NORTHWESTERN UNIVERSITY

Optimizing File System Techniques for Large-Scale Scientific Applications

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Electrical Engineering and Computer Science

By

Avery Ching

EVANSTON, ILLINOIS

December 2007

ABSTRACT

Optimizing File System Techniques for Large-Scale Scientific Applications

Avery Ching

High-performance scientific computing in a modern age uses parallel techniques at a scale of hundreds of thousands of processors. These large-scale applications have I/O system workloads that are primarily driven by small, sparse I/O operations. While parallel file systems have provided application developers with scalable peak I/O bandwidth for large, contiguous I/O operations, the noncontiguous I/O access patterns common to scientific applications have remained a serious performance problem. Our work in this area has contributed several solutions toward improving the gap in I/O performance and other system components through new list I/O and datatype I/O methods. Detailed studies on basic I/O characteristics and application-level I/O strategies have led to performance-oriented suggestions for application programmers on how to best use these new I/O methods. In this paper, we also discuss our optimized DLM and versioning approaches for implementing atomic noncontiguous I/O operations that are the building blocks for handling challenging problems in redundant storage, consistent distributed data layouts for parallel processing, and other producer-consumer problems.

Acknowledgments

There are many people who helped me immensely through the doctoral process. My advisor, Alok Choudhary, played a very large role in my academic development. When I was contemplating my future near the end of my undergraduate career, he helped convince me to pursue a doctoral degree and provided the time and financial support necessary for me to work toward this endeavor. Professor Choudhary inspired me intellectually and also kept my spirits up during the process. Dr. Rob Ross, my mentor for many years at Argonne National Laboratory, helped me a lot with the technical details of our research collaboration and has been a co-author on most of my doctoral work. Dr. Wei-keng Liao helped guide my research, especially during my early doctoral years. I would also like to thank my other committee members including Professor Gokhan Memik and Dr. Rajeev Thakur for supporting me during my thesis defense. Additionally, I have been fortunate to have worked with many other research collaborators at various institutions including Lee Ward, Dr. Neil Pundit, Professor Wu-chun Feng, Dr. Chung-hsing Hsu, Dr. William Gropp, Seung Woo Son, Professor Mahmut Kandemir, Dr. Kenin Coloma, Peter Aarestad, Professor George Thiruvathukal, Dr. Jackie Chen, Dr. Ramanan Sankaran, Dr. Scott Klasky, Heshan Lin, and Professor Xiaosong Ma.

My parents, Randall and Trudy Ching, are responsible for the love and support that gave me a positive attitude in life. My brother, Bryce, and sister, Tiffany, kept me encouraged so we could all be "doctors." The wonderful peers in our research group including Dr. Joseph

Zambreno, Dr. Jayaprakash Pisharith, Dr. Ying Liu, Dr. Steve Chiu, Dr. Jianwei Li, Kenin Coloma, Dan Honbo, Ramanathan Narayanan, Sanchit Mishra, Abhishek Das, and Arifa Nisar have made "work," a lot more fun and my friends outside of school provided me with much needed joy and relaxation through sports, eating out, and other activities. I thank God for the strength to persevere and for providing me with the resources I needed.

This work was supported in part by Sandia National Laboratories and DOE under Contract 28264, DOE's SciDAC program (Scientific Data Management Center) award number DE-FC02-01ER25485, the NSF/DARPA ST-HEC program under grant CCF-0444405, and the DOE HPCSF program.

Table of Contents

ABSTRACT 3

Acknowledgments 4

List of Tables 9

List of Figures 10

Chapter 1. Introduction 18

Chapter 2. High-Performance I/O Software Stack 23

2.1. Definitions and Noncontiguous I/O 23

2.2. Portable File Formats and Data Libraries 24

2.3. MPI-IO and ROMIO 32

2.4. Parallel File Systems 45

Chapter 3. List I/O 55

3.1. Interface 55

3.2. PVFS1 Implementation 57

3.3. ROMIO MPI-IO Implementation 59

3.4. Performance Evaluation 62

3.5. Summary 77

Chapter 4. Datatype I/O 79

4.1. Interface 79

4.2. Datatype I/O Implementation in PVFS1 and ROMIO 80

4.3. Noncontiguous I/O Methods Comparison 84

4.4. Performance Evaluation 87

4.5. Summary 92

Chapter 5. Performance Analysis of Access Pattern Characteristics 94

5.1. I/O Characteristics Discussion 94

5.2. PVFS2 and ROMIO Implementation 96

5.3. HPIO Benchmark 97

5.4. HPIO Results 98

5.5. I/O Guidelines 105

Chapter 6. Exploring I/O Strategies for Parallel Sequence-Search Tools 110

6.1. I/O Algorithms in Parallel Sequence-Search Tools 113

6.2. S3aSim 117

6.3. Performance Evaluation 123

6.4. Summary 137

Chapter 7. Noncontiguous Locking Techniques for Parallel File Systems 139

7.1. History & Our DLM Approach 142

7.2. New Locking Methods 144

7.3. Hybrid Lock Protocols 147

7.4. DLM Implementation 149

7.5. Performance Evaluation 155

7.6. Summary 168

Chapter 8. The Versioning Parallel File System 169

8.1. Fault-Tolerance, Strict Consistency, and Noncontiguous I/O 169

8.2. Atomicity For Parallel I/O 171

8.3. VPFS Protocol 172

8.4. VPFS Implementation 176

8.5. Performance Evaluation 178

8.6. VPFS Discussion 181

Chapter 9. Conclusion and Future Work 184

References 185

List of Tables

2.1 NetCDF library functions. 27

2.2 HDF5 interfaces. 31

2.3 Commonly used MPI datatype constructor functions. Internal offsets can be described in terms of the base datatype or in bytes. 35

2.4 A comparison of HPC production file systems. 46

2.5 A comparison of HPC research file systems. 50

3.1 I/O characteristics of the tile reader benchmark. 69

3.2 I/O characteristics of the ROMIO three-dimensional block test. 72

3.3 I/O characteristics of the FLASH I/O simulation (n is the # of clients). 74

4.1 I/O characteristics comparison. 84

List of Figures

2.1 (a) Abstract I/O software stack for scientific computing. (b) Current components of the commonly used I/O software stack. 23

2.2 Various I/O access cases. (a) refers to contiguous in memory and file (c-c). (b) refers to noncontiguous in memory and contiguous in file (nc-c). (c) refers to contiguous in memory and noncontiguous in file. (d) refers to noncontiguous in memory and file (nc-nc). 24

2.3 Using data libraries in parallel applications: (a) using a serial API to access single files through a single process; (b) using a serial API to access multiple files concurrently and independently; (c) using a parallel API to access single files cooperatively or collectively. 26

2.4 Design of PnetCDF on a parallel I/O architecture. PnetCDF runs as a library between the user application and file system. It processes parallel netCDF requests from user compute nodes and, after optimization, passes the parallel I/O requests down to MPI-IO library. The I/O servers receive the MPI-IO requests and do I/O over the back-end storage on behalf of the user. 28

2.5 File views illustrated: filetypes are built from etypes. The filetype access pattern is implicitly iterated forward starting from the disp. An actual count for the filetype is not required as it conceptually repeats forever, and the amount of I/O done is dependent on the buffer datatype and count. 36

2.6 (a) Example POSIX I/O request. Using traditional POSIX interfaces for this access pattern cost four I/O requests, one per contiguous region. (b) Example two-phase I/O request. Interleaved file access patterns can be effectively accessed in larger file I/O operations with the two-phase method. 39

2.7 (a) Probably data sieve: Data sieving reduces I/O requests by a factor of 4, but almost doubles the I/O amount (b) Do not data sieve: Data sieving I/O requests are reduced by half, but almost 4 (8 if write) times more data is accessed (c) Do not data sieve: Data sieving increases I/O requests and only marginally reduces I/O amount. (d) Do not data sieve (Pareto optimal):Data sieving doubles I/O requests, but has no effect on I/O amount. (e) Probably data sieve: Data sieving reduced I/O requests by a factor of 4, but almost doubles I/O. 40

2.8 Evaluating the file access pattern alone in this case does not paint the entire I/O picture. The small noncontiguous memory pieces break up the large contiguous file access pattern into many small I/O requests. Since these small I/O requests end up next to each other, data sieving can reduce the number of I/O requests by a factor of 4 without accessing any extraneous data, making data sieving Pareto optimal, assuming it takes longer to read/write 1 unit of data 4 times than to copy 4 units of data into or out of the buffer and to read/write 4 units of data. 42

2.9 The three main factors to consider in determining whether to use data sieving are whether the user buffer is noncontiguous with small pieces, the size of the noncontiguous file regions, and the distribution of the file accesses all with respect to the data sieving buffer size. If both memory and file descriptions are contiguous, do not use data sieving. 43

2.10 Typical parallel file system configuration. Clients have parallel access to components within the metadata and data groups. 45

3.1 List I/O read prototype (list_io_write has the same parameters). 56

3.2 Example list I/O write. Since only contiguous regions can be described using the POSIX read/write interfaces, four I/O calls would be required instead of one list I/O write. 57

3.3 PVFS list I/O read prototype (pvfs_write_list has the same parameters. 58

3.4 Example MPI-IO C code. 59

3.5 File datatypes are replicated and read into memory until the read call has accessed that correct amount of data. 60

3.6 Example flattening of a file datatype. File datatypes are converted into lists of file offsets and lengths in order to generate the necessary parameters to use the list I/O interface. 61

3.7 Example one-dimensional cyclic access. An entire file stores a two-dimensional array and each processor is in charge of an equal amount of columns. The file view is also flattened into one-dimension. 64

3.8 Example block-block access. An entire file stores a two-dimensional array of blocks, and each processor is in charge of a single block. The file view has been flattened into 1-dimension. 65

3.9 One-dimensional cyclic read results with various clients. These results are obtained by using 8-32 clients reading data with the one-dimensional cyclic file access pattern. 66

3.10 One-dimensional cyclic write results with various clients. These results are obtained by using 8-32 clients writing data with the one-dimensional cyclic file access pattern. 66

3.11 Block-block read results with various clients. These results are obtained by using 4-16 clients reading data with the block-block file access pattern. 67

3.12 Block-block write results with various clients. These results are obtained by using 4-16 clients reading data with the block-block file access pattern. 68

3.13 Tile reader file access pattern. Each processor is in charge of reading the data from a display file into its own local display, also known as a tile. This results in a noncontiguous file access pattern. 69

3.14 Tile reader benchmark results. 71

3.15 Three-dimensional block test access pattern. The access pattern for 8, 27, and 64 processors is shown in (a), (b), and (c), respectively. 71

3.16 Three-dimensional block test results. 72

3.17 FLASH memory datatype. Each computing processor contains 80 blocks, so as we scale up the number of computing processors, we linearly increase the dataset size. 75

3.18 FLASH file datatype. This figure describes the hierarchy of the file datatype. At the highest level of the hierarchy, variables are contiguous. Within every variable, there are all the FLASH blocks from all the processors. 76

3.19 Results of the FLASH I/O benchmark with 2 - 64 processors. Collective I/O performs exceptionally well due to the aggregate contiguous file access pattern. The overhead of exchanging data is minimal compared to the I/O time. 76

4.1 Datatype I/O prototypes. 79

4.2 Example tile reader file access pattern conversion. (A) shows how we convert a struct datatype into an indexed dataloop for performance optimization. This conversion eliminates the need for the MPI_LB and MPI_UB dataloops, making the dataloop representation smaller. (B) is an example of loop fusion in which we can merge datatypes into a single dataloop. The contig and named dataloops can be sufficiently described by the vector dataloop above them, eliminating the need for them. 81

4.3 Example datatype I/O call. Since file datatype are broken into file offset-length pairs at the I/O servers, the number of I/O requests is dramatically reduced for regular access patterns. 83

4.4 Tile reader performance results. 89

4.5 Three-dimensional block read and write performance. 90

4.6 FLASH I/O performance. 92

5.1 An example of how an access pattern is created from the HPIO parameters. 95

5.2 HPIO results from testing various region counts. 99

5.3 HPIO results from testing various region sizes. 100

5.4 HPIO results from testing various region spacing. 103

5.5 Example code conversion from the POSIX interface to the MPI-IO interface. 105

5.6 (a) Original layout of variables in data cells. (b) Reorganization of data to combine file regions during write operations increases I/O bandwidth. 106

5.7 Cost of collective I/O synchronization. Even if collective I/O (a) can reduce the overall I/O times, individual I/O (b) outperforms it in this case because of no implicit synchronization costs. 107

6.1 Database segmentation. 111

6.2 Results when scaling up the number of processors with no-sync/sync query options. 124

6.3 Individual phase timing results when scaling up the number of processors with no-sync/sync query options for MW. 125

6.4 Individual phase timing results when scaling up the number of processors with no-sync/sync query options for WW-POSIX. 126

6.5 Individual phase timing results when scaling up the number of processors with no-sync/sync query options for WW-List. 128

6.6 Individual phase timing results when scaling up the number of processors with no-sync/sync query options for WW-Coll. 129

6.7 Results when scaling up the compute speed with no-sync/sync query options. 131

6.8 Individual phase timing results when scaling up the compute speed with no-sync/sync query options for MW. 132

6.9 Individual phase timing results when scaling up the compute speed with no-sync/sync query options for WW-POSIX. 133

6.10 Individual phase timing results when scaling up the compute speed with no-sync/sync query options for WW-List. 134

6.11 Individual phase timing results when scaling up the compute speed with no-sync/sync query options for WW-Coll. 135

7.1 The three lock methods described in this chapter: (a) POSIX locking, (b) list locking, and (c) datatype locking. 142

7.2 Lock server architecture. 149

7.3 Lock tests without contention: (a) acquire and (b) unlock. Each client accesses locks within an 8 MB range. 153

7.4 Lock tests without contention: (a) acquire and (b) unlock. Each client accesses locks within a 512 MB range. 155

7.5 Lock tests with contention: (a) datatype, (b) list, and (c) POSIX. Each client accesses locks within an 8 MB range. 157

7.6 Lock tests with contention: (a) datatype, (b) list, and (c) POSIX. Each client accesses locks within a 512 MB range. 158

7.7 (a) Raw I/O bandwidth. (b) I/O bandwidth as a fraction of single writer I/O time. 162

7.8 % of I/O bandwidth compared with no locking. 164

7.9 (a) S3aSim total execution time from 2 - 64 processes. (b) S3aSim I/O time from 2 - 64 processes. 165

7.10 (a) S3aSim I/O time as a % of total execution time from 2 - 64 processes. (b) S3aSim % of I/O bandwidth compared to no locking from 2 - 64 processes. 166

8.1 Atomicity challenges for parallel file systems can occur even with contiguous I/O operations. 171

8.2 Serialization using locking versus concurrent I/O using versioning. 173

8.3 HPIO results from testing various region counts and different atomicity methods. 179

8.4 HPIO results from testing various region spacings and different atomicity methods. 180

CHAPTER 1

Introduction

Large-scale applications have begun to rely heavily on high-level I/O APIs such as HDF5 [**37**] and parallel netCDF [**47**] for their storage needs. These APIs allow scientists to describe their data in meaningful terms to them (as both structured and unstructured, typed data) and to store and retrieve this data in a manner that is portable across various high-performance computing platforms. Scientists have rich abstract languages they can use to describe their data. It is more intuitive for application I/O as a whole to be described in terms of the datatypes and organizations that the scientists are really using, rather than in terms of independent reads or writes of bytes on many processors.

High-level I/O APIs also allow I/O experts to embed the knowledge of how to efficiently access storage resources in a library that many applications can use portably across numerous file systems. The result is a big win for both groups. High-level I/O library developers use MPI-IO as their interface to storage resources. The MPI-IO interface maps higher-level accesses to file system operations and provides a collection of key optimizations. MPI-IO also understands structured data access, enabling high-level I/O API programmers to describe noncontiguous accesses as single units, just as the scientist did, and to interface underlying resources through a portable API.

Numerous studies have shown that I/O for large-scale applications is dominated by numerous noncontiguous I/O requests [**85, 62, 3**]. Today's parallel file systems do not, for the most part, support structured or even noncontiguous access. Instead they tend to favor

the POSIX interface, allowing for only contiguous regions to be accessed and modified. This approach severely limits the ability of the MPI-IO layer to succinctly and efficiently carry out the requested I/O operations at the file system level.

A significant step in the direction of efficient unstructured data access is the *list I/O* interface [**93**], implemented in the Parallel Virtual File System (PVFS) and supported under MPI-IO [**17, 16**]. This new interface, when well supported by the parallel file system, allows unstructured accesses to be succinctly described and serves as a solid building block for a MPI-IO device driver implementation. This interface is general, easy to understand, and could be implemented for most file systems. Emerging file systems will likely have such an interface. However, because it does not retain any information on the regularity of accesses (such as stride information), the representation of structured accesses using a list I/O interface can be very large. Building, transmitting, and processing this representation can significantly limit performance when accesses consist of many small regions [**100**]. The next logical step in efficient support for structured data access, *datatype I/O*, provides a mechanism for mapping MPI datatypes (passed to MPI-IO routines) into a type representation understood by the file system. The new representation maintains the concise descriptions possible with MPI type constructors such as `MPI_Type_vector`. This representation is passed over the network where it is processed directly on the I/O servers to avoid the overhead of building lists of I/O regions at the MPI-IO layer, passing these lists over the network as part of the file system request, or processing these lists during I/O. While list I/O is still well suited for unstructured I/O requests, datatype I/O will outperform list I/O for most structured access patterns.

As our new I/O methods augmented the programmer's repertoire, we conducted a study on basic I/O characteristics to figure out when to best use each I/O method when given

an application-specific set of I/O parameters. We synthesized noncontiguous I/O access for storing scientific data in a modern parallel file system and evaluated the effects of varying three major I/O characteristics: region count, region size and region spacing. We created the High-Performance I/O benchmark, *HPIO*, to help application designers optimize their I/O algorithms. HPIO tests various I/O methods (POSIX I/O, list I/O, two-phase I/O, and datatype I/O methods) in all I/O cases (c-c, nc-c, c-nc, nc-nc) with our chosen I/O characteristics. The results of our testing provides guidelines that can help scientific application developers understand how their design choices with respect to I/O algorithms can significantly affect I/O performance between 1 to 3 orders of magnitude.

As a compliment to the bottom-up basic access pattern study, we examine application-level I/O strategies from an important domain in computational science: parallel sequence-search. Biologists are looking at BLAST, FASTA and other sequence-search heuristics to understand the mysteries of recently mapped DNA or amino-acid sequences. Parallel sequence-search, in tools such as mpiBLAST and TurboBlast, are increasingly being used to reduce application execution times. *S3aSim*, a sequence similarity search algorithm simulator, has been developed as a tool for understanding the implications of various I/O strategies in this class of applications. This case study shows that collective I/O strategies may not make sense for this type of master/worker parallelization. Individual I/O strategies are better at spreading the aggregate I/O utilization across the running time of the application and do not incur unnecessary synchronization overheads.

New list I/O and datatype I/O methods provide an efficient implementation of noncontiguous high-level I/O calls, however, lack an atomic mode. Adding atomicity to these I/O calls is difficult, however, atomicity is an important building block for handling challenges such as redundant storage, real-time visualization, and consistent data placement for parallel

computing. Furthermore, strong atomicity semantics are defined for many I/O APIs. In this paper, we investigate two approaches for adding atomicity for noncontiguous I/O without a significant performance overhead. First, we examine a true byte-range DLM approach where servers hold locks on their own data. In this scheme, we apply the list I/O and datatype I/O access pattern descriptions used in our I/O methods toward lock descriptions with a hybrid lock protocol designed to optimized parallelization. In a range of application benchmarks, lock overhead for noncontiguous I/O operations with list lock or datatype lock that uses a form of the optimized lock protocol does not exceed 30% and is negligible in some cases.

Our second method at handing atomicity efficiently uses a versioning approach. Versioning has been used in many file systems to implement audit trails, provide security features, undelete files, and to manage access to shared data. We have created a parallel versioning protocol to efficiently implement atomic noncontiguous operations for parallel file systems. Our prototype implementation shows a lot of promise for scientific applications since they tend to be write-dominated. The initial versioned write results in our HPIO benchmark have better I/O bandwidth when compared to their traditional I/O counterparts due to the advantage of log structured on-disk storage.

This paper is organized as follows. Chapter 2 provides a brief background on the field of parallel I/O. Chapter 3 describes the design and implementation of the list I/O interface for improving unstructured I/O. Chapter 4 discuss an optimized structured I/O method called datatype I/O. Chapter 5 analyzes the performance effect of changing various I/O characteristics with different I/O methods and access patterns. Chapter 6 describes the parallel sequence-search application domain from an I/O perspective through our S3aSim benchmark. Chapter 7 explains the design and experimentation of our true byte-range granular DLM implementation. Chapter 8 details our version-based approach for implementing

atomic I/O operations in parallel file systems. Finally, Chapter 9 concludes this paper and discusses our future work.

CHAPTER 2

High-Performance I/O Software Stack

High-performance I/O is an established field in computing engineering. This chapter discusses how large-scale applications apply parallel I/O techniques through the commonly used I/O software stack depicted in Figure 2.1. First, Section 2.1 discusses the common terms in the I/O field and the concepts of noncontiguous I/O. Section 2.2 describes application specific libraries. Next, Section 2.3 talks about the MPI-IO specification and optimizations in ROMIO, a MPI-IO implementation from Argonne National Laboratory. Section 2.4 gives a brief summary on current production and research parallel file systems.

2.1. Definitions and Noncontiguous I/O

I/O, or input/output, is a generalized term in computing, which refers to a large collection of interfaces that send and receive information. Common I/O devices include keyboards, mice, network cards, and disk drives. In this paper, I/O specifically refers a method that a client uses to access data on a storage device. When a single client simultaneously accesses

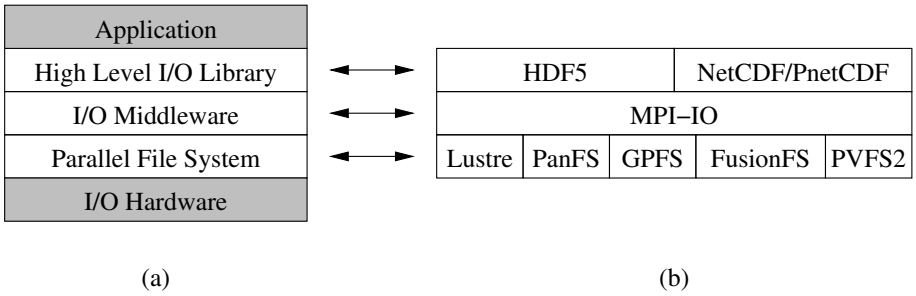

Figure 2.1. (a) Abstract I/O software stack for scientific computing. (b) Current components of the commonly used I/O software stack.

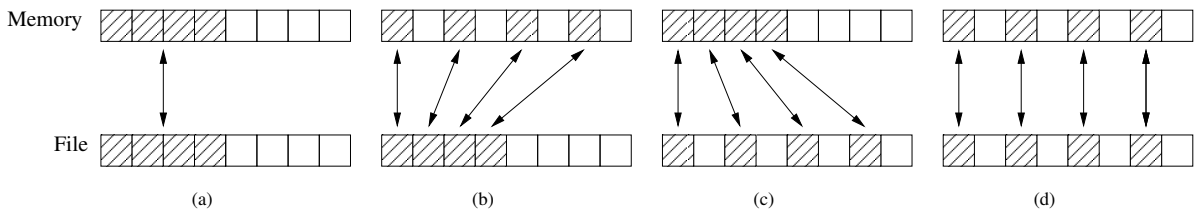

Figure 2.2. Various I/O access cases. (a) refers to contiguous in memory and file (c-c). (b) refers to noncontiguous in memory and contiguous in file (nc-c). (c) refers to contiguous in memory and noncontiguous in file. (d) refers to noncontiguous in memory and file (nc-nc).

multiple storage devices, it uses *parallel I/O*. A file system which has multiple storage devices or nodes that can be accessed simultaneously is called a *parallel file system*. Most of the time, parallel file systems are also considered to be *distributed file systems*, or file systems that support sharing data across a network. However, distributed file systems such as the early version of the network file system (NFS) [**72**], have typically been centralized, and are considered to be distributed, however, not parallel.

All types of I/O access patterns, including both contiguous and noncontiguous cases, are shown in Figure 2.2. For the contiguous in memory and contiguous in file case, the notation c-c is used. The noncontiguous cases are referred to as nc-c for noncontiguous in memory and contiguous in file, c-nc for contiguous in memory and noncontiguous in file, and nc-nc for noncontiguous in memory and noncontiguous in file. The following sections describe various higher-level views of these simplistic I/O access patterns.

2.2. Portable File Formats and Data Libraries

Low level I/O interfaces, like UNIX I/O, treat files as sequences of bytes. Scientific applications manage data at a higher level of abstraction where users can directly read/write data as complex structures instead of byte streams and have all type information and other

useful metadata automatically handled. Applications commonly run on multiple platforms also require data portability so that the data generated from one platform can be used on another without transformation. As most scientific applications are programmed to run in parallel environments, parallel access to the data is desired. This section briefly describes two popular scientific data libraries and their portable file formats: netCDF and HDF5.

2.2.1. File Access in Parallel Applications

Before presenting a detailed description of library design, general approaches for accessing portable files in parallel applications (in a message-passing environment) are analyzed. The first and most straightforward approach is described in the scenario of Figure 2.3a where one process is in charge of collecting/distributing data and performing I/O to a single file using a serial API. The I/O requests from other processes are carried out by shipping all the data through this single process. The drawback of this approach is that collecting all I/O data on a single process can easily create an I/O performance bottleneck and also overwhelm its memory capacity.

In order to avoid unnecessary data shipping, an alternative approach has all processes perform their I/O independently using the serial API, as shown in Figure 2.3b. In this way, all I/O operations can proceed concurrently, but over separate files (one for each process). Managing a dataset is more difficult, however, when it is spread across multiple files. This approach undermines the library design goal of easy data integration and management.

A third approach introduces a parallel API with parallel access semantics and an optimized parallel I/O implementation where all processes perform I/O operations to access a

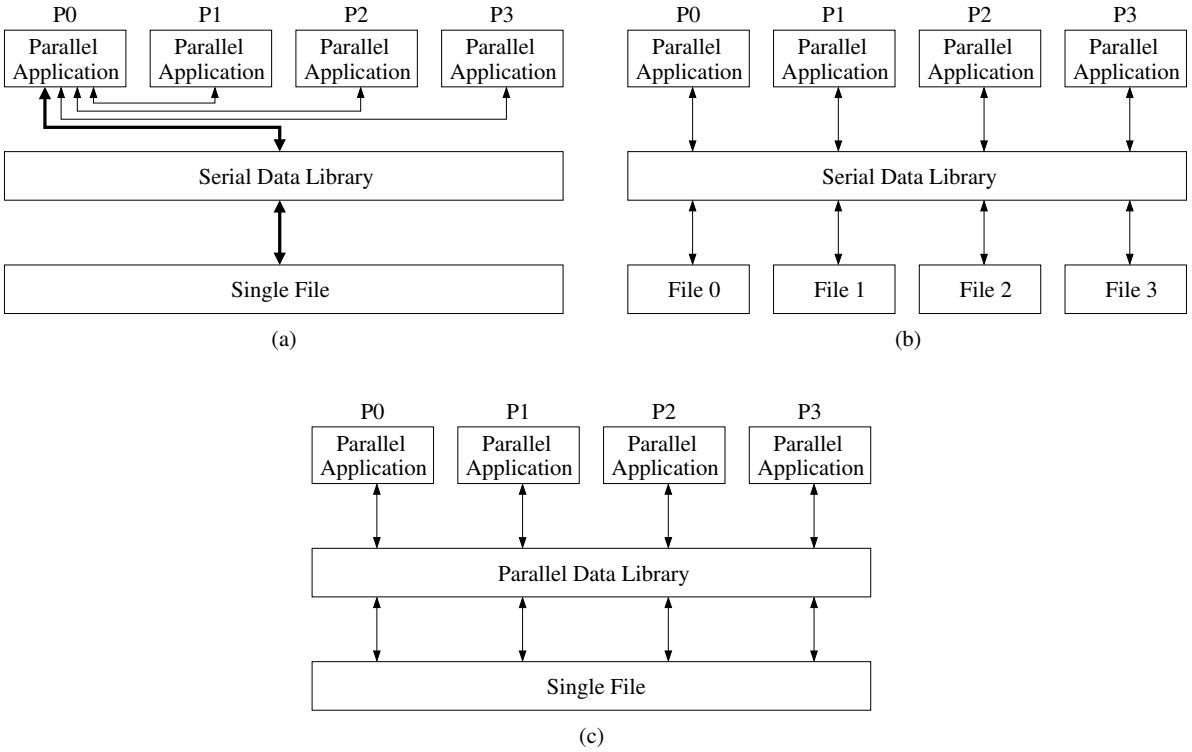

Figure 2.3. Using data libraries in parallel applications: (a) using a serial API to access single files through a single process; (b) using a serial API to access multiple files concurrently and independently; (c) using a parallel API to access single files cooperatively or collectively.

single file. This approach, as shown in Figure 2.3c, both frees the users from dealing with parallel I/O intricacies and provides more opportunities for various parallel I/O optimizations. As a result, this design principle is prevalent among modern scientific data libraries.

2.2.2. NetCDF and Parallel NetCDF

NetCDF [**76**], developed at the Unidata Program Center, provides applications with a common data access method for the storage of structured datasets. Atmospheric science applications, for example, use netCDF to store a variety of data types that include single-point

Table 2.1. NetCDF library functions.

Function Type	Description
Dataset Functions	create/open/close a dataset, set the dataset to define/data mode, and synchronize dataset
Define Mode Functions	define dataset dimensions and variables
Attribute Functions	manage adding, changing, and reading attributes of datasets
Inquiry Functions	return dataset metadata: dim(id, name, len), var(name, ndims, shape, id)
Data Access Functions	provide the ability to read/write variable data in one of the five access methods: single value, whole array, subarray, subsampled array (strided subarray) and mapped strided subarray

observations, time series, regularly spaced grids, and satellite or radar images. Many organizations, such as much of the climate modeling community, rely on the netCDF data access standard for data storage.

NetCDF stores data in an array-oriented dataset which contains dimensions, variables, and attributes. Physically, the dataset file is divided into two parts: file header and array data. The header contains all information (metadata) about dimensions, attributes, and variables except for the variable data itself, while the data section contains arrays of variable values (raw data). Fixed-sized arrays are stored contiguously starting from given file offsets, while variable-sized arrays are stored at the end of the file as interleaved records that grow together along a shared unlimited dimension.

The netCDF operations can be divided into the five categories as summarized in Table 2.1. A typical sequence of operations to write a new netCDF dataset is to create the dataset; define the dimensions, variables, and attributes; write variable data; and close the dataset. Reading an existing netCDF dataset involves first opening the dataset; inquiring about

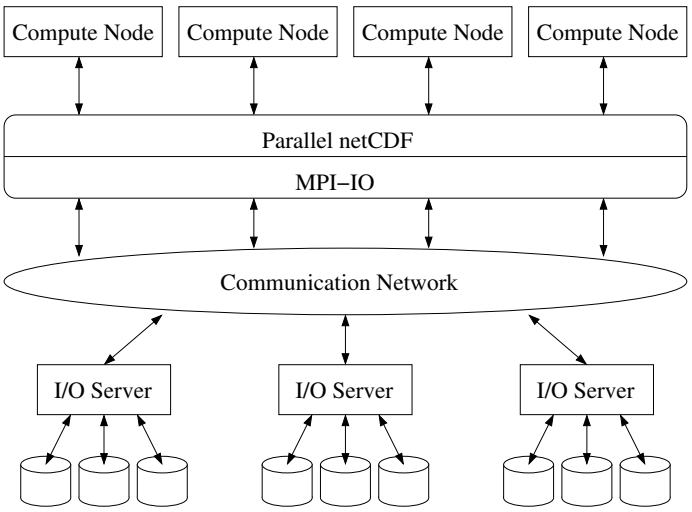

Figure 2.4. Design of PnetCDF on a parallel I/O architecture. PnetCDF runs as a library between the user application and file system. It processes parallel netCDF requests from user compute nodes and, after optimization, passes the parallel I/O requests down to MPI-IO library. The I/O servers receive the MPI-IO requests and do I/O over the back-end storage on behalf of the user.

dimensions, variables, and attributes; then reading variable data; and finally closing the dataset.

The original netCDF API was designed for serial data access, lacking parallel semantics and performance. Parallel netCDF (PnetCDF) [**47**], developed jointly between Northwestern University and Argonne National Laboratory (ANL), provides a parallel API to access netCDF files with significantly better performance. It is built on top of MPI-IO, allowing users to benefit from several well-known optimizations already used in existing MPI-IO implementations, namely the data sieving and two-phase I/O strategies in ROMIO. MPI-IO is explained in further detail in Section 2.3. Figure 2.4 describes the overall architecture for PnetCDF design.

In PnetCDF, a file is opened, operated, and closed by the participating processes in an MPI communication group. Internally, the header is read/written only by a single process,

although a copy is cached in local memory on each process. The *root* process fetches the file header, broadcasts it to all processes when opening a file, and writes the file header at the end of the define mode if any modifications occur in the header. The define mode functions, attribute functions, and inquiry functions all work on the local copy of the file header. All define mode and attribute functions are made collectively and require all the processes to provide the same arguments when adding, removing, or changing definitions so the local copies of the file header are guaranteed to be the same across all processes from the time the file is collectively opened until it is closed.

The parallelization of the data access functions is achieved with two subset APIs, the *high-level API* and the *flexible API*. The high-level API closely follows the original netCDF data access functions and serves as an easy path for original netCDF users to migrate to the parallel interface. These calls take a single pointer for a contiguous region in memory, just as the original netCDF calls did, and allow for the description of single elements (`var1`), whole arrays (`var`), subarrays (`vara`), strided subarrays (`vars`), and multiple noncontiguous regions (`varm`) in a file. The flexible API provides a more MPI-like style of access by providing the user with the ability to describe noncontiguous regions in memory. These regions are described using MPI datatypes. For application programmers that are already using MPI for message passing, this approach should be natural. The file regions are still described using the original parameters. For each of the five data access methods in the flexible data access functions, the corresponding data access pattern is presented as an MPI file view (a set of data visible and accessible from an open file) constructed from the variable metadata (shape, size, offset, etc.) in the netCDF file header and user provided starts, counts, strides, and MPI datatype arguments. For parallel access, each process has a different file view. All processes can collectively make a single MPI-IO request to transfer large contiguous data as

a whole, thereby preserving useful semantic information that would otherwise be lost if the transfer were expressed as per process noncontiguous requests.

2.2.3. HDF5

HDF (Hierarchical Data Format) is a portable file format and software, developed at the National Center for Supercomputing Applications (NCSA). It is designed for storing, retrieving, analyzing, visualizing, and converting scientific data. The current and most popular version is HDF5 [**37**], which stores multi-dimensional arrays together with ancillary data in a portable, self-describing file format. It uses a hierarchical structure that provides application programmers with a host of options for organizing how data is stored in HDF5 files. Parallel I/O is also supported.

HDF5 files are organized in a hierarchical structure, similar to a UNIX file system. Two types of primary objects, groups and datasets, are stored in this structure, respectively resembling directories and files in the UNIX file system. A group contains instances of zero or more groups or datasets while a dataset stores a multi-dimensional array of data elements. Both are accompanied by supporting metadata. Each group or dataset can have an associated attribute list to provide extra information related to the object.

A dataset is physically stored in two parts: a header and a data array. The header contains miscellaneous metadata describing the dataset as well as information that is needed to interpret the array portion of the dataset. Essentially, it includes the name, datatype, dataspace, and storage layout of the dataset. The name is a text string identifying the dataset. The datatype describes the type of the data array elements and can be a basic (atomic) type or a compound type (similar to a *struct* in C language). The dataspace defines the dimensionality of the dataset, i.e., the size and shape of the multi-dimensional

Table 2.2. HDF5 interfaces.

Interface	Function Name Prefix and Functionality
Library Functions	H5: General HDF5 library management
Attribute Interface	H5A: Read/write attributes
Dataset Interface	H5D: Create/open/close and read/write datasets
Error Interface	H5E: Handle HDF5 errors
File Interface	H5F: Control HDF5 file access
Group Interface	H5G: Manage the hierarchical group information
Identifier Interface	H5I: Work with object identifiers
Property List Interface	H5P: Manipulate various object properties
Reference Interface	H5R: Create references to objects or data regions
Dataspace Interface	H5S: Defining dataset dataspace
Datatype Interface	H5T: Manage type information for dataset elements
Filters & Compression Interface	H5Z: Inline data filters and data compression

array. The dimensions of a dataset can be either fixed or unlimited (extensible). Unlike netCDF, HDF5 supports more than one unlimited dimension in a dataspace. The storage layout specifies how the data arrays are arranged in the file.

The data array contains the values of the array elements and can be either stored together in contiguous file space or split into smaller *chunks* stored at any allocated location. Chunks are defined as equally-sized multi-dimensional subarrays (blocks) of the whole data array and each chunk is stored in a separate contiguous file space. The chunked layout is intended to allow performance optimizations for certain access patterns, as well as for storage flexibility. Using the chunked layout requires complicated metadata management to keep track of how the chunks fit together to form the whole array. Extensible datasets whose dimensions can grow are required to be stored in chunks. One dimension is increased by allocating new chunks at the end of the file to cover the extension.

The HDF5 library provides several interfaces that are categorized according to the type of information or operation the interface manages. Table 2.2 summarizes these interfaces.

To write a new HDF5 file, one needs to first create the file, adding groups if needed; create and define the datasets (including their datatypes, dataspaces, and lists of properties like the storage layout) under the desired groups; write the data along with attributes; and finally close the file. The general steps in reading an existing HDF5 file include opening the file; opening the dataset under certain groups; querying the dimensions to allocate enough memory to a read buffer; reading the data and attributes; and closing the file.

HDF5 also supports access to portions (or selections) of a dataset by *hyperslabs*, their unions, and lists of independent points. Basically, a hyperslab is a subarray or strided subarray of the multi-dimensional dataset. The selection is performed in the file dataspace for the dataset. Similar selections can be done in the memory dataspace so that data in one file pattern can be mapped to memory in another pattern as long as the total number of data elements is equal.

HDF5 supports both sequential and parallel I/O. Parallel access, supported in the MPI programming environment, is enabled by setting the file access property to use MPI-IO when the file is created or opened. The file and datasets are collectively created/opened by all participating processes. Each process accesses part of a dataset by defining its own file dataspace for that dataset. When accessing data, the data transfer property specifies whether each process will perform independent I/O or all processes will perform collective I/O.

2.3. MPI-IO and ROMIO

Before the message passing interface (MPI) [**55**], there were proprietary message passing libraries available on several computing platforms. Portability was a major issue for application designers and thus more than 80 people from 40 organizations representing universities,

parallel system vendors, and both industrial and national research laboratories formed the Message Passing Interface (MPI) Forum. MPI-1 was established by the forum in 1994. A number of important topics (including parallel I/O) had been intentionally left out of the MPI-1 specification and were to be addressed by the MPI Forum in the coming years. In 1997, the MPI-2 standard was released by the MPI Forum which included support for parallel I/O among a number of other useful new features for portable parallel computing (remote memory operations and dynamic process management). The I/O goals of the MPI-2 standard were to provide developers with a portable parallel I/O interface that could richly describe even the most complex of access patterns. ROMIO [**77**] is the reference implementation distributed with ANL's MPICH library. ROMIO is included in other distributions and is often the basis for other MPI-IO implementations. Frequently, higher level libraries are built on top of MPI-IO, which leverage its portability across different I/O systems while providing features specific to a particular user community. Examples such as netCDF and HDF5 were discussed in Section 2.2.

2.3.1. MPI-IO Interface

The MPI-IO interface is very powerful and complex. A high learning curve is often a major obstacle to developers using MPI-IO directly, and also one of the reasons most developers subsequently end up indirectly using MPI-IO through higher level interfaces like netCDF and HDF5. A very simple execution order of the functions described in this section is as follows:

(1) MPI_Info_create/MPI_Info_set (optional)

(2) datatype creation (optional)

(3) MPI_File_open

(4) MPI_File_set_view (optional)

(5) MPI_File_read/MPI_File_write

(6) MPI_File_sync (optional)

(7) MPI_File_close

(8) datatype deletion (optional)

(9) MPI_Info_free (optional)

2.3.1.1. Open, Close, and Hints.

```
MPI_File_open(comm, filename, amode, info, fh)
MPI_File_close(fh)
MPI_Info_create(info)
MPI_Info_set(info, key, value)
MPI_Info_free(info)
```

The `MPI_File_open` call, not only opens the file, but is also the typical point at which to pass optimization information to an MPI-IO implementation. `MPI_Info_create` should be used to instantiate and initialize an `MPI_Info` object, and then `MPI_Info_set` is used to set specific hints (*key*) in the info object. The info object should then be passed to `MPI_File_open` and later freed with `MPI_Info_free` after the file is closed. If an info object is not needed, `MPI_INFO_NULL` can be passed to open. The hints in the info object are used to either control optimizations directly in an MPI-IO implementation or to provide additional access information to the MPI-IO implementation so it can make better decisions on optimizations. Some specific hints are described in 2.3.2.

2.3.1.2. Derived Datatypes. Before delving into the rest of the I/O interface and capabilities of MPI-IO, it is essential to have a sound understanding of derived datatypes. Datatypes

Table 2.3. Commonly used MPI datatype constructor functions. Internal offsets can be described in terms of the base datatype or in bytes.

function	internal offsets	base types
MPI_Type_contiguous	none	single
MPI_Type_vector	regular (old types)	single
MPI_Type_hvector	regular (bytes)	single
MPI_Type_indexed	arbitrary (old types)	single
MPI_Type_hindexed	arbitrary (bytes)	single
MPI_Type_struct	arbitrary (old types)	mixed

are what distinguish the MPI-IO interface from the more familiar standard POSIX I/O interface.

One of the most powerful features of the MPI specification is user defined derived datatypes. MPI's derived datatypes allow a user to describe an arbitrary pattern in a memory space. This access pattern, possibly noncontiguous, can then be logically iterated over the memory space. Users may define derived datatypes based on elementary MPI predefined datatypes (`MPI_INT`, `MPI_CHAR`, *etc.*) as well as previously defined derived datatypes. A common and simple use of derived datatypes is to single out values for a specific subset of variables in multi-dimensional arrays.

After using one or more of the basic datatype creation functions in table 2.3, `MPI_Type_commit` is used to finalize the datatype and must be called before use in any MPI-IO calls. After the file is closed, the datatype can then be freed with `MPI_Type_free`.

Seeing as a derived datatype simply maps an access pattern in a logical space, while the discussion above has focused on memory space, it could also apply to file space.

2.3.1.3. File Views.

```
MPI_File_set_view(fh, disp, etype, filetype, datarep, info)
```

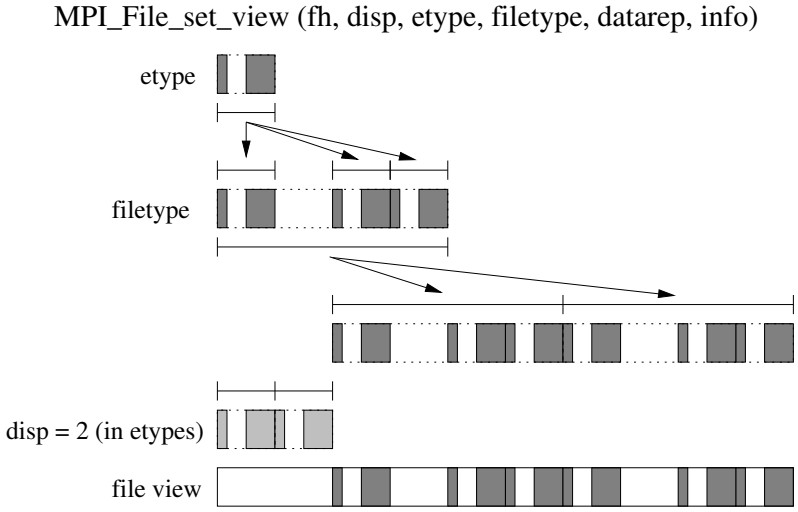

Figure 2.5. File views illustrated: filetypes are built from etypes. The filetype access pattern is implicitly iterated forward starting from the disp. An actual count for the filetype is not required as it conceptually repeats forever, and the amount of I/O done is dependent on the buffer datatype and count.

File views specify accessible file regions using derived datatypes. This function should be called after the file is opened, if at all. Not setting a file view allows the entire file to be accessed. The defining datatype is referred to as the *filetype*, and the *etype* is a datatype used as an elementary unit for positioning. Figure 2.5 illustrates how the parameters in `MPI_File_set_view` are used to describe a "window" revealing only certain bytes in the file. The displacement (*disp*) dictates the start location of the initial filetype in terms of etypes. The file view is defined by both the displacement and filetype together. While this function is collective, it is important each process defines its own individual file view. All processes in the same communicator must use the same etype. The *datarep* argument is typically set to "native," and has to do with file interoperability. If compatibility between MPI environments is needed or the environment is heterogeneous, then "external32" or "internal" should be used. File views allow an MPI-IO read or write to access complex noncontiguous regions in

a single call. This is the first major departure from the POSIX I/O interface, and one of the most important features of MPI-IO.

2.3.1.4. Read and Write.

```
MPI_File_read(fh, buf, count, datatype, status)
MPI_File_write(fh, buf, count, datatype, status)
MPI_File_read_at(fh, offset, buf, count, datatype, status)
MPI_File_write_at(fh, offset, buf, count, datatype, status)
MPI_File_sync(fh)
```

In addition to the typical MPI specific arguments like the MPI communicator, the datatype argument in these calls is the second important distinction of MPI-IO. Just as the file view allows one MPI-IO call to access multiple noncontiguous regions in file, the datatype argument allows a single MPI-IO call to access multiple memory regions in the user buffer with a single call. The *count* is the number of datatypes in memory being used.

The functions `MPI_File_read` and `MPI_File_write` use `MPI_File_seek` to set the position of the file pointer in terms of etypes. It is important to note that the file pointer position respects the file view, skipping over inaccessible regions in the file. Setting the file view resets the individual file pointer back to the first accessible byte.

The `MPI_File_read_at` and `MPI_File_write_at`, "_at" variations of the read and write functions, explicitly set out a starting position in the additional *offset* argument. Just as in the seek function, the offset is in terms of etypes and respects the file view.

Similar to MPI non-blocking communication, non-blocking versions of the I/O functions exist and simply prefix read and write with "i" so the calls look like `MPI_File_iread`. The

I/O need not be completed before these functions return. Completion can be checked just as in non-blocking communication with completion functions like `MPI_Wait`.

The `MPI_File_sync` function is a collective operation used to ensure written data is pushed all the way to the storage device. Open and close also implicitly guarantee data for the associated file handle is on the storage device.

2.3.1.5. Collective Read and Write.

```
MPI_File_read_all(fh, buf, count, datatype, status)
MPI_File_write_all(fh, buf, count, datatype, status)
```

The collective I/O functions are prototyped the same as the independent `MPI_File_read` and `MPI_File_write` functions and have "_at" equivalents as well. The difference is that the collective I/O functions must be called collectively among all the processes in the communicator associated with the particular file at open time. This explicit synchronization allows processes to actively communicate and coordinate their I/O efforts for the call. One major optimization for collective I/O is disk-directed I/O [**44, 57**]. Disk-directed I/O allows I/O servers to optimize the order in which local blocks are accessed. Another optimization for collective I/O is the two-phase method that is detailed in the next section.

2.3.2. Significant Optimizations in ROMIO

The ROMIO implementation of MPI-IO contains several optimizations based on the POSIX I/O interface, making them portable across many file systems. It is possible, however, to implement a ROMIO driver with optimizations specific to a given file system. In fact, the current version of ROMIO already includes optimizations for PVFS2 [**95**], GPFS [**82**], and

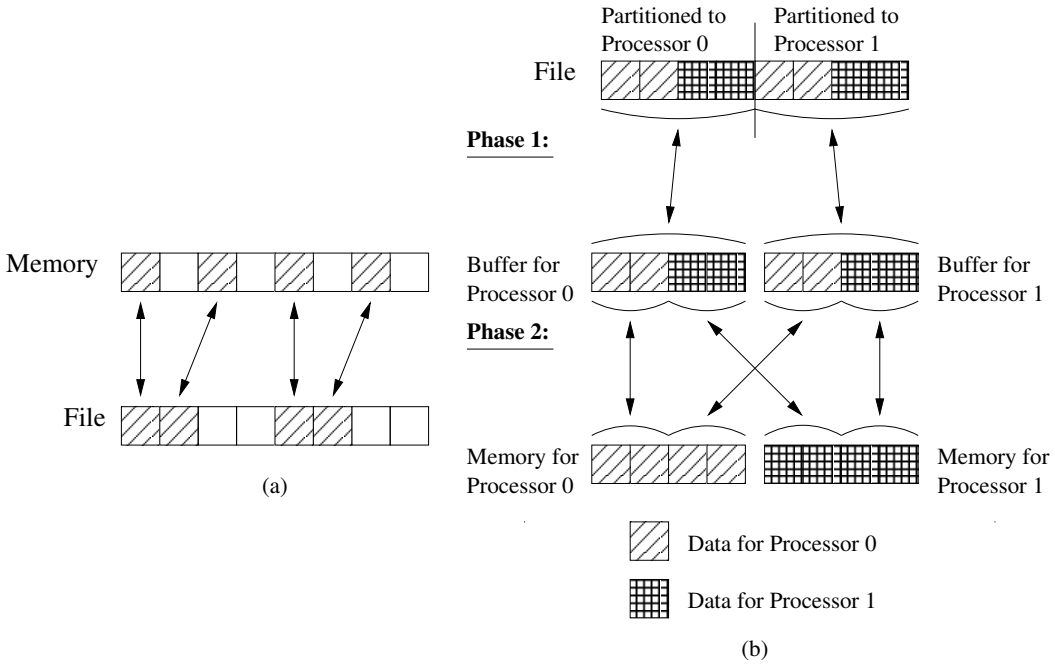

Figure 2.6. (a) Example POSIX I/O request. Using traditional POSIX interfaces for this access pattern cost four I/O requests, one per contiguous region. (b) Example two-phase I/O request. Interleaved file access patterns can be effectively accessed in larger file I/O operations with the two-phase method.

many other file systems. The most convenient means for controlling these optimizations is through the MPI-IO hints infrastructure mentioned briefly above.

2.3.2.1. POSIX I/O. All parallel file systems support what is called the *POSIX I/O* interface, which relies on an offset and a length in both memory and file to service an I/O request. This method can service noncontiguous I/O access patterns by dividing them up into contiguous regions and then individually accessing these regions with corresponding POSIX I/O operations. While such use of POSIX I/O can fulfill any noncontiguous I/O request with this technique, it does incur several expensive overheads. The division of the I/O access pattern into smaller contiguous regions significantly increases the number of I/O requests processed by the underlying file system as shown in Figure 2.6a. The serious

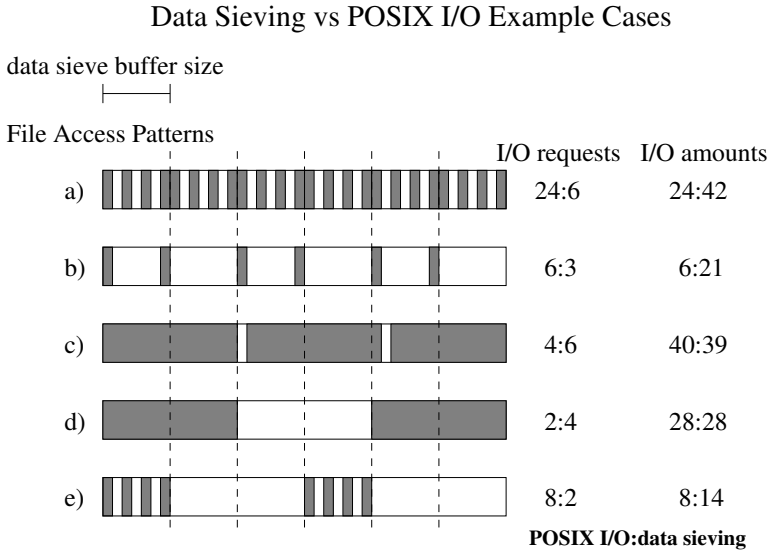

Figure 2.7. (a) Probably data sieve: Data sieving reduces I/O requests by a factor of 4, but almost doubles the I/O amount (b) Do not data sieve: Data sieving I/O requests are reduced by half, but almost 4 (8 if write) times more data is accessed (c) Do not data sieve: Data sieving increases I/O requests and only marginally reduces I/O amount. (d) Do not data sieve (Pareto optimal):Data sieving doubles I/O requests, but has no effect on I/O amount. (e) Probably data sieve: Data sieving reduced I/O requests by a factor of 4, but almost doubles I/O.

overhead sustained from servicing so many individual I/O requests limits performance for noncontiguous I/O when using the POSIX interface. Fortunately for users which have access to file systems supporting only the POSIX interface, two important optimizations exist to more efficiently perform noncontiguous I/O while using only the POSIX I/O interface: data sieving I/O and two-phase I/O.

2.3.2.2. Data Sieving. Since hard disk drives are inherently better at accessing large amounts of sequential data, the *data sieving* technique [**92**] tries to satisfy multiple small I/O requests with a larger contiguous I/O access and later "sifting" the requested data in or out of a temporary buffer. In the read case, a large contiguous region of file data is first read into a temporary data sieving buffer and then the requested data is copied out of the

temporary buffer into the user buffer. For practical reasons, ROMIO uses a maximum data sieving buffer size so multiple data sieving I/O requests may be required to service an access patterns. ROMIO will always try to fill the entire data sieving buffer each time in order to maximize the number of file regions encompassed. In the write case, file data must first be read into the data sieving buffer unless the user define regions in that contiguous file region cover the entire data sieving region. User data can then be copied into the data sieving buffer and then the entire data sieving buffer is written to the file in a single I/O call. Data sieving writes require some concurrency control since data that one process does not intend to modify is still read and then written back with the potential of overwriting changes made by other processes.

Data sieving performance benefits come from reducing the number of head seeks on the disk, the cut in the accrued overhead of individual I/O requests, and large I/O accesses. Figure 2.7a and Figure 2.7e illustrate specific cases where data sieving may do well. Data sieving is less efficient when data is either sparsely distributed or the access pattern consists of contiguous regions much larger than the data sieving buffer size (end case would be a completely contiguous access pattern). In the sparse case, as in Figure 2.7b and 2.7d, the large data sieving I/O request may only satisfy a few user requests, and in even worse, may be accessing much more data than will actually be used (Figure 2.7b). The number of I/O accesses may not be reduced by much, and the extra time spent accessing useless data may be more than the time taken to make more small I/O requests. In the case where the user's access pattern is made up of contiguous regions nearly the size of or greater than the data sieving buffer size, shown in Figure 2.7c and 2.7d, the number of I/O requests generated may actually be greater than the number of I/O requests generated had the user's I/O requests been passed directly to the file system. Additionally, data sieving will have been double

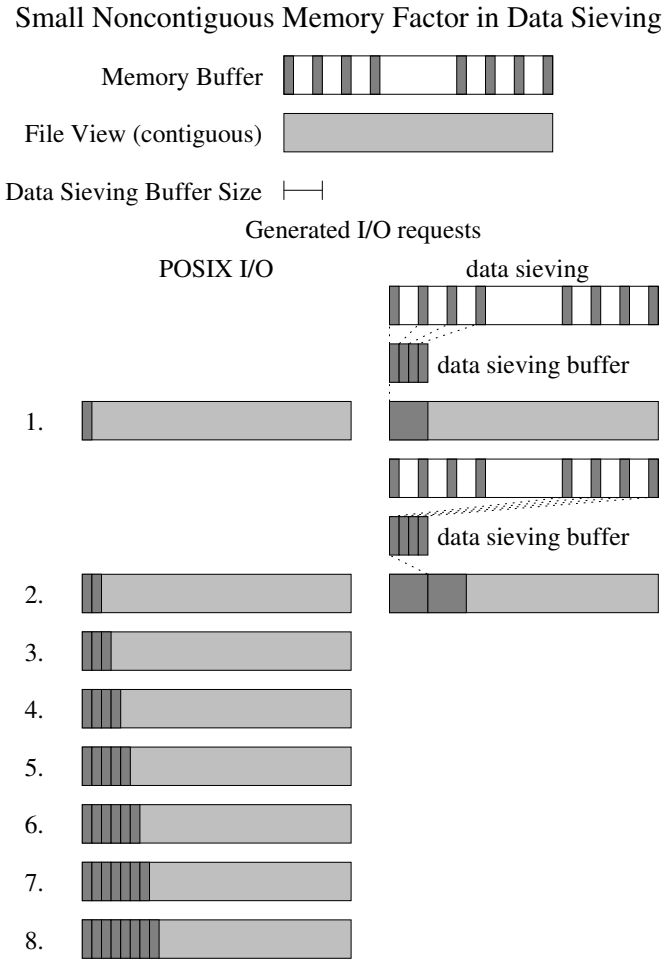

Figure 2.8. Evaluating the file access pattern alone in this case does not paint the entire I/O picture. The small noncontiguous memory pieces break up the large contiguous file access pattern into many small I/O requests. Since these small I/O requests end up next to each other, data sieving can reduce the number of I/O requests by a factor of 4 without accessing any extraneous data, making data sieving Pareto optimal, assuming it takes longer to read/write 1 unit of data 4 times than to copy 4 units of data into or out of the buffer and to read/write 4 units of data.

buffering, and paid an extra memory copy penalty for each time the data sieve buffer was filled and emptied.

One factor not yet considered is the user memory buffer. If the user memory buffer is noncontiguous with small regions (relative to the data sieving buffer), it will have the effect

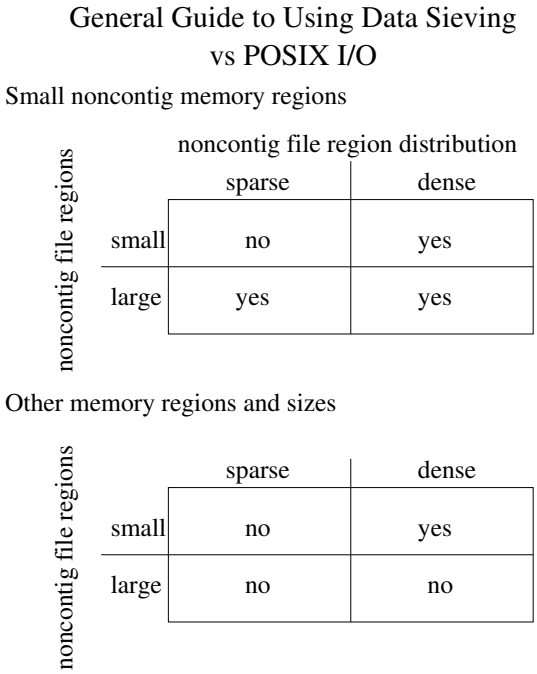

Figure 2.9. The three main factors to consider in determining whether to use data sieving are whether the user buffer is noncontiguous with small pieces, the size of the noncontiguous file regions, and the distribution of the file accesses all with respect to the data sieving buffer size. If both memory and file descriptions are contiguous, do not use data sieving.

of breaking up, but not separating what might have been large contiguous regions in file, thus creating an numerous I/O requests for POSIX I/O. This effect is illustrated in Figure 2.8, and presents an ideal opportunity for data sieving to reduce the overall number of I/O calls, as well as making efficient use of the data sieving buffer. Even if the original filetype consisted of large sparsely distributed regions, data sieving would still likely prove to be very beneficial.

So while data sieving could conceivably result in worse performance (the point at which would be sooner in the case of read-modify-write data sieving writes), some simple considerations can be kept in mind to determine whether data sieving will be a benefit or detriment. Assuming data is fairly uniformly spaced (no locally dense, overall sparse distributions), and

the user access pattern is indeed noncontiguous, Figure 2.9 provides generalizations for determining when data sieving is most appropriate. Small, big, sparse, and dense metrics are all relative to the data sieving buffer size. An MPI-IO implementation ought to preprocess the user's access pattern at least to some degree to determine the appropriateness of data sieving on its own. As mentioned earlier, however, less uniform access patterns may require some user intervention as an automated runtime determination may not catch certain cases. In the previous example (Figure 2.7e), an access pattern which consists of clusters of densely packed data will likely benefit from data sieving. Using only the data sieving technique for I/O will be referred to as *data sieving I/O*.

2.3.2.3. Two-Phase I/O. Figure 2.6b illustrates the two-phase method for collective I/O [**93**], which uses both POSIX I/O and data sieving. This method is referred to as *two-phase I/O* throughout this chapter. The two-phase method identifies a subset of the application processes that will actually do I/O; these processes are called *aggregators*. Each aggregator is responsible for I/O to a specific and disjoint portion of the file.

In an effort to heuristically balance I/O load on each aggregator, ROMIO calculates these *file realms* dynamically based on the aggregate size and location of the accesses in the collective operation. When performing a read operation, aggregators first read a contiguous region containing desired data from storage and put this data in a local temporary buffer. Next, data is redistributed from these temporary buffers to the final destination processes. Write operations are performed in a similar manner. First, data is gathered from all processes into temporary buffers on aggregators. Aggregators read data from storage to fill in the holes in the temporary buffers to make contiguous data regions. Next, this temporary buffer is written back to storage using POSIX I/O operations. An approach similar to data sieving is used to optimize this write back to storage when there are still gaps in the data. As mentioned

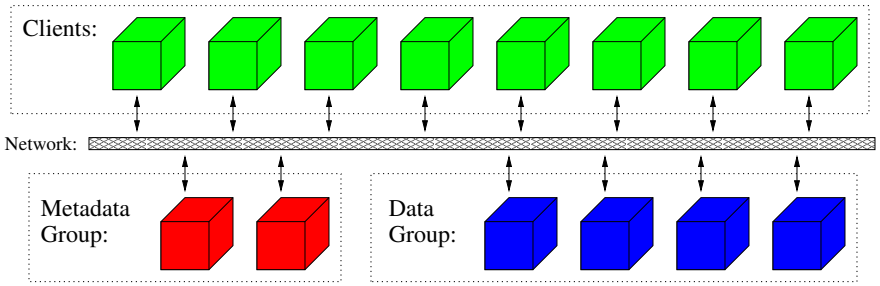

Figure 2.10. Typical parallel file system configuration. Clients have parallel access to components within the metadata and data groups.

earlier, data sieving is also used in the read case. Alternatively, other noncontiguous access methods, such as the ones described in this paper, can be leveraged for further optimization.

The big advantage of two-phase I/O is the consolidation by aggregators of the noncontiguous file accesses from all processes into only a few large I/O operations. One significant disadvantage of two-phase I/O is that all processes must synchronize on the open, set view, read, and write calls. Synchronizing across large numbers of processes with different sized workloads can be a large overhead. Two-phase I/O performance relies heavily on the particular MPI implementation's data movement performance. If the MPI implementation is not significantly faster than the aggregate I/O bandwidth in the system, the overhead of the additional data movement in two-phase I/O will likely prevent two-phase I/O from outperforming direct access optimizations such as list I/O and datatype I/O (discussed later in this paper). Research with techniques such as persistent file realm partitioning [**22**], aligned file realms [**21**] and caching [**23, 49, 48**] has shown much promise in improving collective I/O performance in many cases.

2.4. Parallel File Systems

As the gap between processor and hard disk technologies continues to widen, I/O becomes an increasingly severe performance bottleneck. Parallel file systems, as shown in Figure 2.10,

Table 2.4. A comparison of HPC production file systems.

Category	Lustre	Panasas	GPFS
Dedicated Servers	Yes	Yes	Yes
Consistency Semantics	POSIX	POSIX	POSIX
Consistency Enforcement	DLM	MDS	DLM
Caching	Client memory Block-level	Client memory Block-level	Client memory Block-level
Fault Tolerance	Fail-over servers	OSD-level RAID	Log-based & disk-level RAID
Replication	MDS	OSD-level RAID	Two copies & RAID
Security	Capability	Capability	OpenSSL

help to narrow that gap by scaling up the number of hard disks to increase aggregate I/O bandwidth.

The HPC file system domain can be divided into production file systems and research file systems. Production file systems are typically stable commercial products used in production machines. Some examples of production file systems include Lustre [**52**], Panasas [**59**], GPFS [**82**], SGI's CXFS, IBRIX FusionFS [**38**], and GFS [**33**]. Research file systems are primarily used for trying out new ideas that may one day make it into production if appropriate. There have been a number of research file systems including PVFS [**11, 95**], Clusterfile [**41**], Ceph [**97**], LWFS [**66**], Galley [**61**], Sorrento [**90**], xFS [**2**], Zebra [**36**], Armada [**65**] and many more. The discussion in this section focus on the three most used HPC production file systems (Table 2.4, Lustre, Panasas, and GPFS, in Sections 2.4.1, 2.4.2, and 2.4.3, respectively. Three prominent HPC research file systems (PVFS, LWFS, and Ceph) are described in Sections 2.4.4, 2.4.5, and 2.4.6, respectively (Table 2.5).

2.4.1. Lustre

Lustre [**52**], from Cluster File Systems, gets its name from an amalgam of the terms "Linux" and "cluster." As of the June 2006 TOP500 list, over 70 of the 500 supercomputers use Lustre technology including the number one computer (Lawrence Livermore National Laboratory's BlueGene/L machine). The Lustre architecture is made up of clients, metadata servers (MDSs), and object-storage targets (OSTs). MDSs maintain a transactional record of high-level file system changes, such as the location of related objects and stripe sizes. They are protected from failure through MDS replication and failover techniques. OSTs are responsible for actual file data and locking. Clients make requests to objects on the OSTs, where an object is simply a container of data that may have attributes associated with it. In the future, object-based disks (OBDs) may be able to offload the work necessary to translate file system requests into physical storage requests. Currently, Lustre uses OBD device drivers to implement OBD functionality on top of ext3 or other Linux file systems. Failure of an OST is handled by failover techniques. If a failover OST is unavailable, clients will get errors when trying to access the failed OST and new file create operations will avoid the failed OST.

Lustre uses a distributed lock manager (DLM) to ensure POSIX compliance. The DLM helps Lustre maintain its globally coherent collaborative cache. While locks for an arbitrary byte-range may be requested, OSTs round the granted locks to file system block boundaries. Metadata operations use "intent based" locks (lock requests combined with data requests) for efficient atomic operations that do not require lock revocations. Additionally, Lustre provides snapshots, rollback, and copy-on-write semantics. Lustre uses secure network attached disk (NASD) features for authentication, authorization, and encryption. A preliminary Lustre

driver for the ROMIO MPI-IO implementation has not yet been integrated into the ROMIO distribution.

2.4.2. Panasas

Panasas [**59, 69**] is used on many of the TOP500 supercomputers and was chosen to be deployed on the Los Alamos National Laboratory's new Roadrunner petascale supercomputer. Many application domains, including energy research, high energy physics, atmospheric science and weather predication, seismic data analysis, automotive design and simulation as well as many others, have chosen Panasas as their storage solution. Panasas's main product is the ActiveScale Storage cluster, which uses the Panasas ActiveScale File System (PanFS). The core PanFS architecture is based on the decoupling of the datapath from the control path and the *object* abstraction of file data, similar to Lustre. The PanFS client module accepts POSIX file system commands from the operating system and addresses and stripes the objects across multiple object based storage devices (OSDs). The OSD component in PanFS manages data storage, handles storage-side caching and prefetching, and contains the metadata associated with its objects. Using OSDs instead of the typical block-based storage interface shifts some of the burden of fine-grain layout information to the OSDs. The PanFS metadata server (MDS) coordinates the layout of a file across OSDs, helps maintain RAID integrity, manages file and directory access, and keeps client caches coherent with file locks.

PanFS uses client-side data caching in the Linux buffer/page caches to complement the caching done by the OSDs. It aggregates writes on the client for more efficient I/O operation and also supports prefetching. The MDS handles client cache coherency with a single writer/shared readers protocol with invalidation and flushing. PanFS allows files to individually use different RAID levels across objects. In order to limit *incast* behavior, too many

senders overflowing the network buffers, a two level striping layout is used where most simultaneous accesses are limited to the number of OSDs in a parity stripe. Therefore, files are striped across all the OSDs for maximum bandwidth and the OSDs are broken up into RAID parity groups whenever appropriate (with a maximum of 13 objects per parity group). Panasas OSDs each have two SATA disk drives, a processor, RAM, and a Gigabit Ethernet network interface. An OSD battery-backed RAM cache allows data to be committed even if a power failure occurs.

2.4.3. GPFS

IBM has designed many of the world's top supercomputers, including the recent BlueGene/L architecture. Its flagship file system, the general parallel file system (GPFS) [**82**], is available for its AIX and Linux clusters, and most recently on the BlueGene/L architecture as of December 2005. While GPFS is primarily designed for high-performance computing, it is also used in industries such as media and entertainment, ISPs, finance, telecommunications, electronics, and retail. GPFS uses a shared-disk architecture, where file system nodes have access to all disks through the network fabric. The disks are assumed to use the conventional block I/O interface (as opposed to the object based interfaces used by Lustre and Panasas). GPFS clients communicate directly with file system nodes, which perform I/O on their behalf. GPFS guarantees single-node equivalent POSIX semantics for file system operations across all nodes through the use of distributed locking. The only exception to POSIX compliance is that access time updates are not immediately visible on all nodes. The metanodes that handle metadata in GPFS are allocated dynamically with the help of the global lock manager.

Table 2.5. A comparison of HPC research file systems.

Category	PVFS	LWFS	Ceph
Dedicated Servers	Yes	Yes	Yes
Consistency Semantics	MPI-IO	N/A	POSIX
Consistency Enforcement	Servers	Library	OSD locks
Caching	Server memory Block-level	Library	Client memory Block-level
Fault Tolerance	Fail-over servers	Library	RADOS
Replication	Server RAID	Library	RADOS
Security	In progress	Capability	Capability

The GPFS DLM is composed of a centralized global lock manager and the local lock managers on each file system node. Lock tokens are passed out by the global lock manager to the local lock managers which grant locks. A lock token is revoked only when another node requests conflicting lock operations to the same object. As with Lustre and Panasas, lock tokens play a large role in maintaining cache consistency between nodes. Locks are acquired with byte-range granularity in GPFS and rounded to block boundaries. The first node to write a file will receive a byte-range lock from zero to infinity. When the second node begins writing to the same file, the first node will relinquish part of its byte-range lock token until the offset of the second node's write. As more nodes write to the file, the byte-range lock tokens are further divided. In this way, GPFS attempts to keep locks as large as possible to avoid the increasing overhead of a plethora of locks.

2.4.4. The Parallel Virtual File System

The Parallel Virtual File System (PVFS) is a parallel file system for commodity Linux clusters [**11**]. It provides both a clusterwide consistent name space and user-defined file striping. PVFS is a client-server system consisting of clients, a metadata server, and I/O servers. Clients retrieve a list of the I/O servers that contain the file data from the metadata

server at file open time. Subsequent reading or writing is processed directly by the I/O servers without manager interaction.

The approach that PVFS version 1 (PVFS1) uses for processing requests is detailed in [**50**]. In short, PVFS1 builds a data structure called a *job* on each client and server for every client/server pair involved in an I/O operation. This structure points to a list of *accesses*, which are contiguous regions in memory (on a client) or in file (on a server) that must be moved across the network. This is essentially the flattened representation of the datatype being used to move data. While this is not ideal from a processing overhead standpoint, we will retain this representation in our tests; it would be time consuming to reimplement this core component of PVFS1.

PVFS version 2 (PVFS2) [**95**] is a parallel file system for commodity Linux clusters that is a complete redesign of PVFS1 [**11**], which incorporates lessons learned from the original PVFS file system [4] and also introduces a highly modular infrastructure. PVFS2 uses a client/server architecture, with both the server daemon and client side libraries residing fully in user space. There may be any number of servers, and each server may provide either metadata, file data, or both. Metadata refers to attributes such as timestamps and permissions as well as file system specific parameters. File data refers to the actual data stored in the system. This data is distributed according to rules that are selectable by the user. The default scheme is to stripe data evenly in a similar manner to that of a RAID array. Metadata may also be distributed, though at the granularity level of one server per individual file or directory. Note that there is no need for a shared storage infrastructure; each server manages its own local resources. There is no communication between servers or between clients. Clients communicate exclusively with the servers responsible for the resources that they wish to access. The lowest level network abstraction is provided by a component known

as the Buffered Message Interface. The counterpart disk abstraction, which provides both stream and key/value style access to local storage resources on each server, is called Trove. These two components are coordinated by Flows, which handle buffering, scheduling, and datatype processing between network and disk for bulk transfers. All of these components (along with other peripheral components beyond the scope of this chapter) are coordinated by the Job interface, which manages threading and provides a consistent interface for testing of completion of any pending low-level I/O operation, regardless of which underlying component is ultimately responsible for it. Both the servers and client libraries are implemented through the use of concurrent state machines which operate on top of the Job interface.

2.4.5. LWFS

Catamount, a lightweight operating system for Red Storm (currently number two in the TOP500 as of November 2006) at Sandia National Laboratories (SNL), implements only the required underlying services while avoiding functionality that could compromise application scalability. In the same spirit, the Lightweight File System (LWFS) [**66**] project is a joint collaboration between SNL and the University of New Mexico for investigating the viability of a "lightweight" approach to I/O. The LWFS core only implements a thin layer of software above the hardware, including infrastructure to provide controlled access to distributed data across multiple storage severs, expose the parallelism of multiple storage servers, and allow the client implementation to create additional functionality. Since there are many more compute nodes than I/O nodes, LWFS servers determine when to move data. LWFS clients make asynchronous RPCs and servers either "pull" data for writes or "push" data for reads [**67**]. All data movement is done over the Portals message passing interface which supports one-sided operations.

In accordance with U.S. Department of Energy security requirements, LWFS provide scalable mechanisms for authentication, authorization, and "immediate" revocation of access permissions when policies change. LWFS has coarse-grain access control to *containers* of objects, where every object belongs to a single container. All objects in the same container are subject to the same access control policy. Higher-level libraries are responsible for organizing objects in containers as LWFS does not manage the relationship of objects in a container. To enable scalable security, LWFS uses fully transferable credentials and capabilities. In order to support "immediate" revocation, LWFS invalidates cached entries on each of the storage servers.

2.4.6. Ceph

Ceph [**97**] is a research-oriented file system from the University of California at Santa Cruz. It has three major components: clients that export a near-POSIX file system interface; a cluster of OSDs that collectively store all metadata and data; and a metadata cluster responsible for managing the namespace and coordinating security, consistency, and coherence. As with the other object-based file systems, Ceph separates file metadata management from data storage. Ceph uses its reliable autonomic distributed object store (RADOS) to protect against OSD failures. Primary OSDs forward updates to their replicas in an asynchronous manner for better performance and reads are only serviced by the primary OSD to reduce synchronization costs.

In the metadata cluster, Ceph employs dynamic distributed metadata management that is based on dynamic subtree partitioning. In essence, dynamic distributed metadata maps subtrees of the directory hierarchy to metadata servers based on their workload. Individual directories are hashed across multiple nodes only if they become hot spots.

For data distribution, Ceph uses the Controlled Replication Under Scalable Hashing (CRUSH) algorithm [**98**]. CRUSH relies heavily on a suitably strong multi-input integer hash function. Using the hash function, CRUSH can locate any object with a placement group and an OSD cluster map. Placement rules help CRUSH map the placement groups onto OSDs based on the desired level of replication as well as other constraints. CRUSH also helps Ceph adapt to the addition and removal of storage devices with low overhead.

CHAPTER 3

List I/O

As mentioned in Chapter 2, traditional methods of noncontiguous data access include the use of multiple contiguous I/O calls as well as data sieving techniques. Multiple contiguous I/O builds on traditional POSIX I/O calls (read/write) to perform noncontiguous access. Data sieving is the I/O optimization of reading a large contiguous amount of data from file into a memory buffer, again building on the POSIX API, and performing all noncontiguous data movement using the memory buffer. Alternatively, our research has found that *list I/O* is a promising solution for unstructured noncontiguous I/O. This chapter begins with Section 3.1 describing the new list I/O interface. Section 3.2 and Section 3.3 describe our implementations in both PVFS1 and ROMIO, respectively. Section 3.4 evaluates list I/O performance with both file system direct access and MPI-IO interface results. Section 3.5 summarizes the chapter and discusses future work.

3.1. Interface

The list I/O interface proposed in [**93**] by Thakur et al. is an I/O interface for describing noncontiguous data in memory and in file. It is a simple interface that can describe complex noncontiguous data access in a single function call. We present a visual example of the list I/O interface in Figure 3.2. The list I/O interface proposed is shown in Figure 3.1.

- *mem_list_count* is the total number of contiguous memory regions involved in the data access, which is also the length of the arrays *mem_offsets[]* and *mem_lengths[]*.

```
list_io_read(int mem_list_count,
             char *mem_offsets[ ],
             char mem_lengths[ ],
             int file_list_count,
             int file_offsets[ ],
             int file_lengths[ ])
```

Figure 3.1. List I/O read prototype (list_io_write has the same parameters).

- *mem_offsets[]* is an array of pointers that each point to the beginning of a contiguous memory region.
- *mem_lengths[]* is an array of lengths that match every start of a contiguous memory region with a corresponding memory length.
- *file_list_count* has the same functionality for a file as *mem_list_count* does for memory. It is the total number of contiguous file regions as well as the length of the arrays *file_offsets[]* and *file_lengths[]*.
- *file_offsets[]* is an array of offsets that each point to the beginning of a contiguous file region.
- *file_lengths[]* is the lengths of the file regions that correspond to the file offsets. The sum of the *mem_lengths[]* and *file_lengths[]* must be equivalent.

A naive implementation of list I/O could use the POSIX read/write calls and would provide no performance advantage over those calls. However, building support directly into the parallel file system to handle such a call provides the file system with much needed noncontiguous I/O capabilities.

Using list I/O for noncontiguous data access offers several advantages over traditional methods. Multiple contiguous I/O calls have a large overhead with respect to the number of I/O calls that must be issued to the underlying file system when describing complex noncontiguous I/O access patterns. List I/O can perform a noncontiguous I/O data access

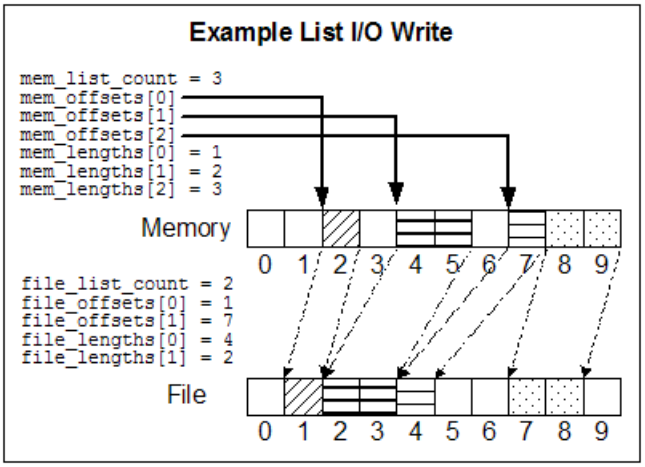

Figure 3.2. Example list I/O write. Since only contiguous regions can be described using the POSIX read/write interfaces, four I/O calls would be required instead of one list I/O write.

with fewer I/O calls with an optimized implementation. Data sieving requires a read-modify-write set of operations and file synchronization (which often has prohibitive overhead) in noncontiguous writes. Some file systems do not provide synchronization and therefore cannot support data sieving in the noncontiguous write case. Also, unlike other noncontiguous methods, data sieving requires memory for the data sieving buffer. List I/O provides a simple interface that can provide high performance I/O for reading and writing data in scientific workloads.

3.2. PVFS1 Implementation

PVFS1 has traditionally supported only contiguous requests for data. To address the performance problems inherent in the access patterns of scientific applications, we have added support for noncontiguous requests in PVFS1. We desired a noncontiguous implementation that would reduce I/O accesses independent of the actual location in file. Based on the interface proposed our implementation of noncontiguous data access, list I/O, would need

```
pvfs_read_list(int mem_list_count,
               char *mem_offsets[ ],
               char mem_lengths[ ],
               int file_list_count,
               int64_t file_offsets[ ],
               int32_t file_lengths[ ])
```

Figure 3.3. PVFS list I/O read prototype (pvfs_write_list has the same parameters.

support to describe any noncontiguous I/O pattern. The I/O servers would require support to process this request appropriately. The user would view the list I/O interface as follows in Figure 3.3.

Mem_list_count holds the total number of contiguous memory locations involved in the noncontiguous access. Similarly, file_list_count is the corresponding number of contiguous file locations. Mem_offsets is an array that references the beginning of each memory region, and the mem_lengths array matches each of these references with the corresponding memory lengths. File_offsets and file_lengths do the same for file regions.

PVFS1 clients make I/O requests through the PVFS1 library. These I/O requests contain information pertaining to a file (metadata, striping parameters) and can ask the I/O servers to perform operations such as read, write, open and close. In order for the I/O request to convey the description of noncontiguous data, we added another field to the I/O request structure to let the I/O servers know that a variable sized trailing data would follow the I/O request. This trailing data contains the file offsets and file lengths of the noncontiguous I/O request.

We modified the I/O server code to correctly process this routine by adding support to receive the trailing data and complete the I/O accesses. We have chosen to allow up to 64 contiguous file regions to be described in trailing data before another I/O request must be issued. Therefore, I/O requests that contain more file regions than the trailing data limit are

```
MPI_File_open(MPI_COMM_WORLD,
              ‘‘/pvfs/test.txt’’,
              MPI_MODE_RDWR,
              MPI_INFO_NULL, &fh);
etype = MPI_INT;
MPI_Type_vector(2, 2, 3,
                MPI_INT,
                &filetype);
MPI_Type_commit(&filetype);
MPI_File_set_view(fh, 0, etype,
                  filetype, datarep,
                  MPI_INFO_NULL);
MPI_File_read(fh, buf, 8,
              MPI_INT, status,
              ierror);
MPI_File_close(&fh);
```

Figure 3.4. Example MPI-IO C code.

broken up into several list I/O requests. This limit was chosen to allow the I/O request and trailing data to travel through the network in a single Ethernet packet (1500 bytes). This is a conservative limit that allows us to see how this approach might be used in a real system.

3.3. ROMIO MPI-IO Implementation

This example MPI-IO C code in Figure 3.4 (graphically depicted in Figure 3.5) performs a collective open of a file with `MPI_File_open`. MPI Type calls are used to create both the memory datatype and the file datatype. Performing the read puts the first two integers from file into the memory buffer, then puts the fourth and fifth integers from file into the memory buffer, continuing the vector pattern of the file datatype until eight `MPI_INTS` are contiguously located in the buffer `buf`.

To take advantage of PVFS1's list I/O in the ROMIO MPI-IO implementation, we implemented the new ADIO read and write functions using `pvfs_read_list` and `pvfs_write_list`.

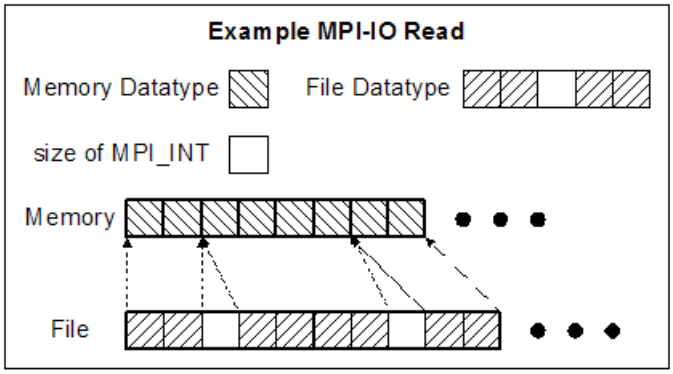

Figure 3.5. File datatypes are replicated and read into memory until the read call has accessed that correct amount of data.

PVFS1 list I/O, requires several parameters, including offset-length pairs for memory locations, offset-length pairs for file locations, and their respective counts. Our new ADIO calls convert MPI types into their respective contiguous regions and create the arrays to pass to the list I/O calls. This conversion process, or *flattening*, is accomplished by decoding the types using the MPI calls `MPI_Type_get_envelope` and `MPI_Type_get_contents`. Figure 3.6 gives an example of this process. The implementation for generating list I/O calls in ROMIO breaks up the file regions into collections of 128 to match the maximum size allowable by the PVFS1 list I/O implementation. Given a large number of noncontiguous accesses, ROMIO will fill the offset-length arrays, perform the required I/O and repeat the sequence until all the noncontiguous regions have been satisfied.

3.3.1. ROMIO Implementation and I/O Methods Discussion

ROMIO list I/O is most effective of the four methods for access patterns that have numerous small noncontiguous file regions that are sparse. In the read case when many small file regions are separated by large holes, data sieving in ROMIO data sieving I/O and ROMIO collective I/O unfortunately accesses the useless data in the file holes. ROMIO POSIX I/O

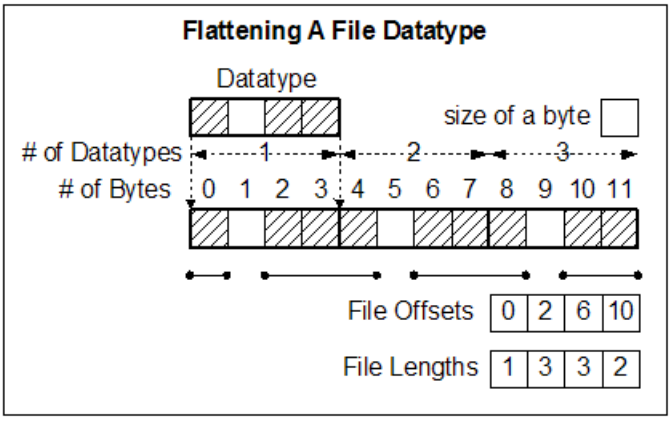

Figure 3.6. Example flattening of a file datatype. File datatypes are converted into lists of file offsets and lengths in order to generate the necessary parameters to use the list I/O interface.

would read only the necessary data, but would potentially generate many I/O request calls to the file system due to the limitation of only being able to describe contiguous data access. In the write case, ROMIO POSIX I/O would also require at least one POSIX write call per file region (more are possible if the memory regions do not align with the file regions), which would be less efficient than using ROMIO list I/O. Collective ROMIO I/O would also be inefficient in the write case for such an access pattern since it also maps into POSIX write calls and may be slower than ROMIO data sieving I/O in some cases since data has to be transferred twice (once to the I/O processors and then again to the processors that requested the data). If file locking was available in the file system, in the write case both ROMIO default I/O and ROMIO collective I/O would have to perform a read-modify-write because of data sieving along with the synchronization overhead.

For access patterns that have numerous small noncontiguous file regions that are very dense, ROMIO data sieving I/O and ROMIO collective I/O will perform very well in the read case because very little unwanted data would be read. Since ROMIO list I/O is split when a maximum number of file regions is reached, if the number of small noncontiguous file

regions is extremely large, ROMIO list I/O will be forced to make many ROMIO list I/O calls, while ROMIO data sieving I/O and ROMIO collective I/O can sieve those I/O calls into a single I/O call if all the file regions fit into the data sieving buffer. ROMIO POSIX I/O would do several orders of magnitude worse than ROMIO list I/O since it always generates more I/O requests than list I/O in noncontiguous access patterns. When writing with a dense access pattern in PVFS1, however, we expect ROMIO list I/O to perform best since the other methods cannot use data sieving when writing. [1]

In the case of an access pattern that is sparse from a processor view but dense in an aggregate view, ROMIO collective I/O will perform best of the four methods when reading data because it can use data sieving very effectively in this scenario. If the number of noncontiguous regions is small, ROMIO list I/O may also be able to perform well. However, if there are many noncontiguous regions, most likely ROMIO list I/O will not perform as well as a ROMIO collective I/O read, again because of the reduced number of I/O calls in the ROMIO collective I/O read. In the write case for PVFS1, list I/O should outperform the other methods because it is more effective than a collection of POSIX write operations.

3.4. Performance Evaluation

To show the effect of the list I/O optimization, we ran a series of tests that access noncontiguous data. Section 3.4.1 discusses our test setup. Our tests were run directly through the file system interface as well through the MPI-IO interface. The file system interface test results are in Section 3.4.2 (Synthetic Tests). The MPI-IO interface test results are in Section 3.4.3 (tiled reader benchmark), Section 3.4.4 (a three-dimensional block benchmark

[1] In other file systems, it is still unclear which implementation method will perform best because while data sieving can reduce I/O calls, the write case involves a read-modify-write and file locking, both of which may present substantial overhead.

from the ROMIO testing suite) and Section 3.4.5 (a simulation of the I/O portion of the FLASH code). In each of the MPI-IO tests, we provide a table summarizing the I/O characteristics of the application for each access method. This information helps in explaining the performance of the methods. The values for data accessed per client and resent data per client are average values across processes.

3.4.1. Machine Configuration

We ran all of our tests on the Chiba City cluster at Argonne National Laboratory [**15**]. The cluster had the following configuration at test time. There are 256 nodes each with dual Pentium III 500 MHz processors, 512 MBytes of RAM, a 9 GByte Quantum Atlas IV SCSI disk, a 100 Mbits/sec Intel EtherExpress Pro fast Ethernet card operating in full-duplex mode, and a 64-bit Myrinet card. We conducted all experiments using fast Ethernet due to some Myrinet instability at the time of experimentation. The nodes are currently using Red Hat 7.1 with kernel 2.4.9 compiled for SMP use. Our I/O configuration included 8 PVFS1 I/O servers with one I/O server doubling as both a manager and an I/O server. PVFS1 files were striped with a stripe size of 16 KBytes. MPICH 1.2.4 was used in all our testing using hints for list I/O, data sieving and collective operations. ROMIO was compiled with PVFS version 1.5.6-pre1. All ROMIO data sieving operations and collective operations were performed using a 4 MByte buffer. All results are the average of three runs.

3.4.2. Synthetic Tests

We created an artificial benchmark in order to test the noncontiguous performance of parallel reads and writes. We set the aggregate data access at 1 GByte in order to access a meaningful amount of data and also to have a baseline comparison. We also kept the I/O nodes constant

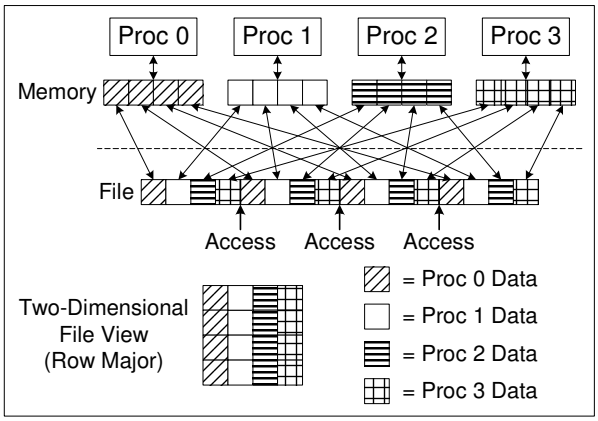

Figure 3.7. Example one-dimensional cyclic access. An entire file stores a two-dimensional array and each processor is in charge of an equal amount of columns. The file view is also flattened into one-dimension.

at 8, with one doubling as both a manager and an I/O daemon. The benchmark varies the number of clients, the number of accesses, and the data access pattern. The data access patterns used in the benchmark are the one-dimensional cyclic and the two-dimensional block-block as shown in Figure 7 and Figure 8, respectively. Increasing the number of accesses further fragments the data access, making it more noncontiguous while preserving the aggregate data size. Changing the number of clients also determines the fragmentation of data. Increasing the number of clients accessing the same amount of data further increases the noncontiguity. The parallel reads and writes were conducted three times, and the I/O request time was averaged over the three runs. Because of the large execution time of multiple I/O in the write cases, however, we ran those tests only once. We decided not to use data sieving I/O with the parallel writes since data sieving requires a read-modify-write and therefore requires synchronization in which only one processor can write at a time in order to ensure the written data will not encounter any race conditions.

3.4.2.1. One-Dimensional Cyclic. This access pattern is a variable-grained, interleaved access, where we merge data from many processes into a single file in a cycling manner. An

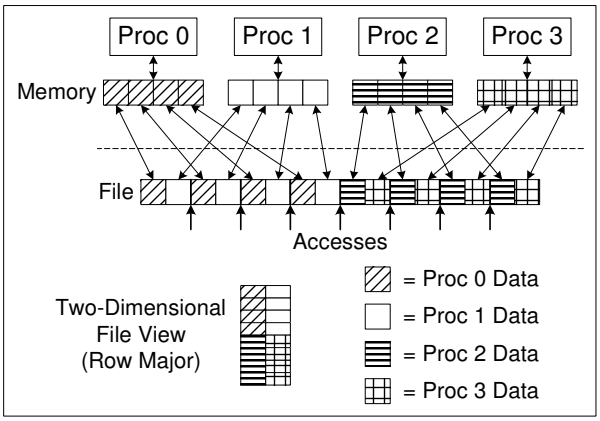

Figure 3.8. Example block-block access. An entire file stores a two-dimensional array of blocks, and each processor is in charge of a single block. The file view has been flattened into 1-dimension.

example of an application that would use this type of access pattern is one in which there is a global two-dimensional array and each processor operates on a region of columns of the array, as shown in Figure 3.7. In these tests we vary the block size while maintaining a constant file size. Thus, a decrease in the block size increases the number of I/O requests for using multiple I/O. List I/O performance is expected to decrease as the accesses increase because of the need for additional requests, but not as rapidly as multiple I/O. Since the actual amount of data read is the same regardless of the number of accesses, we expect data sieving I/O to perform in a near constant time throughout the range of accesses. Note that as we increase the number of clients, data sieving I/O will be reading more and more useless data because the fraction of desired data in the accessed region decreases.

3.4.2.2. Block-Block. This type of access has a data distribution where a two-dimensional global array is partitioned by creating a block for every processor and organizing the blocks as shown in Figure 8. The tile application described later in section 4.4.1 uses an access pattern similar to this one.

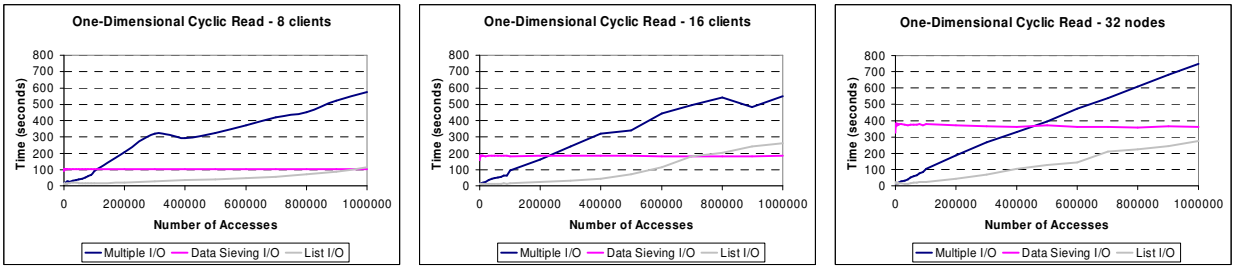

Figure 3.9. One-dimensional cyclic read results with various clients. These results are obtained by using 8-32 clients reading data with the one-dimensional cyclic file access pattern.

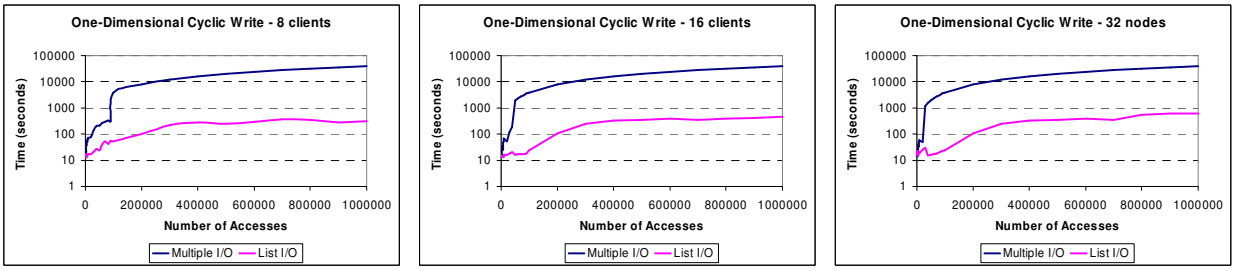

Figure 3.10. One-dimensional cyclic write results with various clients. These results are obtained by using 8-32 clients writing data with the one-dimensional cyclic file access pattern.

3.4.2.3. Experimental Results. For the one-dimensional cyclic access pattern, we expect linear results from both multiple I/O and list I/O since the number of I/O requests will increase linearly with the number of accesses. Data sieving should perform slightly better using the block-block access pattern due to the fact that the useful data is closer, which means accessing less impertinent data.

Figure 3.9 shows that multiple I/O and list I/O scale linearly with the number of accesses. As we increase the number of accesses, the number of contiguous regions also increases, but the size of each contiguous region becomes smaller. Multiple I/O has to increase the number of I/O requests for a larger number of accesses. List I/O must also increase the number of I/O requests for a larger number of accesses, but at a slower rate than multiple I/O. Since

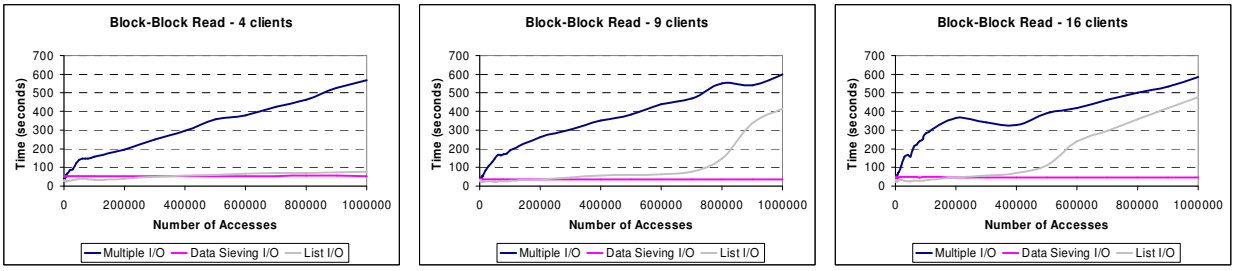

Figure 3.11. Block-block read results with various clients. These results are obtained by using 4-16 clients reading data with the block-block file access pattern.

list I/O can describe 64 file offsets and lengths in a single I/O request, list I/O will not be as affected as multiple I/O by a larger number of accesses. We also notice that data sieving I/O stays fairly constant among any number of accesses for a fixed number of clients. This is because data sieving is moving the same amount of data in all of those cases. Also as expected, the time nearly doubles with data sieving I/O when the clients double due to the doubling of impertinent data read by each client (since each client now only has half as much relevant data in the same overall file region).

The write performance illustrated in Figure 3.10 for the one-dimensional cyclic access pattern is very poor for multiple I/O. In most of the figures we can see that list I/O and multiple I/O have a performance gap of nearly two orders of magnitude. Both list I/O and multiple I/O lose performance as the number of accesses increase but still maintain their two orders of magnitude performance difference.

The results described in Figure 3.11 in the block-block read tests showed the trend expected for multiple I/O and data sieving I/O. Multiple I/O increases at a linear rate with the number of accesses while data sieving I/O remains nearly constant among the range of accesses. List I/O performs unusually in the evaluation of 9 and 16 client block-block reads. When using 4 clients to read a file in a block-block distribution, list I/O scales up

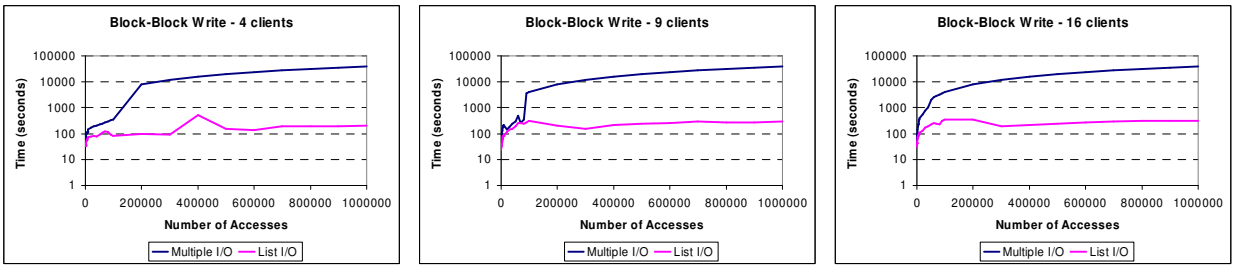

Figure 3.12. Block-block write results with various clients. These results are obtained by using 4-16 clients reading data with the block-block file access pattern.

linearly with the number of accesses. However, we note that in Figure 3.11 for 9 and 16 clients, the list I/O curve sharply turns upward at some number of accesses. For 9 clients, each access is of size (1024*1024*1024 bytes)/(9 clients)/(800,000 accesses) = approximately 149 bytes/access at the turning point. Due to the block-block access pattern, each client heavily uses only a fraction of all the I/O servers, unlike the one-dimensional cyclic access pattern, which distributes a compute nodes I/O load over all the I/O servers. Increasing the number of accesses for the block-block access pattern does not spread out the load as in the one-dimensional cyclic case. We observed the greater increase of list I/O with the number of accesses in the block-block access pattern at about 150 bytes/access for both the 9 client and 16 client cases.

A comparison between the 16 node cases in Figure 3.9 and Figure 3.10 show that the data sieving I/O times are reduced. The reason is that the data sieving I/O accesses less irrelevant data using the block-block access pattern. Figure 3.12 shows that the block-block write results are similar to the one-dimensional cyclic write results for multiple I/O and list I/O. As the number of accesses increases, multiple I/O and list I/O run times increase while maintaining the two orders of magnitude difference. The trend follows the results of the writes of the one-dimensional cyclic case.

	Desired Data per Client	Data Accessed per Client	# of I/O Ops per Client	Resent Data per Client	File Region Size Accessed
POSIX I/O	2.25 MB	2.25 MB	768	—	3 KB
Data Sieving I/O	2.25 MB	5.56 MB	2	—	4 MB
Two-Phase I/O	2.25 MB	1.70 MB	1	1.50 MB	1.70 MB
List I/O	2.25 MB	2.25 MB	12	—	3 KB
Datatype I/O	2.25 MB	2.25 MB	1	—	3 KB

Table 3.1. I/O characteristics of the tile reader benchmark.

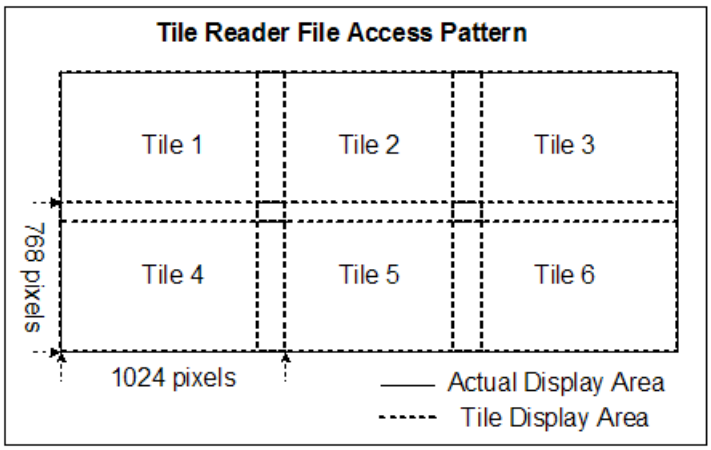

Figure 3.13. Tile reader file access pattern. Each processor is in charge of reading the data from a display file into its own local display, also known as a tile. This results in a noncontiguous file access pattern.

3.4.3. Tile Reader Benchmark

Tiled visualization code is used to study the effectiveness of commodity based graphics systems in creating parallel and distributed visualization tools. The amount of detail in current visualization methods requires more than a single desktop monitor can resolve. Using two-dimensional displays to visualize large datasets or real-time simulation is important for high performance applications. Our version of the tiled visualization code, the tile reader benchmark, uses multiple compute nodes, with each compute node taking high-resolution display frames and reading only the visualization data necessary for its own display. We use six compute nodes for our testing, which mimics the display size of the full application.

The six compute nodes are arranged in the 3 x 2 display shown in Figure 3.13, each with a resolution of 1024 x 768 with 24-bit color. In order to hide the merging of display edges, there is a 270-pixel horizontal overlap and a 128-pixel vertical overlap. Each frame has a file size of about 10.2 MBytes. A set of 100 frames is read for a total of 1.02 GBytes.

Table 3.1 provides a summary of the I/O characteristics of this benchmark for each method of access. We expect ROMIO list I/O to perform best with this access pattern because the noncontiguous file regions are large and there are only 768 noncontiguous file regions. This means that 768 / 128 = 6 I/O requests per processor, which is not too burdensome. ROMIO POSIX I/O will have to make all 768 I/O requests per processor. The data layout is sparse enough to lessen the performance improvements of data sieving in ROMIO data sieving I/O or ROMIO collective I/O. Data sieving ROMIO I/O will perform marginally since only 2.25 MBytes of 5.56 MBytes of the data accessed per processor is useful. Collective ROMIO I/O will be able to use data sieving effectively since no data will be wasted. However the overhead of the second phase of redistribution will make it slower than ROMIO data sieving I/O since it must pass 4.5 MBytes of data to other processors.

We can see in Figure 3.14 that ROMIO list I/O outperforms the other ROMIO I/O methods in both the uncached and cached read cases. All of the methods perform better in the cached case due to getting data from memory on the I/O servers instead of the disk. While ROMIO collective I/O views the file access pattern as contiguous from an aggregate standpoint, the file regions are too large to enable it to overcome the overhead of reading data once from file and then sending the data to the requesting nodes. In fact, the overhead of file redistribution causes it to fall behind ROMIO data sieving I/O. The overhead of 768 I/O calls to the file system caused ROMIO POSIX I/O to lag far behind the other implementations. Other noncontiguous reading benchmarks perform with similar higher performance trends in

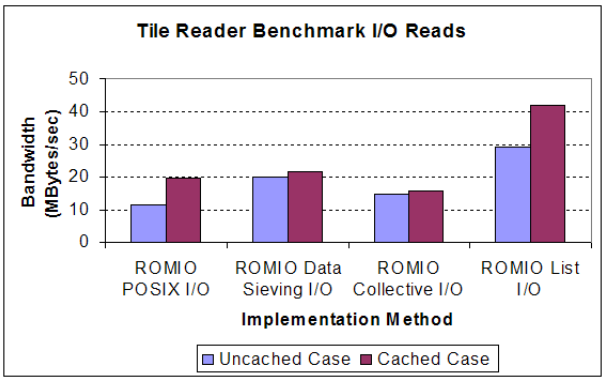

Figure 3.14. Tile reader benchmark results.

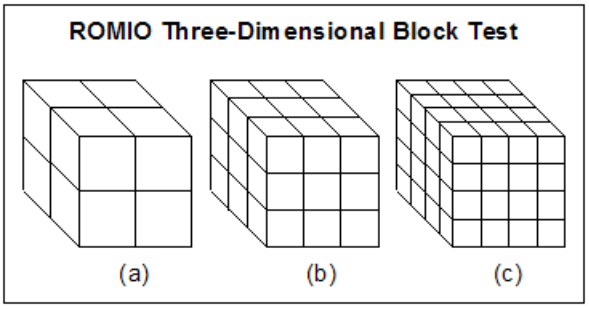

Figure 3.15. Three-dimensional block test access pattern. The access pattern for 8, 27, and 64 processors is shown in (a), (b), and (c), respectively.

cached read cases versus uncached read cases, so we only focus on uncached noncontiguous read performance in the other noncontiguous read tests.

3.4.4. ROMIO Three-Dimensional Block Test

The ROMIO test suite consists of a number of correctness and performance tests. We chose the coll_perf.c test from this suite to compare our methods of noncontiguous data access. The coll_perf.c test measures the I/O bandwidth for both reading and writing to a file with a file access pattern of a three-dimensional block-distributed array. The three-dimensional array, shown graphically in Figure 3.15, has dimensions 600 x 600 x 600 with an element size of an integer (4 bytes).

	Desired Data per Client	Data Accessed per Client	# of I/O Ops per Client	Resent Data per Client	File Region Size Accessed
8 Clients					
POSIX I/O	103 MB	103 MB	90,000	—	1200 B
Data Sieving I/O	103 MB	412 MB	103	—	4 MB
Two-Phase I/O	103 MB	103 MB	26	77.2 MB	4 MB
List I/O	103 MB	103 MB	1408	—	1200 B
Datatype I/O	103 MB	103 MB	1	—	1200 B
27 Clients					
POSIX I/O	30.5 MB	30.5 MB	40,000	—	800 B
Data Sieving I/O	30.5 MB	274.7 MB	69	—	4 MB
Two-Phase I/O	30.5 MB	30.5 MB	8	27.1 MB	4 MB
List I/O	30.5 MB	30.5 MB	626	—	800 B
Datatype I/O	30.5 MB	30.5 MB	1	—	800 B
64 Clients					
POSIX I/O	12.9 MB	12.9 MB	22,500	—	300 B
Data Sieving I/O	12.9 MB	206.0 MB	52	—	4 MB
Two-Phase I/O	12.9 MB	12.9 MB	4	12.1 MB	4 MB
List I/O	12.9 MB	12.9 MB	352	—	300 B
Datatype I/O	12.9 MB	12.9 MB	1	—	300 B

Table 3.2. I/O characteristics of the ROMIO three-dimensional block test.

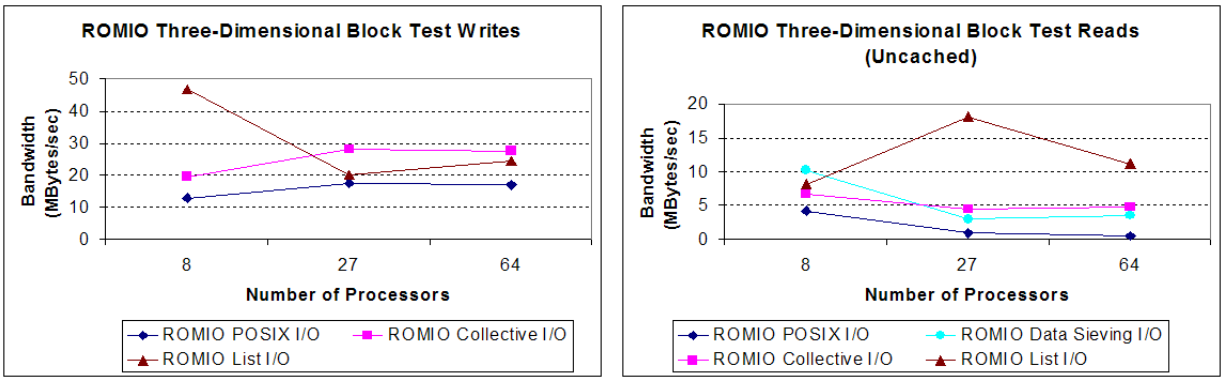

Figure 3.16. Three-dimensional block test results.

Table 3.2 summarizes the I/O characteristics of this test for the four I/O methods and three numbers of processes. Due to the three-dimensional block access pattern, we expected that increasing the number of processors would have a large effect on the performance.

For example when we used 8 processors, data sieving operations would waste 3/4 of the data accessed, roughly 309 MBytes. When we used 64 processors, data sieving operations would waste 15/16 of the data accessed, roughly 193.1 MBytes. When considering ROMIO collective I/O, no file data would be wasted, but the redistribution size in both the read and write cases would be the same as the wasted file data in ROMIO data sieving I/O. When using 8 or 64 processors, 3/4 or 15/16, respectively, of the data accessed by I/O processors would be redistributed to other processors. ROMIO POSIX I/O will have to face 90,000 accesses per processor with 8 processors, 40,000 access per processor with 27 processors, and 22,500 access with 64 processors. ROMIO list I/O faces a reduced number of accesses versus ROMIO POSIX I/O, but must contend with many more I/O operations per client versus data sieving methods.

Figure 3.16 shows the results of the three-dimensional block test. In the write case, we see ROMIO list I/O take a big lead over the other methods and then drop significantly with 27 processors and 64 processes. We can attribute this slowdown to a smaller contiguous file region size and an increased number of system-wide I/O requests to the I/O servers. ROMIO POSIX I/O performs very poorly due to even more I/O requests than ROMIO list I/O in all cases. ROMIO collective I/O sees some gains in this test with more processors since it performs large contiguous writes with the assigned I/O processors instead of small noncontiguous writes like the other methods. In the read case, ROMIO list I/O results improve from 8 to 27 processors due to having more clients outweighing the effect of having smaller accesses. However, at 64 processors, the overhead of increased I/O requests and smaller file regions has lessened performance. ROMIO POSIX I/O performs worse with an increased number of processors due to 128 times more I/O requests than ROMIO list I/O. ROMIO data sieving I/O also performs worse with more processors since it accesses 206

	Desired Data per Client	Data Accessed per Client	# of I/O Ops per Client	Resent Data per Client	File Region Size Accessed
POSIX I/O	7.50 MB	7.50 MB	983,040	—	8 B
Data Sieving I/O	—	—	—	—	—
Two-Phase I/O	7.50 MB	7.50 MB	2	7.5 MB * $\frac{n-1}{n}$	4 MB
List I/O	7.50 MB	7.50 MB	15,360	—	4 KB
Datatype I/O	7.50 MB	7.50 MB	1	—	4 KB

Table 3.3. I/O characteristics of the FLASH I/O simulation (n is the # of clients).

MBytes per client while using only 12.9 MBytes. ROMIO collective I/O suffers from the heavy redistribution cost.

3.4.5. FLASH I/O Simulation

The FLASH code is an adaptive mesh refinement (AMR [**64**] application that solves fully compressible, reactive hydrodynamic equations, developed mainly for the study of nuclear flashes on neutron stars and white dwarfs [**31**]. The I/O performance for FLASH determines how often checkpointing may be performed, so I/O performance is critical. The actual FLASH code uses HDF5 for writing checkpoints, but the organization of variables in the file is the same in our simulation. The element data in every block on every processor is written to file by using noncontiguous MPI Datatypes. The access pattern of the FLASH code is noncontiguous both in memory and in file, making it a challenging application for parallel I/O systems. The FLASH memory datatype, viewable in Figure 3.17, consists of 80 FLASH three-dimensional blocks, or cells in the refined mesh, on each processor. Every block contains an inner data block surrounded by guard cells. Each of these data elements has 24 variables associated with it. Every processor writes these blocks to a file in a manner such that the file appears as the data for variable 0, then the data for variable 1, all the way up to variable 23 as shown in Figure 3.18. Within each variable in file, there exist 80

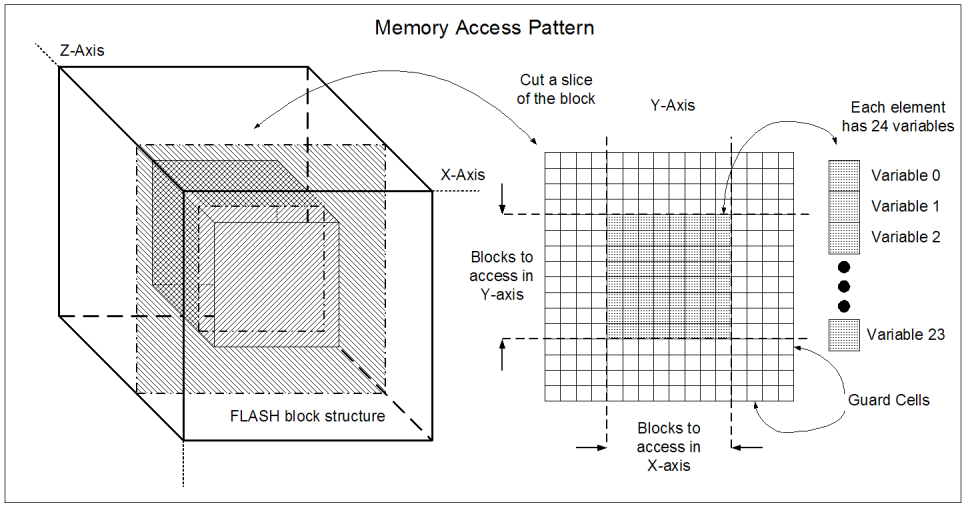

Figure 3.17. FLASH memory datatype. Each computing processor contains 80 blocks, so as we scale up the number of computing processors, we linearly increase the dataset size.

blocks, each of these blocks containing all the FLASH blocks from every processor. Since every processor writes 80 FLASH blocks to file, as we increase the number of clients, the dataset size increases linearly as well. Every processor adds 7 MBytes to the file, so the dataset ranges between 14 MBytes (2 clients) to 448 MBytes (64 clients).

Table 3.3 summarizes the I/O characteristics of this benchmark for the access methods tested, with n referring to the number of compute processors (or clients). Note that because this is a write benchmark and PVFS1 does not have file locking, the data sieving method was not tested. In our testing we vary the number of clients from 2 to 64. Each contiguous memory region is the size of a double (8 bytes). In file, however, the contiguous regions are (8 x-elements)*(8 y-elements)*(8 z-elements)*(sizeofdouble) = 4096 bytes. The FLASH I/O code is worst for the ROMIO POSIX I/O approach since the noncontiguous file region access pattern is sparse. The number of I/O requests for ROMIO default I/O = (80 blocks)*(8 x-elements)*(8 y-elements)*(8 z-elements)*(24 variables) = 983,040 I/O calls per processor. Our implementation of ROMIO list I/O can do a little better since ROMIO list I/O can

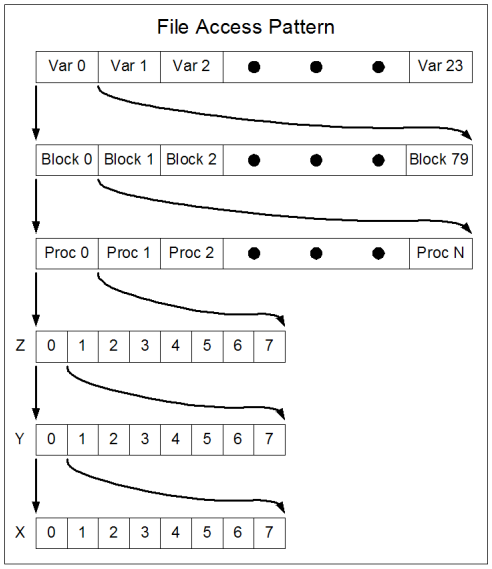

Figure 3.18. FLASH file datatype. This figure describes the hierarchy of the file datatype. At the highest level of the hierarchy, variables are contiguous. Within every variable, there are all the FLASH blocks from all the processors.

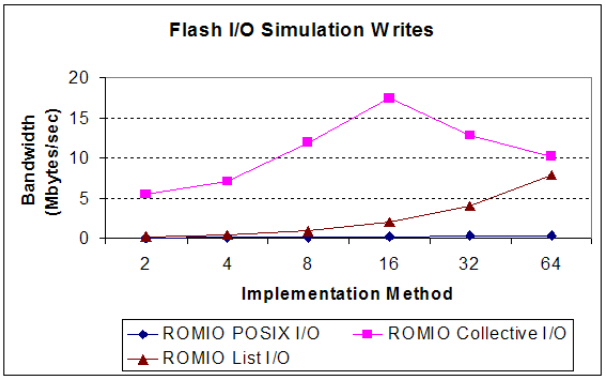

Figure 3.19. Results of the FLASH I/O benchmark with 2 - 64 processors. Collective I/O performs exceptionally well due to the aggregate contiguous file access pattern. The overhead of exchanging data is minimal compared to the I/O time.

describe noncontiguous file regions in a single I/O calls. However, since our maximum limit of file regions was set at 128 for this test, 983,040 / 128 = 7,680 I/O calls per processor are still required. The FLASH I/O benchmark presents a good opportunity to use ROMIO

collective I/O since the aggregate view of the file is actually a contiguous region of file. ROMIO collective I/O only needs to perform I/O calls = (Aggregate data size)/((number of processors)*(Buffer size)) = (7 MBytes * N procs)/(N procs)*(4 MBytes) = 7/4 rounded up to 2 I/O calls per processor. While, $\frac{n-1}{n}$ of the data must be redistributed, the savings in I/O calls is significant enough to overcome the redistribution overhead. We expect ROMIO collective I/O to perform best in this benchmark.

Figure 3.19 shows that ROMIO collective I/O works more efficiently than the other ROMIO I/O methods for numerous noncontiguous file regions that appear contiguous from a global standpoint. ROMIO POSIX I/O suffers from the immense overhead of 983,040 I/O calls per processor while list I/O does roughly two magnitudes better due to only having 7,680 I/O calls per processor. As we increase the number of processors, the dataset size increases. We see that both ROMIO POSIX I/O and ROMIO list I/O increase bandwidth with more processors, but ROMIO collective I/O starts to fall with larger dataset sizes between 16 to 32 nodes. For the two processor case of 14 MBytes, in ROMIO collective I/O, 4 contiguous writes could cover the entire file, but at 448 MBytes, 112 contiguous writes are necessary. Even though it is only 2 contiguous writes per processor in all cases, redistribution becomes more and more expensive as we increase the aggregate data size. For example, with 64 processors, only 1/64 of the data read from the file system is used by the processor doing I/O, the other 63/64, 12.1 MBytes is sent to other processors.

3.5. Summary

List I/O is an efficient interface for noncontiguous data access both through MPI-IO and directly through the system interface. We have seen performance improvements in both the

tile reader benchmark as well as the noncontiguous ROMIO benchmark. In many noncontiguous cases, ROMIO list I/O can outperform current ROMIO methods, but there are cases, such as the FLASH I/O benchmark in Section 3.4.5 where the number of noncontiguous file regions grows too large for ROMIO list I/O to reduce linearly.

While the list I/O interface is a major step in support for efficient noncontiguous I/O access, it is not optimal. Particularly in the case of MPI-IO, noncontiguous accesses often have regular patterns in the file, and these patterns are described concisely in the datatypes passed to the MPI I/O call. The list I/O interface, as described here, loses these regular patterns, instead flattening them into potentially large lists of contiguous regions. The next chapter shows how these descriptions of regular patterns can be retained and passed directly to the parallel file system. This capability can have a significant impact on the size of the I/O request, which is extremely important in cases where many small noncontiguous regions are being accessed.

CHAPTER 4

Datatype I/O

Datatype I/O is an effort to address the deficiencies seen in the list I/O interface when faced with accesses that are made up of many small regions, particularly ones that exhibit some degree of regularity. Datatype I/O borrows from the datatype concept that has proven invaluable for both message passing and I/O in MPI applications. The constructors used in MPI datatypes allow for concise descriptions of the regular, noncontiguous data patterns seen in many scientific applications, such as extracting a row from a two-dimensional dataset. This chapter describes the datatype I/O interface in Section 4.1. Section 4.2 details our implementation of datatype I/O in PVFS1 and ROMIO. Section 4.3 analytically compares the performance of each of the noncontiguous methods. Section 4.4 shows the results of the datatype I/O method several application benchmarks. Section 4.5 summarizes the work in this chapter.

4.1. Interface

The datatype I/O interface, shown in Figure 4.1, replaces the lists of I/O regions seen in the list I/O interface with an address, count, and datatype for memory and a displacement,

```
int dtype_read(int fd, void  *mem_addr, int mem_dtype_count, dtype *mem_dtype,
               int file_dtype_disp, int offset_into_dtype, dtype *file_dtype)

int dtype_write(int fd, void  *mem_addr, int mem_dtype_count, dtype *mem_dtype,
                int file_dtype_disp, int offset_into_dtype, dtype *file_dtype)
```

Figure 4.1. Datatype I/O prototypes.

datatype, and offset into the datatype for file. These correspond directly to the address, count, datatype, and offset into the file view passed into an MPI-IO call and the displacement and file view datatype previously defined for the file. The datatype I/O interface is not meant to be used by application programmers; it is an interface specifically for use by I/O library developers. Helper routines are used to convert MPI datatypes into the format used by the datatype I/O functions. A full-featured implementation of datatype I/O would

- maintain a concise datatype representation locally and avoid datatype flattening,
- use this concise datatype representation when describing accesses, and
- service accesses using a system that processes this representation directly.

Our prototype implementation of datatype I/O was written as an extension to PVFS1. The ROMIO MPI-IO implementation was likewise modified to use datatype I/O calls for PVFS1 file system operations.

We emphasize that while we present this work in the context of MPI-IO and MPI datatypes, nothing precludes our using the same approach to directly describe datatypes from other APIs, such as HDF5 hyperslabs; in fact, because HDF5 uses MPI-IO it can benefit from this improvement without code changes.

4.2. Datatype I/O Implementation in PVFS1 and ROMIO

Our datatype I/O prototype builds on the datatype processing component in MPICH2 [**78**]. Three key characteristics of this implementation make it ideal for reuse in this role:

- Simplified type representation (over MPI datatypes)
- Support for partial processing of datatypes
- Separation of type parsing from action to perform on data

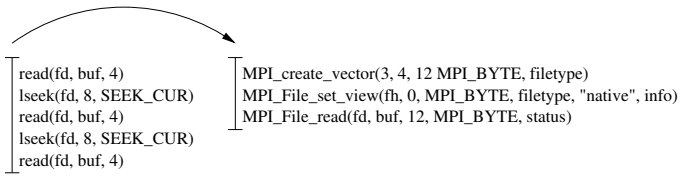

Figure 4.2. Example tile reader file access pattern conversion. (A) shows how we convert a struct datatype into an indexed dataloop for performance optimization. This conversion eliminates the need for the MPI_LB and MPI_UB dataloops, making the dataloop representation smaller. (B) is an example of loop fusion in which we can merge datatypes into a single dataloop. The contig and named dataloops can be sufficiently described by the vector dataloop above them, eliminating the need for them.

Types are described by combining a concise set of descriptors called *dataloops*. Dataloops can be of five types: contig, vector, blockindexed, indexed, and struct [**35**]. These five types capture the maximum amount of regularity possible, keeping the representation concise. At the same time these are sufficient to describe the entire range of MPI datatypes. Simplifying the set of descriptors aids greatly in implementing support for fast datatype processing because it reduces the number of cases that the processing code must handle. The type extent is retained in this representation (a general concept) while the MPI-specific LB and UB values are eliminated. This simplification has the added benefit of allowing resized type processing with no additional overhead in our representation. We use dataloops as the native representation of types in our PVFS1 implementation.

The MPICH2 datatype component provides the functions necessary to process dataloop representations [**78**]. We provide functions to convert MPI datatypes into dataloops and functions that are called during processing to create the offset-length pairs we need to build the PVFS1 job and access structures. Additionally we provide functionality for shipping dataloops as part of I/O requests. In our prototype, MPI datatypes are converted to dataloops by a recursive process built by using the functions `MPI_Type_get_envelope` and

`MPI_Type_get_contents`. By using these MPI functions, we can ensure the portability of our datatype I/O method across different MPI implementations.

An optimization our conversion process employs is the reduction of the size of the access pattern representation. By using loop fusion as well as a struct-to-indexed datatype conversion, we can reduce the amount of data transferred over the network. Loop fusion is the conversion of excessively created dataloops into more concise dataloops, reducing the size of the overall dataloop structure while maintaining the desired access pattern. The struct-to-indexed datatype conversion we employ converts a struct dataloop (which may have multiple underlying dataloops) into an indexed dataloop, which has only one underlying dataloop), generally resulting in a smaller access pattern representation. Both of these optimizations are visualized in an example file access pattern conversion from the tile reader test in Section 3.4.3 in Figure 4.2. The resulting dataloop representation is passed into the datatype I/O calls and from there sent to the relevant I/O servers. The dataloops are converted into the job and access structures on servers and clients side to create the traditional PVFS1 job and access structures. Figure 4.3 outlines this process. PVFS1-specific functions for creating these offset-length pairs are passed to the dataloop processing component. These functions are written to efficiently convert contiguous, vector, and indexed dataloops into offset-length pairs, and they include optimizations to coalesce adjacent regions. The partial processing capabilities of the datatype processing component are used to limit the overhead of storing the intermediate offset-length pairs that are created through dataloop processing.

This is only a partial implementation of the datatype I/O approach; a complete approach would avoid creating of lists of regions on server and client. However, we will show that even without this final capability, our prototype exhibits clear performance benefits over the other approaches.

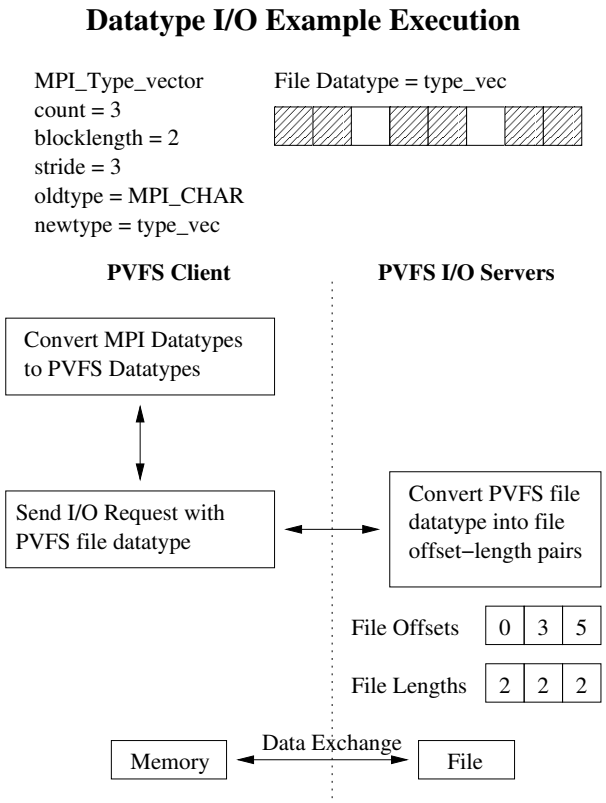

Figure 4.3. Example datatype I/O call. Since file datatype are broken into file offset-length pairs at the I/O servers, the number of I/O requests is dramatically reduced for regular access patterns.

Because the MPI datatypes are converted into dataloops at every MPI I/O operation, we expect there to be slightly higher overhead in the local portion of servicing these operations in comparison with list I/O. On the other hand, because we are concisely describing these types, we expect to see significantly less time spent moving the I/O description across the network. Future optimizations could cache these dataloop representations on clients and servers to ameliorate this overhead.

	Desired Data per Client	Data Accessed per Client	# of I/O Ops per Client	Resent Data per Client	File Region Size Accessed
POSIX I/O	n	n	$\geq$*fr count*	—	$\leq$*fr size*
Data Sieving I/O	n	*file ext*	$\lceil \frac{file\ ext}{ds\ buffer} \rceil$	—	$\leq$*ds buffer*
Two-Phase I/O	n	$\frac{agg\ file\ ext}{\#\ of\ I/O\ aggs}$	$\lceil \frac{agg\ file\ ext}{\#\ of\ I/O\ aggs * tp\ buffer} \rceil$	*ap depend*	$\leq$*tp buffer*
List I/O	n	n	$\lceil \frac{fr\ count}{fr\ per\ access} \rceil$	—	*fr size*
Datatype I/O	n	n	1	—	*fr size*

Table 4.1. I/O characteristics comparison.

4.3. Noncontiguous I/O Methods Comparison

The noncontiguous I/O methods described in the previous section have several important performance characteristics that will be discussed in this section. We will focus on the characteristics that have the greatest performance impact. In Table 4.1 we show how the major I/O characteristics compare.

We used the following variables to define this comparison: n = number of bytes, *file ext* = the extent of the noncontiguous file regions (from first byte of the first file region to the last byte of the last file region) for a single client, *agg file ext* = the extent of the noncontiguous file regions (from first byte of the first file region to the last byte of the last file region) for the aggregate I/O access pattern, *ds buffer* = the size of the data sieving I/O buffer, *tp buffer* = the size of the two-phase I/O buffer, *fr count* = number of file regions, *fr per access* = the maximum number of file regions a list I/O operation can support before breaking up into multiple list I/O operations. *# of I/O aggs* = the number of I/O aggregators *ap depend* = access pattern dependent, and *fr size* = the size of a file regions (we assume same size file regions for simplicity).

We note that in the performance characteristics of the data sieving I/O and the two-phase I/O there a small approximation is made with respect to the filing of the intermediary buffer. The data sieving I/O buffer is the client's memory region used to do the initial access of

data from the I/O system. The two-phase I/O buffer is the memory of the I/O aggregators that access data from the I/O system. Any holes between noncontiguous file regions that are past the last file region in the buffer can be ignored when refilling the buffer in the next iteration. The frequency of this hole being removed from being accessed decreases as the buffer is larger. Therefore, to simplify our calculations, we are essentially approximating an infinitely large intermediary buffer.

4.3.1. Desired Data per Client

Each client needs a certain amount of data before it can return completion of its I/O operation. The desired data per client is this final data size. The desired data per client will be the same for each of the noncontiguous I/O methods for a given I/O access pattern.

4.3.2. Data Accessed per Client

The client must acquire data from the I/O storage system in order to service its request. For POSIX I/O, list I/O, and datatype I/O, the data accessed per client is equal to the desired data per client. For data sieving I/O, the data accessed per client is the sum of the desired data per client and the holes between the file regions. For two-phase I/O, the data accessed per client is the amount of data that is actually exchanged between I/O aggregators (all compute nodes in our tests) and the I/O servers; it does not include the cost of the data that is resent between I/O aggregators and compute nodes. The implementation of two-phase I/O leads to the actual amount of data accessed per client being based on *agg file ext* divided by the number of I/O aggregators.

4.3.3. I/O Operations per Client

For POSIX I/O, the number of I/O operations per client depends on the number of noncontiguous file regions. For every file region, POSIX I/O requires at least one I/O operation. A mismatch between memory regions and file regions can cause multiple contiguous I/O operations to service a single file region. Therefore, in POSIX I/O, the number of I/O operations per client is greater than or equal to the number of file regions in the access pattern. For data sieving I/O, the number of I/O operations is approximately the ceiling of the *file ext* divided by size of the data sieving buffer. For two-phase I/O, the number of I/O operations per I/O aggregator is equal to the ceiling of the data accessed per client divided by the size of the two-phase buffer. For list I/O, because the implementation of list I/O limits the number of file offset-length pairs that can be passed with a single I/O operation, the number of I/O operations per client is equal to the total number of file regions divided by the number of file offset-length pairs allowed per I/O operation. For datatype I/O, the number of I/O operations is always one per MPI-IO operation because the datatype I/O request includes a derived datatype that can correspond to any MPI-IO datatype.

4.3.4. Resent Data per Client

Only two-phase I/O actually resends data between clients. These transfers are not as costly as the data accessed per client because they are from the memory of the I/O aggregator to the memory of the client; hence it is a memory-memory network transfer instead of a disk-memory network transfer. Clients using data sieving I/O access regions of the data sieving buffer in the same manner, but this is done locally on the same processor, so we do not include this overhead in this section. We also do not define a formula for this variable because it

is highly dependent on the access pattern. For example, if each aggregator is accessing data in its own aggregator file region, then there no data is resent. If each aggregator is accessing data in another aggregators file region, then all the data that is accessed is resent. The amount of resent data per client is best calculated based on the access pattern. In Section 4.4, we will determine the resent data size of the I/O access patterns from each of the benchmarks.

4.3.5. File Region Size Accessed

For list I/O and datatype I/O, the file region size is the same as specified in the access pattern. For POSIX I/O, the file region size is also the same as specified in the access pattern except when the memory regions and file regions do not align. If the regions do not align, the I/O operations can be for smaller sizes than the access pattern file region. For data sieving I/O, the file region size is equal to the minimum of the data sieving buffer size or the extent of the remaining file regions. For two-phase I/O, the file region size is equal to the minimum of the two-phase I/O buffer or the extent of the remaining aggregator region extent. When we use this I/O characteristic in Section 4.4, for simplicity we fill in the most commonly used file region size accessed.

4.4. Performance Evaluation

To evaluate the performance of the datatype I/O optimization against the other noncontiguous I/O methods, we ran a series of noncontiguous MPI-IO tests, including a tile reader benchmark, a three-dimensional block access test, and the FLASH I/O simulation. For each test we provide a table summarizing the I/O characteristics of the access pattern using the metrics from Section 4.3 for each of the noncontiguous I/O methods.

4.4.1. Benchmark Configuration

Our results were gathered on Chiba City at Argonne National Laboratory [**15**]. Chiba City has 256 nodes available with dual Pentium III processors, 512 MBytes of RAM, a single 9 GByte Quantum Atlas IV SCSI drive, and a 100 Bits/sec Intel EtherExpress Pro fast-Ethernet card operating in full-duplex mode. Each node uses Red Hat 7.3 with kernel 2.4.21-rc1 compiled for SMP use. Our PVFS1 server configuration for all test cases included 16 I/O servers (one also doubled as a metadata server). PVFS1 files were created with a 64 KByte strip size (1 MByte stripes across all servers). In the tile reader tests we allocate one process per node because so few nodes are involved. In the other two cases we allocate two processes per node.

Our prototype was built using the ROMIO version 1.2.4 and PVFS version 1.5.5. All data sieving I/O and two-phase I/O operations were conducted with a 4 MByte buffer size (ROMIO default size). Our results are the average of three test runs. All write test times include the time for the MPI_File_sync() command to complete besides the normal write I/O time. All read tests are uncached. We added these constraints to our testing environment to avoid simply testing network bandwidth. Instead, we want to examine the performance of our file system optimizations in conjunction with the storage system hard drive performance.

All read benchmarks are conducted with POSIX I/O, data sieving I/O, two-phase I/O, list I/O, and datatype I/O. All write benchmarks are conducted with POSIX I/O, two-phase I/O, list I/O, and datatype I/O. ROMIO can support write operations with data sieving I/O only if file locking is supported by the underlying file system. Since PVFS1 does not support file locking, we cannot perform data sieving writes on PVFS1. We note, however, that for file systems that do allow file locking, data sieving performance for writes will have

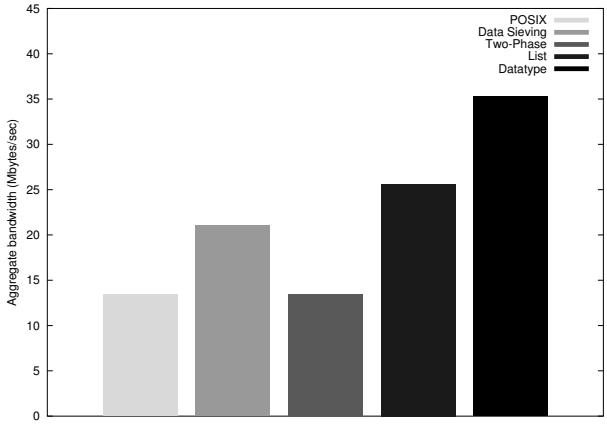

Figure 4.4. Tile reader performance results.

worse performance than data sieving reads for the same access pattern. Data sieving writes perform the same data movement from file system into the data sieving buffer at the client and then data movement from data sieving buffer to memory buffer, but they also have to write this data back, thereby doubling the network data transfer of a data sieving read. Also, locking the modified regions can cause serialization of I/O requests for overlapping requests, another serious overhead. On the other hand, the MPI-IO consistency semantics do allow us to perform a read-modify-write during collective I/O. Thus, an approach similar to data sieving is used in the two-phase I/O write case.

4.4.2. Tile Reader Benchmark

We used the same tile reader benchmark as described in Section 3.4.3 for our tests. We can see in Figure 4.4 that datatype I/O is the clear winner in terms of performance for this benchmark, 37% faster than list I/O. The characteristics of the resulting reads using the optimizations tested are shown in Table 3.1. The datatype I/O result is due to the combination of a single, concise I/O operation, no extra file data being transferred, and no

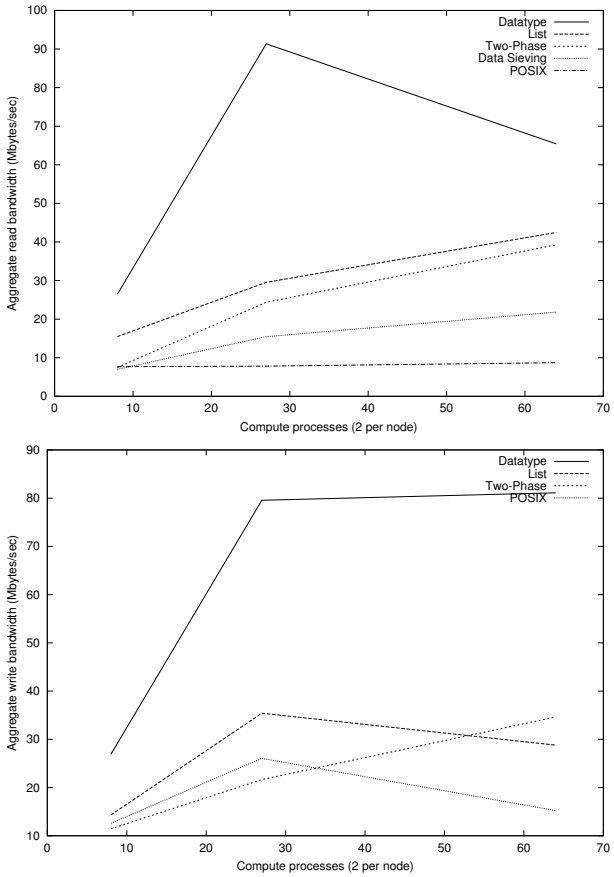

Figure 4.5. Three-dimensional block read and write performance.

data passing over the network more than once. In contrast, POSIX I/O requires 768 read operations, data sieving requires more than twice as much data to be read as is desired, two-phase I/O requires resending about 88% of the data read, and list I/O sends a list of 768 offset-length pairs as part of the requests (9 KBytes of total data in I/O requests from each client).

4.4.3. ROMIO Three-Dimensional Block Test

This benchmark, already described in Section 3.4.4, produced the data in Figure 4.5. Again, datatype I/O is the clear performance winner; peak performance is more than double that

of the next-best approach. Of note is the unusual drop in performance in the read case as number of processes increases. We believe that this is due to the increased overhead of offset-length list processing on the server side. Because the servers are the source of data, and clients are operating on a contiguous region of memory, any delays caused by list processing will directly impact performance. On the other hand, in the write case the servers are data sinks. Buffering in the TCP stack helps hide this inefficiency, although it might appear at larger numbers of processes. This overhead is not visible in the list I/O results because the number of I/O operations and the size of the I/O requests obscure this effect. A full-featured datatype I/O implementation that operated directly on the dataloop representation would likely not exhibit this behavior.

4.4.4. FLASH I/O Simulation

The FLASH benchmark is the same as the one found in Section 3.4.5. Since every processor writes 80 FLASH blocks to file, as we increase the number of clients, the dataset size increases linearly as well. Every processor adds 7 MBytes to the file, so the dataset ranges between 14 MBytes (at 2 clients) to 896 MBytes (at 128 clients).

Table 3.3 provides the I/O characteristics of the test using the available optimizations. Figure 4.6 shows the results of these tests. This is the first test in which the memory datatype is noncontiguous; thus it is the first time that the overhead of list processing might affect the clients. We see this in both the list I/O and datatype I/O cases; both perform poorly at small numbers of clients. As the number of clients increases, the clients are eventually able to feed the servers adequately. At 96 processes, datatype I/O performance rises to nearly 40 MBytes/sec, 37% faster than two-phase. This trend continues at higher numbers of processes. We would expect that a datatype I/O system that operated directly

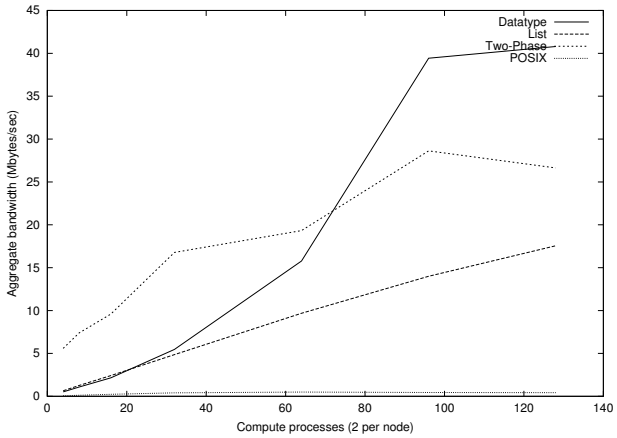

Figure 4.6. FLASH I/O performance.

on the dataloop representation would allow clients to more effectively push data to servers, resulting in improved performance at smaller numbers of clients. List I/O, because of the size and number of I/O requests, is not able to overtake two-phase for the tested numbers of processes. POSIX I/O is nearly unusable in this benchmark because it has to address 983,040 I/O operations of 8 bytes each to service this access pattern.

This case shows that two-phase I/O still has a place as an I/O optimization. Because data is not wasted and I/O accesses are large, two-phase I/O is able to provide good performance despite moving the majority of the data over the network twice.

4.5. Summary

In conclusion, datatype I/O provides the opportunity for extremely efficient processing of structured, independent I/O requests. Our tests show that this approach outperforms both list I/O and data sieving I/O in virtually all situations. Further, it supplants two-phase I/O as the preferred optimization in many cases as well. Datatype I/O in conjunction with the two-phase collective I/O optimization makes a strong MPI-IO optimization suite. We note

that in almost every case POSIX I/O alone would result in a nearly unusable system from the performance perspective; these optimizations are a necessary part of scientific parallel I/O.

Further optimization of the approach can be provided in ROMIO as well. Caching the dataloop representations of types locally would be one way to improve datatype I/O. Also, leveraging datatype I/O as the internal I/O method underneath two-phase I/O would likely boost performance of the collective operations further.

CHAPTER 5

Performance Analysis of Access Pattern Characteristics

The previous chapters have described traditional noncontiguous I/O methods as well as our new list I/O and datatype I/O methods. While application I/O benchmarks show practical performance improvements in today's environment, it is also useful to study basic access pattern parameters to understand how and why the various noncontiguous I/O methods have different performance characteristics. In this chapter, Section 5.1 discusses the basic access pattern characteristics. Section 5.2 details our reimplementation of list I/O and datatype I/O in PVFS2. Section 5.3 describes the HPIO benchmark that we use for our study of I/O characteristics. Section 5.4 provides the results of our study on I/O characteristics and Section 5.5 concludes this chapter with a collection of I/O performance guidelines.

5.1. I/O Characteristics Discussion

There are three major I/O access pattern characteristics that seriously affect noncontiguous I/O performance:

- **Region Count** - For some methods, this can cause an increase in the amount of data sent from the clients to the I/O system over the network. When using POSIX I/O, for example, increasing the region count increases the number of I/O requests necessary to service a noncontiguous I/O call. However, some other methods such as datatype I/O are generally expected to be unaffected by this parameter since

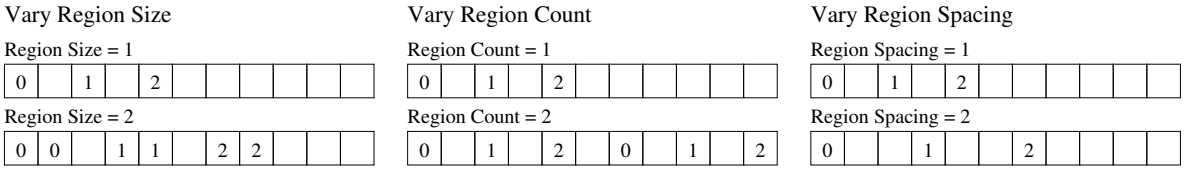

Figure 5.1. An example of how an access pattern is created from the HPIO parameters.

increasing the region count does not change the size of the access pattern representation for that method in structured data access.

- **Region Size** - Due to the mechanical nature of hard drives, a larger region size will achieve better bandwidth for methods that read/write only the necessary data. two-phase I/O is not likely to improve as much as the individual I/O methods when increasing region sizes since it uses the data sieving optimization. Since memory does not exhibit the same properties as disk, we do no expect any performance change due to larger region sizes in memory.
- **Region Spacing** - If the distance between file regions is small, two-phase I/O will improve performance due to internal data sieving. If the distance is small enough, we expect file system block operations may help with caching. We note that spacing between regions is usually different in memory and in file due to the interleaved data operation that is commonly seen in scientific datasets that are accessed by multiple processes. For example, in the FLASH code, the memory structure of the block is different that the file structure, since the file structure takes into account multiple processes.

These characteristics, illustrated in Figure 5.1, have a different effect on performance when regarding memory access descriptions or file access descriptions.

5.2. PVFS2 and ROMIO Implementation

PVFS2 [**95**] is a parallel file system for commodity Linux clusters that is a complete redesign of PVFS1 [**11**]. It provides both a cluster-wide consistent name space and user-defined file striping found in PVFS1, but also adds functionality to provide better scalability and performance. Section 2.4.4 provides a detailed description of PVFS1 and PVFS2. Most relevant to this chapter is the native support for noncontiguous data access. PVFS2 provides a client API for handling derived datatypes and internally contains specific functionality for processing them efficiently. Rather than use the aging PVFS1 for our testing, we opted for PVFS2 due to its optimized datatype processing engine.

In order to test the various noncontiguous I/O methods in PVFS2, we had to implement them in the PVFS2 driver of ROMIO. In essence, the entire read/write section of the PVFS2 driver was completed overhauled. To allow the basic POSIX I/O test scenario, the "contig" section of the driver was rewritten to share a single code path. Instead of using the complicated list I/O implementation of the PVFS1 driver, we started the list I/O PVFS2 implementation from scratch. The MPI-IO datatypes are still flattened and processed in the same manner as in Section 3.3. However, the code paths from read and write were combined and much of the code was simplified by making general purpose code that performs as well as the old case-specific code.

Finally, support for datatype I/O was added by breaking down MPI datatypes into PVFS2 datatypes to produce native PVFS2 I/O datatype requests. We also added support for subarray and darray datatypes, which have been traditionally difficult to support. We changes the ADIO PVFS2 hints to select different I/O methods with the MPI_Info_set()

call. Our modified PVFS2 driver has been sent to ROMIO developers and is expected to be released soon with a new version of ROMIO.

As PVFS2 was new at this time, we also worked with PVFS2 developers to improve noncontiguous I/O support in PVFS2. A number of bugs popped up at various levels in the datatype processing and flow control. With the help of the PVFS2 team, we were able to fix many of these bugs and improve the correctness of the code.

We rewrote the I/O portion of the PVFS2 server trove component to use read(), write() and lseek() calls instead of lio_listio(). We found this optimization improved performance over an order of magnitude in some noncontiguous I/O cases.

5.3. HPIO Benchmark

We have designed an I/O benchmark, *High-Performance I/O Test* (HPIO), for studying I/O performance using various I/O methods, I/O characteristics, and noncontiguous I/O cases. We test all three I/O characteristics (region size, region count, and region spacing) against four I/O methods (POSIX I/O, list I/O, two-phase I/O, and datatype I/O) in all four of the I/O access cases (c-c, nc-c, c-nc, and nc-nc).

We chose to call MPI_File_sync() after every I/O operation. This enables us to include the time to move the data to the hard drive and not simply test network bandwidth. When using two-phase I/O, an optimization in the PVFS2 driver of ROMIO forces only one of the processes to actually call MPI_File_sync(), instead of all 64 processes calling MPI_File_sync() when using individual I/O methods. Test runs that did not synchronize the data to disk showed I/O bandwidth as high as 1.59 GBytes / sec. Our best synchronized results will barely reach 24% of that (roughly 389 MBytes / sec). However, when checkpointing and doing other persistent data storage operations, file synchronization to hard disk is not just

advised, it is essential to ensure that the data is available when the application is restarted or visualized. All tests used 64 compute processes on 32 dual CPU computers and 16 computers each running a single PVFS2 server process. I/O tests have a lot of variance, so for each data point for we averaged 6 repetitions without the high and the low. We also inserted a one second delay between test runs to try to minimize effects from previous runs.

In order to observe the individual effect of varying each of the I/O characteristics, we hold all other characteristics constant during our experiments. We chose a default region size of 8 bytes to match the size of a double on most computing platforms. This makes the test as I/O intensive as possible and provides a worst case scenario for scientific computing. We used a default spacing of 128 bytes since this maps well to applications that use 16 variables per cell (similar to 17 in IPARS and 24 in the ASC FLASH code). We chose a default region count of 4096 to represent 4096 cells, a mid-size grid. We note that whenever memory and/or file descriptions are contiguous, we used a contiguous MPI datatype for data representation. We used a vector MPI datatype for noncontiguous data representation. While many scientific applications store data in multi-dimensional datasets, which can be represented by vector of vector MPI datatype, we chose to use a single MPI vector datatype to keep consistent region spacing. Our results can be extrapolated to approximately determine multi-dimensional region spacing performance.

5.4. HPIO Results

All tests were run on the Feynman cluster at Sandia National Laboratories. Feynman, composed of Europa nodes, Ganymede nodes, and I/O nodes, has a total of 371 computers. In order to keep our testing as homogeneous as possible, we only used the Europa nodes. The Europa nodes are dual 2.0 GHz Pentium-4 Xeon CPUs with 1 GB RDRAM. They are

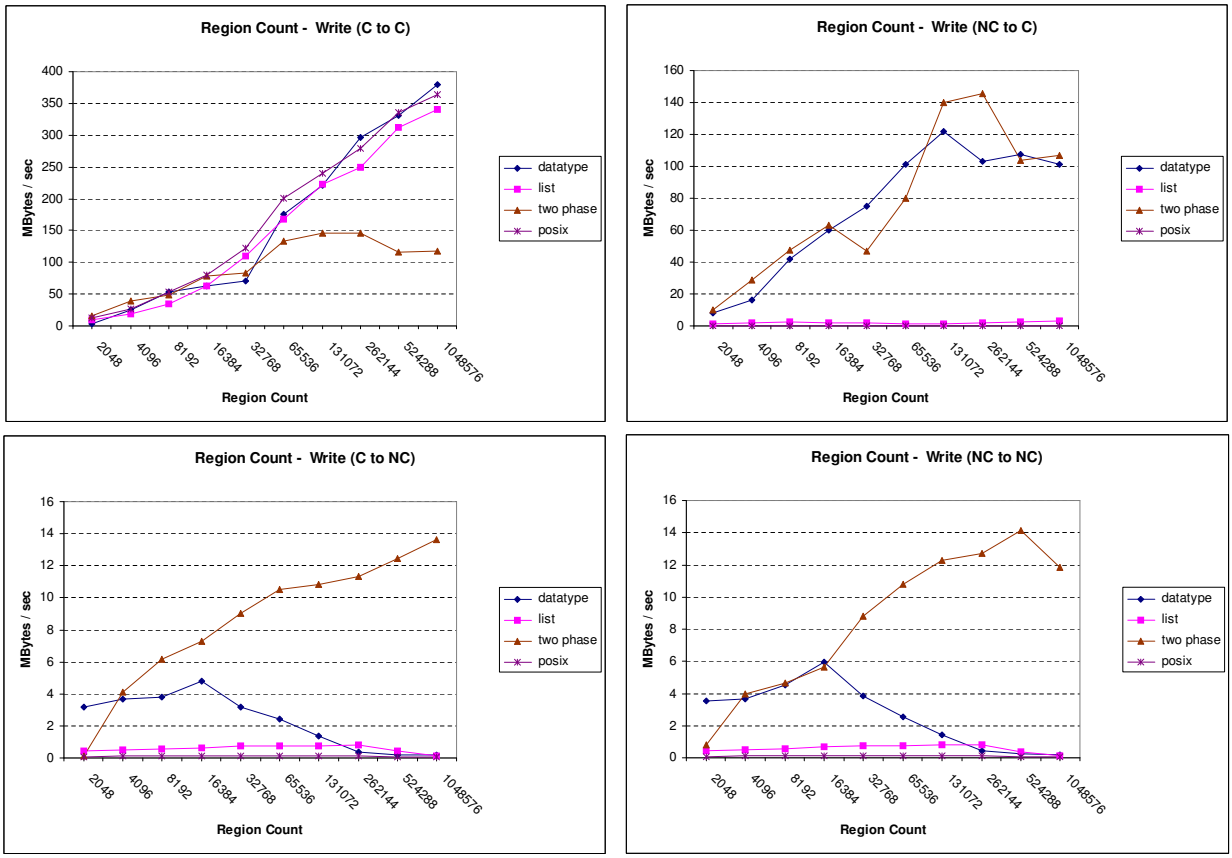

Figure 5.2. HPIO results from testing various region counts.

connected with a Myrinet-2000 network and use the Red Hat Linux Enterprise operating system. Since each computer has dual CPUs, we used 2 compute processes per node.

We used 16 computers for our PVFS2 file system. All 16 computers ran the PVFS2 server with one computer additionally handling metadata server responsibilities. All PVFS2 files were created with the default 64 KByte strip size, totaling to a 1 MByte stripe across all I/O servers. Figure 5.2 shows our results when varying the region count. Figure 5.3 shows our results when varying the region size. Figure 5.4 shows our results when varying the region spacing.

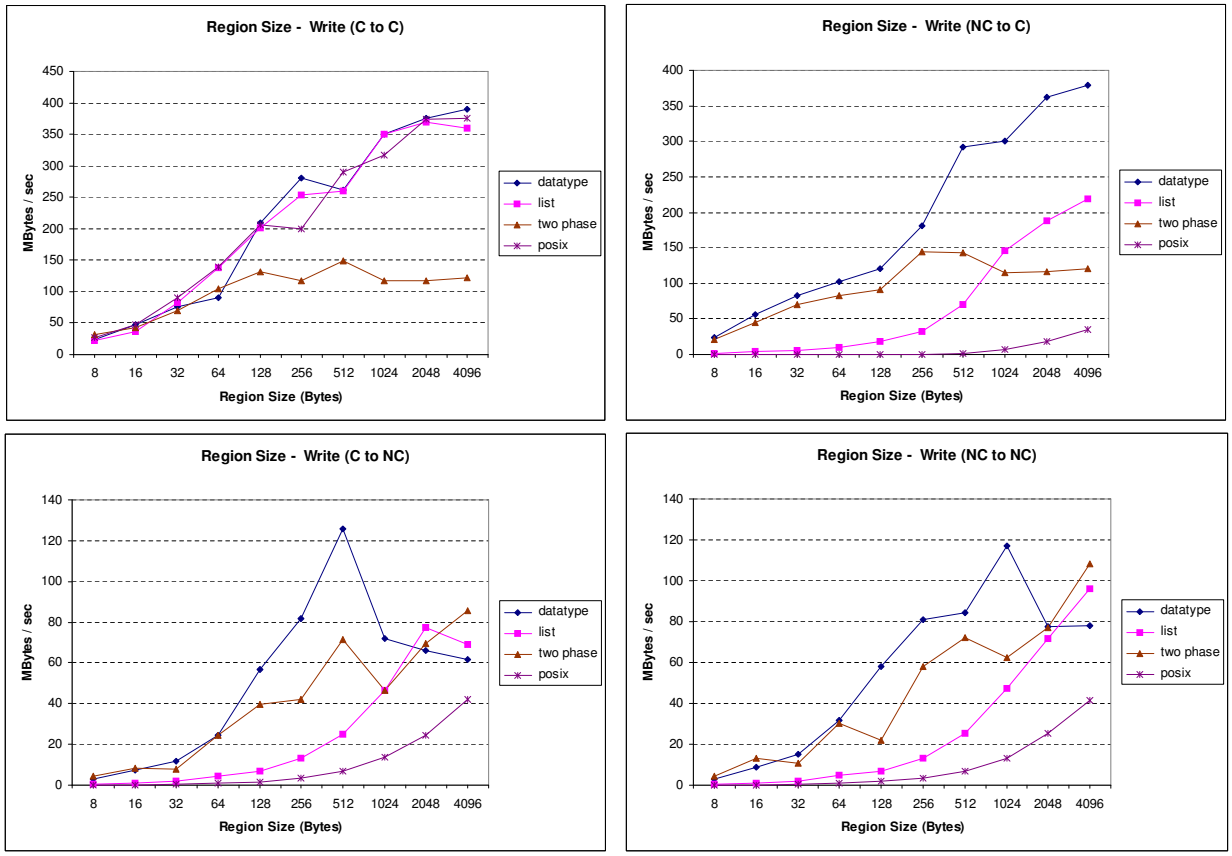

Figure 5.3. HPIO results from testing various region sizes.

5.4.1. HPIO - Vary Region Count Results

In the c-c case, increasing the region count increases overall bandwidth. The data sizes per process range from 16 KBytes at the low end (region count = 2048) to 8 MBytes at the high end (region count = 1048576). The overhead of a double network transfer makes two-phase I/O much less efficient than the other methods.

In the nc-c case, two-phase I/O and datatype I/O perform fairly well as the impact of varying the region count with noncontiguous memory access isn't as good as c-c, but still scales up some. List I/O performance immediately drops since only 64 offset-length pairs are transferred per request. List I/O must process 1048576 / 64 = 16384 requests per

process, where each request is only 64 * 8 = 512 bytes. Such a small size is not suited for a hard disk. Datatype I/O drops significantly as well due to the PVFS2 server side storage implementation. The PVFS2 flow component can only handle processing 1024 offset-length pairs at a time on the server side. Therefore, datatype I/O can only write 8 KByte regions at a time, which is better than the 512 byte regions in list I/O but still not large enough to achieve high disk bandwidth.

In the c-nc case, we note that I/O bandwidth is less than 5% of the maximum bandwidth we saw in c-c. Noncontiguous data access in file is really hard on the individual I/O methods (POSIX I/O, list I/O, and datatype I/O). Performance of datatype I/O drops off at about 16384 regions. This is mostly likely caused by the fact that datatype I/O requests block the PVFS2 server until they completely finish. Fsync() calls immediately come after the datatype I/O requests and block the PVFS2 server from doing other client I/O requests. Since the fsync() calls arrive at intervals after each datatype I/O request is completed the aggregate write I/O pattern will have 64 I/O requests interrupted with intermittent fsync() calls. List I/O performance isn't as affected by this fsync() problem because it breaks down into I/O requests of 64 file offset-length pairs. Therefore, the list I/O requests all finish about the same time (most likely around when the other list I/O requests are around their last 64 file offset-lengths pairs) and then the fsync() calls occur, making it less costly to fsync() versus the datatype I/O case. The MPI_File_sync() time for the list I/O test in the c-nc case never exceeds 18 seconds when region count = 1048576. The datatype I/O MPI_File_sync() time for the c-nc case reaches as high 2061 seconds. POSIX I/O is so slow that with the region count = 1048576, it took 11014.66 second (0.047 MBytes / sec) to complete a repetition. In fact, two-phase I/O is performing quite well in comparison because it only does a single

fsync() call between all the processes. This allows it to scale up much better than the other methods.

In the nc-nc case, performance is nearly identical to the c-nc case. If the file description is noncontiguous then making the memory description noncontiguous has little impact.

5.4.2. HPIO - Vary Region Size Results

In the c-c case, we see the same trends as the c-c case of the region count test. Increasing the total data size from 32 KBytes (region size = 8) to 16 MBytes (region size = 4096) / process scales up the I/O bandwidth. Since we have 16 I/O servers, each server only receives about 1 MByte per compute process at the maximum size. With larger sizes we could certainly achieve higher I/O bandwidth. Two-phase I/O tails off here again due to its double network transfer.

In the nc-c case, datatype I/O performs almost equivalent to the c-c case. Since the default region count = 4096, and there are 16 I/O servers, the flow component isn't a bottleneck for datatype I/O. Two-phase I/O performance is nearly identical to its nc-c case of the region count test. List I/O actually does fairly well here as well since it isn't as limited by its 64 offset-length pair maximum as it was in the nc-c case of the region count test.

In the c-nc case, datatype I/O peaks rapidly and then falls due to the fsync() issue discussed in the c-c case of the region count test. If the fsync() calls happened after all the writes from all the processes finished we would expect the scaling to continue. All I/O methods benefit from the increased region sizes. Even two-phase I/O can benefit quite a bit from the larger region sizes since it doesn't have to pass around such a large amount of offset-length pairs to each of the aggregators and its percentage of useful data acquired while using the data sieving method is improving.

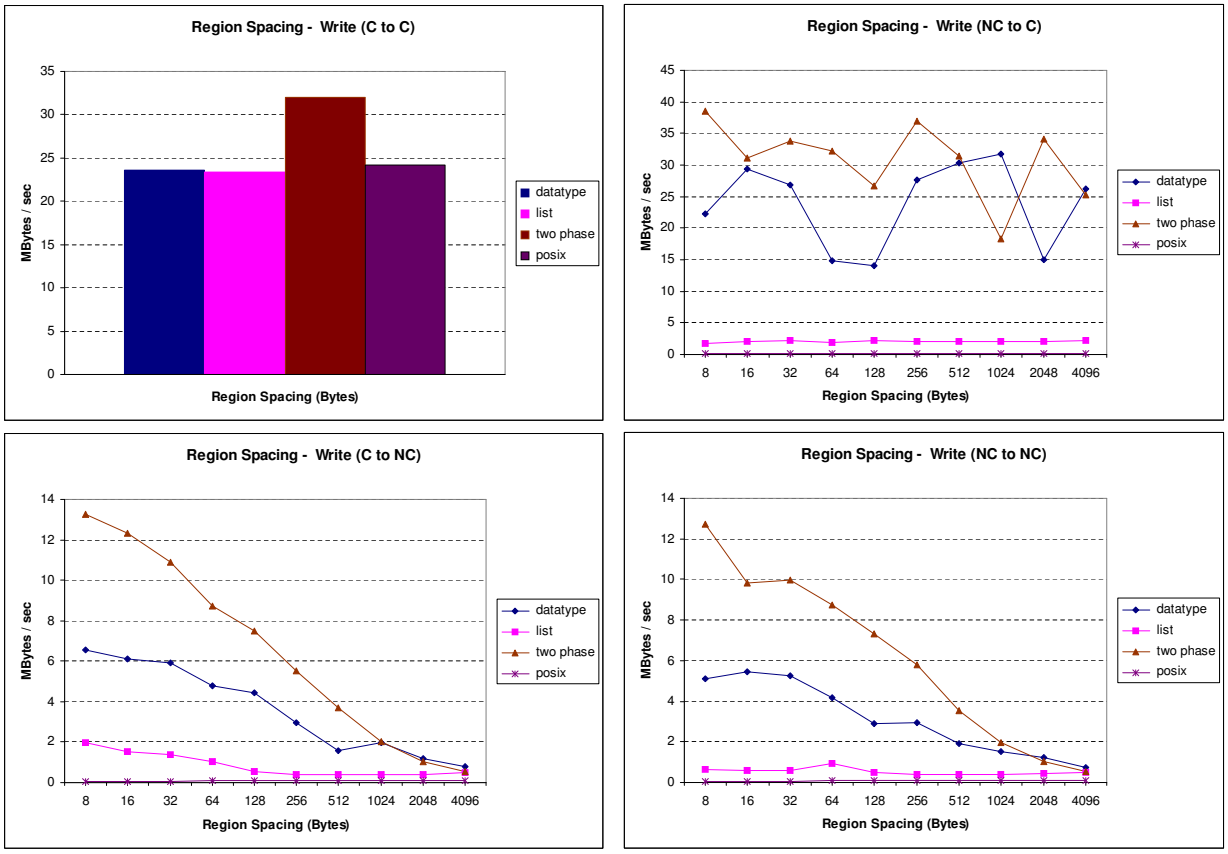

Figure 5.4. HPIO results from testing various region spacing.

In the nc-nc case, the trends of match the c-nc case, again showing that if the file description is noncontiguous, then it makes little performance difference if the memory description is contiguous or noncontiguous.

5.4.3. HPIO - Vary Region Spacing Results

In the c-c case, we only have a single bar since varying region spacing has no effect on a contiguous memory or file description. One interesting thing is that two-phase I/O performs slightly better than the other methods. This is also evident in the c-c cases for the other tests when each process is writing 32 KBytes. This is a case where the MPI_File_sync() costs

are cheaper for two-phase I/O (due to the two-phase MPI_File_sync() optimization). The average MPI_File_sync() cost for the two-phase I/O runs was 0.0391 seconds versus 0.0705 seconds for the normal case, which allows two-phase I/O a win at this small write case.

In the nc-c case, datatype I/O and two-phase I/O fluctuate (even with the average policy we used with 6 repetitions and eliminating the high and the low). The main cause of the fluctuation is that the writes are only 32 KBytes per process and can only fill half a strip on the I/O servers. If we average out the datatype I/O numbers, the result is 23.83 MBytes / sec (23.65 MBytes / sec on the c-c case). If we average out the two-phase I/O numbers, the result is 30.832 MBytes / sec (32.017 MBytes / sec in the c-c case). When going from the c-c case to the nc-c case, datatype I/O and two-phase I/O retain their performance. However, list I/O takes a large drop due to only processing 64 offset-length pairs at a time. Therefore, each process calling list I/O must do a series of small writes (64 * 8 = 512 bytes) to finish the access pattern. POSIX I/O is making 4096 writes of size = 8 bytes, which leads to an aggregate bandwidth of no more than 0.145 MBytes / sec.

In the c-nc case, performance drops for all I/O methods quite significantly. The individual I/O methods are all suffering from disk seek penalties. Writing 8 bytes and then skipping up to 4096 bytes makes performance drop rapidly. two-phase I/O surpasses the other methods due to its internal data sieving implementation making larger I/O operations until about 1 KByte spacing, where it appears that the penalty of data sieving large holes overwhelms the benefits of larger I/O operations and allows datatype I/O to surpass it.

The nc-nc case shows a similar trend to the c-nc case. Again we note that moving from c-nc to nc-nc makes only a small performance difference.

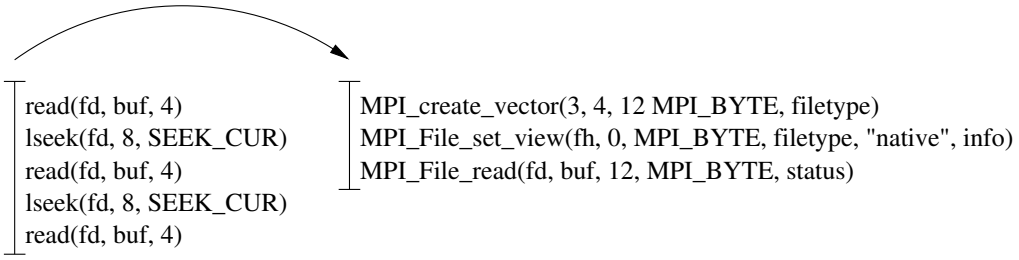

Figure 5.5. Example code conversion from the POSIX interface to the MPI-IO interface.

5.5. I/O Guidelines

- **All large-scale scientific applications should use the MPI-IO interface (either natively or through higher level I/O libraries).** MPI-IO is a portable parallel I/O interface that provides more performance and functionality over the POSIX I/O interface. Whether using MPI-IO directly or through a higher level I/O library which uses MPI-IO (such as PnetCDF or HDF5), applications can use numerous I/O optimizations such as collective I/O and data sieving I/O. MPI-IO provides a rich interface to build descriptive access patterns for noncontiguous I/O access. Most programmers will benefit from the relaxed semantics in MPI-IO when compared to the POSIX I/O interface. If a programmer chooses to use a particular file system's custom I/O interface (i.e. not POSIX or MPI-IO), portability will suffer.
- **Group individual I/O access to make large MPI-IO calls.** Even if an application programmer uses the MPI-IO interface, they need to group their read/write accesses together into larger MPI datatypes and then do a single MPI-IO read/write. Larger MPI-IO calls allow the file system to use optimizations such as data sieving

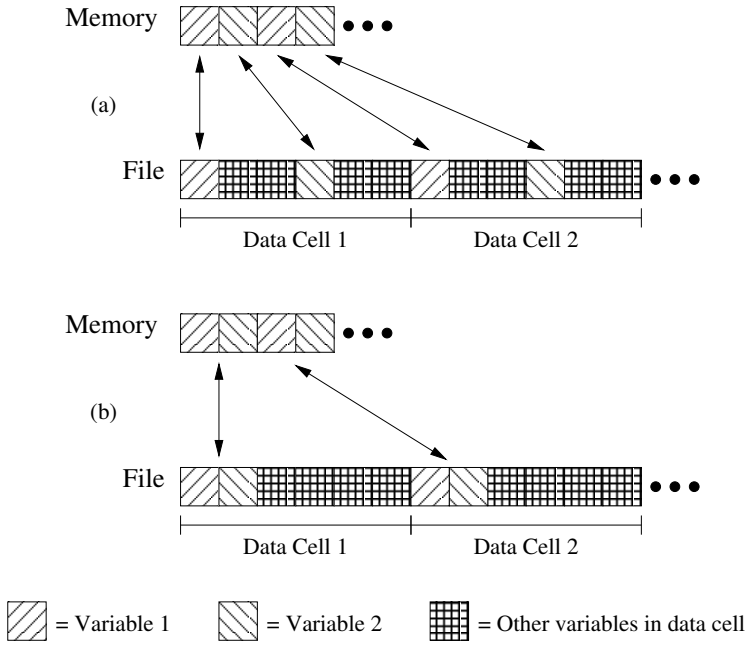

Figure 5.6. (a) Original layout of variables in data cells. (b) Reorganization of data to combine file regions during write operations increases I/O bandwidth.

I/O, list I/O and datatype I/O. It also provides the file system with more information about what the application is trying to do, allowing it to take advantage of data locality on the server side. A simple code conversion example in Figure 5.5 changes 3 POSIX read() calls into a single MPI_File_read() call, allowing it to use data sieving I/O, list I/O, or datatype I/O to improve performance.

- **Whenever possible, increase the file region size in an I/O access pattern.** After creating large MPI-IO calls which service noncontiguous I/O access patterns, try to manipulate the I/O access pattern such that the file regions are larger. One way to do this is data reorganization. Figure 5.6 shows how moving variables around in a data cell combined file regions for better performance. While not always possible, if a noncontiguous file access pattern can be made fully contiguous, performance

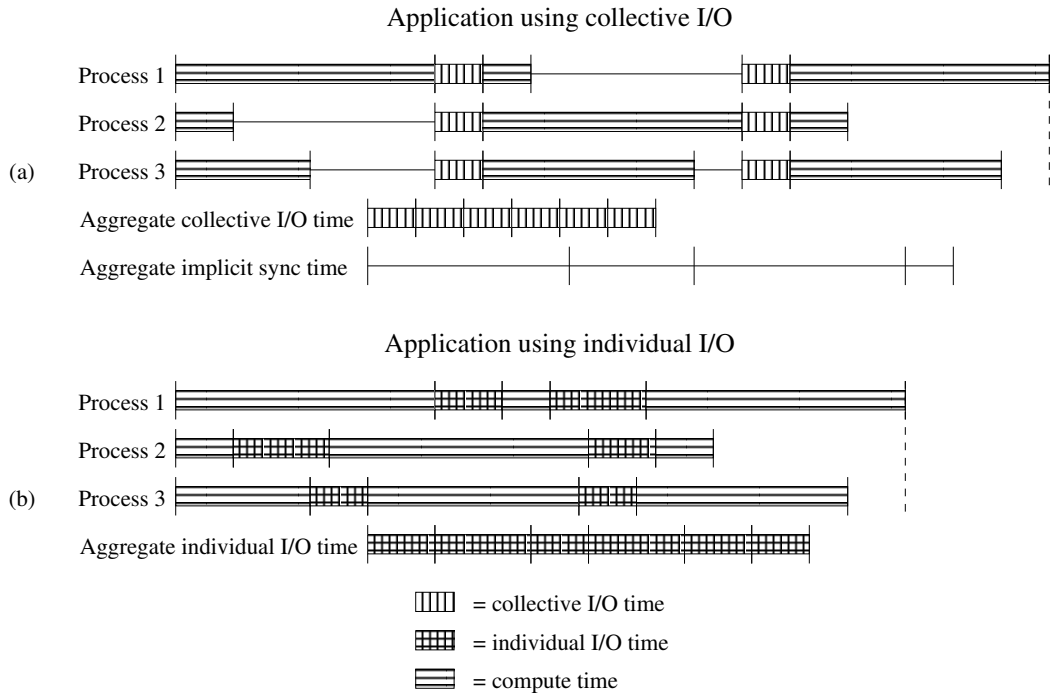

Figure 5.7. Cost of collective I/O synchronization. Even if collective I/O (a) can reduce the overall I/O times, individual I/O (b) outperforms it in this case because of no implicit synchronization costs.

can improve by up to 2 orders of magnitude [**20**]. When storing data cells, some programmers write one variable at a time. Making a complex memory datatype to write this data contiguously in file in a single MPI-IO I/O call will be worth the effort.

- **Reduce the file region spacing in an I/O access pattern.** When using data sieving I/O and two-phase I/O, this will improve buffered I/O performance by accessing less unused data. POSIX I/O, list I/O, and datatype I/O will suffer less disk seek penalties. Again, a couple of easy ways to do this is to reorganize the data layout or combine multiple I/O calls to make fewer, but larger I/O calls.

- **Consider individual versus collective (two-phase I/O).** Two-phase I/O provides good performance over the other I/O methods when the file regions are small (bytes or tens of bytes) and nearby since it can make large I/O calls, while the individual I/O methods (excluding data sieving I/O) have to make numerous small I/O accesses and disk seeks. The advantages of larger I/O calls outweigh the cost of passing network data around in that case. Similarly, the file system can process accesses in increasing order across all the clients with two-phase I/O. If the clients are using individual I/O methods, the file system must process the interleaved I/O requests one at a time, which might require a lot of disk seeking. However, in many other cases, list I/O and datatype I/O outperform two-phase I/O [**20**]. More importantly, two-phase I/O has an implicit synchronization cost. All processes must synchronize before any I/O can be done. Depending on the application, this synchronization cost can be minimal or dominant. For instance, if the application is doing a checkpoint, since the processes will likely synchronize after the checkpoint is written, the synchronization cost is minimal. However, if the application is continually processing and writing results in an embarrassingly parallel manner, the implicit synchronization costs of two-phase I/O can dominate the overall application running time as shown in Figure 5.7.
- **When using individual I/O methods, choose datatype I/O.** In nearly all cases datatype I/O exceeds the performance of the other individual I/O methods. The biggest advantage of datatype I/O is it can compress the regularity of an I/O access pattern into datatypes, keeping a one-to-one mapping from MPI-IO calls to file system calls. In the worst case (unstructured I/O), datatype I/O breaks down to list I/O which is still much better than POSIX I/O.

- **Do not use data sieving I/O for interleaved writes.** Interleaved writes will have to be processed one at a time by the file system because the read-modify-write behavior in the write case requires concurrency control. Using data sieving I/O for writes is only supported by file systems which have concurrency control. Data sieving I/O is much more competitive with the other I/O methods when performing reads, but should still be used in limited cases.
- **Generally, there is no need to reorganize the noncontiguous memory description if file description is noncontiguous.** Some programmers might be tempted to copy noncontiguous memory data into a contiguous buffer before doing I/O, but our results suggest that it will not make any difference in performance. It would most likely just incur additional programming complexity and memory overhead.

CHAPTER 6

Exploring I/O Strategies for Parallel Sequence-Search Tools

Our last chapter studied the performance effects of I/O characteristics from a system perspective. In this chapter we examine performance from an application's view of I/O strategies in the popular domain of sequence-search.

Sequence-search is one of the fundamental tasks routinely performed in computational biology. Sequence-search is typically used to find similarities between newly discovered DNA or amino-acid sequences and those in known nucleotide or protein databases. The results of sequence-search can be used to predict the structures and functions of new sequences. They also allow people to estimate the evolution distance in phylogeny reconstruction and perform gnome alignments. With the introduction of advanced sequencing technologies, sequence databases are rapidly growing. For example, GenBank [**4**] (a widely used DNA sequence database maintained by the National Center for Biotechnology Information) increased in size by over five orders of magnitude from 1982 to 2004 [**60**]. Parallel sequence-search tools are necessary for sequence analysis of modern and future sequence databases.

Query segmentation and database segmentation are the popular design choices for parallel sequence-search tools on general-purpose parallel machines. Many existing parallel sequence-search tools are based on query segmentation [**10, 14, 8, 83, 34**]. In this approach, the entire sequence database is replicated to all processors and a set of query sequences are segmented into fractions. Each processor searches a fraction of query sequences against the entire sequence database. When the sequence database does not fit into the processor

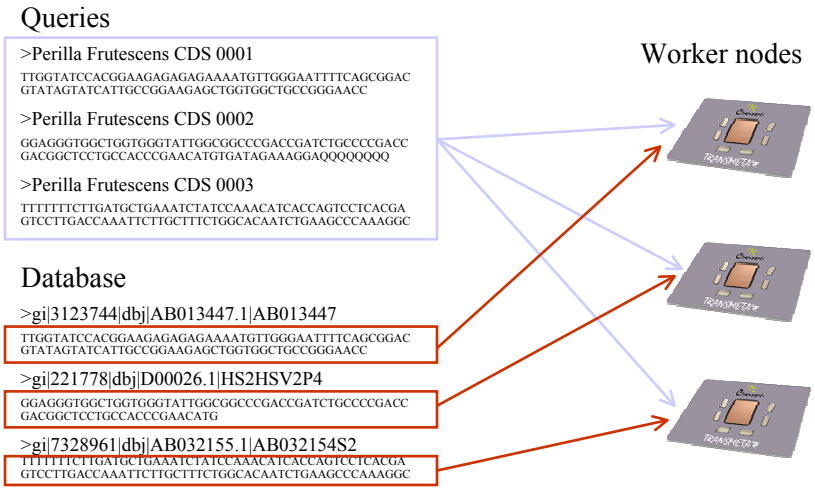

Figure 6.1. Database segmentation.

memory, query segmentation suffers repeated I/O introduced by loading sequence data back and forth between the file system and the main memory. In database segmentation, the entire set of query sequences is replicated to all processors and the sequence database is partitioned (as shown in Figure 6.1. Each processor searches whole query sequences against a fraction of the sequence database. Super-linear speedup is possible when the sequence database is larger than the processor memory by fitting the large database into the aggregate memory of all processors [**28**]. Parallel sequence-search tools that use database segmentation include mpiBLAST [**28**], pioBLAST [**51**], TurboBlast [**6**] and parallel BLAST [**54**].

Database segmentation is expected to be the inevitable trend of future parallel sequence-search tools for following reasons. First, the rapid growth of sequence databases prohibits a sequence database from fitting into the memory of a single processor. Second, as sequence databases increase in size, searching a query against the whole database will take substantial time and result in resource under-utilization when the number of sequences is relatively small compared to the number of processors. Database segmentation offers better resource utilization on large-scale machines regardless of number of input query sequences.

Although current I/O costs in parallel sequence-search tools (such as mpiBLAST) are relatively small in proportion to overall execution time, we believe future I/O performance will be increasingly important to sequence-search throughput because of following reasons. First, the performance gap between processor speed and I/O speed continues to widen, making I/O much more significant in overall execution time. Search times are shrinking as we use advanced computational hardware such as multicore-chip architectures. Solutions based on field programmable gate arrays (FPGAs) or application-specific integrated circuits (ASICs), such as BioScan [**99**], Parcel's GeneMatcher [**70**], Compugen's Biocceleator [**24**] and TimeLogic's DeCypher [**40**], have proven to be very efficient and can deliver orders-of-magnitude performance improvements in comparing large sequences. Second, the development of smarter heuristic algorithms (such as SSAHA [**63**], PatternHunter [**53**], and BLAT [**42**]) greatly reduces the sequence-search costs.

Our past experience with parallel sequence-search tools has led us to believe that an individual worker-writing I/O strategy would significantly improve overall execution times. In this chapter, we compare such an I/O strategy against other I/O strategies for parallel sequence-search tools using database segmentation. We have developed S3aSim, a sequence similarity search algorithm simulator, to get a detailed understanding of I/O strategies in parallel sequence-search tools.

In Section 6.1, we describe and compare the various I/O strategies for parallel sequence-search tools. In Section 6.2, we introduce S3aSim, our simulation tool for understanding I/O in parallel sequence-search tools. In Section 6.3, we present our S3aSim results with respect to increased processors and increased computational ability. In Section 6.4, we conclude with our contributions in this chapter and discuss possibilities for future work.

6.1. I/O Algorithms in Parallel Sequence-Search Tools

Although parallel sequence-search tools employ different sequence-alignment algorithms, they have much in common when using the database segmentation approach, as proposed by [**28**]. First, each processor searches a fraction of the database. These searches on different processors are embarrassingly parallel. Second, the output results in query sequence match database similarities ordered by statistics representing the alignment qualities. Finally, local outputs in different processors need to be merged and sorted according to the search statistics (usually a similarity score) in order to produce the final output. Most parallel sequence-search tools will use the message passing interface (MPI) and its associated I/O chapter (MPI-IO) for code portability. In the rest of the chapter, we will refer to some example parallel database-search tools such as mpiBLAST and pioBLAST.

Older versions of mpiBLAST would store all the results on master/worker nodes until the end of program execution. Since this can result in exceeding memory limits in large application runs, the current design path (for example mpiBLAST 1.4) has headed towards writing the results out immediately after a query is processed or after every n queries have been processed, where n is a fraction of the overall input query set. More frequently writing out the results also allows users to resume a failed application run at the appropriate input query. Workers always send their results back to the master ordered by score. As the master receives the ordered results, it can easily merge them together with its list of ordered results. Basically, the sorting costs are offloaded as much as possible to the workers so the master can focus on its primary job of distributing tasks to the workers.

We note that parallel sequence-search algorithms have a unique set of I/O characteristics when writing out results.

- **Non-uniform result size** - The size of each result is relative to the minimum searching threshold and up to three times the maximum of the input query and the matching database sequence (for BLAST output).
- **Result count** - This is completely data dependent and can range from no results to hundreds of thousands of results depending on the input queries and the database size.
- **Unstructured data** - When the workers write to the output file, the resulting data is noncontiguous and unstructured (i.e., no regularity). When the master writes to the output file, the resulting I/O calls are large and contiguous.

In the upcoming subsections, we describe possible I/O strategies for writing a results file.

6.1.1. Master-Writing

After each worker completes its queries, it sends its ordered results to the master. This involves sending over its scores and actual result data. The master merges the ordered results with its ordered results. If all the fragments for an input query have been processed, it writes the ordered results to a file in one contiguous I/O call. We will refer to this master-writing strategy as the MW strategy in the rest of the chapter.

The MW strategy resembles the I/O strategy used in mpiBLAST 1.2. A large difference, however, is that mpiBLAST 1.2 did not write out results after each query. In mpiBLAST 1.2, the master wrote all its results at the end of the application run. This limited the size of input queries and the target database used. It also provided little opportunity for resuming an application run after failure.

The main advantage of the MW strategy is that it writes the output data contiguously. Contiguous I/O is much more efficient that noncontiguous I/O [**19**]. The MW strategy is

also easy to implement and debug. However, one disadvantage of the MW strategy is that the master is a centralized point of contention where the full result data is sent. Only a single process is gathering all the results and doing the writing on behalf of all the workers. Also, while the master is writing, it cannot assign new tasks to the workers (causing potentially long wait times before a worker can begin a new task). While nonblocking I/O could reduce this overhead, blocking I/O is commonly used in a MW strategy to avoid overloading the memory of the master process. As we scale up the number of workers, the MW strategy will likely not scale as well as the other I/O strategies.

6.1.2. Worker-Writing: Collective I/O

An application designer can choose to have the workers write the results themselves in order to help the master focus on assigning work. After each worker finishes processing its input query against its database fragment, it sends over only its scores of the ordered results to the master. The master merges in these ordered scores with the previously received ordered scores from other workers. If all the fragments for an input query have been processed, it sends the workers the location of where to write each result in the aggregate output file. This location information consists of a list of 64-bit offsets sent to each worker with results. All of the workers then synchronize to write their results collectively to the correct locations in the output file. Since results are written to mutually exclusive locations in the file, the data is interleaved but not overlapping. We will refer to this worker-writing strategy with collective I/O as the WW-Coll strategy. We refer to the more general class of worker-writing strategies as WW strategies.

The WW-Coll strategy, proposed by pioBLAST [**51**], uses MPI-IO collective writes to instruct workers to simultaneously write all of their results at the end of program execution.

When compared to the MW strategy, collective worker-writing allows the I/O bandwidth to scale up with the number of workers. In most cases, having more clients writing simultaneously provides better I/O throughput to a high-performance file system. However, as noted earlier, a disadvantage of the WW strategies is that the workers must use noncontiguous I/O methods. Furthermore, with the WW-Coll strategy, all the workers must synchronize with each other to write. This synchronization cost is at least the time from when the first worker receives its result list to when the last worker receives its result list (before collective I/O begins). On the other hand, since the worker only sends the scores and not the actual results to the master, the amount of data exchanged with the master is reduced from the MW strategy (even including the overhead of the location list data passed from the master to the workers).

6.1.3. Worker-Writing: Individual I/O

We propose to modify the WW-Coll strategy to use individual I/O in parallel sequence-search tools. Instead of using collective I/O, we let the workers write results after completing an input query (or a group of input queries) using individual noncontiguous I/O methods. Our modified WW strategy begins with the master issuing input queries to the workers. The workers are responsible for processing the queries, generating the sorted results, and sending the ordered scores to the master. The master returns the location list to the workers and each worker writes the result data to the output file on its own (not collectively) when it notices it has received the location list from the master. While workers wait for the location list from the master, they can process additional queries, unlike the WW-Coll strategy.

Since collective I/O requires all involved processes to block until synchronized, the WW-Coll strategy cannot allow worker processes to begin upcoming queries until after the I/O operation.

That is, we try to eliminate the synchronization time inherent in collective I/O and relieve pressure on the file system by writing when a worker is ready instead of forcing all workers to simultaneously write. Eliminating the synchronization time should have a significant impact on overall application performance and balance out the load on the file system. We used two different noncontiguous I/O methods (POSIX I/O and list I/O). The POSIX I/O method is the MPI_Write() call without optimization. The list I/O method, described in [**17**], is an optimization for high-performance file systems. We call our modified WW strategies with POSIX I/O and list I/O, the WW-POSIX strategy and WW-List strategy, respectively.

6.2. S3aSim

Each of the aforementioned I/O strategies would be difficult to compare in a single application (such as mpiBLAST or pioBLAST). The main difficulties are implementation time and complexity. Each I/O strategy requires substantial changes to the overall parallel search algorithm. They could also require changes in network protocol and intricacies of the actual search algorithms (for example, modifying NCBI [**60**] BLAST code). We do not wish to compare mpiBLAST 1.2, pioBLAST, and our individual worker-writing strategy in another parallel sequence-search tool to compare I/O strategies. At this time, no benchmarks for parallel I/O in bioinformatics exist. In order to create a fair comparison of I/O strategies that provides flexibility in altering input parameters (such as computational time, input query size, I/O strategies), we created S3aSim: a sequence similarity search algorithm simulator.

Algorithm 1 Master Process

```
1: Distribute input variables to the workers and setup internal data structures.
2: while {1} do
3:    MPI_Recv() a request for work.
4:    if All queries have been scheduled then
5:       Notify worker all queries have been scheduled.
6:    else
7:       MPI_Send worker (query #, fragment #).
8:       Post MPI_Irecv() for worker scores (and results if MW).
9:    end if
10:   Check MPI_Irecv() to see if any workers have finished sending results for their (query
      #, fragment #).
11:   if All queries have been scheduled then
12:      Continue {Inform other workers that all queries have been scheduled before pro-
         ceeding}
13:   end if
14:   if Use Parallel I/O then
15:      MPI_Isend() offset list to workers for any completed queries.
16:      Check to see which MPI_Isend() calls completed.
17:   else
18:      Write finished results to output file for completed queries.
19:   end if
20:   if All queries scheduled, processed, and results written to output file then
21:      Exit()
22:   end if
23: end while
```

S3aSim provides many benefits for our I/O strategy comparison including simple implementation, variability of many input parameters, and integration with the multi-processing environment (MPE) and Jumpshot [**101**] for easy debugging. It is a tool to understand how computation and I/O interact together in a typical parallel sequence-search tool. It uses both MPI and MPI-IO calls for portability on many large-scale machines. S3aSim allows the user to customize the total number of fragments of the database, total number of input queries, a box histogram of input query sizes, a box histogram of database sequence sizes, a min/max count of results per input query, a minimum result size per query, variable

simulated compute speeds, MPI-IO hints, parallel I/O, write all data at the end (similar to mpiBLAST 1.2 and pioBLAST), and many others.

The primary disadvantage of S3aSim is that the modeling of the computational time is inexact. Compute time is modeled as a constant startup cost + linear time based on the size of the result (anywhere from the minimum input size to three times the maximum of the input query and the matching database sequence). We use three times the maximum of the input query and the matching database sequence as the S3aSim result size since the actual BLAST output is generally formatted with the input sequence, database sequence, and the matches between them. This formula can be modified to more accurately model various search algorithms for future work.

Algorithm 2 Worker Process

```
1: Receive input variables from the master and setup internal data structures.
2: while {1} do
3:    MPI_Send() the master a request for work.
4:    MPI_Recv() response from the master.
5:    if (query #, fragment #) then
6:       Compute search algorithm on (query #, fragment #).
7:       if Use Parallel I/O then
8:          Merge current results with previous results for this query.
9:       end if
10:      MPI_Isend() scores (and results if MW) to the master.
11:      if Use Parallel I/O then
12:         Post MPI_Irecv() offset list from the master.
13:      end if
14:   end if
15:   Check all pending MPI_Isend() for completion.
16:   if Use Parallel I/O then
17:      Check all pending MPI_Irecv() for completion.
18:      For all offset lists received, write results to output file.
19:   end if
20:   if All queries scheduled, processed, and results written to output file then
21:      Exit()
22:   end if
23: end while
```

The basic master algorithm is outlined in Algorithm 1 and the basic worker algorithm is outlined in Algorithm 2. In order to maximize the amount of time spent on distributing work, the master uses the blocking MPI_Recv() call when handling worker requests. We used MPI_Isend()/MPI_Irecv() calls for all other communication (receiving results and sending offset lists). When we used parallel I/O (either individual or collective worker writing), the workers only sent the scores of their results to be sorted by the master. If the master was writing, both the scores and the results of a search on a (query #, fragment #) were sent to the master. We note that whenever a process (master or worker) is checking on a nonblocking communication call, it will only test for completion (MPI_Test()) instead of blocking on completion (MPI_Wait()) to allow the process to continue to make progress if the test fails. It will only call MPI_Wait() if the process cannot proceed further until the completion of this particular nonblocking communication call.

S3aSim timing is broken up into different timing phases. We will describe each timing phase with respect to the overall program execution for both the master and the worker.

- **Setup Time** - For the master, this time includes sending out input variable information to the workers (step 1 in Algorithm 1). For the worker, this time includes receiving the input variable information from the master (step 1).
- **Data Distribution** - For the master, this time includes waiting for the next worker request and sending the worker a response (steps 3, 5, and 7). For the worker, this time includes sending the work request and receiving a response (steps 3 and 4).
- **Compute** - For the master, this time is always 0 since the master never does any searching. For the worker, this time includes the search algorithm time (step 6).
- **Merge Results** - For the master, this time is 0. For the worker, this time includes the time to merge results together if we are using parallel I/O (step 8).

- **Gather Results** - For the master, this time includes posting MPI_Irecv() operations for scores (and possibly results) from the worker as well as checking on the associated MPI_Irecv() operations (steps 8 and 10). For the worker, this time includes sending off the scores (and possibly results) as well as checking on the associated MPI_Isend() operations (steps 10 and 15).
- **I/O** - For the master, this time includes all write operations to the output file (step 18). For the worker, this time includes all write operations to the output file (step 18).
- **Sync** - For the master, this time includes waiting for all the processes to synchronize at the end of the application (not shown in the algorithm). For the worker, this time includes waiting for all the processes to synchronize at the end of the application (not shown in the algorithm). When the query sync mode is on, this time includes the time for all processes to synchronize after writing out the results for an input query.
- **Other** - For the master and the worker, this phase includes all remaining time.

6.2.1. Test Environment

All tests were run on the Feynman cluster at Sandia National Laboratories. Feynman, composed of Europa nodes, Ganymede nodes, and I/O nodes, has a total of 371 computers with dual processors. In order to keep our testing as homogeneous as possible, we only used the Europa nodes that were dual 2.0-GHz Pentium-4 Xeon CPUs with 1-GByte RDRAM. They were connected with a Myrinet-2000 network and used the Red Hat Linux Enterprise operating system. Since each of compute nodes had dual CPUs, we ran two compute processes per node.

We also used 16 computers in our PVFS2 file system. All 16 computers ran the PVFS2 server with one computer additionally handling metadata server responsibilities. All PVFS2 files created use the default 64-KByte strip size, which totals to a 1-MByte stripe across all I/O servers. Our version of PVFS2 was tuned for better noncontiguous I/O by adjusting default system parameters.

6.2.2. Test Setup

With so many input variables, there are numerous tests we could run with S3aSim. In this chapter, we chose to focus on testing the scalability of the I/O strategies with respect to processor count and compute speed. In order to get the characteristics of an NCBI database, we chose the NT database (nt.gz from ftp://ftp.ncbi.nih.gov/blast/db/FASTA/) as our example database. The NT database has a minimum sequence length of 6 bytes, a maximum sequence length of slightly over 43 MBytes, and mean sequence length of 4401 bytes. We used the same histogram to represent our input query set of 20 queries (roughly maps to approximately 86 KBytes of input queries). We chose 128 fragments and a result count from 1000 to 2000 per query (over the entire database). Results were written to the output file after each query. MPI_File_sync() was always called immediately after every MPI_File_write() or MPI_File_write_all(). Although we use different numbers of processors, the results are always identical since they are pseudo-randomly generated. Each data point we present generated roughly 208 MBytes of output data and was averaged over 3 test runs.

Our tests were designed for comparing the performance of the various I/O strategies with respect to an increase in processes or reduced computational time (custom search hardware and/or better algorithms). One suite of tests used 2 to 96 processors. The second suite of

tests used compute speeds from 0.1 to 25.6. The breakdown of the application into phases is crucial to understanding why certain I/O strategies work better than others.

Another goal of our testing was to examine whether the inherent I/O synchronization of collective I/O would be very costly and how it might be improved. Collective I/O, in nearly all noncontiguous I/O cases, outperforms POSIX I/O and, in some noncontiguous I/O cases, outperforms list I/O in pure I/O tests. It is rare (if ever) when an I/O comparison takes into account interaction with an application when doing performance evaluations. It is very hard to directly examine the effect of inherent I/O synchronization in collective I/O. In order to expose this effect, our tests used the "query sync" option. Basically, if the query sync option is on, S3aSim will force all worker nodes to synchronize after doing any I/O. When looking at the performance changes from individual I/O methods with query sync off to with the query sync on, we can gain a better understanding of how the inherent synchronization in collective I/O hurts overall performance and whether using individual noncontiguous I/O methods in a collective I/O implementation might actually improve performance. For example, a collective I/O method could be implemented using list I/O with a forced synchronization at the end of the I/O operation (similar to our WW-List tests with query sync on). In our upcoming performance evaluation and discussion we refer to not using and using the query sync option as "no-sync" and "sync", respectively.

6.3. Performance Evaluation

Our first test suite examines process scalability in S3aSim. Figure 6.2 shows the overall S3aSim execution times on a logarithmic scale to emphasize the performance variation in the different I/O strategies. As expected, all no-sync I/O strategies perform as good as or better than their sync counterparts. The individual WW strategies outperform both the WW-Coll

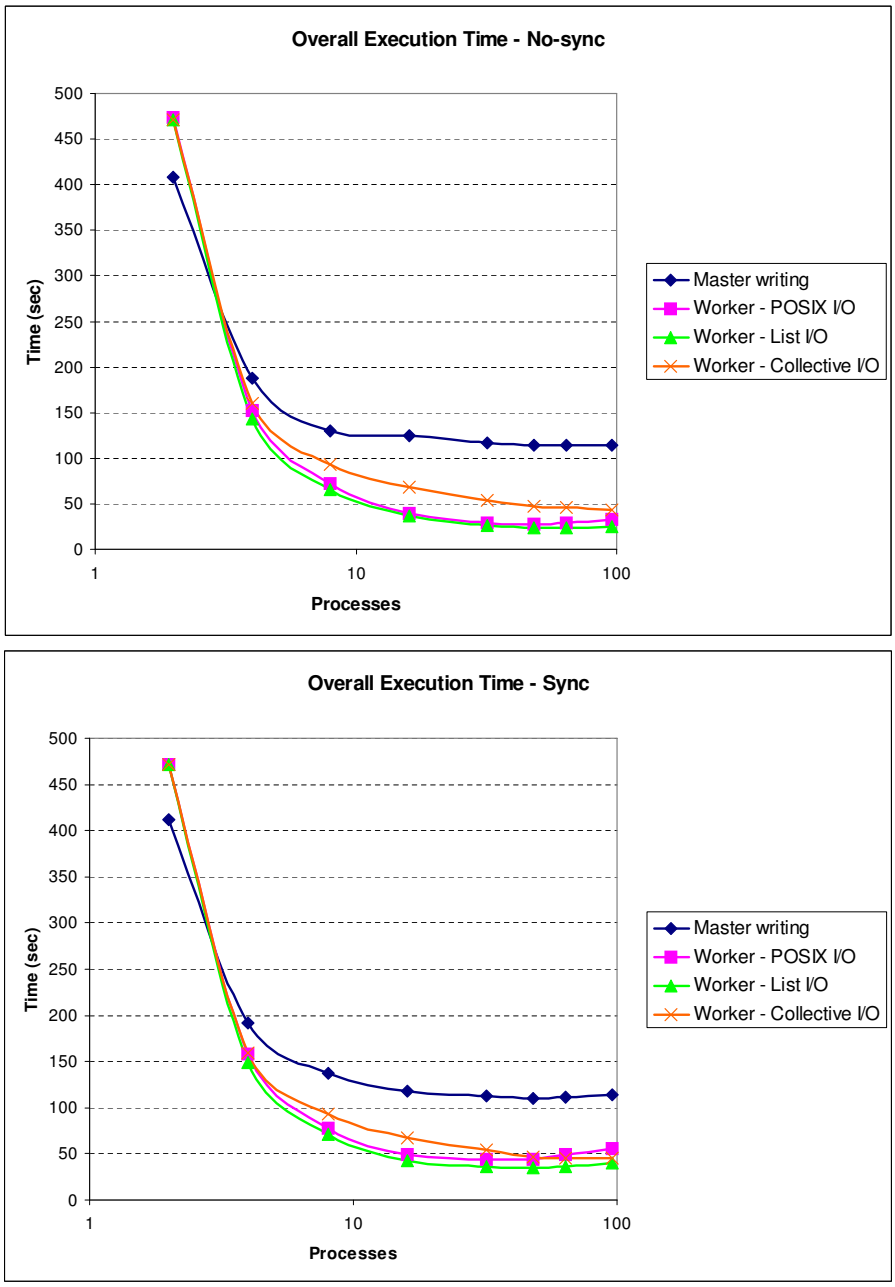

Figure 6.2. Results when scaling up the number of processors with no-sync/sync query options.

and MW in the no-sync cases. WW-Coll performance is about the same with or without the sync option. It is expected that WW-Coll would have roughly the same performance with or

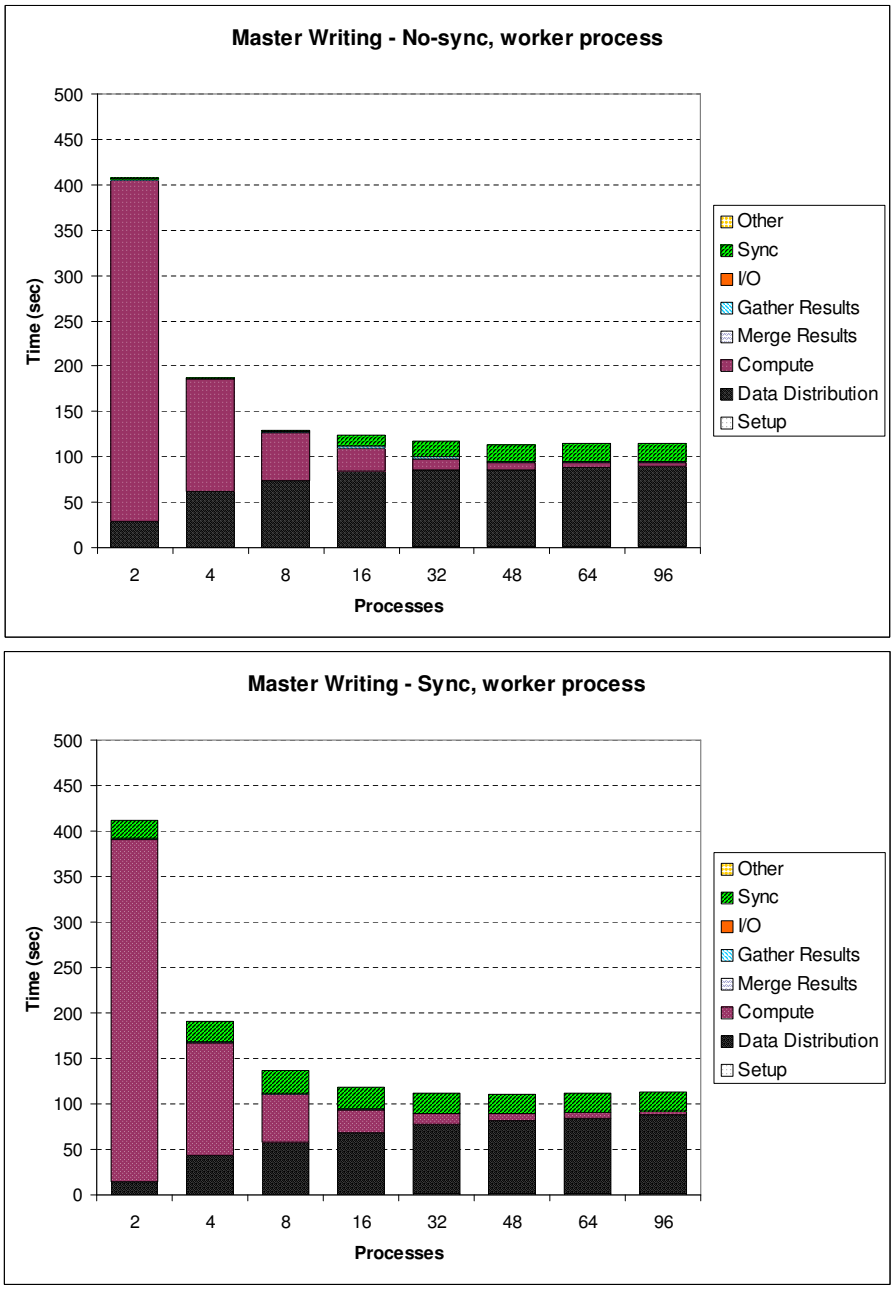

Figure 6.3. Individual phase timing results when scaling up the number of processors with no-sync/sync query options for MW.

without sync since the inherent synchronization in collective I/O means that synchronizing after all workers do I/O is negligible in our test cases. With the query sync option, WW-Coll

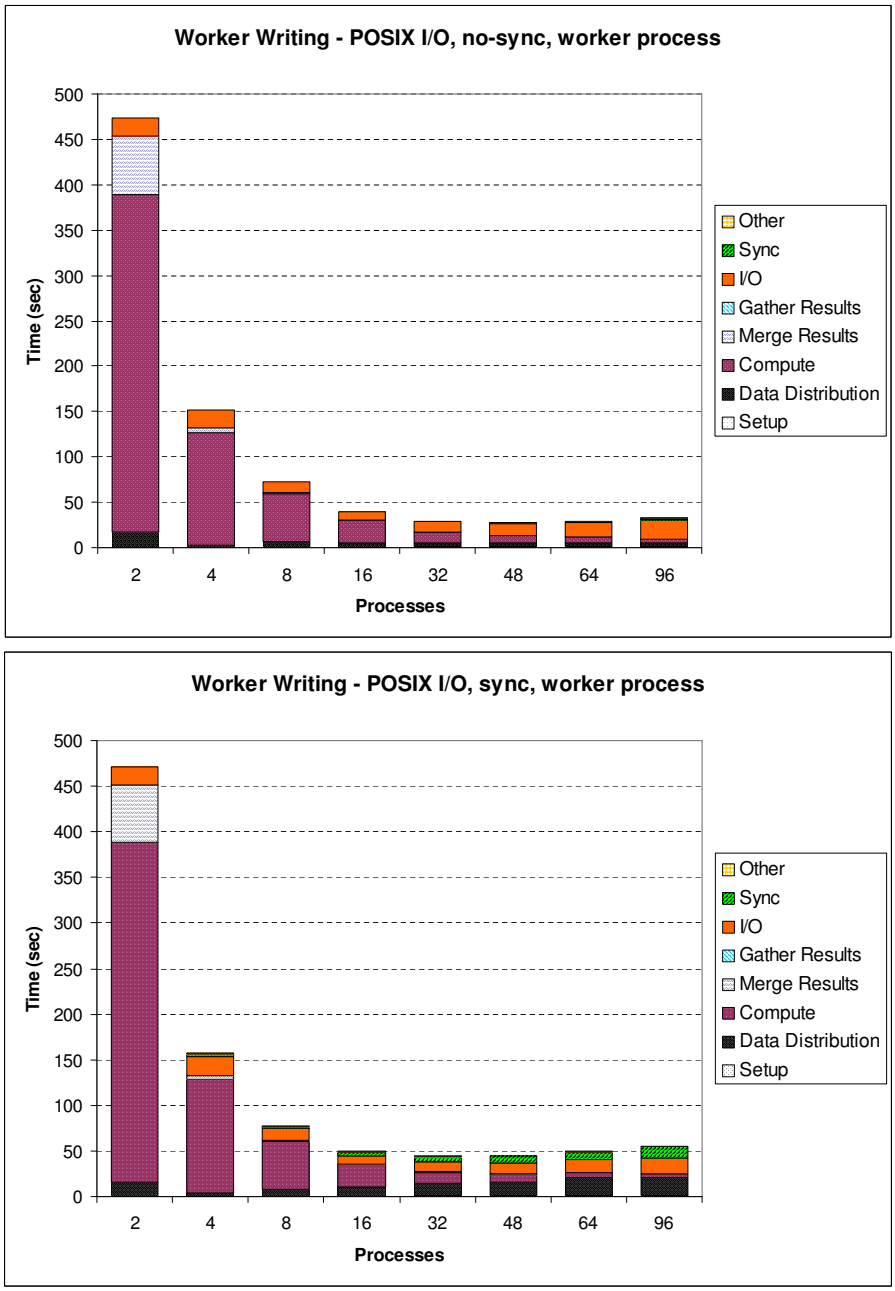

Figure 6.4. Individual phase timing results when scaling up the number of processors with no-sync/sync query options for WW-POSIX.

(45.54 secs) gets closer to WW-List (40.24 secs) at 96 processors. In overall execution time

at 96 processors, WW-List outperforms the other I/O strategies by 364% (MW), 33% (WW-POSIX), and 75% (WW-Coll) in the no-sync cases and 182% (MW), 37% (WW-POSIX), and 13% (WW-Coll) in the sync cases. Noticeable performance gains due to adding more workers slowed considerably at about 32 processes. Generally, at this point the I/O phase time was dominant and even increased slightly with more processes due to the increased frequency of I/O requests (from shorter query processing times). A larger file system configuration with more I/O bandwidth may have provided more scalable I/O performance.

Figures 6.3, 6.4, 6.5, and 6.6 show the overall breakdown of each run with respect to the different timing phases and different I/O strategies. We present both the no-sync and sync results from the workers to illustrate how each I/O strategy is affected by adding a forced synchronization component. The effect of forced synchronization to MW makes a negligible performance difference (a maximum of 5% in overall execution time mostly due to test variance). Since the workers all wait until the master does I/O before beginning the next query, the forced synchronization is cheap.

WW-POSIX is largely affected by synchronization (up to 69% in overall execution time). The sync phase time increases due to the forced synchronization (from 1.01 secs to 12 secs at 96 processors), which also increases the data distribution phase time (from 3.21 secs to 19.04 secs at 96 processors). The WW-POSIX I/O phase time actually decreases steadily from no-sync to sync since each worker is writing data at a slower rate and handing out less I/O ops/s (up to 17% I/O phase time decrease at 96 processors). This slower rate of I/O ops/s allows PVFS2 to improve pure I/O performance slightly.

WW-List is moderately affected by synchronization. Since its overall I/O costs are always less than WW-POSIX, the sync phase time increase from no-sync to sync is less than WW-POSIX (from 0.41 secs to 5.87 secs at 96 processors). Similar to WW-POSIX, the sync phase

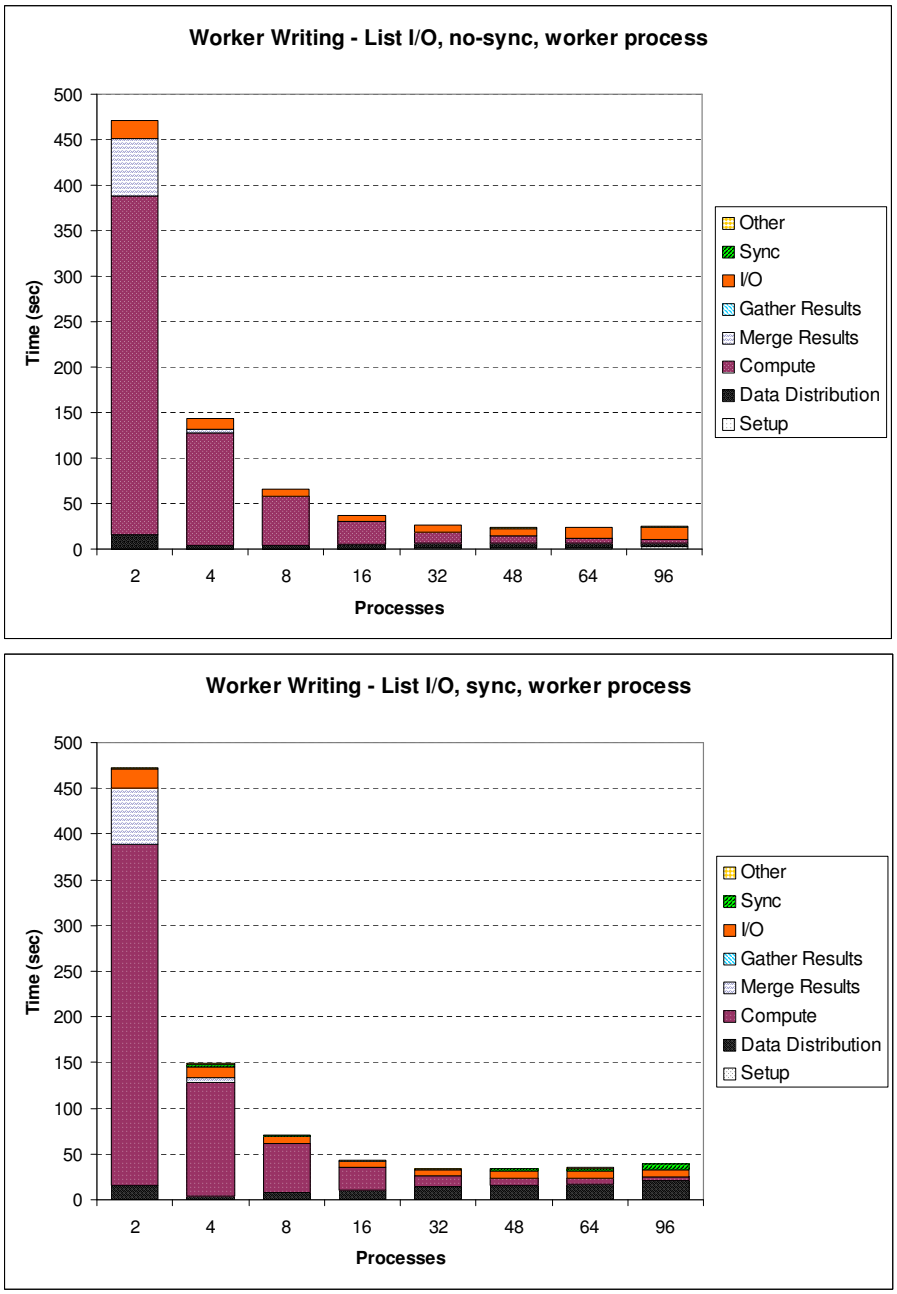

Figure 6.5. Individual phase timing results when scaling up the number of processors with no-sync/sync query options for WW-List.

time increase causes the data distribution time to rise (from 4.47 secs to 18.47 secs at 96

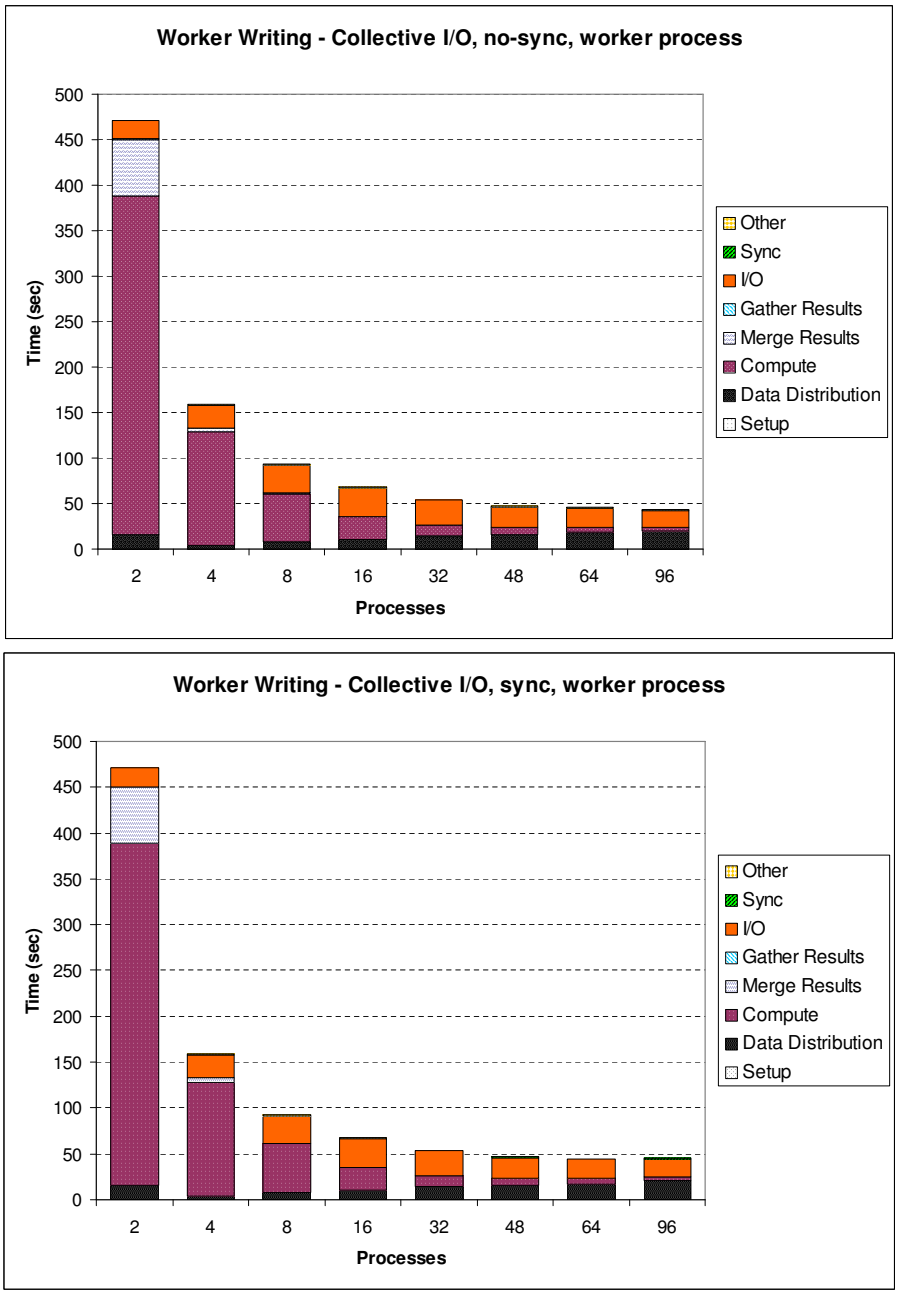

Figure 6.6. Individual phase timing results when scaling up the number of processors with no-sync/sync query options for WW-Coll.

processors). WW-List also shows an I/O phase time reduction of 34% at 96 processors due to less stress on the file system.

WW-Coll is at most affected by 6% in moving from no-sync to sync cases. Since its own inherent sync bears most of the forced synchronization costs, adding an additional synchronization after I/O is quick and negligible in overall execution time. While collective I/O methods generally outperform POSIX I/O methods [**19**], we note that WW-Coll loses to WW-POSIX in the no-sync case. As we described in Section 6.1.3, the inherent synchronization of WW-Coll is more costly than the gains in I/O performance alone over POSIX I/O. While workers are waiting to do collective I/O after processing their portion of the query, they are wasting time, which shows up in the data distribution time.

Our second suite of tests held the number of processors constant at 64 and examined how improving the compute time would affect each I/O strategy. As mentioned in at the beginning of the chapter, improving the compute time could be accomplished in hardware (new processors, custom search hardware) or software (improved search algorithms).

Our first suite of tests in Figure 6.2 used compute speed = 1, (which we refer to as the base compute speed). This test suite (Figure 6.7 varied the compute speed parameter from 0.1 to 25.6. First of all, we note that increasing the compute speed up to 25.6 times faster than the base compute speed made less than a 2% difference in overall execution time for both the no-sync and sync cases for MW. Clearly, the application phases besides the compute phase are the bottleneck here. The other strategies faired much better than MW. The individual WW strategies (WW-List and WW-POSIX) both surpassed WW-Coll and MW significantly in the no-sync cases, indicating that individual WW strategies will strongly benefit from hardware or software improvements on the compute phase. In overall execution time with compute speed = 25.6, WW-List outperformed the other I/O strategies by 592% (MW), 32% (WW-POSIX), and 98% (WW-Coll) in the no-sync cases and by 444% (MW), 65% (WW-POSIX), and 58% (WW-Coll) in the sync cases. Similar to the first suite

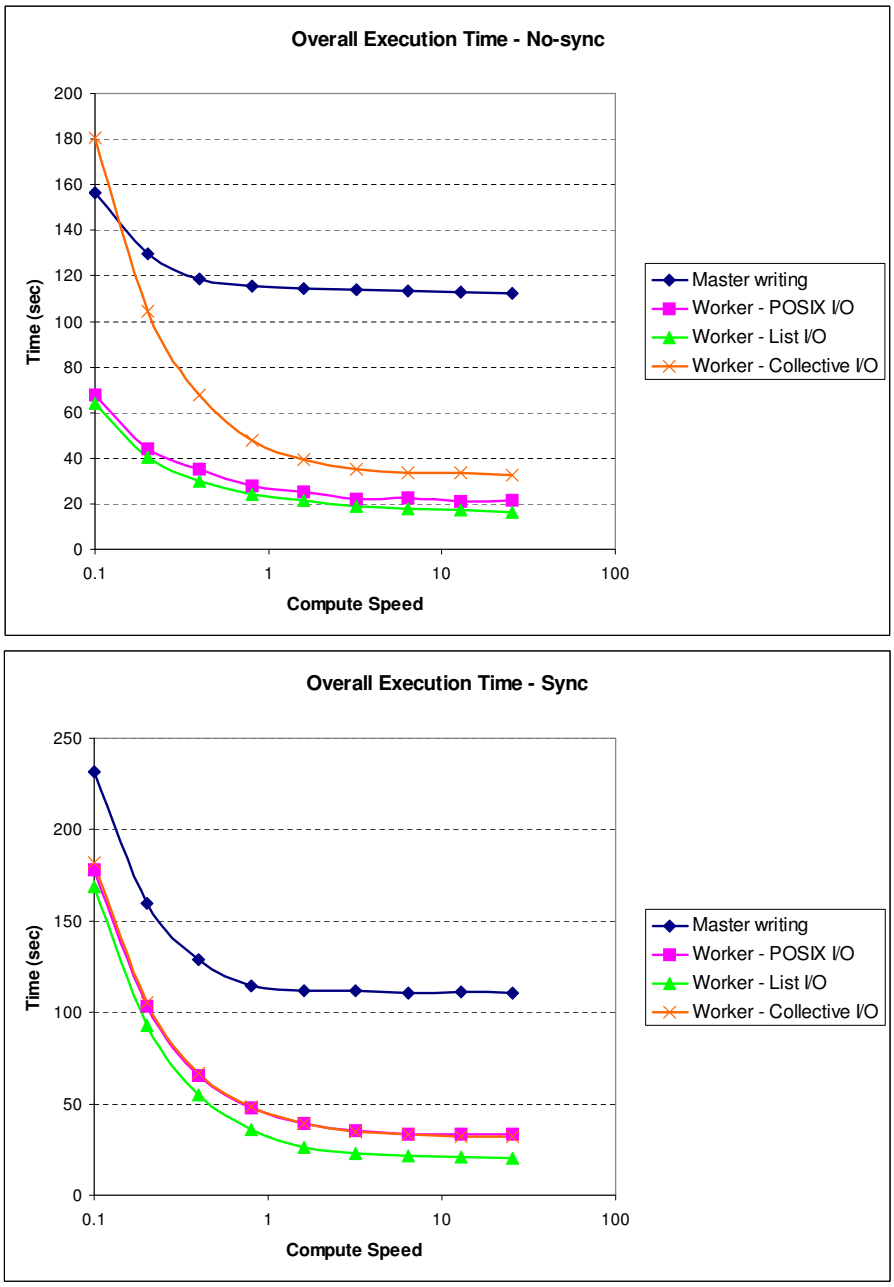

Figure 6.7. Results when scaling up the compute speed with no-sync/sync query options.

of tests, I/O times generally start slightly increasing as we improve the compute speed due to more I/O ops/s.

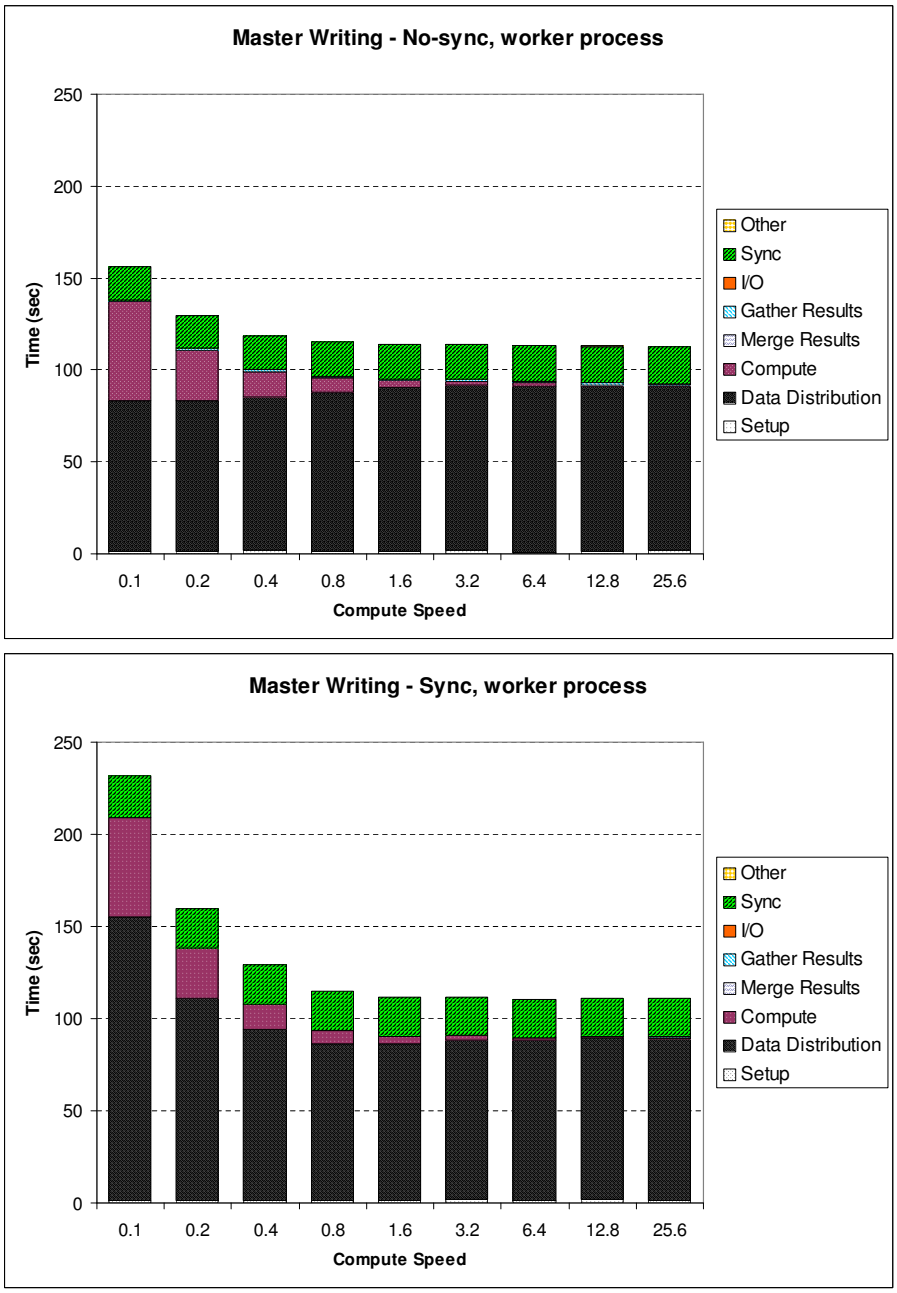

Figure 6.8. Individual phase timing results when scaling up the compute speed with no-sync/sync query options for MW.

Figures 6.8, 6.9, 6.10, and 6.11 show the overall breakdown of each run with respect to the different timing phases and different I/O strategies. As the compute speed increases,

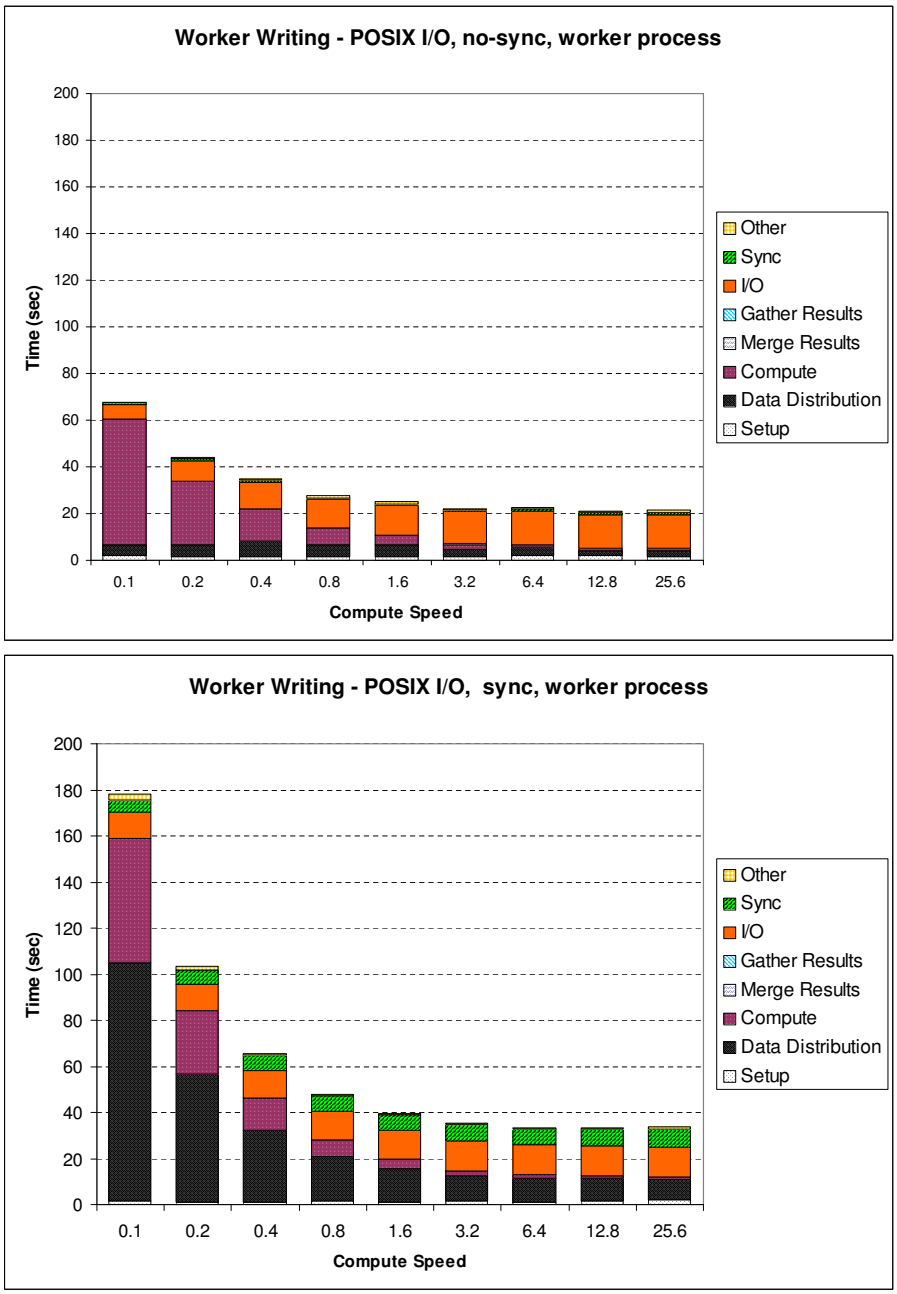

Figure 6.9. Individual phase timing results when scaling up the compute speed with no-sync/sync query options for WW-POSIX.

we note that the effect of the compute speed on the overall execution time is reduced. At compute speed = 0.1, workers spend close to an average of 54 secs in the compute phase

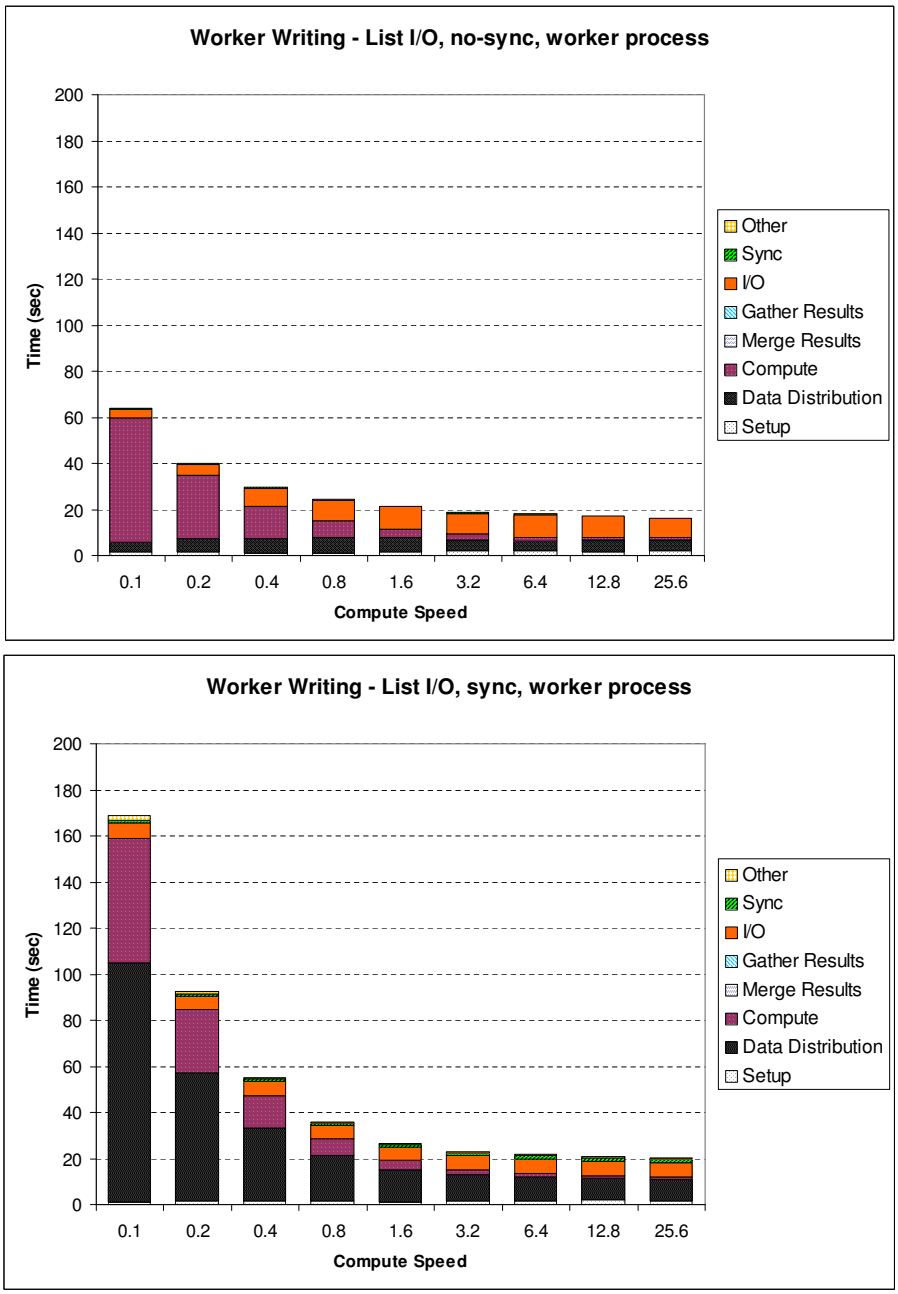

Figure 6.10. Individual phase timing results when scaling up the compute speed with no-sync/sync query options for WW-List.

in both the no-sync and sync cases. At compute speed = 25.6, workers spend slightly more than 0.8 secs in the compute phase in both the no-sync and sync cases. The large variance

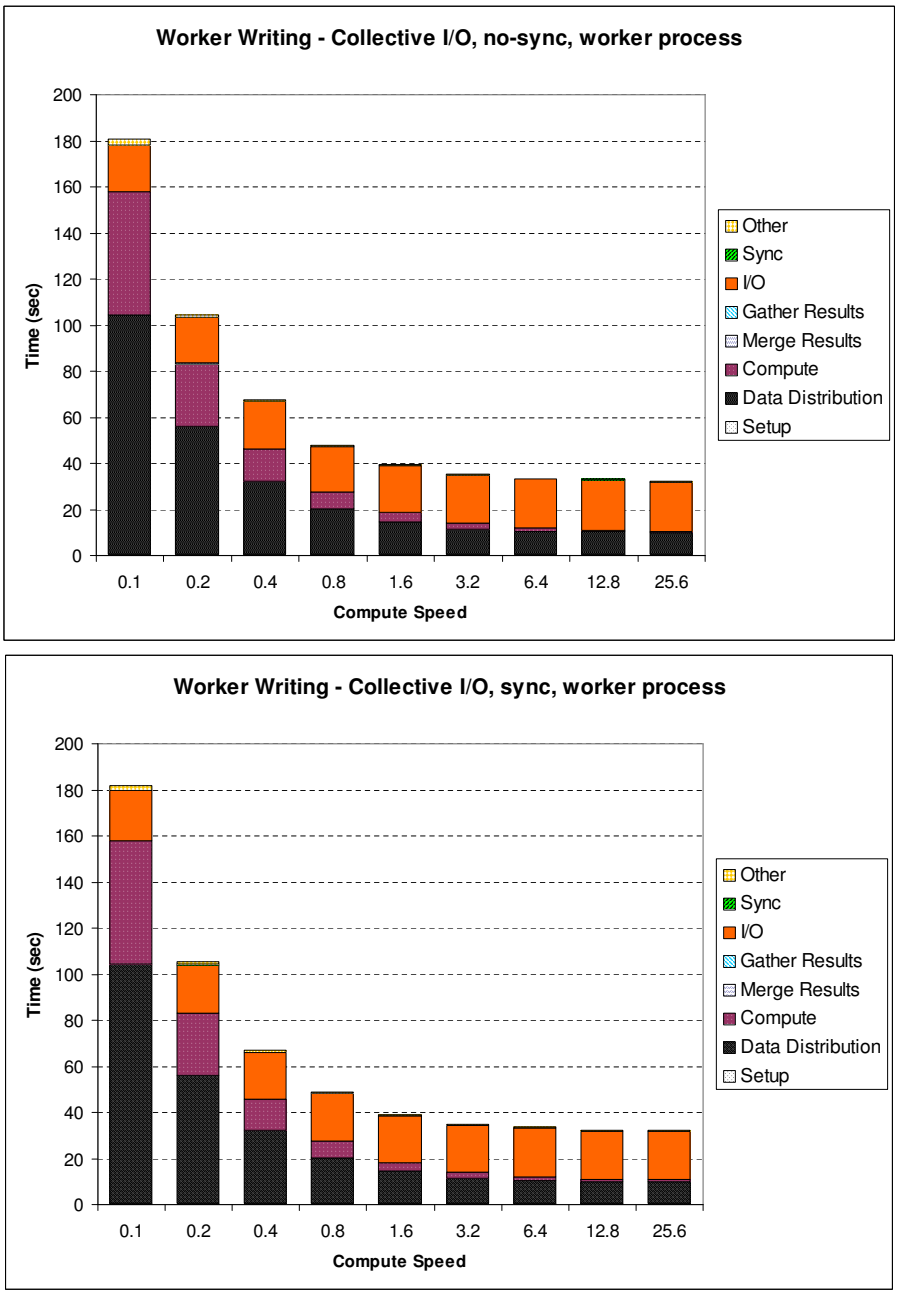

Figure 6.11. Individual phase timing results when scaling up the compute speed with no-sync/sync query options for WW-Coll.

in long compute phase times among workers leads to a large wait time when workers are synchronizing.

At slow compute speeds (0.1 to 0.4) with MW, forced synchronization adds some overhead (48% when compute speed = 0.1). The data distribution phase is mostly to blame as it causes 72.50 secs of the 75.47 secs difference at compute speed = 0.1. Since the absolute compute time variance among workers is high at compute speed = 0.1, the data distribution time is significantly affected. From compute speed = 0.8 to 25.6, synchronization makes little difference with MW (at most 2%).

WW-POSIX is substantially affected by the forced synchronization (at most 162% when compute speed = 0.1) from the high compute time variance. From compute speed = 1.6 to compute speed = 25.6, the overhead of forced synchronization is slightly above 50%. Since the compute time variances are less significant at this point (compute times are less than 4 secs when compute speed = 1.6 and 0.85 secs when compute speed = 25.6), most of the change in execution time is due to the synchronization overhead and the increased data distribution phase time. When compute speed = 25.6, sync phase time increases from 1.09 secs to 7.758 secs and data distribution time increase from 2.30 secs to 9.14 secs when going from no-sync to sync.

Similar to WW-POSIX, WW-List is strongly affected by the large compute time variance at low compute speeds. However, due to its optimized noncontiguous list I/O method, it incurs smaller overhead as sync phase time and data distribution phase time increase (when compute speed = 25.4, sync phase time increases from 0.35 secs to 1.36 secs and data distribution phase time increases from 3.96 secs to 9.36 secs) from no-sync to sync. The benefits of list I/O over POSIX I/O allow WW-List to provide improvements that show up in I/O phase time, sync phase time, and data distribution phase time.

WW-Coll is hardly affected when going from no-sync to sync (at most 4%). Again, since the inherent synchronization in collective I/O pays for the variance in compute times among

workers, the trend of seeing relatively higher data distribution times as in the other I/O strategies is not present in the sync cases. In general, when compute time variance is large, WW-Coll always pays a high synchronization cost unlike individual WW strategies. Two-phase I/O in ROMIO was not as efficient as list I/O with synchronization in almost all of our test cases. A collective I/O implementation based on list I/O might be appropriate for access patterns similar to parallel sequence-search.

6.4. Summary

In summary, this chapter provides several contributions in understanding I/O strategies for parallel sequence-search tools. While these results are based on search applications similar to mpiBLAST and pioBLAST, the performance trends we observed can be extrapolated to other parallel sequence-search applications.

- **Proposed individual worker-writing I/O strategies for parallel sequence-search tools** - The individual WW strategies outperformed the other I/O algorithms in all no-sync test cases. WW-List beat all I/O methods in both no-sync and sync test cases.
- **Developed S3aSim for understanding phase interaction of parallel search algorithms** - S3aSim is a flexible tool for understanding how various I/O strategies perform when using database segmentation.
- **Detailed the possible cost of synchronization with collective I/O in a full application simulation** - To date, most I/O studies compared I/O methods in I/O-only benchmarks, which does not expose the I/O synchronization penalty in collective I/O. Our study also suggests that in some cases, a collective I/O method

implemented with list I/O and forced synchronization may be a more efficient collective I/O method than the default two phase I/O method in ROMIO.

In the future, we would like to pursue further research on I/O strategies with S3aSim. There are many other input variables that can significantly affect overall application performance such as different I/O characteristics (different query sizes, databases, amount of results), hybrid query segmentation/database segmentation strategies, new I/O algorithms, as well as many others. We hope that our work will aid in the development of future parallel database search tools in conjunction with modern parallel file systems.

CHAPTER 7

Noncontiguous Locking Techniques for Parallel File Systems

Researchers in fusion (GTC [**43**]), combustion (S3D [**79**]), molecular dynamics (NAMD [**74**], Desmond [**7**], and Blue Matter [**30**]), astrophysics (FLASH [**31**]) and many other fields are achieving scientific breakthroughs by using large-scale computing systems to simulate experiments which are difficult to pursue in the physical world. CPU, memory, and network components have made great performance leaps that have enabled new problems to be tackled on these resources. Storage systems, however, have lagged significantly. Because of this disparity, scientists find themselves hamstrung by the I/O system in their ability to store simulation results. Once these results are stored, the relatively slow I/O systems further limit the rate at which results can be analyzed. The addition of computational power and memory means that simulations may be performed at a higher data resolution or with more time steps, which compounds an already difficult situation for the storage system.

I/O system software and middleware is used to mitigate this situation. Parallel file systems provide a way for I/O bandwidth to scale on par with their computational counterparts. One area of progress in I/O has been the addition of the MPI-IO interface in 1997 [**56**], which provides the same portability and rich access pattern descriptions for parallel I/O as MPI did for parallel computing. Additionally, high-level I/O libraries built on top of the MPI-IO interface, such as HDF5 [**37**] and parallel netCDF [**47**], provide programmers with high-level I/O APIs and portable file formats. When scientists can use high-level descriptions for the access pattern of their application, optimizations such as datatype I/O [**18**] and collective

I/O [**29**] can be applied to significantly improve performance. One implementation of collective I/O, two-phase I/O [**93**] (unrelated to the two-phase lock protocol), provides good performance in many cases and can help solve the atomicity problem. Two-phase I/O, however, requires that all processes coordinate their I/O and is not suitable for individual I/O and/or unbalanced workloads. Our work can be applied to collective I/O techniques, such as two-phase I/O, although in this chapter we target individual I/O operations.

One area of significant difficulty for the I/O library developers is to support atomicity for these high-level I/O APIs. Atomic high-level I/O operations are useful when regions of data in a file are shared by multiple processes, such as in HDF5 where internal metadata in a file is used by all processes to place application data in a consistent manner. The MPI-IO *atomic mode* consistency semantics guarantee sequential consistency for all I/O operations among processes which opened up a file collectively. This mode may be used to implement the type of sharing described above.

Atomicity is even an issue for contiguous I/O operations. For example, one process *P0* may write a contiguous region of data which spans two I/O servers *I0* and *I1*. Another process *P1* may read the same regions of data. If *P0* writes to *I0* first and *I1* second and *P1* reads from *I1* first and *I0* second, then *P1* may see only part of *P0*'s write, violating atomicity.

The most common technique for implementing atomicity is to use file locking, and most file locking implementations are limited in their ability to describe noncontiguous regions. Unfortunately, noncontiguous access patterns are common in scientific applications [**3, 27**]. Because file locking implementations perform so poorly for these patterns, the MPI-IO atomic mode is rarely used. The lack of an efficient atomicity implementation for high-level I/O APIs, such as MPI-IO, has led many parallel application designers to simply have each

process write to its own file and manage this collection of files with scripts and custom post-processing tools. This is a very inefficient solution, and it lowers the scientist's productivity.

Efficient atomic noncontiguous I/O operations could be used for many producer-consumer problems, including real-time visualization of data, and are critical for in-place data manipulation techniques. In addition to giving support to high-level APIs, efficient atomic noncontiguous I/O operations are an important building block in software-level RAID techniques and file system journaling, both of which are growing in importance as we build file systems from ever-larger collections of storage devices.

Several solutions have been proposed to provide atomic I/O for the file system. Most of them use some form of locking for handling atomicity. Solutions in ROMIO [**77**], the most popular implementation of MPI-IO, include the use of byte-range locking across the entire access pattern and file locking with MPI one-sided communication [**45, 94**]. While these approaches offer some important benefits, they are only useful within the context of MPI-IO accesses and rely on MPI-2 calls that are not available on many large systems. More general solutions approach the problem at the file system level. Atomicity at the file system level typically uses some fixed size granularity that is a multiple of the file system block size.

In this chapter, we propose using a scalable distributed lock manager (DLM) architecture which has true byte-range granularity for handling atomicity within shared files. We present list and datatype locking methods that leverage high-level noncontiguous access pattern information and hybrid two-phase lock protocols that make the best use of our new locking methods. We provide synthetic performance evaluations which test the scalability of our lock methods in non-overlapping and overlapping situations. We also test our ideas against two scientific I/O benchmarks, the S3D I/O benchmark and S3aSim, which show our DLM can achieve near lock-less I/O performance. Additionally, we compare our DLM architecture

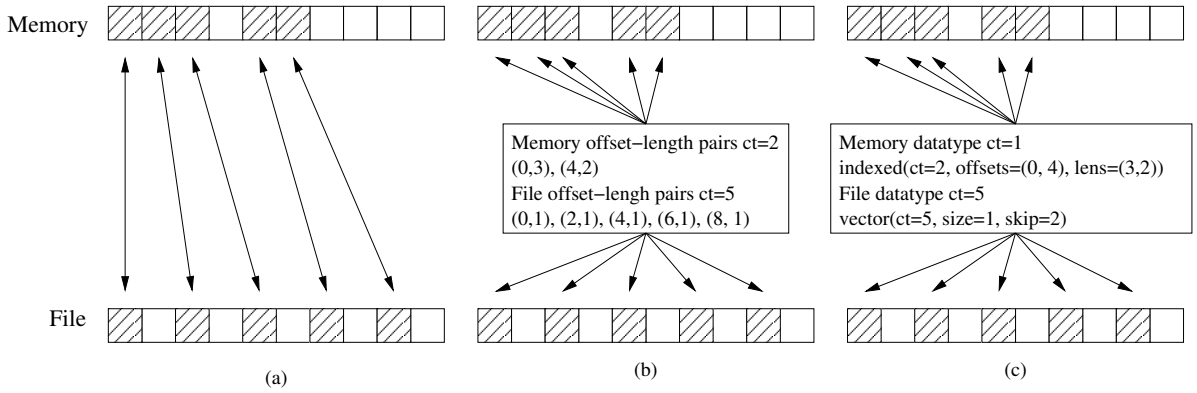

Figure 7.1. The three lock methods described in this chapter: (a) POSIX locking, (b) list locking, and (c) datatype locking.

against a block-based cache and locking Lustre implementation to show how false sharing negatively impacts shared file I/O performance.

Our chapter is organized as follows. In Section 7.1, we discuss previous work for implementing atomic I/O in detail and the basic ideas of our research. In Section 7.2, we describe how we leverage high-level I/O access pattern information to create efficient lock interfaces. In Section 7.3, we explain how we combined the ordered, rigorous two-phase lock protocol with an optimistic lock protocol to improve overall lock performance. In Section 7.4, we discuss the implementation of our DLM at both the client and the server. Additionally, we describe how we translate MPI-IO operations into client lock requests. In Section 7.5, we present the results of our lock tests, S3D I/O benchmark, and S3aSim in a detailed performance analysis. In Section 7.6, we summarize this chapter and discuss possibilities for future work with our DLM.

7.1. History & Our DLM Approach

Most file systems today follow the POSIX standard for I/O [**39**], which states that a read operation, that can be proven to occur after a write, must return the new data (the

entire write should be visible). Further, I/O operations should be "atomic," where they are only seen in their entirety, or not at all. The MPI-IO atomic mode is similar: it guarantees sequential consistency of writes and reads to a group of processes which have collectively opened a file. The POSIX I/O API limits atomicity to contiguous regions of the file. MPI-IO, however, also supports atomic noncontiguous I/O operations.

Parallel file systems today do not have an optimized approach for handling efficient atomic noncontiguous I/O access. IBM's GPFS [**82**], which is POSIX compliant, has an optimized MPI-IO implementation [**75**] for improving collective I/O access. It has a global lock manager which hands out "lock tokens" to clients, which helps with optimizing client-side caching. GPFS also employs a special lock division algorithm for acquiring locks on a file which splits the file contiguously among processes. While this may work well for certain cases, it is not generally well suited for interleaved access. For interleaved access, GPFS relies on its data shipping mode for the best performance. Additionally, while GPFS allows byte-range locks, it rounds them to the file system block size, which causes false sharing when writes are not aligned.

Lustre [**52**] also follows the POSIX standard (except with regard to atime updates and flock()/lockf() system calls) and has a DLM incorporated into its object storage servers (OSSs). The OSSs are responsible for locks on the data they manage. The lock granularity on Lustre is the file system page size, where client conflicts are resolved by sending a message to the lock owner to release the page. All cache pages must be flushed before a lock is released. Panasas [**59**], while supporting file locking, does not appear to have any special support for atomic MPI-IO operations. The Chubby lock service [**9**], used by Google's Bigtable [**12**], is a coarse-grain whole-file lock service with a design emphasis on availability and reliability. The Google file system [**32**] relaxes most file system consistency semantics

and implements a special "atomic append" operation. The Frangipani file system [**96**] has a single-writer/shared-reader whole-file lock service that interacts with its Petal servers [**46**]. NFS version 3 [**88**] introduced support for file locking, while NFS version 4 [**89**] also added simple byte-range locking. UNIX provides whole-file locking via the file locking function `flock()` and byte-range locking via the function `fcntl()`, as specified in the POSIX standard [**68**].

False sharing, which causes serialized I/O operations, occurs when write sizes do not align with the file system lock granularity. In addition to the overhead of serialized I/O operations, file systems that use their lock systems for caching must also flush cache pages before transferring locks among clients. Deadlock avoidance in file systems has been primarily based on the two-phase lock protocol. Our scalable DLM approach seeks to provide efficient noncontiguous lock operations that do not incur any false sharing through true byte-range granular locks. We use high-level access pattern information to create efficient lock APIs for our lock service. Additionally, we have modified the ordered, rigorous two-phase lock protocol to increase performance, while avoiding deadlock (when there are no failures).

7.2. New Locking Methods

Application developers commonly use rich abstractions to describe their data access. In particular, they use high-level I/O libraries such as HDF5 or parallel netCDF, which additionally offer portable file formats. Many high-level I/O libraries, such as those previously mentioned, are built on top of MPI-IO, which is a portable, parallel I/O API. However, some scientific applications are written to directly use MPI-IO. In either case, high-level APIs for I/O can provide opportunities for significant performance improvement. In [**18**], MPI derived datatype abstractions were efficiently supported by the file system and showed up to

several magnitudes of performance improvement over basic POSIX I/O file system operations. We aim to apply these same datatype abstractions to support efficient noncontiguous atomic operations. In following subsections, we describe three different lock methods for implementing atomic noncontiguous I/O.

7.2.1. POSIX Locking

File systems typically present programmers with the POSIX I/O API. POSIX read and write operations use a pointer to a contiguous region of memory, the current file pointer location, and a count, to access data. We denote the locking counterpart of this interface, which locks a contiguous region of bytes, as *POSIX lock*. The POSIX lock interface can be used to make a noncontiguous access pattern atomic by acquiring all locks to the necessary file regions before doing any I/O as shown in Figure 7.1a. While POSIX locking can provide atomicity guarantees for noncontiguous I/O access patterns, it has several important drawbacks. The number of lock requests to the lock servers is at least equal to the number of noncontiguous file regions, which creates a significant request processing overhead. When lock regions span multiple lock servers, the number of lock requests increases further due to "splitting" the locks on lock server boundaries.

7.2.2. List Locking

Using list-based descriptions is a technique that has been used for I/O [**17**] and in a prototype single lock server [**1**]. The concept of list I/O (illustrated in Figure 7.1b) extends the POSIX I/O API to specify multiple noncontiguous regions in both memory and file. Using this technique for locking provides a way to take advantage of the high-level I/O information available from MPI derived datatypes (i.e. we can enumerate the regions).

Some I/O access patterns may have a large number of noncontiguous file regions. We split up list lock requests to the lock servers at every 64 noncontiguous regions since requests should not be arbitrarily large. When application programmers perform unstructured atomic data access, lists of offsets and lengths are a concise way to describe the access patterns to the lock servers. However, offset and length pairs do not concisely capture structured access patterns. When noncontiguous regions have less than 16 bytes, the list locking description with offsets and lengths exceeds the actual data amount being locked.

7.2.3. Datatype Locking

Many simulation applications use multi-dimensional arrays to model scientific events, such as protein folding, combustion, or fusion. It is common that the I/O accesses in these data sets, for instance writing one variable for every cell, is regular and structured. Structured data access can be concisely described with a derived datatype. When structured data access requires atomicity, we can use the derived datatype concept with our DLM. The datatype access pattern, shown in Figure 7.1c, consists of a tree of datatypes as opposed to the offset and length pairs used in list locking. When moving the access pattern description across a network, datatype locking reduces network traffic and the number of lock requests to the lock servers. The lock servers unravel the derived datatypes to determine which locks they are responsible for. If the lock servers were to lack significant processing capabilities, this discovery process could outweigh the benefit of reduced lock requests and network traffic. Additionally, when presented with access patterns containing no regularity, datatype locking breaks down into list locking.

7.3. Hybrid Lock Protocols

Deadlock is always a potential problem when multiple locks are acquired and then released. A variant of the well known two-phase lock protocol [**5**], rigorous two-phase locking, serializes the order in which operations complete while allowing parallelism on nonconflicting regions. Adding order to the locks acquired in the rigorous two-phase lock protocol eliminates the possibility of deadlock when there are no failures by lock system participants. An ordered, rigorous two-phase lock approach is a good match for noncontiguous file system operations since an order can be imposed based on file offsets. The ordered, rigorous two-phase lock protocol separates the operations into two phases: a growing phase and a shrinking phase. During the growing phase, locks must be acquired in a monotonically increasing order. After doing the necessary work on the locked regions, the client enters the shrinking phase to release all the locks it is holding. During the growing phase, no locks that have been acquired in-order can be released. Similarly, during the shrinking phase, no locks can be acquired. If there are several clients all waiting on various locks, once the "highest" one in the ordering is released, another client will be able to make progress. This client will possibly acquire more locks (assuming all other clients are waiting on other locks) and then release its locks, which allows another client to proceed. This strategy eliminates the possibility of deadlock assuming that clients and lock servers do not fail. For brevity, in the rest of the chapter, we refer to the ordered, rigorous two-phase lock protocol as simply the two-phase lock protocol. To improve upon the performance of the two-phase lock protocol, we propose using an optimization that will significantly enhance I/O performance based on the knowledge that it is atypical for a programmer will overwrite their own data. Our optimization is an optimistic lock protocol where clients try to acquire their locks from

all lock servers and then release locks that are out-of-order. A simple example would be that client A wants to acquire locks on offset-length pairs (0, 2), (4, 2), (6, 2), and (8, 2). Offset-length pairs (0, 2) and (4, 2), are on lock server 0. Offset-length pairs (6, 2) and (8, 2) are on lock server 1. Client A optimistically tries to acquire all locks from both lock servers simultaneously. Client A waits for the responses from both lock servers and then revokes locks which are out-of-order. If client A has received locks with the offset-length pair (0, 2) from lock server 0 and offset-length pairs (6, 2), (8, 2) from lock server 1, it must release (6, 2) and (8, 2) before deciding whether to retry the optimistic lock protocol or use the two-phase lock protocol.

Incorporating the partial use of an optimistic lock protocol is very important to achieving maximum performance from our optimized lock methods. If a client must in-order acquire every lock through all its noncontiguous file regions, the communication and processing overhead would be tremendous, as is demonstrated in Section 7.5 with POSIX locking. Below we discuss two combinations of the two-phase and optimistic lock protocols, which helps us to achieve better locking performance in nearly all I/O access patterns.

7.3.1. One-try Lock Protocol

Using only the optimistic lock protocol to acquire locks might cause a set of clients to end up in a state of livelock, where no locking progress is made. In our *one-try* lock protocol, a client first tries the optimistic lock protocol to acquire locks. If it does not acquire all its locks, it releases the out-of-order locks and then reverts to the two-phase lock protocol of acquiring locks in-order. Most I/O access patterns are not overlapping and therefore benefit from the ability to acquire locks from all servers simultaneously.

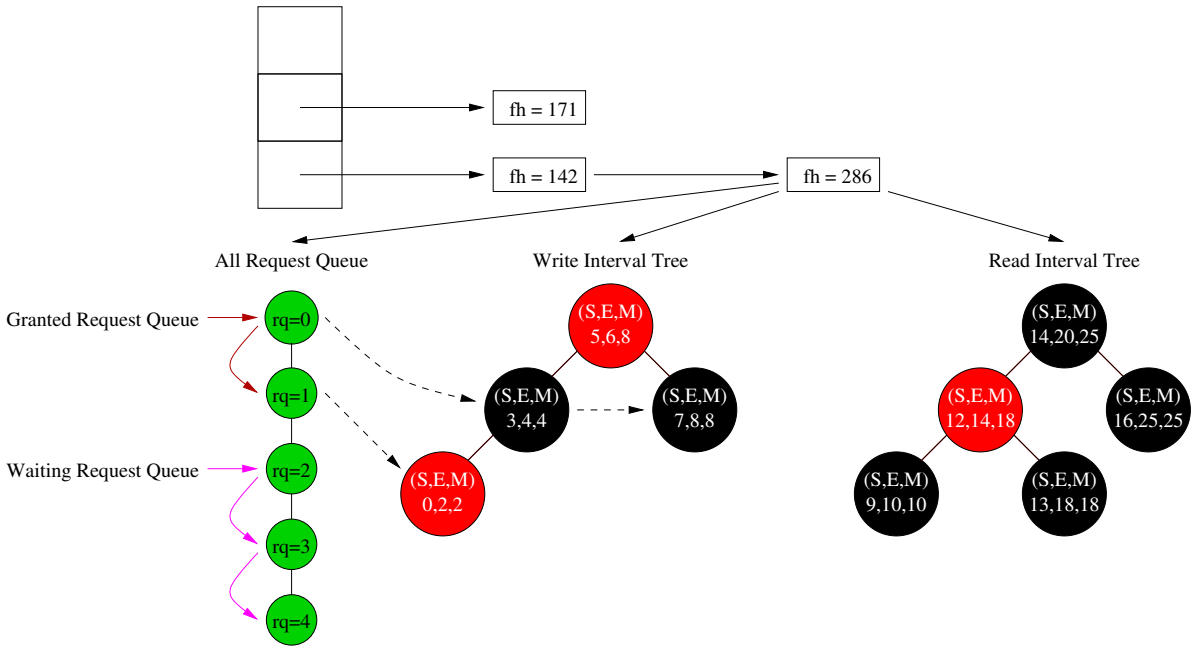

Figure 7.2. Lock server architecture.

7.3.2. Alt-try Lock Protocol

While there are an endless number of ways to combine optimistic and two-phase lock protocols, we felt that alternating protocols would ensure that progress is made while providing ample opportunity for optimistically acquiring locks. The *alt-try* lock protocol first optimistically tries to acquire all locks, releasing those which are out-of-order. Then it uses the two-phase lock protocol to get its next lock. Upon successfully acquiring its next lock, it again tries to optimistically acquire all locks, releasing those which are out-of-order. This alternating strategy repeats until all locks have been acquired.

7.4. DLM Implementation

Our DLM implementation has two major components: clients and lock servers. Both components are integrated into the file system. We chose to implement our DLM into

PVFS2 although the ideas could certainly be ported to other parallel file systems or written into a stand-alone DLM package. This design choice was made for a variety of reasons. As other distributed file systems with atomic capabilities have noted [**52**], separating the DLM from the file system makes handling failure significantly more difficult. The I/O servers and lock servers may lose communication. If clients lose communication with either the lock server or the I/O server, coordination issues could cause non-atomic behavior. Another reason for the integration is that lock requests can use the same server mapping as I/O requests. Both lock and I/O bandwidths scale as the number of servers a file stripes across is increased. From now on, we refer to I/O servers and lock servers as simply "servers," since they are integrated as one component. PVFS2 was chosen since it has derived datatype support, which provided a starting point for implementing datatype locking. When PVFS2 clients access the file system through the MPI-IO interface, they directly interact with the PVFS2 servers (bypassing the Linux buffer cache on the client). PVFS2 clients can request byte-granular regions of data from the PVFS2 servers. Similarly, client lock requests can access actual byte ranges and are not rounded to a sector or file system block size as in other file systems. Therefore, no false sharing is possible with our lock system. Although other file systems may align locks requests to system block sizes which may cause false sharing, they can still benefit from high-level access pattern information and use our lock methods and lock protocols to reduce the number of overall lock requests to the lock system. Section 7.4.1 and Section 7.4.2, respectively, describe the client and the server components in detail.

In order to be able to make our DLM implementation usable for MPI-IO applications or other I/O libraries built on top of MPI-IO, we implemented the necessary procedures to convert MPI-IO calls into the appropriate lock calls. This work is described briefly in Section 7.4.3.

7.4.1. Client

Our client component is a state machine in PVFS2. It begins a lock request by calculating which servers to access, then uses a lock protocol which is chosen at runtime. It supports all three lock protocols (two-phase, one-try and alt-try). All lock protocols are implemented with the two basic lock rounds: optimistic and two-phase. A heap which contains *last_abs_offset* and *next_abs_offset* from each server is used to figure out which servers need to revoke locks and which server has the next in-order lock.

The optimistic round begins when the client attempts to get all locks at once by sending all its lock requests to all the servers involved. It waits for all the replies, which include the last absolute offset locked (*last_abs_offset*) and the absolute offset of the next lock it is waiting on (*next_abs_offset*), and puts this data into the heap. Then the client retrieves the *min(next_abs_offset)* from the heap. If any server has a *last_abs_offset* greater than the *min(next_abs_offset)*, the client must make a request to the relevant servers to revoke locks up to the *min(next_abs_offset)*. At this point, the round is complete.

The two-phase round begins when a client makes a single lock request to the server which has *min(next_abs_offset)* of the heap. The lock request will try to get the locks from *min(next_abs_offset)* to the next *min(next_abs_offset)* of the heap. At this point the server will only reply when it has acquired the necessary locks requested (the client is blocking during this time). Once the client receives the server's reply with *last_abs_offset* and *next_abs_offset*, the client updates the heap and the two-phase round is complete.

The client implements the two-phase, one-try, and alt-try lock protocols described in Section 7.3 using a combination of these two basic lock rounds. The two-phase lock protocol continually uses the two-phase round. The one-try lock protocol begins with an optimistic

round followed by two-phase rounds (if necessary) to fulfill the rest of the lock operation. The alt-try lock protocol begins with an optimistic round, followed by a two-phase round, and continues to alternate lock rounds until the lock operation is satisfied.

Every lock request stores a lock request number for each server involved in the operation. A release request is implemented by sending every server involved in the lock operation the relevant lock request number.

7.4.2. Server

Servers are not aware of the client lock protocols; they only process acquire (nonblocking or blocking) and release (full or partial) requests. The server uses a variety of data structures for fast lock operations as shown in Figure 7.2. Files which have any locks are in a hash table in each server for fast lookup. Each file has two interval trees associated with it: a write tree and a read tree. The interval trees provide $O(lg(n))$ algorithmic inserts and deletes, allows easy lookups for conflicting locks, and are balanced. Each file also has a queue which contains all requests (*all_req_queue*), a queue of waiting requests (*wait_req_queue*), and a red-black tree of granted requests (*granted_req_queue*). The *wait_req_queue* may have lock requests that are blocking or nonblocking. When the server tries to add locks for lock requests in the *wait_req_queue*, it will ignore the lock requests with nonblocking tags.

For a nonblocking acquire request, the server tries to grant as many locks as possible before it overlaps another lock. If the lock request is a read operation, the server checks the write tree and inserts the lock in the read tree if no conflicts are found. If the lock request is a write operation, the server checks the write tree and the read tree for conflicts before inserting locks in the write tree. This implementation provides byte-granular single-writer/shared-reader lock semantics. All locks for a lock request are also chained in a linked

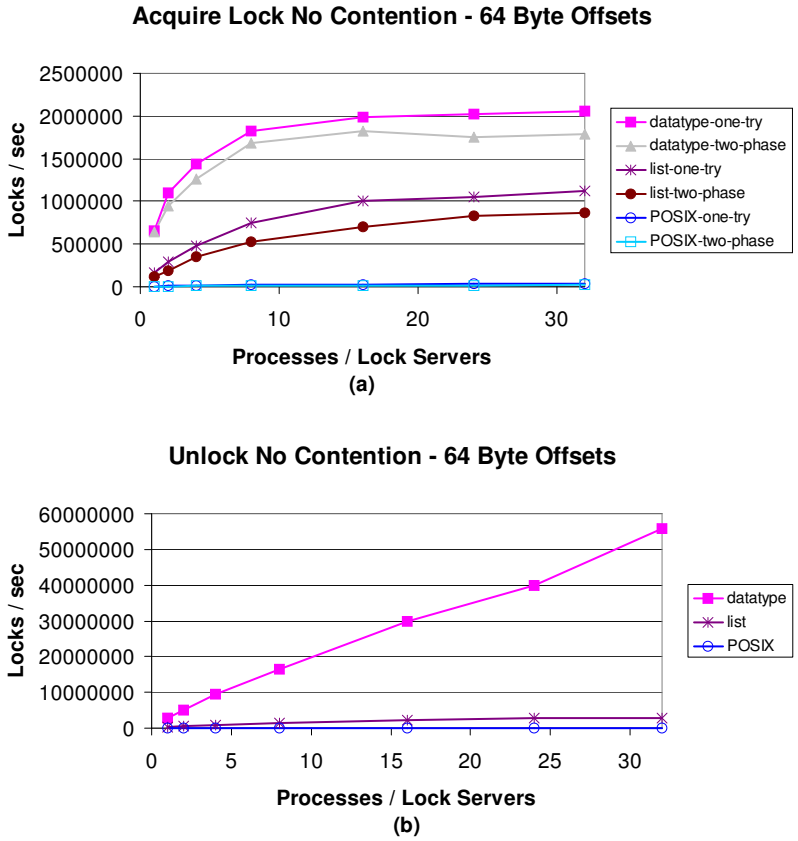

Figure 7.3. Lock tests without contention: (a) acquire and (b) unlock. Each client accesses locks within an 8 MB range.

list for fast removal. If the lock request has been granted, it is added to the *granted_req_queue*, otherwise it is added to the *wait_req_queue* with a nonblocking tag. Then the server returns *last_abs_offset* and *next_abs_offset* to the client immediately. For a blocking acquire request with a specified final offset, the server again tries to grant as many locks as possible before it overlaps another lock. If the lock request reaches the desired offset, the server returns *last_abs_offset* and *next_abs_offset* to the client. If the lock request is completely finished is added to the *granted_req_queue*. Otherwise, it is added to the *wait_req_queue* with a nonblocking tag. If the lock request does not reach the desired offset, the request is also

entered into the *wait_req_queue*, but with a blocking tag. The nonblocking requests in the *wait_req_queue* either add locks or remove locks as specified by later lock requests. The blocking requests in the *wait_req_queue* are checked to see if any locks can be added when locks from other lock requests are released.

Release requests may be full or partial. When the optimistic round is used by a client, partial release requests only remove locks up to a particular absolute offset. A full release typically occurs only after a client has completed all I/O operations protected by its locks. The lock server releases locks by looking up the lock request number in the *wait_req_queue* or *granted_req_queue* and removes the relevant locks using the linked list chain while keeping the interval tree balanced. After the locks are freed, the server returns completion to the client and immediately examines the *wait_req_queue* for lock requests with the blocking tag for servicing. If any of the lock requests in *wait_req_queue* completes, the server notifies the relevant clients.

7.4.3. MPI-IO Implementation

In order to leverage our scalable DLM implementation for MPI-IO and high-level I/O APIs, we modified the PVFS2 device driver in ROMIO, which was developed at Argonne National Laboratories. ROMIO supports many parallel file systems through its abstract device interface for I/O (ADIO). Each file system has its own ADIO device driver. In the PVFS2 device driver we convert MPI file types into PVFS2 derived datatypes for datatype locking, offset and length pairs for list locking, and contiguous regions for POSIX locking. Additionally, we added several new hints to ROMIO for enabling the various lock methods and lock protocols at runtime.

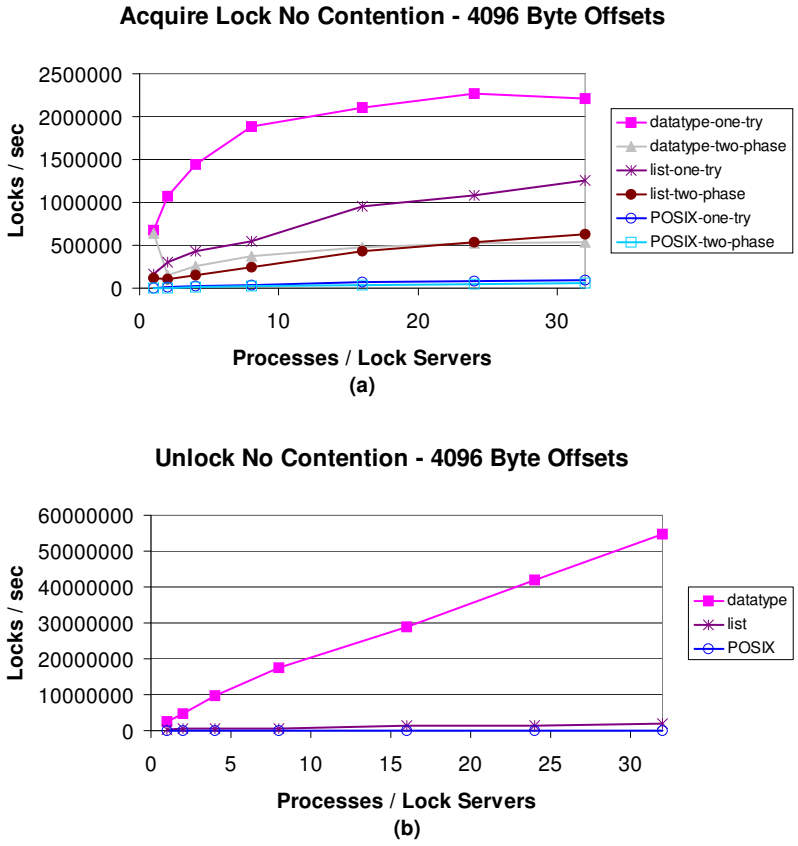

Figure 7.4. Lock tests without contention: (a) acquire and (b) unlock. Each client accesses locks within a 512 MB range.

7.5. Performance Evaluation

We evaluated the performance of our DLM implementation on the Feynman cluster at Sandia National Laboratories. Feynman, composed of Europa nodes, Ganymede nodes, and I/O nodes, has a total of 371 computers with about 160 nodes available at the time of testing. In order to keep our testing as homogeneous as possible, we only used the Ganymede nodes. The Ganymede nodes are dual 2.4 GHz Pentium-4 Xeon CPUs with 2 GBytes RDRAM and 15 GByte Serial ATA hard drives. They are connected with a Myrinet-2000 network, have access to a production Lustre volume, and use the Red Hat Linux Enterprise 2.6.9 operating

system. Since each computer has dual CPUs, we used 2 compute processes per node in all our tests.

We configured our PVFS 2.5.1 file system on up to 32 computers as servers, with one computer also handling metadata responsibilities. On tests where we compare against Lustre, we used 16 servers. We kept the default 64 KByte strip size and other default parameters. Compute nodes access the servers through IP over Myrinet. Our Lustre 1.4.7 test directory was configured to use 16 OSTs with a 64 KByte stripe size to match the PVFS2 configuration. The Lustre lock granularity is aligned to the file system block size for caching reasons and cannot be changed. The Lustre storage nodes (OSSs) are on Infiniband interconnect and connect to the compute nodes via Myrinet to Infiniband routers.

As our DLM uses true byte-range locking, it is not prone to false sharing. In order to understand how this elimination of false sharing would affect performance, we compared our approach to the block-based locking implementation on the Lustre file system on some real-world benchmarks including S3D, a combustion code from Sandia National Laboratories, and S3aSim, a parallel sequence-search algorithm simulator developed at Northwestern University. While, the two systems are not directly compared since they use different hardware, we tried to compare overall trends of how false sharing affects performance. While Lustre has excellent performance in the file-per-process model, shared file performance has been shown to be inefficient due to locking and cache swapping overheads [**48**]. Lustre does not support atomicity at the MPI-IO level; it is supporting the weaker POSIX consistency semantic which only guarantees atomicity for POSIX I/O operations. Although this is a weaker consistency semantic, the overheads of false sharing are still apparent in our benchmarks. Our initial tests runs with Lustre revealed extremely poor performance (less than 1 MByte / sec) from the MPI-IO data sieving optimization [**93**] (most likely due to the locking

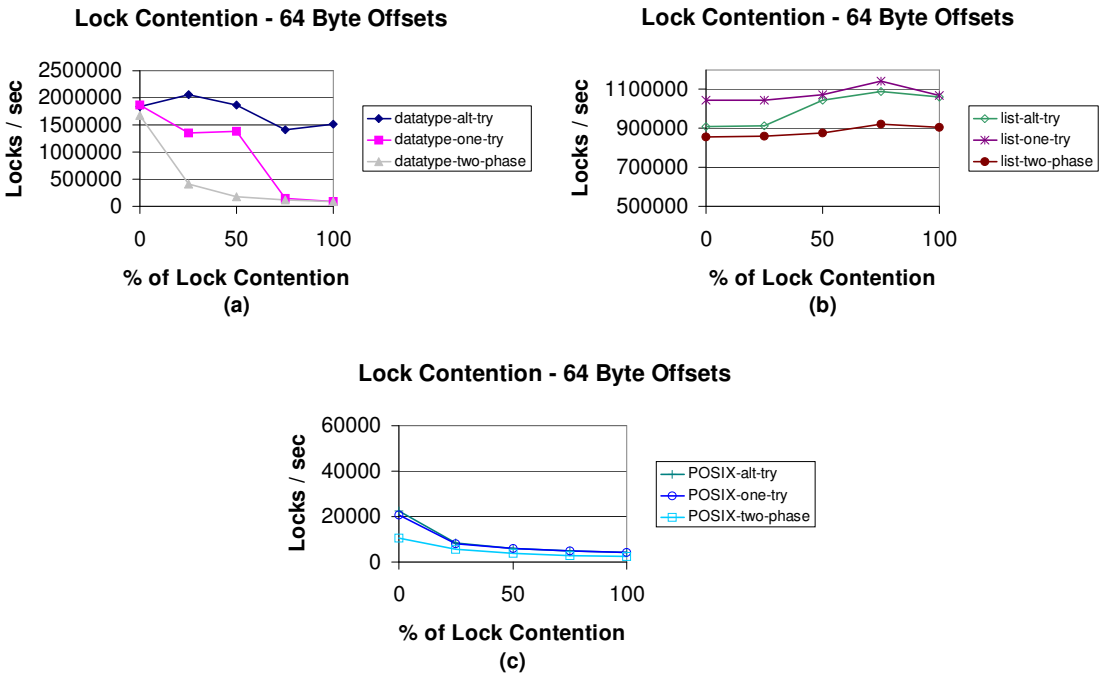

Figure 7.5. Lock tests with contention: (a) datatype, (b) list, and (c) POSIX. Each client accesses locks within an 8 MB range.

and caching overheads). We set hints in ROMIO to turn off data sieving for noncontiguous data access, which improved performance by an order of magnitude in most cases.

We begin our performance evaluation with a series of lock tests to demonstrate the scalability of our DLM and how it efficiently deals with lock contention in Section 7.5.1. We continue our analysis with two application based I/O benchmarks. In Section 7.5.2, we test our DLM in structured data access with the S3D combustion I/O benchmark. In Section 7.5.3, we demonstrate the performance of our DLM in unstructured data access. Each data point was averaged over 3 runs.

We use numerous lock methods in our benchmarks and define them as follows. *Lustre* refers to the Lustre file system and its block-based caching and locking. *no-lock* refers to the PVFS2 file system only doing I/O. *POSIX-two-phase* refers to using the POSIX locking

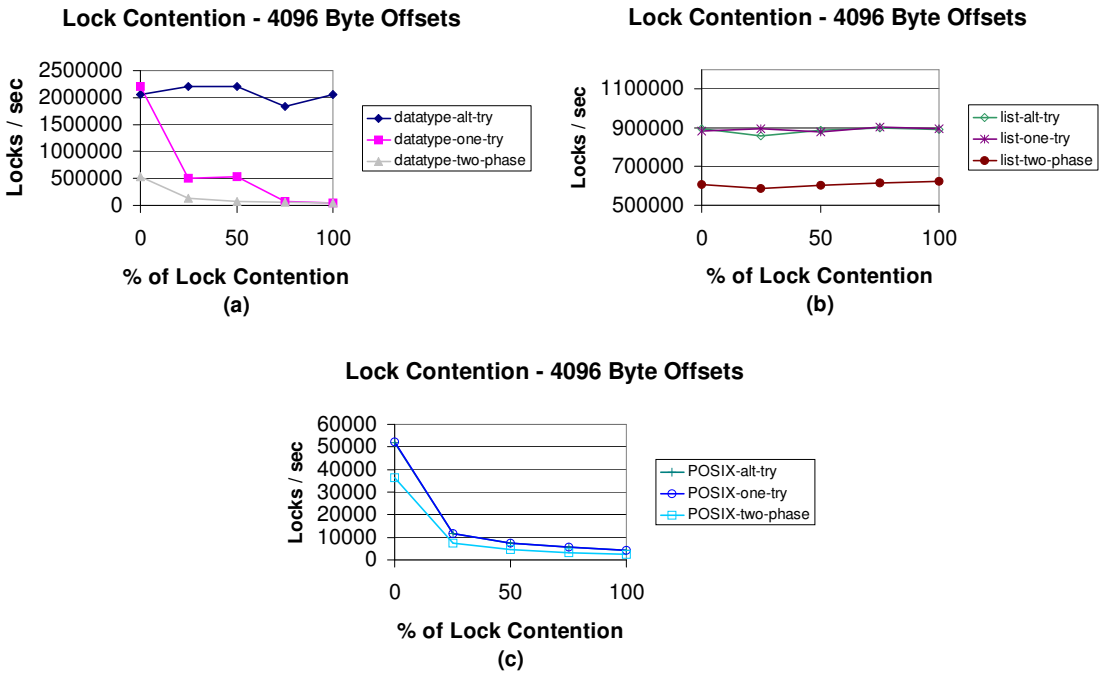

Figure 7.6. Lock tests with contention: (a) datatype, (b) list, and (c) POSIX. Each client accesses locks within a 512 MB range.

method with the two-phase lock protocol. *POSIX-one-try* refers to using the POSIX locking method with the one-try lock protocol. *POSIX-alt-try* refers to the using the POSIX locking method with the alt-try lock protocol. Similar references are made to *list-two-phase*, *list-one-try*, *list-alt-try*, *datatype-two-phase*, *datatype-one-try*, and *datatype-alt-try*, respectively.

7.5.1. Lock Tests

Our lock tests study the locking performance of our DLM directly. We first increased our clients and servers in a 1:1 ratio to see how lock performance scales when given a simple noncontiguous access pattern. Each client attempts to acquire and release 128K locks that are 1 byte long and offset by 64 bytes in Figure 7.3, and 4096 bytes in Figure 7.4, respectively. The benchmark begins with measuring the aggregate acquire time (time for all processes to

acquire all locks), barriers, and then measures the aggregate release time (time for all process to release all locks). The one-try and alt-try lock protocols have identical performance when the locks are non-overlapping and are represented by the one-try lock protocol in this test. Since we scale up the number of processes with the number of servers in this test, a server has at most 128K locks in its interval tree since locks are equally divided among the servers.

In Figure 7.3a, acquiring locks increases very rapidly and then stagnates for all methods. Using the two-phase lock protocol is worse for all lock methods due to the communication overhead associated with contacting each server in-order 4 times at 32 processes and servers (the client access pattern spans 8 MBytes compared to a 2 MByte aggregate stripe size with 32 servers). POSIX-one-try is very slow due to the significant number of 128K lock requests to the lock servers. Datatype-one-try reaches a maximum of 2,059,165 locks / sec, a 51 times improvement over POSIX-one-try (40,532 locks / sec), and a 1.8 times improvement over list-one-try (1,123,928 locks / sec). Lock acquiring cannot increase completely linearly in this test case due to datatype processing. In our implementation, all servers which have some locks locally receive the same access pattern description. They must process this access pattern to figure out which locks they are responsible for. As the number of servers is increased, the processing overhead increases since the access pattern processing engine examines each region to sees if it is the owner. While increasing the number of servers reduces the number of locks per server in a given lock request (assuming uniform distribution), the computational processing overhead is not reduced, and therefore limits scalability.

In Figure 7.3b, we find that unlocking is almost linearly scalable. We only show the unlock bandwidth from each of the three basic lock methods since unlocking is not affected by the acquire lock protocol. In a release request, the server simply finds the matching lock request and removes all locks in a linked list. Since the locks are in an interval tree, keeping

the tree balanced is bounded by $O(lg(n))$. However, in most cases the balancing process is $O(1)$, which provides very good performance for unlocking. At its peak, the datatype method unlocks 55,721,183 locks / sec with 32 clients and servers, which is 20 times faster than list locking and almost 3,000 times faster than POSIX locking. The list and POSIX methods are slower since they keep track of more lock requests. Each client using the list method or POSIX method makes 8K unlock requests or 128K unlock requests, respectively. In comparison, clients using the datatype method make a single unlock request.

In Figures 7.4a and 7.4b, we increase the range of the noncontiguous access pattern by a factor of 64, which increases the overhead for lock methods using the two-phase lock protocol. Since the two-phase lock protocol requires that locks are acquired in-order, each process acquires 16 locks from a server then must ask the next server for next 16 locks. When the lock offset was 64 bytes, the two-phase lock protocol could acquire 1024 locks from a server, before moving to the next server. In Figure 7.4a, all lock methods using the two-phase lock protocol fare poorly in comparison to using the one-try lock protocol. Again, since unlocking is not affected by the lock protocol or the lock offset, the results in Figure 7.4b are nearly identical to Figure 7.3b.

True lock contention is very rare in scientific applications. As previously mentioned, most lock contention arises from false sharing in other lock systems. Taking this into account, we wanted to test true lock contention to understand how it affected the various lock protocols. In Figures 7.5 and 7.6, we kept the locks offset by 64 bytes and 4096 bytes, respectively, and used 32 clients and 32 servers. The locks began with no contention (lined up one after the other) and then overlap each other at 25%, 50%, 75%, and 100% (full overlap) intervals. Each of the graphs has a different scale for clarity. Datatype-alt-try outperforms the other methods with up to 2,062,079 locks / sec in Figure 7.6a. Using the alt-try lock protocol

enables close to full locking bandwidth at all levels of contention. Datatype-one-try and datatype-two-phase significantly drop in lock bandwidth as lock contention reaches 100%, since they must contact the servers multiple times using the slower two-phase protocol. The difference is more pronounced in Figure 7.6a since the two-phase lock protocol has increased the number of server rounds due to the larger lock offset. List locking is fairly effective in both list-alt-try and list-one-try. List-two-phase does not fair as well, at about a constant $\frac{1}{3}$ drop in performance. POSIX-based lock methods react poorly to increasing lock contention in both the 64 byte and 4096 byte offset cases.

In summary, these lock tests demonstrate how our hybrid lock protocols with list locking and datatype locking provide a large performance increase over a naive POSIX-two-phase method in both overlapping and nonoverlapping cases. In particular, the alt-try lock protocol with the datatype lock method improves performance by an order to two orders of magnitude over the naive locking method.

7.5.2. S3D I/O Benchmark

S3D is a parallel direct numerical simulation (DNS) solver designed at Sandia National Laboratories [**79**]. S3D solves the full compressible Navier-Stokes, total energy, species and mass continuity equations coupled with detailed chemistry and is based on a high-order accurate, non-dissipative numerical scheme. S3D writes checkpoint files at periodic intervals which also are used for post-processing in the analysis phase. The three-dimensional Cartesian mesh points of solved variables constitute most of the checkpoint data, which is also in a three-dimensional array. A majority of the checkpoint data is useful during the analysis phase. Since data analysis is an iterative process, the checkpoints are likely to be revisited periodically. Each aggregate checkpoint stores four global arrays, which represent

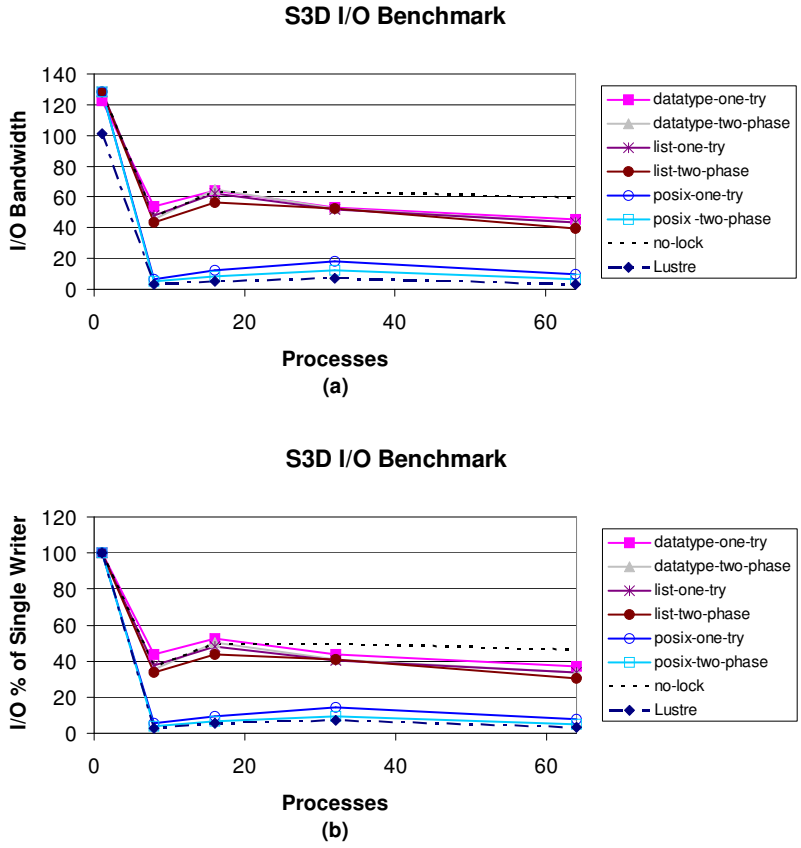

Figure 7.7. (a) Raw I/O bandwidth. (b) I/O bandwidth as a fraction of single writer I/O time.

mass, velocity, pressure, and temperature, respectively. The mass and velocity arrays are four-dimensional and the pressure and temperature arrays are three-dimensional. All four arrays have the same dimensionality for the lowest three spatial dimensions X, Y, and Z. The XYZ dimensions of all arrays are partitioned in the same block-block-block fashion among the MPI processes. The fourth dimension sizes of the mass and velocity data are 11 and 3, respectively, and are not partitioned. The S3D benchmark only performs the checkpoint writes of the S3D code. When a single process is used for checkpointing, its writes are contiguous. As we increase the number of processes for checkpointing, the aggregate data

size of approximately 1.19 GBytes remains constant, which makes the access pattern more noncontiguous and reduces the size of each individual write. Smaller writes are a challenging problem for file systems since hard drives prefer large I/O sizes.

The original S3D application uses Fortran I/O, where each process writes its own sub-arrays to an individual file during each checkpoint. While this is typically very fast due to large contiguous I/O calls to non-shared files, it creates a file management problem with an increased number of processes. Additional problems include the requirements that post-processing techniques must access all the individual files and restarts must use the same number of processes. We added an I/O implementation using the MPI-IO API to write the arrays to a shared file in their canonical order. With this change, there is only one file created per checkpoint, no matter how many MPI processes are used, reducing the data management problem. In this test, we try all our lock methods as well as a no-lock method for understanding lock overhead costs. We also compare our single writer normalized results against the Lustre file system to look at false sharing costs (these writes are not aligned to the file system block size). Since the writes are not overlapping, we represent the one-try and alt-try lock protocols with one-try in these tests.

In Figure 7.7a, we show the overall I/O bandwidth with varying numbers of processes. As expected, as the number of processes increases, overall I/O bandwidth decreases for all methods. Lustre performance begins well with one client since the writes are contiguous and not shared. Then, since the writes are not aligned to the block size, caching and locking overheads reduce performance significantly as the number of processes increases. The POSIX locking method fairs poorly as well due to the large number of lock requests to the server. The list and datatype locking methods fair better with their reduced locking overheads.

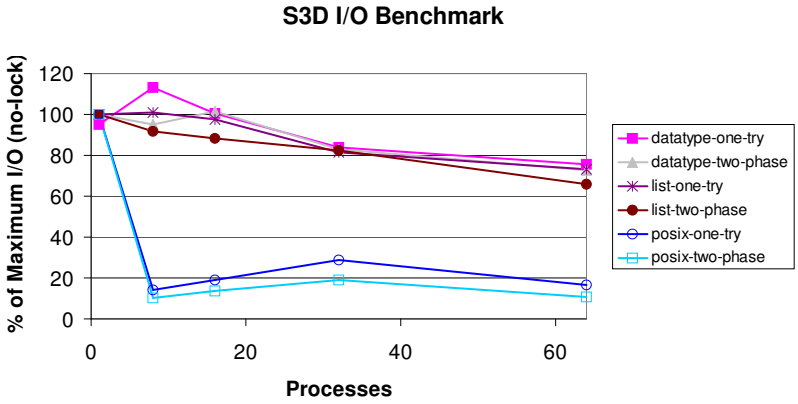

Figure 7.8. % of I/O bandwidth compared with no locking.

In order to make a trend comparison of the Lustre block-based caching and locking methods to our byte-ranged based locking, we normalized the I/O bandwidth as a % of the single write performance in Figure 7.7b. Most of the lock methods increase slightly from 8 to 16 processes, but fall slightly after that due to the noncontiguous file regions getting smaller and less efficient for the file system. The optimistic locking round in the one-try lock protocol makes a noticeable performance improvement over the two-phase lock protocol. The performance trend for Lustre is a large drop due to its false sharing and associated overheads.

Our final chart in Figure 7.8 examines the lock overhead of our DLM compared to the I/O bandwidth. POSIX-one-try can, at best, achieve approximately 35% of the no-lock bandwidth due to a large number of lock requests. From 8 to 64 processes, datatype-one-try maintains between 76% to 100% of the maximum I/O bandwidth. List-one-try also keeps I/O bandwidth between 73% to 100% from 8 to 64 processes.

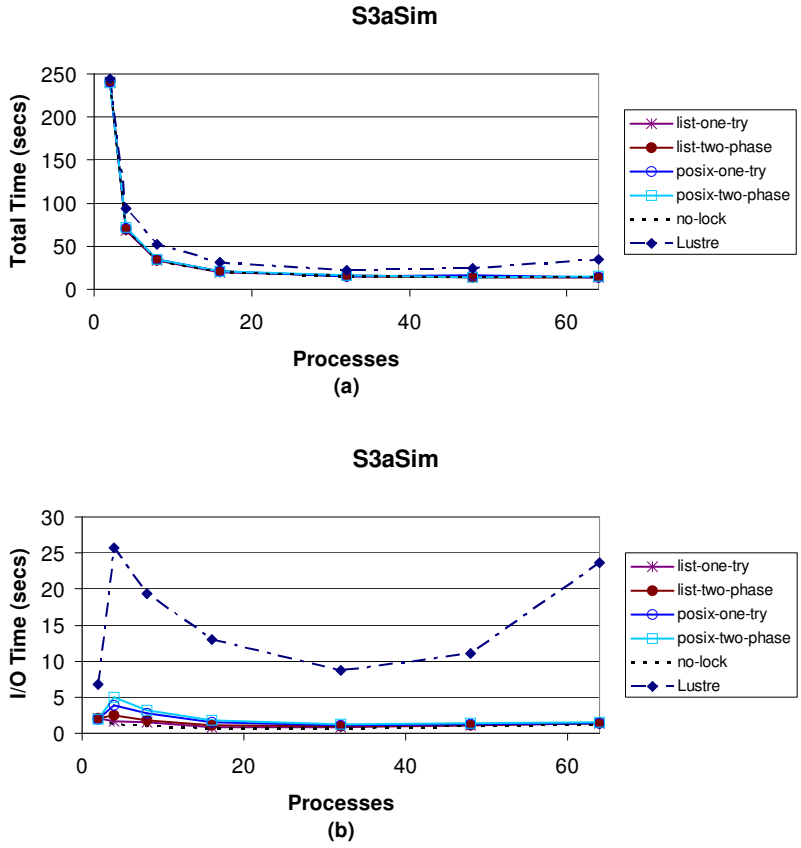

Figure 7.9. (a) S3aSim total execution time from 2 - 64 processes. (b) S3aSim I/O time from 2 - 64 processes.

7.5.3. S3aSim

We also used the S3aSim benchmark described in Chapter 6. Our tests were configured based on the NT database characteristics from NCBI [**60**] with a minimum sequence length of 6 bytes, a maximum sequence length of slightly over 43 MBytes, and a mean sequence length of 4401 bytes. We simulated the search of 20 input queries against 128 database fragments, all with the NT database characteristics. Anywhere between 1000 to 2000 results were pseudo-randomly generated per query and written to file (after the entire query had been completely searched) with MPI_File_write() and then forced to disk with MPI_File_sync().

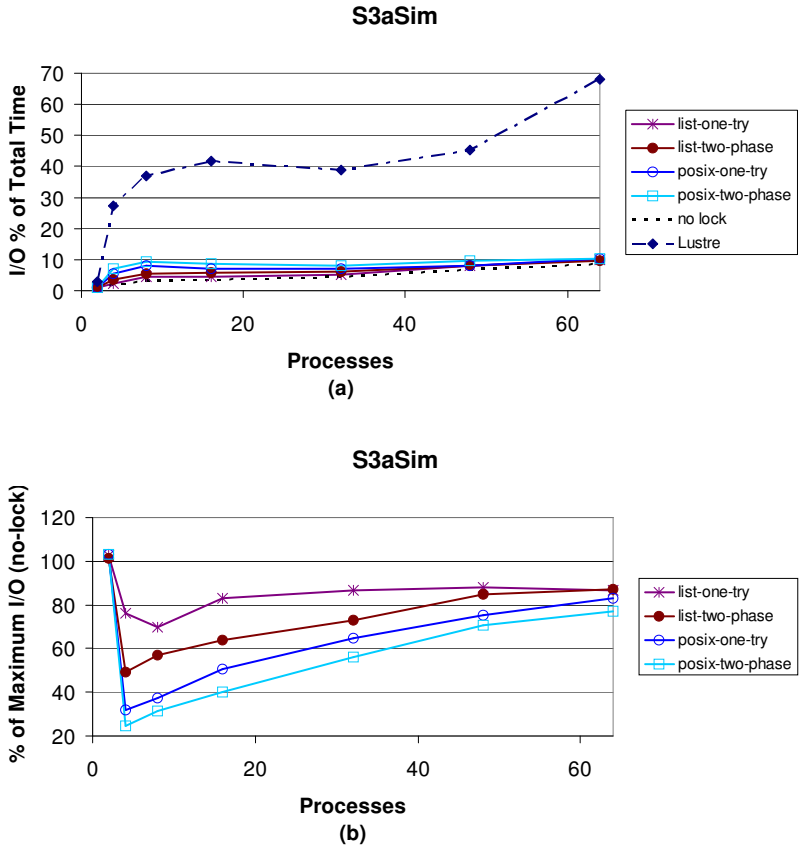

Figure 7.10. (a) S3aSim I/O time as a % of total execution time from 2 - 64 processes. (b) S3aSim % of I/O bandwidth compared to no locking from 2 - 64 processes.

The results generated are consistent and are not dependent on the number of worker nodes. Each test run produced an aggregate 208 MBytes of output data. We used between 2 - 64 processes in our tests. With only 2 processes, there is a master process and a single worker process, therefore, the worker is writing contiguous data to file. When there are more worker processes writing, the data is noncontiguous and unstructured with varying result sizes and counts. Since the data is unstructured, we do not use the datatype locking method as it

breaks down to list locking. Additionally, since the data access is nonoverlapping, the one-try lock protocol and alt-try lock protocol have identical performance, hence we represent both lock protocols with the one-try lock protocol.

In Figure 7.9a, we look at the scalability of total execution time. The total execution time falls as the number of processes is increased due to the embarrassingly parallel computation. As the number of processes reaches 32, however, the curve flattens out and Lustre starts to increase due to its increasing I/O burden. The Lustre block-based caching and locking has increased false sharing as more processes share file access. The isolated I/O times are shown in Figure 7.9b.

In Figure 7.10a, we examine I/O time as a percentage of total execution time. Lustre begins at about 3% when a single worker process is writing to the file. However, as the number of processes increases, the Lustre I/O % increases sharply up to 4 processes and rises up to 68% at 64 processes. The lock methods implemented in our DLM all stay below 11%, rising much slower than Lustre.

In Figure 7.10b, we check the locking overhead of our atomic operations with respect to I/O times. When there is only a single worker, the lock overhead is practically negligible since it is a single contiguous lock and a large amount of I/O. However, as the number of processes increases, the amount of I/O per worker decreases rapidly and becomes more noncontiguous. For instance, results that may have been contiguous on one worker are now split into two. Therefore, the overall I/O time increases due to smaller I/O requests, which causes the locking overhead to be a smaller % of overall I/O time. In the best case (excluding one worker), list-one-try stays between 70 % to 87% of peak I/O performance, a reasonable overhead for atomicity.

7.6. Summary

In this chapter, we have presented a novel DLM approach with true byte-range locking using hybrid lock protocols in combination with highly descriptive lock methods to improve atomic noncontiguous I/O performance. We have shown that this fusion of techniques can improve locking throughput up to between one to two orders of magnitude in performance and maintain a low overhead in achieving atomicity for noncontiguous I/O operations. Additionally, we have shown the benefits of eliminating false sharing with our byte-range granular approach in a comparison with a block-based locking system. Our application benchmarks showed that the list or datatype lock methods in conjunction with hybrid lock protocols would not seriously degrade I/O performance.

There are many areas where we would like to further explore this work. First of all, an open problem is how to handle failure on clients or servers. While timeout solutions have been proposed to handle client failures, server failures remain an issue. We are investigating the persistent storage of locks as a possible method for resolving this problem. Another area for study is how to best use these lock techniques for supporting the atomic mode for nonblocking, noncontiguous I/O operations efficiently. At present, we are not aware of any support for an atomic mode for nonblocking, noncontiguous I/O operations even though they are part of the MPI-IO specification.

CHAPTER 8

The Versioning Parallel File System

As previously mentioned, the growing use of parallel file systems to sustain scalable I/O performance for scientific applications has led to emerging performance problems in fault-tolerance, strict consistency semantics, and noncontiguous I/O access for large-scale computing. In this chapter, we discuss a new parallel file system architecture for large-scale scientific applications to address the above issues. In Section 8.1, we begin by discussing the challenges of future parallel file systems and how atomic noncontiguous I/O plays a large role in handling these problems. In Section 8.2, we describe atomicity implementation difficulties in the file system. In Section 8.3, we describe a protocol for implementing "versioning" in a parallel file system. In Section 8.4 and 8.5 we detail our prototype implementation and experimental results, respectively. Finally in Section 8.6, we explain the advantages and disadvantages of VPFS when compared to traditional methods.

8.1. Fault-Tolerance, Strict Consistency, and Noncontiguous I/O

Future high-performance storage platforms must address the trends in scientific computing. First as discussed previously in Chapter 2, scientific datasets are rapidly growing in size. In particular, noncontiguous I/O methods must scale to larger data access sizes. System snapshots generated for visualizing and/or checkpointing are I/O intensive and become more challenging as data resolution increases. In order to handle large I/O access patterns, storage systems must also scale up in size for both capacity and performance reasons. While

increasing the parallelism of storage systems can provide additional performance capabilities, it also makes the overall storage system more prone to component failure. RAID parity techniques such as those described in [**71, 13, 25, 84**] are often used to provide varying degrees of fault-tolerance and will help storage systems to scale reliably.

RAID parity techniques use both data blocks and parity blocks, where parity blocks contain some redundant information for reconstruction of a lost data block. Data blocks and parity blocks must be consistent with each other to be useful. Atomic I/O operations that update both data blocks and parity blocks simultaneously are required to keep data and parity consistent with each other. An atomic I/O operation is defined as either an I/O operation which fully completes or does not complete at all. In other words, a read operation will never see the effects of a partially completed write. Both atomic contiguous I/O operations and atomic noncontiguous I/O operations must be supported by the file system to keep parity data consistent.

Even if the file system is not using atomicity for parity based fault-tolerance, programmers may require the use of strict atomicity semantics. For example, the MPI-IO API has an atomic mode. If the file system does not provide any atomicity guarantees, MPI-IO must provide atomicity through an external method. Programmers may use a producer-consumer model to visualize or post-process data as the application is running. For example, one application will produce checkpoint or post-processing snapshots and another may post-process and visualize this data in real-time to provide immediate feedback for scientists during an application run.

In summary, file systems that intend to provide fault-tolerant large-scale I/O efficiently for scientific computing must provide atomic high-performance I/O methods.

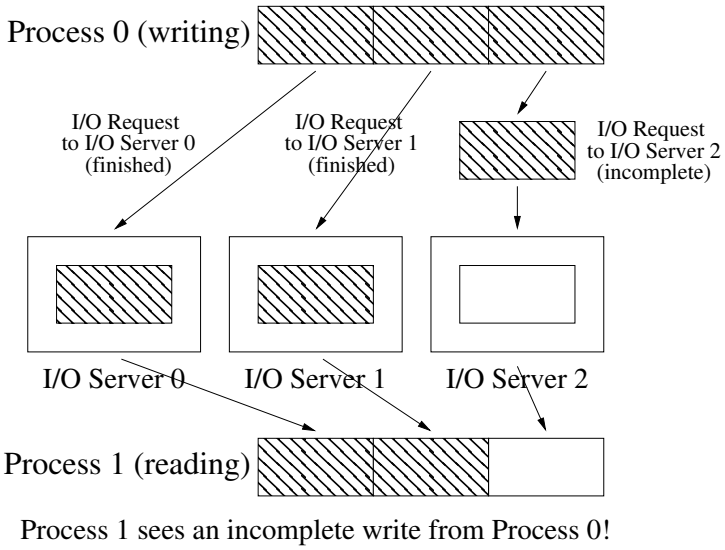

Figure 8.1. Atomicity challenges for parallel file systems can occur even with contiguous I/O operations.

8.2. Atomicity For Parallel I/O

Ensuring atomicity for I/O operations is difficult for several reasons. First, noncontiguous I/O operations may be broken up by higher level libraries into multiple POSIX I/O operations as we discussed in Chapter 2. Secondly, since contiguous I/O operations are divided into multiple I/O operations across multiple I/O servers as shown in Figure 8.1, interleaved read and write operations that are logically contiguous can produce non-atomic results. We begin our discussion on implementing atomicity through the traditional methods.

In many file systems, weak semantics, like those used in AFS [**81**] and Gfarm [**91**] (updates are only visible on close), have been used for performance reasons. A traditional hardware RAID controller serializes I/O requests in order to ensure that they are not interleaved. Such a serialized solution is not practical for cluster computing as it would greatly degrade performance when scaling to hundreds or thousands of devices. In the past, locking has been the popular solution for ensuring atomicity in parallel file systems.

A typical lock-based synchronization solution forces processes to acquire either read or write locks on a file region before an I/O operation. Read locks are shared, which means that multiple processes may be simultaneously granted read locks as long as no write lock regions intersect the read locked region. Write locks are exclusive. Only a single process may have a lock on a file region if it is a write lock. Locks provide synchronization capabilities to implement atomic I/O since I/O operations are automatically serialized when an I/O access pattern overlap occurs.

While a true byte range lock-based synchronization solution can allow concurrent I/O for non-overlapped I/O as demonstrated in Chapter 7, overlapping I/O will always be at least partially serialized if any of the overlapping I/O operations is a write. Another problem with locking is that locks that are held by clients who fail must eventually be reclaimed. However, while the lock is held by a dead client, access to the locked regions is limited or, in many cases, impossible. Some lock solutions may require alignment boundaries, which leads to false sharing. For instance, the GPFS lock token granularity can be no smaller than one system block [**82**]. Finally lock-based synchronization typically can only provide atomic I/O access when a client does not fail. If a client fails during the middle of a write and the lock for that region is recalled, another client may later see the effects of a partial write, which violates atomicity. In order to provide atomicity guarantees while allowing for client failure, I/O writes must use data journaling or related techniques to undo the effects of the partially completed write.

8.3. VPFS Protocol

In order to efficiently provide high-performance atomic noncontiguous I/O access for scientific computing, we introduce the *Versioning Parallel File System*, or VPFS. As described

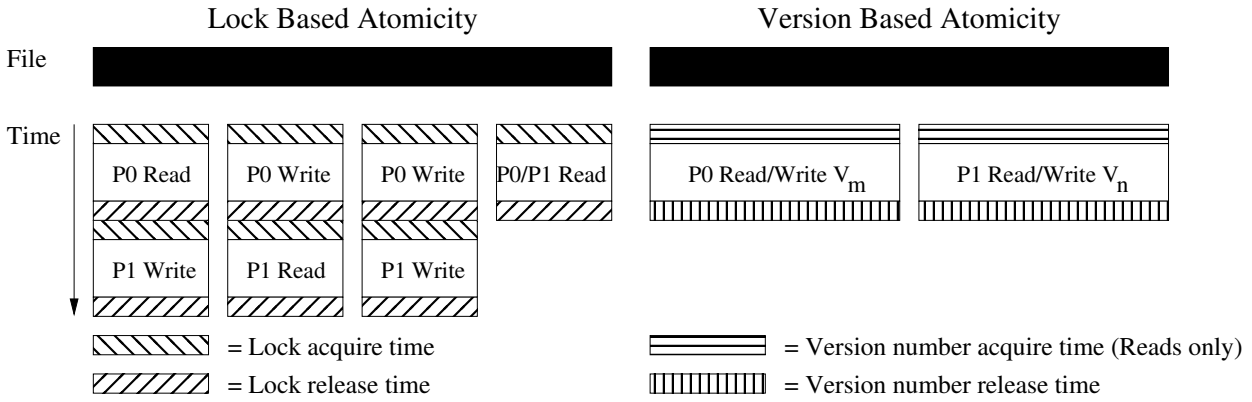

Figure 8.2. Serialization using locking versus concurrent I/O using versioning.

in Section 8.2, locking techniques serialize I/O access in many cases and also create significant overhead. Instead of locking, VPFS uses a technique called *versioning* to handle performance issues for atomic noncontiguous I/O access.

Versioning in the file system is not a new technique. It has been used in various projects [**80, 58, 86, 26**] to allow the file system to see multiple states of a file. The most common use of versioning in file systems is to provide users with the ability to recover deleted files as well as collaborate on large projects. Additionally, several projects have focused on using versioning for implementing security features and audit trails [**87, 73**]. We have applied the versioning technique toward implementing I/O operations in parallel file systems. This combination has tremendous potential beyond simply undoing and merging file changes. Every I/O operation which modifies data in the file system will create a "version", which is a tuple of information consisting of {*operation type, datatype description, and version number*}. The operation type is the operation that this version represents (for example write, truncate, checkpoint, etc). The datatype description is the access pattern of the modify operation. Each version tuple on an I/O server is labeled with a modify number *Vm*, which specifies a global order of completion.

VPFS has three major components: clients, version servers, and I/O servers. Clients are processes accessing files in VPFS. Version servers keep track of version information per file. For simplicity, file metadata is also stored on the version server (although a metadata server could also exist as a stand alone server for performance reasons). I/O servers are responsible for storing actual file data and version tuples. The data is distributed among them in a method described by the metadata on the version servers (for example, striped in a round robin manner). We describe the duties of each of the three major components in detail in the following paragraphs.

Clients, or C, are required to obtain an $ID\#$ before performing any I/O. When a client is about to perform I/O, it increments a counter $IO\#$ associated with the $ID\#$ it obtained. $ID\#$ and $IO\#$ are concatenated to create a unique temporary version number T that can be used for a data modify operation. This temporary version number T will be used as a temporary identification number for modify operations that are still in the process of being completed. They will later be assigned a modify version number Vm, which can be used to determine the global ordering of the file operations. Data retrieve operations first contact the version server to obtain snapshot information of the file associated with a retrieve operation Vr, which includes a list of the other retrieve operations in progress, tuples to map from temporary version numbers to final modify version numbers, and the smallest Vr in progress (Vr_min).

A version server, or Vs, keeps track of a file's current version number (V). The entire group of version servers are VS. V is used to label I/O operations in the order in which they complete for modify operations (for example, write) and the order in which they began for retrieve operations (for example, read). The version servers keep track of the Vr numbers used for retrieve operations that have not yet completed in a list called Vr_list. The minimum

of *Vr_list* is *Vr_min*, the oldest retrieve operation still in progress. Version servers also give out *ID#* to clients when requested, incrementing *ID#* on each request to give each client a unique *ID#*. Modify operations are given file version numbers when they have fully completed. These mappings from temporary version numbers T created on the clients to file version numbers *Vm* are kept in *Vmap*. The *Vs* keeps track of *Vmap* as a list of tuples {*T, Vm, B*}. B is the bit array that keeps track of every I/O server involved in the operation. Versioning information is maintained for each file on *VS*.

An I/O server, or *Is*, maintains the version tuples and a list of the versions in use (*Vuse_list*). The entire group of I/O servers is called *IS*. They also execute version merging, which is a process where version tuples that are committed and are not waiting on any prior read operations are read and merged into the final data version to increase free disk space and reduce version complexity.

We describe the basic VPFS processes of open, merge, read and write. The sync and close operations do not require any special consideration. At this time, we begin a "merge" when either sync or close is called. Further details are included in our implementation and the performance results below.

- Open - *C* requests an *ID#* from the proper *Vs* for file *F*. The *Vs* increments the *ID#* for the file *F* and returns the previous value of *ID#* to *C*. We note that the client's acquisition of *ID#* can happen at any time before a retrieve or modify I/O operation. For simplicity, we define this process as part of the file open.
- Merge - *C* requests to begin a merge operation from the proper *Vs* for file *F*. *Vs* figures out *Bor* (OR all *B* for all *Vmap* tuples up to *Vr_min*) and returns *Vr_min*, *Bor*, and the relevant *Vmap* up to *Vr_min* to *C*. *C* uses *Bor* to figure out the necessary *Is* involved in the merge and sends *Vmap* to each of them. The *Is*

updates all temporary versions in *Vmap* to their final version number and adds them to *Vuse_list*.

- Write - *C* increments *IO#* and concatenates its *ID#* with the old *IO#* to form *T*. *C* sends *T* along with its write data to each relevant *Is*. The *Is* saves the version tuple with its temporary version number *T*. When *C* has completed writing all of its version tuples, it sends *T* and *B* to the proper *Vs*. The *Vs* increments *V* and uses the old *V* (as *Vm*) to form a tuple {*T, Vm, B*} that is added to *Vmap*.
- Read - *C* requests to begin a read operation from the proper *Vs* for file *F*. *Vs* increments *V* for *C* and figures out *Bor* (OR all *B* for all *Vmap* tuples up to *Vr*) and returns *V* (as *Vr*), *Vr_min*, *Bor*, and *Vmap* to *C*. *Vs* adds the returned *Vr* to the *Vr_list*. *C* uses *Bor* to figure out the necessary *Is* involved in the read and sends *Vr*, *Vr_min*, and *Vmap* to each of them. The *Is* updates all temporary versions in *Vmap* (up to *Vr_min*) to their final version number and adds them to *Vuse_list*. The *Is* uses all the versions up to *Vr* to satisfy the read request. Then the *Is* returns the requested read data to *C*. When it completes, *C* sends back *Vr*, *Vr_min*, and *Bor* to *Vs*. *Vs* removes the *Vr* from *Vr_list* and removes *Vmap* entries that are less than *Vr_min* and match *Bor*.

8.4. VPFS Implementation

VPFS was implemented using PVFS2 as a base parallel file system. While it would be ideal to build a completely new file system based around the concepts of versioning, we decided to evaluate the concepts in an established parallel file system to identify where a ground-up implementation could make significant improvements in both performance and code simplicity. Unfortunately, adding versioning to PVFS2 required us to modify many of

the client and server system call implementations. On-disk data storage had to be changed and many data structures were added in both the client and server. Additionally, ROMIO, our MPI-IO implementation, required changes to support PVFS2-based versioning.

The system call client code in PVFS2 is implemented using state machines. We added a new nested version state machine and version system call for handling all *VS* communication and operations. It supports getting *ID#* for one or more clients with a *count* parameter. The nested version state machine can get a snapshot, remove a snapshot, and commit a modify operation. This functionality is implemented using a nested state machine so that any client system call can add versioning support. A version-client-table on each client maps every metadata handle to a {*ID#, IO#*} tuple so that temporary version numbers can be generated. One difficulty was the changing the handle interface in PVFS2. All files are mapped to one or more handles that store metadata and data. We added additional handles to store version-metadata and version-data on the server. System calls that involved create, remove, I/O, and other relevant operations had to be manipulated accordingly.

The PVFS2 servers required a corresponding version state machine as well to support *Vs* operations. A version-metadata table keeps version numbers and version tuples in memory. Any versioning changes are recorded to a log file prior to the server response so that they are not lost during a server crash. The I/O state machine on the servers was modified to support version-data log files. These log files store a tuple of version information (modify operation, offset to next tuple, datatype size, data size, datatype, and data) which is either incomplete or committed but cannot be merged due to incomplete earlier reads. They will wrap around at the desired maximum log size or can be compressed. The tuple of version information is applied to the actual data file when snapshot information is received either from the client or a version server. Reads are handled by unfolding the datatypes of the

committed operations that cannot be merged into a offset-length tree. To find the necessary read data, the server first looks for matching offset-length pairs in the tree and then figures out if it needs to look in the log file or the data file for its requested data.

In order to use our VPFS implementation with ROMIO, we added the version system call to MPI_File_open() to get as many *ID#* as the number of processes in the MPI communication group which opened the file. Each process receives its *ID#* from the root process through a MPI_Bcast() call. A user-specified hint decides at open time whether the file will be versioned. One a file is created, it cannot change its versioned attribute.

8.5. Performance Evaluation

We have acquired some preliminary write results using our VPFS implementation. All tests were conducted on the Blackrose cluster at Sandia National Laboratories as the Feynman cluster has been retired. Blackrose has 78 HP XW9400 nodes in operation. Each node has 2 dual-core 2.8 GHz AMD Opteron CPUs with 8 GBytes of RAM and runs the Red Hat Linux Enterprise operating system. They nodes are connected with both Gigabit Ethernet and Infiniband, however, we were only able to use the Gigabit Ethernet interface with MPICH2.

Our preliminary results used 8 computers for our PVFS2 file system with a default 64 KByte strip size, totaling to a 512 KByte stripe across all I/O servers. Our initial tests used the HPIO benchmark, described in Chapter 5. We tested a datatype I/O-only method versus both our atomic methods (lock server and versioning techniques). Figures 8.3 and 8.4 show our experimental results when varying the region count and region spacing, respectively.

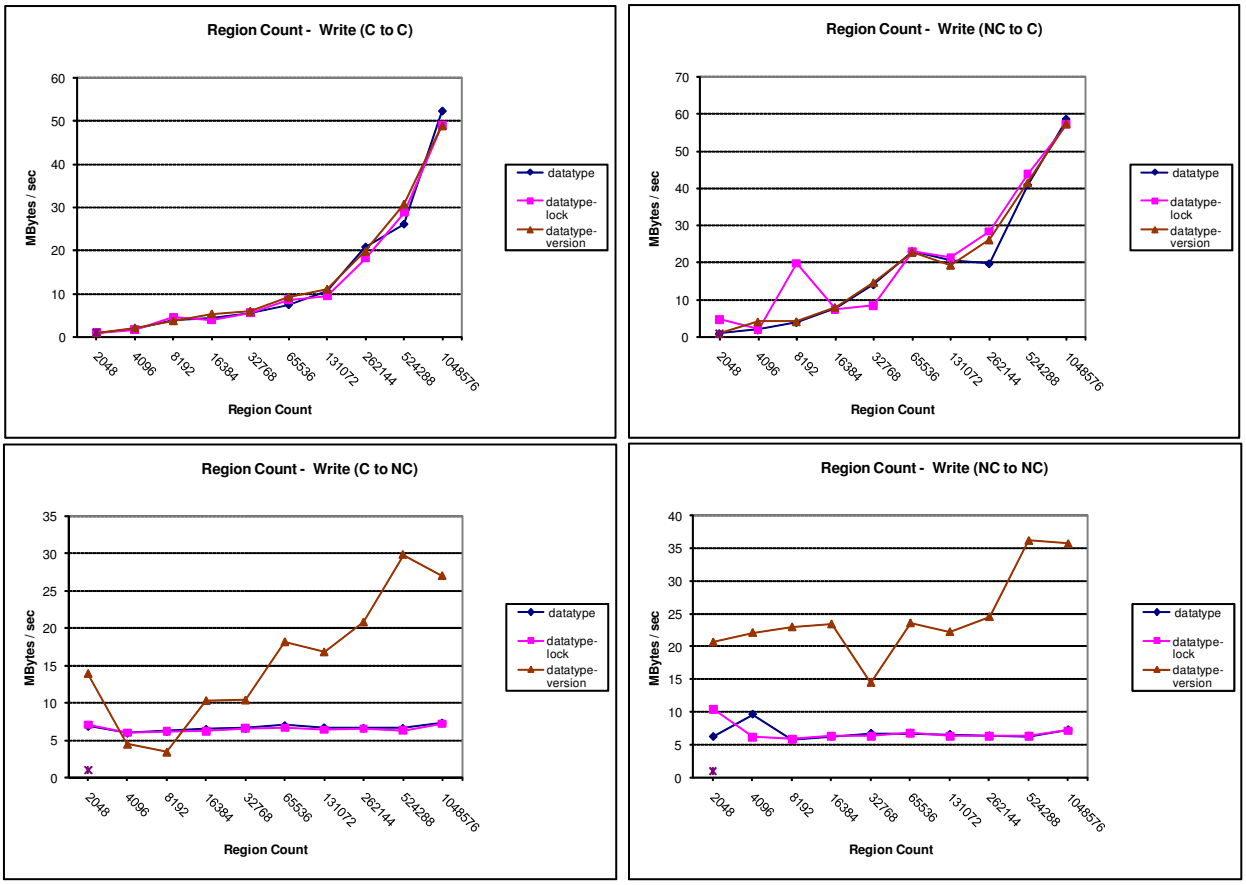

Figure 8.3. HPIO results from testing various region counts and different atomicity methods.

8.5.1. HPIO - Vary Region Count Results

In the c-c case, all methods scale up in I/O bandwidth due to the increasing write sizes that are efficient for today's hard disk technology. We note that the locking overhead is quite small for our atomic methods since they cannot falsely share locks. In the nc-c case, the results are very similar to the c-c case due to the efficiency of the datatype representation for I/O as well as atomicity for datatype-lock. In the c-nc case, both the datatype and datatype-lock methods fall dramatically in performance due to the slow I/O performance of writing small and sparse 8 byte regions. While the datatype-version results are quite

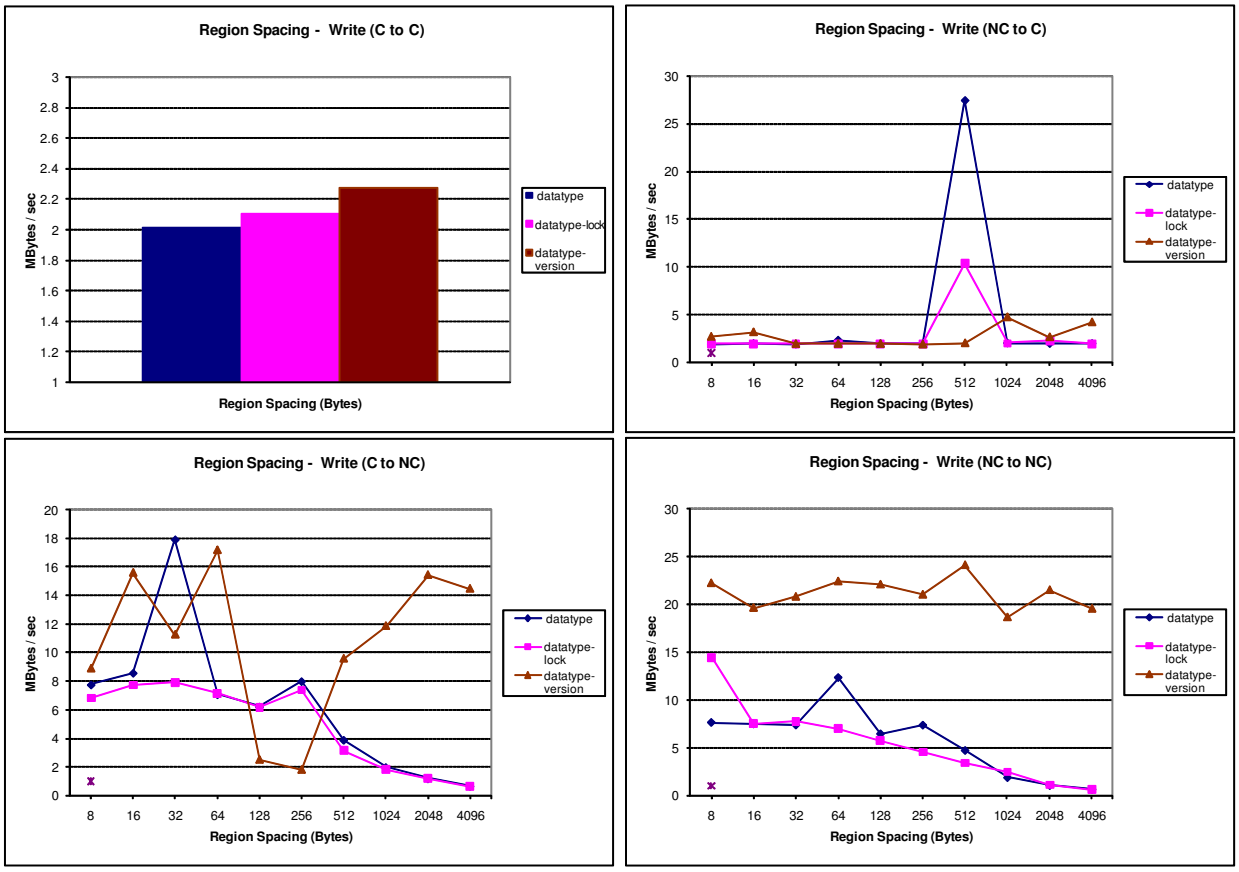

Figure 8.4. HPIO results from testing various region spacings and different atomicity methods.

sporadic due to the volatility of a small aggregate I/O size, it averages more than triple the performance of the other methods due to combining noncontiguous writes into contiguous writes. When moving from c-nc to nc-nc, the performance numbers are roughly the same, again showing the large advantage of datatype-version in reducing disk seeks.

8.5.2. HPIO - Vary Region Spacing Results

Our results in this test are very volatile, most likely due to the extremely small data size. Each process only write 32 KBytes of data, which is 512 KBytes of aggregate data. The region spacing test is designed to show the declining performance efficiency of hard disks

when presented with larger and large gaps between I/O accesses. c-c and nc-c results are very similar for all methods, demonstrating the low overhead of the atomic methods. c-nc performance trails off for both datatype and datatype-lock due to more disk seeking. The locking overhead for datatype-version remains fairly low throughout this test. datatype-version averages more than twice the performance of the other methods since it is not affected by the region spacing. The nc-nc case shows a more consistent picture of this pattern, where datatype-version is not affected by changing the region spacing.

8.6. VPFS Discussion

Using such a protocol will enforce the atomicity of I/O operations. Since write operations are not given a *Vm* until they have fully completed, no read can possibly see a partially completed write. Writes become visible in the order in which they complete due to the *Vm* assignment only after all parts of the (possibly noncontiguous) write have finished. If a client dies during the middle of a write operation, the write will never be assigned a final *Vm* and therefore will never be visible to any client. Cleanup operations issued by system administrators or automatically by the file system can remove these partially completed operations at a convenient time.

Another strong advantage of the use of versions over locks is I/O operation concurrency. All reads and writes may continue in parallel even when they access overlapping regions. As we discussed in Section 8.2, lock-based systems can only allow concurrent access to overlapping regions if both processes are reading. VPFS allows atomic concurrent access to overlapping regions for any combination of reads and writes without any serializing penalties as shown in Figure 8.2. Also discussed earlier, efficient atomicity of I/O operations is a strong requirement for parity based RAID techniques in parallel file systems. For RAID 5, parity is

maintained on a per stripe basis to ensure that any one I/O node may be lost while preserving all data and operating in a degraded mode until the failed node is replaced. In hardware RAID, parity is computed on every write operation by the hardware RAID controller. This method is not scalable or practical for parallel file systems. Since our versioning parallel file system can provide efficient atomic I/O operations, we can also perform atomic data and parity updates in parallel, even if writes are overlapping the same parity block. This is a large improvement over lock-based methods, since they always force serialization when writes have overlapping I/O parity access.

The nature of our datatype-based versions provides us with a solution to the noncontiguous I/O problem of small I/O operations. Since we can provide a description of the object along with file write data, we can describe the format of the data as well as write out the actual file data in a contiguous I/O operation. In this manner we no longer incur the performance penalty of writing small noncontiguous file regions to disk for noncontiguous access patterns. This performance optimization should greatly improve noncontiguous I/O operations over the current datatype I/O method. We still can use the datatype I/O access pattern description over the network for bandwidth savings, but additionally reap performance benefits from writing noncontiguous data contiguously in file. Another important performance optimization for noncontiguous I/O would be that we have a constant versioning overhead for any noncontiguous I/O operation. In comparison, list lock and datatype lock require n locks to service a noncontiguous I/O access pattern with n file regions. In other words, our versioning techniques in VPFS allow us to eliminate the much of the overhead associated with locking all the regions of the file access pattern while providing even greater I/O concurrency.

System snapshots are often dumped at intervals between scientific computational stages. This is done so that if a part of the system fails, the application can be restarted without losing all progress. System snapshots are also used for post-processing and/or visualization. Snapshots are I/O intensive and slow on traditional parallel file systems. In fact, the snapshot frequency is often determined by the time it takes to write a snapshot. If snapshots are cheap and fast, they can be written often without significantly affecting system performance. Our versioning of files natively creates such snapshots. In fact, by simply tagging the version of the last write that we would like to include in the system snapshot, we can ensure that system snapshots remain in our I/O system without significant I/O penalties. Such an optimization for system snapshots would provide a great tool for scientific computing (for example, reducing the cost of making a checkpoint in the ASC FLASH code). We also note that within the MPI-IO interface, we can perform several optimizations with VPFS. We would like to experiment on how to use versioning and calculate write parity on collective I/O operations. We expect that there are important optimizations when using a single version versus using multiple versions when collective I/O is scaled up to thousands of processes.

Some obvious drawbacks for VPFS include possible new read overheads. Since a read operation must examine version tuples on each I/O server involved, we expect that overall read performance will slight worsen, while write speeds will increase. Also, since we make version tuples for every modify operation, this can, in certain cases, significantly increase overall storage. We expect to further examine these potential problems as we continue to develop our implementation.

CHAPTER 9

Conclusion and Future Work

As computational resources continue to outpace storage devices, high-performance I/O will be an even greater challenge in the future. While parallel file systems alleviate some of the performance gap, many scientific applications use noncontiguous I/O access patterns that reduce the effectiveness of I/O parallelization. We have presented list I/O and datatype I/O techniques which improve independent I/O through a significant reduction in the number of I/O requests to the file system. Additionally, we demonstrated how noncontiguous methods are affected by basic access pattern parameters and have investigated the applicability of these results toward a case study for examining various high-level I/O strategies in bioinformatics.

Our work has transitioned to efficient I/O atomicity for supporting various I/O APIs and enabling new application-level techniques such as real-time data visualization and consistent data layout for cooperating processes. Our DLM work has shown that atomicity approaches based on list lock and datatype lock coupled with hybrid optimistic lock protocols can be very effective for non-overlapping data access. Furthermore, we have prototyped a versioning parallel file system as a new architecture for solving next generation I/O challenges.

In the future, we intend to implement fault-tolerant storage using these atomic I/O building blocks we have designed in the parallel file system. In particular, we would like to begin our effort with implementing client-based RAID 0+1 and RAID 5 solutions using both our DLM and versioning approaches.

References

[1] Peter Aarestad, Avery Ching, George Thiruvathukal, and Alok Choudhary. Scalable approaches for supporting MPI-IO atomicity. In *Proceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid*, Singapore, May 2006. IEEE Computer Society Press.

[2] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 109–126. ACM Press, December 1995.

[3] Sandra Johnson Baylor and C. Eric Wu. Parallel I/O workload characteristics using Vesta. In *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 16–29, Santa Barbara, CA, April 1995. IEEE Computer Society Press.

[4] Dennis A. Benson, Ilene Karsch-Mizrachi, David J. Lipman, James Ostell, and David L. Wheeler. Genbank. *Nucleic Acids Res.*, 35:21–25, 2007.

[5] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[6] Robert Bjornson, Andrew Sherman, Stephen Weston, Nathan Willard, and James Wing. TurboBLAST(r): A parallel implementation of BLAST built on the TurboHub. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2002.

[7] Kevin J. Bowers, Edmond Chow, Huafeng Xu, Ron O. Dror, Michael P. Eastwood, Brent A. Gregersen, John L. Klepeis, Istvan Kolossvary, Mark A. Moraes, Federico D. Sacerdoti, John K. Salmon, Yibing Shan, and David E. Shaw. Molecular dynamics—scalable algorithms for molecular dynamics simulations on commodity clusters. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 84, New York, NY, USA, 2006. ACM Press.

[8] R. C. Braun, Kevin T. Pedretti, Thomas L. Casavant, Todd. E. Scheetz, Clayton L. Birkett, and Chad A. Roberts. Parallelization of local blast service on workstation clusters. *Future Gener. Comput. Syst.*, 17(6):745–754, 2001.

[9] Mike Burrows. Chubby distributed lock service. In *Proceedings of the 7^{th} Symposium on Operating System Design and Implementation, OSDI'06*, Seattle, WA, November 2006.

[10] Nick Camp, Haruna Cofer, and Roberto Gomperts. High-throughput BLAST. Whitepaper - http://www.sgi.com.

[11] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000. USENIX Association.

[12] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7^{th} Symposium on Operating System Design and Implementation, OSDI'06*, pages 205–218, Seattle, WA, November 2006.

[13] Peter M. Chen and David A. Patterson. Maximizing performance in a striped disk array. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 322–331, 1990.

[14] Ed Chi, Elizabeth Shoop, John Carlis, Ernest Retzel, and John Riedl. Efficiency of shared-memory multiprocessors for a genetic sequence similarity search algorithm. Technical Report TR97-005, University of Minnesota, Computer Science Department, 1997.

[15] Chiba City, the Argonne scalable cluster. `http://www.mcs.anl.gov/chiba/`.

[16] Avery Ching, Alok Choudhary, Kenin Coloma, Wei Keng Liao, Robert Ross, and William Gropp. Noncontiguous access through MPI-IO. In *Proceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid*, Tokyo, Japan, May 2003. IEEE Computer Society Press.

[17] Avery Ching, Alok Choudhary, Wei Keng Liao, Robert Ross, and William Gropp. Noncontiguous I/O through PVFS. In *Proceedings of the IEEE International Conference on Cluster Computing*, Chicago, IL, September 2002. IEEE Computer Society Press.

[18] Avery Ching, Alok Choudhary, Wei Keng Liao, Robert Ross, and William Gropp. Efficient structured access in parallel file systems. In *Proceedings of the IEEE International Conference on Cluster Computing*, Hong Kong, December 2003. IEEE Computer Society Press.

[19] Avery Ching, Alok Choudhary, Wei Keng Liao, Robert Ross, and William Gropp. Evaluating structured I/O methods for parallel file systems. In *International Journal of High Performance Computing and Networking*, volume 2, pages 133–145, 2004.

[20] Avery Ching, Alok Choudhary, Wei Keng Liao, Lee Ward, and Neil Pundit. Evaluating I/O characteristics and methods for storing structured scientific data. In *Proceedings of the International Parallel & Distributed Processing Symposium*, Rhodes Island, Greece, April 2006. IEEE Computer Society Press.

[21] Kenin Coloma, Avery Ching, Alok Choudhary, Wei Keng Liao, Rob Ross, Rajeev Thakur, and Lee Ward. A new flexible MPI collective I/O implementation. In *Proceedings of the IEEE Conference on Cluster Computing*, September 2006.

[22] Kenin Coloma, Alok Choudhary, Wei Keng Liao, Lee Ward, Eric Russell, and Neil Pundit. Scalable high-level caching for parallel I/O. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, Sante Fe, NM, April 2004. IEEE Computer Society Press.

[23] Kenin Coloma, Alok Choudhary, Wei Keng Liao, Lee Ward, and Sonja Tideman. DACche: Direct access cache system for parallel I/O. In *Proceedings of the International Supercomputer Conference*, Heidelberg, June 2005. Prometeus GmbH.

[24] Compugen, Ltd. Bioccerator. http://eta.embl-heidelberg.de:8000/, 1994.

[25] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-diagonal parity for double disk failure correction. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 1–14, Berkeley, CA, USA, 2004. USENIX Association.

[26] Brian Cornell, Peter A. Dinda, and Fabián E. Bustamante. Wayback: a user-level versioning file system for linux. In *ATEC'04: Proceedings of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference*, pages 27–27, Berkeley, CA, USA, 2004. USENIX Association.

[27] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[28] Aaron Darling, Lucas Carey, and Wu chun Feng. The design, implementation, and evaluation of mpiBLAST. In *Proceedings of the 4th International Conference on Linux Clusters: The HPC Revolution*, 2003.

[29] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *Proceedings of the IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, Newport Beach, CA, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[30] Blake G. Fitch, Aleksandr Rayshubskiy, Maria Eleftheriou, T. J. Christopher Ward, Mark Giampapa, Michael C. Pitman, and Robert S. Germain. Molecular dynamics—blue matter: approaching the limits of concurrency for classical molecular dynamics. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 87, New York, NY, USA, 2006. ACM Press.

[31] Bruce Fryxell, Kevin Olson, Paul Ricker, Frank Timmes, Michael Zingale, Donald Lamb, Peter MacNeice, Robert Rosner, and Henry Tufo. FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *Astrophysical Journal Suppliment*, 131:273, 2000.

[32] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP*, Lake George, NY, October 2003.

[33] Global file system. `http://www.redhat.com/software/rha/gfs/`.

[34] Jeffrey D. Grant, Roland L. Dunbrack, Frank J. Manion, and Michael F. Ochs. Beo-Blast: distributed BLAST and PSI-BLAST on a Beowulf cluster. *Bioinformatics*, 18:765–766, 2002.

[35] William Gropp, Ewing Lusk, and Debbie Swider. Improving the performance of MPI derived datatypes. In Anthony Skjellum, Purushotham V. Bangalore, and Yoginder S. Dandass, editors, *Proceedings of the Third MPI Developer's and User's Conference*, pages 25–30. MPI Software Technology Press, 1999.

[36] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 29–43, Ashville, NC, 1993. ACM Press.

[37] HDF5 home page. `http://hdf.ncsa.uiuc.edu/HDF5/`.

[38] IBRIX FusionFS. `http://www.ibrix.com/`.

[39] IEEE/ANSI Std. 1003.1. Portable operating system interface (POSIX)–part 1: System application program interface (API) [C language], 1996 edition.

[40] TimeLogic Inc. Decypher. http://www.timelogic.com, 1996.

[41] Florin Isaila and Walter Tichy. Clusterfile: A flexible physical layout parallel file system. In *Proceedings of the IEEE International Conference on Cluster Computing*, Newport Beach, CA, October 2001. IEEE Computer Society Press.

[42] W. James Kent. BLAT - the BLAST-like alignment tool. *Genome Research*, 12(4), 2002.

[43] Scott Alan Klasky, Stephane Ethier, Zhihong Lin, Kevin Martins, Doug McCune, and Ravi Samtaney. Grid-based parallel data streaming implemented for the gyrokinetic toroidal code. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 24, Phoenix, AZ, 2003. IEEE Computer Society.

[44] David Kotz. Disk-directed I/O for MIMD multiprocessors. *ACM Transactions on Computer Systems*, 15(1):41–74, February 1997.

[45] Robert Latham, William Gropp, Robert Ross, Rajeev Thakur, and Brian Toonen. Implementing MPI-IO atomic mode without file system support. In *Proceedings of the IEEE Conference on Cluster Computing Conference*, Boston, MA, September 2005. IEEE Computer Society Press.

[46] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, MA, October 1996.

[47] Jianwei Li, Wei Keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Sigel, Brad Gallagher, and Michael Zingale. Parallel netcdf: A high-performance scientific I/O interface. In *Proceedings of Supercomputing*, Phoenix, AZ, November 2003. ACM Press.

[48] Wei Keng Liao, Avery Ching, Kenin Coloma, Alok Choudhary, and Lee Ward. An implementation and evaluation of client-side file caching for MPI-IO. In *Proceedings of the International Parallel & Distributed Processing Symposium*, March 2007.

[49] Wei Keng Liao, Kenin Coloma, Alok Choudhary, Lee Ward, Eric Russell, and Sonja Tideman. Collective caching: Application-aware client-side file caching. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing*, Research Triangle Park, NC, July 2005. IEEE Computer Society Press.

[50] Walter B. Ligon III and Robert B. Ross. Implementation and performance of a parallel file system for high performance distributed applications. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 471–480. IEEE Computer Society Press, August 1996.

[51] Heshan Lin, Xiaosong Ma, Praveen Chandramohan, Al Geist, and Nagiza Samatova. Efficient data access for parallel blast. In *Proceedings of 19th International Parallel and Distributed Processing Symposium*, 2005.

[52] Lustre. `http://www.lustre.org`.

[53] Bin Ma, John Tromp, and Ming Li. PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.

[54] David R. Mathog. Parallel BLAST on split databases . *Bioinformatics*, 19:1865–1866, 2003.

[55] Message passing interface forum. `http://www.mpi-forum.org`.

[56] Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface, July 1997. `http://www.mpi-forum.org/docs/docs.html`.

[57] Jason A. Moore and Michael J. Quinn. Enhancing disk-directed I/O for fine-grained redistribution of file data. *Parallel Computing*, 23(4):477–499, June 1997.

[58] Kiran-Kumar Muniswamy-Reddy, Charles Wright, Andrew Himmer, and Erez Zadok. A versatile and user-oriented versioning file system. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, April 2004.

[59] David Nagle, Denis Serenyi, and Abbie Matthews. The Panasas ActiveScale storage cluster - delivering scalable high bandwidth storage. In *Proceedings of the 2004 ACM/IEEE Supercomputing Conference*, November 2004.

[60] NCBI. National center for biotechnology information. http://www.ncbi.nlm.nih.gov/.

[61] Nils Nieuwejaar and David Kotz. The Galley parallel file system. Technical Report PCS-TR96-286, Dept. of Computer Science, Dartmouth College, May 1996.

[62] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, October 1996.

[63] Zemin Ning, Anthony Cox, and James C. Mullikin. SSAHA: A Fast Search Method for Large DNA Databases. *Genome Res.*, 11(10):1725–1729, October 2001.

[64] Michael L. Norman, John Shalf, Stuart Levy, and Greg Daues. Diving deep: Data-management and visualization strategies for adaptive mesh refinement simulations. *Computing in Science and Engg.*, 1(4):36–47, 1999.

[65] Ron Oldfield and David Kotz. Armada: A parallel file system for computational grids. In *Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 194–201, Brisbane, Australia, May 2001.

[66] Ron A. Oldfield, Arthur B. Maccabe, Sarala Arunagiri, Todd Kordenbrock, Rolf Riesen, Lee Ward, and Patrick Widener. Lightweight i/o for scientific applications. In *Proceedings of the 2006 IEEE International Conference on Cluster Computing*, Barcelona, Spain, September 2006.

[67] Ron A. Oldfield, Patrick Widener, Arthur B. Maccabe, Lee Ward, and Todd Kordenbrock. Efficient data-movement for lightweight i/o. In *Workshop on high-performance I/O techniques and deployment of Very-Large Scale I/O Systems*, Barcelona, Spain, September 2006.

[68] The Open Group, http://www.unix.org/. *The Single UNIX Specification Version 3, 2004 Edition*, 2004.

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

[70] Paracel, Inc. Fast data finder (fdf) and genematcher. http://www.paracel.com.

[71] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 109–116, Chicago, IL, June 1988. ACM Press.

[72] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and Dave Hitz. NFS version 3: Design and implementation. In *USENIX Summer*, pages 137–152, 1994.

[73] Zachary N. J. Peterson, Randal Burns, Giuseppe Ateniese, and Stephen Bono. Design and implementation of verifiable audit trails for a versioning file system. In *FAST'07: Proceedings of the 5th conference on USENIX Conference on File and Storage Technologies*, pages 20–20, Berkeley, CA, USA, 2007. USENIX Association.

[74] James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. NAMD: biomolecular simulation on thousands of processors. In *SC '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, 2002.

[75] Jean-Pierre Prost, Richard Treumann, Richard Hedges, Bin Jia, and Alice Koniges. MPI-IO/GPFS, an Optimized Implementation of MPI-IO on top of GPFS. In *Proceedings of Supercomputing*, Denver, CO, November 2001. ACM Press.

[76] Russ Rew and Glenn Davis. The unidata netcdf: Software for scientific data access. In *Proceedings of the 6th International Conference on Interactive Information and Processing Systems for Meterology, Oceanography and Hydrology*, Anaheim, CA, February 1990. American Meteorology Society.

[77] ROMIO: A high-performance, portable MPI-IO implementation. `http://www.mcs.anl.gov/romio`.

[78] Robert Ross, Neill Miller, and William Gropp. Implementing fast and reusable datatype processing. In *Proceedings of the 10th EuroPVM/MPI Conference*, September 2003.

[79] Ramanan Sankaran, Evatt R Hawkes, Jacqueline H Chen, Tianfeng Lu, and Chung K Law. Direct numerical simulations of turbulent lean premixed combustion. *Journal of Physics: Conference Series*, 46:38–42, 2006.

[80] Douglas Santry, Michael Feeley, Norman Hutchinson, Alistair Veitch, Ross Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, December 1999.

[81] Mahadev Satyanarayanan. Integrating security in a large distributed system. *ACM Trans. Comput. Syst.*, 7(3):247–280, 1989.

[82] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the Conference on File and Storage Technologies*, Monterey, CA, January 2002.

[83] Ilya Sharapov. Computational applications for life sciences on sun platforms: Performance overview. Whitepaper, 2001.

[84] Muthian Sivathanu, Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving storage system availability with d-graid. *ACM Transactions on Storage*, 1(2):133–170, 2005.

[85] Evgenia Smirni, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. I/O requirements of scientific applications: An evolutionary view. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 49–59, Syracuse, NY, 1996. IEEE Computer Society Press.

[86] Craig Soules, Garth Goodson, John Strunk, and Gregory Ganger. Metadata efficiency in a comprehensive versioning file system. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, May 2002.

[87] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: protecting data in compromised system. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 12–12, Berkeley, CA, USA, 2000. USENIX Association.

[88] Sun Microsystems, http://www.faqs.org/rfcs/rfc1813.html. *RFC 1813 - NFS Version 3 Protocol Specification*, 1995.

[89] Sun Microsystems and Network Appliance, http://www.faqs.org/rfcs/rfc3530.html. *RFC 3530 - NFS Version 4 Protocol Specification*, 2003.

[90] Hong Tang, Aziz Gulbeden, Jingyu Zhou, William Strathearn, Tao Yang, and Lingkun Chu. A self-organizing storage cluster for parallel data-intensive applications. In *Proceedings of ACM Supercomputing Conference*, November 2004.

[91] Osamu Tatebe, Satoshi Sekiguchi, Youhei Morita, Noriyuki Soda, and Satoshi Matsuoka. Gfarm v2: A grid file system that supports high-performance distributed and parallel data computing. In *Proceedings of the 2004 Computing in High Energy and Nuclear Physics*, September 2004.

[92] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189, Annapolis, MD, February 1999. IEEE Computer Society Press.

[93] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, Atlanta, GA, May 1999. ACM Press.

[94] Rajeev Thakur, Robert Ross, and Robert Latham. Implementing byte-range locks using mpi one-sided communication. *Lecture Notes in Computer Science*, September 2005.

[95] The parallel virtual file system 2 (PVFS2). `http://www.pvfs.org/pvfs2/`.

[96] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: a scalable distributed file system. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 224–237, New York, NY, USA, 1997. ACM Press.

[97] Sage Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI '06)*, November 2006.

[98] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. Crush: controlled, scalable, decentralized placement of replicated data. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 122, New York, NY, USA, 2006. ACM Press.

[99] C. Thomas White, Raj K. Singh, Peter B. Reintjes, Jordan Lampe, Bruce W. Erickson, Wayne D. Dettloff, Vernon L. Chi, and Stephen F. Altschul. Bioscan: A vlsi-based system for biosequence analysis. In *ICCD '91: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 504–509, Washington, DC, USA, 1991. IEEE Computer Society.

[100] Joachim Worringen, Jesper Larson Traff, and Hubert Ritzdorf. Improving generic noncontiguous file access for MPI-IO. In *Proceedings of the 10th EuroPVM/MPI Conference*, September 2003.

[101] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999.

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1]The macros used in formatting this dissertation are based on those written by Miguel A. Lerma, (Mathematics, Northwestern University) which have been further modified by Debjit Sinha (EECS, Northwestern University) to accommodate electronic dissertation formatting guidelines.