the reading application, thus is kept as short as possible to limit contention.

Next, the interval file contains the process metadata sections. Each process metadata section consists of a count of intervals associated with that process, immediately followed by the metadata for each of those intervals, stored sequentially. This arrangement allows each process to access its metadata section as a contiguous block of data, avoiding any false contention, and allowing accesses to be as efficient as possible. Each metadata section contains the information required to locate the interval data in this file, and to describe where the interval is located in the flat file represented by this interval file, so that the original layout can be reconstructed when needed.

Finally, interval data is stored at the end of the interval file. In the current implementation, the interval data is stored contiguously, in the same order that the metadata sections are stored. Future work will include introducing flexibility in the positioning of interval data to optimize the pattern of underlying writes for a particular file system (i.e. Lustre).

In our implementation, files are written by a simple interval-file layer that lies between the interval-based cache and the ADIO layer. The cache layer writes data to the ADIO driver, which is implemented separately for each file system. Thus the caching system can be utilized on any file system currently available in ROMIO.

File Metadata (One section per file)		
Name	**Size**	**Description**
Magic Number	4 bytes	`0x0b9ec7ed`
Version Number	4 bytes	0x0002 (This is version 2)
Number of Procs	8 bytes	Number of Process Metadata sections in the file
Data Offset	8 bytes	Offset (in bytes) to the beginning of the data section
Meta Offsets	num_procs * 8 bytes	Offset (in bytes) to the start of each proc section. There is one 8 byte offset per proc section, stored sequentially
Process Metadata (One section per process)		
Name	**Size**	**Description**
Number of Intervals	8 bytes	Number of intervals in this Process Metadata section
Interval Metadata (One section per interval, stored contiguously after the Process Metadata for each process)		
Interval File Offset	8 bytes	Location of interval's data relative to the data offset
File Offset	8 bytes	Location of the interval in the corresponding flat file
Size	8 bytes	Size of the interval in bytes
Data (All data for a process is stored contiguously in this version)		

Table 1. Specification of Interval-Based Files

The significant features of the interval-based file layer are listed below.

•**Interval Based:** By allowing a file to be stored as a collection of intervals, our system will eliminate much of the reorganization of file data that is used by other parallel I/O optimization techniques such as two-phase I/O and view I/O.

•**Capable of efficient interval set translation:** A parallel application accessing a data file is likely to be using a different view of the file than the application that created the file. It is essential that there is an efficient mechanism for translating between the stored interval set and an interval set required by an arbitrary access pattern used by another application. Our translation technique is discussed in more detail in the next section.

•**Compatibile with existing parallel file systems:** Our system extends an underlying file system by adding an additional layer over the file system in use. It should work seamlessly with the majority of existing file systems in use on clusters on which parallel applications are executed.

•**Decouple application access pattern from file system access pattern:** Since the arrangement of the Interval file is not linked to the layout of the corresponding flat file, we are free to define a data layout that conforms to the best performance of the file system in use. In Chapter 6 we discuss how the performance of the Lustre file system is linked to the access pattern being used, and how our Interval I/O layer leverages this information to further improve I/O performance.

3.5 Interval Translation

It may be the case that a file is accessed by two or more applications, each of which may use a different access pattern to read or write the file. To address this issue, we have developed an interval translation tool capable of efficiently converting file data from one interval set to another.

3.5.1 Use of the Translator

The following examples should give the reader a better idea of how the translator may be used. One example is a tile reader/writer, in which a pair of cooperating applications collaborate to produce a data visualization. In this case, there are two applications, one is a simulation which produces data to be displayed, and another which 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, thus producing an interval file that is not compatible with the reader. Given a sufficiently fast translator, we can obtain the benefits of interval-based file accesses while allowing data to be shared between applications that use different access patterns. The ability to support this sort of inter-application cooperation is a major advantage of the Interval I/O approach, an advantage which we discuss in more detail in Chapter 4.

Another use for the translator relates to writing large checkpoint files. Resource intensive applications are often designed to periodically produce checkpoint files [60], which allow the applications to be restarted from that point in case of a failure. As detailed in Section 5.1, our research has shown that we can increase the performance of writing large checkpoint files by a factor of 14. The problem however, is that the application may need to read a checkpoint file with a different number of files or a different file view, or the file may need to be read by an entirely different application. In this case the translator is used to translate the original interval set into the one that is required.

3.5.2 Translator Design

We have implemented the interval translation tool as a parallel application, with application processes partitioned into two sets. One set is responsible for reading an input interval file, and a second set is responsible for writing a new interval file with the necessary layout. A diagram of the translation architecture is shown in Figure 3.9. In this example, the source processes (labeled PS_i in the diagram) read the metadata for the source interval set from the source interval file.

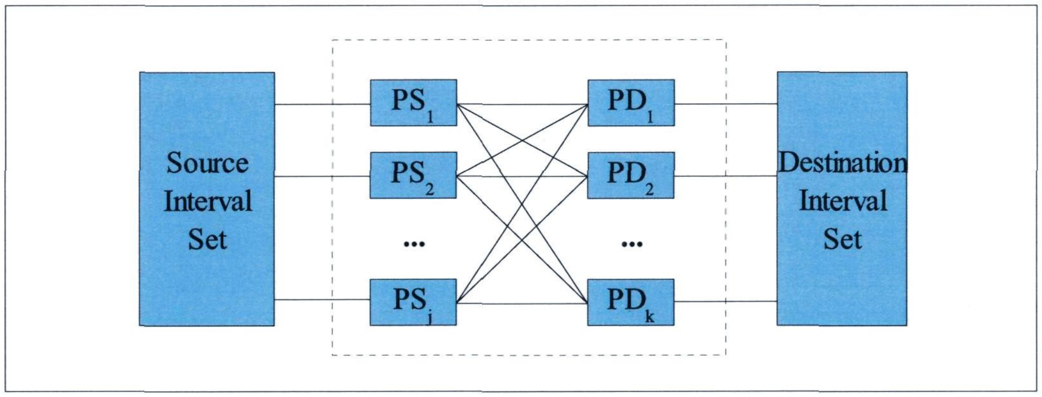

Figure 3.9. The Translator Architecture

The key to achieving an efficient translation algorithm is to provide a fast mapping between the source interval set and the destination interval set. We have implemented this functionality using a red-black interval tree, which combines the fast, O(log n) lookup provided by an interval tree [61] with the self balancing features of a left-leaning red-black tree [62].

4 ADVANTAGES OF USING INTERVALS

As was mentioned earlier, there are five major advantages to using Interval I/O. These advantages include:

- Increasing I/O parallelism by eliminating false sharing
- Significantly increasing I/O performance for noncontiguous access patterns by using interval files
- Introducing a mechanism to support I/O cooperation between applications
- Enabling an efficient technique for distributed lock management
- Reducing lock overhead by detecting private intervals

In this chapter, we discuss each of these advantages in turn.

4.1 Eliminating False Sharing

False sharing occurs when non-conflicting operations are serialized due to granularity of locks and lack of knowledge of I/O access patterns. For instance, a parallel I/O implementation that uses extent locking for non-contiguous operations may exhibit false sharing. Recall that extent locking requires the full extent of a set of noncontiguous regions to be locked, including the portions of the extent not included in the I/O operation. For example, consider the access pattern shown in Figure 4.1. Although P_0 and P_1 are performing nonconflicting accesses (as shown by the shaded regions), the extents of those accesses (indicated by the arrows) do overlap, thus the accesses would be unnecessarily serialized. In contrast, our approach would use three separate locks, two to lock the two noncontiguous regions accessed by P_0, and a third to lock the single region accessed by P_1. Since none of these regions conflict, the locks could all be acquired simultaneously, allowing the accesses to proceed in parallel.

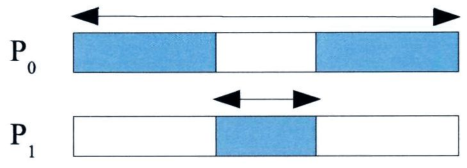

Figure 4.1. False sharing resulting from extent locking

False sharing also occurs in systems using block based caching. For example, Figure 4.2(a) shows an access performed in a block-based caching environment. Again, P_0 and P_1 are performing nonconflicting accesses. The access pattern spans three separate cache blocks (delineated by the dashed lines, and labeled $B_0 - B_2$). Since both of the processes access the middle cache block, access to that region is serialized. As shown in Figure 4.2(b), our approach creates two private intervals that are aligned precisely with the accesses, thus avoiding serialization.

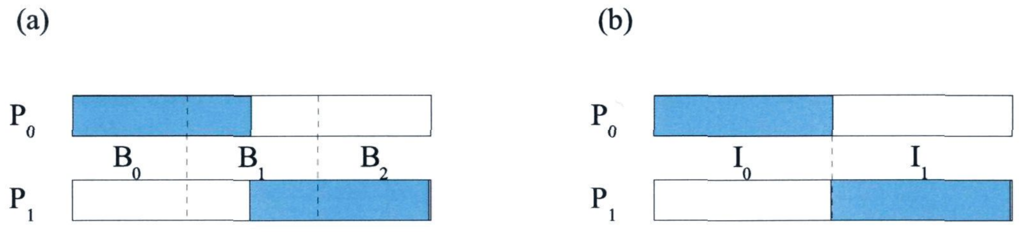

Figure 4.2. False sharing resulting from block-based caching
(a) Block-based caching
(b) Interval caching

4.2 Increasing Performance for Noncontiguous Accesses with Interval Files

The I/O throughput provided by parallel file systems depends greatly on the particular access pattern in use; unfortunately noncontiguous I/O patterns, which are among the most commonly used access patterns in scientific applications, are very difficult to implement with high performance in traditional (legacy) file systems. For example, consider the access pattern shown in Figure 4.3(a), which is stored on disk as shown in (b). ROMIO currently has three options available to deal with noncontiguous I/O, which are:

1. *Separate into contiguous accesses.* This produces poor performance because it incurs the overhead associated with a number of individual system calls to perform the I/O.
2. *Use data sieving.* This approach avoids the overhead of additional system calls to perform the I/O. However, when data sieving is performed it is necessary for the implementation to use extent locking to maintain the

consistency semantics required by the MPI-IO specification. This required use of extent locking typically introduces false contention, as discussed in section 4.1.

3. *Use two-phase I/O.* Two-phase I/O avoids noncontiguous accesses by rearranging data so that file system access can be performed contiguously. However, as discussed in section 2.3, the limited scalability of two-phase I/O limits its use with larger numbers of processes. Furthermore, two-phase I/O cannot be applied to independent I/O operations.

Our approach avoids the need to perform noncontiguous I/O by substituting an equivalent file, which is rearranged so it can be accessed using contiguous accesses, as shown in Figure 4.3(c).

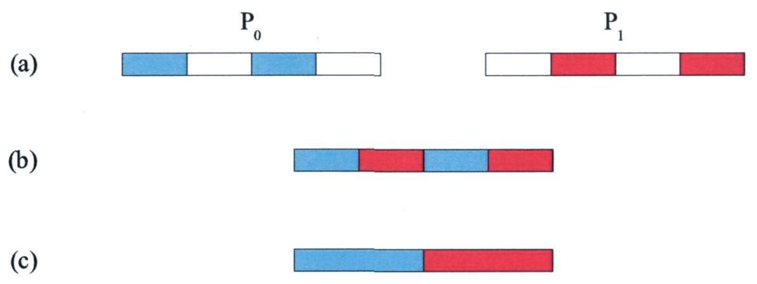

Figure 4.3. Using an interval file to store a noncontiguous I/O pattern. (a) The original access pattern, (b) Stored as a flat file, (c) Stored as an interval file

4.3 Supporting I/O Cooperation

It is often the case that a file is written by one application, the *producer*, and subsequently read by a different application, the *consumer*. One example of this is the remote visualization of data produced by a scientific modeling application. The MPI-Tile-IO benchmark was developed to model this type of application. The benchmark acts as a data consumer by reading overlapping tiles from a file containing two dimensional image data. It can also be configured to run as a data producer by writing image data to the file. Both the producer and the consumer utilize noncontiguous I/O patterns.

With interval I/O, the producer obtains increased performance by writing a file that is tuned to the access pattern of the producer. The problem then, is when the access pattern used by the consumer does not match that of the producer. In this case, reading the mismatched interval file would generally produce poor performance, canceling some or all of the benefit achieved by the producer. To support this inter-application cooperation, interval I/O provides a translator component, which converts the interval set written by the producer into one that can be efficiently read by the consumer. This use of the interval translator is shown in Figure 4.4.

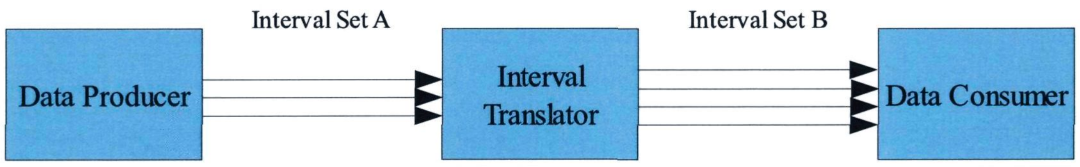

Figure 4.4. I/O cooperation between a data producer and a data consumer

4.4 Efficient Lock Management

The use of interval I/O allows a relatively simple distributed locking mechanism to be used to support atomic operations. There are several advantages to our interval-based locking strategy, which include:

- **Interval Granularity.** Using intervals as the unit of locking provides a simple mechanism for locking non-contiguous regions. Since intervals are based on the application's file views, the intervals (and thus the interval locks) correspond closely to the access pattern of the application.
- **Reverse access set detection.** Our lock management system benefits from the presence of reverse access set information for each interval in two ways. First, when the reverse access set for an interval contains only a single process, that interval is said to be private, thus no locking is required to access that interval. Secondly, the reverse access set information is used to guide the placement of shared locks so as to improve locking performance, for instance, by placing an interval lock on one of the processes that accesses that interval.
- **Distributed locks require no centralized control.** Lack of centralized control is crucial to the scalability of a locking system. Since our intervals are defined to be non-overlapping, the single process to which a particular interval is assigned will always act as a central manager *for that interval.* So the handling of a lock request is always restricted to only those processes managing the requested locks.

4.5 Detection of Private Intervals

A major advantage of interval-based I/O is the ability to identify private intervals, that is, file regions that are visible to only a single process. Since accesses to private intervals cannot be conflicting, it is not necessary to lock private intervals. This can have a significant impact on the level of concurrency depending upon the I/O access patterns of the application.

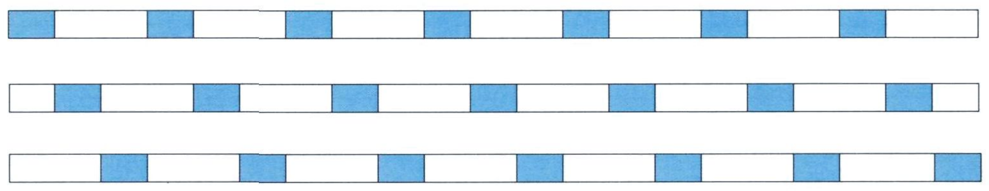

Figure 4.5. An access pattern containing only private intervals

Figure 4.5 shows a complex I/O pattern where all accesses are private. The resulting intervals would all be labeled as private, thus no locking would be required.

5 PERFORMANCE STUDIES

In this chapter we present representative results from the performance testing that has been performed. This testing includes three frequently used I/O benchmarks, FLASH-IO, MPI-Tile-IO, and 2D-Column-IO

The experiments described in this document were performed using four computing clusters that belong to the NSF sponsored Tera Grid Project [63].

1. *The Mercury cluster at the National Center for Supercomputing Applications*

   Mercury consisted of 1,774 Itanium 2 processors connected with Myrinet and running SuSE Linux SLES 8. The file system used was the General Parallel File System (GPFS) developed by IBM. This file system was 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 not used for these experiments. Since the time of our experiments, the Mercury cluster has been decommissioned.

2. *The Lonestar cluster at the Texas Advanced Computing Center*

   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 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.

3. *The Ranger cluster at the Texas Advanced Computing Center*

   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.

4. *The Big Red cluster at Indiana University*

   The Big Red cluster consisted of 768 IBM JS21 Blades, each with two dual-core PowerPC 970 MP processors and 8 GB of memory. The compute nodes were connected to Lustre through 24 Myricom 10-Gigabit Ethernet cards. The Lustre file system (Data Capacitor) was mounted on Big Red, and consisted of 52 Dell servers running Red Hat Enterprise Linux, 12 DataDirect Networks S29550, and 30 DataDirect Networks 48 bay SATA disk chassis, for a capacity of 535 terabytes. There were a total of 96 OSTs on the Data Capacitor, and there was a total aggregate transfer rate was 14.5 Gigabits per second. The MPI implementation used on BigRed was MPICH2.

5.1 FLASH-IO

The FLASH [10] simulation computes the solutions of fully-compressible, reactive hydrodynamic equations. It was developed to study nuclear flashes on the surfaces of neutron stars and white dwarfs. FLASH-IO [64] is a benchmark which is based on the I/O kernel of the FLASH simulation. The benchmark uses identical I/O code to that used in the simulation, thus any performance increase observed in FLASH-IO is expected to translate directly to the FLASH simulation.

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 5.1. 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 intervals created when the file is opened are shown in Figure 5.1 as 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 processor involved in the run.

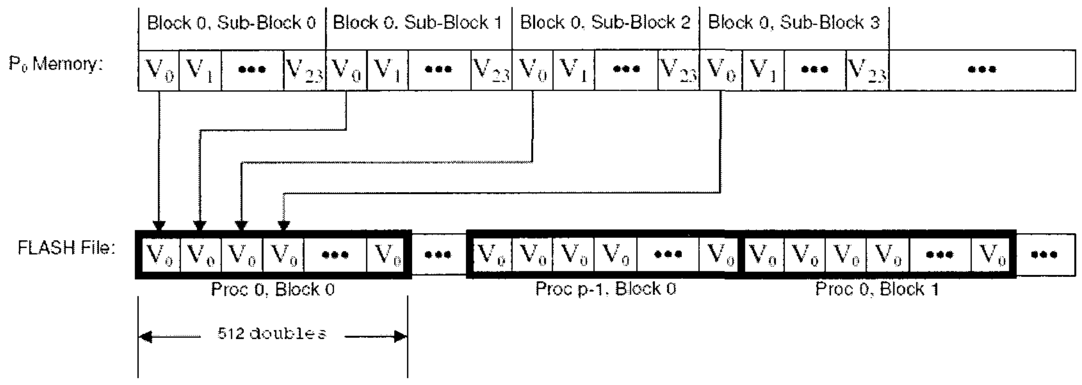

Figure 5.1. The FLASH-IO data layout

We designed our experiments to study the effectiveness of using our Interval I/O system to perform the checkpoint operations done by FLASH. We focused on the FLASH-IO checkpoint operation, in which the current simulation state is written to a file to allow restart in the event of system failure. The checkpoint operation is challenging for current MPI-IO implementations because the ordering of data is different in the file and in memory, thus requiring data reorganization to be performed. To accomplish this reorganization, traditional approaches perform a series of noncontiguous writes or use a two-phase I/O approach.

Using the Mercury cluster, we examined the effects of using Interval I/O on the performance of the FLASH-IO checkpoint operation. These experiments were performed with an early prototype of the Interval I/O System. This work was initially presented in [42], and the results are reproduced here in Figure 5.2. We varied the processor count between 4 and 64 processors and compared the performance of writing FLASH checkpoints using our interval cache with the performance obtained using ROMIO. We observed as much as a 50% reduction in write time (on 32 processors). In the 64 processor case, we demonstrated roughly a 38% reduction in the execution time of the FLASH-IO benchmark as compared to ROMIO. These results were an early indication of the potential of this approach.

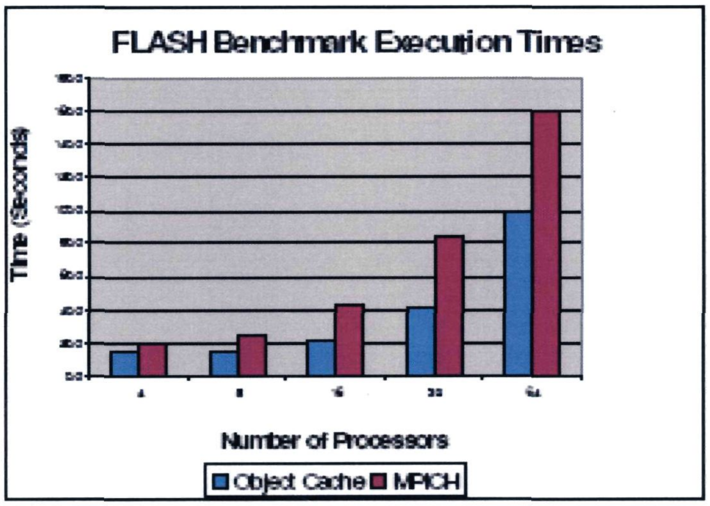

Figure 5.2. Initial FLASH-IO benchmark execution times

Since the time of the initial FLASH-IO study, we have implemented a second version of the interval cache, introducing a more modular design, and improving the overall stability of the system. We have extended the functionality of the cache by integrating it with the distributed lock manager. In addition, we have significantly improved the performance of the cache by using interval trees to determine reverse access sets, significantly reducing the cost of that operation and vastly improving the scalability of interval I/O.

Our latest experiments involving FLASH-IO [43] were performed on the Lonestar cluster. We compared write times for FLASH I/O checkpoint files using three different file formats: Parallel HDF5 and PnetCDF, both of which use ROMIO as the underlying I/O mechanism, and Interval Files using the Interval I/O system. The results of that comparison are shown in Figure 5.3, and reflect up to a 93% reduction in execution time on a 256 processor run.

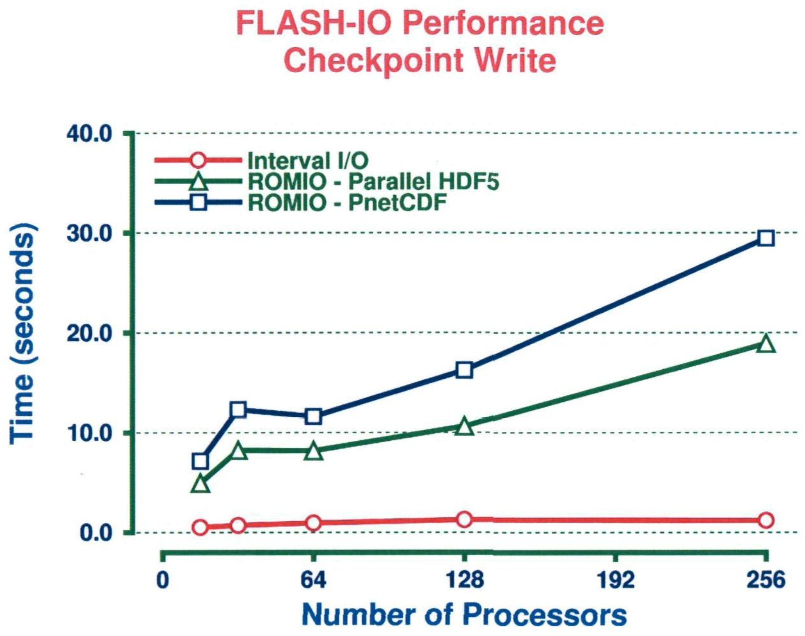

Figure 5.3. More recent FLASH-IO checkpointing performance results

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.

5.2 MPI-Tile-IO

The MPI-Tile-IO benchmark performs I/O on a two-dimensional array that is distributed across a number of processes. It supports both reading and writing of the array data, and accomodates an optional overlap between adjacent tiles, thus simulating a common I/O pattern used by scientific simulation applications. The MPI-Tile-IO access pattern is shown in Figure 5.4. In this example, the two dimensional array stored in the file is partitioned among four processes. The dashed lines illustrates the optional overlap area for P_0 and P_1. The array elements comprising the overlapping area are sometimes referred to as *guard cells*.

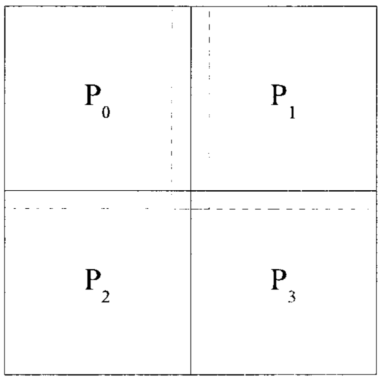

Figure 5.4. MPI-Tile-IO data layout

The MPI-Tile-IO benchmark models quite well the cooperating application scenario described in section 4.3 . The experiment involves a producer and a consumer, each simulated using MPI-Tile-IO. Each producer process writes data from a local two dimensional tile into a shared file containing the global array. Data from that file is then read by the processes that comprise the consumer application.

The producer and the consumer are configured to use slightly different access patterns. Each producer process writes a distinct portion of the array, with no overlap between producer processes. The consumer processes, on the other hand, each read the same area as the corresponding producer process, as well as an additional region of guard cells. These access patterns are challenging for current parallel I/O implementations because 1) they consist of a large number of noncontiguous regions, 2) the extents written by each of the processes overlap with a potentially large number of other processes, leading to serialization of accesses due to false sharing, and 3) complex locking required for access to guard cells.

Since the access patterns used by the producer and the consumer are not identical, the use of Interval I/O requires an additional translation step to convert the producer's interval set into one that can be efficiently read by the consumer. We examined the performance improvement shown by the MPI-Tile-IO benchmark in reading an interval-based file versus the equivalent flat file. The results of that study, which was performed on the Ranger cluster, are shown in Figure 5.4. As can be seen, read time was decreased by as much as 35% (in the 8 x 8 configuration) when compared to ROMIO. More significantly, however, we observe as much as an approximately 90% decrease in write time (in the 7 x 7 configuration[2]). We anticipate that the performance improvements shown here will more than offset the costs of interval translation.

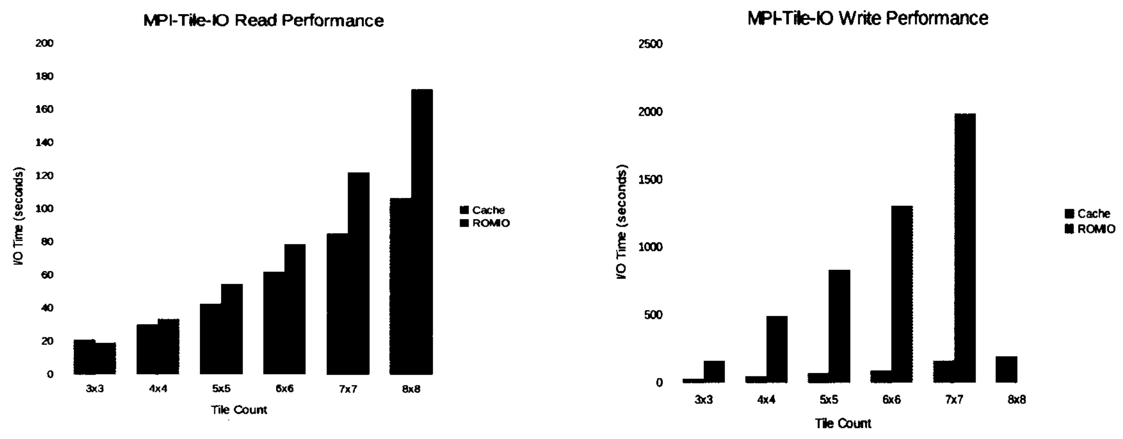

Figure 5.5. MPI-Tile-IO performance

2 The missing value for the 8 x 8 configuration using ROMIO indicates that the run did not finish during the 1 hour allotted run time.

The use of Interval I/O with the MPI-Tile-IO benchmark provides many of the advantages discussed in Chapter 4. First, it illustrates how Interval I/O can be used to facilitate cooperation between applications. Second, the producer's noncontiguous writes benefit from the reduction of false sharing and the use of interval files, and may be performed without acquiring locks since the Interval I/O system detects that none of the intervals are shared. Finally, the Interval-based locking system allows efficient locking in the case of the consumer.

5.3 2D-Column-IO

The 2D-Column-IO benchmark is designed to test the atomic mode capabilities of MPI-IO. File data consists of a large two dimensional array which is partitioned column-wise across a number of processes such that each process writes the same number of rows and columns. Columns written by adjacent processes overlap providing contention, and since MPI atomic mode is used, data written by each process must be atomic. The 2D-Column-IO data access pattern is shown in Figure 5.6.

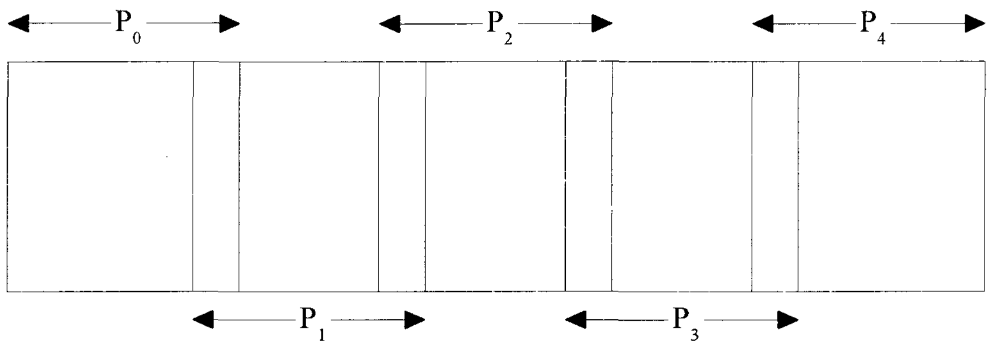

Figure 5.6. The 2D-Column-IO access pattern

We examined the performance of the 2D-Column-IO benchmark on Ranger. For our first experiment, we used a fixed 4 GB file size, and varied the number of processes participating in the write operation from 8 to 128. We began with 8 processes, each writing 16384 rows of data as a single atomic operation. For subsequent runs, we doubled the number of participating processes and reduced the number of rows by half. We compared the time to write the file using Interval I/O with that needed to write the file using ROMIO.

The results, shown in Figure 5.7, were striking for three reasons. First, it was immediately apparent that ROMIO's performance on this problem was extremely slow. In fact, we were only able to get ROMIO performance data for the 8 and 16 process cases. At 32 processes and beyond, our ROMIO jobs failed to complete before the allotted job time (one hour) was up. Secondly, it appeared that we were more than doubling the performance of Interval I/O by doubling the number of processes performing the writes. This is due not only to the fact that the amount of parallelism is increasing, but also because of the way that we adjusted the array to keep the file size fixed. By reducing the number of rows, we were decreasing the number of intervals that needed to be locked, and thus decreasing the overhead of locking. Finally, it was clear that Interval I/O performed extremely well compared to ROMIO for this test of atomic I/O.

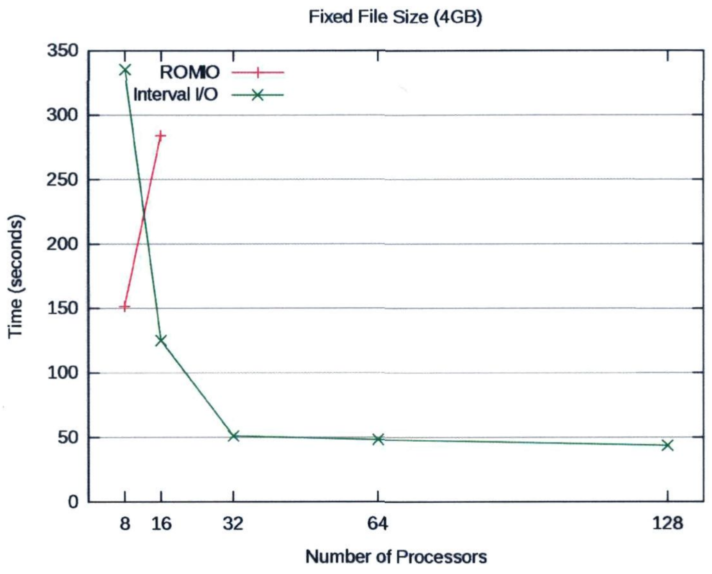

Figure 5.7. 2D-Column-IO performance, 4 GB total file size

For the second experiment with 2D-Column-IO, we kept the amount of data written by each process at a constant 256 MB, and again varied the number of processes between 8 and 128. In this experiment, we kept the number of rows written by each process fixed at 8192, and increased the number of columns with the number of processes.

The results of this experiment are shown in Figure 5.8. Again, ROMIO's handling of this case did not scale beyond 16 processes. The runs with 32 or more processes failed to complete in the allotted one hour run time of the jobs. The performance of the Interval I/O system was somewhat better, but still scaled poorly, with a doubling of processes resulting in more than twice the time to complete the atomic operation in cases beyond 32 processes. We suspected that lock overhead was to blame, since each doubling of processes also resulted in a doubling of the number of shared intervals, requiring a large number of lock messages.

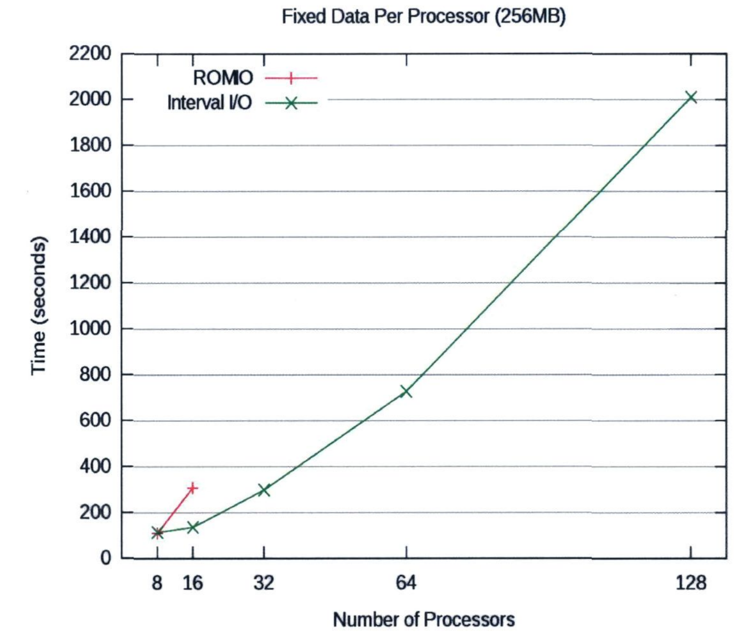

Figure 5.8. 2D-Column-IO performance, 256 MB per process

To test this hypothesis, we developed an alternate locking mode for Interval I/O, which we call Reverse Access Set (RAS) Locking. The RAS approach works as follows. Instead of maintaining a lock for every interval, we group intervals together that have the same reverse access set. So, for instance, the column of data that is shared by P_0 and P_1 has $\{P_0, P_1\}$ as the reverse access set for each interval in the column. By grouping them together, and simply requiring one lock to represent the entire column, we significantly reduce the lock overhead. This works quite well for the 2D-Column-IO benchmark, since all intervals in each column have identical reverse access sets. The access pattern produces only p – 1 distinct reverse access sets, greatly reducing the number of required locks.

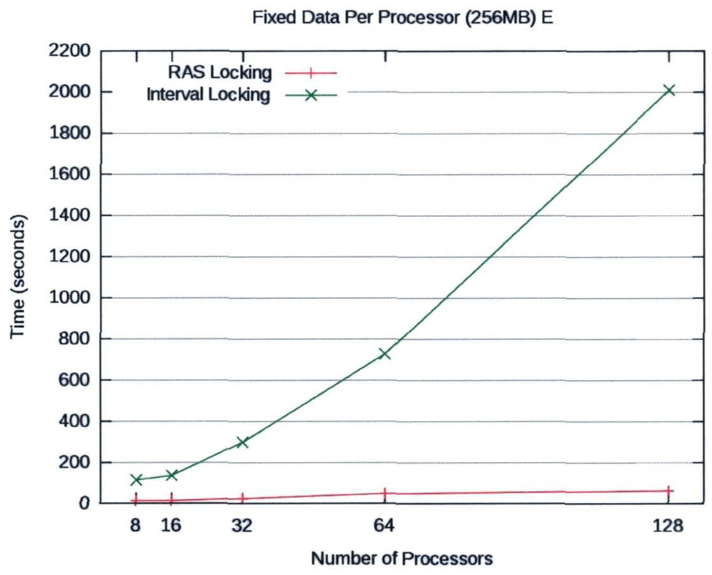

Figure 5.9. 2D-Column-IO performance with RAS Locking

The result obtained with this approach is shown in Figure 5.9. As can be seen, the Reverse Access Set Locking technique caused a significant improvement in atomic I/O performance in this case. RAS Locking would not be feasible for every situation, since the number of distinct reverse access sets would be significantly larger in many cases. In addition, programs which performed a larger number of smaller atomic writes would suffer from false sharing if the RAS locks represented a larger portion of the file than was being written at the time.

The performance of Interval I/O on the 2D-Column-IO benchmark demonstrates the effectiveness of our Interval-based locking system at handling noncontiguous atomic mode write operations. Both private interval detection and RAS locking reduce the expense of obtaining the required consistency semantics. The Interval I/O System performs significantly better than ROMIO on this benchmark.

6 LUSTRE FILE SYSTEM OPTIMIZATION

The Interval I/O System we have described is designed to efficiently handle the file access pattern being used by the application, and, when necessary, to reorder the data to allow application processes to perform separate, contiguous file system accesses. The results presented in Chapter 5 show that this approach can significantly improve the performance of parallel I/O.

Our extensive study of the Lustre file system [31,45,46], however, has produced evidence that large contiguous accesses do not always provide optimal performance. In fact, we have observed that performing a large number of small (noncontiguous) operations can, when the accesses are properly aligned with the Lustre storage architecture, provide significantly improved parallel I/O performance.

In this chapter we present our findings regarding the effects of access patterns on Lustre File System performance and show how the Interval I/O System provides a simple and effective mechanism for tuning the file system access pattern to take advantage of these findings.

6.1 Lustre Performance Overview

There are two key challenges associated with achieving high performance with MPI-IO in a Lustre environment. First, Lustre exports only the POSIX file system API, which was not designed for a parallel I/O environment and provides little support for parallel I/O optimizations. This has led to the development of approaches (or "workarounds") that can circumvent (at least some of) the performance problems inherent in POSIX-based file systems (e.g., two-phase I/O [20,25], and data-sieving [22]). The second problem is that the assumptions upon which these optimizations are based simply do not hold in a Lustre environment.

The most important and widely held assumption, and the one upon which most collective I/O optimizations are based, is that parallel I/O performance is optimized when application processes perform a small number of large, contiguous (non-overlapping) I/O operations concurrently. In fact, this is the assumption upon which collective I/O operations are based. The research presented in this chapter, however, shows that this assumption can lead to very poor I/O performance in a Lustre file system environment. Moreover, we provide a large set of experimental results showing that the antithesis of this approach, where each aggregator process performs a large number of small (non-contiguous) I/O operations, can, when properly aligned with the Lustre storage architecture, provide significantly improved parallel I/O performance.

In this chapter, we document and explain the reasons for these non-intuitive results. In particular, we show that it is the data aggregation patterns currently utilized in collective I/O operations, which result in large, contiguous I/O operations, that are largely responsible for the poor MPI-IO performance. Such data aggregation patterns are problematic because they redistribute application data in a way that conforms poorly with Lustre's object-based storage architecture. To address this issue, we have developed an alternative approach, embodied in a user-level library termed Y-Lib [46]. In a collective

I/O operation, Y-Lib redistributes data in a way that much more closely conforms to the Lustre object-based storage architecture.

In this chapter, we show how Interval I/O can be used to significantly improve parallel I/O performance in the emerging object-based file systems. We begin by demonstrating the data aggregation patterns that are much more closely aligned with Lustre (and other object-based file systems). We provide experimental results, taken across two large-scale Lustre installations, showing that this alternative approach to collective I/O operations does, in fact, provide significantly enhanced parallel I/O performance. However, we also show that the magnitude of such performance improvement depends on several factors, including the number of aggregator processes and Object Storage Devices, and the power of the system's communication infrastructure. We also show that the optimal data redistribution pattern employed by Y-Lib is dependent upon these same factors.

6.2 Lustre Architecture

Lustre consists of three primary components: file system clients (that request I/O services), object storage servers (OSSs) (that provide I/O services), and meta-data servers that manage the name space of the file system. Each OSS can support multiple Object Storage Targets (OSTs) that handle the duties of object storage and management. The scalability of Lustre is derived from two primary sources. First, file meta-data operations are de-coupled from file I/O operations. The meta-data is stored separately from the file data, and once a client has obtained the meta-data it communicates directly with the OSSs in subsequent I/O operations. This provides significant parallelism because multiple clients can interact with multiple storage servers in parallel. The second driver for scalable performance is the striping of files across multiple OSTs, which provides parallel access to shared files by multiple MPI-IO processes.

Lustre provides APIs allowing the application to set the stripe size, the number of OSTs across which the file will be striped (the stripe width), the index of the OST in which the first stripe will be stored, and to retrieve the striping information for a given file. The stripe size is set when the file is opened and cannot be modified once set. Lustre assigns stripes to OSTs in a round-robin fashion, beginning with the designated OST index.

The POSIX file consistency semantics are enforced through a distributed locking system, where each OST acts as a lock server for the objects it controls [65]. The locking protocol requires that a lock be obtained before any file data can be modified or written into the client-side cache. While the Lustre documentation states that the locking mechanism can be disabled for higher performance, we have never observed such improvement by doing so.

Previous research efforts with parallel I/O on the Lustre file system have shed some light on factors contributing to the poor performance of MPI-IO, including the problems caused by I/O accesses that are not aligned on stripe boundaries [58,66]. Figure 6.1 helps to illustrate this problem.

Assume two processes are writing to non-overlapping sections of a file; however, because the requests are not aligned on stripe boundaries, both processes are accessing different regions of stripe 1. Because of Lustre's locking protocol, each process must acquire the lock associated with the stripe, which results in unnecessary lock contention. Thus the writes to stripe 1 must be serialized, resulting in suboptimal performance.

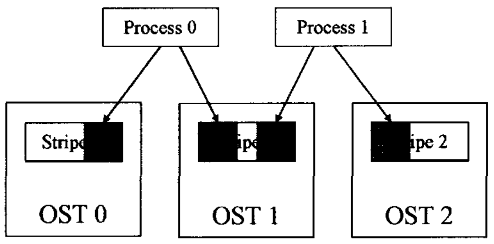

Figure 6.1. Crossing Stripe Boundaries with Lustre

An ADIO driver for Lustre has recently been added to ROMIO, appearing in the 1.0.7 release of MPICH2 [24]. This new Lustre driver adds support via hints for user settable features such as Lustre striping and direct I/O. In addition, the driver insures that two-phase I/O aggregation is performed such that disk accesses are aligned on Lustre stripe boundaries.

6.2.1 Data Aggregation Patterns

While the issues addressed by the new ADIO driver are necessary for high-performance parallel I/O in Lustre, they are not, in our view, sufficient. This is because they do not address the problems arising from multiple aggregator processes making large, contiguous I/O requests concurrently. This point may be best explained through a simple example.

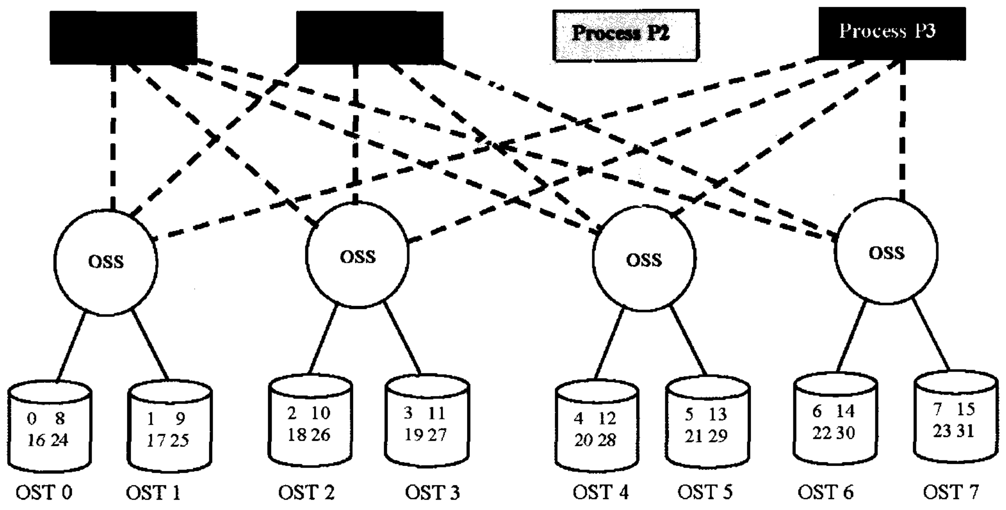

Figure 6.2. Communication pattern for two-phase I/O with Lustre.

Consider a two-phase collective write operation with the following parameters: four aggregator processes, a 32 MB file, a stripe size of 1 MB, eight OSTs, and a stripe width of eight. Assume the four processes have completed the first phase of the collective write operation, and that each process is ready to write a contiguous eight MB block to disk. Thus, process P0 will write stripes 0 – 7, process P1 will write stripes 8 – 15, and so forth. This communication pattern is shown in Figure 6.2.

Two problems become apparent immediately. First, every process is communicating with every OSS. Second, every process must obtain eight locks. Thus there is significant communication overhead (each process and each OSS must multiplex four separate, concurrent communication channels), and there is contention at each lock manager for locking services (but not for the locks themselves). While this is a trivial example, one can imagine significant degradation in performance as the file size, number of processes, and number of OSTs becomes large. Thus, a primary flaw in the assumption that performing large, contiguous I/O operations provides the best parallel I/O performance is that it does not account for the contention of file system and network resources.

6.2.2 Aligning Data with the Lustre Object Storage Model

The aggregation pattern shown in Figure 3 is what we term an *all-to-all* OST pattern because it involves all aggregator processes communicating will all OSTs. The simplest approach to reducing contention caused by such aggregation patterns is to limit the number of OSTs across which a file is striped. In fact, the generally recommended (and often the default) stripe width is four. While this certainly reduces contention, it also severely limits the parallelism of file accesses, which, in turn, limits parallel I/O performance. However, we believe it is possible to both reduce contention and maintain a high degree of parallelism, by implementing an alternative data aggregation pattern. This is accomplished via a user-level library termed *Y-lib*.

The basic idea behind Y-Lib is to minimize the contention for file system resources by controlling the number of OSTs with which each aggregator process communicates. On Ranger, we found the optimal data redistribution pattern to be what we term a *"one-to-one"* OST pattern, where the data is arranged such that each aggregator process communicates with exactly one OST. On BigRed, however, we found that a "one-to-two" OST pattern, where each aggregator process communicates with two OSTs, provided the best performance. The difference in observed optimal performance between BigRed and Ranger is due to the different hardware configurations of the two machines; further study would be required to determine the optimal pattern for an arbitrary platform.

A simple example should help clarify these ideas. Assume there are four application processes that share a 16 MB file with a stripe size of 1 MB and a stripe width of four (i.e., it is striped across four OSTs). Given these parameters, Lustre distributes the 16 stripes across the four OSTs in a round-robin pattern as shown in Figure 6.3. Thus stripes 0, 4, 8, and 12 are stored on OST 0, stripes 1, 5, 9, and 13 are stored on OST 1, and so forth.

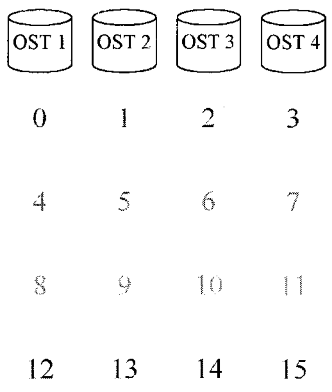

Figure 6.3. Lustre file layout

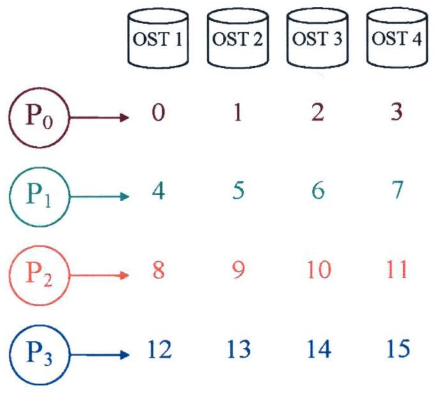

Figure 6.4. Each process has its data in the conforming distribution.

Figure 6.4 shows how the data would be distributed to the aggregator processes in what is termed the *conforming distribution,* where each process can write its data to disk in a single, contiguous I/O operation. This is the distribution pattern that results from the first phase of ROMIO's collective write operations, which, as previously discussed, results in an all to all communication pattern.

Figure 6.5 shows how the same data would be distributed by Y-Lib to create the one-to-one OST pattern. As can be seen, the data is rearranged to reflect the way it is striped across the individual OSTs, resulting in each process having to communicate with only a single OST.

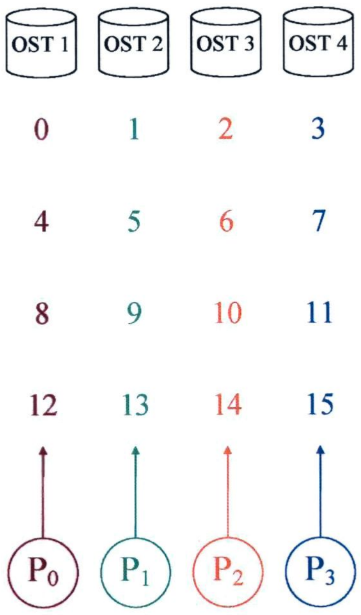

Figure 6.5. The one-to-one OST pattern

Figure 6.6 and Figure 6.7 show the data redistribution patterns for the conforming distribution and the two-to-one OST pattern, respectively.

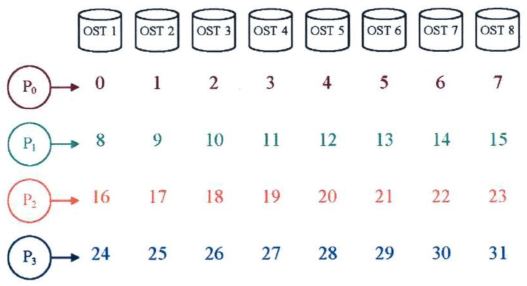

Figure 6.6. The conforming distribution

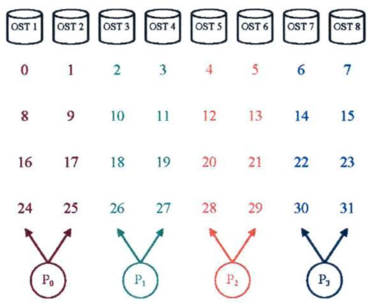

Figure 6.7. The one-to-two OST pattern after redistribution

6.2.3 Tradeoffs in the Aggregation Patterns

It is interesting to consider the trade-offs in the different data aggregation patterns. When the data is redistributed to the conforming distribution, each process can write its data to disk in a single, contiguous, I/O operation. However, this creates a great deal of background activity as the file system client must communicate with all OSTs. In the one-to-one OST distribution, there is significantly less contention for system resources, but each process must perform a (potentially) large number of small I/O requests, with a disk seek between each such request.

Thus the relative performance of the two approaches is determined by the particular overhead costs associated with each. In the following sections, we provide extensive experimentation showing that the costs associated with contention for system resources (OSTs, lock managers, network) significantly dominates the cost of performing multiple, small, and non-contiguous I/O operations.

6.3 Experimental Design

We wanted to determine the impact of the data aggregation patterns on the throughput obtained when performing a collective write operation in a Lustre file system. To investigate this issue, we performed a set of experiments on two large-scale Lustre file systems, at two different research facilities on the TeraGrid [63]: Indiana University and the Texas Advanced Computing Center at the University of Texas. Details of the two platforms, Big Red and Ranger, are given in Chapter 5.

Both Ranger and Big Red host production file systems that are heavily utilized by the scientific research community, and we were unable to obtain exclusive access to either file system for our testing. Thus, we were unable to control the number of other jobs, the I/O characteristics of such jobs, and the level of network contention during our experimentation. The primary problem with not having exclusive access is the potential for large variability in experimental results making them very difficult to interpret. However, as will be seen below, the level of variability in these results is not large, and we thus believe the results obtained here are reflective of what a user would experience when accessing these file systems.

It is important to note that the experimental environment is quite different on these two systems. Big Red has a smaller number of nodes (768 versus 3,936), and a significantly longer maximum runtime (two days to two weeks on Big Red versus 24 hours on Ranger). This resulted in very lengthy queues, where the number of waiting jobs often exceeded one thousand and was rarely less than seven hundred. Thus it was difficult to obtain a large number of nodes, and the time between experiments could be quite large, often taking between four days and one week.

For these reasons, we were able to complete a larger set of experiments, with a larger number of processes and OSTs, on Ranger than we were on Big Red. We begin by discussing our results on Ranger.

6.3.1 Experimental Study on Ranger

We varied two key parameters in the experiments conducted on Ranger: The number of processors that participated in the operation, and the file size. In particular, we varied the number of processors from 128 to 1024, where each processor wrote one gigabyte of data to disk. Thus the file size varied between 128 gigabytes and one terabyte. We kept the number of OSTs constant at 128, and maintained a stripe size of one megabyte. Each data point represents the mean value of 50 trials taken over a five-day period.

6.3.1.1 Data Aggregation Patterns with Redistribution

In this set of experiments, we assigned the data to the processors such that both ROMIO and Y-Lib were both required to redistribute the data to reach the desired aggregation pattern. Thus, in the case of ROMIO, we set a file view specifying the one-to-one OST pattern. When presented with such a pattern, ROMIO uses two-phase I/O to move the data into the conforming distribution. Once in the conforming distribution, each aggregator process writes a single contiguous chunk of data to disk.

In the case of Y-Lib, we assigned the data to the processors in the conforming distribution, and made a collective call to Y-Lib to redistribute the data to the one-to-one OST pattern. Once Y-Lib completed the data redistribution, it wrote the data to disk using multiple independent (but concurrent) write operations.

6.3.1.2 Data Aggregation Patterns without Redistribution

The next set of experiments assumed the data was already assigned to the processors in the required distribution thus negating the need for data redistribution. Thus in the case of ROMIO, each process performed a single contiguous write operation. In the case of Y-Lib, each process performed multiple independent write operations.

6.3.1.3 ROMIO Write Strategies

The final set of experiments was designed to determine if we could improve the performance of MPI itself by forcing it to use the one-to-one OST pattern rather than the conforming distribution. We accomplished this by setting a file view specifying the one-to-one OST pattern, and disabling both two-phase I/O and data sieving. Thus each process was required to perform multiple independent writes. This forces ROMIO to perform the same writes that would be performed by Y-Lib. We then compared the performance of this approach with that of ROMIO where the data was already in the conforming distribution, and ROMIO using two-phase I/O.

6.3.1.4 Experimental Results

The experimental results are shown in Figure 6.8, 6.9, and 6.10. Each data point represents the measured throughput averaged over 50 trials and 95% confidence intervals around the means. Figure 6.8 shows the throughput obtained when both Y-Lib ROMIO were required to redistribute the data before performing the write operations. As can be seen, Y-Lib improves I/O performance by up to a factor of ten. This is particularly impressive given that each process performed 1024 independent write operations.

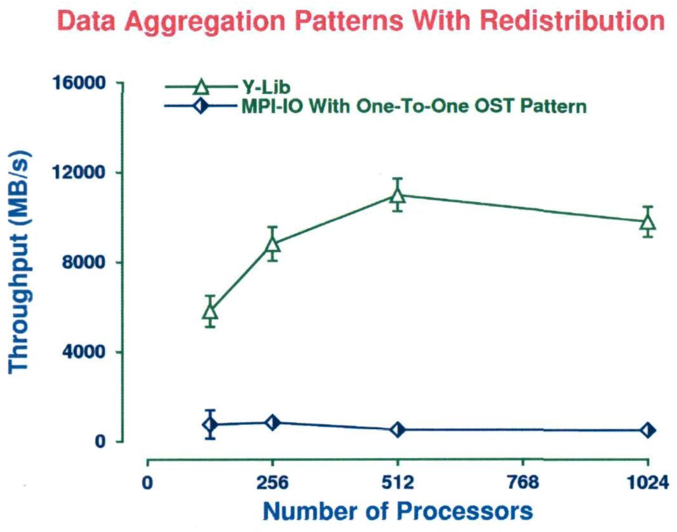

Figure 6.8. Mean throughput with data redistribution

Figure 6.9 shows the throughput obtained assuming the optimal data distribution for each approach. That is, the data was in the conforming distribution for MPI-IO, and in the one-to-one OST distribution for Y-Lib. Thus neither approach required the redistribution of data. As can be seen, the one-to-one pattern, which required 1024 independent write operations, significantly outperformed the **MPI_File_write_at_all** operation, where each process wrote a contiguous one gigabyte buffer to disk. In this case, Y-Lib improved performance by up to a factor of three.

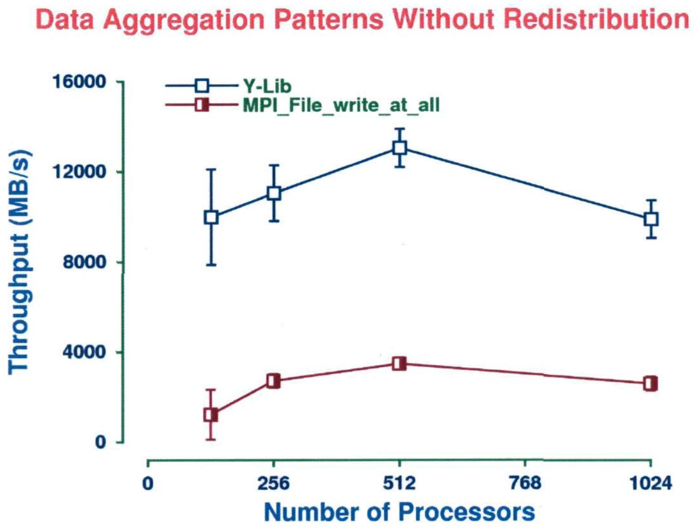

Figure 6.9. Mean throughput without data redistribution

Figure 6.10 depicts the performance of three different MPI-IO collective operations. It includes the two previously described approaches, and compares them with the performance of MPI-IO when it was forced to use independent writes. As can be seen, we were able to increase the performance of MPI-IO itself by over a factor of two, by forcing it to use the one-to-one OST pattern.

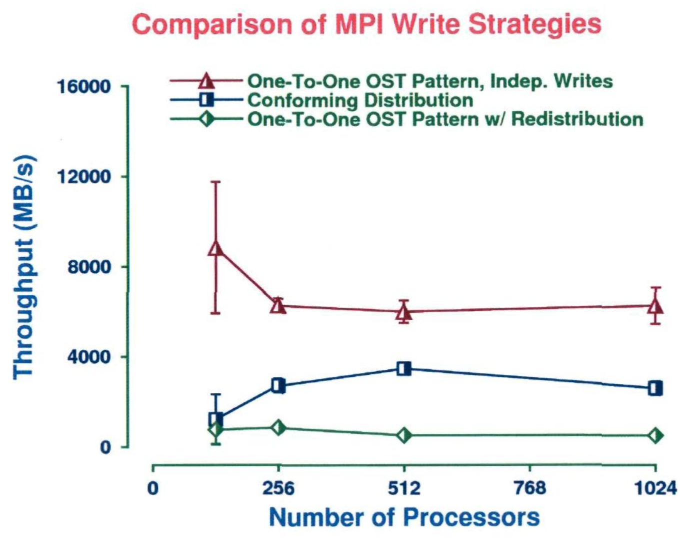

Figure 6.10. A comparison of MPI write strategies

6.3.1.5 Discussion

These results strongly support the hypothesis that ROMIO does, in fact, perform very poorly in a Lustre file system because of the resource contention associated with the all-to-all pattern. They also show that it is possible to utilize all of the system resources quite profitably when utilizing the data redistribution pattern employed by Y-lib. These results also lend strong support to other studies on Lustre showing that maximum performance is obtained when individual processes write to independent files concurrently [56,67]. It also helps explain the commonly held belief of (at least some) Lustre developers that parallel I/O is not necessary in a Lustre environment, and does little to improve performance.

6.3.2 Experimental Studies on Big Red

In our initial exploration of Y-lib on Big Red, we did not obtain the improvement in I/O performance that we observed on Ranger. Further investigation revealed that we were under-utilizing the powerful parallel I//O subsystem by having each process communicate with only one OST. We then experimented with other OST patterns, and found that the best performance was obtained when each process communicated with exactly two OSTs (what we term a *two-OST* pattern). Thus all of the experiments discussed in this section utilized this data redistribution pattern.

6.3.2.1 Data Aggregation Patterns without Redistribution

In these experiments, we compared the I/O performance obtained when the data was arranged according to the conforming distribution or the two-OST distribution. We varied the number of aggregator processes between 32 and 256, and the stripe width between 32 and 96 (the maximum number of OSTs available). We scaled the file size between 32 and 256 gigabytes (*i.e.*, one gigabyte times the number of processes), and, for 32 to 96 processes, set the stripe width equal to the number of processes. In the case of 192 processes, we utilized 96 OSTs. In the 256-process case however, we utilized only 64 OSTs. This was because the number of processes must be a multiple of the number of

OSTs to ensure that each process always communicates with the same two OSTs. In all cases, the stripe size was one megabyte, and the writes were aligned on stripe and lock boundaries.

6.3.2.2 Experimental Results

The results of these experiments are shown in Figure 6.11. It shows the mean throughput and 95% confidence intervals around the means, as a function of the number of processes and I/O strategy. As can be seen, the Y-lib distribution pattern starts to significantly outperform the conforming distribution once the number of processes exceeds 32. The largest improvement comes with 96 processes (and OSTs), where a 36% improvement in performance is observed. The relative improvement in performance was approximately 32% with 192 processes (96 OSTs), and was on the order of 5% in the 256-process case (64 OSTs).

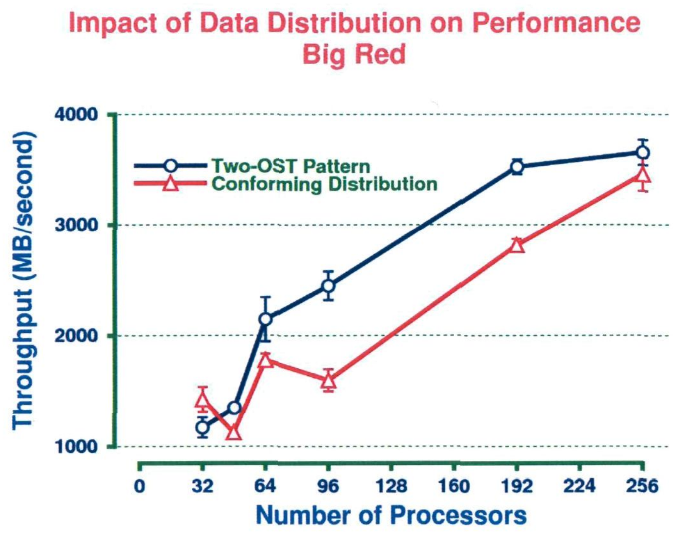

Figure 6.11. Impact of data distribution (Big Red)

6.3.2.3 Discussion

These results are very different from those obtained on Ranger, and it is interesting to consider the causes for such differences. There are really two separate questions: Why did the performance of ROMIO increase with increasing numbers of processes, and why did the rate of performance increases begin to slow for Y-lib in this scenario? We address each question in turn.

We believe the increasing performance observed on Big Red was due to the very powerful parallel I/O subsystem available from the Data Capacitor, combined with an aggregate bandwidth of 240 gigabits per second between the two systems provided by the 24 10-gigabit Myricom connections. Clearly, this infrastructure was powerful enough to handle the all-to-all communication pattern required by the conforming distribution. However, the number of processes we were able to test was relatively small (at least compared to Ranger), and it would be very interesting to execute the same tests with 512 processes.

The reduction in the rate of increasing performance observed in Y-lib was, we believe, related to the ratio of OSTs to aggregator processes. That is, the overhead of performing a large number of small I/O operations becomes more pronounced as contention for OSTs and network services begins to increase. In the case of 256 aggregators and 64 OSTs, each OST is communicating with eight processes (even though each process is only communicating with two OSTs). Thus, while the level of contention in this case is not as significant as that resulting from the conforming distribution, it is apparently enough to begin to impact the performance of Y-lib.

6.4 Striped Interval Files

In our initial development of the Interval I/O System, a guiding principle for optimizing I/O performance was to favor large, contiguous I/O operations rather than using a larger number of smaller, noncontiguous operations. This proved to be a reasonably successful strategy, as indicated by the results presented in Chapter 5. However, our work with the Lustre file system calls this principle into question. Our results indicate that the performance obtained from using a single large, contiguous access can be significantly slower than a noncontiguous access pattern that aligns properly with Lustre's storage architecture.

Fortunately, our Interval I/O System is well suited to take advantage of this new information. Since interval files do not store data in the order dictated by the use of a flat file, their use effectively decouples the application's access pattern from the file system access pattern. This allows file data to be written to disk using a pattern that is known to be efficient provided that the file metadata includes a description of the pattern that was used. A significant advantage of Interval I/O is that it *does not require any additional communication* to obtain this performance improvement.

In Section 3.4.1, we described the layout of the Interval Files. The technique described there aimed to store all data for a process contiguously on disk. Here we show a small modification to the technique which allows process data to be arranged in the one-to-one or one-to-two pattern as explained in Chapter 6.

Consider the parallel I/O example given in Figure 6.12, which shows cached interval data written by three processes to a Lustre file system. The Lustre file is striped across three OSTs as indicated by the dashed lines in the figure. Figure 6.12a shows the cached data written as a conventional sequential file. The pattern requires each process to write its cached data to three noncontiguous data regions, which, in this case requires each process to write data to all three of the OSTs. This is an example of the all-to-all access pattern discussed in Section 6.2.2 which leads to particularly poor performance with Lustre.

Figure 6.12b shows the same cached data written as an Interval File using the scheme described in Section 3.4.1. In this case, each process writes one contiguous data region to the file. We have shown that this technique can improve write performance, however, the amount of improvement is limited because this pattern still produces an all-to-all pattern requiring each process to communicate with every OST.

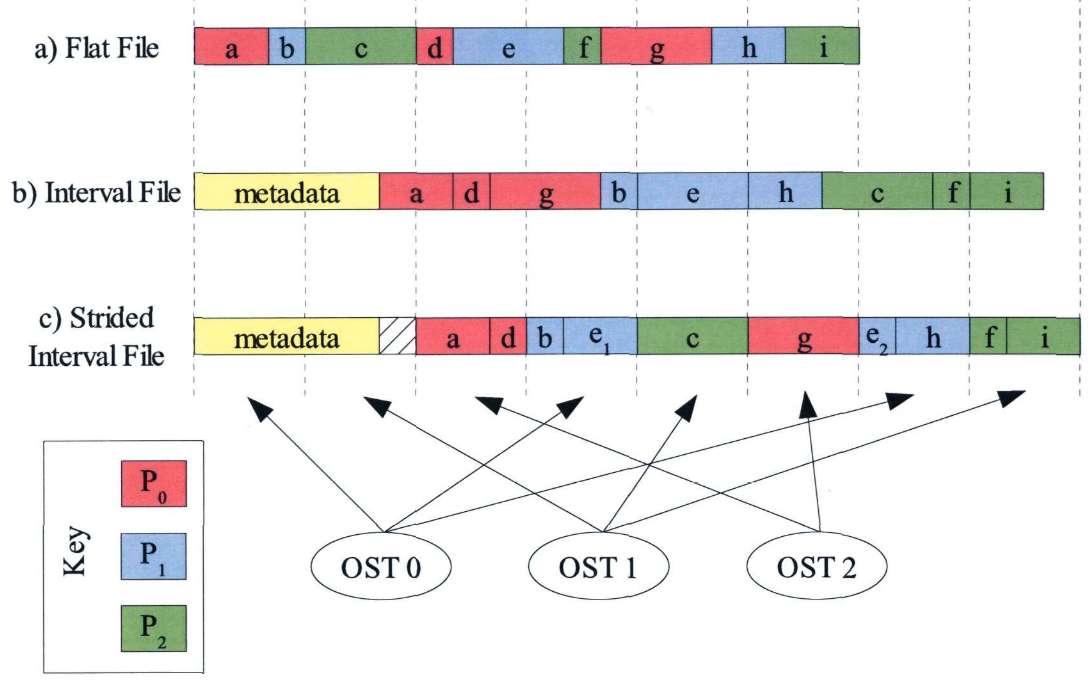

Figure 6.12. Optimizing the Interval File layout for Lustre

To overcome this obstacle, we adjust the Interval File as shown in Figure 6.12c. Here the Interval File is tuned to use a one-to-one pattern for the file data. There are three adjustments to be made, which are:

1. Align the data section with the beginning of the first full stripe after the metadata section. In the example, this is the first stripe assigned to OST 2.
2. Assign each process a set of stripes corresponding to a particular OST. Here, process P_o is assigned the stripes of OST 2, P_1 is assigned OST 0's stripes, and P_2 uses OST 1's stripes. Each process simply writes data in a regular strided pattern into the assigned stripes.

3. Adjust the striping information in the metadata section to reflect the location of the data.

7 CONCLUSION AND FUTURE RESEARCH

This research has investigated the use of an interval-based approach to parallel I/O. In this chapter, we summarize our findings, and present some directions for further study.

7.1 Conclusion

Interval I/O provides a promising new approach to parallel I/O. By considering the available file view information and using it to decide the fundamental access units, we have shown that I/O performance can be significantly improved. The prototype Interval I/O System that we have developed not only provides evidence of this improvement, but also provides a platform upon which future research on Interval I/O can be performed.

This work has shown Interval I/O to be effective in a variety of situations. The interval I/O approach is well suited for handling noncontiguous I/O operations as it reduces false sharing and allows file system accesses to be done contiguously. The Interval I/O locking system provides excellent scaling of atomic mode operations, particularly when compared to ROMIO. Our Interval Files allow accesses to be performed in a way that conforms to the best available access pattern for the underlying file system (*e.g.* Lustre). And finally, we have outlined a strategy to support cooperating applications, and have laid the groundwork for continued research.

7.2 Future Research

In this section we discuss some related areas that would be natural extensions to this work. Items discussed here are included as suggestions to those wishing to extend this research.

A potential research area stemming from this work is the handling of partially cached files, which would be necessary for accessing files that are larger than the collectively available memory across the application processors. Designing such a mechanism would be non-trivial, but would extend the usability of interval-based caching to include arbitrarily large files.

Our initial approach for calculating interval sets described earlier has a potential problem when used with very large numbers of processors. As the problem size and processor count increase, the total number of intervals also increases. Thus the current approach has limited scalability because it requires each process to calculate every interval in the file, regardless of whether a particular interval is used by that process.

A more scalable approach would involve distributing the responsibility for creating intervals across the processors by breaking the file into p contiguous chunks, where p is the number of participating processors. With this approach, a process sends the appropriate file view information only to the processes whose assigned chunks overlap with its file view. Each process then calculates the intervals in its chunk, which can be done in parallel. Finally, the metadata for each interval is sent back to each process in that interval's reverse access set, providing each process with data for only the intervals in its view.

Another area of interest involves analysis of the logic used to place shared intervals on cache processors, and the related question of lock placement. We expect that there will be a class of access patterns that will require a more intelligent approach to interval and lock placement than that which is used in the prototype system. A better algorithm for placing intervals and locks would allow the cache to provide increased performance or a wider range of application access patterns.

Another possible extension of interval-based parallel I/O is the addition of support for parallel streaming of interval-based data through a mechanism similar to Unix pipes. The system would use the Interval Translator to dynamically remap interval-based streams betwen a source interval stream and a target interval stream. Use of such a system could result in a more modular set of tools for parallel programming, similar to the way specialized tools such as cat, grep and sed are often used in Unix scripts.

Our system places knowledge of intervals inside ROMIO, so an application has no direct contact with the interval sets. However, there may be advantages to providing the application with a set of tools to directly access and manipulate the intervals. The design of an interval-based file API would be another interesting area for future research.

REFERENCES

[1] "Top 500 Supercomputing Sites" Available: http://www.top500.org/.

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

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

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

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

[6] 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, Aug. 2005, pp. 177-186.

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

[8] A. Ching, Feng, W., Lin, H., X. Ma, and A. Choudhary, "Exploring I/O Strategies for Parallel Sequence Database Search Tools with S3aSim.," *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," *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, IEEE Computer Society, 2003, p. 24.

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

[11] D. Kotz and R. Jain, "I/O in Parallel and Distributed Systems," *Encyclopedia of Computer Science and Technology*, A. Kent and J.G. Williams, eds., Marcel Dekker, Inc., 1999, pp. 141–154.

[12] A. Choudhary, W. Liao, K. Gao, A. Nisar, R. Ross, R. Thakur, and R. Latham, "Scalable I/O and analytics," *Journal of Physics: Conference Series*, vol. 180, 2009, p. 012048.

[13] P. Crandall, R.A. Aydt, A.A. Chien, and D.A. Reed, "Input/Output Characteristics of Scalable Parallel Applications," *IN PROCEEDINGS OF SUPERCOMPUTING '95*, 1995.

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

[15] A. Ching, A. Choudhary, W. Liao, L. Ward, and N. Pundit, "Evaluating I/O Characteristics and Methods for Storing Structured Scientific Data," 2006.

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

[17] R. Ross, R. Thakur, and A. Choudhary, "Achievements and challenges for I/O in computational science," *Journal of Physics: Conference Series*, vol. 16, 2005, pp. 501-509.

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

[19] "The Open Group Base Specifications Issue 7, IEEE Std 1003.1™-2008" Available: http://www.opengroup.org/onlinepubs/9699919799/.

[20] R. Thakur, W. Gropp, and E. Lusk, "On implementing MPI-IO portably and with high performance," *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, Atlanta, Georgia, United States: 1999, pp. 23-32.

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

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

[23] R. Thakur, R. Ross, and W. Gropp, "Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation."

[24] "MPICH2: High-performance and Widely Portable MPI" Available: http://www.mcs.anl.gov/research/projects/mpich2/.

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

[26] P.M. Dickens and R. Thakur, "A Performance Study of Two-Phase I/O," *IN PROCEEDINGS OF THE 4TH INTERNATIONAL EURO-PAR CONFERENCE. LECTURE NOTES IN COMPUTER SCIENCE 1470*, 1998, pp. 959--965.

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

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

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

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

[31] P. Dickens and J. Logan, "Towards a High Performance Implementation of MPI-IO on the Lustre File System," *On the Move to Meaningful Internet Systems: OTM 2008*, 2008, pp. 870-885.

[32] D. Kotz, "Disk-directed I/O for MIMD multiprocessors," *ACM Trans. Comput. Syst.*, vol. 15, 1997, pp. 41-74.

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

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

[35] X. Ma, M. Winslett, J. Lee, and S. Yu, "Improving MPI-IO Output Performance with Active Buffering Plus Threads," *IPDPS 2003*, 2003.

[36] J.F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS)," *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, Boston, MA, USA: ACM, 2008, pp. 15-24.

[37] W. Liao, A. Choudhary, K. Coloma, G. Thiruvathukal, L. Ward, E. Russell, and N. Pundit, "Scalable Implementations of MPI Atomicity for Concurrent Overlapping I/O," *International Conference on Parallel Processing*, 2003.

[38] P. Aarestad, A. Ching, G. Thiruvathukal, and A. Choudhary, "Scalable Approaches for Supporting MPI-IO Atomicity.," *6th International Symposium on Cluster Computing and the Grid (CCGrid)*, 2006.

[39] P. Dickens and R. Thakur, "Improving Collective I/O Performance Using Threads," *Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, IEEE Computer Society, 1999, pp. 38-45.

[40] N. Nieuwejaar, D. Kotz, A. Purakayastha, C.S.E. Y, and M.B. Z, "File-Access Characteristics of Parallel Scientific Workloads," *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, vol. 7, 1996, pp. 1075--1089.

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

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

[43] P.M. Dickens and J. Logan, "Improving the Performance of MPI-IO Using Object-Based Caching, University of Maine Technical Report T-09-3," Aug. 2009.

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

[45] P.M. Dickens and J. Logan, "A high performance implementation of MPI-IO for a Lustre file system environment," *Concurrency and Computation: Practice and Experience*, 2009.

[46] P.M. Dickens and J. Logan, "Y-lib: a user level library to increase the performance of MPI-IO in a lustre file system environment," *Proceedings of the 18th ACM international symposium on High performance distributed computing*, Garching, Germany: ACM, 2009, pp. 31-38.

[47] D.E. Singh, F. Isaila, A. Calderon, F. Garcia, and J. Carretero, "Multiple-Phase Collective I/O Technique for Improving Data Access Locality," *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 534-542.

[48] K. Coloma, A. Choudhary, A. Ching, W. Liao, S. Son, M. Kandemir, and L. Ward, "Power and Performance in I/O for Scientific Applications," *19th IEEE International Parallel and Distributed Processing Symposium*, Denver, CO, USA: .

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

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

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

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

[53] A. Ching, W. Liao, A. Choudhary, R. Ross, and L. Ward, "Noncontiguous locking techniques for parallel file systems," *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, Reno, Nevada: ACM, 2007, pp. 1-12.

[54] "NetCDF (network Common Data Form)" Available: http://www.unidata.ucar.edu/software/netcdf/.

[55] "The HDF Group" Available: http://www.hdfgroup.org/.

[56] W. Yu, J. Vetter, R.S. Canon, and S. Jiang, "Exploiting Lustre File Joining for Effective Collective IO," *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, IEEE Computer Society, 2007, pp. 267-274.

[57] J.M. Larkin and M.R. Fahey, "Guidelines for Efficient Parallel. I/O on the Cray XT3/XT4," 2007.

[58] W. Liao, A. Ching, K. Coloma, A. Choudhary, and Lee Ward, "An Implementation and Evaluation of Client-Side File Caching for MPI-IO," *2007 IEEE International Parallel and Distributed Processing Symposium*, Long Beach, CA, USA: 2007, pp. 1-10.

[59] P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley Publishing Company, 1987.

[60] W. Liao, K. Coloma, A. Choudhary, and L. Ward, "Cooperative Write-Behind Data Buffering for MPI I/O," *the 12th European PVM/MPI Conference*, 2005, pp. 102-109.

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

[62] R. Sedgewick, "Left-Leaning Red Black Trees" Available: http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf.

[63] "TeraGrid," *TeraGrid* Available: https://www.teragrid.org.

[64] "FLASH I/O benchmark routine -- parallel HDF 5," *FLASH I/O benchmark routine -- parallel HDF 5* Available: http://flash.uchicago.edu/~zingale/flash_benchmark_io/.

[65] P. Braam and Others, "The Lustre storage architecture," *White Paper, Cluster File Systems, Inc., Oct*, vol. 23, 2003.

[66] W. Liao, A. Ching, K. Coloma, A. Choudhary, and M. Kandemir, "Improving MPI Independent Write Performance Using A Two-Stage Write-Behind Buffering Method," *2007 IEEE International Parallel and Distributed Processing Symposium*, Long Beach, CA, USA: 2007, pp. 1-6.

[67] "Lustre: scalable, secure, robust, highly-available cluster file system." Available: www.lustre.org/.

BIOGRAPHY OF THE AUTHOR

Jeremy Logan was born in Scranton, Pennsylvania. He was raised in central Maine, and graduated from Penquis Valley High School. In 2001, he earned the B.S. degree in Computer Science from the University of Southern Maine in Portland, Maine. Jeremy received the M.S. degree in Computer Science from the University of Maine in Orono, Maine in 2006.

After receiving his degree, Jeremy will join the National Center for Computational Science at Oak Ridge National Laboratory as a Postdoctoral Research Associate. He is a candidate for the Ph.D. degree in Computer Science from the University of Maine in December, 2010.