PVFS: A Parallel File System for Linux Clusters*

Philip H. Carns Walter B. Ligon III
Parallel Architecture Research Laboratory
Clemson University, Clemson, SC 29634, USA
{pcarns, walt} @parl.clemson.edu

Robert B. Ross Rajeev Thakur
Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL 60439, USA
{rross, thakur} @mcs.anl.gov

Abstract

As Linux clusters have matured as platforms for low-cost, high-performance parallel computing, software packages to provide many key services have emerged, especially in areas such as message passing and networking. One area devoid of support, however, has been parallel file systems, which are critical for high-performance I/O on such clusters. We have developed a parallel file system for Linux clusters, called the Parallel Virtual File System (PVFS). PVFS is intended both as a high-performance parallel file system that anyone can download and use and as a tool for pursuing further research in parallel I/O and parallel file systems for Linux clusters.

In this paper, we describe the design and implementation of PVFS and present performance results on the Chiba City cluster at Argonne. We provide performance results for a workload of concurrent reads and writes for various numbers of compute nodes, I/O nodes, and I/O request sizes. We also present performance results for MPI-IO on PVFS, both for a concurrent read/write workload and for the BTIO benchmark. We compare the I/O performance when using a Myrinet network versus a fast-ethernet network for I/O-related communication in PVFS. We obtained read and write bandwidths as high as 700 Mbytes/sec with Myrinet and 225 Mbytes/sec with fast ethernet.

*This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, and in part by the National Aeronautics and Space Administration, under Research Grant NAG-5-3835.

1 Introduction

Cluster computing has recently emerged as a mainstream method for parallel computing in many application domains, with Linux leading the pack as the most popular operating system for clusters. As researchers continue to push the limits of the capabilities of clusters, new hardware and software have been developed to meet cluster computing's needs. In particular, hardware and software for message passing have matured a great deal since the early days of Linux cluster computing; indeed, in many cases, cluster networks rival the networks of commercial parallel machines. These advances have broadened the range of problems that can be effectively solved on clusters.

One area in which commercial parallel machines have always maintained great advantage, however, is that of parallel file systems. A production-quality high-performance parallel file system has not been available for Linux clusters, and without such a file system, Linux clusters cannot be used for large I/O-intensive parallel applications. We have developed a parallel file system for Linux clusters, called the Parallel Virtual File System (PVFS) [33], that can potentially fill this void. PVFS is being used at a number of sites, such as Argonne National Laboratory, NASA Goddard Space Flight Center, and Oak Ridge National Laboratory. Other researchers are also using PVFS in their studies [28].

We had two main objectives in developing PVFS. First, we needed a basic software platform for pursuing further research in parallel I/O and parallel file systems in the context of Linux clusters. For this purpose, we needed a stable, full-featured parallel file system to begin with. Our second objective was to meet the need for a parallel file system for Linux clusters. Toward that end, we designed PVFS with the following goals in mind:

- It must provide high bandwidth for concurrent read/write operations from multiple processes or threads to a common file.
- It must support multiple APIs: a native PVFS API, the UNIX/POSIX I/O API [15], as well as other APIs such as MPI-IO [13, 18].
- Common UNIX shell commands, such as `ls`, `cp`, and `rm`, must work with PVFS files.
- Applications developed with the UNIX I/O API must be able to access PVFS files without recompiling.
- It must be robust and scalable.
- It must be easy for others to install and use.

In addition, we were (and are) firmly committed to distributing the software as open source.

In this paper we describe how we designed and implemented PVFS to meet the above goals. We also present performance results with PVFS on the Chiba City cluster [7] at Argonne National Laboratory. We first present the performance for a workload comprising concurrent reads and writes using native PVFS calls. We then present results for the same workload, but by using MPI-IO [13, 18] functions instead of native PVFS functions. We also consider a more difficult access pattern, namely, the BTIO benchmark [21]. We compare the performance when using a Myrinet network versus a fast-ethernet network for all I/O-related communication.

The rest of this paper is organized as follows. In the next section we discuss related work in the area of parallel file systems. In Section 3 we describe the design and implementation of PVFS. Performance results are presented and discussed in Section 4. In Section 5 we outline our plans for future work.

2 Related Work

Related work in parallel and distributed file systems can be divided roughly into three groups: commercial parallel file systems, distributed file systems, and research parallel file systems.

The first group comprises commercial parallel file systems such as PFS for the Intel Paragon [11], PIOFS and GPFS for the IBM SP [10], HFS for the HP Exemplar [2], and XFS for the SGI Origin2000 [35]. These file systems provide high performance and functionality desired for I/O-intensive applications but are available only on the specific platforms on which the vendor has implemented them. (SGI, however, has recently released XFS for Linux. SGI is also developing a version of XFS for clusters, called CXFS, but, to our knowledge, CXFS is not yet available for Linux clusters.)

The second group comprises distributed file systems such as NFS [27], AFS/Coda [3, 8], InterMezzo [4, 16], xFS [1], and GFS [23]. These file systems are designed to provide distributed access to files from multiple client machines, and their consistency semantics and caching behavior are designed accordingly for such access. The types of workloads resulting from large parallel scientific applications usually do not mesh well with file systems designed for distributed access; particularly, distributed file systems are not designed for high-bandwidth concurrent writes that parallel applications typically require.

A number of research projects exist in the areas of parallel I/O and parallel file systems, such as PIOUS [19], PPFS [14, 26], and Galley [22]. PIOUS focuses on viewing I/O from the viewpoint of transactions [19], PPFS research focuses on adaptive caching and prefetching [14, 26], and Galley looks at disk-access optimization and alternative file organizations [22]. These file systems may be freely available but are mostly research prototypes, not intended for everyday use by others.

3 PVFS Design and Implementation

As a parallel file system, the primary goal of PVFS is to provide high-speed access to file data for parallel applications. In addition, PVFS provides a clusterwide consistent name space, enables user-controlled striping of data across disks on different I/O nodes, and allows existing binaries to operate on PVFS files without the need for recompiling.

Like many other file systems, PVFS is designed as a client-server system with multiple servers, called I/O daemons. I/O daemons typically run on separate nodes in the cluster, called I/O nodes, which have disks attached to them. Each PVFS file is striped across the disks on the I/O nodes. Application processes interact with PVFS via a client library. PVFS also has a manager daemon that handles only metadata operations such as permission checking for file creation, open, close, and remove operations. The manager does not participate in read/write operations; the client library and the I/O daemons handle all file I/O without the intervention of the manager. The clients, I/O daemons, and the manager need not be run on different machines. Running them on different machines may result in higher performance, however.

PVFS is primarily a user-level implementation; no kernel modifications or modules are necessary to install or operate the file system. We have, however, created a Linux kernel module to make simple file manipulation more convenient. This issue is touched upon in Section 3.5. PVFS currently uses TCP for all internal communication. As a result it is not dependent on any particular message-passing library.

3.1 PVFS Manager and Metadata

A single manager daemon is responsible for the storage of and access to all the metadata in the PVFS file system. Metadata, in the context of a file system, refers to information describing the characteristics of a file, such as permissions, the owner and group, and, more important, the physical distribution of the file data. In the case of a parallel file system, the distribution information must include both file locations on disk and disk locations in the cluster. Unlike a traditional file system, where metadata and file data are all stored on the raw blocks of a single device, parallel file systems must distribute this data among many physical devices. In PVFS, for simplicity, we chose to store both file data and metadata in files on existing local file systems rather than directly on raw devices.

PVFS files are striped across a set of I/O nodes in order to facilitate parallel access. The specifics of a given file distribution are described with three metadata parameters: base I/O node number, number of I/O nodes, and stripe size. These parameters, together with an ordering of the I/O nodes for the file system, allow the file distribution to be completely specified.

An example of some of the metadata fields for a file `/pvfs/foo` is given in Table 1. The *pcount* field specifies that the data is spread across three I/O nodes, *base* specifies that the first (or base) I/O node is node 2, and *ssize* specifies that the stripe size—the unit by which the file is divided among the I/O nodes—is 64 Kbytes. The user can set these parameters when the file is created, or PVFS will use a default set of values.

Table 1: Metadata example: File `/pvfs/foo`.

inode	1092157504
⋮	⋮
base	2
pcount	3
ssize	65536

Application processes communicate directly with the PVFS manager (via TCP) when performing operations such as opening, creating, closing, and removing files. When an application opens a file, the manager returns to the application the locations of the I/O nodes on which file data is stored. This information allows applications to communicate directly with I/O nodes when file data is accessed. In other words, the manager is not contacted during read/write operations.

One issue that we have wrestled with throughout the development of PVFS is how to present a directory hierarchy of PVFS files to application processes. At first we did not implement directory-access functions and instead simply used NFS [27] to export the metadata directory to nodes on which applications would run. This provided a global name space across all nodes, and applications could change directories and access files within this name space. The method had some drawbacks, however. First, it forced system administrators to mount the NFS file system across all nodes in the cluster, which was a problem in large clusters because of limitations with NFS scaling. Second, the default caching of NFS caused problems with certain metadata operations.

These drawbacks forced us to reexamine our implementation strategy and eliminate the dependence on NFS for metadata storage. We have done so in the latest version of PVFS, and, as a result, NFS is no longer a requirement. We removed the dependence on NFS by trapping system calls related to directory access. A mapping routine determines whether a PVFS directory is being accessed, and, if so, the operations are redirected to the PVFS manager. This trapping mechanism, which is used extensively in the PVFS client library, is described in Section 3.4.

3.2 I/O Daemons and Data Storage

At the time the file system is installed, the user specifies which nodes in the cluster will serve as I/O nodes. The I/O nodes need not be distinct from the compute nodes.

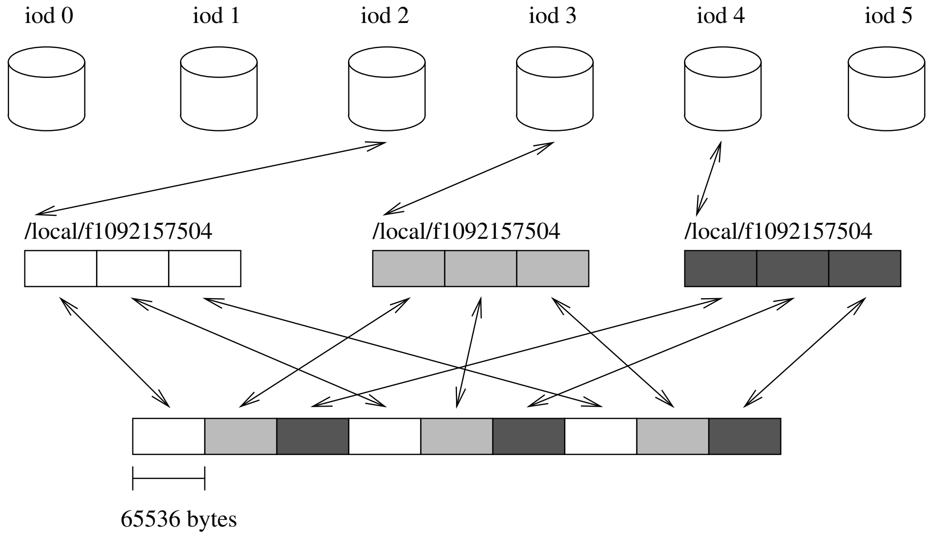


Figure 1: File-striping example

An ordered set of PVFS I/O daemons runs on the I/O nodes. The I/O daemons are responsible for using the local disk on the I/O node for storing file data for PVFS files.

Figure 1 shows how the example file `/pvfs/foo` is distributed in PVFS based on the metadata in Table 1. Note that although there are six I/O nodes in this example, the file is striped across only three I/O nodes, starting from node 2, because the metadata file specifies such a striping. Each I/O daemon stores its portion of the PVFS file in a file on the local file system on the I/O node. The name of this file is based on the inode number that the manager assigned to the PVFS file (in our example, 1092157504).

As mentioned above, when application processes (clients) open a PVFS file, the PVFS manager informs them of the locations of the I/O daemons. The clients then establish connections with the I/O daemons directly. When a client wishes to access file data, the client library sends a descriptor of the file region being accessed to the I/O daemons holding data in the region. The daemons determine what portions of the requested region they have locally and perform the necessary I/O and data transfers.

Figure 2 shows an example of how one of these regions, in this case a regularly strided logical partition, might be mapped to the data available on a single I/O node. (Logical partitions are discussed further in Section 3.3.) The intersection of the two regions defines what we call an I/O stream. This stream of data is then transferred in logical file order across the network connection. By retaining the ordering implicit in the request and allowing the underlying stream protocol to handle packetization, no additional overhead is incurred with control messages at the application layer.

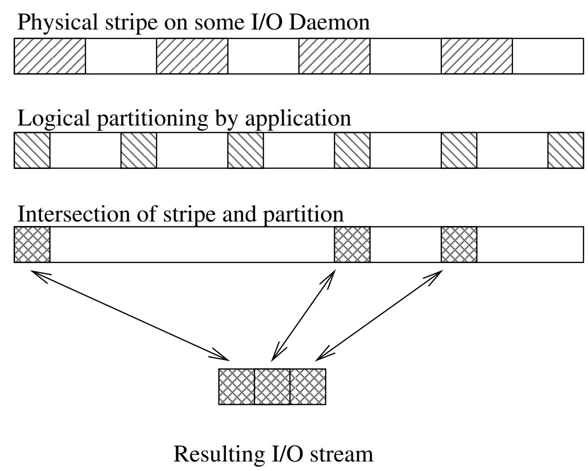


Figure 2: I/O stream example

3.3 Application Programming Interfaces

PVFS can be used with multiple application programming interfaces (APIs): a native API, the UNIX/POSIX API [15], and MPI-IO [13, 18]. In all these APIs, the communication with I/O daemons and the manager is handled transparently within the API implementation.

The native API for PVFS has functions analogous to the UNIX/POSIX functions for contiguous reads and writes. The native API also includes a "partitioned-file interface" that supports simple strided accesses in the file. Partitioning allows for noncontiguous file regions to be accessed with a single function call. This concept is similar to logical file partitioning in Vesta [9] and file views in MPI-IO [13, 18]. The user can specify a file partition in PVFS by using a special `ioctl` call. Three parameters, *offset*, *gsize*, and *stride*, specify the partition, as shown in Figure 3. The *offset* parameter defines how far into the file the partition begins relative to the first byte of the file, the *gsize* parameter defines the size of the simple strided regions of data to be accessed, and the *stride* parameter defines the distance between the start of two consecutive regions.

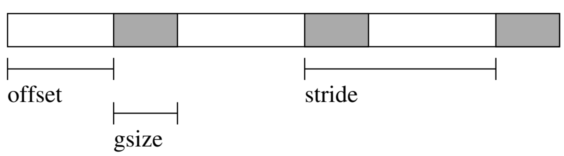


Figure 3: Partitioning parameters

We have also implemented the MPI-IO interface [13, 18] on top of PVFS by using the ROMIO implementation of MPI-IO [24]. ROMIO is designed to be ported easily to new file systems by implementing only a small set of functions on the new file system [30, 32]. This feature enabled us to have all of MPI-IO implemented on top of PVFS in a short time. We used only the contiguous read/write functions of PVFS in this MPI-IO implementation because the partitioned-file interface of PVFS supports only a subset of the noncontiguous access patterns that are possible in MPI-IO. Noncontiguous MPI-IO accesses are implemented on top of contiguous read/write functions by using a ROMIO optimization called data sieving [31]. In this optimization, ROMIO makes large contiguous I/O requests and extracts the necessary data. We are currently investigating how the PVFS partitioning interface can be made more general to support MPI-IO's noncontiguous accesses.

PVFS also supports the regular UNIX I/O functions, such as `read()` and `write()`, and common UNIX shell commands, such as `ls`, `cp`, and `rm`. (We note that `fcntl` file locks are not yet implemented.) Furthermore, existing binaries that use the UNIX API can access PVFS files without recompiling. The following section describes how we implemented these features.

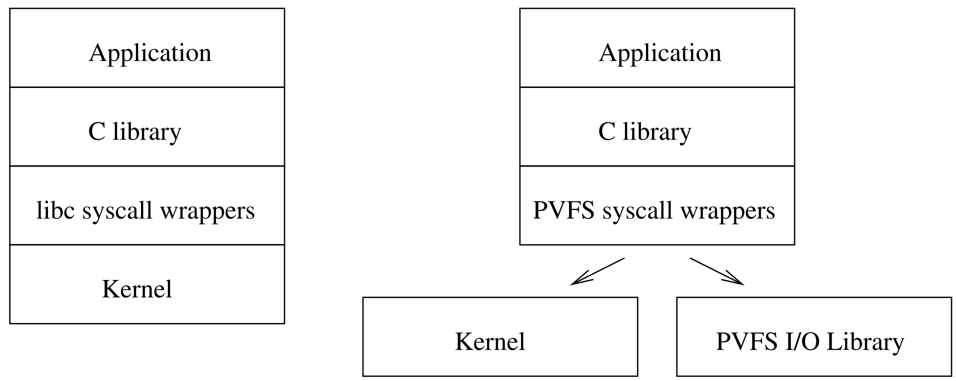
3.4 Trapping UNIX I/O Calls

System calls are low-level methods that applications can use for interacting with the kernel (for example, for disk and network I/O). These calls are typically made by calling wrapper functions implemented in the standard C library, which handle the details of passing parameters to the kernel. A straightforward way to trap system calls is to provide a separate library to which users relink their code. This approach is used, for example, in the Condor system [17] to help provide checkpointing in applications. This method, however, requires relinking of each application that needs to use the new library.

When compiling applications, a common practice is to use dynamic linking in order to reduce the size of the executable and to use shared libraries of common functions. A side effect of this type of linking is that the executables can take advantage of new libraries supporting the same functions without recompilation or relinking. We use this method of linking the PVFS client library to trap I/O system calls before they are passed to the kernel. We provide a library of system-call wrappers that is loaded before the standard C library by using the Linux environment variable `LD_PRELOAD`. As a result, existing binaries can access PVFS files without recompiling.

Figure 4a shows the organization of the system-call mechanism before our library is loaded. Applications call functions in the C library (`libc`), which in turn call the system calls through wrapper functions implemented in `libc`. These calls pass the appropriate values through to the kernel, which then performs the desired operations. Figure 4b shows the organization of the system-call mechanism again, this time with the PVFS client library in place. In this case the `libc` system-call wrappers are replaced by PVFS wrappers that determine the type of file on which the operation is to be performed. If the file is a PVFS file, the PVFS I/O library is used to handle the function. Otherwise the parameters are passed on to the actual kernel call.

This method of trapping UNIX I/O calls has limitations, however. First, a call to `exec()` will destroy the state that we save in user space, and the new process will therefore not be able to use file descriptors that referred to open PVFS files before the `exec()` was called. Second, porting this feature to new architectures and operating systems is nontrivial. The appropriate system library calls must be identified and included in our library. This process must also be repeated when the APIs of system libraries change. For example, the GNU C library (`glibc`) API is constantly changing, and, as a result,



a) Standard operation b) With PVFS library loaded

Figure 4: Trapping system calls

we have had to constantly change our code!

3.5 Linux Kernel VFS Module

While the trapping technique described above does provide the necessary functionality for using existing applications on PVFS files, the shortcomings of this method and the effort required to keep up with changes in the C library encouraged us to seek an alternative solution. The Linux kernel provides the necessary hooks for adding new file-system support via loadable modules without recompiling the kernel. Accordingly, we have implemented a module that allows PVFS file systems to be mounted in a manner similar to NFS [27]. Once mounted, the PVFS file system can be traversed and accessed with existing binaries just as any other file system. We note that, for the performance experiments reported in this paper, we used the PVFS library and not the kernel module.

4 Performance Results

We present performance results using PVFS on the Chiba City [7] cluster at Argonne National Laboratory. The cluster was configured as follows at the time of our experiments. There were 256 nodes, each with two 500-MHz Pentium III processors, 512 Mbytes of RAM, a 9 Gbyte Quantum Atlas IV SCSI disk, a 100 Mbits/sec Intel EtherExpress Pro fast-ethernet network card operating in full-duplex mode, and a 64-bit Myrinet card (Revision 3). The nodes were running Linux 2.2.15pre4. There were two MPI implementations: MPICH 1.2.0 for fast ethernet and MPICH-GM 1.1.2 for Myrinet. The kernel was compiled for a single processor; therefore, one processor on each machine was unused during our experiments. Out of the 256 nodes, only 60 nodes were available at a time for our experiments. We used some of those 60 nodes as compute nodes and some as I/O nodes for PVFS.

The Quantum Atlas IV 9 Gbyte disk has an advertised sustained transfer rate of 13.5–21.5 Mbytes/sec. The performance of the disk measured using the `bonnie` file-system benchmark [5] showed a write bandwidth of 22 Mbytes/sec and a read bandwidth of 15 Mbytes/sec when accessing a 512 Mbyte file in a sequential manner. The write performance measured by `bonnie` is slightly higher than the advertised sustained rates, perhaps because the test accessed the file sequentially, thereby allowing file-system caching, read ahead, and write behind to better organize disk accesses.

Since PVFS currently uses TCP for all communication, we measured the performance of TCP on the two networks on the cluster. For this purpose, we used the `ttcp` test, version 1.1 [29]. We tried three buffer sizes, 8 Kbytes, 64 Kbytes, and 256 Kbytes, and for all three, `ttcp` reported a bandwidth of around 10.2 Mbytes/sec on fast ethernet and 37.7 Mbytes/sec on Myrinet.

To measure PVFS performance, we performed experiments that can be grouped into three categories: concurrent reads and writes with native PVFS calls, concurrent reads and writes with MPI-IO, and the BTIO benchmark. We varied the number of I/O nodes, compute nodes, and I/O size and measured performance with both fast ethernet and Myrinet. We used the default file-stripe size of 16 Kbytes in all experiments.

4.1 Concurrent Read/Write Performance

Our first test program is a parallel MPI program in which all processes perform the following operations using the native PVFS interface: open a new PVFS file that is common to all processes, concurrently write data blocks to disjoint regions of the file, close the file, reopen it, simultaneously read the same data blocks back from the file, and then close the file. Application tasks synchronize before and after each I/O operation. We recorded the time for the read/write operations on each node and, for calculating the bandwidth, used the maximum of the time taken on all processes. In all tests, each compute node wrote and read a single contiguous region of size $2N$ Mbytes, N being the number of I/O nodes in use. For example, for the case where 26 application processes accessed 8 I/O nodes, each application task wrote 16 Mbytes, resulting in a total file size of 416 Mbytes. Each test was repeated five times, and the lowest and highest values were discarded. The average of the remaining three tests is the value reported.

Figure 5 shows the read and write performance with fast ethernet. For reads, the bandwidth increased at a rate of approximately 11 Mbytes/sec per compute node, up to 46 Mbytes/sec with 4 I/O nodes, 90 Mbytes/sec with 8 I/O nodes, and 177 Mbytes/sec with 16 I/O nodes. For these three cases, the performance remained at this level until approximately 25 compute nodes were used, after which performance began to tail off and became more erratic. With 24 I/O nodes, the performance increased up to 222 Mbytes/sec (with 24 compute nodes) and then began to drop. With 32 I/O nodes, the performance increased less quickly, attained approximately the same peak read performance as with 24 I/O nodes, and dropped off in a similar manner. This indicates that we reached the limit of our scalability with fast ethernet.

The performance was similar for writes with fast ethernet. The bandwidth increased at a rate of approximately 10 Mbytes/sec per compute node for the 4, 8, and 16 I/O-node cases, reaching peaks of 42 Mbytes/sec, 83 Mbytes/sec, and 166 Mbytes/sec, respectively, again utilizing almost 100% of the available TCP bandwidth. These cases also began to tail off at approximately 24 compute nodes. Similarly, with 24 I/O nodes, the performance increased to a peak of 226 Mbytes/sec before leveling out, and with 32 I/O nodes, we obtained no better performance. The slower rate of increase in bandwidth indicates that we exceeded the maximum number of sockets across which it is efficient to service requests on the client side.

We observed significant performance improvements by running the same PVFS code (using TCP) on Myrinet instead of fast ethernet. Figure 6 shows the results. The read bandwidth increased at 31 Mbytes/sec per compute process and leveled out at approximately 138 Mbytes/sec with 4 I/O nodes, 255 Mbytes/sec with 8 I/O nodes, 450 Mbytes/sec with 16 I/O nodes, and 650 Mbytes/sec with 24 I/O nodes. With 32 I/O nodes, the bandwidth reached 687 Mbytes/sec for 28 compute nodes, our maximum tested size. For writing, the bandwidth increased at a rate of approximately 42 Mbytes/sec, higher than the rate we measured with `ttcp`. While we do not know the exact cause of this, it is likely that some small implementation difference resulted in PVFS utilizing a slightly higher fraction of the true Myrinet bandwidth than `ttcp`. The performance levelled at 93 Mbytes/sec with 4 I/O nodes, 180 Mbytes/sec with 8 I/O nodes, 325 Mbytes/sec with 16 I/O nodes, 460 Mbytes/sec with 24 I/O nodes, and 670 Mbytes/sec with 32 I/O nodes.

In contrast to the fast-ethernet results, the performance with Myrinet maintained consistency as the number of compute nodes was increased beyond the number of I/O nodes, and, in the case of 4 I/O nodes, as many as 45 compute nodes (the largest number tested) could be efficiently serviced.

4.2 MPI-IO Performance

We modified the same test program to use MPI-IO calls rather than native PVFS calls. The number of I/O nodes was fixed at 32, and the number of compute nodes was varied. Figure 7 shows the performance of the MPI-IO and native PVFS versions of the program. The performance of the two versions was comparable: MPI-IO added a small overhead of at most 7–8% on top of native PVFS. We believe this overhead can be reduced further with careful tuning.

4.3 BTIO Benchmark

The BTIO benchmark [21] from NASA Ames Research Center simulates the I/O required by a time-stepping flow solver that periodically writes its solution matrix. The solution matrix is distributed among processes by using a multipartition distribution [6] in which each process is responsible for several disjoint subblocks of points (cells) of the grid. The solution matrix is stored on each process as C three-dimensional arrays, where

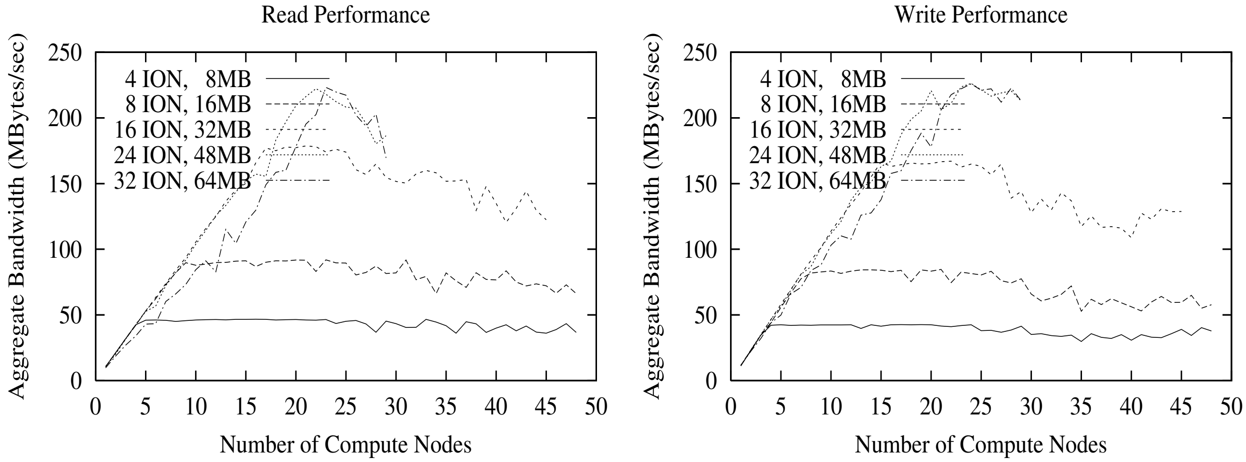


Figure 5: PVFS performance with fast ethernet

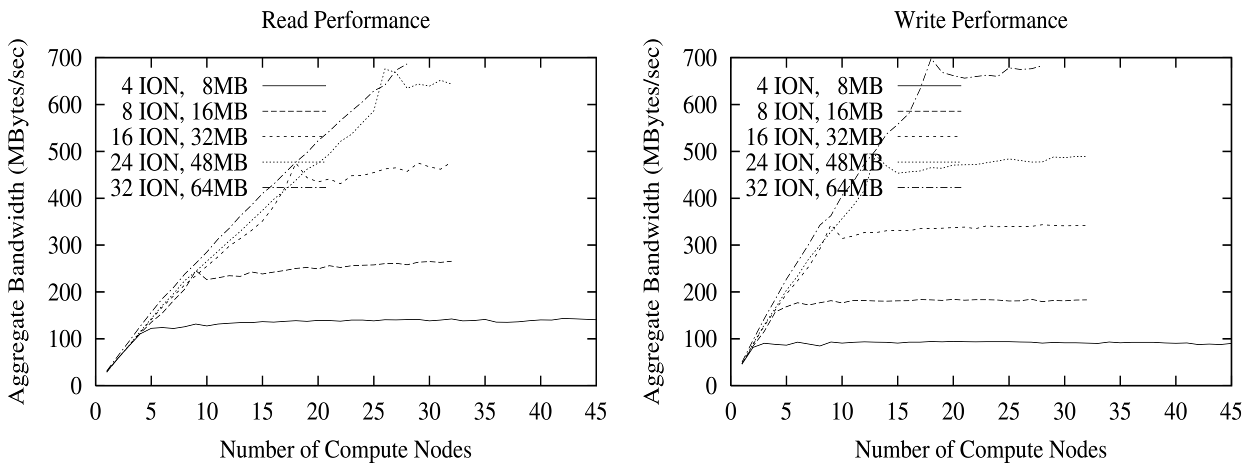


Figure 6: PVFS performance with Myrinet

C is the number of cells on each process. (The arrays are actually four dimensional, but the first dimension has only five elements and is not distributed.) Data is stored in the file in an order corresponding to a column-major ordering of the global solution matrix.

The access pattern in BTIO is noncontiguous in memory and in the file and is therefore difficult to handle efficiently with the UNIX/POSIX I/O interface. We used the "full MPI-IO" version of this benchmark, which uses MPI derived datatypes to describe noncontiguity in memory and file and uses a single collective I/O function to perform the entire I/O. The ROMIO implementation of MPI-IO optimizes such a request by merging the accesses of different processes and making large, well-formed requests to the file system [31].

The benchmark, as obtained from NASA Ames, performs only writes. In order to measure the read bandwidth for the same access pattern, we modified the benchmark to also perform reads. We ran the Class C problem size, which uses a $162 \times 162 \times 162$ element array with a total size of 162 Mbytes. The number of I/O nodes was fixed at 16, and tests were run using 16, 25, and 36 compute nodes (the benchmark requires that the number of compute nodes be a perfect square). Table 2 summarizes the results.

With fast ethernet, the maximum performance was reached with 25 compute nodes. With more compute nodes, the smaller granularity of each I/O access resulted in lower performance. For this configuration, we attained 49% of the peak concurrent-read performance and 61% of the peak concurrent-write performance mea-

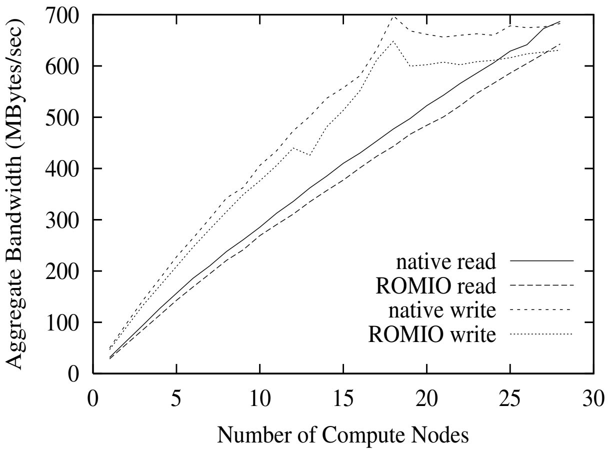


Figure 7: ROMIO versus native PVFS performance with Myrinet and 32 I/O nodes

Table 2: BTIO performance (Mbytes/sec), 16 I/O nodes, Class C problem size (162 × 162 × 162).

Compute Nodes	Fast Ethernet read	Fast Ethernet write	Myrinet read	Myrinet write
16	83.8	79.1	156.7	157.3
25	88.4	101.3	197.3	192.0
36	66.3	61.1	232.3	230.7

sured in Section 4.1. The other time was spent in the computation and communication required to merge the accesses of different processes in ROMIO's collective I/O implementation. Without this merging, however, the performance would have been significantly lower because of the numerous small reads and writes in this application.

With Myrinet, the maximum performance was reached with 36 compute nodes. Here we again see the benefit of a high-speed network in that even for the smaller requests resulting from using more compute nodes, we were able to attain higher performance. The performance obtained was about 51% of the peak concurrent-read performance and 70% of peak concurrent-write performance measured in Section 4.1.

5 Conclusions and Future Work

PVFS brings high-performance parallel file systems to Linux clusters and, although more testing and tuning are needed for production use, it is ready and available for use now. The inclusion of PVFS support in the ROMIO MPI-IO implementation makes it easy for applications written portably with the MPI-IO API to take advantage of the available disk subsystems lying dormant in most Linux clusters.

PVFS also serves as a tool that enables us to pursue further research into various aspects of parallel I/O and parallel file systems for clusters. We outline some of our plans below.

One limitation of PVFS, at present, is that it uses TCP for all communication. As a result, even on fast gigabit networks, the communication performance is limited to that of TCP on those networks, which is usually unsatisfactory. We are therefore redesigning PVFS to use TCP as well as faster communication mechanisms (such as VIA [34], GM [20], and ST [25]) where available. We plan to design a small communication abstraction that captures PVFS's communication needs, implement PVFS on top of this abstraction, and implement the abstraction separately on TCP, VIA, GM, and the like. A similar approach, known as an abstract device interface, has been used successfully in MPICH [12] and ROMIO [32].

Some of the performance results in this paper, particularly the cases on fast ethernet where performance drops off, suggest that further tuning is needed. We plan to instrument the PVFS code and obtain detailed performance measurements. Based on this data, we plan to investigate whether performance can be improved by tuning some parameters in PVFS and TCP, either *a priori* or dynamically at run time.

We also plan to design a more general file-partitioning interface that can handle the noncontiguous accesses supported in MPI-IO, improve the client-server interface to better fit the expectations of kernel interfaces, design a new internal I/O-description format that is more flexible than the existing partitioning scheme, investigate adding redundancy support, and develop better scheduling algorithms for use in the I/O daemons in order to better utilize I/O and networking resources.

6 Availability

Source code, compiled binaries, documentation, and mailing-list information for PVFS are available from the PVFS web site at `http://www.parl.clemson.edu/pvfs/`.

Information and source code for the ROMIO MPI-IO implementation are available at `http://www.mcs.anl.gov/romio/`.

References

[1] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 109–126. ACM Press, December 1995.

[2] Rajesh Bordawekar, Steven Landherr, Don Capps, and Mark Davis. Experimental evaluation of the Hewlett-Packard Exemplar file system. *ACM SIGMETRICS Performance Evaluation Review*, 25(3):21–28, December 1997.

[3] Peter J. Braam. The Coda distributed file system. *Linux Journal*, #50, June 1998.

[4] Peter J. Braam, Michael Callahan, and Phil Schwan. The InterMezzo filesystem. In *Proceedings of the O'Reilly Perl Conference 3*, August 1999.

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

[6] J. Bruno and P. Cappello. Implementing the Beam and Warming Method on the Hypercube. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, January 1988.

[7] Chiba City, the Argonne scalable cluster. `http://www.mcs.anl.gov/chiba/`.

[8] Coda file system. `http://www.coda.cs.cmu.edu`.

[9] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.

[10] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, George S. Almasi, Sandra Johnson Baylor, Anthony S. Bolmarcich, Yarsun Hsu, Julian Satran, Marc Snir, Robert Colao, Brian Herr, Joseph Kavaky, Thomas R. Morgan, and Anthony Zlotek. Parallel file systems for the IBM SP computers. *IBM Systems Journal*, 34(2):222–248, January 1995.

[11] Intel Scalable Systems Division. Paragon system user's guide. Order Number 312489-004, May 1995.

[12] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message-passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

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

[14] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, July 1995. ACM Press.

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

[16] InterMezzo. `http://www.inter-mezzo.org`.

[17] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report Computer Sciences Technical Report #1346, University of Wisconsin-Madison, April 1997.

[18] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997. `http://www.mpi-forum.org/docs/docs.html`.

[19] Steven A. Moyer and V. S. Sunderam. PIOUS: a scalable parallel I/O system for distributed computing environments. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.

[20] Myrinet software and documentation. `http://www.myri.com/scs`.

[21] NAS application I/O (BTIO) benchmark. `http://parallel.nas.nasa.gov/MPI-IO/btio`.

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

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

[24] ROMIO: A High-Performance, Portable MPI-IO implementation. `http://www.mcs.anl.gov/romio`.

[25] Scheduled Transfer—application programming interface mappings (ST-API). `http://www.hippi.org/cSTAPI.html`.

[26] Huseyin Simitci, Daniel A. Reed, Ryan Fox, Mario Medina, James Oly, Nancy Tran, and Guoyi Wang. A framework for adaptive storage of input/output on computational grids. In *Proceedings of the Third Workshop on Runtime Systems for Parallel Programming*, 1999.

[27] Hal Stern. *Managing NFS and NIS*. O'Reilly & Associates, Inc., 1991.

[28] Hakan Taki and Gil Utard. MPI-IO on a parallel file system for cluster of workstations. In *Proceedings of the IEEE Computer Society International Workshop on Cluster Computing*, pages 150–157. IEEE Computer Society Press, December 1999.

[29] Test TCP. `ftp://ftp.arl.mil/pub/ttcp/`.

[30] Rajeev Thakur, William Gropp, and Ewing Lusk. An abstract-device interface for implementing portable parallel-I/O interfaces. In *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187. IEEE Computer Society Press, October 1996.

[31] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, February 1999.

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

[33] The parallel virtual file system. `http://www.parl.clemson.edu/pvfs/`.

[34] VI architecture. `http://www.viarch.org`.

[35] XFS: A next generation journalled 64-bit filesystem with guaranteed rate I/O. `http://www.sgi.com/Technology/xfs-whitepaper.html`.