CHALLENGES FOR PARALLEL I/O IN GRID COMPUTING

AVERY CHING[1] , KENIN COLOMA[2] , AND ALOK CHOUDHARY[3]

[1] Northwestern University, `aching@ece.northwestern.edu`,[*]
[2] Northwestern University, `kcoloma@ece.northwestern.edu`,
[3] Northwestern University, `choudhar@ece.northwestern.edu`

Abstract.
GRID computing uses heterogeneous resources to solve large-scale computational problems. With increasing dataset sizes in data-intensive GRID applications reaching terabytes and even petabytes, high-performance I/O is emerging as an important research area. We discuss much of the current status of research in GRID I/O. We also describe our research ideas for handling noncontiguous I/O access, consistency, caching, fault-tolerance and improved performance.

Key words. GRID I/O, parallel I/O, caching, versioning, locking

AMS subject classifications. 15A15, 15A09, 15A23

1. Introduction.

With virtually limitless resources, GRID computing has the potential to solve large-scale scientific problems that eclipse even applications that run on the largest computing clusters today. The architecture of a computing GRID simply consists of a heterogeneous network infrastructure connecting heterogeneous machines presumed to be larger than most clusters of the future. Most GRID environments consist of clusters-of-clusters (TeraGRID) or harnessing the compute power of ordinary users (Seti@home) across the Internet.

There are numerous research projects involved in GRID I/O. We discuss many of them in detail in this book chapter. Some of the GRID I/O research projects can be classified as the placement of middleware code (most commonly data filtering and data queries) in the GRID. This includes Armada [1], Grid Datafarm [2], DataCutter [3], and Mocha [4]. Creating a GRID based MPI and MPI-IO implementation has been the focus of several projects including [5, 6, 7]. Other GRID I/O work includes GridFTP and data replication.

Much of the previous and current work tackles the challenges of how and where to place executable code in the GRID and how to define GRID computing communities. Replication techniques are another way of moving data closer to the computation. Deciding where and how much code to deploy near data sources reduces network traffic. The work on MPI for GRID computing is mainly designed to allow the existing MPI based scientific applications to leverage the GRID.

GRID applications today generate incredibly large datasets. Some examples of these applications include climate modeling, simulation of physical phenomena, visualization tools, etc. I/O is already the slowest computational component by several orders of magnitude when compared to memory or processor speed. In order to effectively access and manipulate their large datasets, GRID applications must use parallel I/O. Multiple clients and multiple I/O access points are essential for higher bandwidth.

Most scientific applications have complex data structures that result in noncontiguous I/O access patterns. While some applications may directly use MPI derived datatypes, it is more common for application designers to unknowingly use MPI-IO through higher level application I/O libraries such as HDF4, HDF5, NetCDF, or

[*]Corresponding author

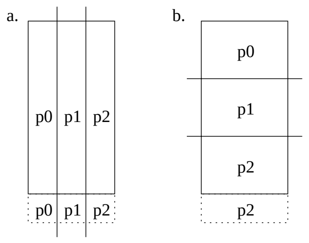

FIGURE 1. File striping versus file fragments.

pNetCDF. These libraries issue file system operations most commonly through MPI-IO and we describe the needs and benefits of having high-performance noncontiguous I/O interfaces at the file system level.

While previous and current GRID I/O research will help reduce network traffic and make existing APIs work in the GRID, they do not fully address some of the most difficult I/O topics for GRID computing: noncontiguous I/O, caching, consistency, and fault-tolerance. We provide our ideas on these topics for improving large-scale I/O that are based on the Versioning Parallel File System (VPFS), file domains, and Direct Access Cache (DAChe).

In this chapter we begin by examining some of the other GRID I/O projects in Section 2. We describe example GRID applications and their I/O profiles in Section 3. In Section 4, we discuss commonly used file formats for scientific computing (HDF and NetCDF) and their associated noncontiguous data access patterns. Section 5 explains the different methods for accessing noncontiguous data and possible improvements for GRID I/O. The next two sections are a preview of the research topics we are currently pursuing. Section 6 describes a next generation file system called the Versioning Parallel File System that will address issues of fault-tolerance, atomicity, and fast noncontiguous I/O for scientific datasets. Section 7 examines issues of caching and consistency for future high-performance I/O. Finally, Section 8 concludes this book chapter.

2. Current GRID I/O Research. Grid Datafarm (Gfarm) provides a global parallel file system designed for grids of clusters that span over 10,000 nodes [2]. The Gfarm model specifically targets applications where records or objects are analyzed independently. With its file distribution and process scheduling techniques, it achieves scalable bandwidth by keeping processes and their associated file data near each other. Gfarm is comprised of three major components: the Gfarm file system, the Gfarm process scheduler, and the Gfarm parallel I/O API. Gfarm file system nodes and Gfarm metadata nodes together make up the Gfarm file system.

One of the major differences between Gfarm and a traditional distributed file system is that Gfarm file system nodes act as both I/O nodes and compute nodes. Computation is moved to the data in the Gfarm resource allocation scheme. Gfarm is a parallel file system with files partitioned into *file fragments*. Traditional parallel file systems [8, 9, 10] stripe files across I/O access points in a round-robin manner similar to hardware RAID techniques as shown in Figure 1. File fragments can be of arbitrary size and can be stored on any node. If an I/O operation does not exceed the size of a file fragment and only a single replica of the relevant file fragment exists, the I/O operation can only attain the maximum bandwidth of a single I/O server. However, file fragments can provide an easier data structure for replication on other I/O access points since they are statically defined in size.

In order to move data closer to computation, Gfarm files can be replicated on additional Gfarm file system nodes. A Gfarm file is write-once. If another write occurs to the same file all other replicas are invalidated. Gfarm follows a similar consistency model to AFS where updated file content can be accessed only by a process that opens the file after a writing process closes it [11].

The Armada parallel file system [1] is a flexible file system that allows application supplied code to run on compute nodes, I/O nodes, and/or intermediate nodes. The core file system is very basic. The only interfaces provided on data servers are open, close, read, and write. Remote datasets are accessed through a network of distributed application objects or *ships* that provide some functionality to define the behavior and structure of the overall system. Armada uses *blueprints* to describe an arrangement of ships on the network. Armada requires a two-step process in accessing file data. First, a programmer must define a blueprint that describes the arrangement of ships in the network. Secondly, the ships are deployed into the network. There are four classes of ships in Armada: structural ships (describes data organization and layout), optimization ships (improve performance), filter ships (manipulate data) and interface ships (provide semantic meaning for data). Much of the work in Armada deals with the graph partitioning problem of laying out ships effectively in the GRID.

MOCHA is a self-extensible database middleware system for interconnecting distributed data sources [4]. It is self-extensible in that new application specific functionality can be added on demand. MOCHA (Middleware Based On a Code Shipping Architecture) is implemented in Java and allows Java bytecode to be shipped to remote data sources. While any code can be run on the data sources through the extensible MOCHA middleware, the main goal of MOCHA is to place data-reducing operators at the data sources and data-inflating operators near the client, thereby significantly reducing network bandwidth needs between the client and the data source. A prototype of MOCHA was able to substantially improve query performance by a factor of 4:1 in the case of aggregates and 3:1 in the case of projections, predicates, and distributed joins when evaluated on the Sun Ultra SPARC platform [4].

DataCutter is a middleware infrastructure that enables processing of scientific datasets stored in archival storage systems across a wide-area network. As a middleware layer, DataCutter uses multidimensional range queries to create subsets of datasets. It can also perform application specific aggregation on scientific datasets stored in an archival storage system.

The application processing structure is decomposed into a set of processes called filters that can be placed anywhere on the GRID, but are usually placed close to the archival storage server. While data processing at the server is useful for eliminating most of the network traffic, servers can be overloaded by several processes simultaneously filtering on the server. This is why DataCutter is purposely flexible enough to place these filters at appropriate places in the GRID.

The Globus Project aims to develop a basic software infrastructure for building grids. The toolkit implements services for security, resource location, resource management, etc. The Globus Data Grid architecture [12] provides a scalable infrastructure for managing storage resources and data in the GRID. A replica selection service has been developed for the Globus Data Grid. GridFTP, a part of the Globus data management services can access data in parallel, providing a high-performance means for transferring datasets from a remote data center to a local site.

Much of the other current GRID I/O research involves helping MPI applications run on the GRID. MPICH-G2 [7] is a GRID-enabled implementation of MPI. It extends the Argonne MPICH implementation of MPI by using the Globus Toolkit for authentication, authorization, resource allocation, executable staging, I/O, process creation, monitoring, and control. In [6], the authors implement a GridFTP driver for ROMIO (Argonne's implementation of the MPI-IO interface).

3. GRID Applications. In order to motivate a need for high-performance I/O in the GRID, we begin by describing the datasets created and manipulated by GRID applications. In the final two applications (ASC FLASH and tile display) we describe their access patterns in detail.

Global climate modeling is a specific group of applications that characterize GRID computing from distributed data collection to modeling and collaboration. The Earth System Grid (ESG) interdisciplinary project is aimed at creating a virtual collaborative environment linking distributed centers, users, models, and data [13]. The largest challenge that such a research project faces is that increasingly complex datasets overwhelm current storage technologies. Simulations of the Earth-system components that are high resolution and contain nearly continuous time steps easily generate petabytes of data. Manipulating, archiving, post-processing, retrieving, and visualizing these datasets is difficult for current I/O technology. By the end of 2004, approximately 100 terabytes of Parallel Climate Model data had been collected and distributed across several data centers [14]. Accessing this data fast enough for useful post-processing applications is a difficult challenge.

IPARS (Integrated Parallel Accurate Reservoir Simulation) [15] is a software system for large-scale oil reservoir simulation studies. This code is highly data dependent, meaning that the next iterative set of simulations depends on the analysis of previous simulations. In this paper [16], Saltz et al. generated data from a black-oil (three phase) flow problem on a grid with 9,000 cells. At every time step, 17 variables are output from each node in the grid. 10,000 time steps make a realization of about 6.9 gigabytes. Accumulating 207 realizations from 18 geostatistical modes and 4 well configuration/production scenarios generates an overall dataset size close to 1.5 terabytes. In order to obtain more precision (adding additional cells, more time steps, or more variables), the overall dataset size could easily scale to hundreds of terabytes. A variety of post-processing operations including the determination of the location and size of regions of bypassed oil, the finding of the representative realization, or the visualization of any of these results requires efficient GRID I/O technologies.

The Advanced Simulation and Computing (ASC) FLASH code is an application designed to simulate matter accreted on the surfaces of compact stars, nuclear ignition of the accumulated material, and the subsequent evolution of the star's interior, surface, and exterior. It incorporates the physics of hydrodynamics, nuclear physics, gravity, cosmology, and particle interaction. The ASC FLASH code also provides tools for setup, Adaptive Mesh Refinement (AMR) usage, parallel I/O through HDF5 or pNetCDF, profiling, and runtime and post-process visualization. AMR is a popular technique for describing scientific datasets due to its ability to place high resolution grids where scientists need them [17]. The ASC FLASH application has been shown to scale to thousands of processors and contains over a half million lines of code.

The ASC FLASH code is heavily I/O intensive according to [18]. A normal production run will create over half a terabyte of data divided between plot files and checkpoint files. Since these I/O requirements were determined from old data, additional precision or larger-scale simulation will easily push I/O needs to hundreds of terabytes or petabytes. As the ASC FLASH application is run on larger-scale machines or across the GRID, it will need to write checkpoints more often. A checkpoint

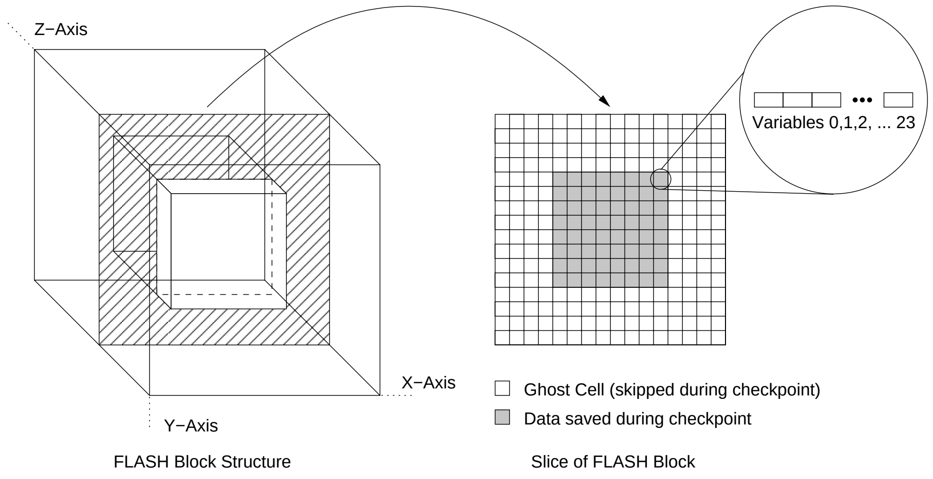

Figure 2. ASC FLASH memory block.

should generally be written out from an application in an interval that is at least less than the system failure interval to make progress. Larger-scale GRIDs or clusters-of-clusters further decrease the mean time to failure due to less reliable networks and the increasing number of machines used.

In order to describe how noncontiguous I/O accesses occur, we explain the data structures of the ASC FLASH code. The ASC FLASH data structures are stored in three-dimensional blocks that have 24 variables. These three-dimensional blocks are surrounded by guard cells in each of the three dimensions. Figure 2 shows a logical view of the memory structure. Each process keeps track of 50-100 blocks (depending on processor workload). When performing a checkpoint operation, the guard cells are not saved. The data is stored variable-first, which results in a noncontiguous access pattern.

High resolution visualization of data is an important analysis tool for many scientific datasets and can be far removed from the point or points of data generation.

Tile displays (using multiple displays) are commonly used to cost-effectively increase viewing resolution. Numerous companies and research institutes (Argonne National Laboratory, Sandia National Laboratory, etc.) use this technique. The I/O access pattern presented by such an application breaks up a movie frame into tiles that a processor outputs to a display as in Figure 3. It is worth noting that the overlapping pixels are sometimes used for seamless blending in projector environments. In our example, a 3 x 2 tile display, each frame consists of approximately 10 megabytes. If we assume 30 frames per second, we need 300 megabytes per second of I/O bandwidth to supply the tiles to the display. Larger tile displays will require additional I/O requirements.

4. Scientific File Formats. Many of the applications listed above require complex data structures. The two most common scientific file formats are NetCDF and HDF. NetCDF is a standard library interface to data access functions for storing and retrieving array data [19] [20]. pNetCDF is a parallel interface for NetCDF datasets [21]. HDF4 and HDF5 are the most popular versions of HDF. They both store multi-dimensional arrays together with ancillary data in portable, self-describing file formats. Similar to NetCDF, HDF4 supports a serial interface. HDF5, however, features a redesigned API and includes parallel I/O though MPI-IO. In this section

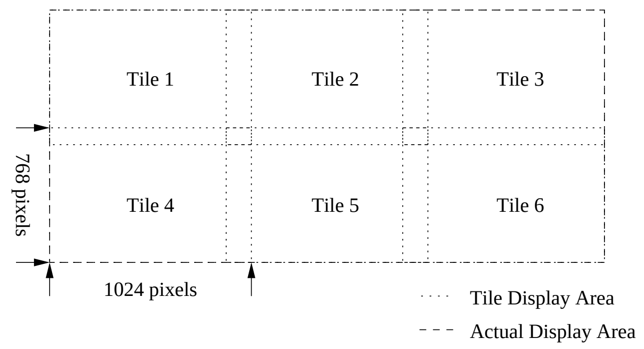

FIGURE 3. Example tile display access pattern.

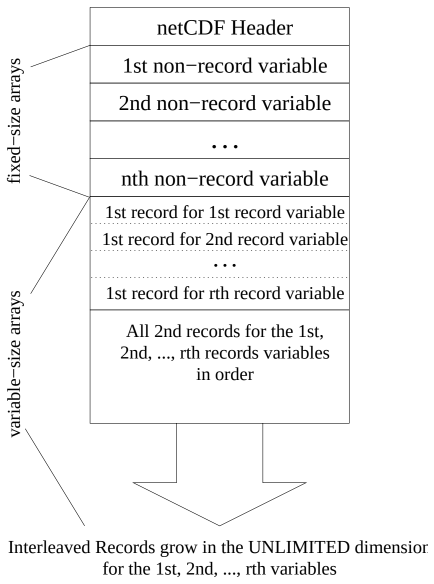
netCDF Header
1st non−record variable
2nd non−record variable
...
nth non−record variable
1st record for 1st record variable
1st record for 2nd record variable
...
1st record for rth record variable
All 2nd records for the 1st, 2nd, ..., rth records variables in order
fixed−size arrays
variable−size arrays
Interleaved Records grow in the UNLIMITED dimension for the 1st, 2nd, ..., rth variables

FIGURE 4. NetCDF file format.

we discuss how and why their data layouts affect high-performance I/O for scientific applications in the GRID.

The NetCDF dataset is array-oriented. Its basic format is a file header followed by an organized data section. The file header contains the metadata for dimensions, attributes, and variables. The data part consists of fixed size data (containing the data for variables that don't have an unlimited dimension) and followed by record data (containing the data records for variables that have an unlimited dimension). For variable-sized arrays, NetCDF makes a record of an array as a subarray comprising all fixed dimensions. The records are interleaved and can grow in the unlimited dimension. The physical data layout is shown in Figure 4. The data is represented in the XDR format for machine-independent use. NetCDF arrays exhibit a great deal of regularity and can be described concisely to the underlying MPI-IO layer.

HDF5 is both a general purpose library and a file format for storing scientific data [22]. It uses a tree-like file structure (similar to the UNIX file format) to store data. Super blocks, header blocks, data blocks, extended header blocks and extended data blocks are irregularly positioned throughout the file. This hierarchical approach

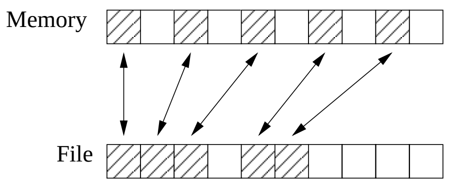

FIGURE 5. Example POSIX I/O call. The traditional POSIX interfaces for this access pattern uses five file system calls, one per contiguous region.

affords some additional flexibility relative to the NetCDF file format. One example of this flexibility is array growth in multiple dimensions. However, from the perspective of MPI-IO, HDF5 data accesses will likely be irregular, unlike NetCDF.

High level application I/O libraries like NetCDF or HDF provide a simple and rich I/O interface for programmers to use when creating scientific applications. The I/O access patterns that result from using these high level libraries are usually noncontiguous in nature. This has various implications when considered in a large-scale environment.

5. Noncontiguous I/O. The scientific applications described in Section 3 have access patterns that are noncontiguous. In most scientific applications these noncontiguous access patterns display a high degree of regularity, (such as the FLASH memory structure and the tile display code). Due to bandwidth requirements, nearly all such applications will access data through a parallel file system such as the Parallel Virtual File System (PVFS) [10], the General Parallel File System (GPFS) [8], Lustre [23], etc. Numerous studies have shown that I/O for large-scale applications is dominated by numerous noncontiguous I/O requests [24, 25, 26].

Application I/O begins with logical data structures and I/O access patterns that are mapped into high level interfaces such as HDF or NetCDF, which then pass through lower level interfaces such as MPI-IO before finally accessing the file system. As access patterns move down through these layers, it is often the case that they progressively become more generic. When they are finally translated into file system calls, these complex access patterns typically become a set of individual UNIX read() or write() file system calls. Recent work on noncontiguous I/O has developed a variety of methods for handling noncontiguous access patterns in efficient ways.

The most naive method for implementing noncontiguous I/O is through the POSIX I/O interface (using UNIX read()/write() file system calls). It is described in Section 5.1. Still only using the POSIX I/O interface, data sieving I/O (Section 5.2) and collective I/O (Section 5.3) methods were created for better performance. Recent work suggests that new file system interfaces (list I/O in Section 5.4 and datatype I/O in Section 5.5) result in much needed performance improvements for noncontiguous I/O in many scientific applications. In Section 5.6 we discuss the use of these file system techniques in GRID applications.

5.1. POSIX I/O. Most parallel file systems implement the POSIX I/O interface. This interface offers contiguous data access only. To support noncontiguous access with POSIX I/O one must break the noncontiguous access pattern into a sequence of contiguous I/O operations. POSIX I/O can service noncontiguous access patterns in this manner, however there can significant overhead in the number of I/O requests that must be processed by the underlying file system. Because operations in parallel file systems often require data movement over a network, latency for I/O

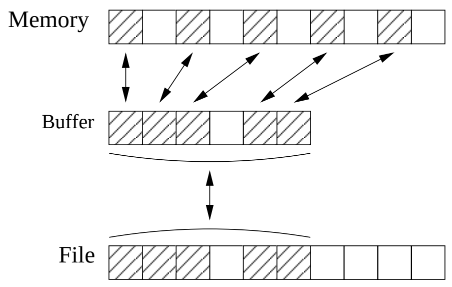

Figure 6. Example data sieving I/O call. Data is first read as a large contiguous file region into a buffer. Data movement is subsequently performed between memory and the buffer. For writes, the buffer is flushed back to disk.

operations can be high. For this reason, performing many small I/O operations to service a single noncontiguous access is very inefficient. An example of the POSIX I/O interface used for noncontiguous I/O access is shown in Figure 5. Fortunately for users of these file systems, two important optimizations have been devised for more efficiently performing noncontiguous I/O using only POSIX I/O calls: data sieving and two-phase I/O.

5.2. Data Sieving I/O. One of the most significant disadvantages of POSIX I/O is that each of the individual POSIX I/O operations incurs significant processing overhead independent of the size of data accessed. An innovative approach called data sieving solves that problem within the limited boundaries of the POSIX I/O interface by using one large contiguous I/O operation to access several requested file regions. By reading a large contiguous region from file into a temporary memory buffer and then performing the user requested data movement operations on the necessary file regions in the buffer instead of directly in the file, a single I/O request can service the needs of a file access pattern that contained multiple file regions as shown in Figure 6. In the read case, nothing further is required, but in the write case, the entire buffer must be written back to disk. Not only does this approach reduce the number of I/O requests required to serve a noncontiguous I/O access pattern, but it also takes advantage of the inherent mechanical properties of the underlying disk (larger I/O operations attain higher bandwidth). While this method helps to fix the numerous I/O shortcomings of the POSIX I/O interface, it also introduces new overheads. While accessing an encompassing contiguous file region reduces the I/O requests necessary, it forces the I/O system to access data that is not desired. Accessing wasted data causes two problems. It introduces higher latency from accessing useless data and also requires file regions to be locked in the write case due to re-writing unwanted (possibly stale) data to the I/O system. The wasted data accessed once during reads and twice during writes can have a significant impact on performance. If the I/O access pattern exhibits small amounts of wasted file data, data sieving can greatly improve performance. However, if the I/O access pattern is sparse in file, performance can actually fall well below that of POSIX I/O because of the time spent accessing unnecessary data on disk and synchronization overhead.

5.3. Two-Phase I/O. Interfaces such as MPI-IO retain a great deal of information about how the application as a whole is accessing storage. One example of this is the collective I/O calls that are part of the MPI-IO API. Collective I/O calls allow applications to tell the MPI-IO library not only that each process is performing

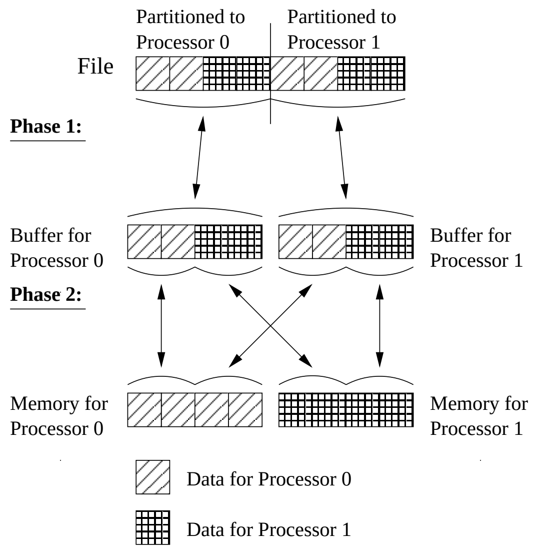

FIGURE 7. Example two-phase I/O call. Interleaved file access patterns can be effectively accessed in larger file I/O operations with the two-phase I/O method.

I/O, but that these I/O operations are part of a larger whole. This information provides additional opportunities for optimization not available to application processes performing independent operations. One example of a collective I/O optimization is the two-phase method from [27]. The two-phase method uses both POSIX I/O and data sieving. The two-phase method identifies a subset of the application processes that will actually perform I/O; these processes are called aggregators. Each aggregator is responsible for I/O to a specific and disjoint portion of the file. ROMIO calculates these regions dynamically based on the aggregate size and location of the accesses in the collective operation. Figure 7 shows how read operations using the two-phase method are performed. First, aggregators read a contiguous region containing desired data from storage and put this data in a temporary buffer. Next, data is redistributed from these temporary buffers to the final destination processes. Write operations are performed in a similar manner. First, data is gathered from all processes into temporary buffers on aggregators. Next, this data is written back to storage using POSIX I/O operations. An approach similar to data sieving is used to optimize this write back to storage in the case where there are still gaps in the data. Data sieving is also used in the read case. Alternatively, other noncontiguous access methods, such as the ones described in forthcoming subsections, can be leveraged for further optimization. Two-phase I/O has a distinct advantage over data sieving alone in that it is significantly more likely to have a higher percentage of useful data accessed in the data sieving buffer read since each aggregator has combined the file regions for all the processes doing the collective I/O while data sieving alone will only do one processor's I/O file regions at a time. Another advantage of two-phase I/O over data sieving is that since none of the aggregators are overlapping in specific file region responsibilities there will be less wasted file data accessed by the aggregators in two-phase I/O than the clients utilizing the data sieving optimization individually. If we can assume that this application is the only application accessing the data, we don't need to synchronize the write operations with locks when performing collective I/O. The combination of these advantages makes the reads and writes in two-phase

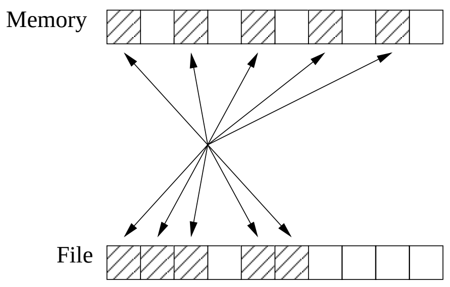

FIGURE 8. Example list I/O call. Only a single I/O request is required to process this noncontiguous access.

```
int listio_read(int fd, int mem_list_count, void *mem_offsets[],
                int mem_lengths[], int file_list_count, int file_offsets[],
                int file_lengths[])

int listio_write(int fd, int mem_list_count, void *mem_offsets[],
                 int mem_lengths[], int file_list_count, int file_offsets[],
                 int file_lengths[])
```

FIGURE 9. List I/O prototypes.

I/O more efficient than data sieving in most cases.

Two-phase I/O performance also relies heavily on the MPI implementation's high-performance data movement. If the MPI implementation is not significantly faster than the aggregate I/O bandwidth in the system, the overhead of the additional data movement in two-phase I/O is likely to prevent two-phase I/O from outperforming the direct access optimizations (data sieving I/O , list I/O, and datatype I/O). The double transfer of data (from I/O system to aggregators to clients or vice versa) can, in certain cases, cause performance worse than data sieving I/O.

5.4. List I/O. The list I/O interface is an enhanced parallel file system interface designed to support noncontiguous accesses shown in Figure 8. List I/O is an interface for describing accesses that are both noncontiguous in memory and file in a single I/O request (see prototypes in Figure 9. With this interface an MPI-IO implementation can flatten the memory and file datatypes (convert them into lists of contiguous regions) and then describe an MPI-IO operation with a single list I/O call. Given an efficient implementation of this interface in a parallel file system, this interface can provide a significant performance boost. In previous work we discussed the implementation of list I/O in PVFS and support for list I/O under the ROMIO MPI-IO implementation [28, 29]. The major drawbacks of list I/O are the creation and processing of these large lists and the transmission of the file lists from client to server within the parallel file system. Additionally, since we want to bound the size of the list I/O requests going over the network, only a fixed number of file regions can be described in one request. So while list I/O does significantly reduce the number of I/O operations (in our implementation by a factor of 64), there is still a linear relationship between the number of noncontiguous regions and the number of I/O operations (within the file system layer). In fact, for noncontiguous access patterns that generate the same number of I/O requests in POSIX I/O as file regions, we see that the list I/O performance curves run parallel to the POSIX I/O bandwidth curves (shifted upward due to the constant reduction in I/O requests). Because of these is-

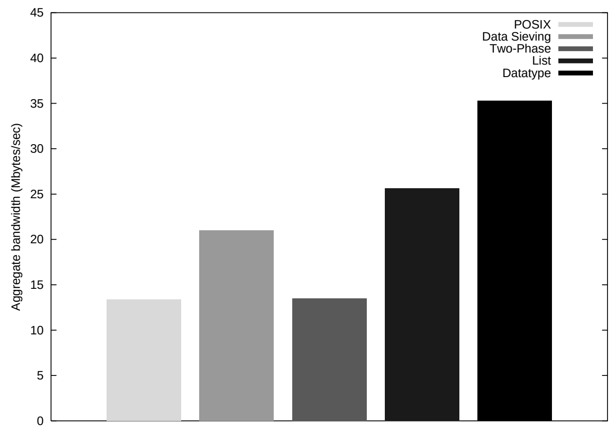

FIGURE 10. Tile display I/O performance.

sues, we see that while list I/O is an important addition to the optimizations available under MPI-IO, it does not replace two-phase or data sieving, but rather complements them.

5.5. Datatype I/O. Datatype I/O is an effort to address the deficiency seen in the list I/O interface when faced with an access that is made up of many small regions, particularly one that exhibits some degree of regularity. Datatype I/O borrows from the datatype concept used in both message passing and I/O for MPI applications. The constructors used in MPI types allow for concise descriptions of the regular, noncontiguous data patterns seen in many scientific applications (such as extracting a row from a two-dimensional dataset). The datatype I/O interface replaces the lists of I/O regions seen in the list I/O interface with an address, count, and datatype for memory, and a displacement, datatype, and offset into the datatype for file. These parameters correspond directly to the address, count, datatype, and offset into the file view passed into an MPI-IO call and the displacement and file view datatype previously defined for the file. The datatype I/O interface is not meant to be used by application programmers; it is an interface specifically for use by I/O library developers. Helper routines are used to convert MPI types into the format used by the datatype I/O functions.

Our prototype implementation of datatype I/O was written as an extension to the Parallel Virtual File System (PVFS) in [30]. The ROMIO MPI-IO implementation was likewise modified to use datatype I/O calls for PVFS file system operations. It is important to note that while we present this work in the context of MPI-IO and MPI datatypes, nothing precludes us from using the same approach to directly describe datatypes from other APIs, as in HDF5 hyperslabs.

Since it can be mapped directly from an MPI-IO I/O operation with a one-to-one correspondence, datatype I/O greatly reduces the amount of I/O calls necessary to service a noncontiguous request when compared to the other noncontiguous access methods. Datatype I/O is unique in comparison with the other methods in that increasing the number of noncontiguous regions that are regularly occurring does not incur any additional I/O access pattern description data to be passed over the network. List I/O, for example, would have to pass more file offset and length pairs in such a case.

5.6. Noncontiguous I/O Implications. The various methods for accessing noncontiguous data have a profound impact on GRID I/O. As most GRID applications use high level interfaces, it is important to see how this affects noncontiguous I/O access. In Section 1, we examined the tile display application access pattern. In Figure 10 performance results, we see that there is quite a large gap in performance between the different I/O methods. In a FLASH I/O checkpoint benchmark, the performance gap between POSIX I/O and datatype I/O grows to over two orders of magnitude [30].

With a high level I/O library like NetCDF that makes regular, structured noncontiguous access, datatype I/O techniques as described in Section 5.5 are well suited. This allows the access pattern description to remain concise as it is passed down to the file system. Using datatype I/O for HDF5 and its hierarchical data layout would most likely map logically regular structured access patterns to irregular noncontiguous data access. Datatype I/O will not perform any better than list I/O when the access pattern is so irregular that it degrades into an Indexed datatype (similar to list I/O).

Since both the NetCDF and HDF file formats are self-describing (they define their data structures within the file), the unique possibility for data structure caching presents itself. With slightly more metadata labeling the data structures, applications can describe noncontiguous access patterns that are stored on disk with a unique ID. This would eliminate nearly all of the noncontiguous access pattern description from traveling over the network. Since self-describing file formats already store their data structure information in file, it would make sense to use these descriptions to reduce network traffic. We are currently working on implementing this I/O technique in the future, which we call *datatype caching*. It will likely have a significant impact for I/O on large-scale GRID applications.

In conclusion, we have discussed the various noncontiguous I/O methods and their performance impact. In this following two sections, we present two projects for future I/O systems: the Versioning Parallel File System (a work-in-progress) in Section 6 and large-scale cache management strategies in Section 7. We believe that these projects will greatly improve performance, increase fault-tolerance, and enable strict high-performance I/O semantics for future scientific computing needs.

6. The Versioning Parallel File System.

The growing use of parallel file systems to sustain scalable I/O for scientific computing has led to emerging performance problems in fault-tolerance, strict consistency semantics, and noncontiguous I/O access for large-scale GRID computing. In this section, we discuss a new parallel file system for large-scale clusters and GRID environments that will address these issues. In Section 6.1, we begin by discussing the challenges of future parallel file systems and how atomic noncontiguous I/O plays a large role in handling these problems. In Section 6.2, we describe how atomicity can be difficult to implement in the file system. In Section 6.3, we describe a protocol for implementation of VPFS. Finally in Section 6.4, we explain the advantages of VPFS over traditional atomic methods.

6.1. Fault-Tolerance, Strict Consistency, and Noncontiguous I/O. Future high-performance storage platforms must address the trends in scientific computing. First as discussed previously in Section 3, scientific datasets are rapidly growing in size. In particular, noncontiguous I/O methods must scale to larger data access sizes. System snapshots generated for visualization or checkpointing are expensive and scale in I/O cost as time resolution increases. In order to handle such larger I/O access patterns, storage systems must also scale up in size for capacity and perfor-

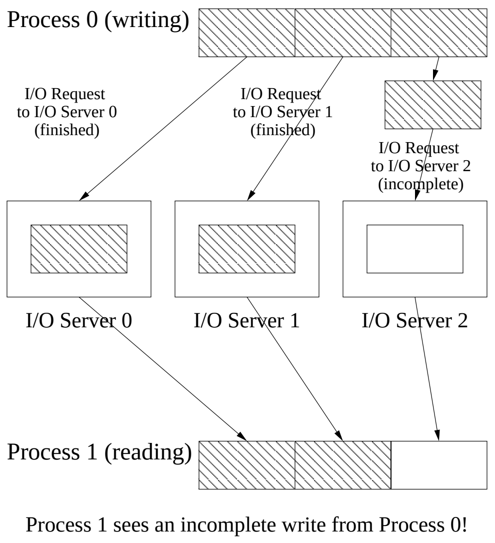
Process 0 (writing)
I/O Request to I/O Server 0 (finished)
I/O Request to I/O Server 1 (finished)
I/O Request to I/O Server 2 (incomplete)
I/O Server 0
I/O Server 1
I/O Server 2
Process 1 (reading)
Process 1 sees an incomplete write from Process 0!

FIGURE 11. Atomicity difficulties for parallel file systems.

mance reasons. While increasing the parallelism of storage systems can provide these additional performance capabilities, it also makes the overall storage system more fault prone. RAID parity techniques such as those described in [31] are often used to provide a degree of fault-tolerance and will help enable storage systems to scale in I/O bandwidth.

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

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

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

6.2. Atomicity For Parallel I/O. Ensuring atomicity of I/O operations is difficult for several reasons. First, noncontiguous I/O operations may be broken up by higher level libraries into multiple POSIX I/O operations as we discussed in Section 5.1. Secondly, since contiguous I/O operations are divided into multiple I/O opera-

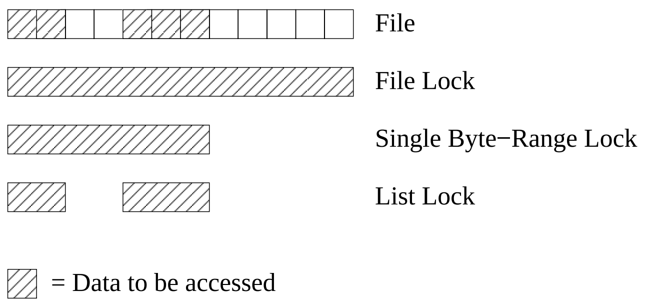

FIGURE 12. A variety of lock-based methods for atomic noncontiguous I/O.

tions for multiple I/O servers in a parallel file system as shown in Figure 11, interleaved read and write operations that are logically contiguous can produce non-atomic results. We begin our discussion on implementing atomicity through the traditional methods.

In past GRID I/O research, weak semantics, like those used in AFS and Gfarm (updates are only visible on close), have been used for performance reasons. A traditional hardware RAID controller would serialize I/O requests in order to ensure that they were not interleaved. Such a serialized solution is not practical for cluster computing as it would greatly degrade performance. In the past, locking has been the only solution for ensuring atomicity for parallel file systems.

A typical lock-based synchronization solution forces processes to acquire either read or write locks on a file region before an I/O operation. Read locks are shared, which means that multiple processes may be simultaneously granted read locks as long as no write lock regions intersect the read locked region. Write locks are exclusive. Only a single process may have a lock on a file region if it is a write lock. Locks provide the synchronization capabilities for atomic I/O since I/O operations are automatically serialized when an I/O access pattern overlap occurs. Locks may have different region size granularity. They may simply encompass an entire file or simply a byte-range. If the I/O access pattern is noncontiguous, a lock-based solution can acquire a single byte-range lock from the initial offset of the noncontiguous access pattern to the final byte accessed. It may alternatively acquire a group of byte-range locks that cover every file region accessed. We call this later solution *list lock*. List lock, to our knowledge, has not been implemented or benchmarked. One of the difficulties in implementing list lock is managing possibly thousands or millions of locks, which scale with the number of noncontiguous file regions accessed. The groups of locks must be acquired in ascending order in a two-phase method (such as in [32]) to ensure no I/O deadlock occurs. In Figure 12 we show the various methods of locking to ensure atomic noncontiguous I/O access.

While a lock-based synchronization solution can use list lock to allow concurrent I/O for non-overlapped I/O, overlapping I/O will always be at least partially serialized if any of the overlapping I/O operations is a write. Another problem with locking is that locks that are held by clients who fail must eventually be reclaimed. However, while the lock is held by a dead client, limited or no progress can occur to the locked regions. Byte region locks may require alignment boundaries, which leads to false sharing. For instance, the GPFS lock token granularity can be no smaller than one sector [8]. Finally lock-based synchronization can provide atomic I/O access only when the client does not fail. If a client fails during the middle of a write and the

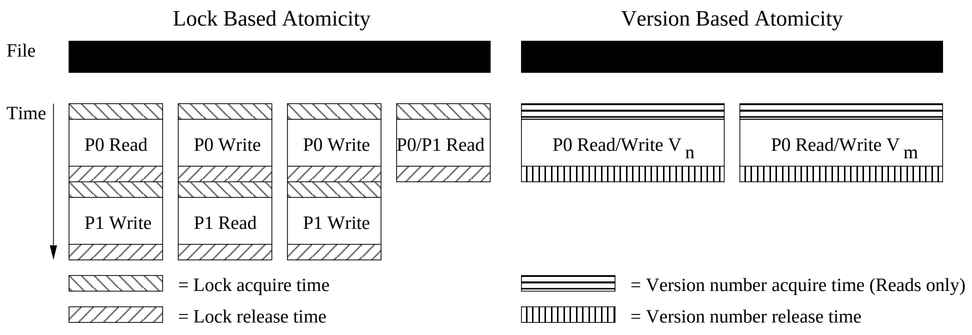

FIGURE 13. Serialization using locking versus concurrent I/O using versioning.

lock for that region is recalled, another client may later see the effects of a partial write, which violates atomicity. In order to provide atomicity while accounting for client failure, I/O writes must use data journaling or similar techniques to be able to undo the effects of the partially completed write.

6.3. VPFS Protocol. In order to efficiently provide high-performance atomic noncontiguous I/O access for scientific computing, we introduce the *Versioning Parallel File System*, or VPFS. As described in Section 6.2, locking techniques serialize I/O access in many cases and also create significant overhead. Instead of locking, VPFS uses a technique called *versioning* to handle performance issues for atomic noncontiguous I/O access.

Versioning in the file system is not a brand new technique. It has been used in various projects [33, 34, 35] to allow the file system to see multiple states of a file. The most common use of versioning in file systems is to provide users with the ability to recover deleted files as well as collaborate on large projects. We plan to apply the versioning technique to a parallel file system. This combination has tremendous potential beyond simply undoing and merging file changes. Our use of versioning in VPFS is simple. Every I/O operation which modifies data in the file system will create a "version", which is a tuple of information consisting of {*operation type, offset-length count, offset-length pairs, offset-length data*}. The operation type is the operation that this version represents (for example write, truncate, checkpoint, etc). The offset-length count is the number of offset-length pairs in the following list of offset-length pairs. The data for the offset-length pairs is written contiguously in the last version tuple component (which provides contiguous I/O performance for noncontiguous I/O operations). Each version tuple on an I/O server is labeled with a modify number n, which specifies an order of completion.

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

Clients, or C, are required to obtain an $ID\#$ before performing any I/O. When a client is about to perform I/O, it increments a counter $IO\#$. When $ID\#$ and $IO\#$ are concatenated, a unique temporary version number T has been created for a data modify operation. This temporary version number T will be used as a temporary number for modify operations that are still in the process of being completed. They will later be renamed by file version numbers in *Vmap* that are piggybacked along with data retrieve operations.

Version servers, or *Vs*, are required to keep track of a file's current version number (*Vf*). *Vf* is used to label I/O operations in the order in which they complete for modify operations (for example, write) and the order in which they are began for retrieve operations (for example, read). The version servers keep track of the *Vf* numbers used for retrieve operations that have not yet completed in a list called *Vr_list*. The minimum of *Vr_list* is *Vo*, the oldest retrieve operation still in progress. Version servers also give out *ID#* to clients when requested, incrementing *ID#* on each request to give each client a unique *ID#*. Modify operations are given file version numbers when they have fully completed. These mappings from temporary version numbers T created on the clients to file version numbers *Vf* are kept in *Vmap*. The *Vs* keeps track of *Vmap* as a list of tuples {*T, Vf*}. All of this versioning information (*Vf*, *Vr_list*, *Vo*, *ID#*, and *Vmap*) is kept on the *Vs* on a per file basis.

I/O servers, or *Is*, maintain the version tuples and a list of the versions in use (*Vuse_list*). They also execute version merging, which is a process where version tuples are read and combined into a single version for increasing free disk space.

We describe the basic VPFS processes of open, merge, read and write. The sync and close operations do not require any special consideration. For simplicity we intend to simply call merge when the either sync or close is called. More detailed explanation will be forthcoming with our implementation and performance results.

- Open - *C* requests an *ID#* from the proper *Vs* for file *F*. The *Vs* increments the *ID#* for the file *F* and returns the previous value of *ID#* to *C*. We note that the client's acquisition of *ID#* can happen at any time before a retrieve or modify I/O operation. For simplicity, we choose this to occur during a file open.
- Merge - *C* requests to begin a merge operation from the proper *Vs* for file *F*. *Vs* increments *Vf* and returns the old *Vf*, *Vo*, and *Vmap* to *C*. *Vs* adds the returned old *Vf* to the *Vr_list* and links the old *Vf* to the entries in *Vmap*. *C* sends *Vf*, *Vo*, and *Vmap* to all *Is*. The *Is* updates all temporary versions in *Vmap* to their final version number and adds them to *Vuse_list*. If the *Is* is not performing a merge for *Vo* already (or has done so in the past), it uses all the versions up to *Vo* in *Vuse_list* to create a new version *Vo* that is also added to *Vuse_list*. When it completes, *C* sends back *Vf* to *Vs*. *Vs* removes *Vf* from *Vr_list* and removes *Vmap* entries linked to *Vf*.
- Write - *C* increments *IO#* and concatenates its *ID#* with old *IO#* to form T. *C* sends T along with its write data to each relevant *Is*. The *Is* saves the version tuple with its temporary version number T. When *C* has completed writing all of its version tuples, it sends T to proper *Vs*. The *Vs* increments *Vf* and adds the old *Vf* as a tuple {*Vf, T*} to *Vmap*.
- Read - *C* requests to begin a read operation from the proper *Vs* for file *F*. *Vs* increments *Vf* and returns the old *Vf*, *Vo*, and *Vmap* to *C*. *Vs* adds the returned old *Vf* to the *Vr_list* and links the old *Vf* to the entries in *Vmap*. *C* sends *Vf*, *Vo*, and *Vmap* to all *Is*. The *Is* updates all temporary versions in *Vmap* to their final version number and adds them to *Vuse_list*. Then the *Is* uses all the versions up to *Vf* in *Vuse_list* to service the read request. If it desires, the *Is* may also perform a merge as described in the merge operation

and add the new version *Vo* to *Vuse_list*. When it completes the read, *C* sends back *Vf* to *Vs*. *Vs* removes *Vf* from *Vr_list* and removes *Vmap* entries linked to *Vf*.

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

Another strong advantage of the use of versions over locks is I/O operation concurrency. All reads and writes may continue in parallel even when they access overlapping regions. As we discussed in Section 6.2, lock-based systems can only allow concurrent access to overlapping regions if both processors are reading. VPFS allows atomic concurrent access to overlapping regions for any combination of reads and writes without any serializing penalties as shown in Figure 13. Also discussed earlier, atomicity of I/O operations is a key enabler of parity based RAID techniques in parallel file systems. For RAID 5, parity is maintained on a per stripe basis to ensure that any one I/O node may be lost while preserving all data and operating in a degraded mode until the failed node is replaced. In hardware RAID, parity is computed on every write operation by the hardware RAID controller. This method is not scalable or practical for parallel file systems. Since our versioning parallel file system can provide efficient atomic I/O operations, we can also perform atomic data and parity updates in parallel, even if writes are overlapping the same parity block. This is a large improvement over lock-based methods, since they always force serialization when writes have overlapping I/O parity access.

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

System snapshots are often dumped at intervals between scientific computational stages. This is done so that if a part of the system fails, the application can be restarted without losing much progress. System snapshots are also used for post-processing and/or visualization. Snapshots are I/O intensive and slow on traditional parallel file systems. The snapshot frequency is often determined by the time it takes

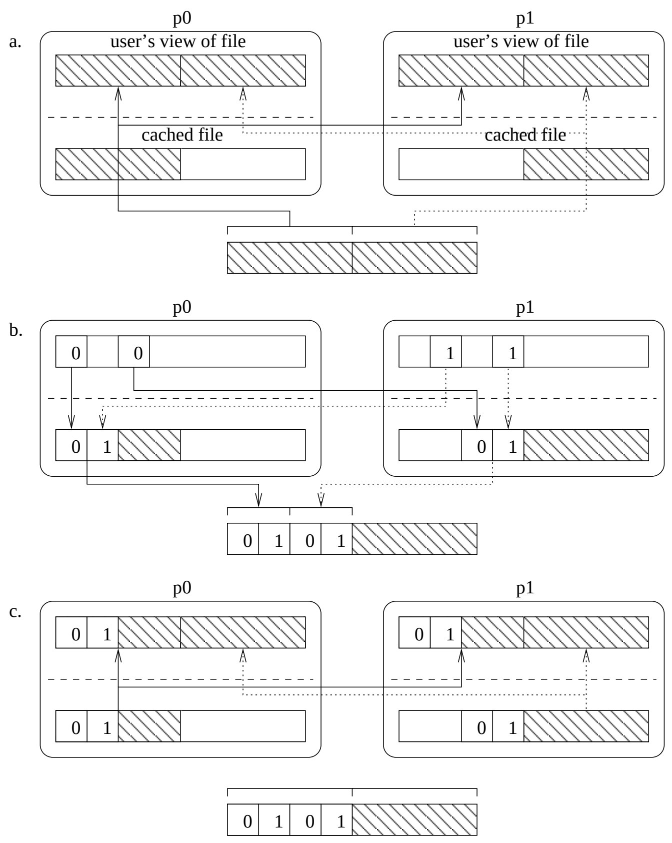

FIGURE 14. Cache inconsistencies in collective I/O

to write a snapshot. If snapshots are cheap and fast, they can be written often without significantly affecting system performance. Our versioning of files natively creates such snapshots. In fact, by simply tagging the version of the last write that we would like to include in the system snapshot to not be deleted when a merge operation occurs, we can ensure that the system snapshots remain in our I/O system without any I/O penalties. Such an optimization for system snapshots would provide a great tool for scientific computing (for example, reducing the cost of making a checkpoint in the ASC FLASH code). We also note that within the MPI-IO interface, we can perform several optimizations with VPFS. We would like to experiment on how to use versioning and calculate write parity on collective I/O operations. We expect that there are important optimizations when using a single version versus using multiple versions when collective I/O is scaled up to thousands of processors.

Some obvious drawbacks to VPFS include possible new read overheads. Since the read operation must look thorough multiple version tuples on each I/O server to be serviced, we expect that overall read performance will slight worsen, while write speeds will increase. Also, since we make version tuples for every modify operation, this can, in certain cases, significantly increase overall storage. We expect to further examine these potential problems as we continue our implementation. VPFS is a work-in-progress. We expect basic VPFS v1.0 to be completed before the end of the summer of 2005.

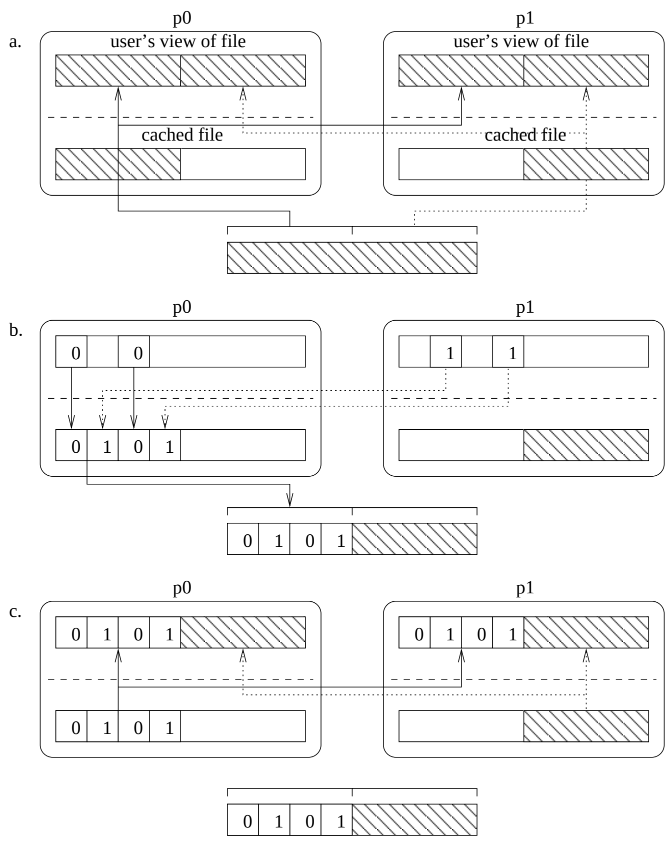

FIGURE 15. Cache consistency issues of collective I/O using persistent file domains.

7. Large-scale Cache Management Strategies. In typical distributed environments, caching is an integral piece of the I/O performance puzzle. Without client-side caching, nearly every I/O request and data transfer must come from some remote location and often disk, thus incurring the potentially large costs of network latency and bandwidth. While network performance may be expected to improve, its relative performance cannot compare with memory bandwidth, much less processor performance. Enforcement of strict semantics is rather expensive, driving most client-side caching schemes to use a relaxed set of consistency semantics. The choice of consistency semantics depends on an application's tolerance for transitory inconsistencies as well as the quality of the network between the client and server. Generally, an application needs to balance semantics and performance needs. In some specific cases, performance need not be sacrificed for stronger semantics, but they are typically regarded as inversely correlated.

A simple solution for ensuring cache consistency uses write locks on a file or some part thereof to signal other processes of an impending write to the file. Through such a lock system, a write lock invalidates the same file data cached elsewhere and ensures stale data is not read. Other semantics do not guarantee data availability until a file is closed.

7.1. Application Cache Coherency in Collective I/O. One way to avoid the complex problems involved in global system-wide cache consistency is to narrow one's scope. Within a single application, cache consistency can be addressed transparently at higher levels than the file system even when client-side caching is done in the file system as in NFS. While an application and an MPI implementation may be aware of complex I/O access patterns, this higher order information is lost by most file systems where client-side caching usually occurs. The MPI-IO interface is designed specifically to handle both complex data structures and to allow application processes to coordinate I/O. Two-phase I/O involves an I/O phase and a communication phase, the order of which depends on whether the operation is a read or write as described in Section 5.3. When performing collective I/O, application processes can divide the file up into distinct persistent file domains. Each process acts as a proxy I/O server for its file domain(s). In the read case, all accesses to a particular file domain are gathered at the file domain's "owner" process. The owner process then performs the read to its file domain on behalf of all the processes. In the second phase, the owner process then distributes the requested read data to each process. Figure 14 steps through several sequential collective I/O operations using the typical two-phase method where file domains are determined separately for each call based on the aggregate access pattern (across processes). In Figure 14a, p0 and p1 collectively read the entire file, each caching data from its own file domain. Later, in Figure 14b, both processes collectively write to the first half of the file, where the file domains split the first half of the file in two. After communicating the write data with each other, it is written, once again going through their own caches. When the entire file is reread in Figure 14c, using the same file domains as in Figure 14a, the stale second quarter of the file on p0 is read out and distributed to both processes. By keeping the file domains persistent across multiple collective reads and writes, only the predetermined owner of a file domain will accesses the domain either on disk or in system cache, thereby enforcing cache coherency in the application. Figure 15 demonstrates the persistent file domain solution and shows why it works. While the file system cache may not be directly manipulated, access can be organized in such a way to avoid any potentially stale data. Originally intended for high-performance computing, persistent file domains can be easily adapted for the GRID in setting file domain sizes and locality. Our evaluation of the persistent file domain solution in clusters was published in [36]. Ideally, the process responsible for a file domain should be somewhat near the processes requesting data from its file domain. Since high-performance computing is characterized by homogeneity, persistent file domains are usually assigned in some uniform manner. The heterogeneous nature of the GRID may make irregular file domain sizes and distributions more desirable. Assignment of file domains should take into account I/O access patterns, locality, as well as other resource parameters such as CPU load or network load. Since file domain distribution can no longer be derived deterministically, some global file domain map would need to be distributed among clients instead. Collective I/O may not always be appropriate for an application, and the collaborative nature of the GRID does not preclude the concurrent use of one file by multiple applications.

7.2. Application Cache Coherency in Independent I/O. A novel approach for handling independent I/O in MPI applications uses Remote Memory Access (RMA). Under current development, the Direct Access Cache (DAChe) System is an application level subsystem designed to manage client-side caching in a coherent manner. Our initial prototype will soon be published in [37]. Built on top of MPI, DAChe is quite portable, but actual RMA performance depends on the underlying communication layers. Systems without network cards that support RMA typically

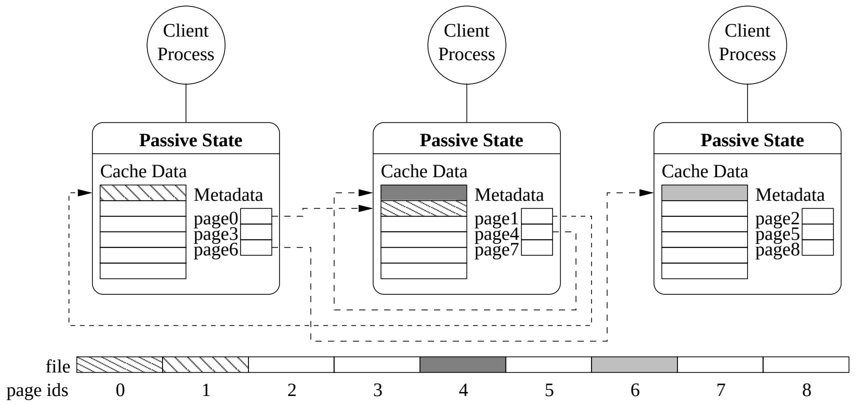

FIGURE 16. Each client process actively executes user code as well as accesses the passive server on any other node.

use threads instead. It could also be put into the file system layer, but it would lose its portability over different RMA systems. An advantage of the DAChe design is that it does not require extra dedicated processes. Each client process has three responsibilities: a passive metadata, lock, and cache server. Each client's cache is remotely accessible by any other client through RMA, though any file page is cached on at most one client. The latter point ensures cache coherence since all clients will have the same uniform view of the file. Currently, each client process handles its own local caching and evictions in an effort to reduce passive coordination complexity and improve locality. For strict sequential consistency there is a lock mechanism stored in the metadata for each file page. More importantly, metadata keeps track of where, on a per page basis, a file page is cached as in Figure 16. The metadata itself is distributed across client processes in a round robin manner, also globally accessible through RMA. Metadata operations must be atomic, and are therefore controlled by a mutual exclusion or lock mechanism. Most RMA interfaces provide either a high-level lock construct as in MPI or some simpler atomic operation like swap, increment, or test and set. Because the clients act as the DAChe servers, the number of DAChe servers automatically scales with the size of the application. The key is getting DAChe performance to scale with the job size. The primary drawback to DAChe in its present incarnation is that it is very much based on passive state, making it quite fault prone. As a user-level library, DAChe should still work on the GRID, but it could also be pushed down into a GRID file system. Fault tolerance issues need to be dealt with, the simplest solution to which would be some sort of state replication scheme. A more complex coherence scheme handling multiple read copies of data and a single write copy to allow for more aggressive caching may be appropriate for the GRID's wider dispersement of resources. Also beneficial would be automatic cache page migration for dynamic load-balancing and improved data locality.

Considering the shear number of potential clients on the GRID, the on-demand nature of their expectations, and their independence and dependence, enforcing some kind of global cache consistency across all the clients for all applications would be inefficient. Applications should surely be allowed to specify what type of consistency semantics are needed or expected at run time. Hierarchical adaptations of DAChe-style caching may be more appropriate for the GRID.

8. Conclusion. In this book chapter we have discussed many aspects of high-performance I/O in large-scale GRID environments. We have presented a summary of current and proceeding projects regarding GRID I/O as well as described how application I/O moves from high level I/O libraries such as NetCDF and HDF down to parallel file systems in the back-end. We described how our future work on the VPFS and large-scale cache management strategies will address issues of performance, reliability, and high-performance I/O semantics.

Acknowledgments. This research was supported in part by the Department of Energy laboratories, SNL, LLNL, and LANL under subcontract No. P028264, the National Science Foundation under subcontract EIA-0103023, and the DOE HPCSF Fellowship. The ASC FLASH software described in this work was in part developed by the DOE-supported ASC / Alliance Center for Astrophysical Thermonuclear Flashes at the University of Chicago

REFERENCES

[1] R. Oldfield and D. Kotz. Armada: A parallel file system for computational grids. in *Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid*, Brisbane, Australia, IEEE Computer Society (2001), p. 194–201.

[2] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, and S. Sekiguchi. Grid datafarm architecture for petascale data intensive computing. in *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, IEEE Computer Society (2002), p. 102–110.

[3] M. Beynon, R. Ferreira, T. M.Kurc, A. Sussman, and J. H. Saltz. Datacutter: Middleware for filtering very large scientific datasets on archival storage systems. in *IEEE Symposium on Mass Storage Systems*, IEEE Computer Society (2000), p. 119–134.

[4] M. Rodgriguez-Martinez and N. Roussopoulos. Mocha: a self-extensible database middleware system for distributed data sources, in *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, Dallas, Texas, ACM Press (2000), p. 213–224.

[5] J. Pérez, FĠarcia, J. Carretero, A. Calderón, and J. Fernández. A parallel I/O middleware to integrate heterogeneous storage resources on grids. in *1st European Across Grids Conference*, LNCS 2970, Springer (2004), p. 124–131.

[6] T. Baer and P. Wyckoff. A parallel i/o mechanism for distributed systems. in *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, IEEE Computer Society (2004).

[7] N. Karohis, B. Toonen, and I. Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. in *Journal of Parallel and Distributed Computing* (2003).

[8] F. Schmuck and R. Haskin. Gpfs: A shared-disk file system for large computing clusters. in *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, San Jose, CA, USENIX (2002).

[9] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, (1996), p. 84–92.

[10] P H. Carns, W B. Ligon III, R B. Ross, and R Thakur. PVFS: A parallel file system for Linux clusters. in *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, GA, (2000) p. 317–327.

[11] O. Tatebe, S. Sekiguchi, Y. Morita, N. Soda, and S. Matsuoka. Gfarm v2: A grid file system that supports high-performance distributed and parallel data computing. in *Proceedings of the 2004 Computing in High Energy and Nuclear Physics* (2004).

[12] S. Vazhkudai, S. Tuecke, and I. Foster. Replica selection in the globus data grid. in *Proceedings of the International Workshop on Data Models and Databases on Clusters and the Grid*, IEEE Computer Society (2001).

[13] Earth system grid (ESG). `http://www.earthsystemgrid.org`.

[14] D. Bernholdt, S. Bharathi, D. Brown, K. Chanchio, M. Chen, A. Chervenak, L. Cinquini, B. Drach, I. Foster, J. Garcia, C. Kesselman, R. Markel, D. Middleton, V. Nefedova, L. Pouchard, A. Shoshani, A. Sim, G. Strand, and D. Williams. The earth system grid:

Supporting the next generation of climate modeling research. in *IEEE Transactions*, (2005 Forthcoming).

[15] IPARS: integrated parallel accurate reservoir simulation. `http://www.ticam.utexas.edu/CSM/ACTI/ipars.html`.

[16] J. Saltz, U. Catalyurek, T. Kurc, M. Gray, S. Hastings, S. Langella, S. Narayanan, R. Martino, S. Bryant, M. Peszynska, M. Wheeler, A. Sussman, M. Beynon, C. Hansen, D. Stredney, and D. Sessanna. Driving scientific applications by data in distributed environments. in *Workshop on Dynamic Data-Driven Application Systems*, LNCS 2660, Springer (2003).

[17] M. L. Norman, J. Shalf, S. Levy, and G. Daues. Diving deep: Data-management and visualization strategies for adaptive mesh refinement simulations. *Computing in Science and Engg.*, (1999) p. 36–47.

[18] R. Ross, D. Nurmi, A. Cheng, and M. Zingale. A case study in application i/o on linux clusters. in *Proceedings of the 2001 Supercomputing Conference*, New York, NY, ACM Press (2001).

[19] R. Rew, G. Davis, S. Emmerson, and H. Davies. Netcdf user's guide for c. in *Uidata Program Center* (1997).

[20] R. Rew and G. Davis. The unidata netcdf: Software for scientific data access. in *Proceedings of the 6th International Conference on Interactive Information and Processing Systems for Meterology, Oceanography and Hydrology*, IEEE Computer Society (1990), p.76–82.

[21] J. Li, W. Liao, A. Chouhdary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Sigel, B. Gallagher, and M. Zingale. Parallel netcdf: A high-performance scientific i/o interface. in *Proceedings of Supercomputing 2003*, ACM Press (2003).

[22] Hdf5 home page. `http://hdf.ncsa.uiuc.edu/HDF5/`.

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

[24] E. Smirni, R. A. Aydt, A. A. Chien, and D. A. Reed. I/O requirements of scientific applications: An evolutionary view. in *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Syracuse, NY, IEEE Computer Society (1996), p. 49–59.

[25] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. Best. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, IEEE Computer Society (1996) p. 1075–1089.

[26] S. J. Baylor and C. E. Wu. Parallel I/O workload characteristics using Vesta. in *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, IEEE Computer Society (1995), p. 16–29.

[27] R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. in *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, Atlanta, GA, ACM Press (1991), p. 23–32.

[28] A. Ching, A. Choudhary, W. Liao, R. Ross, and W. Gropp. Noncontiguous I/O through PVFS. in *Proceedings of the 2002 IEEE International Conference on Cluster Computing*, IEEE Computer Society (2002).

[29] A. Ching, A. Choudhary, K. Coloma, W. Liao, R. Ross, and W. Gropp. Noncontiguous access through MPI-IO. in *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, IEEE Computer Society (2003).

[30] A. Ching, A. Choudhary, W. Liao, R. Ross, and W. Gropp. Efficient structured access in parallel file systems. in *Proceedings of the 2003 IEEE International Conference on Cluster Computing*, IEEE Computer Society (2003).

[31] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Chicago, IL, ACM Press (1998), p. 109–116.

[32] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*, Addison-Wesley (1987).

[33] D. Santry, M. Feeley, N. Hutchinson, A. Veitch, R. Carton, and J. Ofir. Deciding when to forget in the elephant file system. in *Proceedings of the 17th ACM Symposium on Operating System Principles*, ACM Press (1999).

[34] K. K. Muniswamy-Reddy, C. Wright, A. Himmer, and E. Zadok. A versatile and user-oriented versioning file system. in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, USENIX (2004).

[35] C. Soules, G. Goodson, J. Strunk, and G. Ganger. Metadata efficiency in a comprehensive versioning file system. in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, USENIX (2002).

[36] K. Coloma, A. Choudhary, W. Liao, L. Ward, E. Russell, and N. Pundit. Scalable high-level caching for parallel i/o. in *Proceedings of the 2004 International Parallel and Distributed Processing Symposium*, IEEE Computer Society (2004).

[37] K. Coloma, A. Choudhary, W. Liao, and L. Ward and S. Tideman. Dache: Direct access cache system for parallel i/o. in *Proceedings of the 2005 International Supercomputer Conference* (2005).