
Parallel file systems differ from distributed file systems primarily in the effort put into "going fast". Distributed file systems primarily go fast by making very effective use of caches in the servers and in the clients, in order to eliminate traffic traveling further to/from the client memory to the server memory and eventually to the server disk. Beyond that, distributed file systems primarily rely on each client, each network link and each server device to go faster in order improve speed.

Since making any single device go faster is a difficult approach (expensive, technologically limited, small market for resulting device), especially mechanical devices like magnetic disk drives, parallel file systems are constructed as a large collection of devices virtualized as a single system and designed to work together on the work of one application. In this sense, any disk array or striping volume manager device driver can be said to be a small-scale parallel system, although in general we reserve the "parallel" label for systems that try to scale to much larger collections. The second driving characteristic of parallel file systems is that they serve very large numbers of client computers. For example, the largest compute cluster in Intel's back end chip development process has well in excess of 5,000 computers all mounting and accessing the same filesystem.

But perhaps the most interesting technical aspect of parallel file systems is that those large numbers of nodes are often running as few as one parallel application written to use all the processors in the entire cluster for the same parallel task. This small number of highly parallel applications tends to break the design of "scalable" distributed filesystems because designers of the latter assumed that their scaling number of client computers all operate independently, each running unrelated tasks, and that these tasks were not designed to concurrently access the same files. This assumption means that distributed filesystems had to support lots of activity, but each activity did not need much speed and each file would not be written by one client while another was reading or writing it.

As an example of impact of independent client computers, the NFS client server protocol has no provision for ensuring that its client computers' file caches are consistent with the servers' data at all times. NFS assumes that the principle form of "write sharing", when file written by one client and read or written by another client machine, will be sequential write sharing: where one client opens, writes and closes a file without concurrent access from any other client (concurrent access when all clients are readers is a common case, but an easy case since no synchronization is needed, and both distributed and parallel file systems do it well), but then at some later time another client may open and access the file. Unfortunately, if two client computers open the same file for reading and writing at the same time, NFS's cache consistency is insufficient and most implementations cannot guarantee much about the resulting file's contents.

Parallel filesystems generally need to support concurrent write sharing by many compute nodes on the same file at the same time. Perhaps the most common concurrent write sharing access pattern is the parallel application checkpoint. A checkpoint is the image of all the state of a (parallel) application stored on disk. Parallel applications do this because they run on such large computers that the time between application crashes is quite small, 8-48 hours in the biggest computers,

and because these applications quite often run for days and sometimes weeks in order to complete their task (simulating the global implications of a 6 kilometer wide asteroid hitting the earth, for example). So checkpoints are taken periodically so that after each application failure, the entire set of all clients in the parallel application abort then reload from the last checkpoint and start again.

While it is certainly possible for a parallel checkpoint to be composed of one file per processor in the parallel computer, this is undesirable in many ways. First, the number of processors in the restarted execution can be different, forcing a transformation of the data per processor. Second, checkpoints are frequently used to visualize the progress of the parallel application, and the number of nodes running the visualization is always quite different from the primary computer. Moreover, with 100s of thousands of processors (soon to be millions) in a large parallel computer, each checkpoint would be 100s of thousand files to create and name. Creating this many files in many filesystems is not an optimized operation, especially in the same directory, because traditional filesystem design assumes that the use of a directory is for a user to look through the list of files, something that will not happen with millions of names. But perhaps the most compelling reason is data management; a single file is a single thing to keep track of, and it is easier to keep it self- consistent than millions of smaller files.

So when a checkpoint is put into a single file, all clients are writing into the same file at the same time. In the best case each processor gets one big chunk of the file and writes sequentially to it. But, this is not what many parallel applications do. Suppose the parallel application is simulating a dynamically changing mesh of the simulated thing, with lots more points in the mesh where lots is happening (say at the corner of a pipe in a hot engine) and many fewer points where little is happening (say the surface of the ocean that has not yet encountered the wavefront of the asteroid's splash down). The parallel application has to spread these mesh points across nodes in order to balance the work done at each processor and minimize the amount of interprocessor communication because mesh points near the edge of a region assigned to one processor are effected by changes in the mesh points near the edge of a region assigned to a different processor. Then, the variables at that mesh point (such as temperature, pressure, wind pressure, ocean current, radiant heat, etc) may not have an easily determined place to be stored in the checkpoint file. These applications "walk the mesh" computing the appropriate place in the checkpoint file for that mesh point's variables, issuing a "seek" to that point and writing just a few variables.

The storage specialists should be incited to riot by this – 100s of thousand of clients opening one file then issuing a large number of seeks followed by small writes. Can you imagine an uglier access pattern?

[Aside, there is one saving grace in this access pattern: usually each byte in the checkpoint is written exactly once by one client, and no other client reads or writes that byte while the checkpoint is being built. Too bad there is no way to express this property in an POSIX standardized open mode flag.]

Fast concurrent writing based on encapsulated per-client: It has recently been suggested that something like logging be used for this important and difficult access pattern. Suppose the file system represented the writing each client is doing as a log of write commands stored in a sequentially written log file. And, if the parallel filesystem does this logging, it can be responsible for hiding the log representation under the abstraction of a flat file.

Of course, such an encoding requires either long read accesses, reconstruction of the file on the first read, or an approach in between. However, this may be acceptable for some workloads. For example, parallel applications may periodically write out a checkpoint of the state of the compute nodes to the filesystem in order recover in the face of a crash. It is important that such files be written as quickly as possible so as not to waste expensive compute time. However, most checkpoints are never read ("write once, read maybe"), and so a slower read speed is not as important.

The Project: In this project, you will each design a method for changing a parallel filesystem used in some of the largest parallel computers to support this type of concurrent write access pattern with hidden per-server log files.

This is not mission impossible, but it is a cutting-edge parallel filesystem research experiment. The only thing that differentiates your work from publishable research is how well your solutions solve all the problems a parallel filesystem designer would uncover while doing this experiment. This means, of course, that if you do really well at this, maybe there is a research publication in it for you.

You will do this project in groups of three or four students. The project involves coding in the Parallel Virtual Filesystem, an open source parallel filesystem developed at Clemson University and Argonne National Laboratory and used in many production, large systems at various national labs. PVFS is a large and complicated code base; one of the goals of this project is to give you the experience of working on such real code.

Step 0: Form Groups, Download, Compile, Install PVFS: Form groups of three or four students. Email the course staff with your group names and a group number will be assigned to you.

Either download the project 3 distribution from the course homepage or use the links to source code and documentation (A Quick Start Guide to PVFS2) at the end of this document to compile PVFS2 and install it in your home directory either on your own machines or the Linux systems at `unix.andrew.cmu.edu`. Use the client programs like `pvfs2-ls` and `pvfs2-cp` (compiled from `pvfs-2.7.0/src/apps/admin`) to test the installation. It is recommended that you begin this step early early, and ask for help if you have trouble. Understanding the PVFS code base will take much of the time of this project.

Step I: Extended Attributes and mpi io test. Due April 9th: As a warm up to get experience working with the code base, modify the PVFS2 server to set an extended attribute on every file created in the system. Every file should have a `user.groupname` key with a value string equal to the names of your group members. To do this, it is suggested that you look at the behavior of `pvfs-2.7.0/src/server/create.sm` and `set-eattr.sm`. Use the `group-name-test.sh` script included in the project 3 distribution to test your modifications.

Also for this part of the project, compile and install the `mpich2` library and `mpi_io_test` benchmark included in the project 3 distribution and linked at the end of this document (the compilation of `mpi_io_test` can be tricky. Check the README file in the project 3 distribution). Read the `mpi_io_test` documentation to see its options and usage examples. Then use `mpi_io_test` to measure the performance of 10 client processes (for this part of the project, they do not have to

reside on separate nodes) writing 4096 objects of size 1024 bytes to a single file on a single server.

This step is due on April 9th. Turn in a tar ball containing the following:

(a) Your modified PVFS2 source directory (after running `make clean`)

(b) A `GROUP` file containing your group number and names

(c) A `README-step-1` file containing a description of the changes you made to PVFS, the output of `group-name-test.sh` and the results of your `mpi_io_test` experiment

Turn in instructions are included at the end of this document.

Step II: Write Path. Due April 23rd: In this step, you must design and implement a log-style encoding for concurrent and random access writes. If the following series of I/O calls are issued on the buffer in figure 1, under a normal encoding the contents of the file will look just like the buffer (although its on disk representation might obviously be fragmented). Under a log-like encoding, however, the result should look something like the bottom half of the figure.

```
pwrite(file, buffer[100], 50, 100);
pwrite(file, buffer, 50, 0);
pwrite(file, buffer[50], 50, 50);
```

Once you have designed a log-like encoding for your files, implement it in PVFS2. For this project, it is acceptable for all files in the filesystem to be written in log-style. Note that PVFS stripes all data for a given file across multiple servers. See the `distribution.pdf` file in your PVFS2 `doc` directory for details. For the sake of simplicity, we will modify neither client code nor this distribution strategy. To clients, it appears that they are writing to physical blocks through the normal distribution mechanism. Since everything is done on the stripe level, the actual run length encoding will be per-stripe rather than per-file. For sufficiently large stripes compared to the write size, however, this will be sufficient.

Before starting on this part of the project, read the PVFS storage infrastructure documentation mentioned at the end of this document. It'll help to understand how PVFS uses things like *flows, jobs,* and `trove` to avoid being confused by the code structure. When you're ready to look at code, on the client side, you should take a look at the `sys-small-io.sm` and `sys-io.sm` files in the `pvfs-2.7.0/src/client/sysint` directory to see how a client issues I/O requests to the server. Most of your work on the server for this part of the project should take place in files in the `pvfs-2.7.0/src/io/trove/trove-dbpf` directory. Pay particular attention to the `dbpf-bstream.c` file, where you will find the final storage operations initiated in response to the client side operations.

To debug your implementation without a read path implemented yet, use a filesystem exported by a single PVFS2 server and copy a file onto it using `pvfs2-cp`. You can find the name of the "bstream" file (the local UNIX file used by the server to store one part of a striped file) using the

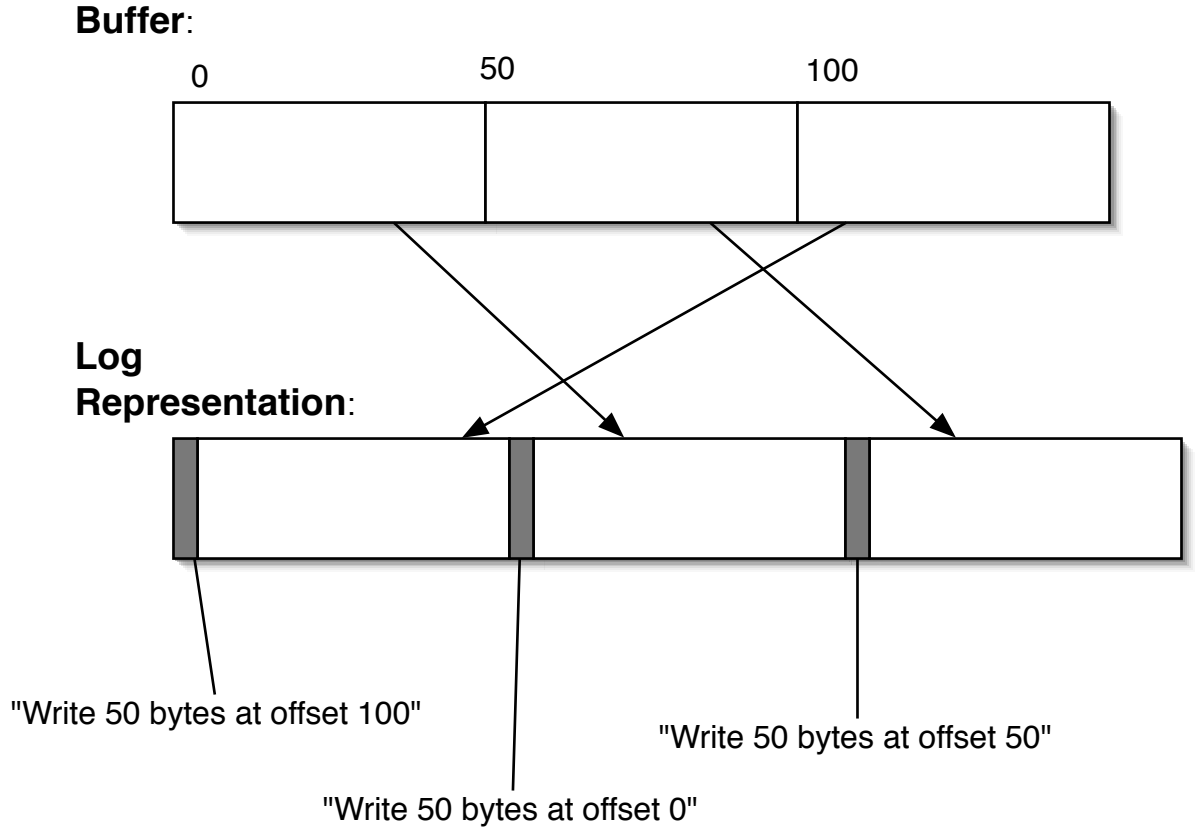

Figure 1: A minimum complexity write encoding

`pvfs2-viewdist` utility. If you find it under the pvfs2 storage space for the server, you can examine the contents using a utility like `hexdump`.

This step of the project is due on April 23rd. Turn in a tar ball containing the following:

(a) Your modified PVFS2 source directory (after running `make clean`)

(b) A `GROUP` file containing your group number and names

(c) A `README-step-2.pdf` file containing:

- A description of your write path strategy for log encoding including the format. Provide a rationale for your design. Also document what modifications you made to the PVFS2 code to achieve it.
- A comparison of the write performance of `mpi_io_test`'s N-to-1 write pattern on your implementation of log-encoding versus unmodified PVFS (use the `-op write` flag to `mpi_io_test`). Vary the write size and the number of clients and make plots of your results. The comparison should be at least partially graphical.

Turn in instructions are included at the end of this document.

Step III: Read Path. Due May 2nd: For the next part of this project, complete the functionality to read the log encoding you implemented in the previous step. Consider implementing this in several phases. For full credit, your implementation only has to be correct. Implementing a read solution that potentially has to scan through the entire file to service a read request is acceptable.

However, a better solution would be one that reconstructed the entire bstream into a normal representation on the first read command. This solution will require being able to tag bstreams to indicate those that are log-encoded, a task for which you may find some of the keyval trove functions useful. Even more sophisticated solutions exist, such as building an indexing structure either while writing or reading the file for faster lookups.

We will be evaluating your read path by issuing three, short random reads to a large, log-encoded file. The group that performs the best at this will have the opportunity to describe their solution to visitors from Los Alamos National Labs during their visit in mid May.

This final step of the project is due on May 2nd. Turn in a tar ball containing the following:

(a) Your modified PVFS2 source directory (after running `make clean`)

(b) A `GROUP` file containing your group number and names

(c) A `README-step-3.pdf` file containing:

- A description of your read path strategy for log encoding including the format. Provide a rationale for your design. Document the modifications you made to PVFS2. Be sure to explain any approach you used to achieve faster performance than full file scanning on every read.
- A comparison of the read performance on the result of `mpi_io_test`'s N-to-1 write pattern on your implementation of log-encoding versus unmodified PVFS. Vary the write size and the number of clients and make plots of your results. The comparison should be at least partially graphical.

Turn in instructions are included at the end of this document.

Deliverables: You are required to make several turn-ins for this project including source code in a tarball named /proj3-step-X.tar.gz. The layout of the tarballshould look like the following:

```
/proj3-step-X/
/proj3-step-X/GROUP
/proj3-step-X/README-step-X
/proj3-step-X/pvfs-2.7.0/
/proj3-step-X/pvfs-2.7.0/...
...
```

Be sure to run `make clean` in the `/proj3-step-X/pvfs-2.7.0/` directory before compressing.

Your source tree should compile in a manner analogous to that of the standard PVFS2 distribution. As in previous assignments, a TA should be able to do something like the following on a `unix.andrew.cmu.edu` machine:

```
unix49{}% mkdir foo ; cd foo
unix49{foo}% tar xfz ../proj3-step-X.tar.gz
unix49{foo}% cd proj3-step-X/pvfs-2.7.0
unix49{foo/pvfs-2.7.0}% ./configure <arguments>
unix49{foo/pvfs-2.7.0}% make
```

and have a compiled `pvfs2-server` binary in the `pvfs-2.7.0/src/server` directory. If you also submit any additional test programs, include a README.txt file documenting what they do, and how to compile and run them.

We have set up a drop box in the course AFS space. Please put your final tar file in the directory: You will not be able to list the directory, only put your file there. Email the TA once you have dropped off the file.

`/afs/ece.cmu.edu/usr/ganger/.vol5/public_html/ece746.spring08/proj3drop`

Resources: The project 3 software distribution contains the following pieces of software necessary for this project. You may also use the links below to download them directly:

- PVFS version 2.7.0: The latest release of PVFS. Contains source code and documentation files.

  `http://mirror.anl.gov/pub/pvfs2/pvfs-2.7.0.tar.gz`

- Berkeley DB: A non-relational database used by PVFS to store its metadata. Installation of Berkeley DB is required to compile PVFS.

  `http://www.oracle.com/technology/software/products/berkeley-db/db/index.html`

- MPICH2: An implementation of the Message Passing Interface standard used for distributed applications. PVFS can be compiled to use the MPI IO interface to support parallel data transfer from multiple nodes.

  `http://www.mcs.anl.gov/research/projects/mpich2/`

- MPI IO TEST: A parallel I/O benchmarking program from Los Alamos National Laboratory. This will be used to measure performance in the last step of this project.

  `http://public.lanl.gov/jnunez/benchmarks/mpiiotest.htm`

Documentation: Documentation for PVFS is included in the `pvfs2-docs` directory in the project 3 distribution on the course website. It can also be generated in the `doc` directory by running `make docs` in the root directory of the decompressed `pvfs-2.7.0.tar.gz`. In particular, the `concepts.pdf, trove-dbpf.pdf, storage-interface.pdf, pvfs2-trove-usage.pdf` and other files in the `/doc/design` directory may be helpful for this project. The total page count for the listed documentation above is around 20 pages. Much of the documentation for PVFS is still incomplete, so consider these files an adjunct to reading the code and asking questions, rather than a replacement.

The following online resources might also be useful:

- A Quick Start Guide to PVFS2: Details on getting PVFS2 installed and running. About 16 pages.

  `http://www.pvfs.org/cvs/pvfs-2-7-branch.build/doc//pvfs2-quickstart/pvfs2-quickstart.php`

- The PCFS2 Developer's Guide: A high level description of the source tree and components of the project. About 25 pages.

  `http://www.pvfs.org/cvs/pvfs-2-7-branch.build/doc//pvfs2-guide/pvfs2-guide.php`