17 PVFS: Parallel Virtual File System

Walt Ligon and Rob Ross

An increasing number of cluster-based systems are being used for applications that involve not only a significant computational component but also a large amount of I/O. These applications consume or produce very large data sets, generate large checkpoint dumps, or use very large databases. In these situations, a Beowulf computer may be especially attractive because each node of the cluster includes a fully functional I/O subsystem. By using all of the available I/O hardware as a parallel I/O system, you can realize serious performance gains at low cost. As is typically the case in Beowulf systems, the key ingredient is software that allows these hardware resources to be orchestrated for use in high-performance applications. In this chapter we discuss parallel file systems—software that allows all of the disks in a cluster to be used as a single high-performance storage resource. In particular, we present details of the Parallel Virtual File Systems (PVFS), an open source implementation of a parallel file system designed for use on Beowulf computers.

17.1 Introduction

In this section we discuss in general terms what a parallel file system is, what one can do, and when it might be appropriate to use one. We also cover ways to use a parallel file system in parallel applications and issues that might affect performance. Subsequent sections present the details of installing and using PVFS.

17.1.1 Parallel File Systems

A parallel file system (PFS) is system software for a parallel computer that provides data distribution and parallel access. Data distribution allows file data to be distributed among disks attached to different nodes of the computer. Parallel access facilitates coordinated access to that file data by the multiple tasks of a parallel application. The primary goal of a parallel file system is to provide very high performance I/O access to large data sets for parallel applications. This point is as important in what it does *not* say as in what it does say. In particular, parallel file systems may *not* provide especially good performance for single-task applications. Parallel file systems may *not* provide especially low latency for small random accesses. Parallel file systems may *not* provide redundancy or other means of security or reliability. That said, the designer of any file system strives to incorporate these features to the extent possible, but in the case of a parallel file system these considerations are typically secondary to the goal of high-performance access to large data sets.

For purposes of this discussion, a parallel computer is a collection of processor nodes connected with an interconnection network. Some of these nodes have I/O devices attached. A parallel application consists of a number of *tasks* that are distributed among some of the nodes for processing. We call nodes that have attached I/O devices *I/O nodes* and nodes that run an application task *compute nodes*. These sets of nodes may be distinct or may overlap, even to the extent that all nodes are both I/O nodes and compute nodes.

A parallel file system generally consists of two software components: *client code*, which runs on the compute nodes; and *server code*, which runs on the I/O nodes. Application tasks, or clients, present I/O requests to the client code on the compute node where the task is running. The client code decomposes the request and sends it to each of the I/O nodes where the affected data resides. The server code on the I/O nodes transfers data between the network and the storage devices. In a typical I/O request, communication between the compute nodes and I/O nodes is an "all-to-all" pattern. For a read operation, data from each I/O node must be scattered to the various compute nodes, and on the compute nodes, data from the various I/O nodes must be gathered into the read buffer. A write request is similar, but the data flow is reversed.

The other critical component of a parallel file system is the file metadata. All file systems must maintain file metadata, which indicates important properties of the file such as its name, owner, access permissions, and type. In a parallel file system, additional information relating the physical distribution of the data must also be maintained. This includes which I/O nodes hold the file data and how the data is split among the I/O nodes. Special files such as directories and symbolic links more closely resemble metadata than true files and may be stored as such. At least one node on a parallel computer must act as a *manager node* where file metadata is stored. Metadata can be stored in a single location or distributed much as file data is distributed. Generally, all metadata for a single file is stored in a single location and is not distributed as file data is.

How is PFS different from NFS or Samba? In many ways a parallel file system is like a network file system such as NFS [30] or Samba [2]. These systems allow applications running on one system, the client system, to access file data stored on another system, the server. While it is possible to use more than one server with NFS or Samba, only one server manages a given file system (or subdirectory) for the client at a time. Individual files are stored entirely on the one server, along with their metadata. See Chapter 6 for more information on file systems such as these.

In a parallel file system, on the other hand, the data representing a single file is distributed among several servers. In addition, the file metadata may be stored on yet another server. This feature allows a parallel file system to take advantage of multiple I/O subsystems to achieve a performance gain. A parallel file system may provide various interfaces that allow the programmer to access file data in different ways. Moreover, a parallel file system may provide a choice of access semantics that affect how multiple tasks coordinate access to file data. This aspect is significant in that it determines when and where data read or written by multiple tasks is stored in the file and how concurrent update of a file is coordinated or synchronized. In contrast, the semantics and interfaces implemented for network file systems such as NFS and Samba generally do not allow for efficient and deterministic access in the presence of concurrent accesses.

Another important difference between general-purpose network file systems and a parallel file system is in the performance characteristics relative to different workloads. Typical network file systems are designed for interactive workloads. Policies for such features as the minimum transfer unit and local caching are relatively conservative and favor frequent small accesses with high locality. A parallel file system, on the other hand, is more typically designed to provide high throughput for relatively large accesses and optimizes transfers based on the access patterns of parallel applications. Thus, while a parallel file system may functionally serve as a general-purpose network file system, the performance of a PFS in this role may be poor.

What is PVFS? The Parallel Virtual File System is an open source implementation of a parallel file system developed specifically for Beowulf clusters and the Linux operating system [3]. PVFS runs as a set of user-level daemons and libraries and includes an optional Linux kernel module that allows standard utilities to access PVFS files. PVFS provides a simple striping distribution that allows the user to specify the stripe size and the number of I/O nodes on which data is striped on a per file basis. PVFS also provides a POSIX-like interface that allows transparent access to PVFS files and a number of other interfaces that offer more powerful and flexible access to the PVFS request mechanism. These interfaces are accessed via user-level libraries to provide the best access performance. Additionally, there is at least one implementation of the MPI-IO interface for PVFS.

The PVFS software consists of three daemons. The first is the `mgr` daemon, which maintains metadata for a file system. One copy of this daemon runs for a given file system. Second is the `iod` daemon, which services requests to read and write file data. One of these daemons runs on each node that will act as an I/O

node. The third daemon, `pvfsd`, works with the optional kernel module to perform PVFS requests for the kernel. One of these daemons is needed on each client node if and only if that node will make use of the optional kernel module interface.

The PVFS client interface is a library in both shared (`libpvfs.so`) and static (`libpvfs.a`) versions. This library also includes the other user interfaces such as the multidimensional block interface (MDBI) and the partitioned file interface (discussed in Section 17.2.1). These are needed on the node(s) used to compile PVFS applications and, in the case of the shared library, on each client node. The optional PVFS kernel module allows a PVFS file system to be mounted to provide transparent use by client applications. All file system requests are relayed via the `pvfsd` on the client node. This module must be loaded into the kernel on each client node that will have non-PVFS applications access PVFS file systems. A good example of such applications are utilities such as `ls`, `cat`, `cp`, and `mv`.

PVFS uses the Berkeley sockets interface to communicate between the clients and servers via TCP/IP (see Chapter 6 for more information on sockets and TCP/IP). In general, a PVFS application will establish a TCP connection with each `iod` for each open file. The `iods` store file data on the I/O nodes under one subdirectory, using whatever file system that subdirectory is implemented with to store the actual data. File metadata is stored on the node running the `mgr` daemon in standard Unix files, one per file in the PVFS file system. Each file in this metadata directory has a unique `inode` number on that node. That `inode` number is used to identify each segment of the file data on each of the I/O nodes.

Clients access the PVFS manager in order to obtain metadata for PVFS files. Using this metadata, they can then directly access data stored on I/O servers. The interface used by the application defines the semantics of this access and constrains how data might be described; we will cover interface options later in this and the following sections.

17.1.2 Setting Up a Parallel File System

The first step in configuring a PFS is deciding what role each node in the system will play, that is, which nodes will act as I/O nodes, which node will serve the metadata, and which nodes will be compute nodes. While assigning these roles may seem to be fairly straightforward, in fact several different approaches are possible. As is often the case, different approaches may be more suitable to different applications. This section outlines the concepts related to setting up a parallel file system. Specific details of administering PVFS are covered in Section 17.3.

How do I decide on a configuration? Typically, the first consideration in configuring a parallel file system is deciding how many I/O nodes will be used. This can reasonably vary from selecting a single I/O node to using all of the nodes in the system as I/O nodes. The primary consideration for a node to be an I/O node is that each I/O node must have at least one disk drive. Technically an I/O node can serve data from a remote mounted disk, but this approach will create unneeded network traffic. For many clusters, each node has a disk installed, and in this case it is reasonable to make every node an I/O node. Note that files need not be distributed to every I/O node in the network, but making each node an I/O node allows every node to be used for I/O. Alternatively, some cluster designs may include larger disks or additional disks on a subset of the nodes or may include disks on only a few of the nodes. In these cases it is more reasonable to limit the set of I/O nodes to those nodes that naturally lend themselves to that role because of their available resources.

A related issue is whether the applications intend to use some or all of the nodes as both I/O nodes and compute nodes at the same time. Here, one extreme is to use every node in the system in both roles, and the other extreme is to have two distinct sets of nodes: one for computation and one for I/O. This choice may reflect having the budget available to build specialized nodes and having a strong sense of the needs of the application(s). Ideally, these considerations can be addressed at the time the cluster is built so that node hardware will suitably match the role that is planned for each node.

What are the key components of an I/O node? If node hardware is to be tuned to the role of the node, hardware that may impact a node's role includes the disks, device bus, network interface, processors, and memory. The guidelines given here should be considered in light of the material in Chapter 3, where a detailed discussion of hardware issues for Beowulf machines is given. An I/O node needs to provide a certain amount of disk I/O throughput, which depends on both the ratio of I/O nodes to compute nodes and the required I/O throughput needed by the applications. In selecting an I/O node, the choice of SCSI versus IDE disks, number of disks, number of disk controllers, and configuration of the peripheral bus (such as PCI) are all important. An I/O node also needs enough network throughput to deliver the available disk throughput to all of the compute nodes. In fact, it is often desirable that an I/O node be able to deliver more throughput via the network than is available at the disk subsystem, because caching often allows for bursts of traffic that exceed the throughput limits of the disk subsystem. This implies not only that the network interface can provide the required throughput but that the peripheral

bus can support both the network and disk I/O load. This also has implications for the choice of network itself.

For dedicated I/O nodes, less powerful CPUs than used in compute nodes might be more cost effective. On the other hand, nodes that serve as both compute and I/O nodes tend to need even more CPU performance, since a certain amount of CPU load is required by the I/O servers in addition to the computational load. Experimental results have shown that having dual CPUs can be advantageous in both dedicated I/O nodes and nodes performing both roles.

17.1.3 Programming with a Parallel File System

Once a parallel file system is operational, data can be stored and accessed by using normal programs and utilities. To really take advantage of a parallel file system, however, you must understand some concepts key to efficient parallel file access and must use these concepts when implementing applications.

The POSIX I/O interface [18] is the standard for file and directory access on Linux systems. With this interface you may seek specific byte positions within a file and access contiguous regions of data at these positions. The semantics of the POSIX I/O interface defines what happens when applications write to overlapping regions as well, placing guarantees on the resulting file. This interface was created to serve as a standard for uniprocessing applications running on Unix machines accessing local file systems.

With parallel file systems, the costs associated with access are often different from those with local access. In particular, performing a single write or read operation often involves a noticeable amount of overhead required to initiate the operation. Thus, when large numbers of small accesses are performed in a parallel I/O environment, performance often suffers. Parallel applications often divide data among processes. This division often leads to situations where different processes want to read from a file a number of records that are not located in a contiguous block [24]. Since each access involves substantial overhead, it would be more efficient to access these regions with a single operation [33] and also more convenient for the programmer. To address this situation, parallel file systems typically offer an interface that allows a large number of noncontiguous regions to be accessed with a single I/O request.

Many parallel applications alternate between I/O and computation. In these types of application a large number of I/O-related operations occur simultaneously, often to the same file. If the operations are all performed independently, it is difficult for the underlying file system to identify and optimize for the access pattern. Grouping these accesses into a “collective access” can allow for further optimization.

Collective I/O is a mechanism provided by many parallel file systems where a single global request to the file system allows each task to access different data in the same file. Collective I/O has been provided in a number of special-purpose interfaces such as PASSION [33] and Panda [28] and is provided in MPI-IO [11]. Collective I/O to PVFS file systems is currently supported through the use of the ROMIO MPI-IO interface [34] (also discussed later in this chapter).

The POSIX semantics also imposes strict requirements on the order in which operations may be completed on I/O servers in a parallel environment. The semantics cripples the ability to perform I/O in parallel in many situations. Interfaces with more relaxed semantics provide more opportunities for parallelism by eliminating the overhead associated with atomicity of access; the application programmer ensures consistency instead.

All of these concepts, aggregating requests, collective I/O, and access semantics boil down to making the best use of available interfaces. We cover these next.

What other interfaces do I have to work with? Many interfaces and optimizations have been proposed and implemented to address the three issues of noncontiguous access, collective I/O, and the semantics of parallel access. These interfaces can be categorized into POSIX-like interfaces, general-purpose parallel I/O interfaces, and specialized (or application-specific) interfaces.

POSIX-like interfaces use the POSIX interface as their basis and provide various extensions in order to better describe parallel access. One such extension is file partitioning. File partitioning allows the interface to create a "view" of a file that contains of some subset of the data in the file. Partitions are usually based on some systematic subsetting of the file, for example, "Starting with the second byte of the file, view every fourth byte of the file." A more useful view might consider the file as a sequence of n byte records and take every pth record starting with record k. If there are p parallel tasks, and each creates a partition with the same value of n but with a different value of k ranging from 0 to $p-1$, then the set of partitions will cover the entire file, and none of the partitions will overlap. This is one way to evenly distribute file data among p tasks.

With an interface that supports partitioning, once the partition is defined, the program accesses the data in the partition as if it is the only thing in the file. In other words, any data not in the partition no longer appears in the file, and the data in the partition appears to be contiguous in the partitioned file. This is a convenient abstraction in that all of the information regarding the distribution of data among the tasks in located only in the part of the program that creates the partition. This can be especially nice in converting sequential programs to parallel

programs because sometimes all that is needed is to set up the partition and then let the program run as originally written. Partitioning is also one way that the programmer can specify a large set of noncontiguous accesses with a single request, which may allow the file system opportunities for optimization. This optimization would be impossible if instead `seek()` were used to access each contiguous region one at a time.

File partitioning has been included in the Vesta interface [5] and MPI-IO [11] (as *file views*), among others, and is supported by PVFS. Some partitioning interfaces require all tasks accessing a file to specify a common partitioning scheme that provides no overlap and complete file coverage. Other interfaces, such as the one implemented for PVFS, may allow individual tasks to specify different partitions, including those that overlap or that do not cover all of the data in the file. Details of the PVFS partitioning interface are given in Section 17.2.

Some applications require that output data from different tasks be interleaved, and yet cannot predict how much data each task will produce for each output. In this case a shared file pointer can be an effective mechanism for coordinating file I/O. With a shared file pointer, sequential access proceeds not from the last location accessed by the task performing I/O but from the last location accessed by any task. In one variation, a shared file pointer can be used with a collective operation, and each task's data is stored in the file in task order.

Shared file pointer interfaces are useful for low-volume I/O such as log messages and infrequent tracing data. A collective shared file pointer interface can also be useful for applications that synchronize well but generate data of varying size. Care must be taken in using a shared file pointer interface, however, since some implementations entail substantial overhead or result in serialized access. Shared file pointers have been provided by the Intel PFS, Vesta [5], and MPI-IO [11]. PVFS does not currently support shared file pointers.

Extensions or modifications to the POSIX interface such as partitioning and shared file pointers can help in providing more usable and higher-performance interfaces for some applications. However, more flexible interfaces can allow for even more convenient access and higher performance for a broader range of applications. One such interface is MPI-IO.

MPI-IO is a parallel I/O interface defined as part of the MPI-2 specification [11]. It is supported by a number of parallel file systems and provides all of the features described above: partitioning, collective operations, a mechanism to create noncontiguous I/O requests, and relaxed access semantics. MPI-IO is a very powerful application-independent interface and is probably the interface of choice for MPI programs. PVFS supports MPI-IO through the use of ROMIO, an MPI-IO imple-

mentation built on top of MPI [34]. Details of MPI-IO are given in Chapter 10 and in [14, Chapter 3].

A number of I/O interfaces also have been designed for special purposes. Examples include Panda [28], HDF5, and PVFS's multidimensional block interface (MDBI) [4]. MDBI provides special methods to allow you to define a regularly spaced n-dimensional data set that will be stored in the file. This description is similar to that of an n-dimensional array of some base type. Once this definition is in place, you can access any element of the data set with its corresponding coordinates. Additional methods allow you to control buffering of elements from the data set. Details of MDBI are given below in Section 17.2.1.

How do I tune my application for high performance? Parallel applications that use a parallel file system typically work by distributing the data in the file among the application's tasks. Exactly how this is done will vary from application to application, but in many cases there are natural divisions in the data. For example, a file may contain an array of records each of which includes several scalar data values. The records may represent a linear sequence or may represent a matrix of two or more dimensions. In the former case, the record boundaries are natural divisions. Thus most tasks will read a multiple of the record size (in bytes) from the file. In the latter case, the rows and columns are perhaps more natural divisions. Thus a task may read one or more whole rows at a time.

With such an application it is important to consider how the data will be distributed. Often this means considering these natural boundaries when selecting a stripe size for file distribution (assuming a striped physical data distribution). If the stripe size is matched to the natural boundaries in the data set, accesses by applications tend to be more uniformly distributed across the I/O servers. One pitfall is that many files include a header that is not the same size as a record. In this case the presence of the header can force the data not to align properly on the I/O nodes, and poor performance can result. The most common solution is to pad the header with blank space to the next physical boundary (depending on the distribution) so that the records will align. Another solution is to store the header information in a separate file or at the end of the file instead of the beginning.

PVFS performs better when multiple clients are accessing the *same* file, rather than each client accessing its own file. This is because the underlying storage systems respond better to single-file traffic. Thus, for PVFS it is best to aggregate output data into a single file; luckily this is often the most convenient option.

The next consideration is how the data accessed by each task is spread across the I/O nodes. When using PVFS, it is often best to store and access data distributed

across multiple I/O servers with single requests. This approach makes the best use of the underlying network by moving data through multiple TCP streams. This same approach applies in the case of multiple simultaneous accesses as well; just as PVFS clients perform more efficiently when exchanging data with multiple servers, PVFS servers perform at their best when servicing multiple clients simultaneously. What this means is that some experimentation is often necessary to determine the optimal matching of natural boundaries to the stripe; sometimes it is more efficient to place multiple rows, for example, in a single stripe. All that said, there are also advantages to accessing data residing locally when an overlapped compute/server environment is in place, especially when the interconnect in the cluster is relatively slow.

We discussed the coordination of access and the advantages of collective I/O earlier in this section. It is important to note, however, that there are cases when collective I/O *doesn't* pay off. In particular, if an application spends a great majority of its time computing, it might be more optimal to stagger the tasks' accesses to the I/O system. This approach lightens the load on the I/O servers and simultaneously allows more time for the I/O as a whole to take place. In all cases it is important to attempt to keep all I/O servers busy.

We emphasize that not all parallel file systems share these characteristics. Many such systems do not perform well with large numbers of clients simultaneously accessing single servers or simultaneously accessing the same file [19]. You should consider these issues when moving an application from one platform to another.

What is out-of-core computation? Another issue to consider is how the file system will be used in a computation. Some applications tend to read an input data file and produce an output data file. Other applications might not do much I/O at all, except that they require an extremely large amount of storage for their data structures and cannot keep everything in memory. In this case, the parallel file system can be used as a large shared-memory region using a technique known as out-of-core (OOC) computation. OOC techniques are similar to virtual memory use in that data is moved between disk and memory as needed for computation. Unlike virtual memory, however, OOC requires the programmer to explicitly read and write data rather than doing it transparently via a demand paged memory management system. While using virtual memory is much easier than OOC, a well-designed OOC program will outperform a virtual memory program by better utilizing memory and by moving data in and out of memory in large blocks, rather than moving a page at a time as demand paging would.

Facilities that support OOC allow the programmer to describe the data structures

stored in the parallel file and then access them in a manner consistent with that logical view of the data. For example, the MDBI provided by PVFS (discussed in Section 17.2.1) allows the program to describe a file as a multidimensional array and then read and write blocks of that array by using array subscripts.

Could you give me a tuning example? Many parallel applications involve regularly structured data that is evenly distributed to the tasks. Records can be assigned to tasks in groups. These groups may consist of contiguous records or non-contiguous but regularly spaced records. For example, consider an image-processing application. Each scan line of the image can be considered a record. Suppose the image has 512 scan lines, and you are using 16 processors. One distribution would be to assign the first 32 scan lines to the first processor, the second 32 to the second processor, and so on. Another distribution would be to assign one of each of the first 16 scan lines to each processor, in order, and then start over again and assign a second scan line from the second set of 16. A third option might be to assign 8 scan lines to each processor until the first 128 scan lines are assign, and then start over and assign the next 128, 8 at a time. In each case each processor gets 32 scan lines. The advantages and disadvantages of each depend on the algorithm being implemented and are beyond the scope of this book. Suffice it say each is a potentially useful distribution.

This example illustrates the concept known as "strided access." In strided access, a task accesses a contiguous region of data (in this case one or more scan lines), skips a region, and then accesses another region of the same size as the first, skips a region the same size as the first, and so on. Strided access can be used to access rows in a matrix, groups of rows in a matrix, columns in a matrix, groups of columns in a matrix, and other patterns. Because this is such a common distribution, many interfaces, including the PVFS library and MPI-IO, provide mechanisms to issue requests for strided accesses.

As alluded to above, the choice of distribution can affect algorithm performance, but it can also have an impact on I/O performance in that it can affect how I/O requests are distributed among the I/O nodes. Continuing the above example, assume each scan line of the image has 512 pixels and each pixel is 4 bytes. Thus an image is a total of 1 MByte. Suppose the file system uses a striped distribution, and you choose a 32 KByte strip size, which results in stripes of 512 KBytes.

The the image will be evenly distributed to all 16 disks, two 32 KByte blocks per node. Now consider the three distributions discussed above for the computational tasks. In the first, each task will access 32 scan lines of 2 KBytes each, which will be evenly distributed across 2 disks; 2 tasks will share 2 disks (consider disks 0 and

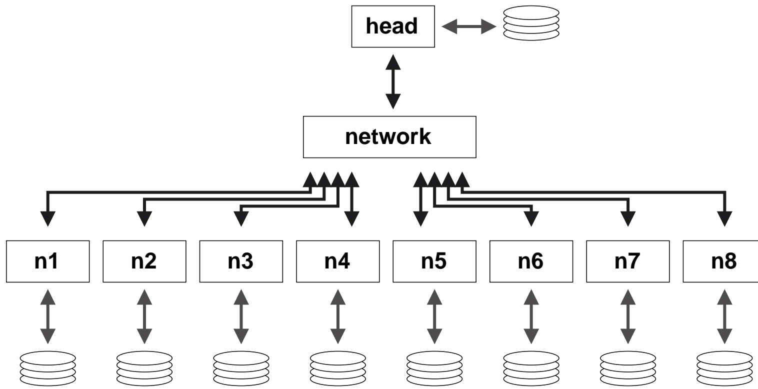

Figure 17.1
Example system.

1, which hold scan lines 0 to 15, 16 to 31, 128 to 143, and 144 to 159, which will map to tasks 0, 0, 8, and 8, respectively). In the second distribution, each task will access 2 scan lines from each of all 16 disks. Thus all 16 disks will service requests from all 16 tasks. In the third distribution, each task's data will map to 4 disks, and each disk will service 4 tasks. Thus the choice of distribution and stripe size affects how requests are spread across disks.

17.2 Using PVFS

In this section we discuss the options for accessing PVFS file systems from applications. We assume that a PVFS file system is already configured and available for use.

For the purposes of discussion we will pretend that PVFS is running on a small cluster of nine machines. In our example system, shown in Figure 17.1, there is one "head" node, called head, and eight other nodes, n1–n8. Each of these systems has a local disk, they are connected via some switch or hub, IP is up and running, and they can talk to each other via these short names. We will come back to this example throughout this chapter in order to clarify installation, configuration, and usage issues. This section, and the following one, are rife with example shell interactions in order to show exactly how things are configured.

17.2.1 Writing PVFS Programs

Programs written to use normal Unix I/O will work fine with PVFS without any changes. Files created this way will be striped according to the file system defaults set at compile time, usually set to a 256 KByte stripe size across all of the I/O nodes, starting with the first node listed in the `.iodtab` file. We note that use of Unix system calls `read()` and `write()` results in exactly the data specified being exchanged with the I/O nodes each time the call is made. Large numbers of small accesses performed with these calls will not perform well at all. In contrast, the buffered routines of the standard I/O library `fread()` and `fwrite()` locally buffer small accesses and perform exchanges with the I/O nodes in chunks of at least some minimum size. Utilities such as `tar` have options (e.g., `--block-size`) for setting the I/O access size as well. Generally PVFS will perform better with larger buffer sizes.

For applications using stdio routines to access PVFS files, you may want to increase the buffer size used by `fread()` and `fwrite()`. The `setvbuf()` call may be used to specify the buffering characteristics of a stream (`FILE *`) after opening. This call must be made before any other operations are performed, for example,

```
FILE *fp;
fp = fopen("foo", "r+");
setvbuf(fp, NULL, _IOFBF, 256*1024);
/* now we have a 256K buffer and are fully buffering I/O */
```

See the man page on `setvbuf()` for more information.

This transparent access involves significant overhead both due to data movement through the kernel and due to our user-space client-side daemon (`pvfsd`). To get around this, the PVFS libraries can be used either directly (via the native PVFS calls) or indirectly (through the ROMIO MPI-IO interface, the MDBI interface, or some higher-level interface such as HDF5).

In this section we begin by covering how to write and compile programs with the PVFS libraries. Next we cover how to specify the physical distribution of a file and how to set logical partitions. Following this we cover the MDBI interface. Finally we touch upon the use of ROMIO with PVFS. In addition to these interfaces, it is important to know how to control the physical distribution of files. In the next three sections, we will discuss how to specify the physical partitioning, or striping, of a file, how to set logical partitions on file data, and how the PVFS multidimensional block interface can be used.

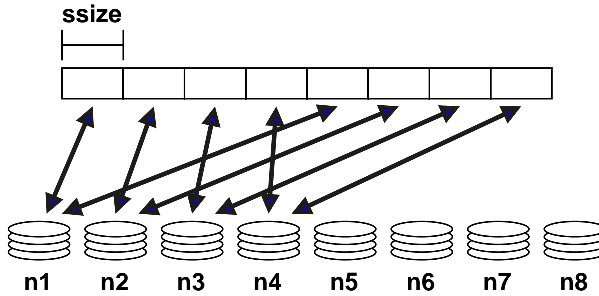

Figure 17.2
Striping example with base of 0 and `pcount` of 4

Preliminaries. When compiling programs to use PVFS, you should include in the source the PVFS `include` file, typically installed in `/usr/local/include/`, by

```
#include <pvfs.h>
```

To link to the PVFS library, typically installed in `/usr/local/lib/`, you should add `-lpvfs` to the link line and possibly `-L/usr/local/lib` to ensure that the directory is included in the library search.

The PVFS interface calls will also operate correctly on standard, non-PVFS, files, including the MDBI interface. When you are debugging code, this feature can help isolate application problems from bugs in the PVFS system.

Specifying Striping Parameters. The current physical distribution mechanism used by PVFS is a simple striping scheme. The distribution of data is described with three parameters:

base — index of the starting I/O node, with 0 being the first in the file system

pcount — number of servers on which data will be stored (partitions, a misnomer)

ssize — strip size, the size of the contiguous chunks stored on I/O servers

In Figure 17.2 we show an example where the base node is 0 and the `pcount` is 4 for a file stored on our example PVFS file system. As you can see, only four of the I/O servers will hold data for this file, because of the striping parameters.

Physical distribution is determined when the file is first created. Using `pvfs_open()`, you can specify the following parameters:

```
pvfs_open(char *pathname, int flag, mode_t mode);
pvfs_open(char *pathname, int flag, mode_t mode,
          struct pvfs_filestat *dist);
```

If the first set of parameters is used, a default distribution will be imposed. If, instead, a structure defining the distribution is passed in and the `O_META` flag is OR'd into the `flag` parameter, you can define the physical distribution via the `pvfs_filestat` structure passed in by reference as the last parameter. This structure is defined in the PVFS header files as follows:

```
struct pvfs_filestat {
   int base;   /* The first iod node to be used */
   int pcount; /* The number of iod nodes for the file */
   int ssize;  /* stripe size */
   int soff;   /* NOT USED */
   int bsize;  /* NOT USED */
}
```

The `soff` and `bsize` fields are artifacts of previous research and are not in use at this time. Setting the `pcount` value to -1 will use all available I/O daemons for the file. Setting -1 in the ssize and base fields will result in the default values being used (see Section 17.3.4 for more information on default values).

To obtain information on the physical distribution of a file, you should use `pvfs_ioctl()` on an open file descriptor:

```
pvfs_ioctl(int fd, GETMETA, struct pvfs_filestat *dist);
```

It will fill in the `pvfs_filestat` structure with the physical distribution information for the file. On the command line the `pvstat` utility can be used to obtain this information (see Section 17.2.2).

Setting a Logical Partition. The PVFS logical partitioning system allows you to describe the regions of interest in a file and subsequently access those regions efficiently. Access is more efficient because the PVFS system allows disjoint regions that can be described with a logical partition to be accessed as single units. The alternative would be to perform multiple seek-access operations, which is inferior because of both the number of separate operations and the reduced data movement per operation. PVFS logical partitioning is an implementation of file partitioning; it is named "logical" because it is independent of any physical characteristics of the file (such as the stripe size).

If applicable, logical partitioning can also ease parallel programming by simplifying data access to a shared data set by the tasks of an application. Each task can set up its own logical partition, and once this is done all I/O operations will "see" only the data for that task.

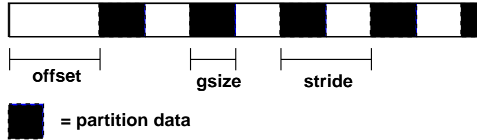

Figure 17.3
Partitioning parameters.

With the current PVFS partitioning mechanism, partitions are defined with three parameters: *offset*, group size (*gsize*), and *stride*. The offset is the distance in bytes from the beginning of the file to the first byte in the partition. Group size is the number of contiguous bytes included in the partition. Stride is the distance from the beginning of one group of bytes to the next. Figure 17.3 shows these parameters.

To set the file partition, the program uses a `pvfs_ioctl()` call. The parameters are as follows:

```
pvfs_ioctl(fd, SETPART, &part);
```

where `part` is a structure defined as follows:

```
struct fpart {
   int offset;
   int gsize;
   int stride;
   int gstride; /* NOT USED */
   int ngroups; /* NOT USED */
};
```

The last two fields, `gstride` and `ngroups`, are remnants of previous research, are no longer used, and should be set to zero. The `pvfs_ioctl()` call can also be used to get the current partitioning parameters by specifying the `GETPART` flag. Note that whenever the partition is set, the file pointer is reset to the beginning of the new partition. Also note that setting the partition is a purely local call; it does not involve contacting any of the PVFS daemons. Thus it is reasonable to reset the partition as often as needed during the execution of a program. When a PVFS file is first opened, a "default partition" is imposed on it that allows the process to see the entire file.

As an example, suppose a file contains 40,000 records of 1,000 bytes each, there are four parallel tasks, and each task needs to access a partition of 10,000 records each for processing. In this case you would set the group size to 10,000 records

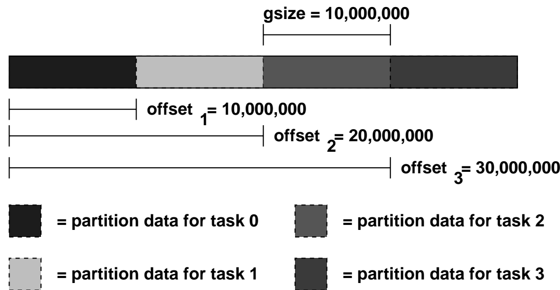

Figure 17.4
Partitioning Example 1, block distribution.

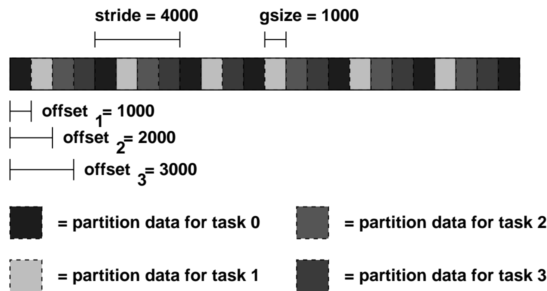

Figure 17.5
Partitioning Example 2, cyclic distribution.

times 1,000 bytes, or 10,000,000 bytes. Then each task would set its offset so that it would access a disjoint portion of the data. This is shown in Figure 17.4.

Alternatively, suppose you want to allocate the records in a cyclic, or "round-robin," manner. In this case the group size would be set to 1,000 bytes, the stride would be set to 4,000 bytes, and the offsets would again be set to access disjoint regions, as shown in Figure 17.5.

Setting the partition for one task has no effect whatsoever on any other tasks. There is also no reason for the partitions set by each task to be distinct; the partitions of different tasks can be overlapped, if desired. Finally, no direct relationship exists between partitioning and striping for a given file; while it is often desirable to match the partition to the striping of the file, you have the option of selecting any partitioning scheme independent of the striping of a file.

Simple partitioning is useful for one-dimensional data and simple distributions of two-dimensional data. More complex distributions and multidimensional data are often more easily partitioned by using the multidimensional block interface.

Using Multidimensional Blocking. The PVFS multidimensional block interface provides a slightly higher level view of file data than does the native PVFS interface. With the MDBI, file data is considered as an n-dimensional array of records. This array is divided into "blocks" of records by specifying the dimensions of the array and the size of the blocks in each dimension. The parameters used to describe the array are as follows:

D — number of dimensions

rs — record size

nb_n — number of blocks (in each dimension)

ne_n — number of elements in a block (in each dimension)

bf_n — blocking factor (in each dimension), described later

Once you have defined the view of the data set, blocks of data can be read with single function calls, greatly simplifying the act of accessing these types of data sets. This is done by specifying a set of *index* values, one per dimension.

Five basic calls are used for accessing files with MDBI:

```
int open_blk(char *path, int flags, int mode);

int set_blk(int fd, int D, int rs, int ne1, int nb1, ..., int nen,
        int nbn);

int read_blk(int fd, char *buf, int index1, ..., int indexn);

int write_blk(int fd, char *buf, int index1, ..., int indexn);

int close_blk(int fd);
```

The `open_blk()` and `close_blk()` calls operate similarly to the standard Unix `open()` and `close()`. The call `set_blk()` is used to set the blocking parameters for the array before reading or writing; this process will be described in a moment. It can be used as often as necessary and does not entail communication. The two calls `read_blk()` and `write_blk()` are used to read blocks of records once the blocking has been set.

Figure 17.6 gives an example of blocking. Here a file has been described as a two-dimensional array of blocks, with blocks consisting of a two by three array of records. Records are shown with dotted lines, with groups of records organized into blocks denoted with solid lines.

In this example, the array would be described with a call to `set_blk()` as follows:

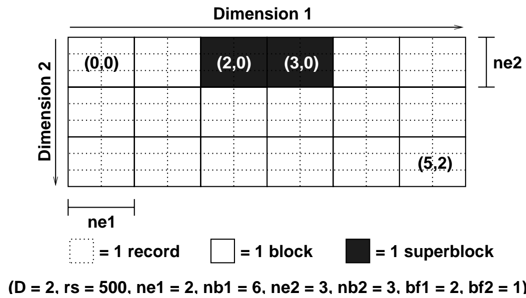

Figure 17.6
MDBI Example 1.

```
set_blk(fd, 2, 500, 2, 6, 3, 3);
```

If you wanted to read block (2, 0) from the array, you could then

```
read_blk(fd, &buf, 2, 0);
```

Similarly, to read block (5, 2), you could use

```
write_blk(fd, &blk, 5, 2);
```

A final feature of the MDBI is block buffering. Sometimes multidimensional blocking is used to set the size of the data that the program wants to read and write from disk. Other times the block size has some physical meaning in the program and is set for other reasons. In this case, individual blocks may be rather small, resulting in poor I/O performance and underutilization of memory. MDBI provides a buffering mechanism that causes multiple blocks to be read and written from disk and stored in a buffer in the program's memory address space. Subsequent transfers using `read_blk()` and `write_blk()` result in memory-to-memory transfers unless a block outside of the current buffer is accessed.

Since it is difficult to predict what blocks should be accessed when, PVFS relies on user cues to determine what to buffer. This is done by defining "blocking factors" that group blocks together. The *blocking factor* indicates how many blocks in the given dimension should be buffered. A single function is used to define the blocking factor: `int buf_blk(int fd, int` bf_1`, ..., int` bf_n`)`.

Looking at Figure 17.6 again, we can see how blocking factors can be defined. In the example, the call

```
buf_blk(fd, 2, 1);
```

is used to specify the blocking factor. We denote the larger resulting buffered blocks as *superblocks* (a poor choice of terms in retrospect), one of which is shaded in the example.

Whenever a block is accessed, if its superblock is not in the buffer, the current superblock is written back to disk (if dirty), and the new superblock is read in its place; then the desired block is copied into the given buffer. The default blocking factor for all dimensions is 1, and any time the blocking factor is changed the buffer is written back to disk if dirty.

We emphasize that no cache coherency is performed here; if application tasks are sharing superblocks, unexpected results will occur if writes are performed. The user must ensure that this does not happen. A good strategy for buffering is to develop a program with buffering turned off, and then enable it later in order to improve performance.

Using the ROMIO MPI-IO Implementation. The MPI specification provides a de facto standard for message passing in parallel programs. The MPI-2 specification, which builds on the successful MPI-1 specification, includes a section on I/O that is commonly referred to as MPI-IO [14, 11]. Just as MPI has become the interface of choice for message passing in parallel applications, the MPI-IO interface has become a prominent low-level interface for I/O access in parallel applications.

ROMIO is one implementation of the MPI-IO interface [34]. ROMIO is unique in that it implements an abstract I/O device (ADIO) layer that aids in porting ROMIO to new underlying I/O systems. The success of this design is evident in the number of systems for which ROMIO provides MPI-IO support, including HP, IBM, NEC, SGI, and Linux.

In addition to merely mapping MPI-IO operations into the correct underlying operations, ROMIO implements two important optimizations that can be of great benefit in a number of scenarios. The first of these is *data sieving* [33], which allows ROMIO to take advantage of MPI-IO noncontiguous requests by accessing larger, contiguous regions containing desired data with single calls. Combining many small accesses into a single large one is a very effective optimization in many I/O systems. The second of these optimizations is a collective optimization termed *two-phase I/O* [32]. The goal of two-phase I/O is to more optimally access the disk. In collective operations, multiple processes often will read small adjoining regions from a single server. Two-phase I/O combines these adjoining accesses into a single access by a single process. Data for the access is then scattered (in the read case) or first gathered (in the write case) in order to attain the desired distribution.

In Linux clusters ROMIO can be configured to operate on top of PVFS, providing applications using the MPI-IO interface direct access to PVFS file systems. This strategy allows applications to use this high-performance I/O option without constraining application programmers to using the PVFS interface. Chapter 10 includes details on using MPI-IO in applications.

The MPI hints mechanism may be used to pass striping information to PVFS through the MPI-IO interface. Hints are passed by the "info" object that is an argument to `MPI_File_open()`. The following three keywords are valid for PVFS:

striping_unit — ssize value

striping_factor — `pcount` value

start_iodevice — base value

When using ROMIO with PVFS, you must be aware of three important cases. First, if ROMIO was 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 was 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 *must* be prefixed with `pvfs:` to indicate that the file is a PVFS file; otherwise ROMIO will not be able to find the file.

17.2.2 PVFS Utilities

A few utilities are provided for dealing with PVFS files and file systems.

Copying Files to PVFS. While the `cp` utility can copy files onto a PVFS file system, the user then loses control over the physical distribution of the file (the default is used). Instead, the `u2p` command supplied with PVFS can be used to copy an existing Unix file to a PVFS file system while specifying physical distribution parameters. The syntax for `u2p` is

```
u2p -s <stripe size> -b <base> -n <# of nodes> <srcfile> <destfile>
```

This function is most useful in converting pre-existing data files to PVFS so that they can be used in parallel programs. The `u2p` command relies on the existence of the `/etc/pvfstab` file to operate correctly.

Examining PVFS File Distributions. The pvstat utility will print out the physical distribution parameters for a PVFS file. After earlier creating a file on our example PVFS file system, we see

```
[root@head /root]# /usr/local/bin/pvstat /pvfs/foo
/pvfs/foo: base = 0, pcount = 8, ssize = 65536
```

The pvstat utility relies on the existence of the /etc/pvfstab file to operate correctly.

Checking on Server Status. The iod-ping utility can be used to determine whether a given I/O server is running:

```
[root@head /root]# /usr/local/bin/iod-ping -h n1 -p 7000
n1:7000 is responding.
[root@head /root]# /usr/local/bin/iod-ping -h head -p 7000
head:7000 is down.
```

In this case, we have started the I/O server on n1, so it is up and running. We are not running an I/O server on the head, so it is reported as down. Likewise the mgr-ping utility can be used to check the status of metadata servers:

```
[root@head /root]# /usr/local/bin/mgr-ping -h head -p 3000
head:3000 is responding.
[root@head /root]# /usr/local/bin/mgr-ping -h n1 -p 3000
n1:3000 is down.
```

The mgr is up and running on head, but we're not running one on n1.

These two utilities also set their exit values appropriately for use with scripts; in other words, they set their exit value to 0 on success (responding) and 1 on failure (down). If no additional parameters are specified, the program will automatically check for a server on localhost at the default port for the server type (7000 for I/O server, 3000 for metadata server). If the "-p" option is not specified, the default port is used.

No fsck currently exists for PVFS, although arguably there should be one.

17.3 Administering PVFS

In this section we cover the specifics of administering PVFS, first building the PVFS components, then installing and configuring the servers, next installing and configuring client software, and finally starting things up and verifying that the

system is operating. We continue to rely on the system described in Section 17.2 and shown in Figure 17.1 as an example.

17.3.1 Building the PVFS Components

The PVFS package has come a long way in the past few versions in terms of ease of compilation. The process is now fairly simple.

Two tar files are needed for compiling PVFS:

- `pvfs` (e.g., `pvfs-1.5.0.tgz`)
- `pvfs-kernel` (e.g., `pvfs-kernel-0.9.0.tgz`)

The first of these contains code for the PVFS servers and for the PVFS library. The second contains code specific to the Linux VFS support, which allows PVFS file systems to be mounted on Linux PCs. This code is not essential for using PVFS, but it makes accessing PVFS files much more convenient.

Obtaining the Source. PVFS is open source and is freely available on the Web. Currently there are two consistent sources for obtaining PVFS via the FTP protocol:

- `ftp://ftp.parl.clemson.edu:/pub/pvfs/`
- `ftp://mirror.chpc.utah.edu:/pub/pvfs/` (mirror site)

Within one of these directories are the files `pvfs-v1.tgz` and `pvfs-kernel-v2.tgz`, where `v1` and `v2` are version numbers. These files are `tar` archives of the PVFS source that have subsequently been compressed with the `gzip` tool. You should download the latest versions of each; the version numbers of the two packages will not match each other. (At the time of writing, the newest version of the pvfs archive was 1.5.1, and the newest version of the pvfs-kernel archive was 0.9.1.)

Untarring the Packages. It is a bit easier to perform the compilations if both the archives are untarred in the same directory, since the pvfs-kernel source relies on `include` files from the `pvfs` source tree. In our example, we will untar into `/usr/src/` on the head node:

```
[root@head /root]# cp pvfs-1.5.0.tgz pvfs-kernel-0.9.0.tgz /usr/src
[root@head /usr/src]# cd /usr/src
[root@head /usr/src]# tar xzf pvfs-1.5.0.tgz
[root@head /usr/src]# tar xzf pvfs-kernel-0.9.0.tgz
[root@head /usr/src]# ln -s pvfs-1.5.0 pvfs
[root@head /usr/src]# ls -lF
total 476
lrwxrwxrwx   1 root   root       15 Dec 14 17:42 pvfs -> pvfs-1.5.0/
```

```
drwxr-xr-x  12 root   root      512 Dec 14 10:11 pvfs-1.5.0/
-rw-r--r--   1 root   root   371535 Dec 14 17:41 pvfs-1.5.0.tgz
drwxr-xr-x   6 root   root     1024 Dec 14 10:10 pvfs-kernel-0.9.0/
-rw-r--r--   1 root   root   105511 Dec 14 17:41 pvfs-kernel-0.9.0.tgz
```

The symbolic link allows the pvfs-kernel package easily to find the `include` files it needs. Once this is finished, the source is ready to be compiled.

Compiling the Packages. Next we will compile the two packages. We also will discuss how to install the packages on the local system; however, most users will wish to wait and distribute the files to the correct machines after compiling is complete. In the Section 17.3.2 we discuss what components need to be where.

First we will compile the `pvfs` package (leaving out the output):

```
[root@head /usr/src]# cd pvfs
[root@head /usr/src/pvfs-1.5.0]# ./configure
[root@head /usr/src/pvfs-1.5.0]# make
```

Then the components can be installed:

```
[root@head /usr/src/pvfs-1.5.0]# make install
```

The following are installed by default:

- `mgr`, `iod` in `/usr/local/sbin/`
- `libpvfs.a` in `/usr/local/lib/`
- `include` files in `/usr/local/include/`
- test programs and utilities in `/usr/local/bin/`
- man pages in `/usr/local/man/`

These installation directories can be changed via options to `configure`. See `configure --help` in the package source directory.

The PVFS-kernel package will perform tests for features based on header files and the running kernel, so it is important that the desired the kernel be running and the matching header files available on the machine on which the compilation will take place. With the matching headers, compiling is easy. Again we omit the output of the compile process:

```
[root@head /usr/src/pvfs-1.5.0]# cd ../pvfs-kernel-0.9.0
[root@head /usr/src/pvfs-kernel-0.9.0]# ./configure \
--with-libpvfs-dir=../pvfs/lib
[root@head /usr/src/pvfs-kernel-0.9.0]# make
```

The configure option here lets the package know where it can find the PVFS I/O library `libpvfs.a`, which is used by this package. The `README` and `INSTALL` files in the source directory contain hints for working around common compilation problems.

Once these steps are complete, the kernel components can be installed:

```
[root@head /usr/src/pvfs-kernel-0.9.0]# make install
```

The following are installed by default:

- `/usr/local/sbin/pvfsd`
- `/sbin/mount.pvfs`

The program `mount.pvfs` is put in that location because that is the only location the system utility `mount` will look in for a file system-specific executable; this is covered in more detail in Section 17.3.5.

You must install `pvfs.o` in the right place. This is usually `/lib/modules/<kernel-version>/misc/`.

17.3.2 Installation

PVFS is a somewhat complicated package to get up and running. The reason is, in part, because it is a multicomponent system, but also because the configuration is a bit unintuitive. The purpose of this section is to shed light on the process of installing, configuring, starting, and using the PVFS system.

It is important to have in mind the roles that machines (a.k.a. nodes) will play in the PVFS system. Remember that there are three potential roles that a machine might play: metadata server, I/O server, and client. A machine can fill one, two, or all of these roles simultaneously. Each role requires a specific set of binaries and configuration information. There will be one metadata server for the PVFS file system. There can be many I/O servers and clients. In this section we discuss the components and configuration files needed to fulfill each role.

Again, we configure our example system so that the "head" node provides metadata service, the eight other nodes provide I/O service, and all nodes can act as clients.

For additional information on file system default values and other configuration options, see Section 17.3.4.

Directories Used by PVFS. In addition to the roles that a machine may play, three types of directories are used in PVFS. A great deal of confusion seems to surround these, so before we begin our example installation we will attempt to

dispel this confusion. The three types of directories are metadata directory, data directory, and mount point.

There is a single metadata directory for a PVFS file system. It exists on the machine that is filling the role of the metadata server. In this directory information is stored describing the files stored on the PVFS file system, including the owner of files, the permissions, and the way the files are distributed across I/O servers. Additionally, two special files are stored in this directory, `.iodtab` and `.pvfsdir`, which are used by the metadata server to find I/O servers and PVFS files.

There is a data directory on each I/O server. This directory is used to store the data that makes up PVFS files. The data is stored in individual files in a subdirectory hierarchy.

Finally there is a mount point on each client. This is an empty directory on which the PVFS file system is mounted and at which `pvfstab` files point (see Section 17.3.4). This empty directory is identical to any other mount point.

Installing and Configuring the PVFS Servers. Three files are necessary for a metadata server to operate:

- `mgr` executable
- `.iodtab` file
- `.pvfsdir` file

The `mgr` executable is the daemon that provides metadata services in the PVFS system. It normally runs as root. It must be started before clients attempt to access the system.

The `.iodtab` file contains an ordered list of IP addresses and ports for contacting I/O daemons (`iods`). Since this list is ordered, once it is created it must not be modified, because this can destroy the integrity of data stored on PVFS.

The `.pvfsdir` file describes the permissions of the directory in which the metadata is stored.

Both the `.iodtab` and `.pvfsdir` files may be created with the `mkmgrconf` script. In our example we will use the directory `/pvfs-meta` as our metadata directory.

```
[root@head /usr/src/pvfs-kernel-0.9.0]# cd /
[root@head /]# mkdir /pvfs-meta
[root@head /]# cd /pvfs-meta

[root@head /pvfs-meta]# /usr/local/bin/mkmgrconf
This script will make the .iodtab and .pvfsdir files
in the metadata directory of a PVFS file system.
```

```
Enter the root directory:
/pvfs-meta
Enter the user id of directory:
root
Enter the group id of directory:
root
Enter the mode of the root directory:
777
Enter the hostname that will run the manager:
localhost
Searching for host...success
Enter the port number on the host for manager:
(Port number 3000 is the default)
3000
Enter the I/O nodes:  (can use form node1, node2, ...  or
nodename#-#,#,#)
n1-8
Searching for hosts...success
I/O nodes:  n1 n2 n3 n4 n5 n6 n7 n8
Enter the port number for the iods:
(Port number 7000 is the default)
7000
Done!
```

```
[root@head /pvfs-meta]# ls -al
total 9
drwxr-xr-x    2 root    root           82 Dec 17 15:01 ./
drwxr-xr-x   21 root    root          403 Dec 17 15:01 ../
-rwxr-xr-x    1 root    root           84 Dec 17 15:01 .iodtab*
-rwxr-xr-x    1 root    root           43 Dec 17 15:01 .pvfsdir*
```

The `mkmgrconf` script is installed with the rest of the utilities.

I/O servers have their own executable and configuration file, distinct from client and metadata server files:

- `iod` executable
- `iod.conf` file

The `iod` executable is the daemon that provides I/O services in the PVFS system. It normally is started as root, after which time it changes its group and user to some nonsuperuser ids. These `iods` must be running in order for file I/O to take place.

The `iod.conf` file describes the `iod`'s environment. In particular it describes the location of the PVFS data directory on the machine and the user and group under which `iod` should run. There should be a comparable configuration file for the `mgr`, but there is not at this time.

In our example we're going to run our I/O server as user `nobody` and group `nobody`, and we're going to have it store data in a directory called `/pvfs-data` (this could be a mount point or a subdirectory and doesn't have to be this name). Lines that begin with a pound sign are comments. Here's our `iod.conf` file:

```
# iod.conf file for example cluster
datadir /pvfs-data
user nobody
group nobody
```

We then create the data directory and change the owner, group, and permissions to protect the data from inappropriate access while allowing the `iod` to store and retrieve data. We'll do this on our first I/O server (n1) first:

```
[root@n1 /]# cd /
[root@n1 /]# mkdir /pvfs-data
[root@n1 /]# chmod 700 /pvfs-data
[root@n1 /]# chown nobody.nobody /pvfs-data
[root@n1 /]# ls -ald /pvfs-data
drwx------    2 nobody   nobody         35 Dec  1 09:41 /pvfs-data/
```

This must be repeated on each I/O server. In our example case, the `/etc/iod.conf` file is exactly the same on each server, and we create the `/pvfs-data` directory in the same way as well.

Installing and Configuring Clients. Five files and one directory are necessary for a client to access PVFS file systems:

- `pvfsd` executable
- `pvfs.o` kernel module (compiled to match kernel)
- `/dev/pvfsd` device file
- `mount.pvfs` executable
- `pvfstab` file

- mount point

As mentioned in Section 17.2.1 there are two approaches to client access: direct library access and kernel support. The first four items in the above list are specifically for kernel access and may be ignored if this method of access is not desired.

The `pvfsd` executable is a daemon that performs network transfers on behalf of client programs. It is normally started as root. It must be running before a PVFS file system is mounted on the client.

The `pvfs.o` kernel module registers the PVFS file system type with the Linux kernel, allowing PVFS files to be accessed with system calls. This is what allows existing programs to access PVFS files once a PVFS file system is mounted.

The `/dev/pvfsd` device file is used as a point of communication between the `pvfs.o` kernel module and the `pvfsd` daemon. It must exist before the `pvfsd` is started. It need be created only once on each client machine.

The `mount.pvfs` executable is used by `mount` to perform the PVFS-specific mount process. Alternatively it can be used in a standalone manner to mount a PVFS file system directly. This will be covered in Section 17.3.3.

The `pvfstab` file provides an fstab-like entry that describes to applications using the PVFS libraries how to access PVFS file systems. This is *not* needed by the kernel client code. It is used only if code is directly or indirectly linked to `libpvfs`. This includes using the ROMIO MPI-IO interface.

The mount point, as mentioned earlier, is just an empty directory. In our example we are placing our mount point in the root directory so that we can mount our PVFS file system to `/pvfs`. We then create the PVFS device file. The `mknod` program is used to create device files, which are special files used as interfaces to system resources. The `mknod` program takes four parameters: a name for the device, a type ("c" for character special file in our case), and a major and minor number. We have somewhat arbitrarily chosen 60 for our major number for now.

We'll do this first on the head machine:

```
[root@head /]# mkdir /pvfs
[root@head /]# ls -ald /pvfs
drwxr-xr-x    2 root    root           35 Dec  1 09:37 /pvfs/
[root@head /]# mknod /dev/pvfsd c 60 0
[root@head /]# ls -l /dev/pvfsd
crw-r--r--    1 root    root      60,   0 Dec  1 09:45 /dev/pvfsd
```

If one is using the `devfs` system, it is not necessary to create the `/dev/pvfsd` file, but it will not hurt to do so.

In our example system we are going to use the PVFS libraries on our nodes, so we will also create the `pvfstab` file using `vi` or `emacs`. It's important that users be able to read this file. Here's what it looks like:

```
[root@head /]# chmod 644 /etc/pvfstab
[root@head /]# ls -al /etc/pvfstab
-rw-r--r--    1 root    root           46 Dec 17 15:19 /etc/pvfstab
[root@head /]# cat /etc/pvfstab
head:/pvfs-meta  /pvfs  pvfs  port=3000  0  0
```

This process must be repeated on each node that will be a PVFS client. In our example we would need to copy out these files to each node and create the mount point.

Installing PVFS Development Components. A few components should also be installed if applications are going to be compiled for PVFS:

- `libpvfs.a` library
- `include` files
- `man` pages
- `pvfstab` file

The `libpvfs.a` library and `include` files are used when compiling programs to access the PVFS system directly (what we term "native access"). The man pages are useful for reference. The `pvfstab` file was described in the preceding section; it is necessary for applications to access PVFS file systems without going through the kernel.

In our example we expect to compile some programs that use native PVFS access on our "head" node. By performing a `make install` in the PVFS source on the head, everything is automatically installed.

Configuring ROMIO to Use PVFS. In order for ROMIO to access PVFS files most optimally, it must be configured with PVFS support. Since ROMIO is typically installed as part of the MPICH package, a full coverage of the configuration, compilation, and installation process is outside the scope of this section. One should instead reference Section 9.6.1 for this information.

In short, when compiling ROMIO either as a standalone package or as part of MPICH, two important additional flags must be provided:

- `-file_system=pvfs+ufs+nfs`
- `-lib=/usr/local/lib/libpvfs.a`

The first of these specifies that ROMIO support regular Unix files, PVFS files, and NFS files. The second indicates the location of the PVFS library for linking to the ROMIO package.

17.3.3 Startup and Shutdown

At this point all the binaries and configuration files should be in place. Now we will start up the PVFS file system and verify that it is working. First we will start the server daemons. Then we will initialize the client software and mount the PVFS file system. Next we will create some files to show that things are working. Following this we will discuss unmounting file systems. Finally we will discuss shutting down the components.

Starting PVFS Servers. First we need to get the servers running. It doesn't matter what order we start them in as long as they are all running before we start accessing the system.

Going back to our example, we'll start the metadata server daemon first. It stores its log file in /tmp/ by default:

```
[root@head /root]# /usr/local/sbin/mgr
[root@head /root]# ls -l /tmp
total 5
-rwxr-xr-x    1 root   root            0 Dec 18 18:22 mgrlog.MupejR
```

The characters at the end of the log filename are there to ensure a unique name. A new file will be created each time the server is started.

Next we start the I/O server daemon on each of our I/O server nodes:

```
[root@n1 /root]# /usr/local/sbin/iod
[root@n1 /root]# ls -l /tmp
total 5
-rwxr-xr-x    1 root   root           82 Dec 18 18:28 iolog.n2MjK4
```

This process must be repeated on each node.

Getting a Client Connected. With the servers started, we can now start up the client-side components. First we load the module, then we start the client daemon pvfsd, and then we mount the file system:

```
[root@head /root]# insmod pvfs.o
[root@head /root]# lsmod
Module                  Size  Used by
```

```
pvfs                   32816   0  (unused)
eepro100               17104   1  (autoclean)
[root@head /root]# /usr/local/sbin/pvfsd
[root@head /root]# ls -l /tmp
total 7
-rwxr-xr-x    1 root   root           0 Dec 18 18:22 mgrlog.MupejR
-rwxr-xr-x    1 root   root         102 Dec 18 18:22 pvfsdlog.Wt0w7g
[root@head /root]# /sbin/mount.pvfs head:/pvfs-meta /pvfs
[root@head /root]# ls -al /pvfs
total 1
drwxrwxrwx    1 root   root          82 Dec 18 18:33 ./
drwxr-xr-x   20 root   root         378 Dec 17 15:01 ../
[root@head /root]# df -h /pvfs
Filesystem            Size  Used Avail Use% Mounted on
head:/pvfs-meta       808M   44M  764M   5% /pvfs
```

Now we should be able to access the file system. As an aside, we note that the `-h` option to `df` simply prints things in more human-readable form.

Checking Things Out. Let's create a couple of files:

```
[root@head /root]# cp /etc/pvfstab /pvfs/
[root@head /root]# dd if=/dev/zero of=/pvfs/zeros bs=1M count=10
[root@head /root]# ls -l /pvfs
total 10240
-rw-r--r--    1 root    root           46 Dec 18 18:41 pvfstab
-rw-r--r--    1 root    root     10485760 Dec 18 18:41 zeros
[root@head /root]# cat /pvfs/pvfstab
head:/pvfs-meta  /pvfs  pvfs  port=3000  0  0
```

Everything looks good. Now we must repeat this on the other nodes so that they can access the file system as well.

Unmounting File Systems and Shutting Down Components. As with any other file system type, if a client is not accessing files on a PVFS file system, you can simply unmount it:

```
[root@head /root]# umount /pvfs
[root@head /root]# ls -al /pvfs
total 1
drwxrwxrwx    1 root    root           82 Dec 18 18:33 ./
```

```
drwxr-xr-x   20 root    root          378 Dec 17 15:01 ../
```

You could then remount to the same mount point or some other mount point. It is not necessary to restart the `pvfsd` daemon or reload the `pvfs.o` module in order to change mounts.

For clean shutdown, all clients should unmount PVFS file systems before the servers are shut down. The preferred order of shutdown is as follows:

- unmount PVFS file systems on clients
- stop `pvfsd` daemons using `kill` or `killall`
- unload `pvfs.o` module using `rmmod`
- stop `mgr` daemon using `kill` or `killall`
- stop `iod` daemons using `kill` or `killall`

17.3.4 Configuration Details

As the preceding discussion suggests, the current PVFS configuration system is a bit complex. Here we try to shed more light on configuration file formats, file system defaults, and other options. This information is all supplementary, but it might be useful in the event of an error.

The `.pvfsdir` and `.iodtab` Files. In Section 17.3.2, we discussed the creation of both the `.pvfsdir` and the `.iodtab` files. In this section we cover the details of the file formats. The `.pvfsdir` file holds information for the metadata server for the file system. The `.iodtab` file holds a list of the I/O daemon locations and port numbers that make up the file system. Both of these files can be created by using the `mkmgrconf` script, whose use is also described in Section 17.3.2.

The `.pvfsdir` file is in text format and includes the following information in this order, with an entry on each line:

- `inode` number of the directory in which the `.pvfsdir` resides
- userid for the directory
- groupid for the directory
- permissions for the directory
- port number for metadata server
- hostname for the metadata server
- metadata directory name
- name of this subdirectory (for the `.pvfsdir` file in the metadata directory this will be “/”)

Here's a sample `.pvfsdir` file that might have been produced for our example file system:

```
116314
0
0
0040775
3000
head
/pvfs-meta
/
```

This file would reside in the metadata directory, which in our example case is `/pvfs-meta`. There will be a `.pvfsdir` file in each subdirectory under this as well. The metadata server will automatically create these new files when subdirectories are created.

The `.iodtab` file is also created by the system administrator. It consists simply of an ordered list of hosts (or IP addresses) and optional port numbers. Lines beginning with `#` are comments and are ignored by the system. It is stored in the metadata directory of the PVFS file system.

An example of a `.iodtab` file is as follows:

```
# example .iodtab file using IP addresses and explicit ports
192.168.0.1:7000
192.168.0.2:7000
192.168.0.3:7000
192.168.0.4:7000
192.168.0.5:7000
192.168.0.6:7000
192.168.0.7:7000
192.168.0.8:7000
```

Another example, assuming the default port (7000) and using hostnames (as in our example system), is the following:

```
# example .iodtab file using hostnames and default port (7000)
n1
n2
n3
n4
```

```
n5
n6
n7
n8
```

Manually creating .iodtab files, especially for large systems, is encouraged. However, once files are stored on a PVFS file system, it is no longer safe to modify this file.

iod.conf Files. The iod will look for an optional configuration file named iod.conf in the /etc directory when it is started. This file can specify a number of configuration parameters for the I/O daemon, including changing the data directory, the user and group under which the I/O daemon runs, and the port on which the I/O daemons operate.

Every line consists of two fields: a key field and a value field. These two fields are separated by one or more spaces or tabs. The key field specifies a configuration parameter whose value should be changed. The key is followed by this new value. Lines starting with a pound sign and empty lines are ignored. Keys are case insensitive. If the same key is used again, it will override the first instance. The valid keys are as follows:

port — specifies the port on which the iod should accept requests. Default is 7000.

user — specifies the user under which the iod should run. Default is nobody.

group — specifies the group under which the iod should run. Default is nobody.

rootdir — gives the directory the iod should use as its rootdir. The iod uses chroot(2) to change to this directory before accessing files. Default is /.

logdir — gives the directory in which the iod should write log files. Default is /tmp.

datadir — gives the directory the iod should use as its data directory. The iod uses chdir(2) to change to this directory after changing the root directory. Default is /pvfs_data.

debug — sets the debugging level; currently zero means don't log debug info and nonzero means do log debug info. This is useful mainly for helping find bugs in PVFS.

The `rootdir` keyword allows you to create a chroot jail for the `iod`. Here is a list of the steps the `iod` takes on startup:

1. read `iod.conf`
2. open log file in `logdir`
3. `chroot()` to `rootdir`
4. `chdir()` to `datadir`
5. `setuid()` and `setgid()`

The log file is always opened with the entire file system visible, while the datadir is changed into after the `chroot()` call. In almost all cases this option should be left as the default value.

Here is an example `iod.conf` file that could have been used for our example system:

```
# IOD Configuration file, iod.conf

port 7000
user nobody
group nobody
rootdir /
datadir /pvfs-data
logdir /tmp
debug 0
```

An alternative location for the `iod.conf` file may be specified by passing the filename as the first parameter on the command line to `iod`. Thus, running "`iod`" is equivalent to running "`iod /etc/iod.conf`".

`pvfstab` Files. When the client library is used, it will search for a `/etc/pvfstab` file in order to discover the local directories for PVFS files and the locations of the metadata server responsible for each of these file systems. The format of this file is the same as that of the `fstab` file:

```
head:/pvfs-meta  /pvfs  pvfs  port=3000  0  0
```

Here we have specified that the metadata server is called head, that the directory the server is storing metadata in is `/pvfs-meta`, that this PVFS file system should

be considered as "mounted" on the mount point `/pvfs` on the client (local) system, and that the TCP port on which the server is listening is 3000. The third field (the file system type) should be set to "pvfs" and the last two fields to 0. The fourth field is for options; the only valid option at this time is `port`.

It is occasionally convenient to be able to specify an alternative location for the information in this file. For example, if you want to use PVFS calls but cannot create a file in `/etc`, you might instead want to store the file in your home directory.

The `PVFSTAB_FILE` environment variable may be set before running a program to specify another location for the `pvfstab` file. In a parallel processing environment it may be necessary to define the variable in a `.cshrc`, `.bashrc`, or `.login` file to ensure that all tasks get the correct value.

Compile-Time Options. The majority of file system configuration values are defined in `pvfs_config.h` in the PVFS distribution. You can modify these values and recompile in order to obtain new default parameters such as ports, directories, and data distributions. Here are some of the more important ones:

`__ALWAYS_CONN__` — if defined, all connections to all I/O servers will be established immediately when a file is opened. This is poor use of resources but makes performance more consistent.

`PVFSTAB_PATH` — default path to `pvfstab` file.

`PVFS_SUPER_MAGIC` — magic number for PVFS file systems returned by `statfs()`.

`CLIENT_SOCKET_BUFFER_SIZE` — send and receive size used by clients.

`MGR_REQ_PORT` — manager request port. This should be an option to the manager, but it isn't at the moment.

`DEFAULT_SSIZE` — default strip size.

`__RANDOM_BASE__` — if defined, the manager will pick a random base number (starting I/O server) for each new file. This can help with disk utilization. There is also a manager command line parameter to enable this.

Additionally a `--with-log-dir` option to configure has recently been added to the PVFS package. This option specifies a new subdirectory in which to place log files. It has the side effect of turning off the use of unique strings on the end of log file names, making it easier to manage the log files.

17.3.5 Miscellanea

This section contains some notes on options to the `mgr` and on using `mount` with PVFS.

Currently the only important option to `mgr` is "-r", which enables random selection of base nodes for new PVFS files. The `mgr` by default logs a message any time a file is opened. Here is an example:

```
i 2580, b 0, p 4, s 65536, n 1, /pvfs-meta/foo
```

The fields printed are as follows:

i — `inode` number/handle

b — base node number

p — `pcount`

s — strip size

n — number of processes which have this file open

Finally the name of the metadata file is listed. This information is particularly helpful when debugging applications using parallel I/O.

We mentioned earlier that the `mount.pvfs` program is used to mount PVFS file systems. This is a little bit different from most file systems, in that usually the `mount` program can be used to mount any kind of file system. Some versions of the Linux `mount`, which is distributed as part of the util-linux package, will automatically look in `/sbin` for an external file system-specific mount program to use for a given file system type. At the time of writing all versions later than 2.10f seem to have this feature enabled. If this is enabled, then `mount` will automatically call `/sbin/mount.pvfs` when a PVFS file system mount is attempted. Using our example, we have

```
[root@head /root]# /sbin/mount -t pvfs head:/pvfs-meta /pvfs
```

If this works, then the administrator can also add entries into `/etc/fstab` for PVFS file systems. However, it is important to remember that the module must be loaded, the `pvfsd` daemon must be running, and the server daemons must be running on remote systems before a PVFS file system can be mounted.

17.4 Final Words

PVFS is an ever-developing system. As the system evolves, it's fairly likely that documentation updates will trail software development.

The PVFS development team is open to suggestions and contributions to the project. We are especially interested in scripts and tools that people develop to make managing PVFS easier. Users who have developed utilities to help manage their system are encouraged to contact us. We'll try to include such programs into the next PVFS release.

Lots of development is taking place in PVFS, particularly to help handle issues such as redundancy, more interesting data distributions, and the use of zero-copy network protocols (described in Chapter 6). For the newest information on PVFS, check the Web site: `www.parl.clemson.edu/pvfs`.