PARALLEL I/O FOR SHARED MEMORY APPLICATIONS USING OPENMP

A Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Kshitij V. Mehta

August 2013

PARALLEL I/O FOR SHARED MEMORY APPLICATIONS USING OPENMP

Kshitij V. Mehta

APPROVED:

Edgar Gabriel, Chairman
Dept. of Computer Science

Barbara Chapman
Dept. of Computer Science

Jaspal Subhlok
Dept. of Computer Science

Lennart Johnsson
Dept. of Computer Science

Lei Huang
Prairie View A&M University

Dean, College of Natural Sciences and Mathematics

Acknowledgments

I am deeply obliged and thankful to Dr. Edgar Gabriel, my PhD advisor, for his motivation and help throughout my PhD. A master's thesis under him motivated me to pursue a PhD and I have learnt a lot from him over the past few years.

My sincere thanks to my wife Swaroop, for her patience and support all these years. She has been a strong support, and I am very thankful to her.

I am grateful to my parents and my sister who have been supportive of me. Also, I would like to thank my close friends for their encouragement.

PARALLEL I/O FOR SHARED MEMORY APPLICATIONS USING OPENMP

An Abstract of a Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Kshitij V. Mehta

August 2013

Abstract

I/O is a major time-limiting factor in high performance computing (HPC) applications. The combined effects of hard drive latency and bandwidth make I/O the slowest operation in a system. A lot of work has been done in the field of parallel I/O for scientific computing, specially for distributed memory machines. As shared memory systems gain popularity with the increasing number of cores in a node, implementing efficient parallel I/O for shared memory machines has become an important challenge. Currently, popular shared memory programming models like OpenMP do not provide a framework for implementing parallel I/O. This thesis provides a parallel I/O specification for shared memory architecture. In particular, focus has been laid on implementing parallel I/O for OpenMP. In the process, the characteristics of shared memory machines and the behavior of parallel file systems have been studied and an effort has been made to optimize parallel I/O. Also, this research provides insights into semantic analysis of data using the HDF5 technology suite.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last couple of decades, the use of multi-core and multiprocessor architectures has become popular for building systems with high computational power. Supercomputers have been built that can perform operations at the rate of a few petaflops per second [1]. However, performance of storage and memory technologies has essentially fallen behind. This growing disparity between CPU and memory speeds outside the chip is what is called the memory wall [2].

There are many high performance scientific (HPC) applications that work on tremendous volumes of data. For such applications, a serious scalability limitation comes from the performance of I/O operations. This is mainly due to the fact that current hard drives have an order of magnitude higher latency and lower bandwidth than any other component in a computer system. Most popular hard drives have latency in the range of 7-12 ms and average sustained bandwidth of less than 150 MB/sec [3]. Although various techniques such as buffering and caching, RAID configurations, etc. have been used to improve I/O performance, the I/O performance of

an application varies and depends largely on the characteristics of the storage device, the file system being employed, the network interconnect between compute nodes and storage, and the I/O pattern of the application. In the following sections, popular parallel programming paradigms along with techniques for performing parallel I/O are discussed.

1.1 Parallel Programming Models

Parallel computer systems can be divided into two categories, namely, distributed and shared memory systems, on the basis of their architecture. While distributed memory machines have traditionally been of more importance in HPC, recent advances in multi-core architectures have made shared memory machines equally important.

1.1.1 MPI

Applications for distributed memory machines typically use the message-passing model. MPI (Message Passing Interface) is a message-passing library interface specification [4]. An MPI program spawns processes, which communicate with each other by passing messages between them. MPI was developed with the goal of achieving high performance on distributed memory systems, while making applications portable at the same time. There are multiple implementations of MPI, such as OpenMPI [5], MPICH [6], etc.

The Parallel I/O feature introduced with MPI-2 [4] refers to a collection of functions designed for managing I/O on distributed systems. Some important features of MPI-I/O are collective I/O, non-blocking and split-collective I/O, etc. We shall

look at MPI-I/O in detail in the next chapter.

1.1.2 Threads

Threads are the programming model of choice on shared memory systems. A thread represents an execution context within a process. It is represented by a *thread-id* that identifies the thread within a process, a set of register values, a stack, a scheduling priority and policy, a signal mask, and thread-specific data. Everything within a process is shared amongst the threads in the process. Threads require fewer system resources than processes themselves. Inter-thread communication is more efficient, and in many cases easier to use than inter-process communication [7].

The POSIX threads interface is a standardized programming interface for creation and implementation of threads. It has been specified by the IEEE POSIX 1003.1c standard [8]. Implementations which adhere to this standard are referred to as POSIX threads, or Pthreads.

Unix is designed such that when a process forks an additional thread, the thread inherits the attributes of the parent process including file pointers. As such, a file pointer is shared amongst threads and is not unique for every thread.

1.1.3 OpenMP

OpenMP (Open Specifications for Multi Processing) is an Application Program Interface (API) that may be used to achieve multi-threaded, shared memory parallelism [9]. It supports shared memory parallel programming in C, C++, and Fortran. OpenMP is a portable, scalable model which was jointly defined by a group

of major computer hardware and software vendors. It gives shared memory parallel programmers a simple and flexible interface for developing parallel applications.

Most parallelism in OpenMP is specified through the use of compiler directives which are inserted in the source code. OpenMP requires explicit parallelization by the programmer, and hence is not automatic. An underlying OpenMP implementation would typically use threads as a means of achieving parallelism, using the fork-join model of parallel execution, as shown in Figure 1.1. OpenMP currently does not provide a way to perform I/O in parallel. Performing I/O in parallel in an OpenMP application can be a difficult task.

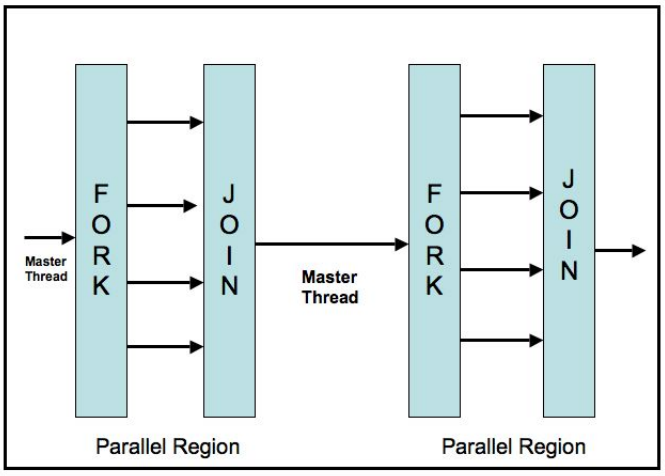

Figure 1.1: OpenMP fork-join model.

Another set of programming models includes Partitioned Global Address Space (PGAS) languages like UPC [10], Co-array Fortran [11], etc. which focus on presenting a single shared, partitioned address space where variables may be directly

read and written by any processor, but each variable is physically associated with a single processor. Unified Parallel C (UPC) [10] is an extension of the C programming language designed for high performance computing on large-scale parallel machines. The language provides a uniform programming model for both shared and distributed memory hardware. The programmer is presented with a single shared, partitioned address space, where variables may be directly read and written by any processor, but each variable is physically associated with a single processor. UPC uses a Single Program Multiple Data (SPMD) model of computation in which the amount of parallelism is fixed at program startup time, typically with a single thread of execution per processor. UPC has a parallel I/O specification which is similar in terms with the MPI-IO specification.

1.2 Factors affecting I/O Performance

I/O can play a significant role in the overall time to completion for many parallel applications that read and write high volumes of data from disk. I/O is often a bottleneck in compute-intensive applications. The combined effects of latency and bandwidth cause I/O to be the slowest operation in a system, which is a critical factor in applications whose performance is largely measured by the time to completion. Table 1.1 shows the typical access times for different types of memory.

Although I/O is always much slower than computation, a combination of high-speed I/O hardware, appropriate file-system software, and a suitable programming interface (API) for I/O can improve I/O performance. Caching, buffering of data,

Table 1.1: Memory access times.

Memory type	*Size*	*Access Time (cycles)*
Backup (Tape)	Terabytes, Petabytes	seconds
Primary data storage(disk)	$\sim 100GB$	$> 10^6$
Main Memory	1-4 GB	100-1000
Caches	1-4MB	2-50
Registers	$<$ 256 words	1-2

and disk striping are important techniques that are applied to improve the performance of I/O.

1.2.1 Buffering and Prefetching Data

Data buffers are memory regions that are used to temporarily store data while it is being moved. Data buffers play an important role when the data transfer rates between the transferring units are not the same. Hence, data buffers are useful in hiding the disk latency for write operations.

Prefetching is a technique that hides latency for read operations. It predicts future data accesses and requests data before it is requested. It thus overlaps computation with fetching data, but only works for contiguous access patterns.

1.2.2 Disk Striping

The concept of disk striping lays the foundation for parallel I/O and parallel file systems. Disk striping is the use of multiple hard disks in a file system such that file data are broken down into blocks of data and distributed on different hard drives. Typically, these hard drives reside on separate nodes so that the system can provide

high degree of parallelism during file I/O.

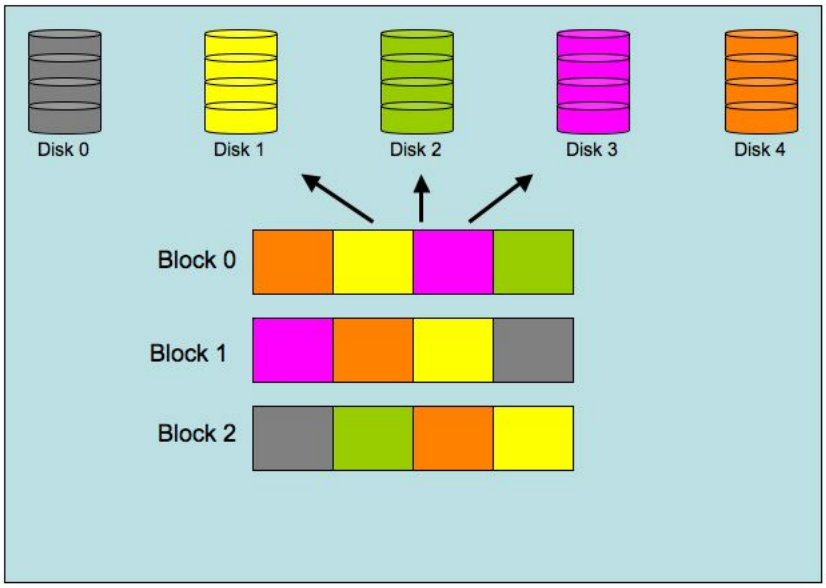

Figure 1.2: Disk striping.

Figure 1.2 shows an example of disk striping. The number of disks in this simple striping scheme is called the *stripe factor*. The *stripe factor* determines the degree of parallelism.

Disk striping is utilized in parallel file systems. In parallel file systems, multiple, dedicated nodes serve as I/O nodes and are connected to the rest of the cluster. An application can now access multiple file data chunks in parallel from different I/O nodes. Parallel file systems will be discussed in detail in the next chapter.

1.2.3 Access Patterns

Access patterns play an important role in the I/O performance of an application. An access pattern can be classified as contiguous or noncontiguous depending on the layout of memory blocks in memory or in file. As shown in Figure 1.3, contiguous I/O moves data from a single memory block into a single file region. Contiguous I/O is generally easy to manage and perform. Noncontiguous I/O often becomes a limiting factor in achieving good system throughput since it requires issuing many small I/O requests. Memory latency contributes significantly when noncontiguous I/O is a dominant pattern.

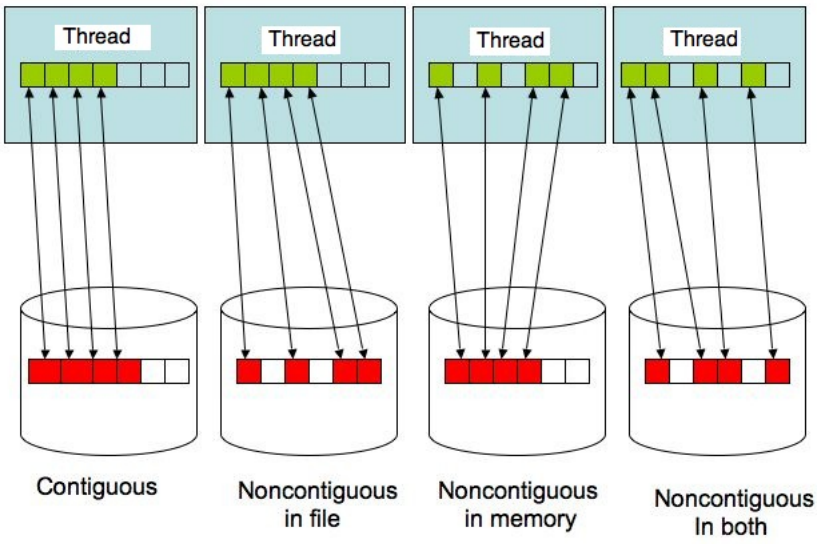

Figure 1.3: Data layout in memory and file.

As will be discussed later, popular parallel I/O libraries try to perform various optimizations on noncontiguous data to overcome the effects of memory latency.

1.2.4 File Pointers

A file pointer defines the current position in the file. File pointers to the same file in a parallel environment may be independent or shared. Each process may have an independent file pointer that is updated independently of the other processes. To avoid overlapping file accesses, each process needs to know what part of the file the others will use.

There are situations in which all processes need a common view of the file pointers, in which case the I/O library provides shared file pointers. Shared file pointers allow coordinated access to the pointer between processes.

In scientific applications, many processes access the same file concurrently. They read and write data to a file simultaneously, instead of one process performing I/O while other processes wait on it.

A good parallel I/O library would allow multiple processes to perform I/O to/from a common file efficiently.

1.3 Low Level I/O

In this section, we look at the low level interfaces available for performing I/O.

1.3.1 POSIX

POSIX provides a list of functions for performing I/O that include basic file manipulation, regular I/O primitives, list I/O functions, memory mapping etc.

- File Manipulation:

  The *open* function creates and returns a new file descriptor for a file. The *close* function closes the file associated with the input file handle.

- Basic I/O primitives:

  Functions *read* and *write* perform I/O on an input file handle that was opened using the *open* function call. Data are read from or written to the position in file pointed to by the file handle. The *pread* and *pwrite* functions provide similar functionality, except that a user provides an explicit offset in file to these functions.

- File Positioning:

  The *lseek* function allows to set the file position of an open file to a value specified by the user as an input argument.

- List I/O:

  The *readv* and *writev* functions scatter/gather data into buffers in memory. These functions accept a list of data pointers as input argument and provide the functionality of performing more I/O tasks in a single function call than regular I/O primitives.

- Memory mapping:

  This set of functions allows mapping a file to a region of memory. When this is

done, the file can be accessed just like an array in the program. This could be more efficient than read or write, as only the regions of the file that a program actually accesses are loaded. Accesses to not-yet-loaded parts of the mmapped region are handled in the same way as swapped out pages.

- Asynchronous I/O:

  This set of functions allows a program to initiate one or more I/O operations and then immediately resume normal work while the I/O operations are executed in parallel. This feature can significantly reduce the time an application spends waiting at I/O.

POSIX semantics dictate that a file can be connected to multiple processes/threads at a time. Writes to a file must be immediately visible to all other processes, and all file accesses must be performed in an atomic manner.

1.3.2 C I/O

The C standard I/O library exports a set of functions that work on file data as streams. It provides functions like *fopen*, *fclose*, *fwrite*, *fread* etc. The main difference is that a data stream is associated with an open file and the I/O is buffered. It is overall similar in many aspects to POSIX, though POSIX provides a larger set of functions that facilitate greater control on file I/O.

1.3.3 Fortran I/O

Fortran I/O defines a file as a sequence of records. In fortran 2008 [12], a file is composed of either a sequence of file storage units (stream file) or a sequence of

records. to be a record.

A file in Fortran may be accessed in a sequential or direct manner. When connected for sequential access, the order of the records is the order in which they were written. When connected for direct access, each record of the file is uniquely identified by a positive integer called the record number. The order of the records is the order of their record numbers.

Programming in Fortran is inherently sequential, it does not support a multi-threaded programming model. An extension to Fortran 95 is co-array fortran [11], which provides an explicit notation for data decomposition for MPI-style SPMD programming.

1.4 Challenges

Traditional low level I/O functions were originally written keeping the sequential execution model in mind. These functions do not cater to performing I/O in parallel easily; for example, when multiple processes call an I/O routine, it is the programmer's responsibility to make sure that simultaneous calls to these functions do not yield undesired results. As such, these I/O routines are not best suited for being used directly in applications for parallelizing I/O.

A significant amount of work has been done in achieving high performance I/O in distributed memory systems through the use of specifications like MPI-IO. Various techniques and optimizations for parallel I/O have been studied and implemented in MPI-IO, which are discussed in the next chapter. However, shared memory systems differ fundamentally from distributed memory machines. The techniques and

optimizations applied in a cluster of computers for high performance I/O cannot be translated directly to a shared memory system. While message passing along with I/O are the main contributing factors to the I/O performance in an MPI application, one should note that exchanging messages or sharing data in a shared memory system is relatively much cheaper.

Further, the recent past has shown a rising interest in incorporating shared memory machines in HPC systems, since space constrains, power constraints and limitations of message passing limit our ability to use clusters with fewer cores per node effectively in reducing application run time. In short, one cannot stack up thousands of few-core machines to attain desired compute and I/O bandwidth. Parallel I/O in shared memory systems remains widely unexplored and it is only a matter of time when it becomes the pushing need of the hour.

The following points are recognized to pose challenges in developing a parallel I/O model for shared memory architecture.

- Lack of a parallel I/O framework for shared memory programming models:

  With the increasing number of cores in a single node and the tremendous I/O requirements of scientific applications, provision of parallel I/O in parallel programming models for shared memory systems will become a necessity. Currently, popular shared memory parallel programming models like OpenMP do not provide support for parallel I/O.

- Performance:

  The performance of I/O depends on a variety of factors, ranging from architecture-dependent features like number of threads, amount of main memory, thread

affinity,etc. to the behavior of parallel file systems being utilized. As such, providing an optimized implementation of parallel I/O is a non-trivial task and requires us to work on various levels of an HPC system.

- System and application specific parameters:

  Thread-based parallel I/O can be different from process-based parallel I/O as in MPI, since threads share all resources on a single node. It is challenging and important to understand the influence of factors like thread affinity, number of threads as I/O aggregators, size of data chunks being read to or written, etc. Behaviour of I/O also changes with variation in system configuration, such as amount of main memory, type of network interconnect, number of I/O servers, etc.

- Scalability:

  As the number of cores in a single node keeps on increasing, a parallel I/O implementation must prove to be scalable with the increasing number of threads in an application. Providing scalability guarantees is a challenging, as various other factors apart from number of threads need to be considered to ensure scalability with optimal system throughput.

- Robustness:

  As we think about designing a parallel I/O framework that is aimed to work with a shared memory programming model like OpenMP, it is necessary to understand how a parallel I/O library should work in collaboration with the features of OpenMP, as well as how OpenMP is likely to evolve in the future.

As an example, it is important to decide how the parallel I/O library should work with OpenMP loops, sections, tasks, nested parallelism, directives and runtime calls, etc. Also, upcoming versions of OpenMP could introduce the notion of subteams of threads, asynchronous tasks, etc. Hence, a parallel I/O framework must be designed keeping these features in mind so that it can adapt gracefully to such changes without requiring large-scale modifications to the parallel I/O implementation.

- Hybrid programming:

  As scientific applications are ported on large clusters that contain fat nodes, it would be necessary for distributed memory technologies like MPI to collaborate with shared memory paradigms like OpenMP. Constructing a hybrid programming model is a challenging task.

- Novel methods for storing Exascale data

  As we get closer to the era of exascale computing, the amount of data generated by applications is expected to rise, possibly even exponentially. Low level libraries would have to analyze data and store it accordingly, and not just focus on ways to improve performance of I/O. Libraries like HDF5 that serve data centric applications are candidates that can benefit from such techniques.

1.5 Goals

This thesis aims to develop a parallel I/O framework for shared memory systems. While it is not possible to work on all challenges associated with parallel I/O as

discussed in the previous section, focus has been laid on developing concepts and an exhaustive framework along with optimized algorithms.

- Parallel I/O Specification for OpenMP

  This thesis aims to propose a parallel I/O specification for OpenMP. We aim to explore various possibilities of incorporating parallel I/O in OpenMP in conjunction with its existing constructs. In particular, the specification should be portable across various unix-like platforms.

- Parallel I/O library for shared memory machines

  This thesis aims to develop an optimized parallel I/O library for shared memory machines. This implementation would allow threads to write to the same file without explicitly having to lock the file handle. As such, we aim to surpass the traditional issues seen when threads write to the same file, and hence also avoid requiring threads writing to separate files to exploit maximum available bandwidth.

- Identifying I/O Optimizations for Shared Memory Applications

  This research studies the I/O characteristics of various file systems including PVFS2, Lustre, and how the performance of parallel I/O differs on each of these file systems specifically from the point of view of shared memory I/O. Factors such as the optimal number of threads required to perform I/O, best I/O practices for different file systems, optimal size of data chunks being read/written, etc. are studied.

- Improving I/O performance and enabling semantic analysis of data in shared

memory applications

Finally, this thesis aims to propose ways to facilitate applications to perform semantic analysis on data in an application library like HDF5. The objective is to store data in a more object-based format rather than a linear array of bytes, and improve performance of I/O at the same time.

The rest of this document is organized as follows: Chapter 2 gives an overview of the current state of the art. It includes the various techniques, optimizations and other developments in the field of parallel I/O. Chapter 3 provides details about proposed parallel I/O extensions for OpenMP as part of this research. It includes details about a parallel I/O library developed for shared memory machines and its integration with the OpenUH compiler. Chapter 4 describes plugins developed for the HDF5 technology suite that provide multi-threading to obtain parallel I/O and store data in an innovative manner. Chapter 5 provides a summary of the research performed and concludes by highlighting areas of future research in the field.

Chapter 2

Background and Related Work

In this chapter, we look the current state of the art in parallel I/O. We discuss various techniques and optimizations performed to implement parallel I/O in high performance computing systems.

2.1 I/O Options in Parallel Applications

This section describes the two basic techniques for processes in an MPI application to perform I/O, viz. independent and collective I/O.

2.1.1 Independent I/O

Independent I/O is a technique in which each process/thread of an application issues I/O requests independently of other processes/threads. It is a straight-forward form of I/O which has traditionally been used in many applications.

Independent I/O suffers from a few important disadvantages when used in parallel applications. First, a process or a thread does not collaborate with other processes/threads and issues its own I/O requests. In a scientific application, this can soon lead to a large number of small I/O requests being spawned, which are typically harmful for performance of I/O operations. Second, unless the system is configured with sufficient I/O channels with enough network bandwidth to satisfy the application's requirements and a number of parallel disks, threads may experience conflicts while accessing data that resides on the same disks.

Independent I/O does not capture the complete data access pattern of a parallel application, which causes the underlying parallel I/O library to lose the opportunity of performing optimizations with the knowledge of multiple processes/threads. This is in contrast to collective functions, which allow I/O requests from multiple processing elements to be serviced together.

2.1.2 Collective I/O

When applications spawn a large number of small I/O requests, they are often interleaved such that they together span large contiguous portions of a file. Collective I/O [13, 14] is a class of optimizations that improves performance by merging separate I/O requests. Collective I/O merges requests across multiple processes/threads. The semantics of collective I/O require all threads to call a collective routine. Collective write operations gather data from multiple threads into large, contiguous chunks before storing it to disk. Collective read operations retrieve large chunks of disk and distribute this data to multiple requesting threads. This reduces the number of disk

accesses and makes each access more efficient.

Collective I/O has generally been explored in two different ways. Two-phase I/O performs collective I/O at the client side, whereas disk-directed I/O employs collective I/O at the disk or the I/O server level.

2.2 Client- and Server-side Optimizations

In this section, we look at various techniques adapted at the client- and server-side to implement parallel I/O or make improvements to it.

2.2.1 Two-phase I/O

On the client side, two-phase I/O [15] is an optimization technique that uses collective I/O in two phases. The first phase consists of the communication phase in which processes/threads analyze their independent I/O operations to determine what data regions must be transferred between them. These regions are then split up between a set of aggregator processes that interact with the file system. Thus the first phase consists only of communication between processes/threads. In the second phase, the aggregator threads perform the actual read or write operations.

The advantage of two-phase I/O is that by making all file accesses large and contiguous, the I/O time is reduced significantly. The added cost of interprocess communication for redistribution is small compared with the savings in I/O time.

2.2.2 Data Sieving

Data sieving is a technique introduced in ROMIO [13], a portable implementation of MPI-IO. Data sieving works in a way that when processes/threads make requests for noncontiguous chunks of data, the underlying implementation actually fetches a big contiguous block starting from the first byte upto the last requested byte into a temporary buffer in memory. It then puts the requested data from the temporary buffer into the calling process's/thread's buffer. The advantage of using such a technique is that we now make fewer I/O requests to disk, and hence, performance deterioration due to combined effects of I/O latency are alleviated. Although we read more data than is required, the *holes* between the required chunks might not be too large, and the cost of reading additional data is negligible as compared to the cost of issuing multiple I/O requests.

A problem with this algorithm could be satisfying the memory requirements. The size of the temporary buffer must be as large as the total amount of data being read spanning the user's request. The system could run out of memory is this requirement is too large.

Similarly, data sieving can be used for writing data. A read-modify-write must be performed, to avoid destroying the data already present in the holes between the contiguous data segments.

2.2.3 Disk-Directed I/O

Disk-directed I/O [16] is a feature in which compute nodes collectively send a request to all I/O processors, which then arrange the flow of data to optimize disk, buffer,

and network resources. The I/O processors control the order and timing of the flow of data. When an I/O processor receives a data request, it first determines the set of file data local to it. It then determines the file blocks needed and sorts them to optimize disk movement, and as each block arrives from the disk, it sends it to the appropriate compute node.

Some of the improvements that disk-directed I/O shows are that now the I/O can conform to both the logical, as well as the physical layout of the data in file. There is only one request submitted to each I/O server. Also, disk scheduling is improved, thereby improving performance of I/O overall. Although disk-directed I/O has not yet been implemented extensively in many I/O intensive systems, experimental simulations have shown that it is a promising technique.

2.2.4 Compiler-Directed I/O

Another technique for improving parallel I/O performance can be applied at the compiler level [17]. The idea stems from the fact that while independent I/O can be seen to be harmful for performance for many cases, applying collective I/O indiscriminately can also deteriorate performance. Hence, it can be beneficial to let a compiler select the cases where independent I/O should be used, and where collective I/O should be preferred.

In compiler-directed I/O, the compiler analyzes the data access patterns of an application and determines suitable file storage patterns and I/O techniques. The compiler implements collective I/O only when necessary. For other cases, it defaults

to an application's naive implementation of I/O, where processes perform independent I/O.

Compiler-directed I/O can be complicated as it requires working at the core compiler level. Analyzing of the source code of the application and the file storage patterns can be non-trivial.

2.3 Parallel File Systems

One of the important pieces of hardware that largely affects the performance of parallel applications are parallel file systems. Here we look at some of the file systems that have been popularly used with scientific applications.

2.3.1 PVFS2

PVFS2 provides a high-performance and scalable parallel file system for clusters [18]. PVFS2 is open source and released under the GNU General Public License [19]. PVFS2 provides four important capabilities in one package:

- a consistent file-name space across the machine
- transparent access for existing utilities
- physical distribution of data across multiple disks in multiple cluster nodes
- high-performance user space access for applications

Figure 2.1 shows how nodes might be assigned for use with PVFS2. Nodes are divided into compute nodes on which applications are run, a management node

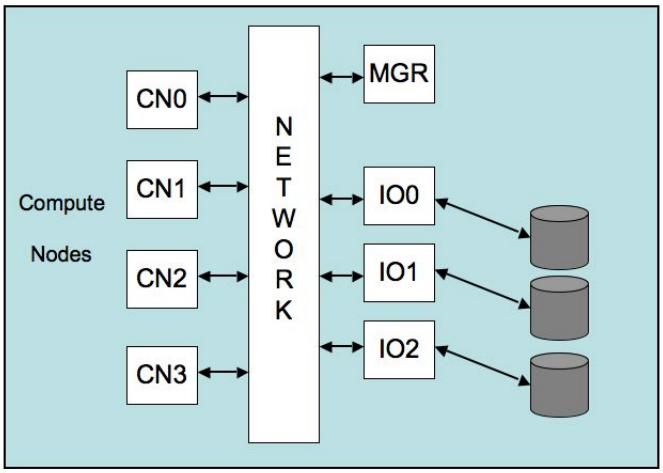

Figure 2.1: PVFS2 system view.

which handles metadata operations, and I/O nodes which store file data for PVFS2 file systems.

There are four major components to the PVFS2 system:

1. Metadata server (mgr)
2. I/O server (iod)
3. PVFS2 native API (libpvfs)
4. PVFS2 Linux kernel support

The metadata server and the I/O server are daemons which run on nodes in the cluster. The metadata server, named *mgr*, manages all file metadata for PVFS2 files. Metadata describes a file, that is, its name, its location in the directory hierarchy,

its owner, and how it is distributed across nodes in the system. The I/O server handles storing and retrieving file data stored on local disks connected to the node. These servers create files on an existing file system on the local node, and use the traditional read(), write(), and mmap() operations for access to these files.

The PVFS2 native API provides user-space access to the PVFS2 servers. It handles the scatter/gather operations necessary to move data between user buffers and PVFS2 servers, while keeping it transparent to the user. For metadata operations, applications communicate through the library with the metadata server. For data access, the metadata server is eliminated from the access path and instead I/O servers are contacted directly.

Finally, the PVFS2 Linux kernel support provides the functionality necessary to mount PVFS2 file systems on Linux nodes. This allows existing programs to access PVFS2 files without any modification. This support is not necessary for PVFS2 use by applications, but it provides an extremely convenient means for interacting with the system.

2.3.2 Lustre

Lustre is an open-source, distributed parallel file system [20]. It is an object-based file system. It consists of three main components: clients, metadata servers (MDSs), and object storage targets (OSTs). It allows data access through standard POSIX calls. Lustre provides essential file system services through distributed lock management. A client creates a file through an MDS. The MDS creates objects on all OSTs. The clients now communicate with the OSTs directly, without requiring the help of the

metadata server.

Lustre stripes file data between a number of OSTs. Striping can be specified on a per-file or a per-directory basis. The configurable parameters are stripe size, stripe width, and stripe index. Lustre serializes data access to a file using distributed lock management. All processes first have to acquire a lock before they can update a shared file. Thus processes performing I/O on a Lustre file system are likely to see effects of lock contention. In Lustre, each I/O server maintains locks for the file stripes its stores. If a client requests a lock currently held by another client, the lock holder is requested to release the lock. Lustre uses an extent-based locking mechanism for granting locks. When a client requests a lock, the file system makes an effort to grant the lock for the largest file region possible. For example, the first requesting process to a file is granted a lock on the entire file. When another process makes a request for a non-overlapping part of the file, the first process will relinquish that part of the file to the requesting process. The lock granularity of a file system, defined as the smallest size of file region a lock can protect, is generally set to the file stripe size for Lustre.

2.3.3 GPFS

The IBM General Parallel File System (GPFS) [21] is a commercial file system. Applications can access files through standard file system interfaces. GPFS provides data access in case of node failures. Unlike Lustre, data and metadata servers are not separated and can be handled by the same servers. It uses token passing to implement file locking. Like Lustre, GPFS is POSIX compliant.

Other popular parallel file systems include GFS [22], Panasas [23], etc.

2.3.4 PLFS

PLFS (Parallel Log-Structured File System) is a middle-ware virtual file system developed at Los Alamos National Lab (LANL) [24]. It is situated between the application and the parallel file system responsible for the actual data storage. PLFS was developed to tackle the issue of poor I/O performance exhibited by certain parallel file systems when certain conditions arise.

Popular parallel file systems like Lustre, GPFS [21] etc. are known to exhibit sub-standard performance when multiple processing elements access a common, shared file (this access pattern is generally referred to as the N-1 pattern) [25, 26]. In contrast with the N-1 access pattern, the N-N pattern is known to demonstrate better performance. Here, N processes access N files, that is, one file per process. Although N-N performs better in general than N-1, many applications prefer to write data to a single file rather than modifying the application's data pattern to fit N-N access. PLFS was specifically designed to obtain optimal I/O throughput in these applications without requiring them to make changes in the way the application stores data.

PLFS is an interposition layer that transparently rearranges an N-1 access pattern into an N-N pattern. It converts writes to a shared logical file into writes to multiple physical files. As shown in Figure 2.2 it transforms N-1 into N-N, where every process participating in I/O writes data to its own, separate file. The basic operation of PLFS is as follows. For every writer to a logical file, PLFS creates a unique physical file on

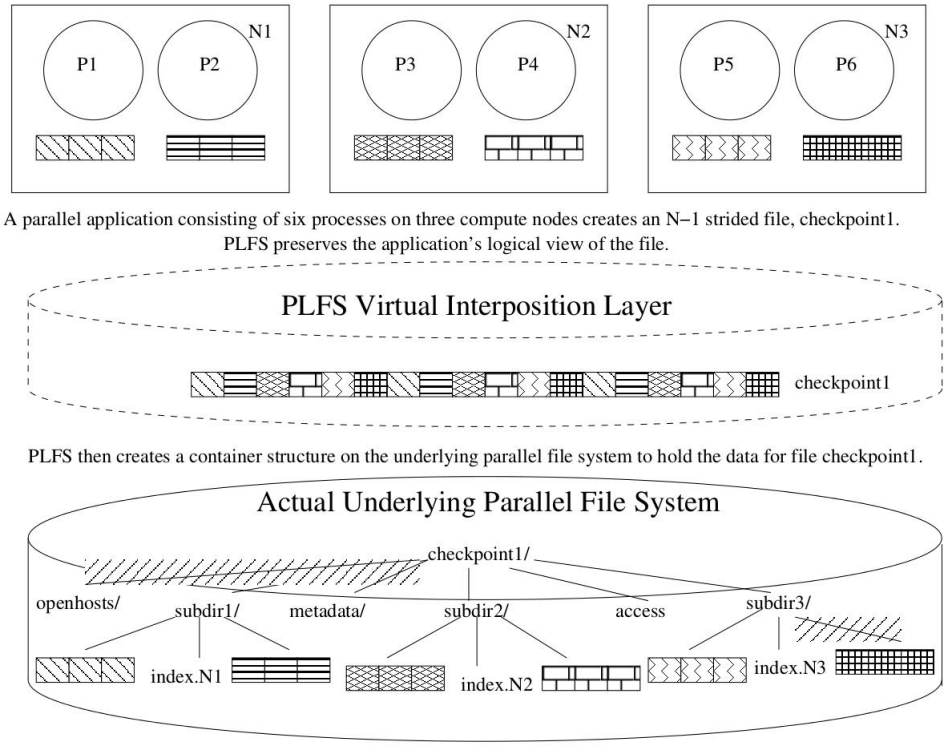

Figure 2.2: Converting N-1 access pattern into N-N in PLFS.

the underlying parallel file system and maintains sufficient metadata to recreate the shared logical file in the correct order of writes issued.

Users can interface with PLFS directly by using the PLFS API or by using its MPI-IO driver (ad_plfs). PLFS has demonstrated benefits of converting this N-1 access pattern into N-N for a variety of applications and parallel file systems.

2.3.5 IOFSL

At this point, it would be worthwhile to mention the concept of I/O forwarding. I/O forwarding is the technique of forwarding I/O requests from compute nodes

to dedicated I/O nodes that perform I/O to a parallel file system on behalf of the compute processes. Generally, a subset of compute nodes maps to an I/O node which invokes the corresponding file system calls. There are many advantages related to this scheme. The I/O traffic between compute nodes and the file system is regulated and the load on the file system is reduced since the file system sees fewer clients. Secondly, data can be cached at the I/O nodes which allows them to aggregate and reschedule I/O requests. The IO Forwarding Scalability Layer (IOFSL) is a scalable, unified high-end layer that implements I/O forwarding that addresses the issue of the lack of a portable, open source implementation. It supports multiple network interconnects and parallel file systems.

2.4 Parallel I/O Specifications

2.4.1 MPI-I/O

MPI-IO [4] refers to a collection of functions designed for managing I/O on distributed systems. It also allows files to be easily accessed in a patterned fashion using the existing derived datatype functionality.

In this section, we look at parallel I/O techniques provided by MPI-IO.

Definitions:

Communicator: An MPI communicator specifies a group of processes inside which communication occurs.

Etype: An etype (elementary datatype) is the unit of data access and positioning.

File View: A file view defines the current set of data visible to a process and accessible from an open file. Each process has its own view of the file.

MPI I/O functions can be categorized into the following classes:

1. Simple I/O for File Manipulation:

   e.g. `MPI_File_open` - A collective routine for opening an MPI file.

   `MPI_File_close` - MPI routine for closing a file.

2. Data-Access Routines:

   Data are moved between files and processes by issuing read and write calls. There are three orthogonal aspects to data access: positioning (explicit offset vs. implicit file pointer), synchronism (blocking vs. nonblocking), and coordination (non-collective vs. collective).

   Examples:

   `MPI_File_read_at` - explicit, blocking, non-collective

   `MPI_File_read_at_all` - explicit, blocking, collective

3. Collective I/O:

   Collective functions are called by all processes in the communicator. For example, `MPI_File_read_all` is called by all processes in the communicator that was passed to the `MPI_File_open` function with which the file was opened.

   Examples:

   `MPI_File_read_all`, `MPI_File_write_all`

4. Non-blocking I/O and Split Collective I/O:

   Non-blocking functions allow overlapping I/O with other computation/communication

in a program. MPI supports non-blocking versions of all independent read/write functions.

For collective I/O, MPI supports only a restricted form of non-blocking I/O called split collective I/O. To use split collective I/O, a user must call a begin function to start the collective I/O operation and an end function to complete the operation.

2.4.2 UPC

Amongst other features for parallel computation, UPC provides functionality for parallel I/O [27]. All UPC-IO functions are collective and must be called by all threads collectively. Collective UPC-IO accesses can be done in and out of shared and private buffers, thus local and shared reads and writes are generally supported. In each of these cases, file pointers could be either common or individual. UPC-IO also provides file-pointer-independent list file accesses by specifying explicit offsets and sizes of data that is to be accessed. Non-contiguous accesses may be performed using lists of explicit offsets and lengths in the file using list I/O interfaces. I/O operations can be synchronous (blocking) or asynchronous (non-blocking). Synchronous calls block and wait until the corresponding I/O operation is completed. On the other hand, an asynchronous call starts an I/O operation and returns immediately.

UPC consistency semantics state that data written by a thread is only guaranteed to be visible to another thread after all threads have called upc_all_close or upc_all_file_sync. Writes from a given thread are always guaranteed to be visible to subsequent reads by the same thread.

2.5 External libraries and software

2.5.1 ROMIO

ROMIO is a high-performance, portable implementation of MPI-IO [28]. ROMIO is designed to be used with any MPI implementation and is included as part of several MPI implementations. It utilizes an Abstract-Device Interface called ADIO [14] for file system specific operations. ADIO allows to reduce the number of functions that have to be modified for a file system. This approach allows users to use the high level interface whereas ADIO maps it to the file system desired.

ROMIO provides a client-side collective I/O implementation, based on the two phase strategy described previously. Two phase I/O consists of two steps; one where processes exchange information about data patterns and second, the I/O step. It uses data sieving where large chunks of data, starting from the first offset requested upto the last offset, are read/written taking care to make sure the *holes* are left unmodified.

ADIO is an abstract-device interface for parallel I/O [29]. It enables any parallel I/O API to be implemented on different file systems by implementing the API portably on top of ADIO. The usability of ADIO stems from the fact that instead of having a number of different APIs supported by different vendors, it is more advantageous to have a single API. This enables users to run applications on a wide range of platforms, regardless of the parallel I/O API used in the applications. It is not intended to be used directly by application programmers, but is a strategy for implementing other APIs. ADIO consists of a small set of basic functions for performing

parallel I/O. It must be implemented in an optimized manner on each different file system. The functions included in ADIO cover the following functionality:
Opening and closing of files, noncontiguous reads and writes, nonblocking reads and writes, collective reads and writes, seeking, test and wait, file control, and some other miscellaneous routines.

Some of the advantages of using ADIO are:

- Portability

  ADIO provides a single API that users can use over different file systems. The underlying implementation must be optimized for each file system. The application programmer is freed from the intricacies of writing optimized I/O routines for each file system.

- Experiment with new APIs

  ADIO enables users to experiment with new APIs and new file-system interfaces. Once a new API is implemented on top of ADIO, it becomes available on all file systems on which ADIO has been implemented.

2.5.2 OMPIO

OMPIO [30] is the parallel I/O architecture of Open MPI [5]. OMPIO separates parallel I/O functionality into frameworks. This allows to encapsulate various aspects of parallel I/O into smaller functional units, such as dealing with file system specific operations, individual I/O, collective I/O, or shared file pointer operations. Each framework typically has multiple modules providing the required functionality, each module being designed for different scenarios. The selection criteria that determines

which module is being used is highly dependent on the functionality provided by a framework and on external parameters such as the file system utilized, hardware configuration, process placement by the batch scheduler or application characteristics. It is a highly modular and flexible architecture that allows one to dynamically choose a different module in each framework independent of other aspects of the parallel I/O library.

2.5.3 ADIOS

The Adaptable IO System (ADIOS) [31] provides a simple, flexible way for scientists to describe the data in their code that may need to be written, read, or processed outside of the running simulation. A user can set various options in an XML file that tell the library how to process data. For example, the user could select MPI individual I/O, collective I/O or POSIX I/O and simply restart the application, without having to recompile the code. Having an XML file allows scientists to change how the IO in their code works simply by changing a single entry in the XML file and restarting the code. ADIOS provides the advantage of switching to a different I/O method for a different platform in a simple manner, since different methods show variations in their behavior over different HPC environments. The XML file allows switching to the best I/O method known for a particular platform without change to the source code.

2.5.4 HDF5

Hierarchical Data Format (HDF5) is a technology suite for efficient management of large and complex data [32]. It supports complex relationships between data and dependencies between objects. HDF5 is widely used in industry and scientific domains, in understanding global climate change, special effects in film production, DNA analysis, weather prediction, financial data management etc. [33] Parallel HDF5 (PHDF5) [34] enables developing high performance, parallel applications using standard technologies like MPI in conjunction with HDF5. HDF5 is a versatile data model containing complex data objects and metadata. Its information set is a collection of datasets, groups, datatypes and metadata objects. The data model defines mechanisms for creating associations between various information items. The main components of HDF5 are described below.

File: In the HDF5 data model the container of an HDF5 infoset is represented by a file. It is a collection of objects that also explains the relationship between them. Every file begins with a root group "/", which serves as the "starting-point" in the object hierarchy.

Dataset: HDF5 datasets are objects that represent actual data or content. Datasets are arrays which can have multiple dimensions. A dataset is characterized by a dataspace and a datatype. The dataspace captures the rank (number of dimensions), and the current and maximum extent in each dimension. The datatype describes the type of its data elements.

Group: A group is an explicit association between HDF5 objects. It is synonymous with directories in a file system. A group could contain multiple other groups,

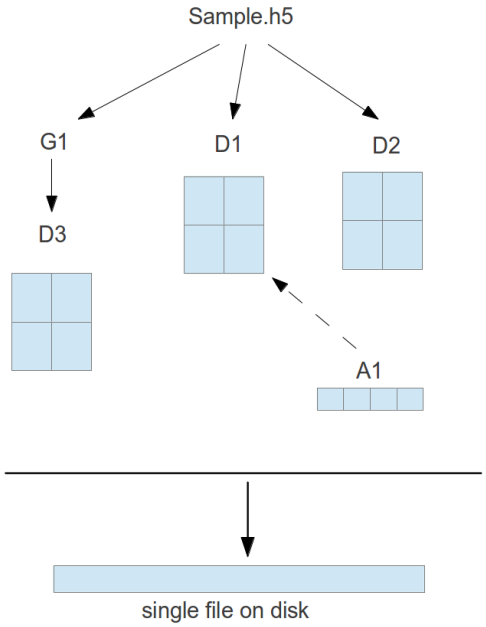

Figure 2.3: A sample HDF5 file.

datasets or datatypes *within* it.

Attribute: Attributes are used for annotating datasets, groups, and datatype objects. They are datasets themselves, and are *attached* to existing objects they annotate.

For example, as shown in Figure 2.3, the file "Sample.h5" contains the root group which itself contains a group G1 and two datasets, D1 and D2. Group G1 contains a dataset D3. Attribute A1 is linked to dataset D1. The objects and the relationships between them can be represented as a B-tree, which is used internally by HDF5 to index its objects.

An HDF5 file is a self-describing format which combines data and metadata. It is a container in which users typically store multiple HDF5 objects alongside their metadata.

HDF5 supports parallelism using MPI. A PHDF5 (Parallel HDF5) application has multiple processes accessing a single file (i.e. N-1 access pattern). PHDF5 exports a standard parallel I/O interface which itself uses MPI's parallel I/O functionality. This is used along with parallel file systems to achieve high performance I/O in data centric applications.

2.5.5 NetCDF

NetCDF [35] is an abstraction that views data as a collection of array-oriented objects. The file format is self-describing in the sense that the NetCDF file contains header information that describes the data in the file. It exports an interface for storing and retrieving data in the form of arrays from files. NetCDF has many similarities to HDF5, but while HDF5 is hierarchical, PNetCDF presents a linear data layout. PNetCDF provides a parallel interface for accessing NetCDF datasets. The underlying parallel I/O utilizes MPI-IO to achieve high performance gains through the use of collective interfaces.

2.6 Summary

As described in the previous sections, the most widely used parallel I/O specification is based on the Message Passing Interface (MPI) [36], which has introduced the notion of parallel I/O in version two. Secondly, UPC provides an abstraction for parallel I/O [27] mostly following the MPI I/O specification.

However, no specification for parallel I/O exists yet in shared memory programming models like OpenMP.

iHarmonizer [37] by Weng et al., optimizes I/O operations in multi-threaded applications. It uses a scheme where a separate I/O thread is used to prefetch data according to information provided by other threads. It accesses data in the order most friendly to the disk. However, it uses a single thread dedicated for prefetching data and does not address HPC type I/O workloads.

Chapter 3

Parallel I/O for OpenMP

In this chapter, we discuss various design alternatives for parallel I/O interfaces in OpenMP, followed by the actual specification and its implementation in the OpenUH compiler.

3.1 I/O options in multi-threaded applications

I/O options are limited as of today for applications using shared memory programming models such as OpenMP [38]. Most OpenMP applications use the routines provided by the base programming languages (e.g. Fortran, C, C++) for accessing a data file. In order to maintain consistency of the resulting file, read and write operations are performed outside of parallel regions. In case multiple threads are accessing a file, access to the file handle is protected within an *omp critical* construct to avoid concurrent access by different threads. If serialized access in thread order is required, one could use an *ordered omp for* with a *static* schedule. Unfortunately,

this would be considered *non-conforming* even though it is technically consistent with the OpenMP 3.0 API specification. Facilitating an arbitrarily complex scheme might require some very creative combinations of worksharing constructs and locks. This however would quickly become unsightly, non-conforming and unmaintainable. Even with tricks, some common approaches may not be possible since the primary goal of OpenMP is to make worksharing and reasoning about program correctness easier for the programmer. At no point, however, does it attempt to address cross-cutting issues such as I/O.

Another approach has each thread utilizing a separate file to avoid race conditions or synchronizations when accessing a single, shared file. While this approach often leads to a better performance than the previously discussed methods, it has three fundamental drawbacks. First, it requires (potentially expensive) pre- and post-processing steps in order to create the required number of input files and merge the output files of different threads. Second, it is difficult to support application scenarios where the number of threads utilized is determined dynamically at runtime. Third, managing a large number of files often creates a bottleneck on the metadata server of the parallel file system. Congestion on the metadata server might not be an issue for a smaller number of threads, but it could become relevant in the near future as the number of cores of modern micro-processors is expected to grow into the hundreds or even thousands. This will lead to an according increase in the number of threads used by parallel applications.

3.2 Specification

In this section, we discuss the various alternatives for implementing a parallel I/O framework in OpenMP, followed by the actual specification.

3.2.1 Directive based interfaces vs. Runtime based library calls

As discussed before, a user can express parallelism in OpenMP using directives. The primary design decision while designing a framework for parallel I/O is whether to use compiler directives to indicate parallel execution of read/write operations, or whether to define an entirely new set of library functions. The first approach would have the advantage that the changes made to an application are minimal compared to using an entirely new set of functions. Furthermore, it would allow an application to execute in a sequential manner in case OpenMP is not enabled at compile time.

On the other hand, there are well known idioms for hiding OpenMP runtime functions. For example, in Fortran, one may hide arbitrary code in *conditional compilation* lines behind a *sentinel*, such as `!$`. Similarly, C/C++ codes may check to see if `_OPENMP` is defined.

Additionally, the syntax of the directive based parallel I/O operations are implicitly assumed to behave similar to their sequential counterparts. This poses the challenge of having to first identify which functions to support, e.g. C I/O style `fread/fwrite` operations vs. POSIX I/O style `read/write` operations vs. `fprintf/fscanf` style routines. Furthermore, due to the fact that OpenMP also

supports Fortran and C++ applications, one would have to worry about the different guarantees given by POSIX style I/O operations vs. the record-based Fortran I/O routines or how to deal with C++ streams. Because of the challenges associated with the latter aspects to a parallel I/O library, an entirely new set of library functions has been defined.

3.2.2 Individual vs. Shared file pointers

The notion of parallel I/O implies that multiple processes or threads are performing I/O operations simultaneously. A preliminary question when designing the interfaces is whether to allow each thread to operate on a separate file pointer, or whether a file pointer is shared across all threads. Due to the single address space that the OpenMP programming model is based on, shared file pointers seem to be the intuitive solution to adapt. Note, that the overall goal is that all threads are able to execute I/O operations on the shared file handle without having to protect access to this handle.

3.2.3 Collective vs. Individual interfaces

A follow-up question to the discussion on individual vs. shared file pointers is whether threads are allowed to call I/O operations independent of each other or whether there is some form of restriction on how threads can utilize the new I/O functions. Specifically, the question is whether to use collective I/O operations, which require all threads in a parallel region to call the I/O operations, or whether to allow each thread to execute I/O operations independent of each other. Although collective I/O operations sound initially very restrictive, there are two very good reasons why

to use them. First, collaboration among the threads is a key design element to improve the performance of I/O operations. The availability of multiple (application level) threads to optimize I/O operations is only guaranteed for collective interfaces. Second, individual file I/O operations could in theory be implemented on a user level by opening the file multiple times and using explicit offsets into the file when accessing the data. Therefore, collective I/O interfaces have been supported in the current specification.

Using collective I/O interfaces requires a specification of the order by which the different threads access the data. The current specification read/writes data in the order of the thread-id's. However, relying on a thread's id is not a robust method of coordinating file operations implicitly among threads. The official OpenMP specification makes it clear that relying on thread id order for things such as predetermining the work a thread gets from a worksharing construct is at best benignly non-conforming (as in the case of a *static* schedule used by a parallel loop). Furthermore, in the cases of nested parallelism, each nested *parallel* region encountered by a thread creates a new thread team of the specified number of threads; within this new subteam, thread ids are assigned starting at 0. Therefore, in order to get a truly unique thread id that may then be used to provide a true total order over all threads in a nested situation, one must know the local thread ids of all ancestor threads[1].

Despite this fact, implicit ordering among threads on the total thread id order has been chosen due to the lack of useful alternatives. If the order of data items can be determined using a different mechanism in an application, interfaces that allow each thread to specify the exact location of the data item in the file are also provided.

[1]OpenMP provides for runtime functions to determine this

Our future work exploring parallel file I/O in OpenMP will consider it in the context of nested parallelism and explicit tasks, particularly as the latter continues to evolve and mature.

3.2.4 Synchronous vs. Asynchronous interfaces

Synchronous I/O interfaces block the execution of the according function to the point that it is safe for the application to modify the input buffer of a write operation, or the data of a read operation is fully available. Asynchronous interfaces on the other hand only initiate the execution of the according operation, and the application has to use additional functions to verify whether the actual read/write operations have finished. On the operating system level, the `aio_read/aio_write` functions provide examples for asynchronous I/O operations. These functions however also highlight the implementation challenges of these operations, since they very often imply spawning of additional threads in order to execute the according read/write operations in the background. In a multi-threaded programming model, where the user often carefully hand-tunes the number of threads executing on a particular processor; creating additional threads in the background can have unintended side affects that could influence the performance negatively. For this reason, the initial version of the OpenMP I/O routines only support synchronous I/O operations. However, this might change in the near future, specifically with the notion of asynchronous tasks gaining popularity in OpenMP.

3.2.5 Algorithmic vs. List I/O interfaces

A general rule of I/O operations is, that the more data an I/O function has to deal with, the larger the number of optimizations that can be applied to it. Ideally, this would consist of a single, large, contiguous amount of data that has to be written to or read from disk. In reality however, the elements that an application has to access are often not consistent neither in the main memory nor on the disk. Consider for example unstructured computational fluid dynamics (CFD) applications, where each element of the computational mesh is stored in a linked list. The linked list is in that context necessary, since neighborhood conditions among the elements are irregular (e.g. a cell might have more than one neighbor in a direction), and might change over the lifetime of an application. The question therefore is how to allow an application to pass more than one element to an I/O operation, each element pointing to potentially a different memory location and being of different size.

Two solutions are usually considered: an algorithmic interface, which allows one to easily express repetitive and regular access patterns, or list I/O interfaces, which simply take a list of *<input buffer pointers, data length>* as arguments. Due to the fact that OpenMP does not have a mechanism on how to express/store repetitive patterns in memory (unlike e.g. MPI using its derived data types), supporting algorithmic interfaces would lead to an explosion in the size of the interfaces that would be cumbersome for the end-user. Therefore, list I/O interfaces have been supported in the current specification, but not algorithmic interfaces. We might revisit this section however, since *Array shaping*[39] is being discussed under the context of OpenMP accelerator support.

3.2.6 Error Handling

As of today, OpenMP does not make any official statements or has any constructs to recognize hardware or software failures at runtime; though there is active investigation of this topic by an OpenMP ARB subcommittee. Dealing with some form of failures is however mandatory for I/O operations. Consider for example that a common scenario in reading data from a file is to continue reading until the end-file-marker(EOF) has been returned by the corresponding I/O function. Similarly, recognizing when a write operation fails, e.g. because of quota limitations, is paramount for many applications. Therefore, all I/O routines return an error code. The value returned is either *0* in case of success, or *-1* in case the routine has encountered a problem. The model can be refined by more precise error codes in subsequent versions.

3.2.7 Introduction to the annotation used

In the following, we present the C versions of the parallel I/O functions introduced. Since all functions presented here are collective operations, i.e. all threads of a parallel region have to call the according function, some input arguments can be either identical or different on each thread. Furthermore, the arguments can be either shared variables or private variables. For convenience, we introduce the following annotation to classify arguments of the functions:

- *[private]:* The argument is expected to be a private variable of a thread, values between the threads can differ.

- *[private']:* Argument is expected to be different on each thread. This can be either achieved by using private variables, or by pointing to different parts of a shared data structure/array.

- *[shared]:* The argument is expected to be a shared variable.

- *[shared']:* An argument marked as *shared'* is expected to have exactly the same value on all threads in the team. This can be either accomplished by using a shared variable, or by using private variables having however exactly the same value/content.

3.2.8 File management functions

As of now, this category consists of two routines to collectively open and close a file. All threads in a parallel region should input the same file name when opening a file. The *flags* argument controls how the file is to be opened, e.g. for reading, writing, etc.. It is a bit mask whose value can be set using the bitwise OR operation of the appropriate parameters (using the | operator in C). The returned file descriptor *fd* is a shared variable. Note, that it is recommended to use as many threads for opening the file as will be later on used for the according read-write operation. However, a mismatch in the number of threads used for opening vs. file access is allowed, specifically, it is allowed to open the file outside of a parallel region and use the resulting file handle inside of a parallel region. Having the same number of threads when opening the file as in the actual collective read-write operation could however have performance benefits due to the ability of the library to correctly set-up and initialize internal data structures.

Note also, that a file handle opened using *omp_file_open_all* can not be used for sequential POSIX read/write operations, and vice versa.

3.2.9 Different Argument Interfaces

The routines specified in this section assume that each thread in a collective read/write operation passes different arguments, except for the file handle. Specifically, each thread is allowed to pass a different buffer pointer and different length of data to be written or read. This allows, for example, each thread to write data structures that are stored as private variables into a file.

In the explicit offset interfaces, i.e. interfaces that have the keyword *at* in their name, each thread should provide the offset into file where to read data from or write data to. If two or more threads point to the same location in the file through the according offsets, the outcome of a write operation is undefined, i.e. either the data of one or the other thread could be in the file, and potentially even a mixture of both. For read operations, overlapping offsets are not erroneous.

For implicit offset interfaces, data will be read or written starting from the position where the current file pointer is positioned. Data will be read from the file in the order of the threads' OpenMP assigned IDs.

All functions also take an argument referred to as *hint*. A hint is an integer value that indicates whether buffer pointers provided by different threads are contiguous in memory and file, or not. The according constants are *CONTIG, NONCONTIG, NULL*. The latter constant indicates that user does not know whether data are contiguous or not. All threads must pass the same value for the hint. Providing an

indication that data are contiguous across threads could lead to performance benefits since it allows the parallel I/O library to skip certain code steps internally.

The List I/O interfaces, i.e. the functions taking more than one pair of < *buffer pointer, length*> arguments, have an addition keyword *list* in the name. For example, the collective file read function with list input is *omp_file_read_list_all*, data will be read from file in order of the threads' OpenMP assigned IDs starting from the current position of the file pointer. *buffer* is in this case a pointer to an array of *iovec* structures, which contains an argument for a buffer pointer and a data length. The *offsets* argument is a pointer to an array of offsets. The length of both arrays is defined by the argument *size*.

3.2.10 Common Argument Interfaces

The interfaces discussed in this subsection define functions where each thread has to pass exactly the same arguments to the function calls. The main benefits from the perspective of the parallel I/O library is that the library has access to multiple threads for executing the I/O operations. Thus, it does not have to spawn its own threads, which might under certain circumstances interfere with the application level threads.

Both, single argument and list I/O versions of all read and write operations have been defined in this section as well, for both implicit and explicit offset operations.

Table 3.1 lists the functions described above. Only read functions are shown in the table for the sake of brevity.

File Management Interfaces
*int omp_file_open_all([shared] int *fd, [shared'] char *filename, [shared'] int flags)*
int omp_file_close_all([shared] int fd)

Different Argument Interfaces
int omp_file_read_all ([private']void buffer, long length, [shared]int fd, [shared']int hint)*
int omp_file_read_at_all ([private']void buffer, long length, [private']off_t offset, [shared]int fd, [shared']int hint)*
*int omp_file_read_list_all ([private']void** buffer, int size, [shared]int fd, [shared']int hint)*
*int omp_file_read_list_at_all ([private']void** buffer, [private']off_t* offsets, int size, [shared]int fd, [shared']int hint)*

Common Argument Interfaces
int omp_file_read_com_all ([shared]void buffer, [shared']long length, [shared]int fd, [shared']int hint)*
int omp_file_read_com_at_all ([shared]void buffer, [shared']long length, [shared']off_t offset, [shared]int fd, [shared']int hint)*
*int omp_file_read_com_list_all ([shared]void** buffer, [shared']int size, [shared]int fd, [shared']int hint)*
*int omp_file_read_com_list_at_all ([shared]void** buffer, [shared']off_t* offsets, [shared']int size, [shared]int fd, [shared']int hint)*

Table 3.1: OpenMP I/O general file manipulation routines

3.3 Implementation

This section presents a prototype implementation of the interfaces in the OpenUH compiler. Internally, the parallel I/O operations are based on a POSIX threads based I/O library. In the following, we discuss first the OpenUH compiler, I/O library as well as the integration of the two components.

3.3.1 The OpenUH compiler and its runtime

OpenUH [40, 41], a branch of the Open64 4.x compiler suite, is a very high quality, fully functional C/C++ and Fortran optimizing compiler under development at the University of Houston that supports the bulk of the OpenMP 3.0 API, including explicit tasking. It is freely available and used primarily as a basis upon which language extensions (e.g., Co-array Fortran [42]) and new ideas for better supporting OpenMP during both the compilation and runtime phases are explored. OpenUH was first realized when Open64 was extended to support OpenMP 2.5 [41]; and from this point, it has been used as a platform for conducting compiler research.

Examples include selective performance instrumentation [43] of a source program during the different phases of the compilation and the only known open source implementation of the OpenMP ARB sanctioned Collector API [44]. The Collector API is an event-based framework used to enable the creation of "collector tools" - custom written dynamic libraries that may be registered and called during specified events triggered by the OpenMP runtime during program execution. In addition to supporting most of the OpenMP 3.0 API, novel extensions have been designed and implemented to provide greater scalability when mapping work to many cores [45]. Barrier, serialization and reduction enhancements have been implemented [46] and allow the user to employ these enhancements at runtime. Creating compiler-based tools is also a main use of OpenUH [47], and research into providing a more flexible basis for doing this and for enabling better compiler/tool interactions is being heavily pursued. Other work includes the implementation of full OpenMP 3.0 compliance and the full support of nested parallel sections.

OpenUH supports OpenMP through two means. The first is during the compilation of the source program containing OpenMP directives. When the program is first compiled, it is put into the very highest level of the Open64's intermediate representation, WHIRL. The WHIRL nodes in the IR tree representing OpenMP regions are first *lowered* into a type of WHIRL node supporting more generic multi-processor, or "MP" program units. Over the course of various optimization and IR lowering phases, these MP program units are transformed into explicitly multi-threaded code, aided by the insertion of internal runtime functions available to the compiler through the OpenMP runtime library. For example, the beginning of a parallel region, denoted in C with the *#pragma omp parallel* directive, corresponds directly with a call to *__ompc_fork*, which manages the creation or waking of threads during program execution. Included in the call to this function is a function pointer to the code contained inside of this parallel region, which at this point has itself been turned into a series of functions through a compiler technique called *inlining* [41]. There are similar runtime functions that make the transformation into explicitly threaded code and dynamic management of work easier, such as those associated with worksharing, synchronization and barriers, cache flushing, atomic writes to shared variables, mutual exclusion, task creation, and the facilitation of private and shared data environments.

The object code resulting from source that contains OpenMP includes with it, the framework necessary to direct the execution of the multi-threaded program as described through the use of the very high level OpenMP directives and the external facing runtime library functions which are designed to be used directly by the user

in their code; thus OpenMP is not only a set of directives that the compiler may use for transformations during compilation, but it's a set of runtime functions the user may directly call to explicitly query and modify the execution environment. An example of an external interface is, *omp_get_thread_num*, which returns the unique thread id of the thread in a thread team making this call. Below, the integration of the the parallel I/O library with the OpenUH compiler is discussed.

3.3.2 A multi-threaded parallel I/O library

The parallel I/O library used internally provides collective I/O operations based on POSIX threads. The architecture of the library consists of the following main components as shown in Figure 3.1. In the following, each component has been discussed in detail.

- Initializer:

  The initializer routines are called when threads open a file. The *init* routines initialize various data structures required by the library on a per-file level. Some important tasks of the *init* routines include opening the file multiple times and to internally maintain a list of file descriptors to the file. Note that only a single file descriptor is returned to the calling program. POSIX mutexes and condition variables are declared and initialized for the threads operating on the file.

- Base Function:

  The base function is the first internal library function called by the collective I/O interfaces. It collects the input arguments provided by all threads in a

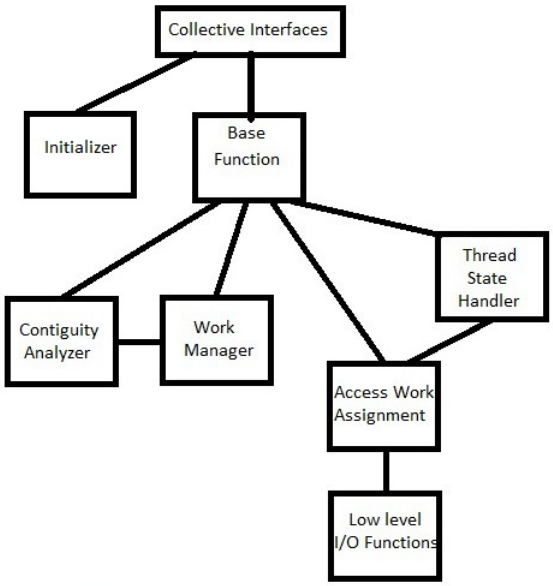

Figure 3.1: Architecture of the I/O Library.

single array. It then redirects all but one thread, the slave threads, to the wait state, while the master thread analyzes the input arguments and redirects control to the contiguity analyzer or the work manager accordingly. Finally, the master thread proceeds towards executing an I/O task, if necessary.

- Contiguity Analyzer:

  The contiguity analyzer is called by the main thread. It performs the optimization of merging buffers by scanning the input array of memory addresses to look for contiguity between them. For those interfaces that accept file offsets explicitly, the analyzer looks for contiguity both in memory as well as in file.

  If the analyzer finds discontiguity between buffers, it passes the contiguous block found so far to the *work manager* and proceeds with the scan on the rest

of the array. Once the entire array has been scanned, it sets a *FINISH* flag, and the remaining block of contiguous memory addresses is passed to the work manager.

- Work Manager:

  The work manager performs the task of assigning blocks of data to be read/written to threads. Once it accepts a contiguous block of data from the contiguity analyzer (or from the base function), it assigns the block to the next available slave thread and sets the *ASSIGNED* flag for the thread. It also manages the internal file offset used for those interfaces that do not accept a file offset explicitly. The work manager can be programmed to wake up a thread immediately once an I/O request is assigned to it or wake up all threads once the contiguity analyzer completes its analysis and the *FINISH* flag is set. Currently, for list I/O interfaces, the work manager waits for the *FINISH* flag to be set whereas for the single argument interfaces, it wakes up a slave thread immediately.

- Thread-State Handler:

  Slave threads enter a wait state after they call the base function. They sleep on a condition variable until they are explicitly invoked by the master thread (through the work manager). Once a slave thread is invoked from its sleep mode, it checks to see if it has been assigned an I/O request. All slave threads that have been assigned an I/O request proceed to access the low-level I/O interfaces while those that do not have an I/O assignment exit the function.

- Access Work Assignment:

This module allows a thread to configure some data structures before proceeding to the low level IO functions. As an example, a thread with multiple I/O assignments can create an array of *struct iovec* to enable performing list I/O.

- Low-Level I/O Interfaces:

  The low level interfaces list the functions available to a thread for performing I/O. As an example, for a thread with multiple I/O assignments, it creates an array of *struct iovec* and calls the *readv / writev* routines. The *pread* and *pwrite* functions allow the programmer to specify a file offset along with a memory address and the length of the data to be read/written.

3.3.3 Integration of I/O library with the OpenUH compiler

Since the collective I/O interfaces were originally developed as part of a stand-alone library for POSIX threads, integration of the library with the compiler and providing the OpenMP syntax discussed previously required some modification. For example, data structures were originally introduced to map POSIX IDs to OpenMP style IDs (0,1,2..). Further, private routines were written to mimic the behavior of OpenMP runtime functions like *omp_get_thread_num()* to query the data structures holding this information. These routines were not required anymore after the integration with the compiler.

The main bulk of the integration work was to take advantage of the functionality of the compiler's OpenMP runtime within the parallel I/O library. This includes using the runtime's functionality to determine the number of threads in a parallel region, thread ID's etc.. Furthermore, the parallel I/O library has been modified

to take advantage of the highly optimized synchronization routines among threads instead of the original implementation in the parallel I/O library.

In the following, we discuss two aspects of the library which have a strong influence on the performance in more details.

3.3.4 Optimizing Collective I/O operations

As discussed previously, collaboration among the threads is a key design element to improve the performance of I/O operations. Collective I/O interfaces provide an abstraction from the actual implementation of the operation, allowing to use multiple approaches/algorithms without requiring any modification in the user code. The actual approach taken to implement a collective read or write operation will depend on the input parameters of the operation, i.e. amount of data to be read/written by each thread and offset into the file, as well as characteristics of the underlying file system and storage.

The implementation used in the library presented here is based on two fundamental concepts. First, there is a minimal data size s_{min} required to saturate an individual file stream. For example, if multiple threads request to write a small amount of data each, it might be beneficial to combine the items of all threads and issue a single, larger write request. This approach reduces the number of context switches and improves the overall performance of the I/O operations [48].

Secondly, one data stream might not be able to saturate the read/write bandwidth of a storage system. The consequence of this assumption is that a very large I/O request might need to be split into multiple smaller requests. This distributes the

work among multiple threads, and generates multiple, parallel data streams to/from the storage, which often improves the performance again. The precise meaning of 'large' data blocks depends on various factors, mainly the underlying parallel file system being used.

The library further provides the notion of "active threads", which is the number of threads that participate in I/O. A user can set the number of active threads in a config file. This feature can be used so that only a subset of threads participates in I/O, whereas remaining threads can proceed with program execution. When the number of active threads is less than the total number of threads available, the first *active_threads* number of threads are chosen for I/O based on their OpenMP IDs. As an example, if an OpenMP program spawns eight threads, out of which two are set to be active, threads 0 and 1 will be chosen as active threads. The advantages of having active threads are two-fold: depending on the number of cores, external workload, network interconnect etc., not all threads may be required to obtain the best I/O performance; a smaller number of threads participating in I/O may prove to be sufficient. In fact, having all threads participate in I/O could have a negative effect on the I/O performance. Active threads allow us to overlap I/O with computation, thereby accomplishing the main advantage provided by asynchronous functions.

To determine the minimal data size s_{min} as well as a reasonable number for the active threads, two simple benchmarks are used. The first one writes various data sizes repeatedly (ensuring that no caching effects occur). Plotting the bandwidth obtained over the data size used allows to determine the minimal data saturation

point. A second benchmark is used to repeatedly write data of size s_{min} with increasing number of threads. The combined bandwidth of all threads writing data is then used to determine the ideal number of active threads, which we define as the minimal number of threads obtaining the maximum bandwidth observed.

As of today, a user can set these values in a configuration file that is read by the library whenever a file is opened. Algorithms which allow to determine (close to) optimal values for these two parameters are currently being discussed and evaluated.

3.3.5 Alternative low-level interface functions

The goal of the parallel OpenMP I/O interfaces is to allow a user to perform I/O to a common file in an efficient manner without having to manage and lock a (shared) file handle. However, different file systems react differently when multiple threads perform I/O to a shared file. PVFS2 allows for efficient I/O when multiple threads access a common file, whereas Lustre does not scale at all when multiple threads access the same file [49]. In this case, we perform an optimization at an intermediate level to extract better I/O bandwidth from the underlying file system.

The Parallel Log-structured File System (PLFS) is a virtual file system developed by Los Alamos National Laboratory [50]. It remaps the data layout of an application into one optimized for the underlying file system. Specifically, it transparently rearranges the pattern where N data streams are targeting one file into a pattern where each data stream accesses a separate file. Thus, the application still has a single logical file that it can operate and access, however from the underlying file systems perspective there are multiple files. This prevents contention when trying to

lock a file for write operations [51] to ensure consistency, and thus greatly improves performance without sacrificing usability on some file systems such as Lustre.

PLFS is a virtual file system situated between the parallel application and an underlying parallel file system responsible for the actual data storage. PLFS creates a container structure on the underlying parallel file system. Internally, the basic structure of a container is a hierarchical directory tree consisting of a single top-level directory and multiple sub-directories. Multiple threads opening the same logical file for writing share the container, although each thread gets a unique data file within the container into which all of its writes are appended. When the thread writes to the file, the write is appended to its data file and a record identifying the write is appended to an index file. A file created using the PLFS API can either be read by regular POSIX I/O routines in case the FUSE daemon has been installed, or using the PLFS API otherwise.

PLFS has been incorporated into the parallel I/O library as an alternative implementation of the *low level interfaces* described above. Thus, the library has the ability to choose between the PLFS read/write functions and the POSIX I/O interfaces depending on the file system being used. Currently we allow the user to choose the PLFS API instead of the regular low level I/O routines by specifying it in the config file. In the future, the library will be able to detect the type of the underlying file system and choose the appropriate *low level interface* functions automatically.

3.4 Evaluation

The performance of the new interfaces and the according implementation is evaluated with a set of micro-benchmarks on various platforms, storage systems and file systems.

3.4.1 Description of Benchmarks

As part of this dissertation, we developed a set of micro-benchmarks that provide commonly used I/O patterns in OpenMP applications and/or options to express I/O patterns in OpenMP applications.

Writing/Reading in parallel to/from one file using the ordered directive In this benchmark threads perform I/O to the same file. Since the file descriptor is shared between all threads, access to it needs to be exclusive to a thread at any given instance of time for file is opened for writing. This restriction is not implemented for read tests. Threads access non-overlapping parts of a large shared buffer in a *ordered for* loop. Note that access to the file descriptor can also be protected using OpenMP's *critical* section. The *ordered* clause was not necessarily used to achieve ordering between the threads. For write purposes, this test exposes the performance drawback that can be seen when access to a shared file needs to be exclusive. As such, this is a worst case scenario when threads write to the same file. For reading, this test highlights the effects of multiple threads accessing the same file (no explicit locking or synchronization).

Writing/Reading in parallel to/from separate files Here, all threads perform reads and writes to separate, individual files. Each thread has exclusive access to its own file and can perform I/O freely, without contention with other threads. Access to each file descriptor does not need to be protected via locks for writing. Threads read/write non-overlapping parts of a large, shared matrix in a loop. This benchmark targets exploring the maximum I/O bandwidth available to the application.

Collective I/O using *omp_file_write_all*/*omp_file_read_all* This benchmark aims to evaluate the collective interface *omp_file_write_all*/*omp_file_read_all*. The file is opened using *omp_file_open_all*. Threads read/write non-overlapping parts of a large, shared matrix from/to a common, shared file. For write tests, the shared matrix is ultimately written multiple times using a *for* loop to achieve the desired file size. Note that access to the open file does not require synchronization between threads.

Application Benchmarks Further, we evaluate the performance of our interfaces using a version of the NAS parallel benchmarks and an image processing application.

BT I/O: The new OpenMP I/O routines have also been evaluated with two OpenMP applications. We present here results obtained with the Block-Tridiagonal (BT) NPB benchmark [52], which has in its MPI version an I/O performance component. An OpenMP version of the BT benchmark is available since version 3 of NPB, however without the I/O part. We extended the NPB OpenMP BT benchmark to include I/O in a way similar to its MPI-IO implementation. Note, that subtle differences remain between the two implementations. NPB-MPI writes a slightly lesser

amount of data and reads it back for verification.

Experiments have been performed with the class D benchmark, where approximately 128 Gigabytes of data are written over the course of the program (approximately 2.5GB of data over 50 iterations).

MSG: Multiscale Gabor (MSG) is an image processing application used to analyze smear sample from fine needle aspiration cytology, with the overall goal being to assist medical doctors in identifying cancer cells [53]. The challenge imposed by this application is due to the high resolution of the microscopes and the fact that images are captured at various wave-length to identify different chemical properties of the cells. For a $1cm \times 1cm$ sample with 31 spectral channels the image can contain overall up to 50GB of raw data. Furthermore, the code has the option to write the texture data into output files to facilitate future processing steps in realizing a complete computer aided diagnosis (CAD) solution. This makes the application compute and I/O intensive.

We performed experiments with an image containing 8192 x 8192 pixels and 21 spectral channels. The application writes 13 files, each of size 256 MB, hence a total of 3.25 GB.

3.4.2 Description of the platforms used

For the experiments, two PVFS2 (v2.8.2) [18] file system installations and a Lustre file system [20] were used.

PVFS2 over the Crill Compute Cluster (PVFS2): The Crill compute cluster consists of 16 nodes, each node having four 2.2 GHz 12-core AMD Opteron processors

(48 cores total) with 64 GB main memory. Each node has three 4x SDR InfiniBand links, one of them being reserved for I/O operations and two for message passing communication. A PVFS2 file system has been configured over the Crill nodes such that all 16 Crill nodes act as PVFS2 servers and clients, and a secondary hard drive on each node is used for data storage.

PVFS2 over SSD (PVFS2-SSD): Apart from regular hard drives, the Crill cluster has a RAMSAN-630 Solid State Disks (SSD) based storage from Texas Memory Systems. This SSD is made of NAND based enterprise grade Single Level Cell (SLC) flash. The SSD installation has four 500GB cards, thus making a total of 2TB. It has two dual port QDR Infiniband cards, and we use one of two ports on each card. The peak I/O bandwidth of the SSD storage is 2 GB/s. The PVFS2 parallel file system configured over the SSD employs two separate I/O servers, each I/O server serving exactly half of the SSD storage.

Lustre: The *Atlas* cluster at the University of Dresden employs nodes with a maximum of 64 cores. The file system is connected to the compute nodes using an SDR Infiniband link. The cluster has 92 AMD Opteron nodes with 64 cores each and 64 to 512 GB memory. The storage consists of 12 OSTs and the stripe size was set to 1MB. The Lustre version used is 1.8.

Tests have been executed multiple times. To avoid effects of data caching, all data are flushed to disk using *fsync* before closing a file. The maximum of the bandwidth values observed across all runs has been presented. The variance of measurements on these two file systems was generally low.

3.4.3 Results

Results using the Microbenchmarks

First, the results of the first two micro-benchmarks described above are shown. These benchmarks allow us to set upper and lower bounds for the expected performance of the collective OpenMP I/O interfaces.

Fig. 3.2 shows the performance of the write benchmark on the PVFS2 file system. The left part of the figure shows the write bandwidth obtained on the PVFS2 file system when threads perform I/O to a shared file. The I/O bandwidth observed reaches a maximum of 212 MBytes/sec, independent of the number of threads used. This can be explained by the fact that the benchmark serializes the access to the file handle and therefore the I/O operation itself. The right part of the figure shows the results obtained with the second micro-benchmark where threads write to individual files. The bandwidth obtained in this case is significantly higher than when threads write to a shared file, reaching a maximum of almost 500 MBytes/sec. This value is an indication of the upper bound on the I/O performance that can be achieved from a single node on this machine.

As discussed before, most scientific applications require a separate merge step to consolidate data from separate files into a single file. This is done in a sequential manner and causes the overall I/O performance to drop significantly. As an example, we see a maximum bandwidth of 496 Mbytes/sec when eight threads write separate files of 2 Gigabytes each on PVFS2. However, on merging all those files into a single 16 Gigabyte file, the sustained bandwidth drops to 101 Mbytes/sec.

Results of the microbenchmarks on the PVFS2-SSD (see Figure 3.3) and the

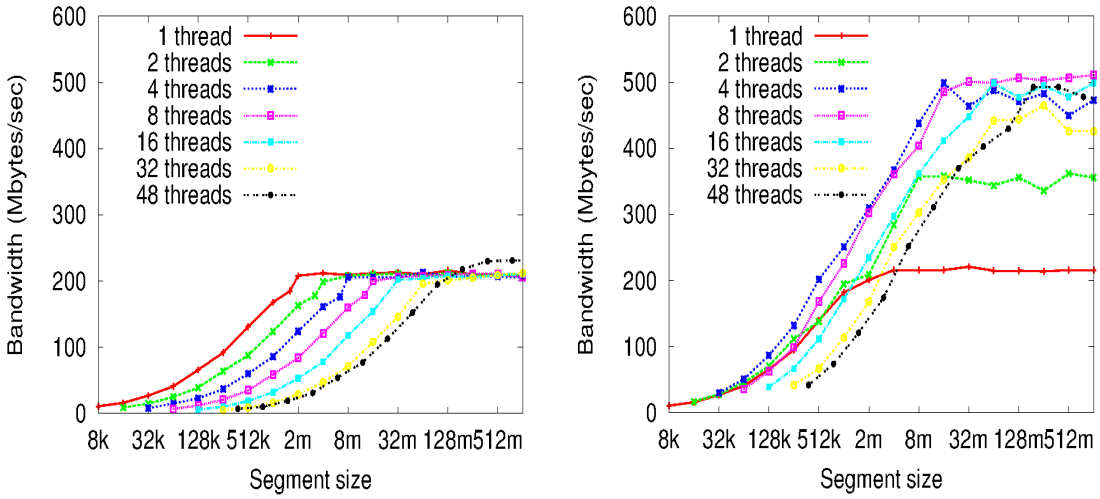

Figure 3.2: Shared file (left) and individual file write microbenchmark(right) on PVFS2.

Lustre file systems (Figure 3.4) show a similar trend, where we see performance limitations when threads write to a shared file, but higher bandwidth when they write to individual files.

The read performance of these benchmarks is discussed below. Fig. 3.5 shows the effect of multiple threads reading from a common/separate files. The figure shows that there is virtually no difference between the two cases on the PVFS2 file system. Note that for reading purposes, a file is not locked by a thread. On PVFS2, there are no real benefits of reading from multiple files as opposed to reading from a single file.

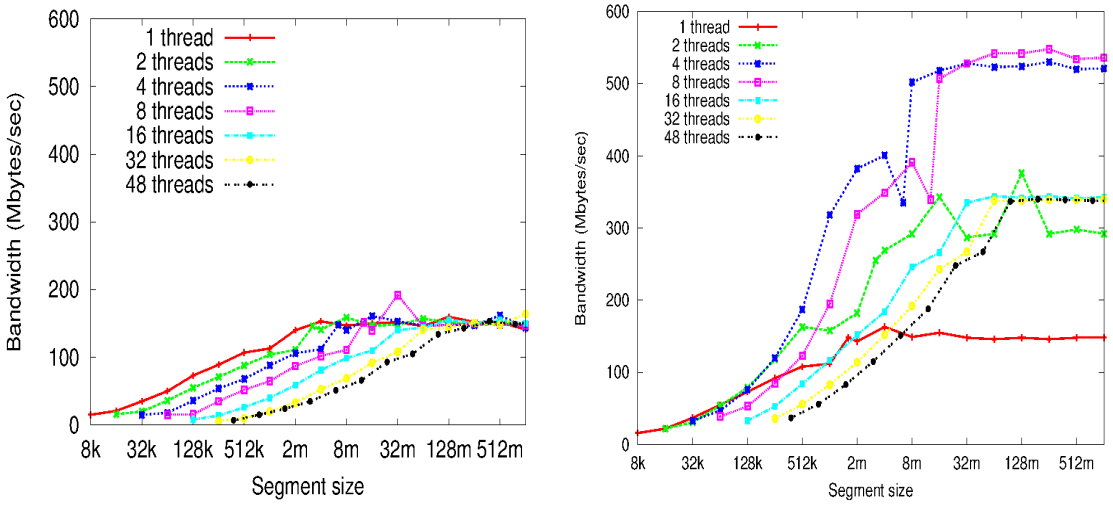

Figure 3.3: Shared file (left) and separate file (right) write microbenchmark on PVFS2-SSD.

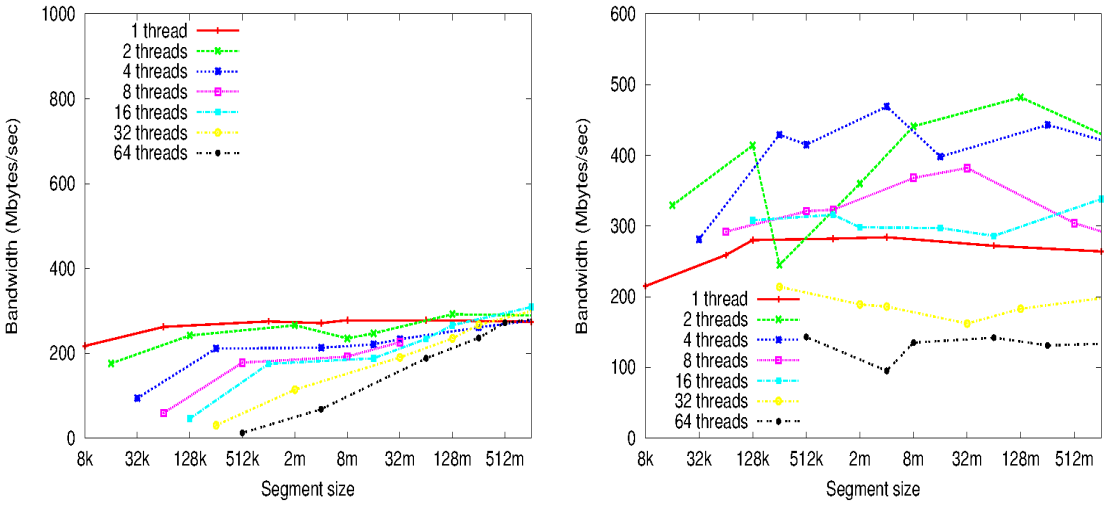

Figure 3.4: Shared file (left) and separate file (right) write microbenchmark on Lustre.

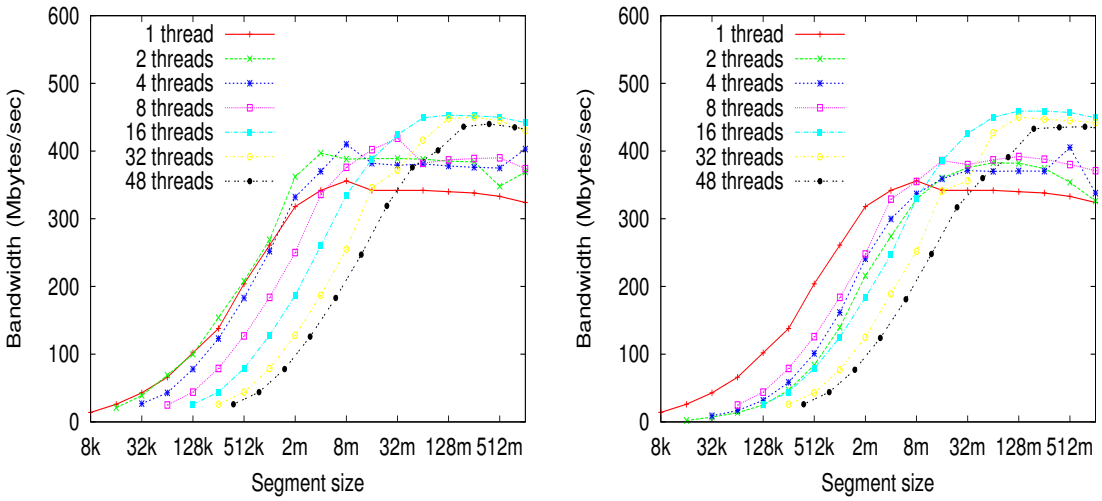

Figure 3.5: Shared file (left) and separate file (right) read microbenchmark on PVFS2.

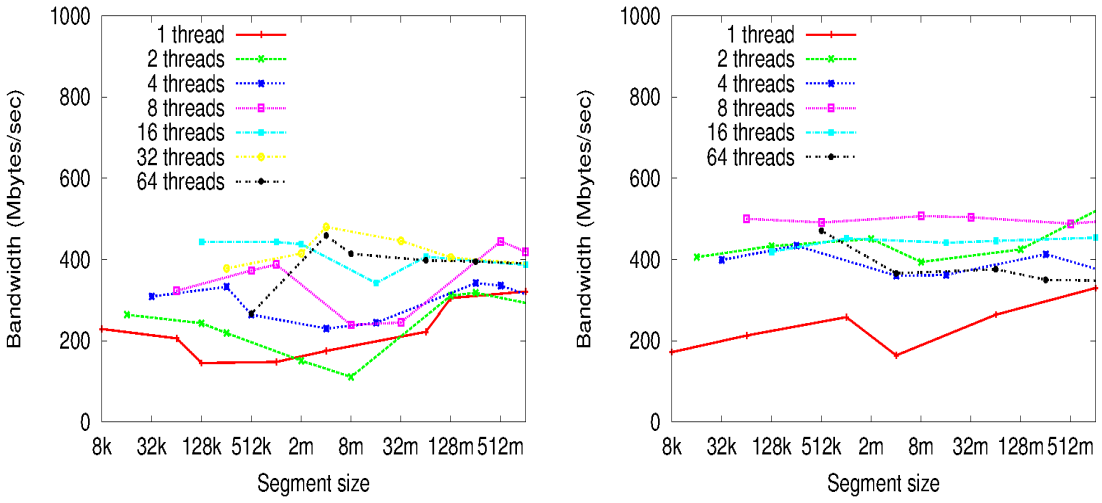

Figure 3.6: Shared file (left) and separate file (right) read microbenchmark on Lustre.

Fig. 3.6 shows the read performance on the Lustre file system. In contrast with PVFS2, accessing the same file on Lustre results in limited performance whereas reading from separate files shows improvements in bandwidth, specially for large number of threads.

3.4.4 Determining s_{min} and Active Threads

As discussed previously, the size of a data chunk at which the best performance for a single thread is achieved is called s_{min}. Similarly, active threads denote the minimum number of threads required to obtain the optimum bandwidth from a node. These values are determined by a brute force method, where the value of s_{min} is determined by testing the I/O performance for one thread using various data sizes. We start with a size of 8KB and increment it in multiples of 2, upto a maximum size of 1GB. Similarly, these tests are performed for different number of threads to determine the number of active threads. Figure 3.7 shows these values for the PVFS2 file systems. It can be seen that the s_{min} value is 2MB and the number of active threads is 8.

Similarly, Figure 3.8 shows the values for the Lustre file system. The graph on the left shows the test performed to determine s_{min}, whereas on the right is the test for active threads. It can be seen that although the s_{min} value here is 1MB, the performance for smaller data sizes like 8KB is reasonably high. On Lustre, active threads is a more important parameter. The active threads for Lustre as seen in the graph is 4. Although the performance obtained using two threads is similar, the performance using four threads is better when the data size equals s_{min} value. Also, it must be noted that the microbenchmarks perform the simple task of only writing

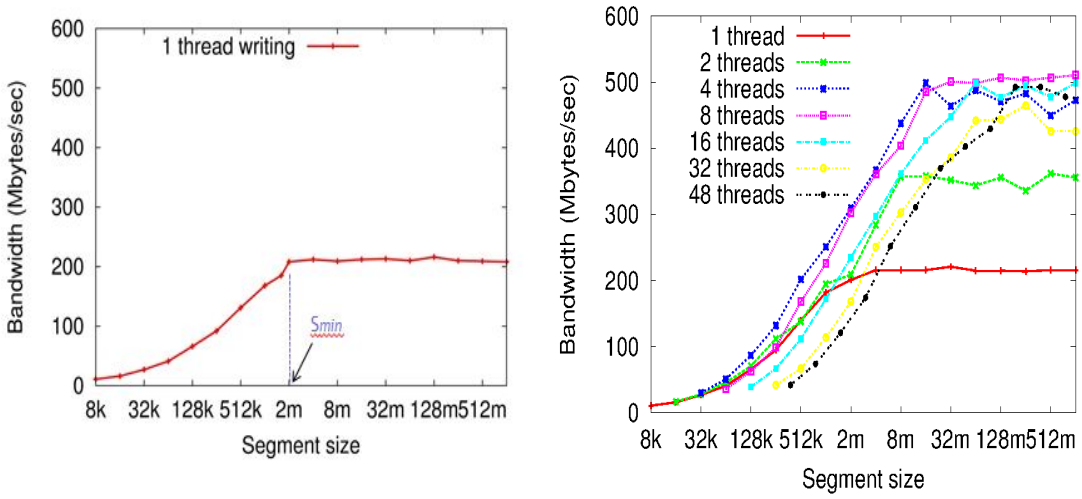

Figure 3.7: Determining s_{min} and active threads on PVFS2.

data, and hence exhibit a pattern different from real applications which may have a computational component, which could have an influence on the value observed for active threads. Also, since PLFS creates a file per thread, for an application that creates many output files, the overhead of creating files can also have an impact on the performance of I/O. There are subtle differences seen in the two graphs for the single thread case. This is mainly attributed to the fact that the file system is a shared resource.

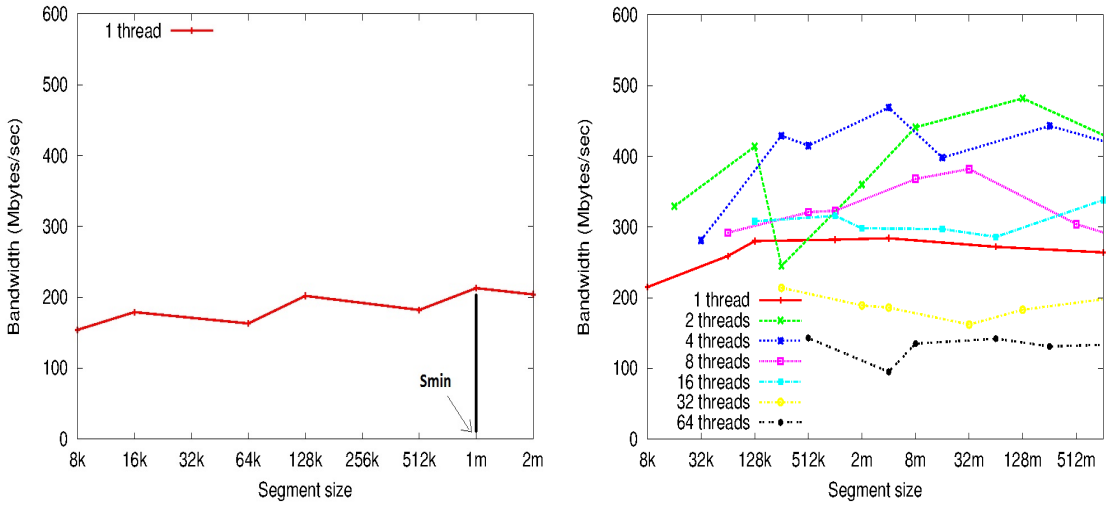

Figure 3.8: Determining s_{min} and active threads on Lustre.

Results using Library Interfaces

The left part of Figure 3.9 shows the performance of *omp_file_write_all* on PVFS2. Note that *omp_file_write_all* is a collective function where every thread provides a fixed amount of data and the data points shown on this graph (segment size) indicate the total amount of data written across the threads by each *omp_file_write_all* call. The results indicate that our implementation of the OpenMP I/O interfaces achieved a bandwidth in excess of 500 Mbytes/sec. Performance for 1, two threads reaches a maximum of 214 Mbytes/sec and 360 Mbytes/sec respectively, whereas it is much higher for a larger number of threads. The benefits of multiple threads performing I/O are clear in this case. For these tests, all threads were used as active threads.

It can also be seen that increasing the segment size, i.e. the amount of data written in a single function call, results in better performance. However, the bandwidth obtained does not necessarily increase beyond a certain threshold. For the PVFS2

file system, the main limitation comes from how fast data can be transferred out of the node, while for the SSD storage the limitation is sustained write bandwidth of the storage itself.

The right part of the figure shows the read performance on PVFS2. Similarly, Figure 3.10 shows the performance of the functions on PVFS2-SSD.

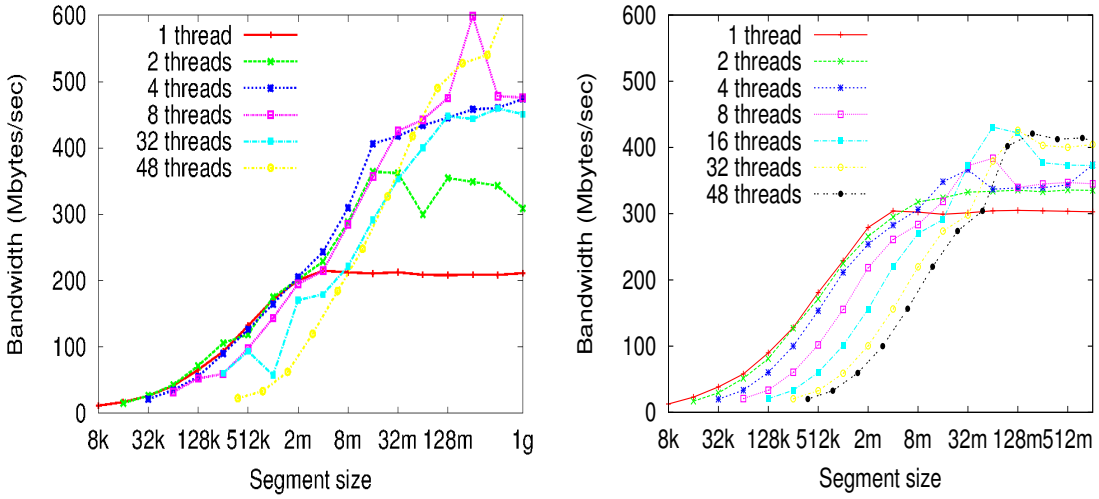

Figure 3.9: Performance of omp_file_write_all (left) and omp_file_read_all (right) on PVFS2.

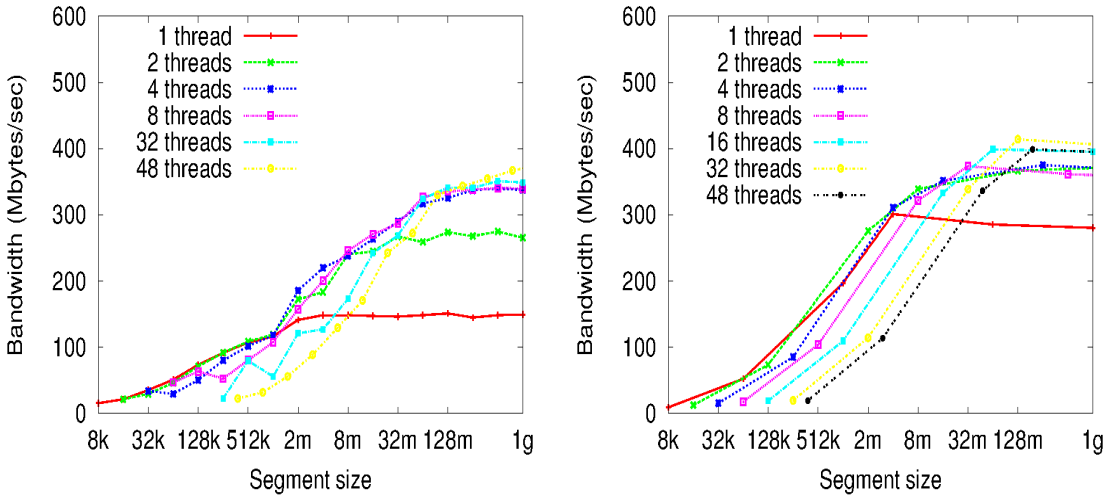

Figure 3.10: Performance of omp_file_write_all (left) and omp_file_read_all (right) on PVFS2-SSD.

Despite the fact that *omp_file_write_all* writes to a shared file, its performance is consistently better than when writing to a shared file using explicit serialization. The collective I/O routines perform typically close to the performance of the benchmark where threads write to separate files, which as discussed, often represents a best-case scenario. Furthermore, taking into account that the 'separate files' scenario would require an explicit merging step after executing the application, the new routines clearly represent the best of three solutions evaluated in the corresponding micro-benchmarks.

In Figure 3.11, the performance improvements using multiple threads are clearly seen. Note that by using PLFS, the library creates multiple files at the file system level to take advantage of Lustre's performance improvements demonstrated by the use of PLFS. The absolute value of the bandwidth is greater than that observed

using the microbenchmarks. This can be attributed to the fact that Lustre is a shared resource and the results obtained are subject to the load on the file system due to other applications accessing it.

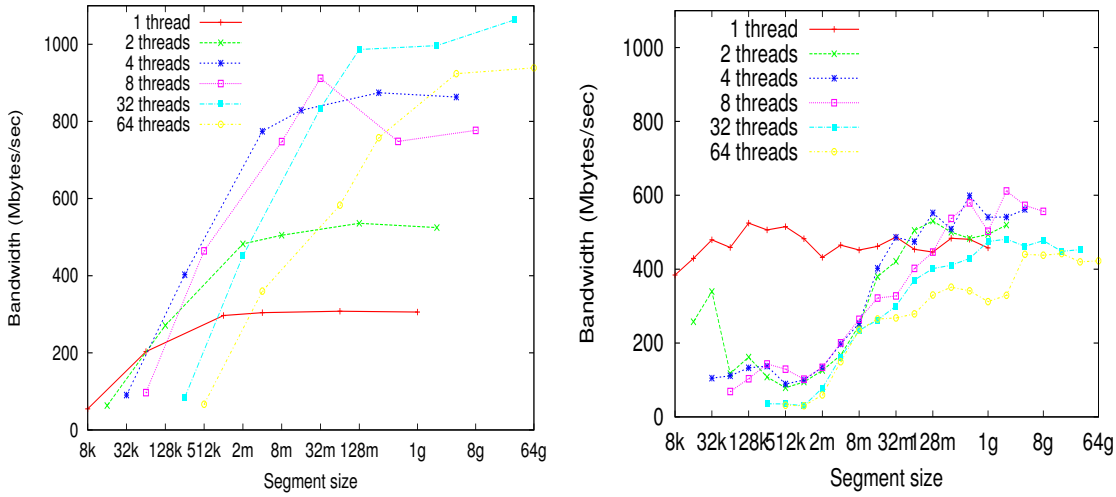

Figure 3.11: Performance of omp_file_write_all (left) and omp_file_read_all (right) on Lustre.

Performance of BTIO and MSG

Table 3.2 shows the write times for BTIO for all file systems. Note, that in these tests the number of active threads was set to eight and the s_{min} value used was set to 2 MB for PVFS2. For Lustre, the number of active threads was set to four and s_{min} was set to 1MB as determined previously.

The results show that with increasing number of threads but keeping the number of active threads constant, a consistent performance is achieved. We can see that using multiple threads significantly improves performance as compared to the

sequential case. The runtime for the single thread case was very high on the PVFS2-SSD and Lustre file systems. Due to limitations on the resource utilization, these tests have not been run.

No. of threads	PVFS2	PVFS2-SSD	Lustre
1	398	-	-
2	319	725	326
4	173	447	499
8	177	426	564
16	165	425	527
32	156	429	496
48	181	423	-
64	-	-	572

Table 3.2: BTIO results showing I/O times (seconds).

No. of threads	PVFS2	PVFS2-SSD	Lustre
1	12	17	15
2	6	17	8
4	5	13	12
8	5	12	12
16	6	12	11
32	5	12	10
48	7	18	–
64	–	–	13

Table 3.3: MSG write times (seconds).

A similar trend can be seen in the write times for MSG in Table 3.3. On PVFS2, the best performance is seen for two threads, since the maximum performance that can be obtained has already been achieved (for two threads, the write bandwidth for PVFS2 is 515 MBytes/sec).

No. of threads	Lustre
1	14.7
2	8.2
4	12.12
8	7.5
16	8.4
32	10.4
64	10.4

Table 3.4: MSG write times (seconds) on Lustre with 2 active threads.

On Lustre, the two threads case shows the best performance, and the I/O time seems to increase when more threads are spawned by the application. Table 3.4 shows the performance when two threads are used as active threads. The performance using two threads as active threads is better than when using four threads as active threads. This is because we use PLFS on Lustre, which creates a file for every active thread. Since MSG creates 13 output files in total, when four threads are used as active threads, a total of 13*4=52 files are created at the file system, whereas when two threads are used as active threads, a total of 26 files are created. Since these cases differ in the sense that the total number of files created at the file system level is different, it could be one of the reasons contributing towards the difference in the performance times observed.

Overall, we can see that for both BTIO and MSG, a slowdown in performance occurs when the application spawns as many threads as the number of cores in the system. In general, for the Magny-Cours and Interlagos processors, spawning a large number of threads on a node is likely to have performance drawbacks since it leaves the operating system with fewer system resources to utilize. Kerbyson et al., discuss

the performance characteristics of Magny-Cours processors in detail [54].

3.5 Comparing OpenMP-IO with MPI-IO

It is known that spawning threads consumes fewer system resources as compared to spawning processes. In this section, we present results that determine if having multiple threads has advantages over having multiple processes on a node. In this section, we compare the performance of the MPI and OpenMP versions of BTIO. The MPI applications were run by spawning all processes on the same node. For the four process test case, one process was set as the aggregator whereas for all other cases with more number of processes, four processes were set to be used as aggregators. Fig 3.12 shows the performance of BTIO on PVFS2. The I/O performance of the OpenMP based application is clearly better than its MPI counterpart.

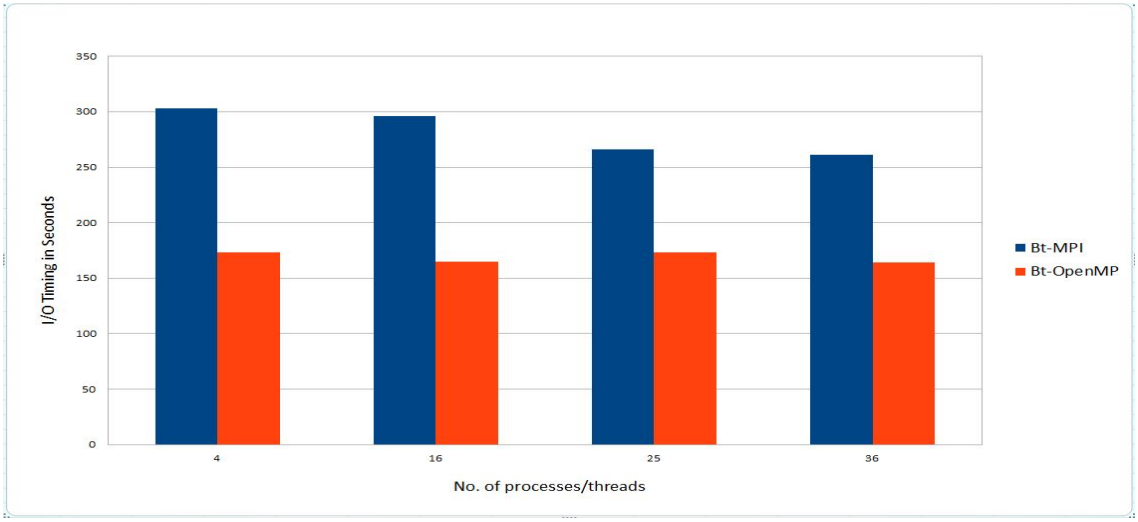

Figure 3.12: Comparing MPI-IO and OpenMP-IO on PVFS2.

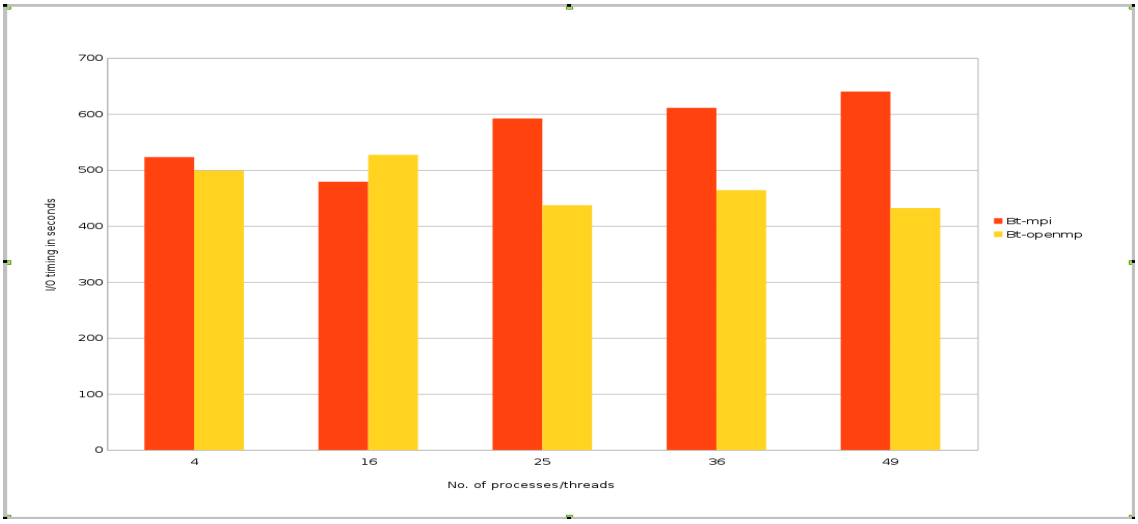

Figure 3.13: Comparing MPI-IO and OpenMP-IO on Lustre.

Fig 3.13 shows the performance on the Lustre file system. It can be seen that as the number of processes/threads increases, shared memory I/O outperforms MPI-IO.

These tests further stress the importance of having the capability of performing I/O in parallel in a shared memory programming model like OpenMP. As supercomputing systems continue to grow, a high performance I/O model for a shared memory programming model can play a crucial role.

Chapter 4

High Performance I/O in HDF5

4.1 Introduction

In this chapter, we discuss two plugins developed for HDF5. The motivation behind developing these plugins is two-fold: a) to provide a novel means of storing HDF5 data on disk that allows for effective semantic analysis of data, and b) provide high I/O throughput for shared memory applications.

As discussed previously, HDF5 is a data model, library and data format. It exports an API and provides a means to store data in a hierarchical format. Natively, all data along with metadata is stored in a single file on disk. This can pose certain limitations as having the ability to distinguish data objects at the storage level can have some advantages, as will be described in the following sections.

Recently, a new layer called the Virtual Object Layer (VOL) has been introduced in HDF5 that allows developing plugins that can store data in a format different from the native format. VOL is a new abstraction layer internal to the HDF5 library [55].

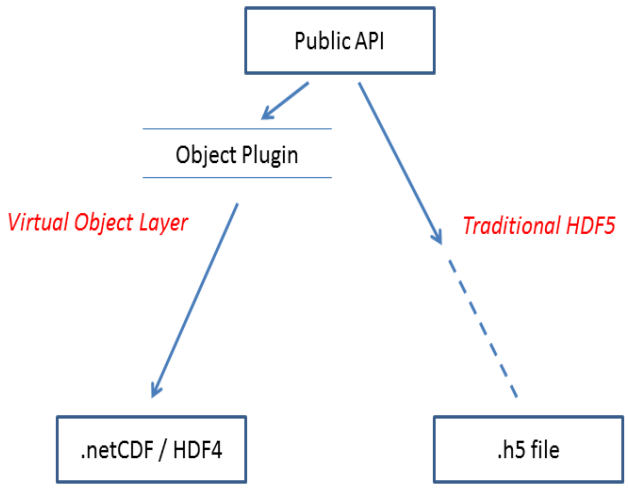

Figure 4.1: HDF5 Virtual Object Layer (VOL)

As shown in Figure 4.1 it is implemented just below the public API. The VOL exports an interface that allows writing plugins for HDF5, thereby enabling developers to store objects in a format different from the default HDF5 file format (like native NetCDF or HDF4 format). Plugin writers provide an implementation for a set of functions that access data on disk. These include functions for file management, dataset creation and access, group creation, to name a few.

Parallel HDF5 (PHDF5) allows multiple processes to access data from the same file, typically using MPI-IO. However, many popular parallel file systems are known to behave poorly under these circumstances where multiple processes access a single, shared file. Secondly, since no information is maintained about the various objects contained in the .h5 file, the file can simply be seen as a linear array of bytes. This prevents any meaningful analysis to be performed on the data independently of the HDF5 application. As an example, it is very difficult to parse and analyze individual

datasets since the file on disk contains metadata along with raw data, which by itself can be interspersed through the file.

4.2 Semantic Analysis of Data

Keeping the limitations of the single file format of HDF5 in mind, two plugins have been developed which store data in a format which enables performing semantic analysis on the data. Instead of storing all objects in a single file, it stores every HDF5 object in a separate location, so that data from different objects are not contained in the same file. In short, a raw mapping of HDF5 objects to the file system has been provided. HDF5 files and groups are stored as directories, whereas datasets and attributes, which contain raw data, are stored in files.

Consider the example shown in Figure 2.3. Using the plugin, file Sample.h5 is stored as a directory. Group G1 is stored as a directory under it, and datasets D1 and D2 are files. Attribute A1 is a file created at the same location as dataset D1 to which it is attached. Additionally, the name of the attribute is stored as *dataset name.attribute name* to denote the dataset to which the attribute is attached. Thus these objects are stored at the following paths as shown in Figure 4.2:

/Sample.h5/

/Sample.h5/G1

/Sample.h5/G1/D3

/Sample.h5/D1

/Sample.h5/D1.A1

/Sample.h5/D2

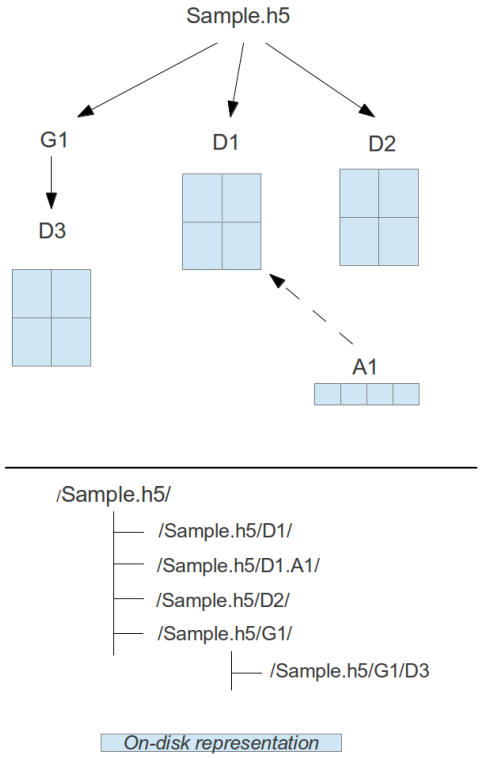

Figure 4.2: HDF5 data stored using the plugin

We can see that the relationship between the objects is represented by their relative paths at the file system. That is, the path /Sample.h5/G1/D3 tells us that D3 is a dataset belonging to Group G1 under the root group of the file. This approach gives us the ability to distinguish HDF5 objects at the storage level. Also, the plugin eliminates the need to explicitly store the metadata describing the relationship between objects. Metadata about datasets, such as the datatype, extent, dimensions etc. are stored in separate hidden files with the naming system *.dataset_name.property_name*. As such, these files represent unix style Xattributes which represent a key value pair, the filename being the *key* and its contents being the *value*.

The above format of storing data allows us to perform at least two different analysis optimizations. To illustrate via an example, imagine storing a three-dimensional ocean model within a PLFS file. The storage system sees the file as an opaque linear array of bytes. With the structure, however, PLFS can provide *active analysis* as well as *semantic restructuring*.

4.2.1 Active Analysis

Active analysis borrows the transducers idea from the Semantic File System [56] which has since been producticized in Google's BigTable and Apache Hbase technologies [22, 57, 58]. With active analysis, the application can ship a data parser function when it creates the PLFS file. As the data is written into PLFS, PLFS can apply the data function on the streaming data. The function will output key-value pairs which PLFS can embed in its extensible metadata (Figure 4.3). In this example, one simple function might record the height of the largest wave. Due to PLFS's model of storing a logical file across multiple physical files, the PLFS extensible metadata can record the height of the largest wave within each physical file. However, given that PLFS now understands the structure of the logical file, these multiple physical files, within the PLFS container, are more accurately thought of as shards. In a future burst buffer architecture [59], these semantic shards will be spread across multiple burst buffer nodes. Therefore, subsequent analysis of the ocean model can quickly find the burst buffer containing the shard with the largest wave by searching a small amount of extensible metadata instead of scanning the entire ocean model.

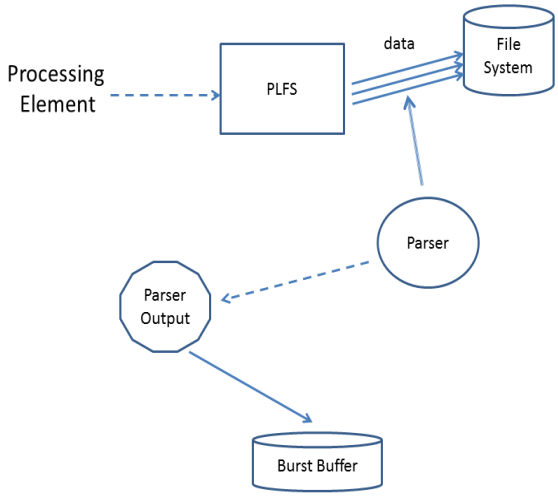

Figure 4.3: Active Analysis of Data

4.2.2 Semantic Restructuring

Semantic restructuring is the idea of reorganizing the data into a new set of semantic shards. This would be done to speed future analysis routines. For example, assume that the ocean model was originally sharded using a row-order organization (i.e. across latitude instead of longitude). An analysis routine which will explore the model along a column-ordering will suffer poor performance with the row-order organization as its access pattern will result in a large number of small reads from a large set of semantic shards. However, by knowing the semantic structure, the analysis routine can request a semantic restructuring (Figure 4.4) which will be a compact, intuitively described request such as "restructure into column-ordering." Without structural knowledge, a semantic restructuring would be significantly more complicated: the analysis routine would have to send a large list of logical offsets

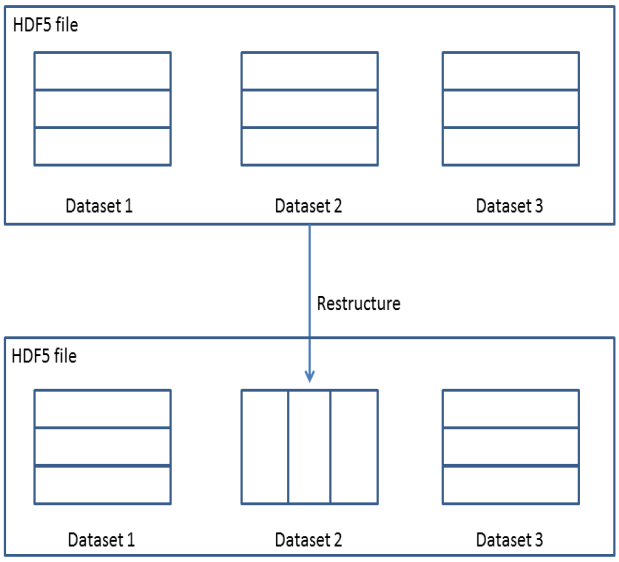

Figure 4.4: Semantic Restructuring of Data

to PLFS to inform it of expected read patterns. In an exascale system, the list of logical offsets will be in the order of one billion. Semantic restructuring shrinks the size of the request to a small constant value. A storage format where datasets are stored in separate files allows us to make modifications to the storage layout of one dataset without affecting other datasets and data.

4.3 A Plugin for Shared Memory Parallelism

4.3.1 Design

As of now, HDF5 does not support multi-threading [60]. HDF5 API calls are not thread-safe. If a user needs to use HDF5 in an OpenMP application, API calls are required to be placed in critical sections using locks. This provides correctness but not parallelization, and having locks generally has a negative impact on the I/O

performance.

A new plugin has been developed using the VOL that aims to provide parallelism in shared memory HDF5 applications. This plugin provides a way to parallelize a process internally using OpenMP threads. The plugin spawns threads and uses *pcollio* to achieve high bandwidth. The current version uses the *omp_file_write/read_com_at_all* interface, where threads provide the same arguments to the function. Thus, the plugin parallelizes an HDF5 application implicitly using threads without user intervention. The user may set the number of threads spawned inside the plugin using the OMP_NUM_THREADS environment variable and set the s_{min} and *active_threads* parameters as desired. The user may also set the library to use PLFS for file systems like Lustre. Note that threads are spawned inside the plugin, they are not spawned by the user application. As such, from the top-level API point of view, HDF5 remains thread-unsafe. Consequently, it implies that multiple threads work on the same dataset and read/write it in parallel, as opposed to having each thread work on a different dataset, which requires threads to be spawned in the main application. This is a simplistic, yet efficient way to achieve high performance I/O in HDF5 when used on a shared memory machine, specially with large datasets.

4.3.2 Evaluation

Experiments were performed on the PVFS2, PVFS2-SSD and the Lustre file systems. The Details on the configuration of these file systems can be found in the previous chapter. These experiments were performed to exhibit improvements in I/O, not

to explore the advantages of performing semantic analysis on data. For experimentation, HDF5's *h5perf* tool was modified to utilize the new plugin. *h5perf* allows configuring various parameters, such as the number of processes, number of datasets, amount of data read/written by a process in a single I/O call (transfer size) etc. In this case, a single large dataset of size 64GB was written to file in transfer sizes of 1MB, 4MB, 64MB and 1GB. Experiments were performed with 1,2,4,8,16,32,48,64 threads. Note that PLFS was used to perform I/O on the Lustre file system. All threads were used as *active threads* and s_{min} was set to 1MB.

Figure 4.5 shows the performance of the plugin on the PVFS2 file system. The left part of the figure shows the write performance whereas the right part shows the read performance. Performance of a single thread is restricted to more than 200MB/s, whereas using multiple threads, a maximum in excess of 600MB/s can be seen. As seen in previous results with *pcollio* on BTIO and MSG, a maximum of 8 threads is sufficient to obtain the best possible bandwidth from the system.

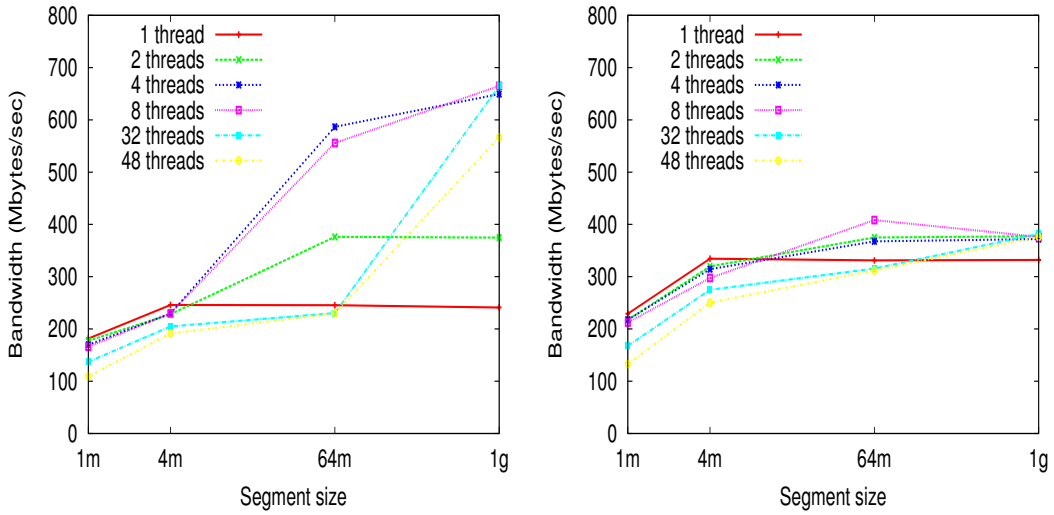

Figure 4.5: Write (left) and Read (right) performance of a multi-threaded plugin for HDF5 on PVFS2

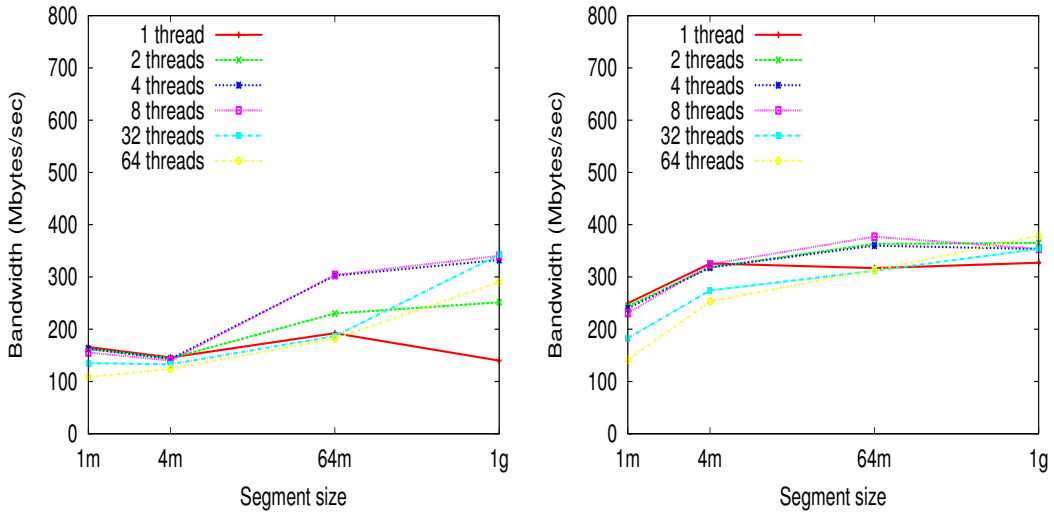

Figure 4.6: Write (left) and Read (right) performance of a multi-threaded plugin for HDF5 on PVFS2-SSD

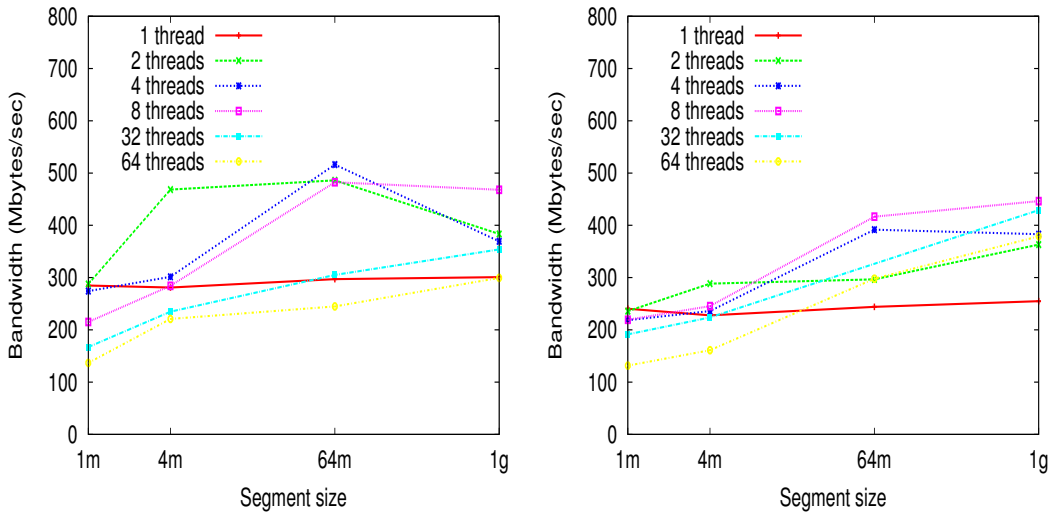

Figure 4.7: Write (left) and Read (right) performance of a multi-threaded plugin for HDF5 on Lustre

A similar trend can be seen on PVFS2-SSD and Lustre, as shown in Figure 4.6 and Figure 4.7. On PVFS2-SSD the performance improvement is limited since the file system is configured over 2 servers only which limits the amount of parallelization that can be performed, whereas for Lustre, more than 2x improvement in performance can be seen for 8 threads as compared to the single-threaded version.

These experiments clearly show the benefit of multi-threading on a widely used library like HDF5. This plugin can be used to improve the I/O performance in high workloads HDF5 applications on shared memory machines using *pcollio*.

4.4 A Plugin using MPI-IO

4.4.1 Design

While the previous plugin was designed to specifically target shared memory machines, another plugin has been developed to target MPI applications. Since parallel HDF5 provides a set of API calls that can be used by parallel MPI applications, this plugin makes the new data format available for these applications.

Traditionally, PHDF5 uses MPI-IO internally to perform I/O. The plugin however uses PLFS instead. It makes direct calls to the PLFS API and does not use its MPI-IO driver (called *ad_plfs*). As such, it has specifically been developed to target file systems like Lustre that benefit from PLFS. All metadata is read/written only by the root process and currently only individual I/O is used. Support for collective I/O operations is not provided as of now.

4.4.2 Evaluation

In this section, the performance of our plugin is presented. These experiments are not targeted towards performing a semantic analysis of data, such as active analysis or semantic restructuring discussed in the previous section. Experiments only evaluate the I/O component of the plugin.

For evaluation purposes, HDF5's *h5perf* performance tool [61] is used. For the measurements, 10 1-dimensional datasets were created, and the total file size was 64 GB or more. In every run, every process contributed equal amount of data per dataset.

Tests were performed on the Lustre parallel file system [20] on the Atlas cluster at University of Dresden. As described previously, the file system has 12 OSTs with a stripe size of 1MB. Tests were run thrice and we present the average bandwidth values, which does not include the time taken to open and close the file.

Tests were performed with 1,2,4,8,32 and 64 processes with a maximum of 4 processes per node. Reads and writes are either contiguous or interleaved; processes either access contiguous locations in file or execute a strided pattern. The performance of the default MPI-IO driver, the plugin, and the PLFS MPI-IO driver (ad_plfs) is compared and presented.

The left part of Figure 4.8 shows the write performance for a transfer size of 1MB for contiguous writes. It can be seen that the plugin regularly outperforms MPI-IO except for the 64 process case, where the metadata overhead incurred by the plugin is high. The performance of ad_plfs is the best for higher process counts. The right part of the figure shows the interleaved write performance for an unaligned transfer size (1M + 10bytes) for a maximum of 8 processes. The write performance of MPI-IO is quite poor in this case. The plugin easily outperforms MPI-IO and almost matches the performance of ad_plfs. Similarly, Figure 4.9 shows the performance of reads.

Overall, results show that the plugin consistently shows good performance, however it does not scale as well as ad_plfs. This is because the plugin currently only supports individual I/O operations. For higher number of processes, the metadata overhead is high since the plugin makes no effort at optimizing those. ad_plfs however does have some optimizations that alleviate the overhead due to metadata creation for larger process sizes. However it should be noted that there are some MPI libraries

tailored to suit specific types of applications and which do not provide an implementation for MPI-IO. Such libraries can benefit from using a plugin that does not rely on MPI-IO.

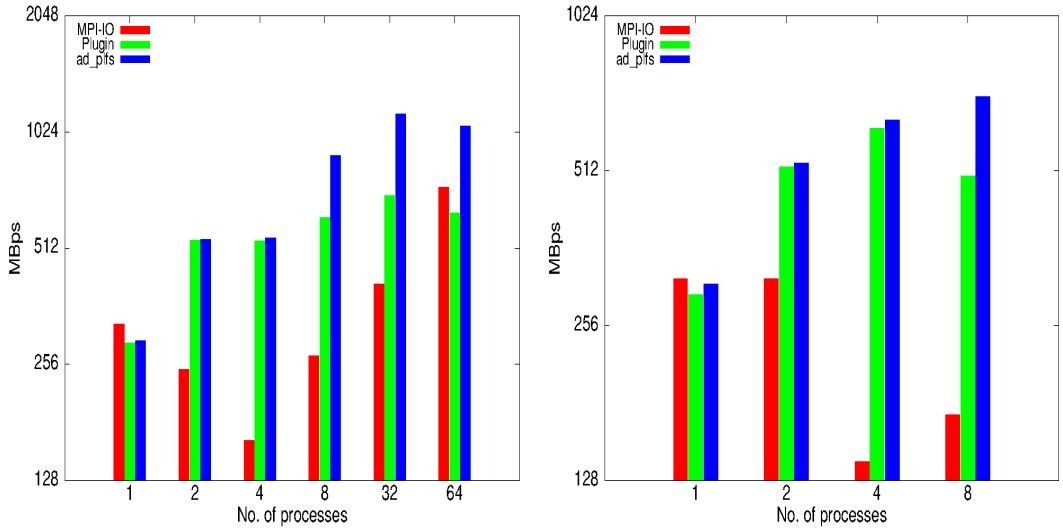

Figure 4.8: Write performance of the HDF5 plugin using MPI

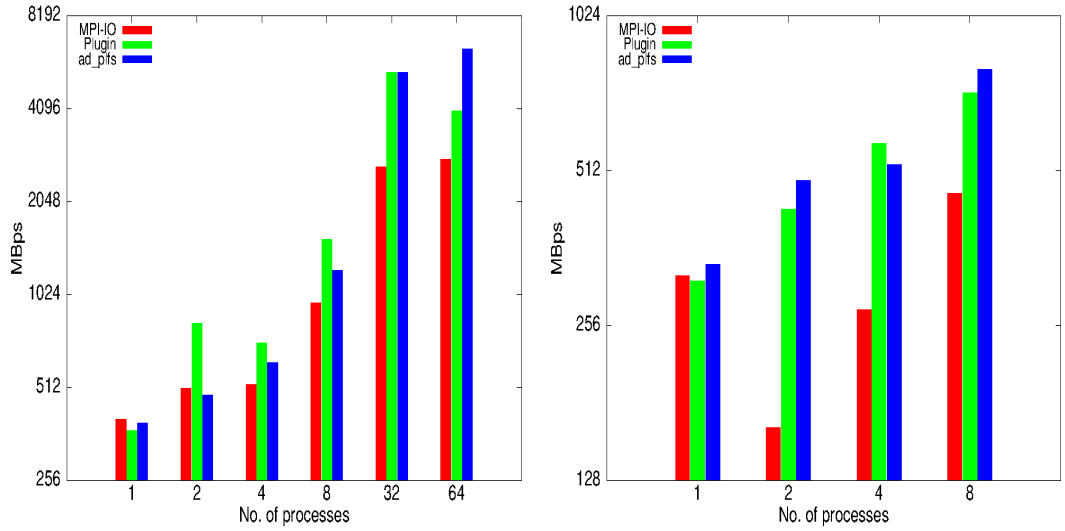

Figure 4.9: Read performance of the HDF5 plugin using MPI

Chapter 5

Summary

This research explores high performance I/O in shared memory applications. Various design alternatives have been discussed and a specification has been provided for parallel I/O in OpenMP. The interfaces provided in the API have been implemented in the OpenUH compiler developed and maintained at University of Houston. All interfaces are collective such that all threads are required to participate in the I/O call. The interfaces are divided into two main categories; one which takes different arguments and the other takes same arguments from threads. Various optimizations, including file system dependent ones such as using PLFS for parallel file systems like Lustre are performed by the library.

Using a set of micro-benchmarks and two application benchmarks the performance of the prototype implementation has been evaluated on three different platforms. The results obtained demonstrate significant performance improvements compared to sequential I/O operations for most scenarios. Further, two important parameters, viz. S_{min} and *active threads* have been identified which influence the I/O

performance of a shared memory application. These represent the optimal amount of data to be read/written per call and the minimum number of threads required to participate in actual I/O respectively.

A comparison of MPI-IO with the I/O library developed for OpenMP shows that the light-weight nature of multi-threaded applications can have significant advantages over using applications that spawn many process on the same node.

Also, the performance of the library has been evaluated on the HDF5 technology suite and a novel way to store data has been proposed for HDF5 which can be of advantage for performing semantic analysis of data.

This work can be extended in multiple directions. First, the interface specification itself can be updated to take advantage of recent developments in OpenMP into account, such as array shaping or explicit tasks. Secondly, a study needs to be performed to determine the S_{min} and *active threads* for a given system configuration. The library could then tune these parameters dynamically to optimize I/O for a given system setup. Finally, the library can be utilized with more real-world applications to determine its true potential.

Bibliography

[1] TOP 500 webpage. http://www.top500.org, 2009.

[2] Wm. A. Wulf and Sally A. Mckee. Hitting the memory wall: implications of the obvious. *ACM Sigarch Computer Architecture News*, 1995.

[3] Seagate Desktop HDD Specifications. http://www.seagate.com/files/staticfiles/docs/pdf/datasheet/disc/barracuda-desktop-hdd-ds-1770-1-1212us.pdf.

[4] Message Passsing Interface Forum. MPI: A Message-Passing Interface Standard, Version 2.2.

[5] Open MPI: Open Source High Performance Computing. http://www.open-mpi.org/.

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[7] W. Richard Stevens and Stephen A. Rago. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 2005.

[8] The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition. http://pubs.opengroup.org/onlinepubs/009695399.

[9] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, Cambridge, Massachusetts and London, England, 2008.

[10] Berkeley UPC - Unified Parallel C. http://upc.lbl.gov/.

[11] R. Numrich and J. Reid. Co-Array Fortran for Parallel Programming. In *ACM Fortran Forum 17(2)*, pages 1–31, 1998. http://citeseer.ist.psu.edu/numrich98coarray.html.

[12] N1836, Summary of Voting/Table of Replies on ISO/IEC FDIS 1539-1, Information technology - Programming languages - Fortran - Part 1: Base language. `ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1836.pdf`.

[13] Rajeev Thakur, William Gropp, and Ewing Lusk. Data Sieving and Collective I/O in ROMIO. In *FRONTIERS 99: Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, page 182. IEEE Computer Society, 1999.

[14] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 23–32, 1999.

[15] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel i/o via a two-phase run-time access strategy. *SIGARCH Comput. Archit. News*, 21(5):31–38, 1993.

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

[17] Gokhan Memik, Mahmut T. Kandemir, and Alok N. Choudhary. Design and evaluation of a compiler-directed collective i/o technique. In *European Conference on Parallel Processing*, pages 1263–1272, 2000.

[18] PVFS2 webpage. *Parallel Virtual File System.* http://www.pvfs.org.

[19] GNU General Public License. `http://www.gnu.org/licenses/gpl.html`.

[20] Lustre webpage. *http://www.lustre.org.*

[21] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *In Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 231–244, 2002.

[22] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[23] PANASAS. *PANASAS ActivStor Parallel Storage Clusters* . http://www.panasas.com/activestor.

[24] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. Plfs: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance*

Computing Networking, Storage and Analysis, SC '09, pages 21:1–21:12, New York, NY, USA, 2009. ACM.

[25] J. Logan and P. Dickens. Towards an understanding of the performance of mpi-io in lustre file systems. In *Cluster Computing, 2008 IEEE International Conference on*, pages 330 –335, 29 2008-oct. 1 2008.

[26] I/O Patterns from NERSC Applications. `https://outreach.scidac.gov/hdf/NERSC\_User\_IOcases.pdf`.

[27] Tarek El-Ghazawi, Francois Cantonnet, Proshanta Saha, Rajeev Thakur, Rob Ross, and Dan Bonachea. *UPC-IO: A Parallel I/O API for UPC. V1.0*, 2004.

[28] ROMIO. `http://www.mcs.anl.gov/research/projects/romio/`.

[29] Rajeev Thakur, William Gropp, and Ewing Lusk. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *In Proceedings of The 6th Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187. IEEE Computer Society Press, October 1996.

[30] Mohamad Chaarawi, Edgar Gabriel, Rainer Keller, Richard L. Graham, George Bosilca, and Jack J. Dongarra. Ompio: a modular software architecture for mpi i/o. In *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, EuroMPI'11, pages 81–89, Berlin, Heidelberg, 2011. Springer-Verlag.

[31] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, metadata rich io methods for portable high performance io. In *In Proceedings of IPDPS'09, May 25-29, Rome, Italy*, 2009.

[32] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, AD '11, pages 36–47, New York, NY, USA, 2011. ACM.

[33] HDF group. HDF5 Users. `http://www.hdfgroup.org/HDF5/users5.html`.

[34] Parallel HDF5. `http://www.hdfgroup.org/HDF5/PHDF5/`.

[35] S. A. Brown, M. Folk, G. Goucher, and R. Rew. Software for Portable Scientific Data Management. *Computers in Physics*, 7(3):304–308, May/June 1993.

[36] Message Passing Interface Forum. *MPI-2.2: Extensions to the Message Passing Interface*, September 2009. `http://www.mpi-forum.org`.

[37] Yizhe Wang, K. Davis, Yuehai Xu, and Song Jiang. iharmonizer: Improving the disk efficiency of i/o-intensive multithreaded codes. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 921 –932, may 2012.

[38] OpenMP Application Review Board. *OpenMP Application Program Interface, Draft 3.0*, October 2007.

[39] Eduard Ayguadé, Rosa M. Badia, Pieter Bellens, Daniel Cabrera, Alejandro Duran, Roger Ferrer, Marc González, Francisco D. Igual, Daniel Jiménez-González, and Jesús Labarta. Extending openmp to survive the heterogeneous multi-core era. *International Journal of Parallel Programming*, 38(5-6):440–459, 2010.

[40] The OpenUH Compiler Project. `http://www.cs.uh.edu/~openuh`, 2011.

[41] Chunhua Liao, Oscar Hernandez, Barbara Chapman, Wenguang Chen, and Weimin Zheng. OpenUH: An optimizing, portable OpenMP compiler. In *12th Workshop on Compilers for Parallel Computers*, January 2006.

[42] Deepak Eachempati, Hyoung Joon Jun, and Barbara Chapman. An open-source compiler and runtime implementation for coarray fortran, 2010.

[43] O. Hernandez, R.C. Nanjegowda, B. Chapman, V. Bui, and R. Kufrin. Open Source Software Support for the OpenMP Runtime API for Profiling. In *ICPPW'09.*, pages 130–137. IEEE, 2009.

[44] M. Itzkowitz, O. Mazurov, N. Copty, and Y. Lin. White Paper: An OpenMP Runtime API for Profiling. Technical report, Sun Microsystems, Inc., 2007.

[45] Barbara M. Chapman, Lei Huang, Haoqiang Jin, Gabriele Jost, and Bronis R. de Supinski. Toward enhancing OpenMP's work-sharing directives. In *Europar 2006*, pages 645–654, 2006.

[46] R. Nanjegowda, O. Hernandez, B. Chapman, and H. Jin. Scalability evaluation of barrier algorithms for OpenMP. *Evolving OpenMP in an Age of Extreme Parallelism*, pages 42–52, 2009.

[47] Oscar Hern, Chunhua Liao, and Barbara Chapman. Dragon: A static and dynamic tool for openmp, 2005.

[48] Mohamad Chaarawi and Edgar Gabriel. Automatically Selecting the Number of Aggregators for Collective I/O Operations. In *Workshop on Interfaces and Abstractions for Scientific Data Storage, IEEE Cluster 2011 conference*, page t.b.d, Austin, Texas, USA, 2011.

[49] Sarp Oral, Feiyi Wang, David Dillow, Galen M. Shipman, Ross Miller, and Oleg Drokin. Efficient object storage journaling in a distributed parallel file system. In *Randal C. Burns and Kimberly Keeton (Eds), 8th USENIX Conference on File and Storage Technologies*, pages 143–154, 2010.

[50] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. Plfs: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 21:1–21:12, New York, NY, USA, 2009. ACM.

[51] Phillip M. Dickens and Jeremy Logan. Y-lib: a user level library to increase the performance of mpi-io in a lustre file system environment. In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 31–38, New York, NY, USA, 2009. ACM.

[52] P. Wong and R. F. Van der Wijngaart. *NAS Parallel Benchmarks I/O Version 3.0*. Technical Report NAS-03-002, Computer Sciences Corporation, NASA Advanced Supercomputing (NAS) Division.

[53] Edgar Gabriel, Vishwanath Venkatesan, and Shishir Shah. Towards High Performance Cell Segmentation in Multispectral Fine Needle Aspiration Cytology of Thyroid Lesions. In *11th International Conference on Medical Image Computing and Computer Assisted Intervention, Workshop on High-Performance Medical Image Computing and Computer Aided Intervention*, page t.b.d., New York, NY, USA, September 2008.

[54] D.J. Kerbyson, K.J. Barker, A. Vishnu, and A. Hoisie. Comparing the performance of blue gene/q with leading cray xe6 and infiniband systems. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, pages 556–563, 2012.

[55] Virtual Object Layer. `https://confluence.hdfgroup.uiuc.edu/display/VOL/Virtual+Object+Layer`.

[56] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole, Jr. Semantic file systems. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, SOSP '91, pages 16–25, New York, NY, USA, 1991. ACM.

[57] Jeff Dean. Designs, Lessons and Advice from Building Large Distributed Systems, 2009. `http://www.odbms.org/download/dean-keynote-ladis2009.pdf`.

[58] Andrew Purtell Mingjie Lai, Eugene Koontz. Apache HBase. https://blogs.apache.org/hbase/entry/coprocessor_introduction.

[59] Ning Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership-class storage systems. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1 –11, april 2012.

[60] Thread Safe HDF5. http://www.hdfgroup.uiuc.edu/papers/features/mthdf/index.html.

[61] h5perf User Guide. http://www.hdfgroup.org/HDF5/doc/UG/index.html.