20 Parallel I/O and the Parallel Virtual File System

Walt Ligon and Rob Ross

Ever more frequently users of clusters find themselves in an interesting situation: it isn't the processors, communication network, or memory that is limiting their application; it is the storage system. This might force the users to checkpoint less frequently than they would like, might limit the resolution of output visualization data, or might prevent the use of out-of-core solutions needed for the largest of problems. What's worse, the I/O hardware in the system may indeed be adequate for the user's needs but may be being used ineffectively by one of the many software layers involved.

A lot of mystery surrounds I/O solutions in clusters today. For this reason we have rewritten this chapter in the second edition. We begin by covering what we believe are some of the most important issues in parallel I/O systems. These include parallel access patterns, parallel I/O system components and architectures, and consistency semantics. Knowing how parallel I/O systems operate and the issues involved can be useful when performance tuning an application for a particular system or choosing an I/O solution to match expected workloads. This material builds on material in many preceding chapters, including the I/O hardware discussion in Chapter **??**, the local and distributed file system discussion in Chapter **??**, and the network hardware discussion in Chapter **??**.

Following this more general discussion, we delve into PVFS, specifically covering some of the quirks of PVFS, management and tuning, and approaches for narrowing down the source of problems that may crop up. Finally, we discuss some critical issues for parallel file systems and how PVFS2, the next-generation parallel file system being developed by the PVFS team, attempts to address these.

These are very interesting times for parallel file systems on Linux clusters. As we are writing this chapter, the Lustre, PVFS2, and GPFS groups are all bringing new parallel file systems to the Linux cluster environment. The relative success of each of these is not likely to be known for quite some time, but we can certainly hope that at least one of these projects will result in a new, high-performance parallel file system designed to operate on systems with thousands of nodes (and,we hope, more!).

20.1 Parallel I/O Systems

What do we mean by a "parallel I/O system"? At a high level three characteristics are key:

- multiple hardware I/O resources on which data will be stored,

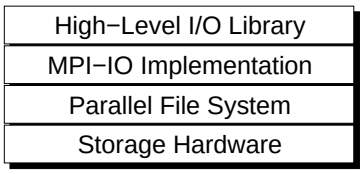

Figure 20.1: Parallel I/O System Components

- multiple connections between these I/O resources and compute resources, and
- high-performance, concurrent access to these I/O resources by numerous compute resources.

Parallel I/O systems get their performance by using multiple I/O resources that are connected to compute resources through multiple I/O paths. Multiple physical I/O devices and paths are required to ensure that the system has enough bandwidth to attain the performance desired. The hardware could consist of nodes with local disks attached via more traditional IP networks, a separate storage area network, or something else entirely; all of these are valid options for parallel I/O systems.

The third characteristic is easily as important as the first two but is considerably more difficult to pin down. Parallel I/O systems should be designed from the bottom up with the assumption that performance is a key attribute and that concurrent access to resources will be commonplace. This characteristic is heavily dependent on the software architecture; the software managing the hardware resources can make or break a parallel I/O system.

Often I/O systems that have multiple connections and hardware devices but don't cater to high-performance concurrent access are called *distributed* file systems. The software in these systems is tailored to other workloads. Chapter **??** discusses distributed file systems such as NFS.

A parallel file system is simply a component of a parallel I/O system that presents a hierarchical (file- and directory-based) view of data stored in the system. In the next section will see where this component fits into the big picture.

20.1.1 Components of a Parallel I/O Stack

A parallel I/O system includes both the hardware and a number of layers of software, as shown in Figure 20.1. While this chapter really focuses on parallel file systems and PVFS in particular, it is important to understand what other com-

ponents might be involved and how these work together to provide a reasonable solution to a tricky problem.

At the lowest level is the I/O hardware, described briefly in Chapter **??**. This layer comprises the disks, controllers, and interconnect across which data is moved. Obviously, this hardware determines the maximum raw bandwidth and the minimum latency of the system. The bisection bandwidth (defined in Chapter **??**) of the underlying I/O transport is an important measure for determining the possible aggregate bandwidth of the resulting parallel I/O system, just as it is an important measure for the communication network as seen in Chapter **??**. At the hardware level, data is usually accessed at the granularity of blocks, either physical disk blocks or logical blocks spread across multiple physical devices, such as in a RAID array.

Above the hardware is the parallel file system. The role of the parallel file system is to manage the data on the storage hardware, to present this data as a directory hierarchy, and to coordinate access to files and directories in a consistent manner. Later in this chapter we'll talk more about what "consistent manner" means, as this is an interesting topic in itself. At this layer the file system typically provides a UNIX-like interface allowing users to access contiguous regions of files. Additional low-level interfaces may also be provided by the file system for higher-performance access.

While some applications still choose to access I/O resources by using a UNIX-like interface, many parallel scientific applications instead choose to use higher-level interfaces. These higher-level interfaces allow for richer I/O description capabilities that enable application programmers to better describe to the underlying system how the application as a whole wants to access storage resources. Furthermore, these interfaces, especially high-level I/O interfaces, provide data abstractions that better match the way scientific applications view data.

Above the parallel file system layer sits the MPI-IO implementation. The MPI-IO interface [12], part of the MPI-2 interface specification, is the standard parallel I/O interface and exists on most parallel computing platforms today. The role of the MPI-IO implementation, in addition to simply providing the API, is to provide optimizations such as collective I/O that are more effectively implemented at this layer. In some sense the job of MPI-IO is to take accesses presented by the user and translate them, as best as possible, into accesses that can be performed efficiently on the underlying parallel file system. This makes the MPI-IO interface the ideal place to leverage file system-specific interfaces transparently to the user. The MPI-IO API is covered in Chapter **??**.

The MPI-IO interface is useful from a performance and portability standpoint, but the interface is relatively low level (basic types stored at offsets in a file), while

most scientific applications work with more structured data. For this reason many scientific applications choose to use a higher-level API written on top of MPI-IO (e.g., HDF5 [7] or Parallel netCDF [21]). This allows scientists to work with data sets in terms closer to those used in their applications, such as collections of multidimensional variables. These high-level interfaces often provide the same level of performance as using MPI-IO directly. However, one should be aware that in practice the implementation details of some of these systems do sometimes add significant overhead [29].

20.1.2 Access Patterns and Scientific Applications

Applications exhibit all sorts of different access patterns, and these patterns have a significant effect on overall I/O performance. The `cat` program, for example, accesses blocks of a file starting from beginning to end. This is the ideal pattern of access for most file systems, because many systems can identify this pattern and optimize for it, and the prefetching implemented in many I/O devices also matches with this well. This pattern is seen for a large number of applications, including video and audio streaming, copying of data files, and archiving.

Database systems use I/O resources as another level of memory. In doing so, they tend to access it in very large blocks (contiguous data regions) in an order that the I/O system cannot always predict. However, because the blocks are large and are aligned to match well with the underlying disks, this access pattern can also match well with the I/O system.

Studies tell us that the access patterns seen in scientific applications are significantly different from what we see in these other application domains. Scientific applications are in some sense worst-case scenarios for parallel I/O systems. One such study, the CHARISMA project [23], provides a great deal of insight into the patterns seen in scientific applications. We will extract some of the more important points here.

The CHARISMA project defines *sequential* access as a pattern where each subsequent access begins at a higher file offset than the point at which the previous access ended. Most of the write-only files were written sequentially by a single process. This behavior was likely because in many applications each process would write out its data to a separate file. This may have been an artifact of poor concurrent write performance on the studied platform. Read-only files were accessed sequentially as well, but regions were often skipped over by processes indicating that multiple processes were somehow dividing up the data. About a third of the files were accessed with a single request.

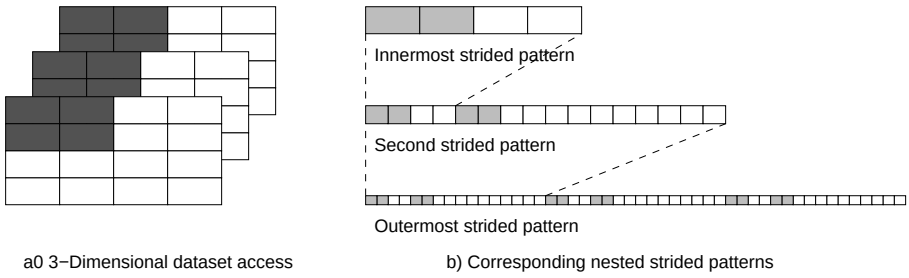

Figure 20.2: Nested-Strided Example

Figure 20.2 shows an example of a nested-strided access, in this case utilizing three strided patterns in order to access a block of a 3D data set. The study noted that strided access patterns were very common in these applications, with both simple (single) strides and nested strides present. A nested-strided pattern is simply the application of multiple simple-strided patterns, allowing the user to build more complex descriptions of stored data. These patterns arise from applications partitioning structured data such as multidimensional arrays. More recent studies, such as an analysis of the FLASH I/O benchmark [29], support these findings, although in this particular case the strided patterns occur in memory rather than in the file (which is written sequentially) and data from all processes is always written to a single file.

What does all this mean to us? First, it indicates that application programmers really can benefit from the descriptive capabilities available in high-level interfaces. Second, it suggests that the layers *below* these high-level interfaces should be capable of operating in terms of structured data as well. As we will see in the next section, some parallel file systems fall short in this area.

Because of the differences in access patterns between various applications, I/O solutions that work well for one application may perform poorly for another. This situation encourages us to consider using multiple file systems in the same cluster to fill particular roles. For example, a very reliable distributed file system that might not handle concurrent writes well could be a very useful file system for storing home directories in a large cluster. For smaller clusters NFS might fill this role. On the other hand, a very fast parallel file system with no fault tolerance capabilities might be perfect for storing application data used at run time that is backed up elsewhere. With this in mind, we will now discuss some typical parallel file system architectures with specific examples.

20.2 Parallel File System Architectures

Numerous parallel I/O systems have been built, although few have seen wide use. If we look at these file systems we do see trends in how the systems are designed. The architecture of these systems, both hardware and software, can have a significant effect on application performance, particularly with the demanding access characteristics of scientific applications.

We discuss the two most common architectures, including the components of the systems and some of the key characteristics. These architectures serve as a starting point for discussion of specific parallel file systems. For each architecture we give three example file systems, all of which have run on Linux at one time or another, and all of which have had an impact on parallel I/O systems of today.

20.2.1 Shared Storage Architectures

File systems relying on a shared storage architecture are the most popular approach for large production systems. The reason rests at least in part on the popularity of storage area networks (SANs) and fibre channel (FC) storage. File systems using shared storage have the common feature of accessing block devices remotely, either through direct attach hardware (such as FC disks) or through intermediate I/O servers that provide a block-oriented interface to disks attached locally to the server. In either case, a key component of these systems is a locking subsystem. The locking subsystem is necessary to coordinate access to these shared resources. While we will not discuss the issue of fault tolerance with respect to the locking subsystems of the example file systems, we note that a significant amount of effort has been put forth to ensure that locks can be recovered from failed nodes. This is a complicated problem, and the cited works discuss the issues in detail.

Some file systems that use shared storage implement a "virtual block device" in order to separate the access of logical blocks of data from their physical representation on storage. This virtual block device provides a mapping from logical blocks to physical storage. A file system component builds on this to provide the directory hierarchy for the file system, just as a local file system builds on a disk or RAID volume. This approach is advantageous from a system management point of view. The virtual block device, because it abstracts away physical data location, can provide facilities for data migration and replication transparent to the upper layers of the system. This approach simplifies the implementation of the upper level components. Further, this virtual block device provides a mechanism for adding and removing hardware while the system runs. Data blocks can be migrated off a

device before removal and can later be moved onto a newly installed device. This capability is very valuable in systems that must provide high availability.

The abstraction is, however, limiting in some ways as well. First, all file system accesses must be translated to block accesses before hitting this component. Because scientific applications often have noncontiguous access patterns, this approach can result in read/modify/write patterns that could have been avoided if more fine-grained accesses were allowed. Second, control over physical data locations is lost to the upper layers. While few scientific applications currently try to perform careful block placement for performance reasons, this could be an issue as groups attempt to further push the boundaries of I/O performance. Finally, this additional level of indirection adds overhead in the system, increasing the latency of operations.

A number of systems are available with this architecture. The first two example systems that we cover, Frangipani and GFS, rely on virtual block devices. The last, GPFS, uses a slightly different organization. SGI's CXFS file system, not discussed here, has a similar architecture to GFS.

Frangipani and Petal

The Frangipani and Petal systems, originally developed at Digital Equipment Corporation (DEC), together form a good example of the virtual block device approach. The Petal [20] component implements a virtual block device with replication, snapshotting, and hot swapping of devices. It presents a simple RPC-like API for atomically reading and writing blocks that higher-level components can use to build a file system. The Petal component runs on nodes that have attached storage. Instances of the Petal component communicate to manage these devices, as not all Petal instances can directly access all the devices that make up the virtual block device.

The Frangipani [37] component implements a distributed file system on top of Petal. A distributed locking component is used by Frangipani to manage consistency. Locks are multiple-reader, single-writer and are granted on a per file basis. Locks are "sticky"; clients hold onto locks until asked to release them by the locking subsystem, allowing for read and write caching at the client side. The Frangipani component runs on nodes that access the shared storage region. Instances of the Frangipani component do not communicate with each other, only the locking component and Petal.

This architecture provides us with a good opportunity to introduce a common term in distributed file systems. A system can be considered a *serverless* distributed

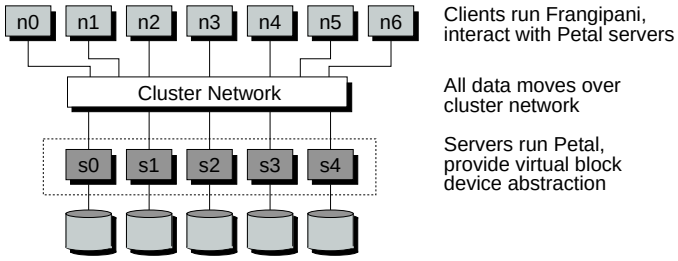

Figure 20.3: Frangipani and Petal File System Architecture

file system if nodes work together as peers to provide a shared storage region, as opposed to some specific server or servers providing this functionality [1]. When the term was coined back in the mid-1990s, systems weren't particularly large (the referenced paper tested on 32 nodes), and the point was really to distribute both metadata and data across multiple nodes more than to actually use every node as a storage resource.

In any case, it's easy to imagine that the Frangipani and Petal approach could be used in this "serverless" mode with Petal running on all clients, or it could be used in a system with a collection of heavy-duty servers with RAID arrays running Petal, with most nodes running only Frangipani. Without knowing more about a particular architecture, it's not clear which of these would be the right choice.

Frangipani and Petal are an early and well-documented example of this architecture. The Frangipani and Petal code is still around, although obtaining it seems difficult. At the time of writing rumor was that the code has been ported to Linux 2.4 and is floating around at one of the major hardware vendors. Perhaps it will pop up again to compete with some of the currently available systems.

GFS

The Global File System (GFS) was originally developed at Minnesota and is now developed and supported by Sistina [26, 25]. GFS is actively maintained and improved by Sistina. An older version of the source code, originally released under the GPL, is also available under the name OpenGFS. GFS also uses a virtual block device architecture, in this case using LVM (Logical Volume Manager) underneath the GFS file system layer.

GFS currently uses a "Pool" driver to organize storage devices into a logical space. They are investigating the use of LVM [14], a newer system for organizing

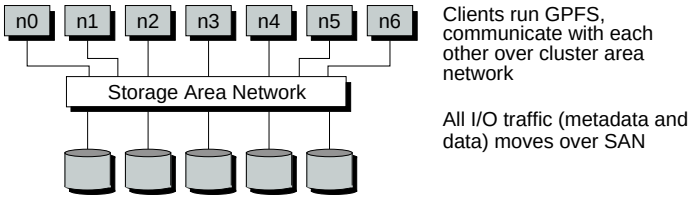

Figure 20.4: GPFS Architecture Using Storage Area Network

multiple physical storage devices into "volume groups" and then partitioning these into "logical volumes," which are the virtual equivalent of partitions on a disk. Just as with Petal, the Pool driver (and eventually LVM) provides capabilities for snapshotting and hot swapping of devices. The typical installation of GFS uses some number of nodes connected to shared fibre channel storage, with all nodes running both the LVM and GFS software (making it serverless). Alternatively a GNBD component can be used to provide remote access to a storage device over IP. This is similar to the VSD component in GPFS, which will be discussed in the next section.

GFS stores data as blocks on this virtual block device. A locking subsystem, OmniLock, provides the locking infrastructure necessary to ensure consistency. A number of locking modules are available with OmniLock allowing the locking granularity to be tuned to match expected workloads. Locks are sticky here as well, again allowing for read and write caching of data at the client.

GPFS

The General Parallel File System (GPFS) from IBM grew out of the Tiger Shark multimedia file system [31] and has been widely used on the AIX platform. Unlike the other file systems described, GPFS has no explicit virtual block device component. Instead GPFS simply uses one of two techniques for accessing block devices remotely and manages these devices itself. IBM's Virtual Shared Disk (VSD) component allows storage devices attached to multiple I/O nodes to be accessed remotely. VSD is different from the previous two approaches in that no logical volume management is performed at this level; it just exports an API to allow access to the devices. Alternatively, the VSD component can be avoided by attaching all nodes that wish to access the system to a SAN that gives them direct access to storage devices (Figure 20.4). This can be an expensive solution for large clusters, thus the existence of the VSD component. The newer Linux version of GPFS uses

a similar component, called the Network Shared Disk (NSD), to provide remote access to storage devices.

In either case, GPFS operates on a shared storage region using block accesses. Because there is no volume management, however, GPFS sees multiple devices. This approach was a conscious decision on the part of the developers to provide the file system with direct control over striping of data across devices. A side effect of this decision is that volume management and fault tolerance capabilities must be handled outside of the VSD, either below the VSD or in GPFS. RAID devices can be used below the VSD layer (or directly attached via the SAN). In addition to or in place of RAID, GPFS also supports data and metadata replication at the file system layer. If this capability is enabled, GPFS will allocate space for a copy of data on a different disk and keep copies synchronized. In the event of a temporary failure, GPFS will continue to operate and will update the device when it is returned to service. Likewise, functionality for migrating data onto new devices or off bad ones is also implemented within GPFS.

GPFS relies on a distributed locking component to guarantee consistency. Similarly to the other two systems, locks are acquired and kept by clients who then cache data. The granularity of locking in GPFS is at the byte-range level (actually rounded to data blocks), so writes to nonoverlapping data blocks of the same file can proceed concurrently.

GPFS provides as an alternative a consistency management system called *data shipping*. This mode disables the byte-range locks described above. Instead nodes become responsible for particular data blocks, and clients forward data belonging in these blocks to the appropriate node for writing. This approach is similar to the *two-phase I/O* approach often applied to collective I/O operations [33]. It is more effective than the default locking approach when fine-grained sharing is present, and it forms a building-block optimization for MPI-IO implementations.

The GPFS system also recognizes metadata blocks as distinct from data blocks. A single node that is accessing a file is given responsibility for metadata updates for that file. A multiple-reader, multiple-writer system then is applied to metadata that allows concurrent updates in many circumstances.

GPFS is arguably the most successful parallel file system to date. It is in use on a variety of large parallel machines, such as ASCI White, a 512-node Power3-based system. We note that only 16 I/O server nodes (running VSD) are used in that particular instantiation. At this time GPFS has been made available in a limited fashion on IA32 and IA64 Linux systems but has not seen widespread use on these platforms.

20.2.2 Intelligent Server Architectures

The second common approach to parallel file systems is the use of "intelligent" I/O servers. By this we mean that the servers do more than simply export a block-oriented interface to local storage devices. These systems usually communicate with clients in terms of higher-level constructs, such as files (or parts of files) and directories. Specific operations to act on metadata atomically might be included as well, rather than treating them as data operations as in the previous systems. Further these servers have knowledge that the data they are storing corresponds to particular file system entities (e.g., files or directories), not just arbitrary blocks on a storage device. Hence they have the potential to accept more complex, structured requests than are possible with other approaches. This is a particularly useful capability for scientific applications given their structured file accesses.

Designers of systems using this architecture often logically separate the storage of metadata from the storage of file data. This approach allows for flexibility in configuration because they can choose to handle metadata operations with different servers from the I/O traffic. Because providing distributed metadata services is more complicated than placing metadata in a single location, some systems support only a single metadata server while maintaining many I/O servers. On the other hand, using a single metadata server adds a potential bottleneck, so some systems distribute metadata across multiple servers, possibly even all the I/O servers. We will see examples of both of these approaches in upcoming sections.

Groups have been implementing parallel file systems using this approach for quite some time as well. Two of these systems are the Galley parallel file system and the Parallel Virtual File System (PVFS). An emerging parallel file system, Lustre, also has this type of architecture.

Galley

The Galley parallel file system [22] was developed at Dartmouth College in the mid-1990s (Figure 20.5). It was a research file system designed to investigate file structures, application interfaces, and data transfer ordering for parallel I/O systems. As such many things that we expect from a production file system were never implemented, including kernel modules to allow mounting of Galley file systems and administrative tools.

Galley breaks user's files into *subfiles*, which are stored on Galley servers. These subfiles have *forks* that allow for multiple byte streams to be associated with a particular subfile as well and can be used for more complex storage organizations. The client-side code handles placement of file data into appropriate subfiles and

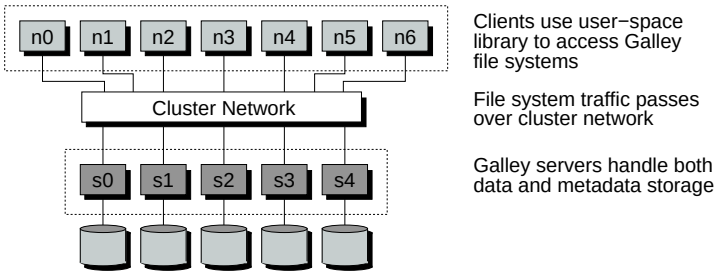

Figure 20.5: Galley Architecture

forks. Metadata is also stored on all the Galley servers. File names are hashed to find a server on which to store data (a technique also used by the Vesta parallel file system [10], which we will not cover in detail here).

Galley servers understand strided and batch accesses, making the interface quite rich. Many of the application access patterns seen in the CHARISMA study, as well as the patterns seen in the Flash I/O study, could be described with Galley's I/O language as single accesses.

Galley also implements *disk-directed I/O* [19], a method for organizing how data is moved between client and server. In disk-directed I/O, the server calculates a preferable ordering of data transfer based on predicted disk access costs. This ordering is then used when moving data. The method worked well for many access patterns, although the designers of Galley did see low performance due to network flow control problems in some cases. Later work showed that a more general approach of optimizing for the bottleneck resource can be more effective [30].

While Galley never made it into production, it is an excellent example of the intelligent server approach. Further, many of the ideas embodied in this design, in particular rich I/O request capabilities and more complex file representations, are becoming key components of new parallel file system designs. The Galley source code is available online [11].

PVFS

The Parallel Virtual File System (PVFS) [6] was originally developed at Clemson University by the authors of this chapter, starting in the mid-1990s, and is now a joint project between Clemson University and the Mathematics and Computer

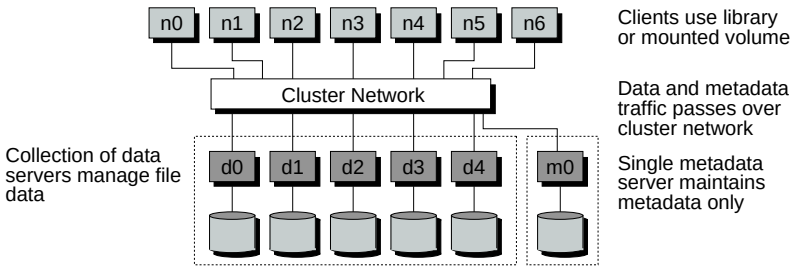

Figure 20.6: PVFS File System Architecture

Science Division at Argonne National Laboratory. PVFS is designed to be used as a high-performance scratch space for parallel applications.

PVFS file systems are maintained by two types of servers (Figure 20.6). A single metadata server, typically called the "mgr" because of the name of the daemon that runs on this server, maintains metadata for all files. For many workloads and configurations this is not seen as a bottleneck, although it is increasingly becoming one as systems grow in numbers of nodes. Separate I/O servers handle storage of file data. File data is distributed in a round-robin fashion across some set of I/O servers using a user-defined stripe size. Thus a simple algorithm can be used to determine the I/O server holding a particular file region. This simplifies the metadata stored on the metadata server and eliminates the need for metadata updates as files are written. I/O servers write to local file systems, so local disk management is managed by the local file system. Likewise, single disk failures can be tolerated by using a RAID to store local file system data at the I/O server.

PVFS uses what the authors term *stream-based I/O* for data movement. PVFS transfers data using TCP, and the stream-based I/O technique leverages this by predefining a data ordering and eliminating all control messages in the data stream. This approach is able to attain very high utilization of TCP bandwidth; however, in many cases PVFS is disk bound, not network bound. The more adaptive approach given in [30] would likely provide better overall performance, but it was not merged into the PVFS source.

PVFS implements simple strided operations. These can be useful for some patterns; however, a more general approach is necessary for implementing MPI-IO operations. More recently a more flexible (but less concise) system was added for accepting arbitrary lists of I/O regions as single operations [9]. Called List I/O, this was first proposed in [35] and has been shown to be of great benefit to some

access patterns. Support is provided in ROMIO for leveraging this; the hint to enable this is described in Section 20.4.4.

PVFS has no locking component. Instead, the metadata server supplies atomic metadata operations, eliminating the need for locking when performing metadata operations. Data operations are guaranteed by I/O servers to be consistent for concurrent writes that do not overlap at the byte granularity, but byte-overlapping concurrent writes result in undefined file state. This approach allows for a relatively simple system with no file system state held at clients, but it precludes client-side caching, which makes for very poor performance in a number of cases, particularly uniprocess workloads where systems from the preceding section would perform well.

Further, PVFS does not implement any form of fault tolerance. RAID can be used to tolerate disk failures, but node failures cause the system to be at least temporarily unusable. High-availability (HA) software is being investigated as a solution to this problem.

PVFS is also missing many of the administrative features that file systems such as GPFS offer. This limitation, combined with the lack of fault tolerance, has dissuaded many sites from using PVFS.

Nevertheless, PVFS has made it into production use at a number of sites around the world, mainly as a large, shared scratch space. PVFS is actively developed and supported, and the source for the file system, now commonly referred to as PVFS1 by the developers, is freely available online [27]. Because of its easy installation and source availability, many I/O researchers have chosen to compare their work to PVFS or to use PVFS as the starting point for their own research. We couldn't be happier that so many people have found this work to be so useful!

PVFS1 is showing its age, and a new version is under development to replace it before typical systems scale beyond its capabilities. We discuss this version, PVFS2, later in the chapter.

Lustre

The Lustre file system [2] is being developed by Cluster File Systems. At the time of writing the Lustre file system is under development, but much documentation and early code is available. The Lustre design benefits heavily from previous work in parallel file systems.

One of the key features of Lustre is the use of modules connected by well-defined APIs. This is seen in at least three areas: networking, allowing for multiple underlying transports; metadata storage, allowing for multiple underlying metadata targets; and object (data) storage, allowing for caching and multiple underlying data

storage technologies. In the latter two cases modules can be stacked to implement additional functionality. This provides great potential for the reuse of significant portions of the code when porting to new platforms or adding support for new hardware. Lustre uses the Portals API [4] for request processing and data transfer. Portals is a full-featured, reliable transfer layer designed for use in large-scale systems over multiple underlying network technologies.

Lustre breaks the nodes of the system into three types: clients, Object Storage Targets (OSTs), and Metadata Servers (MDSs). Object Storage Targets store objects, similar to inodes, which hold file data. OSTs perform their own block allocation, simplifying the metadata for a file in a manner similar to previous systems [6]. Objects can be stored on a number of back-end resources attached to OSTs, including using raw file system inodes. Alternatively data can be stored on more traditional SAN resources. In this case OSTs would still be in place, but would handle only authentication and block allocation, allowing data to be transferred directly between clients and SAN storage devices. This is similar to the GPFS approach when the VSD component is not used. This configuration could be convenient for sites with a SAN already in place.

Metadata servers store attributes and directory hierarchy information that is used to build the name space for the file system. Lustre's design calls for multiple MDS nodes in order to help balance the load on these systems. The protocol for metadata operations is explicit and transaction based, allowing for the avoidance of locks. An option is provided for using a node as both a MDS and as an OST.

A snapshot capability is also provided in Lustre, similar to the approach seen in [20], except that snapshotting is performed on object volumes (collections of objects) rather than a collection of blocks.

The designers of Lustre also propose a collaborative caching capability, where caching servers aggregate accesses to particular objects so that a single cache can be shared by multiple applications. This is similar in some ways to the data shipping scheme used in GPFS and distributed caches seen in research parallel file systems [16, 39]. However it is of particular note that Lustre is able to provide this functionality in a modular way.

Lustre relies on a distributed locking system for data coherence. Locks are available at different granularity levels to allow for concurrent access to disjoint file regions. Locks are managed by the OST that stores the object. Metadata operations are also performed by using locks to allow for client-side caching. Lustre adds *intent* locks for use in metadata operations. These are special locks that are used to perform some type of atomic operation at lock time. While in many instances an explicit operation to perform the intent could be used instead, this approach may

lead to fewer opportunities for races between atomic operations by immediately returning a lock that could be used for a subsequent operation.

Lustre implements full POSIX semantics, but this can be turned off on a per file or per file system basis. An interface similar to the List I/O interface described in [35] is proposed as an optimization as well.

Beta versions of Lustre are available, and development is very active. Also released under the GPL license, Lustre could become the next widely used parallel file system for Linux clusters; license compatibility with the PVFS2 project means that the two projects could share components if appropriate APIs were developed.

20.3 File System Access Semantics

From the user's point of view, two aspects of the file system API should be considered: what types of accesses can be described to the file system, and what happens when multiple processes access a file at the same time. We discussed earlier the importance of structured access descriptions; in this section we will focus on the second aspect, concurrent access semantics.

20.3.1 POSIX I/O Semantics

The most significant barrier to scalable parallel I/O systems today is the POSIX I/O interface and its associated semantics.

The POSIX I/O interface [17] was specified with local file systems in mind. The POSIX I/O interface specifies the `open`, `close`, `read`, `write` interface that we are all accustomed to using. It further specifies that writes through this interface must be performed in a sequentially consistent manner. Writes to the file must also appear as atomic operations to any readers that access the file during the write; the reader will see either all or none of any write. These semantics apply to any processes that access the file from any location.

Internal to a single system, the disadvantages of the POSIX semantics are not so apparent. In the single system, all operations to a file will pass through to a single device, and locks can be used to efficiently manage atomic access to files. However, the semantics of the POSIX interface have broad-reaching implications on any type of distributed or parallel file storage. In stark contrast to the single system, now we have multiple devices that might be accessed by any single operation, and all these devices, plus the clients, are distributed across some type of network. In this situation maintaining the POSIX semantics can be a complicated

and communication-intensive process, particularly when many processes access the same resources.

POSIX I/O and Locking

The most common approach to providing these semantics is to use a locking subsystem to manage access to files, and this is in fact the approach applied in all our example systems that implement the POSIX semantics (Frangipani, GFS, GPFS, and Lustre). POSIX semantics require that all accesses be atomic operations. When implemented with locks, this means that before a process can write to a region of a file, it must obtain the lock associated with that region. It can then write, then release the lock. Sophisticated lock caching and forwarding are used to alleviate the overhead of the locking subsystem in systems that expect to see a high degree of concurrent access.

Locks may be applied at the block, file, or extent granularity. The most coarse grained of these is file-based locks. File-based locks associate a single lock with an entire file. No distributed file system employing file-based locks should be seriously considered as part of a parallel I/O system because the contention for locks during concurrent access will ruin the performance of all but the least I/O-bound problems.

The second most coarse grained is block-based locks. This approach is often used in systems that use block-based accesses when communicating between clients and the underlying storage. Block-based locks have the advantage of being much finer grained than are file based locks. For large files, however, this approach can result in a very large number of locks being present in the system. Often these file systems address this by simply increasing the size of blocks. This, however, results in a situation where false sharing of blocks is more likely to occur.

The third, and most flexible, locking approach is extent-based locks. This approach can result in fewer locks in use because large ranges may be described as a single extent. This advantage is lost; however, if accesses are interleaved at a fine granularity. This approach, when coupled with noncontiguous access, can also result in a very large number of locks being processed in the system. Even with these two disadvantages this is the best locking approach for concurrent access under POSIX in use in parallel file systems today.

Scientific access patterns have a great deal of regularity. None of this information is retained in any of these locking approaches, however, leading to all these approaches being relatively inefficient, either in number of locks or in contention for a small number of locks. Approaches like IBM's data shipping can certainly help

make lock approaches perform more effectively, especially when accesses are interleaved. We will discuss the similar two-phase I/O approach later in Section 20.4.4.

From this discussion, and the presence of optimizations such as data shipping, it should be clear that the POSIX semantics are known in the community to be a problem. In fact, this problem is very similar to those seen in distributed shared memory (DSM) systems, where hardware and software are used to build globally accessible memory regions [38, 15]. The DSM community has for the most part abandoned the sequential consistency model in favor of more relaxed consistency models, in large part because of the overhead of maintaining such a model as systems scale. Perhaps it is time for the I/O community to follow suit.

20.3.2 NFS Semantics

The Network File System (NFS) protocol [24, 5, 32] is probably the most popular means for accessing remote file systems. Typically, remote file systems are "mounted" via NFS and accessed through the Linux virtual file system (VFS) layer just as local file systems are. What many users don't understand is that these NFS-mounted file systems do not provide the POSIX consistency semantics! The NFS version 3 RFC notes [5]:

> The NFS version 3 protocol does not define a policy for caching on the client or server. In particular, there is no support for strict cache consistency between a client and server, nor between different clients.

The story is a little more complicated for NFS version 4, but the lack of cache consistency on the client side remains.

NFS is an everyday example of relaxing the POSIX I/O consistency semantics in order to gain performance. NFS clients cache file data, checking every now and again to see whether the file has changed. This loosely synchronous consistency model makes for convenient, low-latency access to one's home directory stored on a remote system. Further, the locking systems typically used to implement the POSIX semantics are avoided along with their overheads.

On the other hand, NFS semantics are nearly useless from a parallel computing point of view. Clients can cache data indiscriminately, and tend to do so at arbitrary block boundaries. This causes unexpected results when nearby regions are written by processes on different clients; if two processes concurrently write to the same block on different processes, even if they write to different bytes, the result is undefined! Figure 20.7 shows an example of how this happens. Two nodes have cached the same block, and processes have written to different parts. First one

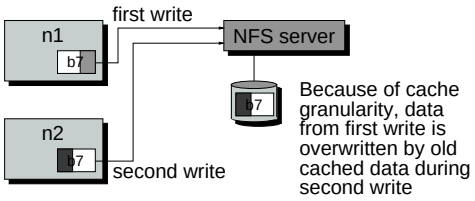

Figure 20.7: Concurrent Writes and NFS

block is committed back to storage, then the second. Because of the blocking, and the lack of consistency control, the data from the first write is lost when the second write completes.

Nevertheless, the semantics implemented by most NFS clients are sufficient to provide a usable file system for a number of situations.

20.3.3 MPI-IO Semantics

One could argue that the POSIX semantics are stricter than necessary for use in parallel I/O in that they force I/O systems to implement more consistency control than applications really need. Do scientific application programmers typically write to overlapping regions and let the file system sort it out? Probably not; they have better things to do with the I/O bandwidth! On the other hand, NFS semantics are definitely too loose; the nondeterminism introduced by uncoordinated client-side caching makes NFS semantics troublesome for concurrent writes.

The MPI-IO semantics [12] provide a very precise, but less strict, set of consistency semantics. The *Using MPI-2* [13] book provides a very thorough description of these semantics; they are actually relatively complicated. We touch on the semantics for some common cases here.

First, the scope of the MPI-IO semantics is the MPI communicator used to open the file. MPI says nothing about the semantics of access from different communicators, leaving this coordination to the application programmer. Second, by default MPI-IO does guarantee that concurrent *nonoverlapping* writes will be written correctly (unlike NFS) and that the changes will be immediately visible *to the writing process*. These changes are not visible by other processes in the communicator right away. Instead, explicit synchronization between the processes is necessary. This can be accomplished in a number of ways, all outlined in [13]. Simply closing and reopening the file is one method of synchronization, and the use of explicit file synchronization operations is another.

This model makes a lot of sense for many access modes seen in parallel applications, including checkpointing and of course all read-only modes. More importantly it relaxes the requirements on the underlying I/O components significantly and provides many opportunities for optimization within the MPI-IO implementation. We will discuss two such optimizations later in this chapter in the context of using ROMIO with PVFS.

20.3.4 PVFS Semantics

Noting the increased system complexity and potential overhead in implementing full POSIX I/O semantics (and having limited resources!), the PVFS developers chose to implement a different set of I/O semantics. With PVFS, concurrent nonoverlapping writes are written correctly and are immediately visible to all processes. Note that this approach is stronger than the default MPI-IO semantics. Overlapping writes will leave some undefined combination of the written data in the overlapping file region, and reads that occur concurrently with writes may see pieces of old and new data.

These semantics are adequate for implementing most of MPI-IO and are more than adequate for most access methods while simultaneously simplifying the system significantly: no coordination is needed at write time between clients or servers. The result is a more scalable system, at the cost of POSIX semantics.

20.4 Using PVFS

In the previous edition of this book, the majority of this chapter was dedicated to the specifics of PVFS configuration and use. This information is all available at the PVFS Web site [27], in particular in the User's Guide [28]. Rather than rehash that document, we'll talk a little bit about practical aspects of using PVFS, including implications of the PVFS design on certain types of operations, managing and tuning PVFS file systems, using ROMIO with PVFS, and bug spotting. We hope that this information supplements the online documentation nicely. Section 20.5 describes PVFS2, the next generation of PVFS, which addresses many of the design limitations of PVFS.

20.4.1 Implications of the PVFS Design

The preceding sections have prepared us to discuss the implications of the PVFS design from a practical standpoint. First, PVFS does not perform client-side caching for metadata. Hence, all metadata operations have to travel across the network to

the metadata server. For heavy metadata workloads, this design can cause sluggish performance.

Additionally, PVFS does not keep a file size as part of the metadata stored at the metadata server; rather, it calculates this value when it is requested. The advantage is that, during writes, the metadata need not be updated. However, a `stat` on a file requires not only a message to the metadata server to obtain the static metadata but also a sequence of messages to the I/O servers (performed by the metadata server) in order to obtain the partial sizes necessary to fill in the file size. The `ls` program performs this operation on *every* file in a listed directory, which can cause `ls` to be very slow for PVFS file systems. In practice, this makes PVFS a poor performer for small files, too, because users tend to put all the small files in one directory. Then they `ls` the directory and are frustrated by the delay. A `pvfs-ls` utility is provided with PVFS that avoids gathering this metadata, instead just printing directory contents. For users who simply want to see what resides in a directory, this is a much faster option.

PVFS does not cache data at the client side because it has no mechanism for ensuring that cached data is kept synchronized with data in other caches or on I/O servers. Hence, all data reads and writes must cross the network as well. Thus, the size of reads and writes to large files does have a significant impact on performance, especially through the VFS interface, which has particularly high overhead. This design decision makes PVFS perform poorly for benchmarks such as Bonnie [3]. Along these same lines, executing programs stored on a PVFS volume can be quite slow because pages are read one at a time on demand.

Missing Features

Users are occasionally surprised by the fact that some features are missing from PVFS. Here's a list as of version 1.5.8:

- links (both hard and symbolic)
- write-sharing through `mmap`
- `flock` and `fcntl` locks
- fault tolerance (other than using RAID, described later)

That's about it! If a user requires one of these features, perhaps one of the systems described earlier in the chapter will suffice instead.

20.4.2 Managing PVFS File Systems

PVFS allows for many different possible configurations. In this section we'll discuss some of these options.

While PVFS is relatively simple for a parallel file system, it can sometimes be difficult to discover the cause of problems when they occur simply because there are many components that might be the source of trouble. Here we discuss some tools and techniques for finding problem spots.

Monitoring File System Health

The `pvfs-ping` utility is the most useful tool for discovering the status of a PVFS file system and has turned into something of a "Swiss army knife" for PVFS debugging at this point.

A simple example of its use is as follows:

```
# pvfs-ping -h localhost -f /pvfs-meta -p 3000
mgr (localhost:3000) is responding.
iod 0 (127.0.0.1:7000) is responding.
pvfs file system /pvfs-meta is fully operational.
```

In this case the I/O server is dead and needs to be restarted:

```
# pvfs-ping -h localhost -f /pvfs-meta -p 3000
mgr (localhost:3000) is responding.
pvfs-ping: unable to connect to iod at 127.0.0.1:7000.
iod 0 (127.0.0.1:7000) is down.
pvfs file system /pvfs-meta has issues.
```

Using Multiple File Systems

Since PVFS includes no fault tolerance, for large systems it can make sense from a fault tolerance point of view to create multiple PVFS volumes. A single metadata server can serve multiple file systems if desired; however, if multiple file systems are chosen for fault tolerance reasons, it is definitely better to use multiple servers for I/O (one per file system). A single I/O server daemon (`iod`) cannot serve more than one file system. However, more than one daemon may be run on the same server if desired by specifying a different `port` value in the `iod.conf` file used to start the server.

Tolerating Disk Failures

Disk failures can be tolerated by using any of the many available RAID solutions under Linux, including both hardware devices and software RAID. There have been very few reported instances of data loss with PVFS because of software failures. Using RAID to tolerate disk failures is an effective mechanism for increasing the reliability of PVFS.

Increasing Usable File Descriptors

While some improvements have been made in PVFS with respect to file descriptor (FD) utilization, the servers in particular still can end up using all of their available FDs. The I/O servers will print a little message when this is about to happen:

```
NOTICE: exceeded 90 percent of available FDs (1024)!
```

Luckily this is easy to fix. The limits are set in `/etc/security/limits.conf`. Lines are of the following format:

```
<domain> <type> <item> <value>
```

The domain can be "*" for everyone, a userid, or a group using "@group". The type can be soft (setting the default) or hard (setting the maximum). The item parameter controls what limit this affects and can take many values, including nofile (open files). "Value" is the new value to set.

For example, the following lines would set the maximum number of FDs for root to 8192 and the default to 4096:

```
root hard nofile 8192
root default nofile 4096
```

Likewise one can set a new maximum and then use `limit` or `ulimit` as appropriate in the startup script for the servers.

Migrating Metadata

When upgrading to a newer PVFS version, occasionally the format of metadata on disk changes. This is due to oversights in the original design of the metadata format. Tools are now provided that can be used to convert metadata to the new format (assuming you haven't gotten too far behind on updates).

For example, if you are moving from version 1.5.6 to version 1.5.8, a utility called `migrate-1.5.6-to-1.5.8` is provided (there were no changes from 1.5.7 to 1.5.8 in the metadata format). This tool is used in conjunction with `find`:

```
# find /pvfs-meta -type f -not -name .pvfsdir -not \
  -name .iodtab -exec migrate-1.5.6-to-1.5.8 \{\} \;
```

Warning messages will be printed and the process aborted if the utility detects that the metadata is not the correct version. This process should be performed after stopping the mgr.

20.4.3 Tuning PVFS File Systems

We often get questions about how to tune PVFS file systems for the best performance. Truthfully, system hardware varies widely enough that it is difficult for us to supply any single set of parameters that will work best for everyone. Instead, in this section discuss some specific parameters common to all machines and some general techniques for improving overall PVFS file system performance. Chapters **??** and **??** include many tips for improving the overall performance of Linux nodes; all that information certainly applies to PVFS servers as well.

Of course, in addition to tuning the file system itself, many steps can be taken *above* the file system that can make a huge difference. Given the discussion of the PVFS design, many of these are obvious: using large requests rather than small ones, using MPI-IO so PVFS List I/O optimizations can be leveraged, and avoiding lots of metadata operations (opens, closes, and stats). Often such optimizations in application code can make more difference than any tuning within PVFS itself. An in-depth discussion of improving the performance of MPI-IO access can be found in [13].

Adjusting Socket Buffers

PVFS relies heavily on the `select` call and kernel handling of multiple TCP connections for parallelism. For this reason, it is often useful to tune the network-related parameters on the system. Chapter **??** covers this process in some detail; in particular increasing the `wmem_max` and `rmem_max` values is often very helpful.

Once these have been increased, the `socket_buf` option in the I/O server's configuration file (`iod.conf`) can be used to adjust the socket buffer size up to the new maximum.

Enabling DMA for Hard Drives

Chapter **??** describes the `hdparm` tool. It can be used to verify that DMA is turned on for the hard drives that are being used for PVFS storage and to turn this on if it is not enabled. Because PVFS pushes both the network and storage hardware,

alleviating any load on the CPU is helpful. Note that DMA isn't reliable on some hardware, so you should check the support of your hardware if this isn't turned on by default.

Improving Space Utilization

Originally we thought that users would want to know where their data was striped so that they could distribute processes to match data locations. Hence, we set up default striping so that data always started on the first I/O server. It turns out that for the most part people don't care about this and rarely use this information. Additionally, when users create lots of small files, this unbalances the distribution of data across the I/O servers.

We have subsequently added a "`-r`" flag that can be passed to the metadata server (`mgr`). This flag will cause the metadata server to choose a random starting I/O server when no server is specified (this can be done through the MPI-IO interface, for example). This will better distribute files and has a particularly large effect in the small files case.

Here we examine the free space on the I/O servers of a PVFS file system using the additional "`-s`" option to `pvfs-ping`:

```
# pvfs-ping -h localhost -f /pvfs-meta -s
mgr (localhost:3000) is responding.
iod 0 (192.168.67.51:7000) is responding.
iod 0 (192.168.67.51:7000): total space = 292825 Mbytes,
 free space = 92912 Mbytes
iod 1 (192.168.67.52:7000) is responding.
iod 1 (192.168.67.52:7000): total space = 307493 Mbytes,
 free space = 121154 Mbytes
iod 2 (192.168.67.53:7000) is responding.
iod 2 (192.168.67.53:7000): total space = 307485 Mbytes,
 free space = 121155 Mbytes
iod 3 (192.168.67.54:7000) is responding.
iod 3 (192.168.67.54:7000): total space = 307493 Mbytes,
 free space = 121199 Mbytes
```

We see that the first I/O server has significantly less free space than the others. This will show up in the df output:

```
Filesystem            Size  Used Avail Use% Mounted on
localhost:/pvfs-meta
```

```
1.2T  824G  363G  69% /pvfs
```

PVFS calculates the available space returned to the system by the minimum amount available on any single I/O server (in this case 92.9 Gbytes) times the number of I/O servers (in this case 4). Because so much less space is available on the first server, we get a very low reported available space. Using the "`-r`" manager flag described above will help alleviate this problem.

Testing Aggregate Bandwidth

Since users are mostly interested in PVFS for high performance, obtaining a baseline performance number for a particular configuration is fairly important. The `pvfs-test` tool supplied with PVFS can be used for this purpose. This is an MPI program that opens a file from a large number of processes and writes or reads that file in parallel with each process accessing a different large block of the file. A "`-h`" option will cause it to list its options. This program can be used as a simple benchmark for testing the effects of configuration changes.

Here's the output of one of our favorite runs, using 80 nodes of Chiba City (see Chapter **??**) as clients for PVFS and 128 separate nodes for I/O servers back in April of 2001:

```
mpirun -nolocal -np 80 -machinefile mach.all pvfs-test -s 262144 -f
 /sandbox/pvfs/testfile -b 268435456 -i 1 -u
# Using native pvfs calls.
nr_procs = 80, nr_iter = 1, blk_sz = 268435456, nr_files = 1
# total_size = 21474836480
# Write:  min_t = 3.639028, max_t = 6.166665, mean_t = 4.755538,
 var_t = 0.334923
# Read:  min_t = 6.490499, max_t = 7.171075, mean_t = 6.977580,
 var_t = 0.023353
Write bandwidth = 3482.406857 Mbytes/sec
Read bandwidth = 2994.646755 Mbytes/sec
```

We did not sync after the writes ("`-y`" option), so the data was at the servers but not necessarily on disk. Nevertheless we were able to create a 20 Gbyte file in just over 6 seconds and read it back in just over 7 seconds. Not too shabby at the time. Note that we found a strip size of 256 Kbytes to be the best for that particular configuration, where a *strip* is the amount of data written to a single server (and a stripe is the amount written across all servers in the round-robin fashion).

Adjusting the Default Strip Size

By default the strip size (the size of the regions distributed in round-robin fashion to I/O servers) is set to 64 Kbytes (as of version 1.5.8). For some systems, particularly ones using large RAID volumes at each I/O server, this is simply too small.

The `pvfs-test` tool can be used to experiment with various strip sizes in order to find a good one for a particular configuration. Using the "`-y`" option will help ensure more accurate results by forcing data to the disk. Once a good value has been found, an additional "`-s ssize`" option can be used with the metadata server in order to provide the new default value (ssize is in bytes).

It is also useful to adjust the I/O server write buffer size to be larger than this size. That value is set in the I/O server configuration file with the `write_buf` option (value is in Kbytes, and the default is 512 Kbytes).

20.4.4 ROMIO and PVFS

MPI-IO implementations provide a number of services over using a local file interface. First and foremost these implementations provide a portable interface to which application programmers can code. The MPI-IO implementation takes MPI-IO operations and translates these into operations that can be performed by the underlying file system. Depending on the underlying file system, the MPI-IO implementation has a number of options with respect to how it translates an MPI-IO read or write operation into file system operations. If the underlying file system supports only POSIX operations, the MPI-IO layer might convert the MPI-IO request into a collection of contiguous operations. For a file system such as PVFS, MPI-IO requests might instead be converted into List I/O operations.

The second service that MPI-IO implementations provide is I/O optimizations. As we have discussed before, the MPI-IO semantics leave some opportunities for performance optimizations that are not available under the POSIX semantics. Further, the information provided by the use of collective I/O calls provides additional opportunities for optimizations. For more information on MPI-IO in general, including examples, see Chapter **??** of this book or [13]. In this section we will touch upon building ROMIO with PVFS support and then discuss in detail the optimizations available within ROMIO that are usable with PVFS.

Building MPICH and ROMIO with PVFS Support

Chapter **??** introduced the MPICH implementation of the MPI standard. ROMIO is included as part of the MPICH package. When configuring MPICH with ROMIO

and PVFS support, a few additional parameters are necessary. Particularly we want to tell ROMIO what kinds of file systems to support, link to the PVFS library, and provide the path to PVFS include files.

For example, let us assume that PVFS was previously installed into `/soft/pub/packages/pvfs-1.5.8`, and we want both PVFS and "regular" (UFS) file system support:

```
# ./configure --with-romio="-file_system=pvfs+ufs"
 -lib="-L/soft/pub/packages/pvfs-1.5.8/lib/ -lpvfs"
 -cflags="-I/soft/pub/packages/pvfs-1.5.8/include"
```

The standard MPICH build and installation procedure can be followed from here. Building with LAM is very similar.

If ROMIO is not compiled with PVFS support, it will access files only through the kernel-supported interface (i.e., a mounted PVFS file system). If PVFS support is compiled into ROMIO and you attempt to access a PVFS-mounted volume, the PVFS library will detect that these are PVFS files (if the `pvfstab` file is correct) and use the library calls to avoid the kernel overhead. If PVFS support is compiled into ROMIO and you attempt to access a PVFS file for which there is *no* mounted volume, the file name passed to the MPI-IO call *must* be prefixed with `pvfs:` to indicate that the file is a PVFS file; otherwise ROMIO will not be able to find the file.

ROMIO Optimizations

ROMIO implements a pair of optimizations to address inefficiencies in existing file system interfaces and to leverage additional information provided through the use of collective operations. These optimizations, as well as PVFS options such as striping parameters, are controlled through the use of the MPI_Info system, commonly known as "hints." Much of the information in this section comes from the ROMIO users guide [36]; this guide provides additional information on these topics as well as covering the use of ROMIO on file systems other than PVFS.

ROMIO implements two I/O optimization techniques that in general result in improved performance for applications. The first of these is *data sieving* [34]. Data sieving is a technique for efficiently accessing noncontiguous regions of data in files when noncontiguous accesses are not provided as a file system primitive or where the noncontiguous access primitives are inefficient for a certain datatype. In the data sieving technique, a number of noncontiguous regions are accessed by reading a block of data containing all of the regions, including the unwanted data

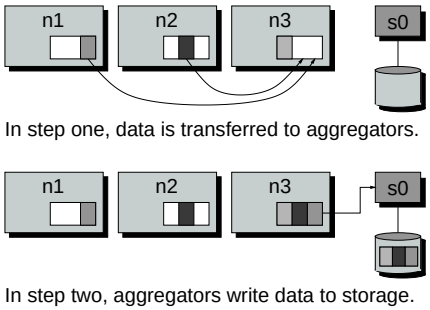

Figure 20.8: Two-Phase Write Steps

between them (called "holes"). The regions of interest are then extracted from this large block by the client. This technique has the advantage of a single I/O call, but additional data is read from the disk and passed across the network. For file systems with locking the data sieving technique can also be used for writes through the use of a read-modify-write process. Unfortunately, since PVFS does not have file locking of any kind currently, this is not available for PVFS.

Two hints can be used to control the application of data sieving in ROMIO for PVFS:

- `ind_rd_buffer_size` controls the size (in bytes) of the intermediate buffer used by ROMIO when performing data sieving during read operations. Default is `4194304` (4 Mbytes). If data will not all fit into this buffer, multiple reads will be performed.

- `romio_ds_read` determines when ROMIO will choose to perform data sieving. Valid values are `enable`, `disable`, or `automatic`. Default value is `automatic`. In `automatic` mode ROMIO may choose to enable or disable data sieving based on heuristics.

The second optimization is *two-phase I/O* [33]. Two-phase I/O, also called collective buffering, is an optimization that applies only to collective I/O operations. In two-phase I/O, the collection of independent I/O operations that make up the collective operation are analyzed to determine what data regions must be transferred (read or written). These regions are then split up among a set of aggregator processes that will actually interact with the file system. In the case of a read, these aggregators first read their regions from disk and redistribute the data to the final

locations; in the case of a write, data is first collected from the processes before being written to disk by the aggregators. Figure 20.8 shows a simple example of the two-phase write using a single aggregator process. In the first phase (step), the two nonaggregator processes pass their data to the aggregator. In the second step the aggregator writes all the data to the storage system. In practice many aggregators are used to help balance the I/O rate of the aggregators to that of the I/O system. Because the MPI semantics specify results of I/O operations only in the context of the processes in the communicator that opened the file, and all these processes are involved in collective operations, two-phase I/O *can* be applied on PVFS files.

Six hints can be used to control the application of two-phase I/O:

- `cb_buffer_size` controls the size (in bytes) of the intermediate buffer used in two-phase collective I/O (both reads and writes). If the amount of data that an aggregator will transfer is larger than this value, then multiple operations are used. The default is `4194304` (4 Mbytes). If the data size exceeds this buffer size, multiple iterations of the two-phase algorithm will be used to accomplish data movement.

- `cb_nodes` controls the maximum number of aggregators to be used. By default this is set to the number of unique hosts in the communicator used when opening the file.

- `romio_cb_read` controls when collective buffering is applied to collective read operations. Valid values are `enable`, `disable`, and `automatic`. Default is `automatic`. When enabled, all collective reads will use collective buffering. When disabled, all collective reads will be serviced with individual operations by each process. When set to `automatic`, ROMIO will use heuristics to determine when to enable the optimization.

- `romio_cb_write` controls when collective buffering is applied to collective write operations. Valid values are `enable`, `disable`, and `automatic`. Default is `automatic`. See the description of `romio_cb_read` for an explanation of the values.

- `romio_no_indep_rw` indicates that no independent read or write operations will be performed. This can be used to limit the number of processes that open the file.

- `cb_config_list` provides explicit control over aggregators, allowing for particular hosts to be used for I/O. See the ROMIO users guide for more information on the use of this hint.

ROMIO Data Placement Hints

Three hints may also be used to control file data placement. These are valid only at open time:

- `striping_factor` controls the number of I/O servers to stripe across. The default is file system dependent, but for PVFS it is `-1`, indicating that the file should be striped across all I/O devices.
- `striping_unit` controls the striping unit (in bytes). For PVFS the default will be the PVFS file system default strip size.
- `start_iodevice` determines what I/O device data will first be written to. This is a number in the range of 0 ... striping_factor - 1.

ROMIO and PVFS List I/O

Two hints are available for controlling the use of list I/O in PVFS:

- `romio_pvfs_listio_read` has valid values `enable`, `disable`, and `automatic`. The default is `disable`. This hint takes precedence over the `romio_ds_read` hint.
- `romio_pvfs_listio_write` has valid values `enable`, `disable`, and `automatic`. The default is `disable`.

Clearly, a wide variety of parameters can be used to control the behavior of ROMIO and PVFS when used together. Because no single set of parameters works best for all applications, experimentation is often necessary to attain the best set of parameters. A study examining some of these parameters has been published [8]; this can serve as a starting point for your own tuning.

20.4.5 Bugs

Users sometimes encounter bugs in PVFS. When they do, we generally guide them through a predictable set of steps to help us discover where the problem lies. This section outlines this process. The purpose is not to discourage users from reporting bugs or asking for help, but to streamline the process. If you have already tried these steps, we can skip a number of email exchanges and get right to the root of the problem!

Checking the List Archives

The very first thing to do is to check the PVFS mailing list archives. These are searchable online and available from the PVFS Web site [27]. Many problems have already been reported, so checking here might provide you with an immediate solution.

Reporting Versions and Logged Output

Bugs should always be reported to the PVFS users mailing list. This is an open list for discussion of many PVFS issues, one of them being bugs. By reporting to the mailing list you reach the maximum number of people that might be able to solve your problem, and you guarantee that an archive of the discussion will be saved.

We will always ask what version of the code you are running, especially if the problem that you report looks like something that has already been fixed. The distribution and kernel version you are using are helpful as well. If the problem is related to compiling, we'll ask for configure output and a log of the make process. If the problem is a runtime one, we'll ask for any information in the logs that might help. This includes `dmesg` output, the `pvfsd` log, the `iod` logs, and the `mgr` log. By default the three types of log files are all placed in `/tmp`, although this can be changed with configure-time options.

Providing this information in your first message is the easiest way to get the bug reporting and fixing process started.

Client Side or Server Side

The most common runtime bugs seen in PVFS at this time concern the Linux kernel module. One of the first things that we do in the case of a runtime problem is try to determine whether the problem is related to the servers themselves or to a particular client. We usually ask the user to look at the state of other clients in order to determine this. For example, one bug that we have seen prevented new files from showing up on certain clients. One client would see the new file while others did not. By looking at the state of multiple clients, the user was able to report this back and help us narrow down the problem.

Simplifying the Scenario

The simpler the set of conditions necessary to cause the problem on your system, the more likely we are to be able to replicate it on some system we have access to. Hacking out portions of a scientific code so that it performs only I/O or writing

a script that uncovers a metadata incoherence problem really helps us see what is going on and replicate the problem on our end.

20.5 Parallel I/O in the Future

Machines with tens of thousands of nodes are on the horizon. For a parallel I/O system to efficiently operate at this scale, a number of issues must be addressed.

Adapting to new technologies is critical in this environment. It is not clear what processor, storage, or network technologies will be present in future machines, or even what operating system will run on nodes. Any new parallel file system design should be built with abstract interfaces to allow adoption of new technologies and porting to new operating systems.

Leveraging collective operations, rich I/O request languages, and relaxed consistency semantics will be key to operating efficiently on these machines and exploiting the inherent hierarchy in these systems. Opportunities exist at many levels in the I/O component stack to boost performance.

Management of I/O systems is a growing concern because the systems continue to become more complex. Tools to aid the administrator are key, and self-maintaining solutions would be ideal.

Our next-generation parallel file system, PVFS2, is being designed to tackle just these problems. By the time this book is published, early versions of the next-generation Parallel Virtual File System, PVFS2, should be available online. The core of PVFS2 has been designed to provide

- modular networking and storage subsystems,
- a structured data request format modeled after MPI datatypes,
- flexible and extensible data distribution modules,
- distributed metadata,
- tunable consistency semantics, and
- and support for data redundancy.

PVFS2 is the culmination of a 3-year effort to redesign PVFS as a production-capable parallel file system based on experience gained in the design and operation of the original PVFS, observations of other parallel file systems, and interactions with the scientific data management community.

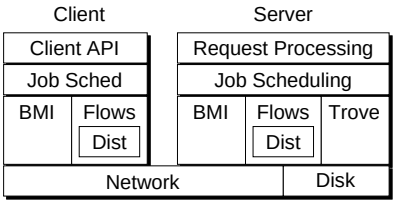

Figure 20.9: PVFS2 Software Architecture

In this section we will examine some of the challenges facing parallel I/O systems both today and in the near future. We will use PVFS2 as one example of how these problems might be addressed.

20.5.1 Supporting New Hardware Technologies

While in some sense cluster computing is about using commodity parts, we often see new technologies in use in larger clusters before they hit the commodity market. Networks are a great example of this; we see many interesting network technologies, including Myrinet, Quadrics, and InfiniBand, in use in clusters today. Likewise on the storage side we see locally attached hardware, SANs, and iSCSI as some of the potential mechanisms for storage access. Leveraging these technologies requires appropriate abstractions in the I/O system. In the Lustre design we see a very modular system used to attack just this problem [2].

20.5.2 PVFS2 Abstract Interfaces

PVFS2 also addresses this problem with abstraction layers. The first two of these are BMI, through which client and server messages flow, and Trove, through which storage is accessed. Figure 20.9 shows the overall software architecture of PVFS2; we will discuss the major components here.

The Buffered Messaging Interface (BMI) provides a nonblocking network interface that can be used with a variety of high-performance network fabrics and is tailored for use in file system servers and clients. Currently BMI modules exist for both TCP/IP and GM networks.

The Trove storage interface provides a nonblocking interface that can be used with a number of underlying storage mechanisms. Trove storage objects, called data spaces, consist of both a stream of bytes and a keyword/value pair space, similar in some ways to the data and resource forks available in other local file

systems. Keyword/value pairs are convenient for arbitrary metadata storage and directory entries, while the stream of bytes is a natural place to store file data. The current implementation uses Unix files and Berkeley db4, but many other implementations are possible.

The third major abstraction in PVFS2 is Flows. Flows combine the functionality of the network and storage subsystems by providing a mechanism to specify a flow of data between network and storage. Flows also incorporate the request and distribution processing system that allows PVFS2 to handle highly complex access patterns. Finally, Flows provide a point for optimization: specific flow implementations to optimize data movement between a particular network and storage pair can be implemented to exploit fast paths.

Above all these the job scheduling layer provides a common interface for posting BMI, Trove, and Flows and checking on their completion. Within this layer, scheduling decisions are made, and multiple threads are used to manage asynchrony and leverage multiple CPUs. This is tightly integrated with a state machine processing system that is used to track operations in progress. With this layer in place, new underlying components may also be added and integrated with minimal effort. At the highest level within the server, the request processing component handles incoming requests and initializes new state machines to process these requests.

20.5.3 Tolerating Faults

Parallel computing systems continue to grow in numbers of components (nodes, disks, etc.), and because components are becoming no more reliable, the likelihood of component failure is increasing. While application and middleware are beginning to be adapted to handle faults, most users depend on the I/O system to be a reliable and available location for data storage. On the other hand, because providing fault tolerance usually lowers performance, some users will desire to forego fault tolerance at the I/O system level and instead implement it in a more efficient manner at the application level. Doing so allows them to get the highest performance from the I/O system. For a parallel file system to be usable in many domains, the level of redundancy should be configurable. The approach PVFS2 takes to redundancy is much the same as it takes to semantics and other issues involving a trade-off between performance and protection: that is, it provides a choice of various levels of protection, with the requisite loss or gain in performance. Thus PVFS2 aims to allow files to be stored with no redundancy or with varying degrees of redundancy, as needed. Multiple technologies may be leveraged to accomplish this, some built into the file system and others external components.

Redundant Storage

Many tools are available for providing fault tolerance in storage systems. One is the use of local RAID arrays. This is a time-proven approach to handling disk failures, and a RAID provides high-performance I/O with minimal performance degradation when directly accessed by a single I/O server. We encourage this application of RAID with both PVFS1 and PVFS2. RAID like-techniques can also be applied across the devices on a SAN; the file system examples implementing a VSD use this type of approach. Using RAID in this way can incur performance penalties because of the fine-grained locking often used to control concurrent access when multiple nodes have access to the resources.

In PVFS2 we will provide what we term *lazy redundancy* as an option. In this approach writes to files do not update redundant information automatically as they would in a RAID-like approach. Instead redundant information is updated only when clients make explicit calls. These calls can be automatically made within I/O middleware libraries at logical points, such as MPI sync or close operations. By delaying the update to these explicit points we allow the I/O layers the option of aggregating updates to redundant data. Further, in the context of MPI-IO we have control of all the processes accessing the file; we can use these processes to update redundant data in parallel for higher performance. The data distribution component of PVFS allows us to describe where this redundant data is located in a convenient manner, and the approach can be applied on a per file basis. Lazy redundancy can be coupled with server failover to provide an even greater degree of protection.

Failover

High availability (HA) software provides a mechanism for server failover in the case of node failure. Dual-attach storage hardware can be used with this software to tolerate single-node failures by creating pairs of nodes that provide "backup" for each other. This allows systems to run in what is termed *active-active* mode (meaning that neither node sits idle in absence of failure), with somewhat degraded performance in the event that one node fails. Of course, if you don't mind having half your system sitting idle, *active-passive* mode can be used, leaving an extra server for each of the ones in service. More complicated HA solutions are becoming available that allow for a pool of backup servers that can be brought online as needed. In contrast to active-active pairs, this architecture would allow for a small number of extra servers that could fill in without degrading performance. However,

these extra servers would need access to many different storage resources; providing this capability could be prohibitively expensive.

Having the hardware and software infrastructure necessary to restart a server on backup hardware is just the first step. A second issue to be considered with respect to failover is shared state. Clients and servers in a *stateless* system do not maintain information about other entities in the system that is necessary for correct operation (i.e., they can cache information for performance reasons, but the system must be able to function without this information). Assuming that a system is stateless and that no file system data is cached in volatile storage, a server restart need not cause the loss of any data. Unfortunately, shared state is used in many parallel file systems; write-back caches are an example, where a client is holding onto the state of blocks (for performance reasons) that a server is in fact responsible for. Servers and clients can checkpoint their state on shared storage if it is available. This is a viable option for systems where clients and servers have access to shared storage, but this connectivity may not be available. Another option is to implement an arbitration process that allows the system as a whole to reclaim resources and synchronize state in the event of a node loss. Handling all the failure cases can be very difficult. PVFS2 servers and clients are stateless in order to simplify the use of failover solutions and minimize complication in failure scenarios.

20.5.4 Aiding Management

Most parallel file systems today (excluding PVFS1) have mechanisms for checking the status of devices involved in the system, migrating data on and off particular resources, checking the consistency of the file system, and adding or removing devices from the file system. Looking beyond this functionality, we can imagine I/O systems that can suggest optimizations based on observed access patterns or, even better, manage themselves. The areas of "autonomic computing" [18] and "autonomous storage" in particular cover just this type of operation. PVFS1 lacked most of the management tools that administrators expect from such a system, and we believe that this discouraged its use in a number of cases. We intend to take management very seriously in PVFS2, and we will discuss some of the basics here.

System Monitoring

The first step in easing the management task is providing tools for system monitoring. These tools should allow for both examining the real-time state of the system and looking at trends over time for optimization purposes. A complicating factor

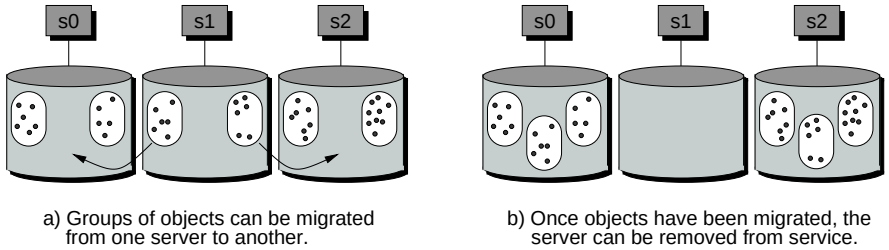

Figure 20.10: Migrating Storage Objects

in parallel I/O monitoring is the sheer amount of data available, particularly information on access patterns. This process is similar to gathering logs from parallel programs in that data from many cooperating components must be collected and presented coherently.

The PVFS development team has experimented with instrumentation of the PVFS1 servers and has developed tools to aggregate performance metrics, collect access pattern statistics, and visualize the results. Using the tools and techniques developed for PVFS and looking at other work in the area, we have slated monitoring operations to be an integral part of the suite of operations that PVFS2 supports. These monitoring functions can be used as a starting point for visualization, analysis, and self-management tools.

Data Migration

In PVFS2, file data is distributed to Trove data objects for storage. Trove data objects are referenced by a handle. These data object handles are clustered into logical groups such that all handles within a logical group are managed by a single server. In the simplest case, each server manages a single logical group of Trove handles, and therefore the objects referenced by those handles. These groups can be split and merged if necessary for repartitioning purposes, and servers may be responsible for many of these logical groups. The mapping of handles to the servers where they are stored is a part of system configuration and is easily changed. This not only provides a decoupling from handles to servers but potentially allows storage objects to be moved from server to server by transferring the control of an entire logical group and updating the handle mapping appropriately.

Figure 20.10 shows three servers, each with two logical groups of storage objects referenced by different handle ranges. If resources on the middle server need to

be freed (e.g., to replace faulty hardware), the groups of objects stored on the server can be relocated to one or more servers, and requests from the clients can be redirected by updating the mapping of these groups. Because these Trove objects are used to store both metadata and data, metadata can be migrated in the same manner.

Automated Management

Ideally, these systems would simply manage themselves! If components fail, data could be migrated appropriately to allow for continued fault tolerance and minimum degradation of performance. Data that is used frequently as input could be replicated so that multiple copies were available for reading or redistributed to match observed access patterns. Infrequently used data could be kept on slower disks or moved to tertiary storage. Caching and scheduling policies could be tuned to match access patterns as well.

We are very interested in this type of system, and we plan to start working in this area once the PVFS2 system matures. The first step is to provide a suite of management operations as part of the server API. With this API in place, an additional set of monitoring processes can interpret the performance monitoring information over time and direct changes to the file system accordingly. Separating these management processes into their own components will maintain the simplicity of the underlying PVFS2 core.

20.5.5 Leveraging I/O Languages and Semantics

Earlier in the chapter we discussed consistency semantics; obviously we feel that experimentation in this area could lead to useful alternative semantics. Equally important is the use of structured I/O descriptions from the highest-level interfaces down as low as possible in the I/O stack. Certainly, parallel file systems should be supporting these operations.

MPI-IO

So far, MPI-IO implementations have been very conservative in their exploitation of the more relaxed MPI-IO consistency semantics. As systems scale, taking further advantage of these semantics allows us to potentially improve I/O system performance using the same hardware as before. During this process, however, some users are likely to experience surprising behavior from the I/O system because of

assumptions about what level of consistency MPI-IO will provide. We will do everything possible to minimize the pain experienced by users in order to keep them from abandoning this powerful API.

Caching at the MPI layer is one of the biggest opportunities that has so far been unexploited in production systems. The constrained scope of the MPI-IO semantics, coupled with the explicit synchronization points, makes caching in MPI-IO a straightforward process. This is in stark contrast to the infrastructure necessary to cache under the POSIX interface. File systems such as PVFS2 can benefit greatly from caching at this layer.

Operations such as `MPI_File_open` can be further optimized with appropriate support from the parallel file system. PVFS2 does not keep state regarding open files. Instead, clients essentially find only a file handle during an open call. A scalable implementation of `MPI_File_open` for PVFS2 can have a single process perform the mapping from file name to handle, then broadcast the file handle to the rest of the processes. This type of optimization can be applied in a number of cases where MPI collectives are used.

Configurable Semantics

Earlier in the chapter we discussed the file consistency semantics of a number of interfaces. We noted that for some types of workloads the NFS semantics were acceptable, while for others they were not. It is not difficult for a parallel file system to relax its semantics; usually this is a matter of simply neglecting to perform consistency checks it might have otherwise. This approach should be considered seriously. As a real example, many large physics datasets are being put online today. Files in these datasets are never modified once written. Aggressive caching of these files can be performed because the semantics applied to the dataset by the scientists permit it. We should allow for these optimizations.

Likewise, relaxing the consistency of directory contents provides another potential point of optimization, as could metadata of files and directories (in particular file size). PVFS2 will provide a configurable window of time for which previous metadata values and directory contents are treated as up to date. This allows for caching without locks, at the cost of short periods of time where views of the file system on different clients are slightly different. Such an approach might be useful as we attempt to share parallel file system access across wide-area networks.

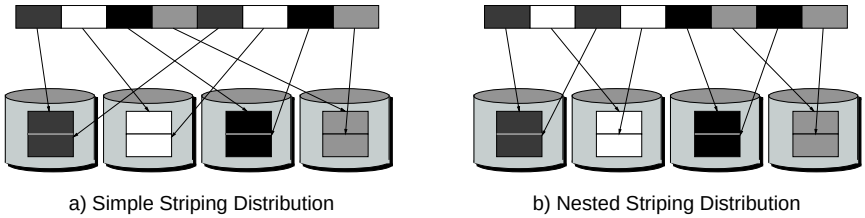

Figure 20.11: Examples of Data Distributions

Describing I/O Operations and Data Distributions

PVFS2 allows for structured I/O requests via a format based directly on MPI datatypes. Currently a set of datatype constructor functions identical in function to the equivalent MPI calls is provided, and the format can readily be translated from existing MPI datatype formats, making it trivial to leverage this functionality within an MPI-IO implementation such as ROMIO. PVFS2 servers directly process this format (in the flow component) to service I/O requests; the type is not converted into a vector before processing.

With structured data sets comes the potential for leveraging more sophisticated data distributions. Most parallel file systems use striping. In PVFS2, however, the distribution mechanism has been abstracted so that different files can be stored with different distributions. PVFS2 relies on an algorithmic mechanism for distribution of data to servers. The functions that define the distribution can now be selected at file creation time, permitting a number of potential data distributions.

One such alternative distribution pattern is nested striping. As shown in Figure 20.11, simple striping distributes data round-robin to all IO nodes used to store the file. Nested striping distributes data round-robin first to one subset of nodes and then to another subset in a round-robin pattern among subsets. This pattern better matches block distributions of multidimensional datasets. Any distribution that can be represented algorithmically can potentially become a PVFS2 distribution scheme.

20.6 Conclusions

The many software and hardware layers of parallel I/O systems and the terminology used to describe them can be very confusing. Underlying this complexity, however, are simple concepts: methods of describing accesses, consistency seman-

tics, distributing data across many resources, and surviving component failures. Armed with knowledge of these concepts, one can both qualitatively assess the appropriateness of a particular system to a given problem and devise tests to measure quantitatively the effectiveness of the system. Many of these I/O systems share common traits, so the example systems presented here can be used as a frame of reference when examining new systems as well.

Parallel I/O continues to grow in importance as a component of clusters. While existing parallel file systems such as PVFS1, GPFS, and GFS are filling existing needs, new systems such as Lustre and PVFS2 are already being built to meet the needs of upcoming systems. These systems build on the successes of the past but also address issues germane to upcoming systems, in particular parallel I/O system portability and increased scale. Even so, additional effort will be necessary to see exciting concepts such as autonomous storage become reality.

As parallel I/O researchers and developers, we definitely have our work cut out for us!

The submitted manuscript has been created by the University of Chicago as Operator of Argonne National Laboratory ("Argonne") under Contract No. W-31-109-ENG-38 with the U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.

This chapter was processed by LaTeX on May 18, 2004.

References

[1] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[2] Peter J. Braam. The Lustre storage architecture. Technical report, Cluster File Systems, Inc., 2003.

[3] Tim Bray. Bonnie file system benchmark. `http://www.textuality.com/bonnie/`.

[4] Ron Brightwell, Tramm Hudson, Arthur B. Maccabe, and Rolf Riesen. The Portals 3.0 message passing interface. Technical Report SAND99-2959, Sandia Technical Report, November 1999.

[5] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. Technical Report RFC 1813, Sun Microsystems, Inc., June 1995.

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

[7] Albert Cheng and Michael Folk. HDF5: High performance science data solution for the new millennium. In ACM, editor, *SC2000: High Performance Networking and Computing. Dallas Convention Center, Dallas, TX, USA, November 4–10, 2000*, pages 149–149, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2000. ACM Press and IEEE Computer Society Press.

[8] Averg Ching, Alok Choudhary, Kenin Coloma, Wei keng Liao, Robert Ross, and William Gropp. Noncontiguous I/O accesses through MPI-IO. In *Proceedings of the Third IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2003)*, May 2003.

[9] Avery Ching, Alok Choudhary, Wei keng Liao, Robert Ross, and William Gropp. Noncontiguous I/O through PVFS. In *Proceedings of the 2002 IEEE International Conference on Cluster Computing*, September 2002.

[10] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 20, pages 285–308. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[11] The galley parallel file system. `http://www.cs.dartmouth.edu/~dfk/nils/ /galley.html`.

[12] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI—The Complete Reference: Volume 2, The MPI-2 Extensions*. MIT Press, Cambridge, MA, 1998.

[13] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[14] Michael Hasenstein. The logical volume manager (LVM). Technical Report Whitepaper, SuSE Inc., 2001.

[15] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, December 1998.

[16] James V. Huber, Jr., Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 22, pages 330–343. IEEE Computer Society Press and Wiley, New York, NY, 2001.

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

[18] Jeffrey Kephart and David Chess. The vision of autonomic computing. *IEEE Computer*, pages 41–50, January 2003.

[19] David Kotz. Disk-directed I/O for MIMD multiprocessors. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 35, pages 513–535. IEEE Computer Society Press and John Wiley & Sons, 2001.

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

[21] J. Li, W.-K. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, and R. Latham. Parallel netCDF: A scientific high-performance I/O interface. Technical Report ANL/MCS-P1048-0503, Mathematics and Computer Science Division, Argonne National Laboratory, May 2003.

[22] Nils Nieuwejaar and David Kotz. The Galley parallel file system. *Parallel Computing*, 23(4):447–476, June 1997.

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

[24] Bill Nowicki. NFS: Network file system protocol specification. Technical Report RFC 1094, Sun Microsystems, Inc., March 1989.

[25] Kenneth W. Preslan, Andrew Barry, Jonathan E. Brassow, Russell Cattlelan, Adam Manthei, Erling Nygaard, Seth Van Oort, David C. Teigland, Mike Tilstra, Matthew O'Keefe, Grant Erickson, and Manish Agarwal. A 64-bit, shared disk file system for Linux. In *Proceedings of the Eighth NASA Goddard Conference on Mass Storage Systems and Technologies*, March 2000.

[26] Kenneth W. Preslan, Andrew P. Barry, Jonathan E. Brassow, Grant M. Erickson, Erling Nygaard, Christopher J. Sabol, Steven R. Soltis, David C. Teigland, and Matthew T. O'Keefe. A 64-bit, shared disk file system for Linux. In *Proceedings of the Seventh NASA Goddard Conference on Mass Storage Systems*, pages 22–41, San Diego, CA, March 1999. IEEE Computer Society Press.

[27] The parallel virtual file system. `http://www.pvfs.org`.

[28] Using the parallel virtual file system. `http://www.parl.clemson.edu/pvfs/user-guide.html`.

[29] R. Ross, D. Nurmi, A. Cheng, and M. Zingale. A case study in application I/O on linux clusters. In *Proceedings of SC2001*, November 2001.

[30] Robert B. Ross. *Reactive Scheduling for Parallel I/O Systems*. PhD thesis, Dept. of Electrical and Computer Engineering, Clemson University, Clemson, SC, December 2000.

[31] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *First USENIX Conference on File and Storage Technologies (FAST'02)*, Monterey, CA, January 28-30 2002.

[32] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS version 4 protocol. Technical Report RFC 3010, Sun Microsystems, Inc., Hummingbird Ltd., Zambeel, Inc., and Network Appliance, Inc., December 2000.

[33] Rajeev Thakur and Alok Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Scientific Programming*, 5(4):301–317, Winter 1996.

[34] Rajeev Thakur, Alok Choudhary, Rajesh Bordawekar, Sachin More, and Sivaramakrishna Kuditipudi. Passion: Optimized I/O for parallel applications. *IEEE Computer*, 29(6):70–78, June 1996.

[35] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, May 1999.

[36] Rajeev Thakur, Robert Ross, Ewing Lusk, and William Gropp. Users guide for ROMIO: A high-performance, portable MPI-IO implementation. Technical Report ANL/MCS Technical Memorandum No. 234, Mathematics and Computer Science Division, Argonne National Laboratory, May 2002.

[37] C. Thekkath, T. Mann, and E. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles (SOSP)*, October 1997.

[38] The treadmarks distributed shared memory (DSM) system. `www.cs.rice.edu/~willy/TreadMarks/overview.html`.

[39] M. Vilayannur, A. Sivasubramaniam, M. Kandemir, R. Thakur, and R. Ross. Discretionary caching for I/O on clusters. In *Proceedings of the Third IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2003)*, May 2003.

Index

B

block devices, 498

C

CHARISMA project, 496

D

data migration
  in PVFS2, 530
data shipping, 502
data sieving, 520
disk-directed I/O, 504

F

file
  locks, 498
file system
  CXFS, 499
  Galley, 503
  GPFS, 501
  Lustre, 506
  PVFS, 504
  PVFS2, 525
Frangipani, 499

G

Galley, 503
GFS, 500
GPFS, 501

H

hdparm, 516
high availability, 528

I

I/O access patterns, 496
intelligent I/O servers, 503

L

locking subsystem, 498
locks, 509
Lustre, 506
LVM, 500

M

MPI-IO, 495
  semantics, 511
MPICH, 519

N

nested-strided access, 497
NFS
  semantics, 510

P

parallel file system, 494, 495
parallel I/O, 493
Petal, 499
POSIX I/O
  implementing, 509
  semantics, 508
PVFS, 504
  aggregate bandwidth, 518
  file descriptors, 515
  handling disk failures, 515
  MPI-IO hints, 521
  multiple file systems, 514
  semantics, 512
  strip size, 519
  tuning, 516
  upgrading, 515
  using with ROMIO, 519
`pvfs-ping`, 514
`pvfs-test`, 518
PVFS2, 525
  data migration, 530

R

ROMIO, 519
  data sieving, 520
  hints, 521
  optimizations, 520
  two-phase I/O, 521

S

sequential access, 496
shared storage architecture, 498
stream-based I/O, 505
strided access, 497
strip, 518

T

two-phase I/O, 502, 521

V

virtual block device, 498