A New Object Based File Model To Improve Parallel I/O Performance

Prepared By

Sanket Parmar

10MICT21

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AHMEDABAD-382481

May 2012

A New Object Based File Model To Improve Parallel I/O Performance

Major Project

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology in Information and Communication Technology

Prepared By

Sanket Parmar

10MICT21

Guided By

Prof. Zunnun Narmawala

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AHMEDABAD-382481

May 2012

Declaration

This is to certify that

i) The thesis comprises my original work towards the degree of Master of Technology in Information and Communication Technology at Nirma University and has not been submitted elsewhere for a degree.

ii) Due acknowledgement has been made in the text to all other material used.

Sanket Parmar

Certificate

This is to certify that the Major Project entitled **"A New Object Based File Model To Improve Parallel I/O performance"** submitted by Parmar Sanket S. (10MICT21), towards the partial fulfillment of the requirements for the degree of Master of Technology in Information and Communication Technology of Nirma University of Science and Technology, Ahmedabad is the record of work carried out by him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Prof. Zunnun Narmawala
Guide, Associate Professor,
Department Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Prof. Gaurang Raval
Associate Prof. and PGICT-Coordinator,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Prof. D. J. Patel
Professor and Head,
Department Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Dr. K. Kotecha
Director,
Institute of Technology,
Nirma University, Ahmedabad

Abstract

Advanced data intensive scientific applications with thousands of processing nodes are used to execute very high resolution scientific model. 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. Poor I/O performance has been a bottleneck in these applications. The main reason for this I/O bottleneck is the non-contiguous I/O access patterns exhibited by scientific applications.

Most of the file systems stored the file as linear sequences 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.

This thesis addresses the storage problem of application data in parallel file system's legacy view. We develop a new file model and required software infrastructure. With the help of MPI views, we create intervals and store those intervals as objects. We use the data layout awareness technique to distribute the objects on I/O servers. This provides contiguous, single I/O operations. Result shows the significant improvement in read operations.

Acknowledgements

My deepest thanks to **Prof. Zunnun Narmawala**, the guide of the project that I undertook for giving his valuable inputs and correcting various documents of mine with attention and care. He has taken the pain to go through the project and make necessary amendments as and when needed.

My deep sense of gratitude to **Prof.Gaurang Raval**, for an exceptional support and continual encouragement throughout the major project.

I would like to thanks **Prof. D. J. Patel** for his unmentionable support, providing basic infrastructure and healthy research environment.

I would like to thanks **Dr. Ketan Kotecha** for his unmentionable support, providing basic infrastructure and healthy research environment.

I would also thank my Institution, all my faculty members in Department of Computer Science and my colleagues without whom this project would have been a distant reality. Last, but not the least, no words are enough to acknowledge constant support and sacrifices of my family members because of whom I am able to complete my dissertation work successfully.

Sanket Parmar

10MICT21

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Parallel computing has been an active area of research for last couple of decades. Parallelism is one of the key techniques to achieve high performance. Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel") [5]. The rate at which these applications create and use large-scale data sets has spurred the design of petabyte-scale file systems which consist of potentially billions of files [25]. These large-scale computing clusters are coupled with state of the art parallel file systems such as Lustre [1], PVFS [6], and Panasas [4], which offer massive storage capabilities and are designed to provide scalable access to thousands of clients concurrently. Parallel I/O API, such as MPI (Message Passing Interface) [3], designed to provide access to advanced parallel hardware, MPI support large-scale applications executing in such extreme environments by providing sophisticated mechanisms for message passing and process management [16]. MPI-IO was developed in order to provide parallel I/O support for MPI. It's a message parsing system used in most of the super clusters. It defines an I/O access interface for parallel applications. With the use of MPI library calls, programmers can easily transfer data between tasks without having to resort to low level TCP/IP calls.

Scientific applications such as climate modeling, earthquake modeling, and genomic pattern matching simulations might sometimes require larger amount of data. Such data-intensive applications manipulate data sets in order of petabytes. These applications are

capable of executing very high resolution scientific models. Parallelism provides high performance framework to get maximum throughput.

The problem however, is that large scale applications have I/O throughput requirements on the order of tens of gigabytes per second. 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 [17]. For a variety of reasons, the I/O subsystems of such parallel computing systems, have not kept pace with the computational increases, and the time required for I/O in an application and become one of the dominant bottlenecks. This problem, known as the scalable I/O problem [13], 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 [9]), poor file system support for parallel I/O optimizations [20], enforcing strict file consistency semantics [15], and the latency of accessing I/O devices across a network. A fundamental problem is the legacy view of a file as a linear sequence of bytes. Stored file in disk is nothing than a 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. Generally scientific applications don't use this continuous access pattern. In fact, the data model used by application processes is more accurately described as an object model, where each process maintains a set of unrelated objects [16]. In this model, each object is part of file data. Each object is contiguous in the file.

This research is addressing the scalable I/O problem by developing new file model and proposing the software infrastructure required for its support. The approach is based on term intervals [16], 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.

1.1 General Overview

Parallel computing systems have shown a huge performance increase over the last decades. This increasing performance allows parallel computing systems to process problems of increased size. This increasing performance allows parallel computing systems to process more input data and generates more output data. This data needs to be transferred from and to the underlying file systems. The I/O performance growth rate of traditional file systems, however, cannot keep up with the pace of processing performance [10] [18]. The standard access to I/O resources is serialized. This results in an I/O bottleneck, where the I/O subsystem is limiting the overall application performance.

The most obvious solution to achieve high performance is to use parallel I/O. It uses multiple connections to multiple I/O storage resources. By utilizing multiple connections to data, applications can access the storage resources in parallel, increasing the overall achievable I/O performance. Parallel file systems support this approach, mainly by distributing file data (data strips) among multiple IO storage servers. This can increase the achievable I/O performance up to the product of the number of IO servers to different I/O devices and the performance of a single I/O device.

This standard approach of parallel file systems can become a bottleneck again, when the file distribution parameters do not fit the access schemes of applications. A typical I/O device delivers best performance when accessing the data in large contiguous blocks. Another potential bottleneck is the communication of file system servers and clients. When client needs to communicate with servers that do not hold the requested data, the performance decreases. To overcome these issue different kind of optimization strategies are used at different level of the system.

1.1.1 Sequential I/O

POSIX(Portable Operating System Interface) is used as most commonly used file system interface. POSIX was designed for serial file access by a single process. It provides basic functionality for a single process. Fore example, it provides system calls for reading from and writing to a file, and seeking to a given location, locking), but provides no specific support for parallel I/O. This makes providing high-performance I/O in a parallel environment

quite difficult. This encourages the research activity aimed at overcoming (or providing work around for) the limitations of the POSIX API.

1.1.2 Parallel File Systems

Parallel file systems offer the high performance necessary for running complex parallel scientific applications. To provide maximum scalability, the machines processors are generally divided into compute nodes and I/O nodes. The main advantages a parallel file system can provide include a global name space, scalability, and the capability to distribute large files across multiple nodes. In a cluster environment, large files are shared across multiple nodes, making a parallel file system well suited for I/O subsystems. Parallel file systems are designed to concurrently provide access to large 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.

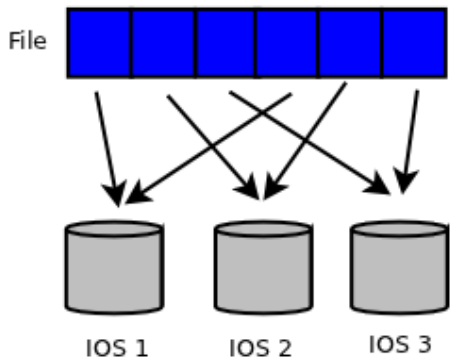

Figure 1.1: File strips over multiple IO servers

Figure 1.1 illustrates file striping in a parallel file system. The long rectangle at the top represents a large file divided into equal-sized sections called stripes. The 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

Multiple processes of a parallel program accessing data (reading or writing) from a common file. Parallel I/O has been developed to take advantage of the additional bandwidth provided by parallel file systems. 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

MPI is a standard message passing library that provides a number of point-to-point and collective communication primitives. It is a very widely used library for high performance computing clusters. MPI offers a standard interface that allows portability between systems with different operating systems, memory models, or interconnection networks. MPI-2, offers additional features such as dynamic process management and support for parallel I/O. Writing is like sending a message and reading is like receiving.

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.

1.1.5 MPI-IO

MPI-IO is an extension to MPI that would incorporate independent and collective I/O operations, asynchronous I/O calls. File access vi independent file pointers, shared file pointers, and explicit offsets, local and distributed data types. MPI-IO provides function calls that abstract the underlying file system structure. Through a series of internal system calls, MPI is able to convert the system specific name to a universal handler [11]. 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

Many applications define file views to express the accessed data of a file for each process. A file view describes the parts of a file that are visible to each process. File views are for example used to describe striped partitioning of files among multiple processes. By setting file views the processes can access their parts as if they had a linear address space. If its data is stored on disk as it is defined in the file view, only a single I/O operation is required to move the data to and from the disk. However, if the data is stored non-contiguously on disk, multiple I/O operations are required.

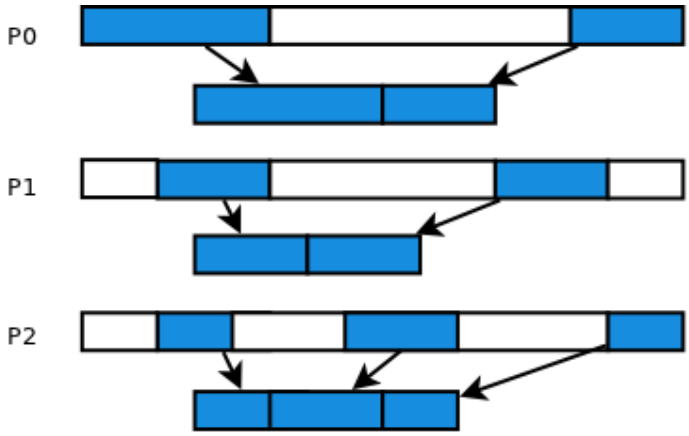

Figure 1.2: File views

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 shows 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.

Setting a file view is a collective operation, which means that all processes sharing a given file must participate in the operation. After this collective call, runtime system has information about the file access patterns of each process.

1.1.7 ROMIO

ROMIO [7] is most widely used implementation of MPI-IO [3] which was developed at Argonne National Laboratory. ROMIO delivers high performance in the presence of non-contiguous I/O requests. ROMIO is a portable MPIIO implementation that works on many different machines and file systems. The portability of ROMIO stems from an internal layer termed ADIO (Abstract Device Interface for parallel I/O) [22] upon which ROMIO implements the MPI-IO interface. ADIO implements the file system dependent features, and is thus implemented separately for each file system.

1.2 What are the problems?

As discussed earlier, I/O requirement of large-scale scientific applications are very large. This massive I/O requirement leads the system performance to the significant bottleneck. This is a critical problem for scientific applications and lot of research work has been going on to solve this problem.

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 are the most cited problems to achieving high-performance I/O. Legacy representation of file as a linear sequence of bytes is the root cause of all these issues in parallel I/O. 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

As noted earlier, MPI provides one important feature "MPI file views". We can use this feature to develop scalable infrastructure to merge the power and flexibility of MPI-IO interface with interval-based file model. Intersections between MPI views are known as intervals. Using intervals, we can identify shared and private regions of the files. Intervals are also known as objects.

1.3.1 System Design

The interval-based I/O infrastructure consists of five primary components: an interval integration interface, an interval cache, a lock management system, an interval-based file layer, and an interval set transformation tool. We will discuss these components in implementation chapter.

1.4 Scope of the Work

This research work deals with parallel I/O performance. Initial problem is the typical file systems store the file data sequentially. Other problem in parallel file system is storing pattern. Typical parallel file system uses round robin algorithm to store file strips to multiple I/O servers. The proposed solution will store the file data as an interval objects and collect all the relative objects and store it on one I/O server. This provides I/O servers to read large continuous blocks. The scope of this study would be restricted to mentioned boundaries.

a. Number of processes remain constant throughout the run.

b. File view created by each process is not dynamic. File views remain constant through out the run.

1.5 Thesis Organization

The rest of the thesis is organized as follows.

Chapter 2 (Literature survey): This chapter describes the general architecture of parallel I/O systems, with high-level interfaces, the parallel file systems and the underlying storage hardware. It also includes different approaches to optimize the performance of parallel I/O.

Chapter 3 (Proposed System): In this chapter we proposed a new solution to improve parallel I/O performance using interval based I/O.

Chapter 4 (Implementation and Experimental Results): This chapter describes the tools and techniques to implement the proposed system. All the components of interval I/O are described here in details. It shows experimental results.

Chapter 6 (Future work): This chapter concludes the document and provides an overview of future directions for Interval I/O.

Chapter 2

Literature Survey

Today's most advanced large-scale, data-intensive applications run on large clusters consisting of hundreds of thousands of processing cores. In modern high performance systems, I/O architectures have been designed such that the compute nodes and storage servers are separated in groups and connected through high speed networking devices. Data generated by applications must pass through many abstraction layers of I/O stack before reaching the storage devices. These large-scale computing clusters are coupled with parallel file systems such as Lustre, PVFS, and Panasas, which offer massive distributed storage capabilities and are designed to provide scalable access to thousands of clients concurrently.

Software systems, such as MPI (Message Passing Interface) [3], 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.

Using these parallel architecture data-sensitive applications manipulate data sets in the order of terabytes to petabytes and beyond. These high performance computing systems are used to study scientific or natural phenomena. For example, climate modeling, earthquake modeling, and genomic pattern matching.

Even with all of this hardware and software support, in data-intensive applications, as the number of processors grows to a large number, achieving high efficiency for computation

becomes very difficult. In other words, increased I/O overhead limits the incremental gain in efficiency with large number of processors. There are many factors that make the problem so challenging. I/O access patterns exhibited by scientific applications (e.g., non-contiguous I/O), 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.

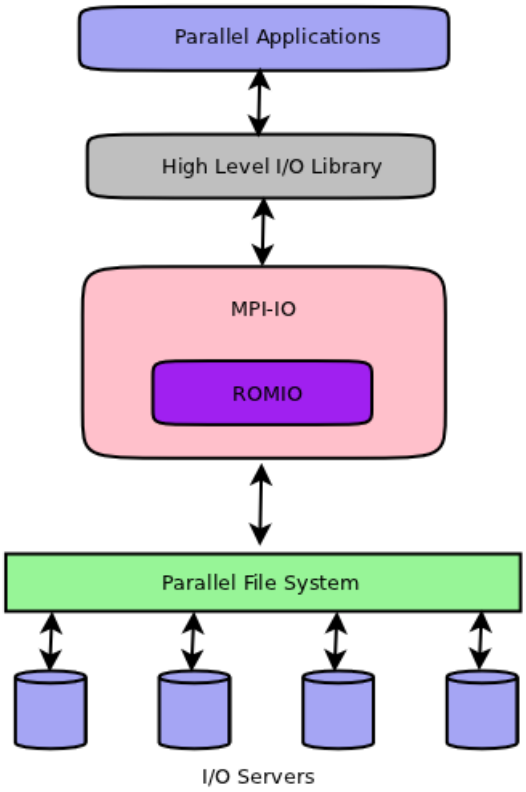

Figure 2.1: Parallel File system and I/O architecture

Figure 2.1 shows the different layers of parallel architecture. Parallel applications use the MPI interface to talk with the parallel file system. In this chapter we discuss in detail about MPI and its uses in parallel computing, as well as present some background knowledge about common optimization techniques.

2.1 MPI-IO

MPI-IO is a standard interface for parallel I/O, defined as a part of the MPI-2 standard. Several implementations of the MPI-IO standard are available; the most commonly used one is ROMIO [24]. ROMIO is included in MPICH, Open MPI and other MPI implementations. It implements the "abstract device interface for parallel I/O " ADIO [22] to create an abstraction layer of the underlying file systems. Any file system that implements the ADIO interface can be used by ROMIO for MPI-IO calls. ROMIO takes standard MPI-IO calls from an application and translates them in the ADIO layer into calls optimized for a specific file system. This flexible architecture is illustrated in figure 2.2. As can be seen, the application interacts with ROMIO via the MPI-IO interface.ROMIO contains implementations of important parallel I/O optimizations such as data sieving and two-phase collective I/O.

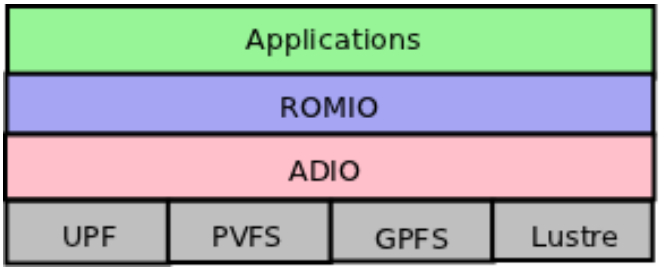

Figure 2.2: The ADIO architecture used by ROMIO

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 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 ability is key element of this research.

2.2 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 [9]. MPI-IO provides sophisticated access operations to data that is read or written noncontiguously in many small pieces. Large number of small

I/O requests is the example of this type of access patterns. These will resultant in incurring the high cost of I/O on each small request. A simple example of such a noncontiguous access pattern is shown in figure 2.3. In this example, a two-dimensional array is read from a shared file by an application consisting of four processes. 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. Process 0 performs one continuous I/O operation on block 0 to 1 and another continuous I/O operation on block 4 to 5. This may be a negligible cost when small amount of 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.

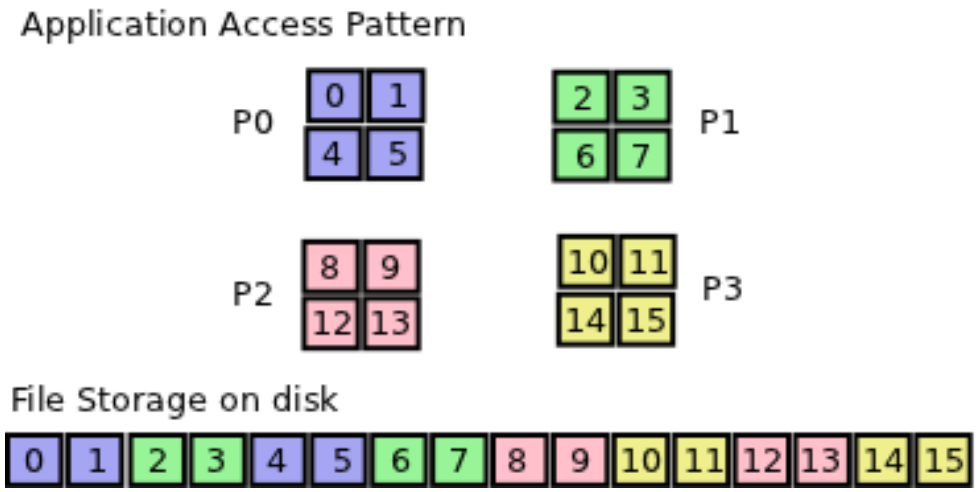

Figure 2.3: Mapping a noncontiguous access pattern onto the disk [16]

Noncontiguous I/O operations show the features and power of the MPI-IO interface. Using file view mechanism, it can specify non contiguous I/O requests in a single operation. The runtime system can use the information available in the file views to optimize accesses to the parallel file system.

2.3 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 the noncontiguous I/O techniques like

collective I/O, data sieving, view I/O.

2.4 Collective I/O

The data of parallel applications is often partitioned among multiple processes according to multidimensional distribution functions. This partitioning results in noncontiguous requests of single processes. The noncontiguous requests of different processes, however, "may together span large contiguous portions of the file", for example when a matrix is partitioned among multiple processes. MPI-IO offers functions to describe collective I/O operations. The underlying ADIO device for a file system can then again either use file system specific functions for collective I/O or use ROMIO optimizations. ROMIO basically uses a technique called "two-phase I/O" [8] for collective I/O. Using two-phase I/O in the first phase multiple processes access file data in large contiguous chunks and in the second phase this data is distributed between the processes "to the desired distribution". This results in few, contiguous I/O accesses. This mechanism requires the exchange of buffers between processes but can often deliver better results than noncontiguous data access of individual processes.

More generalized version of two-phase I/O is known as multiple-phase collective I/O [21]. This technique also uses a single disk I/O phase, but allows the number of redistribution phases to vary. Disk-directed I/O [14] variation of two-phase I/O, 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.

2.4.1 Data Sieving

To reduce the effect of high I/O latency, it is critical to make as few requests to the file system as possible. Data sieving [23] is a technique that enables an implementation to

make a few large, contiguous requests to the file system even if the user's request consists of several small, noncontiguous accesses. For many systems, the cost of performing several small accesses is much greater than the cost of performing a single large file access.

Assume that the user has made a single read request for five noncontiguous pieces of data. Instead of reading each noncontiguous piece separately, the implementation reads a single contiguous chunk of data starting from the first requested byte up to the last requested byte into a temporary buffer in memory. It then extracts the requested portions from the temporary buffer and places them in the user's buffer. The user's buffer happens to be contiguous in this example, but it could well be noncontiguous.

A potential problem with this simple algorithm is its memory requirement. The temporary buffer into which data is first read must be as large as the extent of the user's request, where extent is defined as the total number of bytes between the first and last byte requested (including holes). The extent can potentially be very large much larger than the amount of memory available for the temporary buffer because the holes (unwanted data) between the requested data segments could be very large. The basic algorithm must therefore be modified to make its memory requirement independent of the extent of the user's request.

In interval I/O technique, all the intervals accessed by the processors are stored contiguously on disk, so it eliminates the need of data sieving.

2.4.2 View I/O

[12] describe another approach to parallel I/O based on file views called as "View I/O". In this technique, it combines multiple noncontiguous blocks of an I/O access into a single file system transfer. View I/O requires support from the file system. It rearranges the non-contiguous blocks into linear order before storing them to disk.

View-based I/O uses the MPI-IO file view mechanism, for transferring view information at aggregators at view declaration. Additionally, this approach reduces the cost of scatter/gather operations at application compute nodes. View-based I/O reads can take advantage of collective buffers managed by aggregators, which, unlike in two-phase I/O are cached across collective operations. A collective read following a previous read or write, may find the data already at the aggregator.

View I/O, however, relies on support from the underlying file system to reorganize the

file data. In contrast, proposed system requires no modifications at file system level. In proposed system, it stores the data as per the access pattern of the application, along with additional metadata. This approach also enables the support of application check pointing. We can create a check point and restore it in case of application process restart.

2.4.3 List I/O

List I/O [22] is another technique used in ROMIO to deal with noncontiguous I/O. Like other techniques described in earlier in this section, List I/O also combine the noncontiguous data access to be transferred as a single contiguous chunk. In list I/O, A typical access would be specified with a list of (offset, size) pairs describing a block of the file being accessed.

All these approaches reduce the large number of noncontiguous accesses to small number of contiguous accesses. But all these techniques fail to because of the access pattern of the applications. All these techniques, stores the file in linear order. Overhead of conversion between two different models are high. Proposed method directly addresses the problem by removing this requirement, thus eliminating the problem that List I/O attempt to solve.

2.5 Caching

Caches have always been, and will continue to be, an important part of parallel file systems and computing systems. This is particularly true in environments where accesses to the working data set results in completion delays. Intermediate level caches feed data to the requesting source at a much faster rate then their larger, but slower, hierarchical counterparts. Often times moving from one level of memory to the next results in at least a magnitude of access time and throughput lose. There has been some research directed toward using collective caching to improve parallel I/O performance [8]. 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.

2.5.1 Delegate Caching

In [19], They propose an additional I/O layer termed as I/O Delegate Cache System (IODC) for MPI applications. This allows I/O performance to scale with processing power of massively parallel large-scale systems. Lock contention at disks occurs when two processes compete with each other to access the same file region. With the increase in number of processors, I/O request load at data disks increases. Also, lock acquiring becomes more difficult task, with more number of competitors trying to acquire lock. This also increases the number of processors waiting to receive I/O service at any given time.

As several processes contend to write to the same file, the file system must lock it and contending processes must be queued. By reducing the number of clients, there are fewer file locks. Furthermore, they implement a caching mechanism that reduces the number of I/O operations required by coupling smaller requests into a few larger ones.

2.6 ADIOS

The Adaptable I/O System (ADIOS) [22] developed at Oak Ridge National Laboratory provides a simple, flexible way for scientists to manage their data in the code that may need to be written, read, or processed during simulation runs. ADIOS utilizes an external XML file to describe user data, e.g., data types, sizes and I/O operations. The primary goal of this I/O system is to offer a level of adaptability such that the scientist can change the I/O transport simply by changing a single attribute in the XML file. 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.

This framework allows the applications to use different I/O patterns like MPI-IO, POXIS I/O, data taps etc. It uses these I/O pattern to grouping the data from a single applications. ADIOS is a computerization of IO layer. It has been designed to be easy to program, and to be fast and portable. By allowing users the flexibility to switch between different I/O implementations.

The design of ADIOS is somewhat similar to the proposed system. Here are some points where it differs:

- ADIO 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. While in our proposed method, interval file is the standard storage format.

- With the help of locking proposed system supports atomic mode, while it is not supported in ADIOS.

- With no modifications to the source code, Interval based I/O system directly supports MPI-IO.

2.7 Summary

The works above have in common the agreement that some intermediary layer must be added between the computational nodes and the I/O servers for continuing scalability and minimum or no code changes in application. They agree that contemporary parallel file systems will be a bottleneck as the number of compute cores continues to scale.

In data sieving technique, the performance is depends on the number of non contiguous file blocks being accessed, and the size of the gaps between required file blocks. If the size of the gap between required file blocks are more than the sizes of the required file blocks than this can degrade the overall performance of the system. Buffer size should be also tuned carefully. If the buffer size is very large, less memory is available for the applications.

Working of view IO is same as collective data technique. Multiple noncontiguous blocks of an I/O access to be combined into a single file system transfer. But all the operations are performed at the file system level. View I/O requires support from the file system. File system with such a support rearranges the non-contiguous blocks into a linear ordering before storing them to disk.

Delegate caching uses extra delegate nodes between file system and computation nodes. This will increase the hardware cost. By reducing the number of clients, there are fewer file locks but this will lead to serialize system.

Interval based I/O system is one of the most promising approach to solve the problem of parallelism. In this technique uses the MPI file views to create intervals, it reduces

serialization and contention. False sharing is completely removed in this approach.

Chapter 3

The Proposed System

In this chapter we discuss the design of proposed I/O optimization system. The system is an additional layer for MPI applications, which is integrated into an MPI-IO implementation ROMIO. This layer contains 3 sub layers. This allows I/O performance to scale with processing power of massively parallel large-scale systems.

3.1 Interval I/O System

First layer is interval I/O system [16]. The system consists of following 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.

3.1.1 Interval Integration Interface

Interval integration interface is used to translate the MPI-IO calls. It converts the MPI-IO calls and extracts the useful information to create the interval sets. To create an interval, it uses the file view information from MPI-IO calls.

3.1.2 Interval Set Creation

Interval integration interface provide the view information from MPI-IO call to the interval set creation process. It takes place once all the views have been set for each process. ROMIO represents the views as a list of the (offset, size) pair each of which describes a contiguous

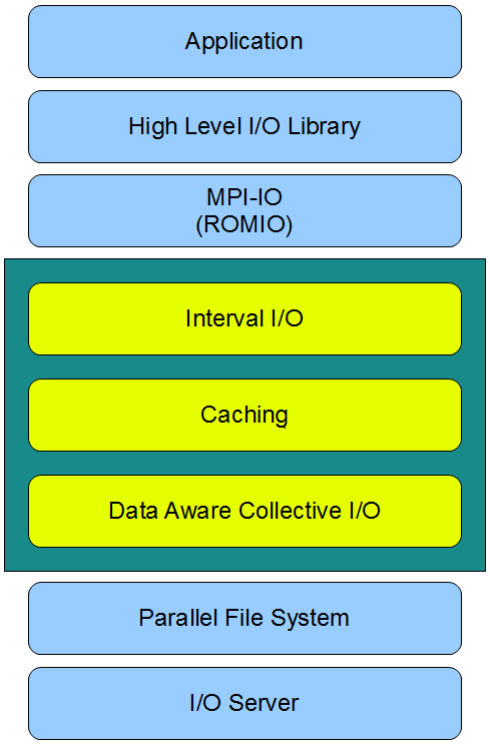

Figure 3.1: Additional proposed layer

file region visible to that process.

The first step of interval set creation for each process is to share its view with all the other process. Once this has been done, the interval creation process is executed. It finds the intersection amongst all the views and creates intervals. This process is shown in figure 3.2. As result of this function, we get the interval sets which will be used during execution of all the processes and remain constants during execution. After these processes we will get two type of intervals i) private ii) shared. Private intervals are used by only single process while shared intervals, are used by multiple processes. We can add certain attributes in these intervals like lock status, offset, processes, its type, modification time. An interval with all these attributes is known as object. In figure 3.3, shared blocks are shown with dark shades.

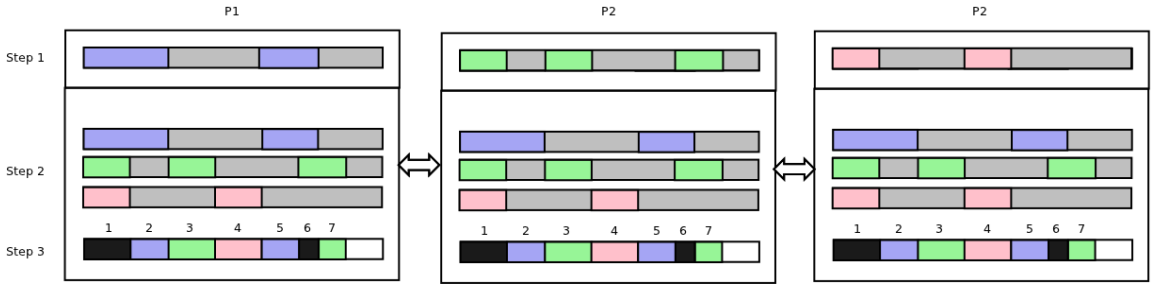

Figure 3.2: Intervals exchange [16]

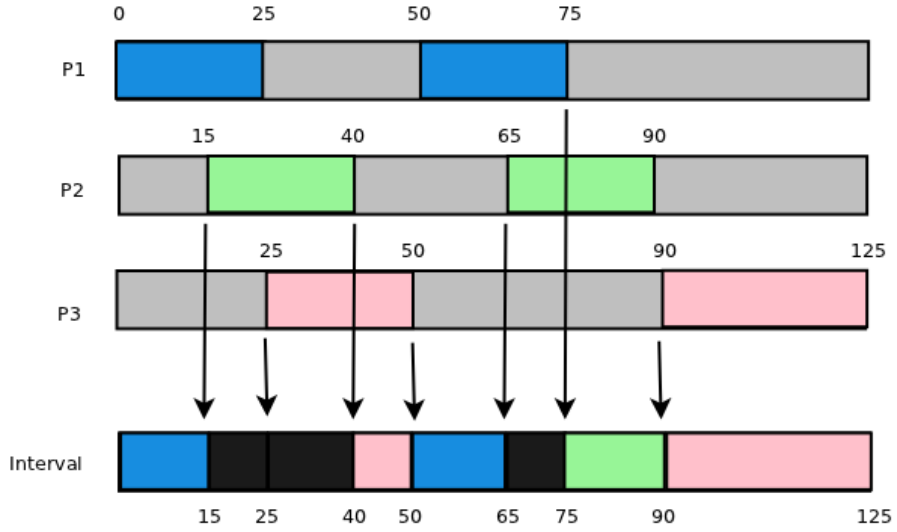

Figure 3.3: Intervals using MPI views [16]

3.2 Interval-Based Caching

In this approach global software cache is used. It uses to cache the intervals. Global cache coherence is maintained by keeping only a single valid copy of each interval in global cache. This interval based caching provides many benefits like prefetching and write behind. It also simplifies the global locking.

All operations like read and write are performed on cache only. Actual data transfer occurs during the synchronization phase. During synchronization phase, an I/O server opens a file handler, reads data from the file and writes the data into the cache or wise versa. To find the latest copy of the interval, modification time is used. To perform atomic

operations, It checks the lock status of the interval before performing any action.

3.3 Lock Manager

Lock manager is a very important part of the Interval I/O System. In typical parallel file system, striped file blocks are the fundamental unit of locking, while in interval based I/O, interval is the fundamental unit of locking. A centralized lock manager is used to manage. All the processes have direct access to the lock manager. Lock manager works with cache manager to perform atomic operations. As discussed earlier, locks are required only for shared intervals; it reduces the overhead of locking for private intervals.

3.4 Data Layout Awareness Strategy

Data layout describes how data is distributed among multiple file servers. It is a crucial factor that determines the data access latency and the I/O subsystem performance for parallel computing systems.

As discussed earlier, typical parallel file system uses simple striping data distribution function, that data is distributed using a fixed block size in a round robin manner among available I/O servers. Using these applications access information and data layout information on file servers, we can optimize accesses at various levels including application level, middleware level, and file system level.

In proposed system, we use the information provided by the file views. Each process defines its view at the beginning. Using this information, the interval objects are rearranged as per the processes view. All the Interval objects, accessed by single process, are stored on single I/O servers. It provides continuous access to each process.

3.5 Advantages of Using Intervals

Using interval I/O, we get following advantages:

- It eliminates false sharing of files between processes.
- Using interval we get the performance improvement for noncontiguous access patterns.

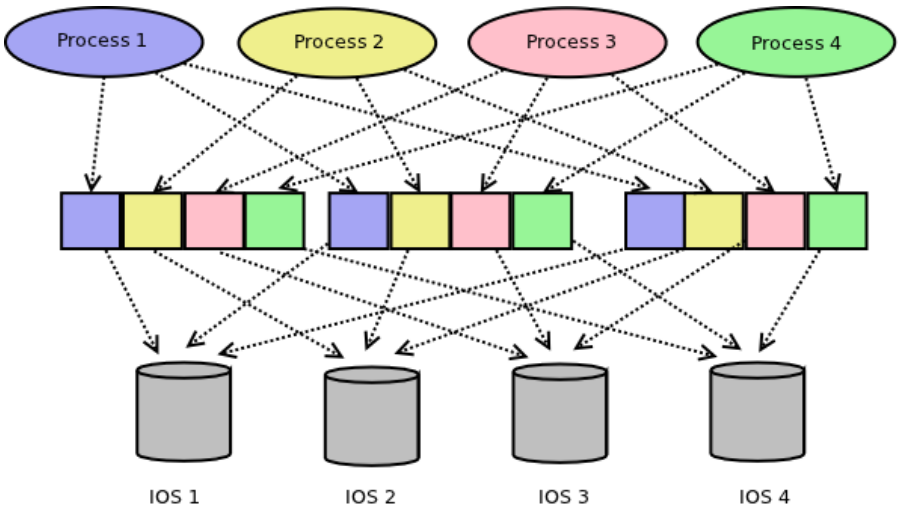

Figure 3.4: Collective I/O without Layout Awareness

- Used data layout awareness techniques, It provides significant performance improvement in read operations.
- It reduces the lock overhead by detecting private and shared intervals.
- " By modifying applications, we can add check point support.

3.5.1 Eliminating False Sharing

When non-conflicting operations are blocked because of locked status of the resources is known ad false sharing. A parallel I/O implementation that uses extent locking for non-contiguous operations may lead to false sharing. For example, consider the access pattern shown in Figure 3.6

P0 and P1 are performing nonconflicting operations. This is shown by the shaded area in figure. But here the border area is overlapped so the accesses would be serialized although it is not required. In the proposed system, it creates three intervals, so three locks would be used. Two locks for process P0 and one lock for process P1. All the locks could be acquired at a same time, allowing both the processes to access file simultaneously.

Figure 3.7 shows another interesting point. False sharing also occurs using block based caching. Figure 3.7(a) shows bock base caching. Again P0 and P1 perform nonconflicting

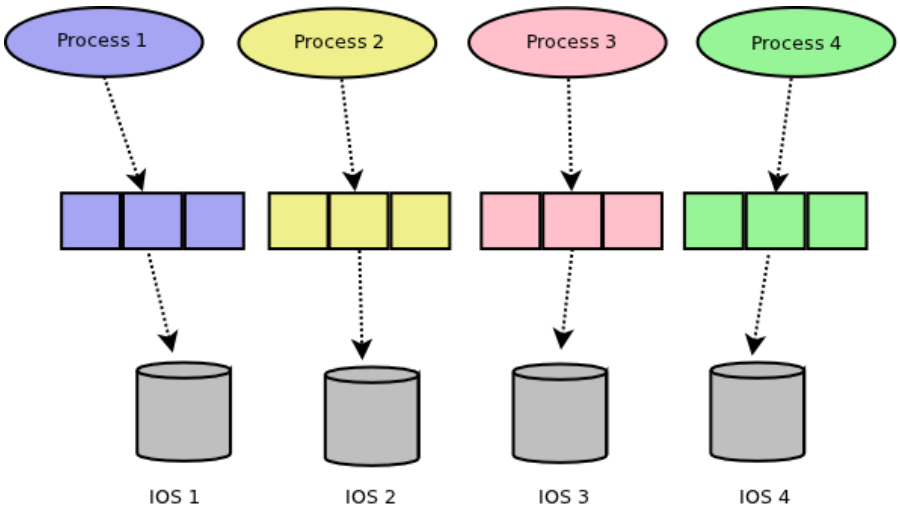

Figure 3.5: Collective I/O with Layout Awareness

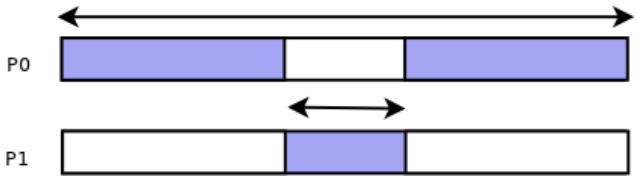

Figure 3.6: False sharing resulting from extent locking [16]

access. P0 uses B0 and B1 blocks and P1 uses B1 and B2 blocks. Here false sharing occurs for block B1. If we use interval based approach, It creates two intervals(I0 and I1). It requires two locks and all the locks could be acquired simultaneously.

3.5.2 Performance Improvement For Noncontiguous Access Patterns

The performance of parallel I/O systems depends on the access pattern of the application. Most of the scientific applications performs noncontiguous I/O patterns, and it is very difficult to achieve high throughput on legacy file systems. For example, consider the access pattern shown in figure 3.8(a). Typical parallel I/O system stores the data on disk as shown in (b). Using interval I/O rearranges the intervals as shown in (c).

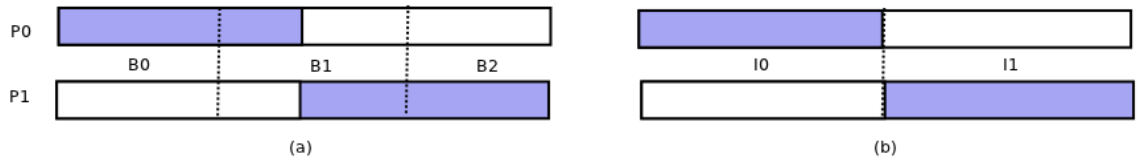

Figure 3.7: False sharing resulting from block-based caching (a) Block-based caching (b) Interval caching [16]

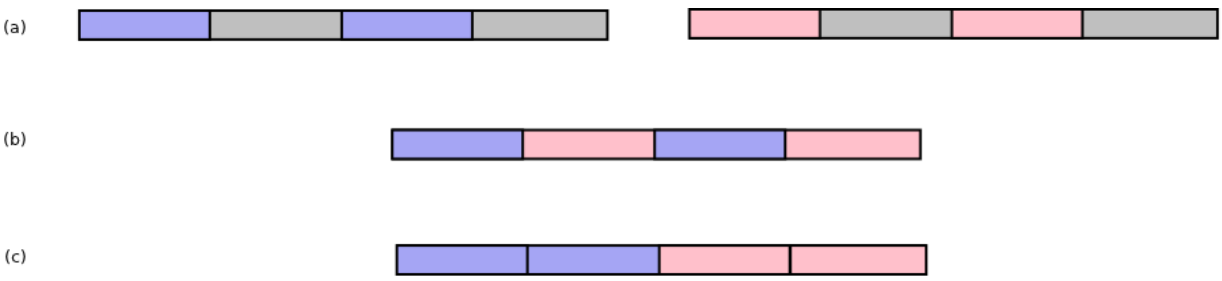

Figure 3.8: False sharing resulting from block-based caching (a) Block-based caching (b) Interval caching [16]

3.5.3 Data Layout Awareness Technique

All the relative intervals are stored in single I/O servers. It reduces the communication overhead between multiple I/O servers. All the intervals, required by one process are available to single servers. Single continuous access is required to perform desire operations.

3.5.4 Detection of Private Intervals

Interval I/O can detect the private and shared intervals. Private intervals are accessed by single process while shared intervals are accessed by multiple processes. Only shared interval required locking. This can have a significant impact on the level of concurrency depending upon the I/O access patterns of the application.

Consider the I/O pattern shown in figure 3.9. All interval created using these view information, would be private. So no locking would be required.

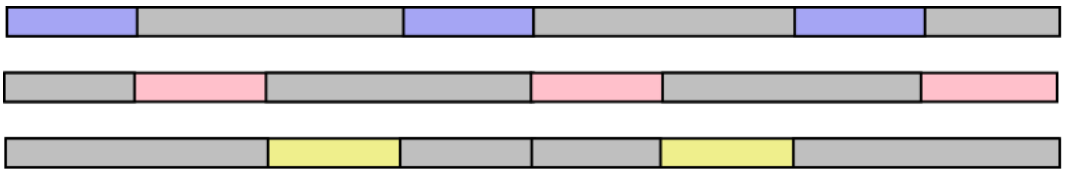

Figure 3.9: An access pattern containing only private intervals [16]

3.5.5 Check point support

In proposed solution, Objects are store with metadata. If we modify the application code, so that it will provide the seek offset to the MPI call. This offset is stored with object metadata. Now when the application process is restarted, it will read this offset and continue its execution from that offset.

Chapter 4

Implementation

To measure the performance of proposed system, We have developed a framework. This framework is based on client server architecture. It has been developed in PERL. Clients act as a application processes. They bind themselves to predefined MPI views. They generate random data and send a request to server to write/read the data. Servers act as a IO servers. On receiving signals from clients, IO servers take a lock if require, and perform appropriate actions. In this research we are working on new approach which is dealing with file system, so we are ignoring the networking part. We assume that the effect of networks remain constant throughout the simulation.

For caching purpose, we use memcached [2]. It provides global caching to support the framework. As we removed the networking part, Signaling is used to communicate between application and IO servers.

All the parameters are configurable. Path of configuration file is /src/lib/SimConfig.pm. Generally used parameters are described in table I.

4.1 Client application

Predefined view information is stored in src/data/view_data file. Each line of this file represents view information of processes. Format of this view is offset:value. Client application reads view information from that file, creates interval objects. After that application forks MAX_PROCESSES processes. Each child process bind itself to interval objects. Following data structure is created to store interval objects.

STRIP_SIZE	Stirp size, It requires for typical parallel file systems
MAX_IO_SERVERS	Maximum number of IO servers
MAX_PROCESSES	Maximum application processes
MAX_VIEWS_PER_PROCESS	Maximum number of view per application process
MAX_VIEW_SIZE	Maximum view size
MAX_FILE_SIZE	Maximum file size
FILE_SIZE_UNIT	File size unit(MB/KB)

Table I: Configuration option

lock_status	Bool	Locking status of perticular object
offset	Long Int	Offset position of object
processes	Array	Array of associated processes' pid
type	Char	p- private, s-shared
modified_time	Long int	Last modified time
data		Data

Table II: Object metadata

4.1.1 Write operation

Child process generates random data, and fill the data field of the related interval object. This object is now cached. Once the process is done with the data part, it sends the signal to attached IO server.

4.1.2 Read operation

Child process reads the object data from cache. If there is cache miss, it will send the signal to attached IO server.

4.2 IO Servers

Server application forks MAX_IO_SERVERS processes. Each process act as a independent IO servers. After the initialization, each IO server waits for signals from the application process.

4.2.1 Write operation

On receiving write signal, IO server reads object data from cache, and writes the objects into the storage device.

4.2.2 Read operation

On receiving read signal, IO server reads object data from storage device, and writes the objects into the cache.

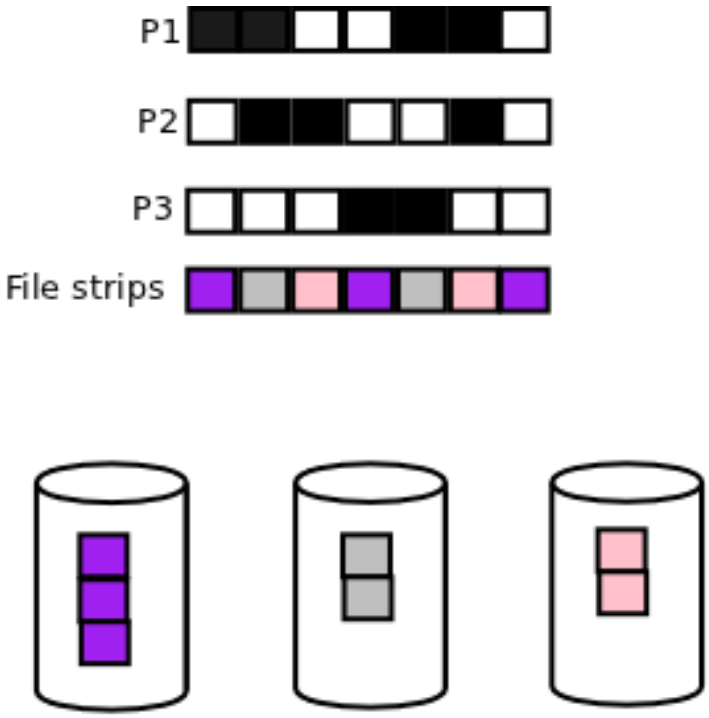

Figure 4.1: File strips over IO servers

Figure 4.1 shows the distribution of file strips over IO servers. It uses round robin algorithm to store the file strips to IO servers.

Figure 4.2 shows the object distribution of interval based IO system. In this system metadata server is used to find the mapping between objects and IO servers. Here the objects are distributed same as typical parallel file system. But all the actions are performed on cache.

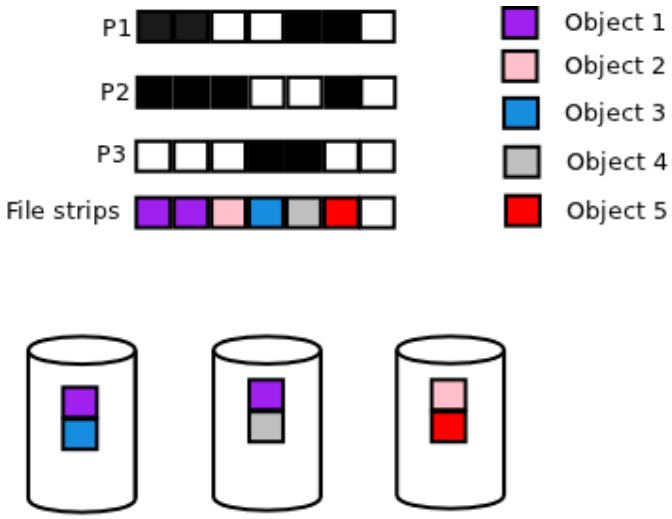

Figure 4.2: Object distribution over IO servers using Interval

Figure 4.3 shows the object distribution of the modified interval IO. Shared objects are stored multiple times on multiple servers. It provides continues access to the data on single IO servers. If P1 wants to read its view data, All the data is available on first IO server.

4.3 Experiment Result

In this section we present representative results from the performance testing that has been performed. These experiment has beed perfomed on three differnt models, 1) Typical strip based 2) Interval based I/O suggested in [16] 3) Proposed system based on interval IO. The experiments described were performed using 2GHz desktop class machine, with 4 GB of DDR 2 memory and 120 GB hard drive. Cache setting for this testing was 1 GB global cache with FIFO. This system was running 32 bit linux (ubuntu 11.10). Underlying filesystem used was ext3.

We designed our experiments to study the effectiveness of using our proposed Interval I/O system to perform read/write operations. We developed a tool, that generates the random test cases. These test cases contains write and read operations for randomly selected objects/stripes. This tool selects objects /strips only from valid MPI views for each process.

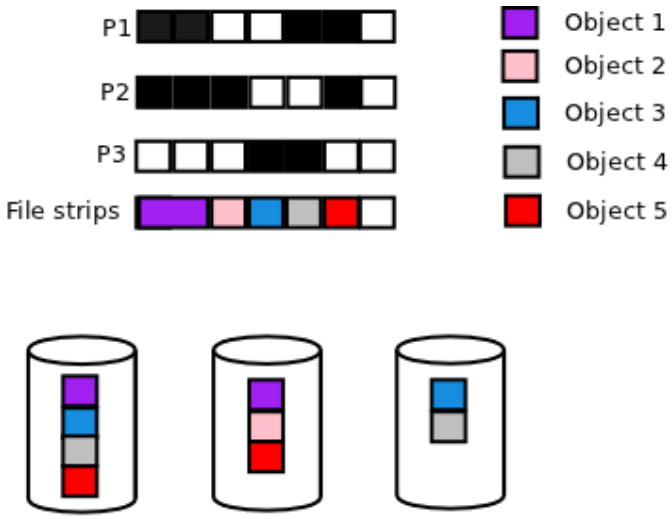

Figure 4.3: Object distribution over IO servers using modified Interval

Processes	Strip based	Interval based	Modified interval based
4	3.9345	5.0984	6.6588
6	3.9899	6.7890	10.1342
8	4.0186	8.2875	13.1865
10	4.3120	10.1102	14.6540
12	5.6825	12.1860	15.2418

Table III: Write performance

Figure 4.4 and 4.5 shows the result. We can see that proposed system perform very well. global caching and reduced lock contention were the key player of the driven performance. Interval IO and proposed system both perform all their operations on cache. Only during synchronization phase, data will be read from file or write to file. It also shows the effect of Scalable I/O Problem. As per the expectation, It shows high performance improvement in read operations. Write operations also improved because of global caching and reduced contention. Table III and IV shows the result data.

Figure 4.6 shows the result of the interval creation time and read/write time for 100MB data. It gives the idea of whole operations. Interval creation time is too small. Table V show the result data.

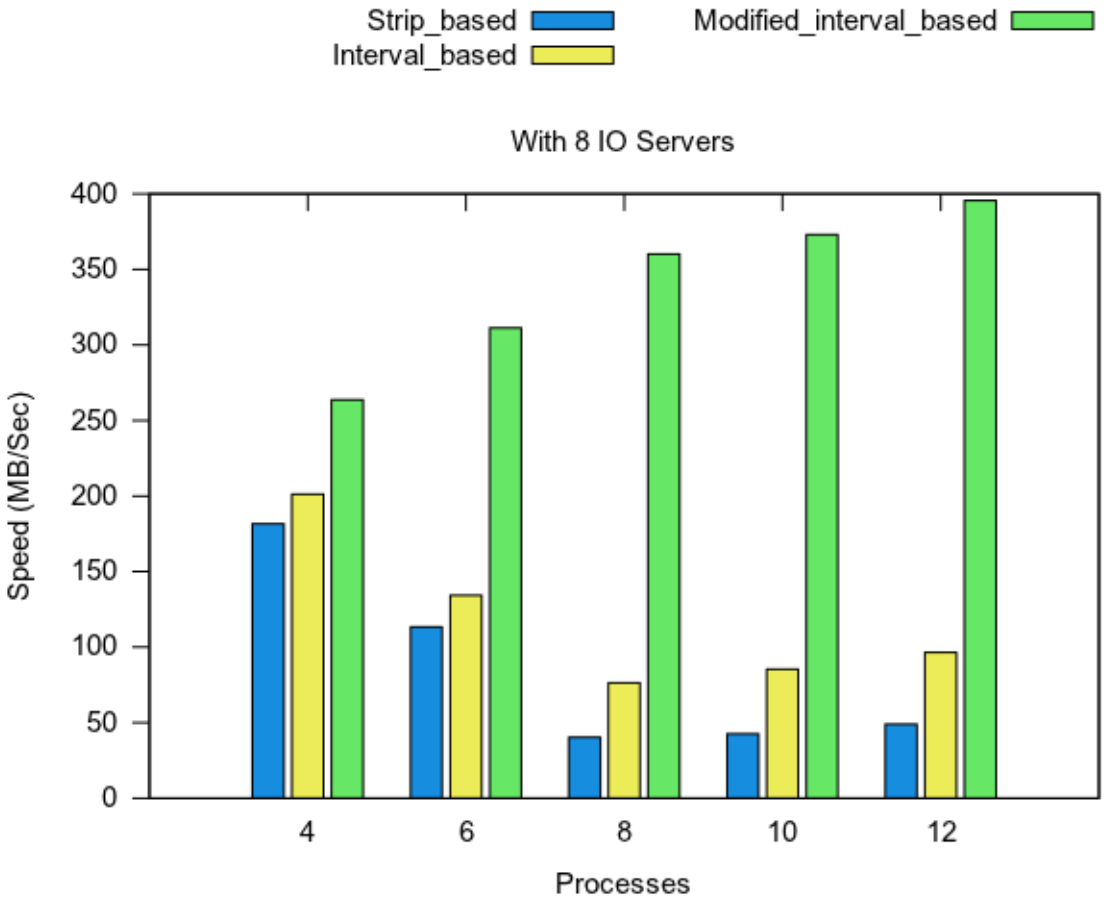

Figure 4.4: Read performance

4.4 Discussion

Result shows that proposed system based on interval I/O provides a promising new approach to parallel I/O. MPI view information and using it to decide the intervals, we have shown that I/O performance can be significantly improved.

In the developed model, we assumed the overall effect of the computer network remains constant. If we implement this model in actual parallel system, computer network plays a very important role in that case. Because of the networking overhead, overall performance may decrease up to some extends. Another important point is the distribution of MPI views. If the arrangement of views are dense, then the resultant intervals are small in size and most of them are shared. This situation is very critical, It is same as the strip

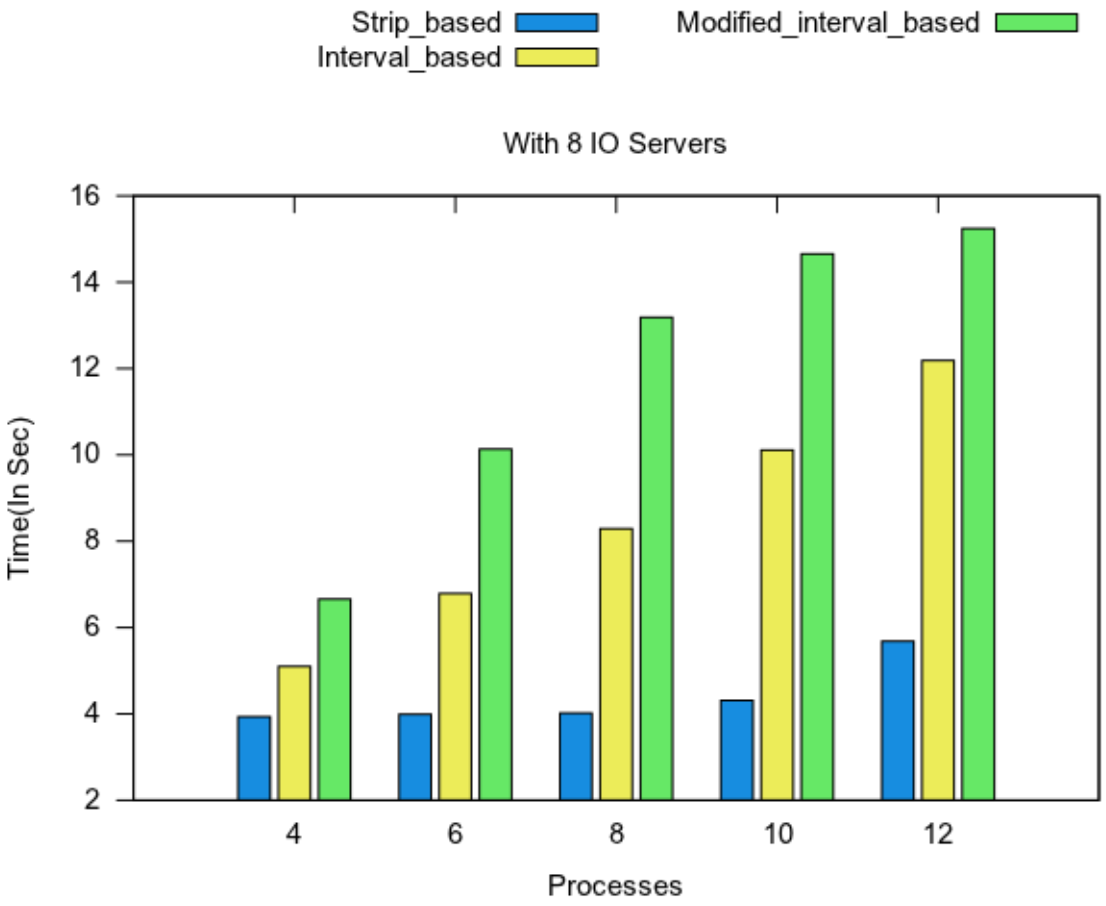

Figure 4.5: write performance

based parallel system. So if the intervals are small and shared, Overall performance of this proposed system will be decreased.

This model stores the shared intervals multiple times. It uses more disk space than any other existing parallel IO systems. But writing extra data may not be the concern today, because nowadays the secondary memory is not an issue. Price of secondary storage is cheap and computers with terabytes of secondary storage exists. The main issue is the amount duplicate data. If the total size of shared objects are more, It affect the performance of overall system. So it is advisable to use this system where the shared objects are less. Metadata is also stored with object. This will create overhead. But the size of medadata described in table II is around 300 bytes, so we can ignore this overhead.

Processes	Strip based	Interval based	Modified interval based
4	181.6713	201.1478	263.6452
6	113.4563	134.2356	311.3433
8	40.3980	76.3695	360.2251
10	42.6674	85.3344	373.0034
12	48.9388	96.4390	395.5798

Table IV: Read performance

Processes	Write	Read	Interval creation
4	12.7318	0.4736	0.0136
8	9.7421	0.3296	0.0143
12	7.3771	0.2589	0.0108
16	6.2580	0.1739	0.0168
20	5.5127	0.1863	0.0158

Table V: Performance for 100MB data

One more feature checkpoint capability is yet to test. To test this feature application level modifications are required.

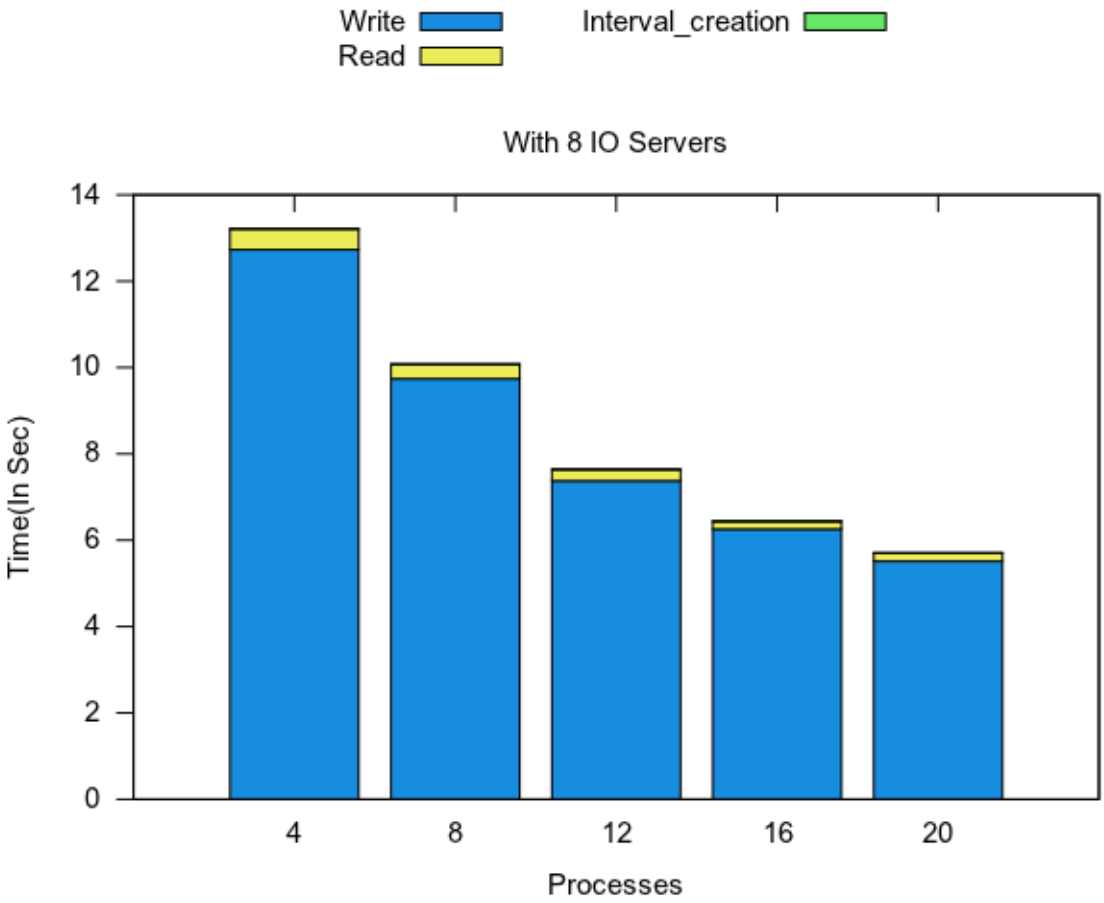

Figure 4.6: Performance for 100MB data

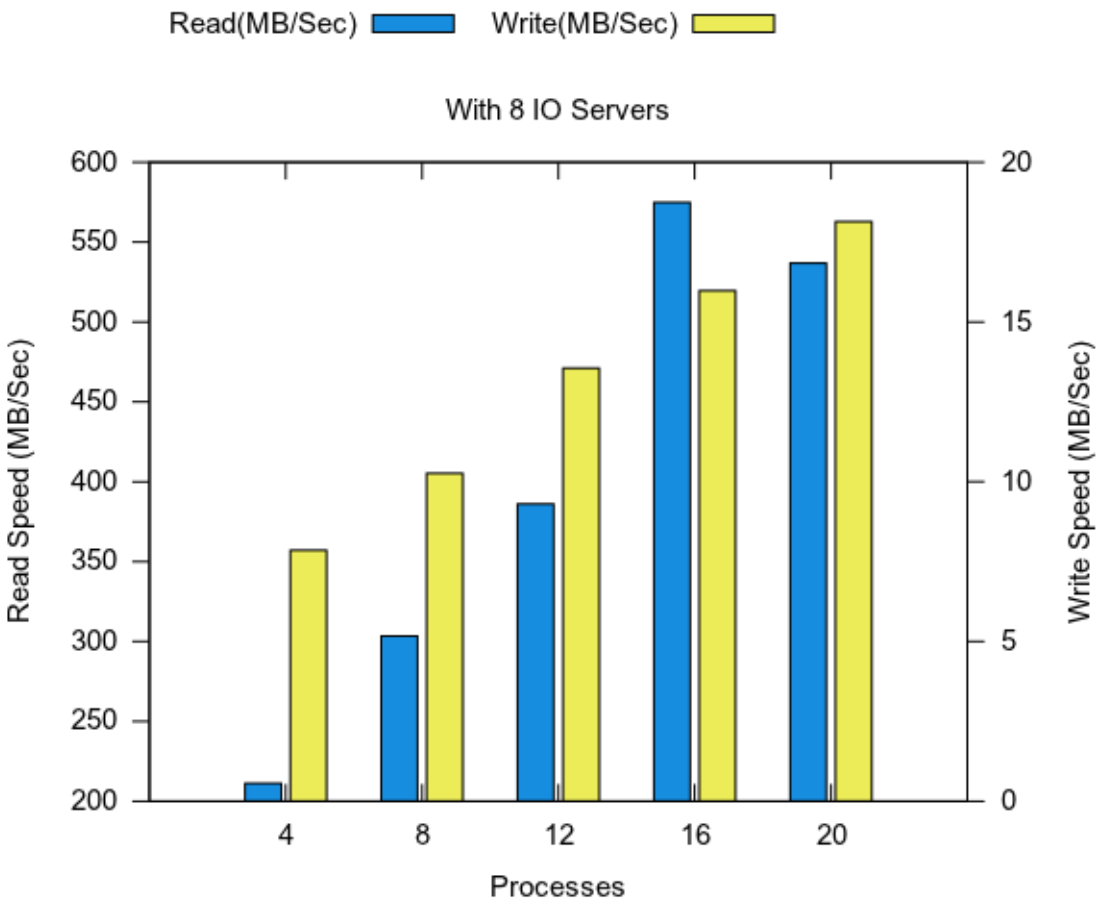

Figure 4.7: Read/Write Performance

Chapter 5

Conclusion and Future Scope

This thesis presented a new interval-based approach to parallel I/O. In this chapter we summarize the work of this thesis and mention potential advances of the development in the future work section.

5.1 Conclusion

In this thesis we introduced a new layer in parallel I/O architecture. This layer contains interval-based I/O approach to deal with noncontiguous access. Caching layer to boost the performance and data layout aware distribution, to distribute relative objects on single I/O server. We have developed a prototype of this additional layer. Result shows the significant improvement. Distribute the relative objects on I/O servers, provides continuous, single I/O operation. This provides boost up in read operations. It also shows the improvement in write operations.

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. Data duplication is one of the major drawbacks of this system. In the proposed system, it stores shared objects multiple times. As a result it will take more time to write duplicate data. If the view contains more shares intervals, the overall performance of this proposed system may decreases.

5.2 Future Scope

First of all this proposed system would be implemented for any state of the art parallel file system like Luster or PVFS. This would be the ideal platform to extend the testing and performance measurement. Result of typical benchmarks like FLASH-IO, MPI-Tile-IO on such platform would be helpful to understand the overall performance of new system.

Caching plays an important role in overall performance of the system. Another area of interest involves analysis of different caching strategies and algorithms. It would be nice to have comparisons of different caching strategies for different type of intervals. In testing, we used global cache with FIFO behavior.

Data-duplication is the major drawback of this system. It would be good to have some another technique that will give the same performance with less or no data duplication.

We assume that file view remain constant throughout the execution of the experiment. In some real life applications, this may not be the true. Runtime interval creation and translation is the natural choice for these kinds of the applications.

As describe earlier, proposed system can be extends to support checkpoint capability. Application level modifications are required to fulfill this requirement.

References

[1] Lustre. `http://wiki.lustre.org/index.php/Main_Page`.

[2] Memcached. `http://memcached.org/`.

[3] Mpi-2.2. `http://www.mpi-forum.org/docs/docs.html`.

[4] Panasas. `http://www.panasas.com/`.

[5] Parallel computing. `http://en.wikipedia.org/wiki/Parallel_computing`.

[6] Pvfs. `http://www.pvfs.org/`.

[7] *Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation.*

[8] Kenin Coloma, Alok Choudhary, and Wei keng Liao. Scalable high-level caching for parallel i/o. In *In Proceedings of the International Parallel and Distributed Processing Symposium*, pages 23–32, 2004.

[9] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Input/output characteristics of scalable parallel applications. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, New York, NY, USA, 1995. ACM.

[10] W. Gropp and E. Lusk. *"Beowulf Cluster Computing with Linux"*. Volume 2. MIT Press, 2003.

[11] W. Gropp, E. Lusk, and R. Thakur. Using mpi-2 advanced features of the message-passing interface, 1999.

[12] Florin Isaila and Walter F. Tichy. View I/O: Improving the performance of Non-Contiguous I/O. *Cluster Computing, IEEE International Conference on*, 2003.

[13] D. Kotz and R. Jain. I/o in parallel and distributed systems, 1999.

[14] David Kotz. Disk-directed i/o for mimd multiprocessors. *ACM Trans. Comput. Syst.*, 15:41–74, February 1997.

[15] Rob Latham, Rob Ross, and Rajeev Thakur. The impact of file systems on MPI-IO scalability. pages 87–96. 2004.

[16] Jeremy Logan. *Improving Parallel I/O Performance Using Interval I/O*. PhD thesis, The University of Maine, April 2010.

[17] Jeremy Logan and Phillip Dickens. Improving i/o performance through the dynamic remapping of object sets.

[18] J. May. *Parallel I/O for High Performance Computing.* Academic Press, 2001.

[19] Arifa Nisar, Wei-keng Liao, and Alok Choudhary. Scaling parallel i/o performance through i/o delegate and caching system. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Piscataway, NJ, USA, 2008. IEEE Press.

[20] R. Ross, R. Thakur, and A. Choudhary. Achievements and challenges for i/o in computational science. *Journal of Physics: Conference Series*, 16, 2005.

[21] David E. Singh, Florin Isaila, Alejandro Calderon, Felix Garcia, and Jesus Carretero. Multiple-phase collective i/o technique for improving data access locality. In *Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 534–542, Washington, DC, USA, 2007. IEEE Computer Society.

[22] R. Thakur, W. Gropp, and E. Lusk. An abstract-device interface for implementing portable parallel-i/o interfaces. In *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, FRONTIERS '96, Washington, DC, USA, 1996. IEEE Computer Society.

[23] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective i/o in romio. In *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, FRONTIERS '99, Washington, DC, USA, 1999. IEEE Computer Society.

[24] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing mpi-io portably and with high performance. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 23–32, New York, NY, USA, 1999. ACM.

[25] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. Dynamic metadata management for petabyte-scale file systems, 2004.