Using the Parallel Virtual File System

Robert B. Ross, Philip H. Carns, Walter B. Ligon III

January 2001

Abstract

This document describes in detail the use of the Parallel Virtual File System (PVFS) software. We first discuss obtaining and compiling the source packages. Next we describe installing and configuring the system. Following this we detail how to bring up the system and how to verify its operation.

Once these preliminaries are covered, we delve into the additional interfaces supported by PVFS, including the PVFS native interface and the Multi-Dimensional Block Interface (MDBI). We cover compiling programs to use these interfaces and alternative methods for specifying PVFS file system locations.

Next we cover the utilities provided with the PVFS package. These utilities allow one to check the status of the system components and measure PVFS performance. Finally we cover the details of the configuration files used by PVFS.

Contents

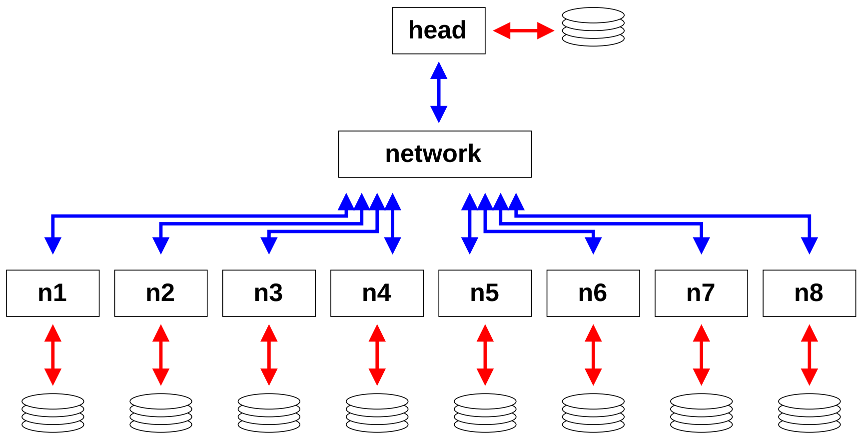

Figure 1: Example System

1 Introduction

The Parallel Virtual File System (PVFS) is a parallel file system. It allows applications, both serial and parallel, to store and retrieve file data which is distributed across a set of I/O servers. This is done through traditional file I/O semantics, which is to say that you can open, close, read, write, and seek in PVFS files just as in files stored in your home directory.

The primary goal of PVFS is to provide a high performance "global scratch space" for clusters of workstations running parallel applications. That said, there are all sorts of other ways that people have tried to or might want to use PVFS. We try to support other uses as long as they do not interfere with the system's ability to provide high performance parallel I/O.

PVFS development is performed on x86 Linux boxes; however, Alpha Linux is also supported. The PVFS servers are not particularly platform dependent, so it is possible that PVFS could be compiled on other platforms.

For the purposes of discussion we will pretend that we are installing PVFS on a small cluster of nine machines. In our example system, shown in Figure 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 text in order to clarify installation and configuration issues. This text is rife with example shell interactions in order to show exactly how things are configured, hopefully this is helpful and not irritating!

In Section 2 we will discuss high-level organizational issues in the system. In Sections 3 and 4 we cover the details of compiling and installing PVFS. Section 6 discusses how to make best use of PVFS in applications, both those using standard interfaces and ones using ROMIO or the native PVFS libraries. Section 7 covers some useful utilities supplied with PVFS. Finally a set of appendices provide additional details on topics such as configuration file formats, scripts, RPMs, and mounting PVFS file systems.

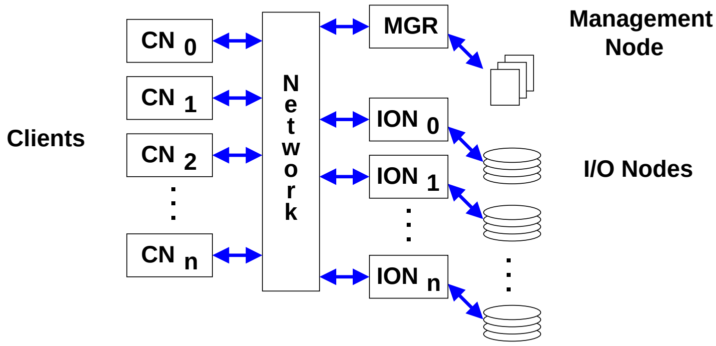

Figure 2: PVFS System Diagram

2 Theory of operation

When beginning to install PVFS it is important to understand the the roles that machines may play from the PVFS system perspective. There are three different roles that a given machine might play:

- metadata server
- I/O server
- client

Any machine can fill one or more of these roles.

The metadata server, of which there is one per PVFS file system, maintains information on files and directories stored in a PVFS file system. This includes permissions, owners, and the locations of data. Clients contact the metadata server when the want to create, remove, open, or close files. They also read directories from the metadata server.

I/O servers, of which there may be many, store PVFS file data. Each one does so by creating files on local file systems mounted on the machine, such as an existing ext2fs partition. Clients contact these servers in order to store and retrieve PVFS file data.

Clients are the users of the PVFS system. Applications accessing PVFS files and directories run on client machines. There are PVFS system components which perform operations on behalf of these clients, and these components will be discussed in a moment.

Figure 2 shows the PVFS system view, including a metadata server, a number of I/O servers with local disk, and a set of clients. For our example system we will set up the “head” node as the metadata server, the eight other nodes as I/O servers, and all nodes as clients as well. This would allow us to run parallel jobs accessing PVFS files from any node and striping these files across all the nodes except the head node.

3 Building the PVFS components

The PVFS package has come a long way in the last six months in terms of ease of compilation (if after doing this you think it was hard, thank yourself for not trying a year ago). The process is now pretty simple.

There are two tar files which you will need 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 absolutely necessary to use PVFS, but it makes accessing PVFS files much more convenient.

3.1 Obtaining the source

PVFS is open source and is freely available on the web. At the time of writing there were two consistent sources for 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 one should find 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 which have subsequently been compressed with the `gzip` tool. One should download the latest versions of each; the version numbers will not match. At the time of writing the newest version of the pvfs archive was 1.5.0 and the newest version of the pvfs-kernel archive was 0.9.0.

3.2 Untarring the packages

It is a bit easier to perform the compilations if you untar both the archives in the same directory, as 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 to easily find the include files it needs. Once this is finished the source is ready to be compiled.

3.3 Compiling the packages

Next we will compile the two packages. We discuss how to install the packages on the local system in this section as well; however, you probably don't want to do that. More likely you will want to wait and distribute the files to the correct machines after you have finished compiling. In the Installation section (Section 4 we will discuss what components need to be where, so don't worry about that yet.

3.3.1 Compiling PVFS

First we will compile the pvfs package. Continuing our example (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 you can install the components if you like:

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

The following are installed by default:

- `mgr`, `iod` in `/usr/local/sbin/`
- `libpvfs.a`, `libminipvfs.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 can be changed via options to `configure`. See `configure --help`.

3.3.2 Compiling PVFS-kernel

The PVFS-kernel package will perform tests for features based on header files and the running kernel, so it is important that you be running the kernel you want to use PVFS with and have the matching header files available on the machine on which you will compile. Assuming you have the matching headers, compiling is easy. Again we leave out 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 option here to configure lets the package know where it can find the PVFS I/O library, which is used by this package. It is statically linked to. If you run into any problems during this process, check the `README` and `INSTALL` files for hints.

Once this is done you can install the kernel components if you like:

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

The following are installed by default:

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

`mount.pvfs` is put in that location because that is the only location mount will look in for a file system specific executable.

You will have to install `pvfs.o` in the right place on your own. This is usually `/lib/modules/<kernel-version>/misc/`.

4 Installation

PVFS is a somewhat complicated package to get up and running. Partially this is because it is a multi-component system, but it is also true that 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 there are three potential roles that a machine might play:

- metadata server

- I/O server
- 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 will discuss the components and configuration files needed to fulfill each role.

As mentioned before, we will 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 8.1.

4.1 Directories used by PVFS

In addition to the roles which a machine may play, there are three types of directories which are utilized 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
- mount point

There is a single metadata directory for a PVFS file system. It exists on the machine which 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 how 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.

Finally there is a mount point on each client. This is an empty directory on which the PVFS file system is mounted. This empty directory is identical to any other mount point.

4.2 Installing and configuring the metadata server

There are three files necessary for a metadata server to operate:

- `mgr` executable
- `.iodtab` file

- `.pvfsdir` file

The `mgr` executable is the daemon which 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). As this list is ordered, it is important that once it is created it is not modified, as 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 of these files may be created with the `mkmgrconf` script. In our example we will use the directory `/pvfs-meta` as our metadata directory. We have boldfaced the user input to the `mkmgrconf` script for clarity.

```
[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.

4.3 Installing and configuring the I/O servers

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 which 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 non-superuser 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 "#" 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.

4.4 Installing and configuring clients

There are five files and one directory 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

The `pvfsd` executable is a daemon which 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 stand-alone manner to mount a PVFS file system directly. This will be covered in a later section.

The `pvfstab` file provides an fstab-like entry which describes to applications using the PVFS libraries how to access PVFS file systems. This is *not* needed by the kernel client code. It is only used if code is directly or indirectly linked to `libpvfs` (or `libminipvfs`). 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. `mknod` 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 you are using the `devfs` system, it is not necessary to create the /dev/pvfsd file, but it will not hurt to do so.

We are going to possibly 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 which will be a PVFS client. In our example we would need to copy out these files to each node and create the mount point.

4.5 Installing PVFS development components

There are a few components which 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 previous 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:

```
[root@head /usr/src/pvfs-1.5.0]# make install
if [ ! -d /usr/local/sbin ]; then install -d \
```

```
        /usr/local/sbin; fi
if [ ! -d /usr/local/bin ]; then install -d \
        /usr/local/bin; fi
if [ ! -d /usr/local/include ]; then install -d \
        /usr/local/include; fi
if [ ! -d /usr/local/bin ]; then install -d \
        /usr/local/bin; fi
if [ ! -d /usr/local/lib ]; then install -d \
        /usr/local/lib; fi
install -m 755 iod/iod /usr/local/sbin
install -m 755 lib/libminipvfs.a /usr/local/lib
install -m 755 lib/libminipvfs.a /usr/local/lib/libpvfs.a
install -m 755 mgr/mgr /usr/local/sbin
install -m 644 include/*.h /usr/local/include
set -e; for i in mkdot u2p mkmgrconf pvstat enableiod enablemgr pvfs-size
pvfs-mkfile pvfs-unlink pvfs-statfs iod-ping; do install -m 755 utils/$i \
        /usr/local/bin; done
set -e; for i in twrite tread; do install -m 755 examples/$i \
        /usr/local/bin; done
make -C docs install
make[1]: Entering directory '/usr/src/pvfs/docs'
if [ ! -d /usr/local/man/man3 ]; then install -m 755 -d
/usr/local/man/man3; fi
cp *.3 /usr/local/man/man3
ln -f /usr/local/man/man3/pvfs_chmod.3 /usr/local/man/man3/pvfs_fchmod.3
ln -f /usr/local/man/man3/pvfs_chown.3 /usr/local/man/man3/pvfs_fchown.3
ln -f /usr/local/man/man3/pvfs_dup.3 /usr/local/man/man3/pvfs_dup2.3
ln -f /usr/local/man/man3/pvfs_stat.3 /usr/local/man/man3/pvfs_fstat.3
ln -f /usr/local/man/man3/pvfs_stat.3 /usr/local/man/man3/pvfs_lstat.3
if [ ! -d /usr/local/man/man5 ]; then install -m 755 -d
/usr/local/man/man5; fi
cp *.5 /usr/local/man/man5
if [ ! -d /usr/local/man/man8 ]; then install -m 755 -d
/usr/local/man/man8; fi
cp *.8 /usr/local/man/man8
make[1]: Leaving directory '/usr/src/pvfs/docs'
```

4.6 PVFS and ROMIO

See `http://www.mcs.anl.gov/romio/` for information on ROMIO. We will be providing more information here in the future.

When compiling ROMIO you will want to use the "-file_system=pvfs" option to get PVFS support, or "-file_system=pvfs+nfs" for both PVFS and NFS support.

5 Start up and shut down

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.

5.1 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 insure 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.

5.2 Getting a client connected

Now we have the servers started, and we can start up the client-side components. First we load the module, then we start the client daemon `pvfsd`, 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, the “-h” option to `df` simply prints things in more human-readable form.

5.3 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 is good to go on head, now we must repeat this on the other nodes so that they can access the file system as well.

5.4 Unmounting file systems

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 ../
```

We 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.

5.5 Shutting down components

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

- 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`

6 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 64 Kbytes stripe size and all of the I/O nodes, starting with the first node listed in the `.iodtab` file. Note that use of UNIX system calls `read()` and `write()` result 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. On the other hand, 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.

The `setvbuf()` call may be used to specify the buffering characteristics of a stream (`FILE *`) after opening. This must be done before any other operations are performed:

```
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.

There is significant overhead in this transparent access 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 or the MDBI interface). 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 multi-dimensional block interface (MDBI). 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 as well. 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 multi-dimensional block interface can be used.

6.1 Preliminaries

When compiling programs to use PVFS, one 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/`, one should add "-lpvfs" to the link line and possibly "-L/usr/local/lib" to ensure the directory is included in the library search.

Finally, it is useful to know that the PVFS interface calls will also operate correctly on standard, non-PVFS, files. This includes the MDBI interface. This can be helpful when debugging code in that it can help isolate application problems from bugs in the PVFS system.

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

pcount – the number of I/O servers on which data will be stored (partitions, a bit of a misnomer)

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

In Figure 3 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 due to the striping parameters.

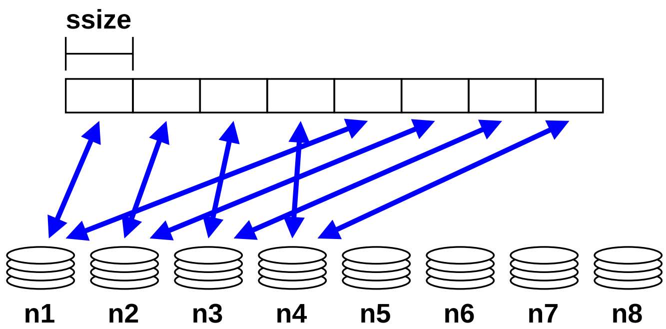

Figure 3: Striping Example

Physical distribution is determined when the file is first created. Using `pvfs_open()`, one can specify these 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, the physical distribution is defined by the user via the `pvfs_filestat` structure passed in by reference as the last parameter. This structure, defined in the PVFS header files, is defined 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.

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

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

It will fill in the structure with the physical distribution information for the file.

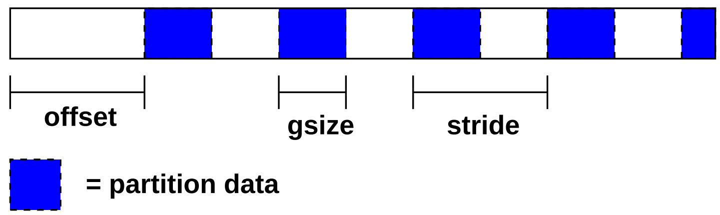

Figure 4: Partitioning Parameters

6.3 Setting a logical partition

The PVFS logical partitioning system allows an application programmer to describe the regions of interest in a file and subsequently access those regions in a very efficient manner. 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 both due to the number of separate operations and the reduced data movement per operation.

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.

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 4 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

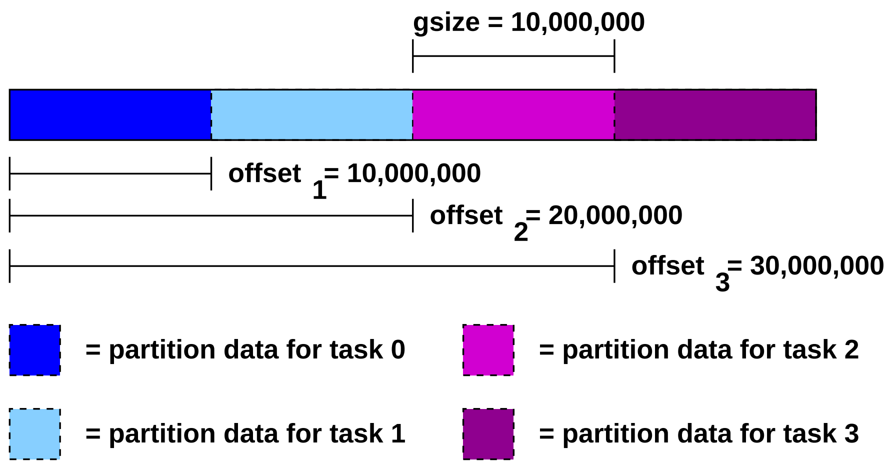

Figure 5: Partitioning Example 1

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 which allows the process to see then entire file.

As an example, suppose a file contains 40,000 records of 1000 bytes each, there are 4 parallel tasks, and each task needs to access a partition of 10,000 records each for processing. In this case one would set the group size to 10,000 records times 1000 bytes or 10,000,000 bytes. Then each task (0..3) would set its offset so that it would access a disjoint portion of the data. This is shown in Figure 5.

Alternatively, suppose one wants to allocate the records in a cyclic or ”round-robin” manner. In this case the group size would be set to 1000 bytes, the stride would be set to 4000 bytes and the offsets would again be set to access disjoint regions, as shown in Figure 6.

It is important to realize that setting the partition for one task has no effect whatsoever on any other tasks. There is also no reason that the partitions set by each task be distinct; one can overlap the partitions of different tasks if desired. Finally, there is no direct relationship between partitioning and striping for a given file; while it is often desirable to match the partition to the striping of the file, users have the option of selecting any partitioning scheme they desire 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 multi-dimensional data is often more easily partitioned using the multi-dimensional block interface.

6.4 Using multi-dimensional blocking

The PVFS multi-dimensional block interface (MDBI) provides a slightly higher-level view of file data than 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

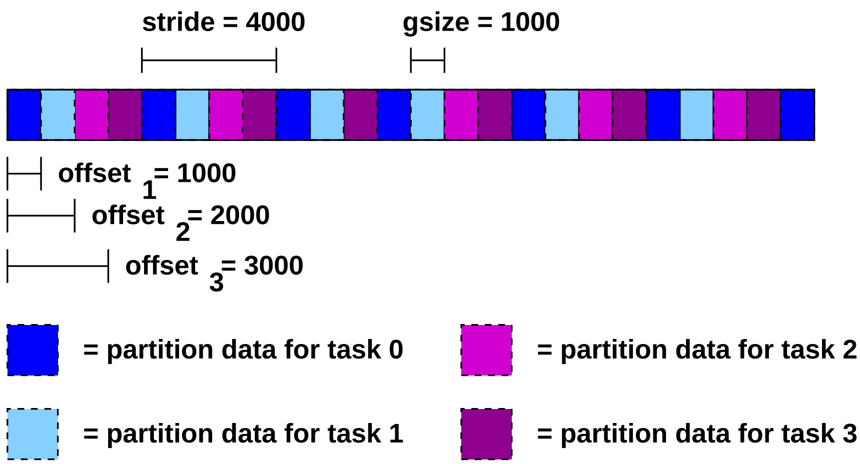

Figure 6: Partitioning Example 2

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 the programmer has 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.

There are five basic calls 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()`. `set_blk()` is the call used to set the blocking parameters for the array before reading or writing. It can be

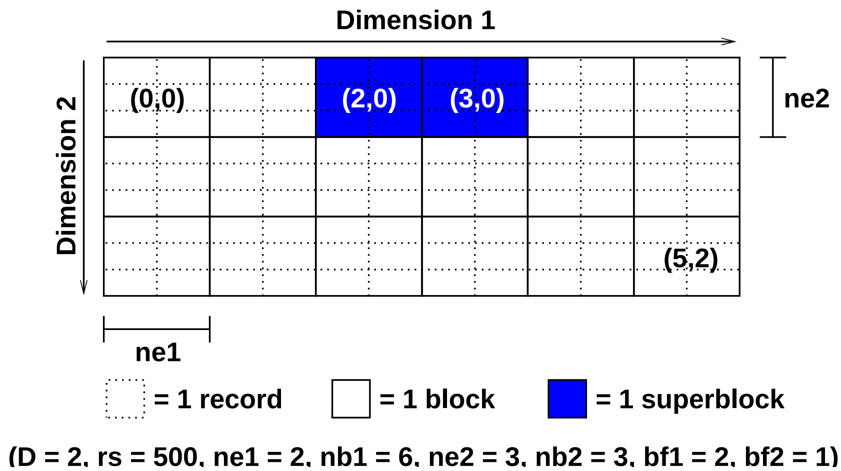

Figure 7: MDBI Example 1

used as often as necessary and does not entail communication. `read_blk()` and `write_blk()` are used to read blocks of records once the blocking has been set.

In Figure 7 we can see 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:

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

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

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

Similarly, to read block (5, 2):

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

A final feature of the MDBI is block buffering. Sometimes multi-dimensional 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 an under utilization 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 ahead of time, PVFS relies on user cues to determine what to buffer. This is done by defining "blocking factors" which group blocks together. A single function is used to define the blocking factor:

int buf_blk(int fd, int bf_1, ..., int bf_n);

The *blocking factor* indicates how many blocks in the given dimension should be buffered.

Looking at Figure 7 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.

It is important to understand that no cache coherency is performed here; if application tasks are sharing superblocks, unexpected results will occur. It is up to the user to ensure that this does not happen. A good strategy for buffering is to develop your program without buffering turned on, and then enable it later in order to improve performance.

6.5 Using ROMIO MPI-IO

ROMIO is an implementation of the MPI-IO interface which supports PVFS and includes a pair of optimizations which can be of great benefit to applications performing collective I/O.

See `http://www.mcs.anl.gov/romio/` for additional information.

7 PVFS utilities

There are a few utilities provided which are useful in dealing with PVFS files and file systems.

7.1 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). 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. `u2p` relies on the existence of the `/etc/pvfstab` file to operate correctly.

7.2 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.

7.3 Checking on server status

The `iod-ping` utility can be used to determine if a given I/O server is up and 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). Additionally, specifying no additional parameters will cause the program to automatically check for a server on localhost at the default port for the server type (7000 for I/O server, 3000 for metadata server). Not specifying a "-p" option will use the default port.

There is no `fsck` for PVFS at the moment, although arguably there should be.

8 Appendices

This section contains additional information on configuration file formats, some notes on `mount` and PVFS, and a description of using RPMs with PVFS.

8.1 Configuration details

The current PVFS configuration system is a bit of a mess. Here we try to shed more light on:

- configuration file formats
- file system defaults
- other options

8.1.1 `.pvfsdir` and `.iodtab` files

The creation of these files we discussed in Section 4. In this section we cover the details of the file formats. `.pvfsdir` files hold 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 using the `mkmgrconf` script, whose use is also described in Sections 4 and 7.

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 `.pvfsdir` file in the metadata directory
- 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 subdirectories 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. It is stored in the metadata directory of the PVFS file system.

An example of a `.iodtab` file would be:

```
# 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, assuming the default port (7000) and using hostnames (as in our example):

```
# 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.

8.1.2 `iod.conf` files

The iod will look for an optional configuration file named `/etc/iod.conf` 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 an value field. These two fields are separated by one or more spaces or tabs. The key field specifies a configuration parameter which should be set to the value.

Lines starting with a hash mark ("#") 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:

`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 non-zero means do log debug info. This is really only useful for helping find bugs in PVFS.

The `rootdir` keyword allows one to create a chroot jail for the `iod`. It's a little bit confusing to use. 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()`

So 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 in order to preserve ones sanity.

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 the same as running "`iod /etc/iod.conf`".

8.1.3 `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 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 as a user 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 used to specify an alternate location. Simply set the environment variable before running the program. 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.

8.1.4 Changing compile-time file system defaults

The majority of file system configuration values are defined in `pvfs_config.h` in the PVFS distribution. One 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 as 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.

8.1.5 Options to `mgr`

Currently the only important option to `mgr` is "-r", which enables random selection of base nodes for new PVFS files.

8.2 Using `mount` with PVFS

We mentioned earlier that the `mount.pvfs` program is used to mount PVFS file systems. This is a little bit different than for 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:

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

If this works on your system, then you can 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.

8.3 Startup scripts

This section will eventually describe startup scripts for PVFS. We should have a contrib section on the web site somewhere for this sort of thing.

8.4 Log files

This section will eventually describe the log files in more detail.

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:

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.

8.5 Using RPMs

It's helpful for large systems with identical installations to use some sort of packaging mechanism for software installs. One such mechanism is the RPM system from RedHat.

It's really not reasonable to attempt to provide binary RPMs for PVFS any more, mainly due to the use of kernel modules, which are very kernel specific. However, it is possible for us to provide an SRPM which can be built by administrators specifically for their machine.

9 Final words

PVFS is an ever-developing system, and as such things change. This revision of this document reflects what was, at the time of writing, the current state of the system. Undoubtedly, however, things will continue to evolve, hopefully for the better. As that happens, it's fairly likely that updating of this documentation will trail software development.

We the PVFS development team are open to suggestions and contributions to the project. We're especially interested in scripts and tools that people develop to make managing PVFS easier. If you have developed utilities to help manage your system that you would like to share, please contact us! We'll try to get your programs into the next release.

For more information on PVFS, check the web site: `http://www.parl.clemson.edu/pvfs`.