SANDIA REPORT

SAND2004-1425
Unlimited Release
Printed April 2004

The Parallel Virtual File System for Portals

James A. Schutt

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2004-1425
Unlimited Release
Printed April 2004

The Parallel Virtual File System for Portals

James A. Schutt
Advanced Networking Integration
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-0806

Abstract

This report presents the result of an effort to re-implement the Parallel Virtual File System (PVFS) using Portals as the transport. This report provides short overviews of PVFS and Portals, and describes the design and implementation of PVFS over Portals. Finally, the results of performance testing of both stock PVFS and PVFS over Portals are presented.

Contents

Figures

Intentionally Left Blank

The Parallel Virtual File System for Portals

1 Introduction

The Parallel Virtual File System* (PVFS)(Ligon and Ross, 2001) allows a group of cooperating processes running a parallel application on a cluster of computers to transparently share access to a file or files distributed across a set of I/O servers in the cluster. PVFS is designed to be scalable on both the client and server sides. In other words, file system bandwidth increases with the number of clients until the servers are saturated, and maximum bandwidth also increases as servers are added. PVFS is available in source code form under the GNU General Public License (GPL), and is primarily supported for Linux and Unix platforms. PVFS clients and servers use the Unix sockets API to communicate via TCP/IP.

At the time of this writing, there are two major versions of PVFS available, version 1.x and version 2. All discussion of PVFS in this report references version 1.x, since there are some significant design and implementation differences between PVFS 1.x and PVFS 2.

The Portals API is a message-passing interface designed by researchers from Sandia National Laboratories and the University of New Mexico (Brightwell et.al.1999). The Portals interface is designed to facilitate scalable, high-performance implementations for massively parallel processors (MPPs). As such, Portals forms the basis for much of the parallel computing platform research at Sandia. In addition, the ASCI Red Storm supercomputer (Tomkins and Camp, 2003) currently being built for Sandia is designed specifically to support a high-performance implementation of Portals.

The Portals API has undergone multiple major revisions. The current major version is version 3, and any reference to Portals in this document refers to that major version. The Brightwell et.al. (1999) document describes version 3.0 of the API, while the Red Storm implementation is based on version 3.3 of the API. A document describing Portals 3.3 was in preparation at the time of this writing.

When the work reported here was begun, Sandia did not have a parallel file system that used the Portals API. The goal of this work was to produce a PVFS implementation that used the Portals API natively, so that it could be used on Sandia MPPs that provide only a Portals transport on compute nodes. The alternative, writing an interface adapter that converted sockets calls to Portals calls, could not have taken advantage of the bypass mode of operation that Portals allows.

Since the Portals and sockets APIs are significantly different, using the Portals API natively in PVFS entailed rewriting the PVFS client library and daemons. The hope was that such a rewrite would allow the full performance potential of Portals to be realized. In addition, the rewrite allowed a major security issue in PVFS to be addressed.

This report reviews the major architectural features of PVFS, and discusses some differences between the PVFS and PVFS over Portals (PVFSP) designs and implementations. Next, a performance comparison is reported. Finally, a short discussion of the work remaining to complete the PVFSP implementation is presented.

* PVFS is available from http://www.parl.clemson.edu/pvfs/index.html.

2 Overview of PVFS

PVFS was designed for use as a high-speed scratch file system, with an emphasis on high-speed data movement rather than metadata operations. The components that comprise PVFS are shown in Figure 1. On the server side is the metadata manager daemon, *mgr*, and one or more I/O daemons, *iod*. The *mgr* daemon is responsible for namespace and other metadata operations, while the *iod* daemons are responsible for data movement. On the client side is the PVFS application I/O library, as well as a Linux kernel module and accompanying daemon, *pvfsd*, which together provide PVFS file access through the Linux kernel virtual file system switch (VFS). This latter data path is particularly important to users as it allows all the standard Unix utilities, such as *ls*, *rm*, and *mv*, to be used to manipulate PVFS files.

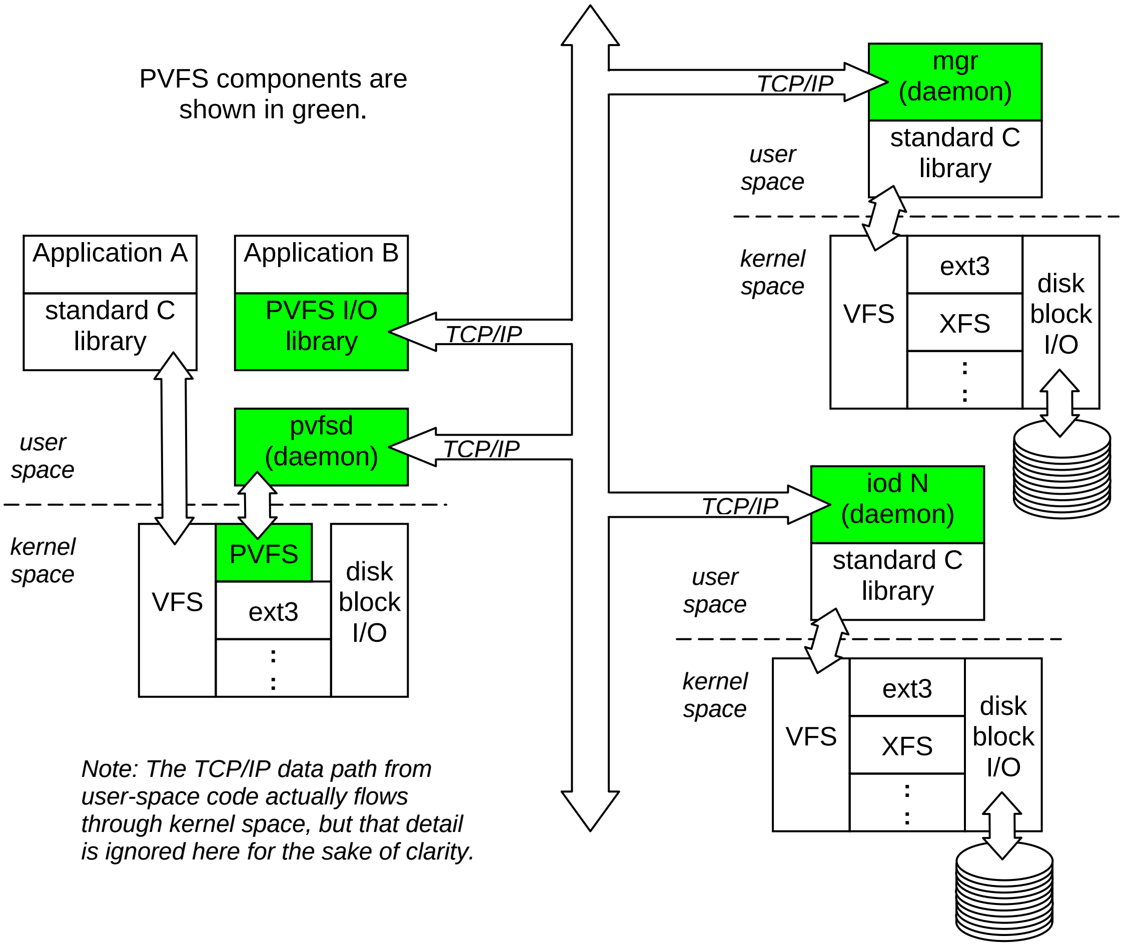
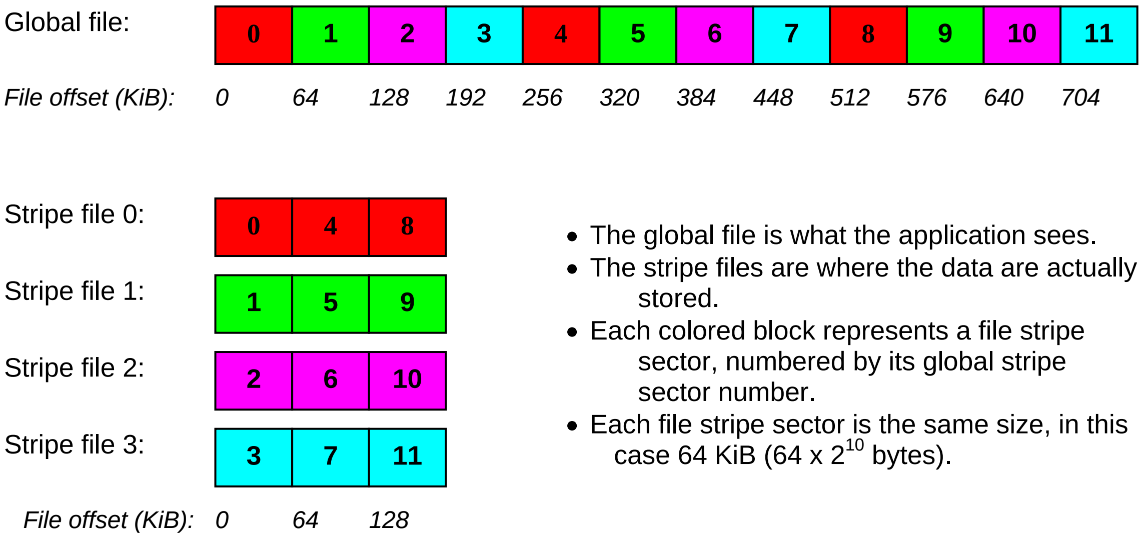


Figure 1 Components and data paths in the PVFS file system.

The PVFS daemons and client library communicate using TCP/IP via the standard sockets API. The *pvfsd* daemon communicates with the PVFS kernel module through standard Unix I/O system calls on a device file, /dev/pvfsd. The *mgr* and *iod* daemons use standard Unix system calls to manipulate the data that is stored on disk.

The PVFS design is based on the idea of striping the data from a single file across multiple file servers. For example, with a 64 KiB[*] stripe sector size, the first 64 KiB of data in the file is served by the first *iod*, the second 64 KiB by the second *iod*, etc, as shown in Figure 2. PVFS allows multiple clients to simultaneously read/write to a single file, as well as allowing multiple clients to each read/write its own file, again simultaneously. PVFS does not implement any locking primitives, so parallel applications that write to a single file must protect themselves against simultaneous write or read/write operations to the same file offset.



- The global file is what the application sees.
- The stripe files are where the data are actually stored.
- Each colored block represents a file stripe sector, numbered by its global stripe sector number.
- Each file stripe sector is the same size, in this case 64 KiB (64 x 2^{10} bytes).

Figure 2   Global and stripe file offsets in PVFS.

For most requests, such as open, close, unlink, and fstat, PVFS clients contact the single metadata manager daemon, *mgr*. Most of these requests require that the *mgr* daemon in turn contact the *iod* daemons to complete the request. However, the *mgr* is not involved in read/write operations, which mitigates its potential to serialize access to data stored in PVFS files. For these operations, clients contact the *iod* daemons directly.

The *iod* and *mgr* daemons are implemented as single-threaded processes that loop continuously, accepting new requests and responding to existing requests. They use select() to discover sockets that are ready to send or receive data, and use non-blocking send()/recv() calls for network traffic. The *iod* daemons use mmap() to read data from disk, and madvise() to minimize delays waiting for data to be paged in from disk. Data is written to disk using write().

As alluded to in the introduction, prior to version 1.6.1 PVFS was susceptible to a simple attack that would circumvent file permission restrictions. As with any network-based file system, PVFS requests specify the user and group identity of the process making the request. However, there was no authentication mechanism preventing an attacker from using a copy of the PVFS I/O library that

[*] The prefixes for binary multiples are Ki = 2^{10}, Mi = 2^{20}, and Gi = 2^{30}, developed by International Electrotechnical Commission (IEC) Technical Committee 25. These compare with the SI prefixes, k = 10^3, M = 10^6, and G = 10^9. Thus 64 KiB = 64 x 2^{10} bytes, or 65,536 bytes, while 64 kB = 64 x 10^3, or 64,000 bytes. See http://physics.nist.gov/cuu/Units/binary.html for more information.

was modified to specify user/group ids of a user other than the attacker. For example, an attacker could specify a privileged user id in all requests, and gain access to any file stored on a PVFS file system.

This author submitted a patch, accepted into PVFS 1.6.1, that added an optional trusted-host level of authentication. Unix systems typically have a range of ports from which only privileged users can initiate connections. Starting with version 1.6.1, the *mgr* and *iod* daemons could be configured to accept only connections originating from privileged ports on a limited range of hosts. The practical effect of this option is that the *mgr* and *iod* daemons would accept only requests from hosts where site administrators had confidence that *pvfsd* was the only privileged process making PVFS requests. The benefit is that user and group identities in requests can be trusted, and file permissions have meaning. The drawback is that only a privileged user (i.e., root) can directly employ the PVFS client I/O library.

3 Overview of PVFS over Portals

3.1 Portals Basics

A familiarity with the basic features of the Portals API will make the discussion of PVFS over Portals that follows more accessible. Readers wishing further details should consult Brightwell et.al.(1999).

The Portals specification describes a message passing API whose primary goal is to facilitate implementations that scale to tens of thousands of nodes. The Portals API facilitates implementations that are both OS-bypass, where the operating system does not participate in making progress on messages, and application-bypass, where the application does not participate in message progress. Scalability is facilitated because the data movement specified by Portals is connectionless, allowing an implementation to keep minimum state information. Data movement in Portals is also asynchronous, where an application is notified of request completion through an event mechanism.

Application memory used for messaging is identified to Portals using a construct known as a memory descriptor. A memory descriptor contains a starting memory address and extent, as well as other data specifying the disposition of the descriptor under various conditions. Memory descriptors may be inserted into match lists, where the descriptors will respond to remote operations under the control of various matching parameters. These matching parameters include type of operation, remote node and Portals process identifiers, and match bits.

Match bits are 64 bits of data associated with each request and match entry. In addition, each match entry has 64 ignore bits, which specify a mask to be applied to the entry match bits when determining matches. Thus, match bits can be used to specify both unique matches and class matches. In the unique case, the match entry ignore bits are all zero, and the request match bits must be identical to the match entry match bits. For class matching, some of the ignore bits are set, and only a subset of match bits are considered when determining matches.

Memory descriptors are also used to make send (put) and receive (get) requests. When a request is made, the request match bits are compared on the target with the match bits of each match list entry in turn. The memory descriptor for the first entry that matches is used to service the request. If there are no suitable match entries, the request is dropped by the target.

Applications are notified of request completion via events and event queues. An application can instruct Portals to create one or more event queues, and can specify the event queue to be used for notification when creating a memory descriptor. Thereafter, events are delivered to the event queue whenever Portals takes some action on the memory descriptor. Note that event delivery is strictly a local operation. The application polls its event queues to dequeue new events.

In Portals 3.0, there are five event types:

PTL_EVENT_GET
PTL_EVENT_PUT
PTL_EVENT_ACK
PTL_EVENT_SENT
PTL_EVENT_REPLY

The put and get event types are delivered after a remote put or get operation has succeeded on a local memory descriptor associated with a match entry. The ack, sent, and reply event types are delivered in response to a local put or get operation. If requested, an ack event is delivered for a put operation after the local process receives an acknowledgement from the remote process that the data successfully arrived. A sent event is delivered for a put operation after the data has successfully left the local process, and for a get operation after the request has successfully left the local process. A reply event is delivered for a get operation after the requested data has successfully arrived in the local process.

Notice that the Portals 3.0 specification does not admit the possibility of a transport malfunction, as there is no way for the application to be notified of an error condition. Later versions of the Portals specification, still being finalized at the time of writing, address this deficiency.

3.2 PVFS over Portals Implementation

The PVFS over Portals (PVFSP) implementation is centered on the lifecycle of a request. Since Portals is asynchronous, request processing for both clients and servers is inherently asynchronous.

3.2.1 PVFSP Daemon Request Processing

TCP/IP-based daemons have two mechanisms to throttle the rate of incoming requests, which is necessary to avoid overloading the daemon with too much work. One mechanism is the rate at which the daemons accept new connections. The second is the rate at which daemons read data from an established connection, since a client cannot push additional data into a connection once its TCP window is closed.

Accepting new requests poses several challenges for the Portals-based daemon subjected to a high influx of requests. If the daemon has not allocated sufficient memory, and prepared sufficient memory descriptors linked into the match list, incoming requests will simply be dropped. In addition, if the event queue associated with memory descriptors allocated for incoming requests is too small, it can be overrun, producing the same effect as one or more dropped requests. A dropped request requires a timeout mechanism to detect, which is tricky to implement correctly when a single client request may spawn multiple secondary requests between *mgr* and *iod* daemons.

Another challenge is mitigating the effect of a denial-of-service attack, where the attacker attempts to flood the daemon with bogus requests.

These issues are addressed in PVFSP daemons by using fixed-size requests, and allocating memory in slabs large enough to hold many requests. An event queue is allocated for each slab, and is large enough to handle all events delivered as requests are received into that slab. Slabs for which all requests have completed are reused. If the number of available request slots falls below some threshold, the daemon can allocate additional slabs and event queues. Request timeout/resend and duplicate request detection are implemented, but they are a recovery mechanism of last resort and not well tested.

A new request is subject to several simple checks for validity. Is the request the right length? Does it have a known request type? Is it properly authenticated? If any of these tests fail, the request is logged and dropped.

After an incoming request is validated, it is associated with additional data that records the processing state of each request, and added to the tail of the pending request queue. The daemons maintain queues of active and pending requests, and pending requests are moved to the active queue in FIFO order. By controlling the number of requests it is actively servicing, a daemon can minimize the possibility that its rate of request completion will decrease as the outstanding request count increases.

Each type of request has an associated request handler function that implements a simple state engine, which attempts to advance the state of any request for which it is called. A daemon iterates through its active request queue, calling the appropriate handler for each request in turn. As with PVFS, PVFSP daemons are single-threaded, so it is important that request handlers do not make system calls that block, or request processing will stall.

Some operations take variable length data to specify, for example, operations on one or more file paths. For such operations, the request contains unique match bits for a memory descriptor identifying the memory holding the variable-length data. In these cases, the first step in request processing is to retrieve the variable length data. Once the variable length data is retrieved and the operation is completely specified, it is attempted. Note that some operations performed by the *mgr* daemon require secondary requests made to the *iod* daemons to complete. If so, the operation is not complete until the secondary requests have completed.

Once the operation has completed, a reply containing any requested data, and the operation status, is constructed and sent to the request originator, again using unique match bits specified in the original request. When the daemon receives the Portals sent event for the reply, the request is considered complete, the request is deleted from the active queue, and its resources are freed.

An additional Portals event queue is required to receive notification for the Portals operations used to advance the state of a request. In the PVFSP implementation, these operations are known as data operations, and the associated event queue is known as the data event queue. If the data event queue is overrun, processing for any requests depending on the lost events will stall.

This situation is difficult to recover, and is to be avoided at all costs. PVFSP daemons use two Portals events queues for this, known as the add queue and the poll queue. Memory descriptors for new data operations reference the add event queue, and the poll event queue is checked first for new events. A count of outstanding operations is kept for each queue. When the operation count for the add event queue exceeds a limit based on the size of the event queue, action must be taken. If the operation count for the poll queue is zero it is reallocated larger than the add queue. Regardless of whether the poll queue is reallocated, the roles of the poll and add queues are exchanged. This method allows the available data event queue space to grow, although it cannot cope with a sudden large increase in need for data event queue space.

Polling of the request slab and data event queues is interleaved with request processing, so that no subset of the work performed by a daemon is starved by another subset.

3.2.2 PVFSP Authentication

When this work was begun, one of the main goals was to incorporate a framework to address the authentication shortcomings of PVFS. This framework should provide authentication that scales to tens of thousands of clients, minimizes the resources used for authentication, and leverages any unique features of parallel computing systems where PVFS over Portals may be installed.

Of the many potential attacks against an authentication system, one class that was not considered was the class of attacks that begin with the assumption that the attacker has gained privileged access to a host that is authorized by site administrators to participate in a PVFSP file system. The reason is that any client authentication method that scales to tens of thousands of clients will require a key or authentication token resident on the client host. An attacker with privileged access on that host has access to any such keys, and can use them to impersonate authorized users. However, it should be noted that in the event a client host is compromised, authentication systems are most secure if they restrict attackers to impersonating users whose keys were actually resident on that system.

File system deployments can be cleaved into two classes: those where file system network traffic can be sniffed by potential attackers, and those where it cannot. In the former case authenticators must be based on strong encryption, while in the latter they should be difficult to guess but need not employ encryption. The PVFSP authentication framework admits either case, or both cases simultaneously.

The authentication framework implemented in PVFS over Portals is based on a shared authentication context, which in PVFSP is named a session. When a user wishes to access PVFSP files, that user presents identity credentials to the *mgr* daemon, and receives a session authenticator that uniquely identifies the user to the *mgr* daemon. The *mgr* daemon also forwards the authenticator to the *iod* daemons, but not the user identity associated with it, as they do not need the actual user identity.

Every PVFSP request includes the session authenticator. If the user submits a parallel job, the session authenticator is distributed along with the executable so that it is available to each client in the job. The daemons search for the request authenticator in their list of valid authenticators, and if it is not present discard and log the request. The *mgr* daemon can determine the user identity associated with the authenticator for use in file permissions checking.

When an application opens a file, the *mgr* daemon generates a file access authenticator once it determines that the user is authorized to access the file. The access authenticator is returned to the client, and is also sent to the *iod* daemons, along with the access permissions it authorizes. A client read or write request contains the access authenticator, so that the *iod* daemons can refuse the request if the file was not opened with the appropriate access.

PVFSP daemons must also authenticate to each other, so that requests made between daemons can be authenticated. The framework implementation in PVFSP assumes that this will be done using authenticators that are stored on the hosts running the daemons.

If PVFSP were to be deployed in a mixed environment, where the daemons were serving some clients over a network that could not be sniffed, and others over a network that could be sniffed, the implementation would need to be able to detect this. When the *mgr* daemon granted a session authenticator, it would be a cryptographic-based authenticator unless the *mgr* could determine that all the clients and daemons participating in the session were reachable over the network that was secure from sniffing. The framework supports making this decision on a session-by-session basis.

At the time of this writing, PVFSP is implemented for systems where it is assumed that file system network traffic cannot be sniffed, so that encryption-based authenticators are unnecessary. The session authenticator is implemented as a 64-bit random number, and the access authenticator is a 32-bit random number, both generated by the *mgr* daemon. The current implementation lacks a finite lifetime for the authenticators, and also lacks a lockout mechanism to deny access when multiple requests with invalid authenticators are detected. The lockout mechanism would prevent a guessing attack on the random numbers used as authenticators, and legitimate users that were locked out would alert system administrators to possible attacks in progress, or other problems.

The implementation also has a compile-time option to replace session-based authentication with a user id in each request. This is subject to the same authentication issues as the PVFS implementation, and its use should be viewed as a convenience while the implementation is under development.

3.2.3 PVFS and PVFSP Implementation Differences

This section documents some specific implementation differences between PVFS and PVFS over Portals. Some of these changes are a direct consequence of the Portals implementation, some are an effort to increase performance, and some address perceived shortcomings in the PVFS implementation.

Asynchronous client I/O requests

As previously mentioned, Portals is inherently asynchronous, and so is the PVFSP implementation. Thus, it was trivial to implement asynchronous client read and write calls, along with a client function to poll for completion. In fact, the synchronous versions of read and write merely bundle the asynchronous calls with a completion polling loop. PVFS does not implement such calls, although it would not be difficult to do so.

Stripe sector size

In the PVFS client library, the open function includes extra arguments that allow the caller to specify file striping parameters at file creation. The file striping parameters are the stripe sector size, the number of stripes, and the *iod* assigned to the first stripe sector in the file. While some benefit may be gained by using a smaller stripe sector size in an application that uses smaller I/O buffer sizes, there is really nothing to be gained by allowing application control of the other stripe parameters.

The downside to allowing such control is that if it is widely used, the disk capacity serving the individual *iod* daemons is very likely to be consumed unevenly, causing the file system to prematurely appear full. In addition, aggregate system throughput may be reduced if some *iod* daemons are busier than others.

For these reasons, the PVFSP implementation does not allow file striping parameters to be specified at file creation time. Instead, stripe sector size and number of stripes is set when a PVFSP file system is created, and the *iod* assigned to the first stripe is determined randomly at file creation.

Note that if a PVFS application does not specify striping parameters at file creation, default values are used, and the default number of stripes is the maximum number of *iod* daemons available. In addition, the PVFS *mgr* daemon has a run-time option specifying that the *iod* assigned to the first stripe is to be determined randomly at file creation.

Multiple file systems

PVFS daemons can each serve only one file system. PVFSP daemons can each serve multiple file systems. One application of this is to statically partition the PVFSP namespace into multiple sub-trees, each served by its own *mgr* daemon, but with all *mgr* daemons served by one set of *iod* daemons.

On-disk format

PVFS metadata files contain the file striping parameters mentioned above, as well as file inode number, owner, group, file size, modification time, access time, change time, mode, and file system root directory inode number. The metadata file is continuously updated.

PVFSP metadata files contain only the file inode number and the file striping parameters.

PVFS and PVFSP *iod* daemons use a different directory structure and naming scheme to store the data files.

File permissions checking

PVFS uses the user and group ids stored in a metadata file to do its own file permissions checking.

For the PVFSP implementation, this author did not believe that it was useful to duplicate the file permissions checking already implemented in the kernel, with the attendant possibility of introducing bugs. Instead, permissions checking is implemented by having the *mgr* daemon set its effective group and user ids to that of the user making the request, and attempting to perform the requested operation. The status of the operation is recorded for later return to the client, and the daemon resets its effective group and user ids back its real (privileged) values to continue processing. In this way the kernel hosting the *mgr* daemon has complete responsibility for permissions checking.

Opening files

In the PVFS implementation, for each client open request the *mgr* daemon opens the metadata file, does permissions checking, closes the metadata file, and sends open requests for the associated data files to the *iod* daemons. This implementation causes the *iod* daemons to open the data file once for each client open request for a given file, and complicates correct handling of unlink after open since the metadata file is not held open.

In the PVFSP implementation, the *mgr* daemon opens the metadata file, and sends open requests to the *iod* daemons for the associated data files, on the first successful client open request for a file. Subsequent open requests for the same file are counted, but no further action is taken unless the *iod* daemons need to be updated with a new access authorization, or an open request specifies the O_TRUNC option. Note that because of the permissions checking implementation described above, the *mgr* opens and closes the metadata file once per open request using the effective user and group ids specified in the request. However, the metadata file is always held open while any client has the file open.

As a file is closed by clients, the open file count is decremented. When the last client holding a file open closes the file, the *mgr* daemon sends requests to the *iod* daemons to close the associated data files, and then closes the metadata file. Note that unlink after open is trivially correct with this implementation.

Memory-mapped regions in *iod*

Both PVFS and PVFSP use the mmap() system call in *iod* daemons to implement the read function. However, they differ in the way mapped regions are managed.

The PVFS *iod* implementation maintains one mapped region per open data file instance. The extent of each mapped region is either 512 KiB (default), or the size of the read request aligned to system page boundaries, whichever is larger. The first read request to a file causes the region to be mapped, but the region is not remapped until a subsequent read request falls outside of the current mapped region. After a region is mapped, the madvise() system call is made for the region, and processing of the request list is continued without attempting to read the mapped region until the next pass through the request list. The madvise() system call causes disk requests for the entire region to be queued, which on some systems results in bringing the requested data into memory faster than if it were paged in as the region was accessed.

The PVFSP *iod* implementation maintains a list of mapped regions that are managed in least-recently-used (LRU) fashion. Each mapped region is the same size, which can be specified as a multiple of the stripe sector size when an *iod* is started. The amount of address space an *iod* can use for file mappings can also be specified at runtime. When a new read request arrives at an *iod*, it first attempts to attach each stripe sector needed to satisfy the request to an existing mapping. If any stripe sectors cannot be attached, new mappings are created as needed. If the address space allotted to mappings has been exhausted, the oldest mapping not currently in use is remapped suitably for the read request.

When a new mapping is created, madvise() is called for the mapped region. Reading the region is not allowed until the mincore() system call indicates that all pages of the mapped region are in memory. This prevents read() system calls from blocking while data is paged into memory.

This method of managing mapped memory prevents duplicated and overlapping mapped regions. It also prevents resources devoted to file mapping from growing without bound as the number of files being read increases. Finally, it may minimize thrashing of mapped regions if a file is read in some order other than sequential.

4 Performance Testing

This section describes a series of tests that compare the performance of PVFS 1.6.2 and the PVFS over Portals implementation. The testing is limited to I/O performance, and does not address metadata operation performance.

At the time this testing was performed, there was no Portals 3 implementation available to the author that provided both OS bypass and application bypass, and supported user-space applications. This testing was performed using the Cplant Portals library, which is based on the Portals 3.0 specification. The Cplant Portals library is implemented using the concept of a Network Abstraction Layer (NAL) to provide data transport. This author implemented a TCP/IP-based NAL that operated completely in user space, and used that NAL for the testing reported in this section.

Thus, when assessing the performance comparisons between PVFS and PVFSP presented here, the reader should keep in mind that the full performance potential of a Portals-based implementation could not be realized.

4.1 Baseline

The test program used to measure I/O throughput was Lee Ward's parallel I/O benchmark program, *piob,* slightly modified so that it could be compiled and linked against either the PVFS or the PVFSP client library. This program sequentially writes and then reads files from multiple clients in parallel. Each client can write and read its own file, or all clients can write and read the same file, with each client responsible for a different subset of data in the file. This program also has an option that causes each client to read a file different from the one it wrote, or in the case of all clients using the same file, a subset of the file different from what it wrote. This option eliminates any benefit from client data caching. Although neither the PVFS nor the PVFSP client libraries implement data caching, this option was used for the sake of consistency with the testing of other file systems in use at Sandia.

PVFS was configured using default settings. Both PVFS and PVFSP used eight *iod* daemons, with *mgr* and *iod* daemons each running on its own host. The testing was conducted with both sets of daemons running simultaneously on the same set of hosts, but only one test at a time was performed, against either a PVFS or a PVFSP file system. Both PVFS and PVFSP were configured to use a 64 KiB stripe sector size and a 64 KiB socket buffer size.

PVFS *iod* daemons used the default 512 KiB minimum file mapping extent. PVFSP *iod* daemons used a 512 KiB file mapping extent, with mappings limited to 32 MiB of address space. Note that this is the same amount of address space consumed by a PVFS *iod* serving a file to 64 clients, if each mapping was using the default minimum extent of 512 KiB.

Testing was performed on the Vplant cluster, comprising 224 nodes in addition to the nine nodes used to run the *mgr* and *iod* daemons. These 224 nodes are partitioned into a 128-node group intended for visualization applications, and a 96-node group used for simulation applications. Only the visualization nodes were used to host clients for this testing, as the simulation nodes generally have a much higher utilization. Compute nodes were shared with other users during the testing, but the I/O nodes were dedicated to this testing.

Characteristics of the client and server nodes are presented in Table 1. The cluster used a Myrinet 2000 interconnect with Lanai 9 interface cards, running Myrinet GM 1.6.5. The *iod* nodes were each equipped with a Qlogic QLA2312 fibre channel adapter, and connected to two Data Direct Networks (DDN) S^{2}A 8000 fibre channel controllers using 2 Gb/s optical links. Version 4.06.10 of the qla2xxx device driver from Qlogic was used, with the tagged queuing depth set to 255. All the daemons in this testing read from and wrote to Linux ext3 file systems, which were mounted with the default "data=ordered" option.

Extensive previous experience with PVFS has shown that write performance can be greatly affected by kernel dirty block write-out tuning. The parameters in the file /proc/sys/vm/bdflush control this tuning. These parameters can be read with the command *cat /proc/sys/vm/bdflush*, and on a stock kernel.org 2.4.24 kernel the values reported are "30 500 0 0 500 3000 60 20 0". The first value is the percent of available blocks that must be dirty before write-out begins.

Table 1 Test Cluster Node Configurations.

	Compute Nodes	Server Nodes
Processor	2-way SMP 2.0 GHz Pentium 4 Xeon 256 KiB L2 cache	2-way SMP 2.4 GHz Pentium 4 Xeon 512 KiB L2 cache
Memory	1.0 GiB	2.0 GiB
Operating System	Red Hat Linux 7.3	Red Hat Linux 7.3
Kernel	Red Hat 2.4.20-28.7	kernel.org 2.4.24

Apparently, Linux 2.4 kernels put processes to sleep if they dirty "too many" blocks "too fast." The criteria for deciding when to put a heavy writer to sleep are not clear to this author, but with the default bdflush parameters, PVFS and PVFSP *iod* daemons are often identified as heavy writers and put to sleep. Even though the sleeping *iod* is quickly woken again, it is often asleep long enough for all clients to fill their TCP buffers sending to that *iod*, and stall. The result is that under a heavy write load, often only one *iod* at a time is accepting data, which kills performance.

If dirty block write-out is started immediately when blocks are dirtied, *iod* daemons are rarely put to sleep, and write-out proceeds smoothly and simultaneously from all daemons. This is the desired operating state, and gives the highest aggregate write performance.

Thus, on the nodes hosting *iod* daemons, the block write-out tuning was modified using the command *echo "0 500 0 0 500 3000 60 20 0" > /proc/sys/vm/bdflush*, and these values were used for all testing.

Read performance is affected by the tuning parameters /proc/sys/vm/min-readahead and /proc/sys/vm/max-readahead. Although they don't affect the performance of madvise(), they certainly affect the result of the baseline read bandwidth test described below. The value of min-readahead was left at its default value of three, while the value of max-readahead was modified using the command "*echo 255 > /proc/sys/vm/max-readahead*."

In order to gauge the implementation efficiencies of the file systems under test, it is desirable to have a measure of maximum throughput for the disk subsystem. The baseline write and read bandwidth were measured using the parallel distributed shell (*pdsh*) to run *dd* commands on the nodes attached to the DDN controllers, with all ports driven. The baseline write and read performance was measured using these two commands:

```
pdsh 'time { date; dd if=/dev/zero of=/fc/zero.dat bs=16k count=1024k; sync; date; }'
pdsh 'time { date; dd if=/fc/zero.dat of=/dev/null bs=16k count=1024k; date; }'
```

The first command causes each of the *iod* nodes to write a 16 GiB file using a 16 KiB write request, and times both the write command and a sync, to make sure that data isn't written out after the timing has stopped. The second command reads the file back. The commands are bracketed with *date* commands to allow verification that the I/O happened reasonably simultaneously on all hosts. The size of the file was chosen to be 16 GiB since that is eight times the memory on the *iod* nodes, which makes it reasonably unlikely that any data was read from the system page cache, rather than disk.

These tests showed that the DDN controllers and disks were capable of 864 MB/s aggregate write performance, or 108 MB/s per I/O node. Aggregate read performance was 832 MB/s, or 104 MB/s per I/O node.

4.2 Test Matrix

The set of tests performed for this report were chosen to explore the performance boundaries of the PVFS and PVFSP design and implementations. Of the many variables that might impact file system performance, due to time constraints this author chose to investigate the effects of three: file size, number of clients, and client request size. Each of these variables was investigated both for one file per client, which is how most scientific/engineering application codes at Sandia run today, and all clients working on a single file, which is how many Sandia users want their codes to run in the future.

The effect of file size is interesting because it can reveal caching effects, since *piob* writes and then reads back the files its uses. Neither PVFS nor PVFSP client libraries implement caching, but the *iod* daemons will be impacted by virtual memory and caching algorithms implemented in the host kernel.

The effect of the number of clients and client request size are important because they reveal algorithmic and implementation deficiencies. Although each of the 128 client nodes in the test cluster is SMP, only one client process per node was run for all of the tests. This eliminates the risk of mistaking contention for the single network adapter per host with other issues during the client scaling tests.

Three trials of each test were run, in order to provide some idea of the amount of variability in performance. Test results are presented graphically in the next sections, and in tabular form in the Appendix. Standard deviations are found only in the tabular results.

Each of the three series of tests presented below includes the case with 64 clients and 1 MiB client request size. Instances of this test case were run for each test series it appeared in, so the results for it may not agree exactly between test series. Any discrepancies for this case in the results that follow are a further indication of performance variability, and do not indicate an error in data collection or presentation. Furthermore, dedicated time on the cluster was not available for this testing, so it is impossible to estimate the impact of other users on the results presented here.

4.2.1 File Size

For this series of tests, 64 client processes were used to write and read files ranging from 2 GiB to 64 GiB total size. This compares to the 16 GiB of total memory in the eight nodes running the *iod* daemons. When testing with one file per client, the file size per client varied from 32 MiB to 1 GiB. The client data transfer size, i.e., the size of each write and read request, was 1 MiB. Aggregate throughput in MB/s is shown in Figure 3.

This series of tests exposes several aspects of PVFS and PVFSP performance, and how it is affected by Linux kernel implementation choices. The most striking is the large decrease in performance when the file size increases from 8 GiB to 16 GiB. Since the total memory on the eight *iod* nodes is 16 GiB, and some memory must be consumed by the operating system and executables, a file of exactly 16 GiB is just too large to fit completely in the memory available for page caching. Linux uses a least-recently-used (LRU) page replacement algorithm, and *piob* writes files sequentially, so the data for the beginning of the file gets pushed out of the page cache as the end of the file is written. Then, when the file is read, the beginning of the file must be read from disk, and the pages used to hold it are the oldest, which just happen to contain the data that is about to be read next. The result is that the entire file must be read from disk, even though most of it is actually in the page cache.

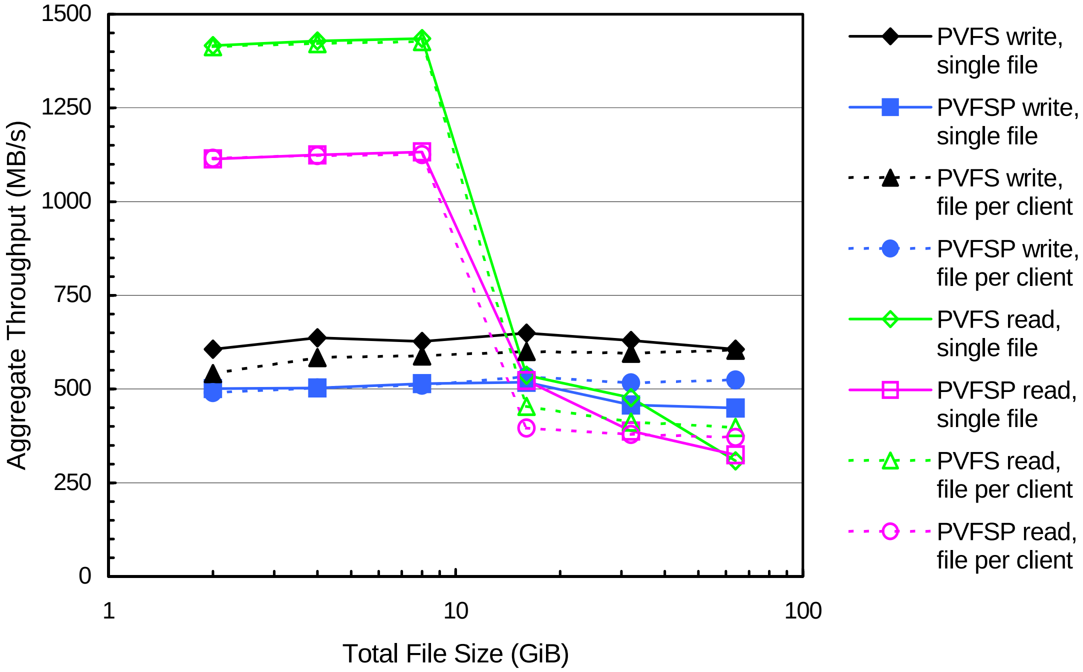


Figure 3 Aggregate throughput for various file sizes.

For a file smaller than 16 GiB, the whole file can fit in the Linux page cache if it has just been written, so no disk accesses are required to read it. Thus, aggregate read bandwidth for files smaller than the total memory available on the *iod* nodes is a measure of the efficiency of the daemon implementation. Since the aggregate read bandwidth for PVFS is ~300 MB/s higher than that for PVFSP, the latter's implementation is less efficient for one or more reasons.

One possibility is the Portals NAL implementation used by PVFSP for this testing. Since this NAL is implemented using the sockets API to TCP/IP, it represents an extra layer of code, not present in PVFS, which must be traversed for every message that is sent or received. Moreover, constraints imposed by the Portals API may have the result that even the most efficient and effective TCP/IP-based NAL cannot use the sockets API as effectively as an application that uses the sockets API natively.

Another possibility for the PVFSP implementation's lower read performance when files are cached on the *iod* nodes may be its use of the mincore() system call to avoid the possibility that read() might block while data is paged into memory. In this case, since the data is always in memory already, the mincore() call is overhead that is not present in PVFS.

A second aspect of performance revealed by these tests is that the aggregate write performance for PVFS is consistently ~100 MB/s higher than that for PVFSP. In addition to the overhead added by the TCP/IP Portals NAL, as discussed above, there is a difference in the interaction between client and server on write requests. For PVFS, after sending the write request on a socket, the client can immediately start sending the data to be written, i.e., PVFS uses a client-push data model. For PVFSP, the client sends the request, and after receiving it the server performs a Portals get operation to retrieve the data, i.e., PVFSP uses a server-pull data model. This adds a minimum of one round-trip time to the latency for each write message.

Another aspect of performance exhibited by both PVFS and PVFSP is their inability to drive the disk subsystem at full bandwidth. Recall from the discussion of the testing baseline that the disk subsystem is capable of 864 MB/s write and 832 MB/s read throughput. Read performance for daemon-cached files shows that both daemon implementations are capable of sourcing data to the network at 1100 MB/s or greater. Although there is no direct evidence that the daemons would be able to sink data from the network at that rate, there isn't an obvious reason why they should not be able to do so. The question remains, why are the daemons unable to drive the disk subsystem at its maximum throughput? A plausible explanation is that neither implementation is fully overlapping disk I/O with request processing.

Finally, for files larger than the total *iod* node memory, there is no clear explanation why both PVFS and PVFSP read performance falls off as file size increases. This is most apparent for the single-file case.

4.2.2 Client Scaling

For this series of tests, the number of clients was varied from 1 to 128. For each number of clients, files of both 2 GiB and 32 GiB total size were created and read. The client transfer size was again 1 MiB. Throughput in MB/s is shown in Figures 4 and 5.

In these tests, PVFS and PVFSP show the same write performance behavior, with performance levels that are roughly equivalent[*]. Throughput starts to fall off with more than four clients, and by sixteen clients the *iod* daemons are essentially saturated. Either a much larger cluster, or much more capable I/O nodes, would be required in order for a clear trend for write performance scaling with the number of clients to be apparent.

As the number of clients increases, PVFS demonstrates better read performance than PVFSP when the data to be read is completely cached on the I/O servers. Moreover, the drop in PVFSP performance from 64 to 128 clients indicates contention for some resource in the *iod* read implementation.

The variation of read performance with the number of clients when data must be read from disk on the I/O servers is confusing for both PVFS and PVFSP. For the single-file case PVFS and PVFSP show essentially identical performance for eight clients or more, with no indication that the *iod* daemons are saturated even for 128 clients. However, PVFSP saturates at eight clients for the one-file-per-client case, while PVFS saturates at 32 clients in this case.

[*] Note that there is a discrepancy of ~150 MB/s between the results presented here and in both the previous and next sections for the PVFS write performance with a single 32 GiB file written by 64 clients. Again, this is not a mistake, but a measure of the performance variability inherent in the implementation and hardware configuration.

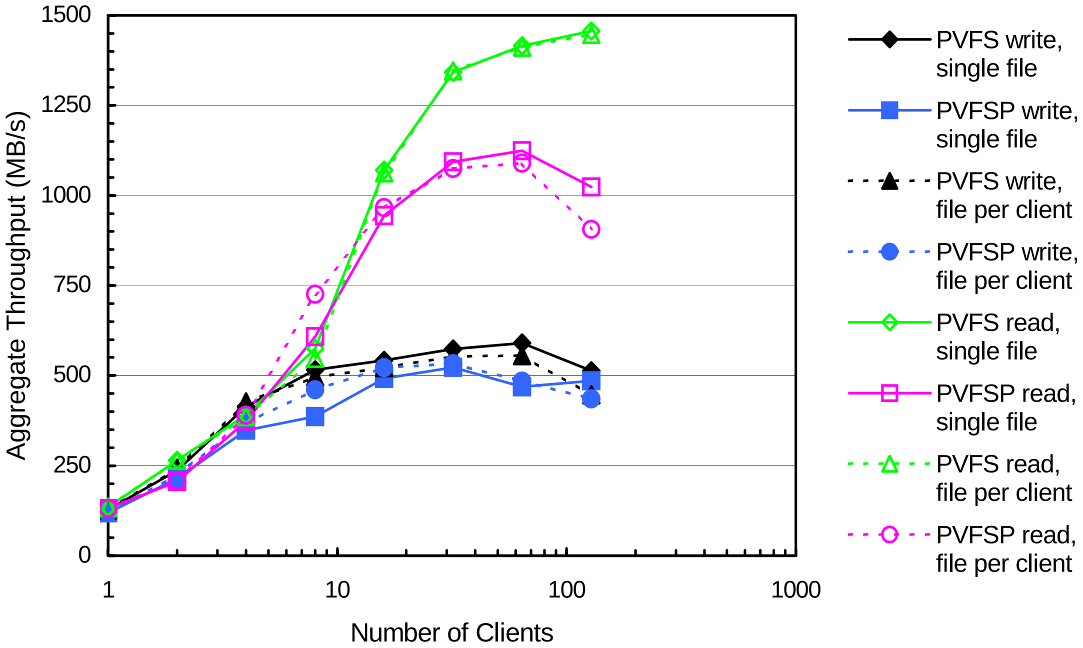


Figure 4 Aggregate throughput for various numbers of clients, using a 2 GiB file.

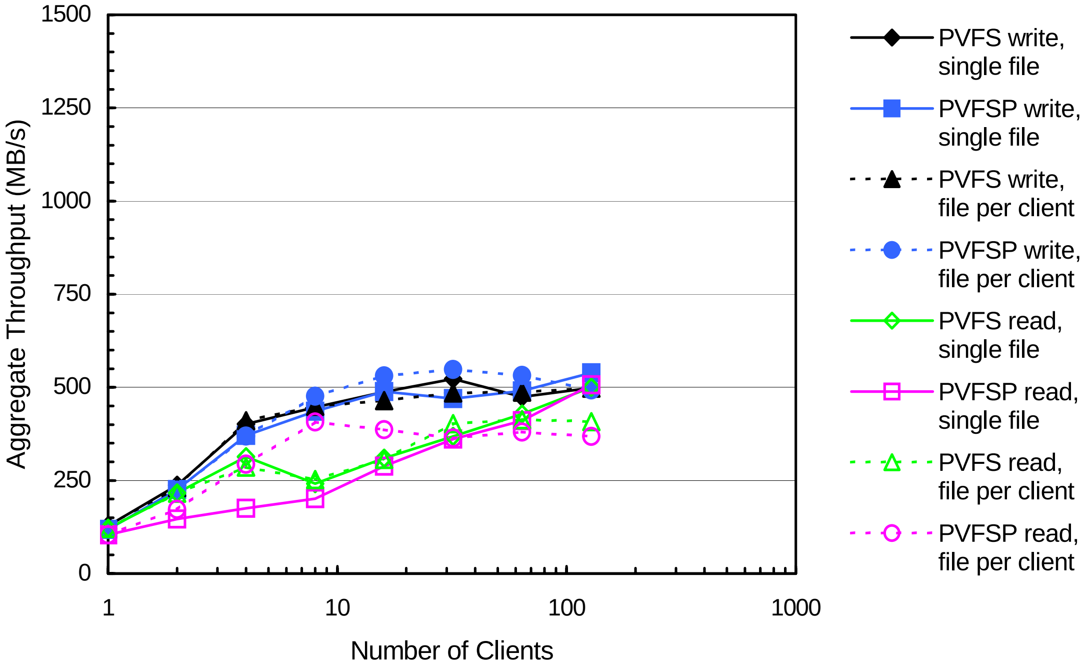


Figure 5 Aggregate throughput for various numbers of clients, using a 32 GiB file.

The two things most apparent from the results of this section are: 1) a cluster of 128 compute and 8 I/O nodes is not large enough for useful I/O scaling testing; and 2) the efficiency of both PVFS and PVFSP *iod* daemons is not good enough. Extrapolating from the results presented here, a cluster with 1024 compute nodes and 64 I/O nodes is the smallest cluster that can present useful scaling results.

4.2.3 Client Request Size

For this series of tests, the client transfer size was varied from 4 KiB to 4 MiB. For each buffer size, files of both 2 GIB and 32 GiB total size were created and read, using 64 client processes in all cases. Throughput in MB/s is shown in Figures 6 and 7.

These tests show the clear superiority of the PVFSP read implementation over that of PVFS. What is puzzling about the PVFS results is the extremely poor performance for small client request sizes, even when the data should be completely cached in memory on the I/O nodes. Whatever mechanism is responsible for this poor performance is likely to also be the cause of poor PVFS read performance with small requests when data must be read from disk by the *iod* daemons.

The interaction of client request size and file system stripe sector size is clearly displayed by the PVFSP read results for the 2 GiB total file size case, where the data is completely cached in memory on the I/O nodes. For request sizes up to 64 KiB, which is the stripe sector size, the data for a single read request always fits into a single reply message from a single *iod* daemon. Thus, there is a fixed amount of overhead per request, and read bandwidth grows nearly linearly with increasing request size.

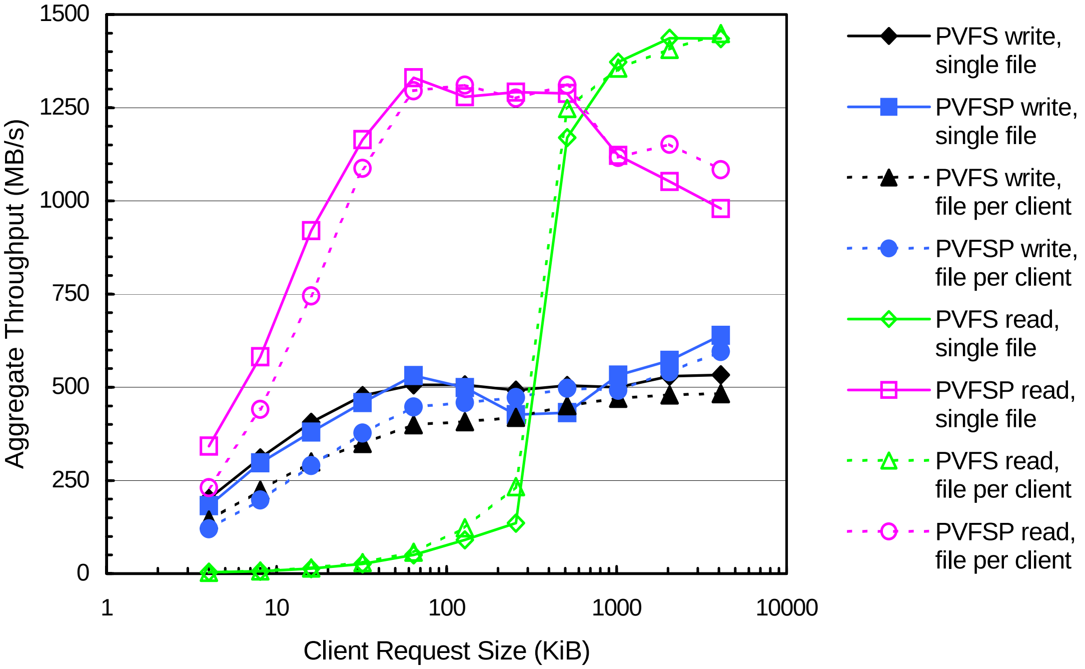


Figure 6 Aggregate throughput for various client request sizes, using a 2 GiB file.

For request sizes between 64 KiB and 512 KiB, the read request is decomposed into a message for one or more *iod* daemons, and each results in a single reply from each daemon. Since these messages are sent to the daemons asynchronously, they can be processed in parallel, and read bandwidth stays constant with increasing request size.

For request sizes larger than 512 KiB, the read request is decomposed into a message for every *iod* daemon, but since the reply data may span multiple stripe sectors on each daemon, there may be multiple replies from each daemon. At this stage, the inefficiencies mentioned previously come into play, and throughput decreases as client request size increases.

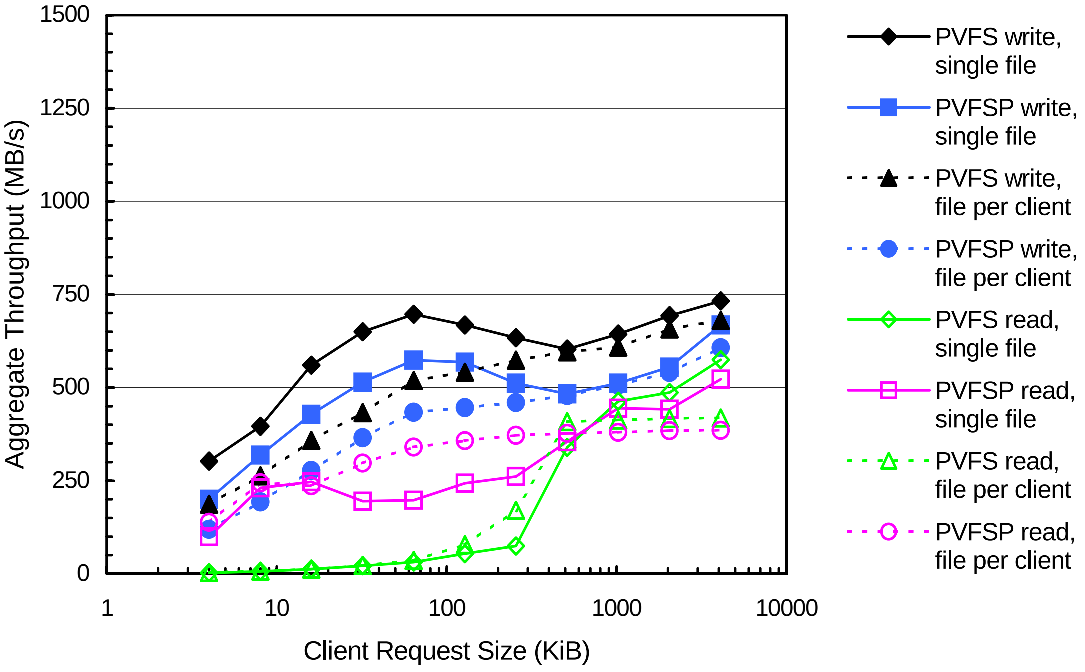


Figure 7 Aggregate throughput for various client request sizes, using a 32 GiB file.

5 Conclusions and Future Work

In general, the test results indicate that PVFS is somewhat more efficient for writing, and for reading when the data is cached in memory on the I/O nodes and the client request size is at least 512 KiB. PVFSP is much more efficient for reading when the client request size is less than 256 KiB. It is the opinion of this author that much of the PVFSP implementation inefficiencies can be attributed to the TCP/IP Portals NAL, and the mismatch between the Portals API and the sockets API.

It should be noted that the NAL used for this testing was not a bypass NAL. However, the NAL implementation does have a compile-time option to enable a separate POSIX progress thread to be started when the NAL is initialized, which makes it an application-bypass (but not OS-bypass) NAL.

In retrospect, it would have been useful to test with the bypass option, to see if there was any performance impact. This path was not explored because early performance testing of the NAL, which was limited to two nodes, indicated that the non-bypass version had better throughput and latency numbers.

Thus, the true performance potential of a Portals-based implementation of PVFS may not have been demonstrated by the testing reported here.

5.1 Future Work

Much work remains to advance the PVFS over Portals implementation from a research vehicle into a production-quality distribution. This section documents some of the required tasks.

Kernel module

The current PVFSP implementation includes only the client library and server daemons. Thus, only those applications that make explicit PVFSP calls can access files stored on a PVFSP file system. For the file system to be useful to users, they must be able to manipulate their files with standard Unix tools such as *ls*, *mv*, and *rm*. This requires a kernel module to integrate PVFSP into the kernel's virtual file system switch (VFS).

As can be seen from Figure 1, integrating PVFS into the Linux kernel VFS was accomplished with two components, a kernel module and a user-space client daemon. The module integrates into kernel data structures and procedures, the client daemon integrates into PVFS network messaging, and they communicate with each other using a third API based on a device file, /dev/pvfsd. Since PVFSP shares its philosophy with PVFS, the PVFS kernel module should work almost, if not completely, unchanged in PVFSP, and only the client daemon would need to be rewritten to employ the PVFSP network messaging. In addition, the client daemon would need added functionality to acquire session authenticators on behalf of the users it is servicing.

Multiple manager daemons

A potentially serious drawback of PVFSP is its single metadata daemon. Data throughput is not affected since, in general, neither a client nor a data daemon need contact the metadata daemon to complete a read or write request[*]. However, for large clusters or applications that make many metadata requests, the inability to service metadata requests in parallel may be a drawback.

It would be relatively straightforward to enhance PVFSP to support multiple, cooperating metadata daemons. The approach would be to assign responsibility for a file or directory to one of several *mgr* daemons based on a hash of the last component in its pathname. When a directory is created or unlinked, the responsible *mgr* would inform all the other *mgr* daemons of the fact. Thus, each *mgr* would maintain the full directory tree so each could do path lookup without communication.

Directory creation or unlinking would be synchronous with respect to updating the other *mgr* daemons, so that when such a call completed the application would know any further metadata operation referencing the affected directory would complete as expected. This is consistent with the

[*] A read response that returns fewer than the number of bytes requested requires that the implementation of *read*() may need to make an internal *fstat*() call to disambiguate end of file from reading into a hole in the file, when the hole occurs at the end of a stripe file. This is also true of PVFS.

semantics offered by the lack of locking provided by PVFSP — the application is responsible for avoiding races.

Unlinking a directory, or listing the contents of a directory, would require that the *mgr* responsible for the directory in question contact all the *mgr* daemons for their contributions to the contents of the directory. This follows directly from the rule that the last component in a path determines the responsible *mgr*. When creating a new directory or file, the *mgr* responsible for the new entry would contact the *mgr* responsible for the parent directory, and update its size and mtime. This eliminates the need for all *mgr* instances to be contacted when responding to a stat() of a directory.

The result of this design is that clients know exactly which *mgr* daemon to contact for any given metadata request. The most common metadata operation, path traversal, does not require communication between metadata daemons. Finally, only a small number of less common operations require that all *mgr* daemons participate.

Fault tolerance

As noted in the Portals overview, this implementation was based on the Portals 3.0 specification, which does not admit the possibility of a transport malfunction. While a draft Portals 3.3 does allow operations to complete in error, this author does not have access to a Portals library based on the newer specification that is accessible from user space. Were such a library available, some additional work would need to be done to the PVFSP implementation to take advantage of error notification. For example, undoing or limiting the effect of a partially completed request needs significant work. Detecting failed server hosts or block devices, and switching over to backup hosts or redundant capability, is completely unaddressed in the current implementation.

Security enhancements

As noted in Section 3.2.2, the current PVFSP implementation assumes deployment on a network that cannot be sniffed, so that authenticators not based on strong encryption can be used. However, the implementation needs to be enhanced to add a finite lifetime to the authenticators, and to add lockout functionality that disables access from a client if it has submitted multiple requests with an invalid authenticator.

In the event PVFSP were to be deployed over a network that was not secure from sniffing, an implementation of authentication based on strong encryption would need to be added.

Performance enhancements

The testing performed for this report provides some clues about how performance might be enhanced, for both PVFS and PVFSP. The biggest clue is provided by the difference between the aggregate read performance when the data is cached in memory on the *iod* hosts, and that when the data must be read from disk. The fact that the former is higher by a factor of two or more suggests that the messaging subsystem is not the performance bottleneck. Furthermore, the fact that the performance level when the *iod* daemons are saturated is so much lower than either the demonstrated messaging performance, or the demonstrated disk subsystem performance, suggests that the PVFSP implementation, and to a lesser extent the PVFS implementation, is not doing a good job of overlapping disk access with request processing and network messaging.

For writes, this is not too surprising, as data is copied from user to kernel space, and a non-trivial amount of bookkeeping is done in the kernel to keep track of dirtied blocks, before the *write*() system call returns. For reads it is a little more surprising, since the *mmap*() call eliminates a data copy from kernel to user space, and great care is taken through the use of *madvise*() and *mincore*() to prevent accesses of the data being read from blocking while the data is paged into memory.

However, it may be possible that more disk I/O can be overlapped with request processing and messaging if the POSIX asynchronous I/O interface was used in the *iod* daemons to move data on and off disk. In addition, the Linux 2.6 kernel may provide significantly better disk performance. Finally, a true bypass implementation of Portals might also increase performance by offloading network data movement out of the kernel.

Execution profiling of the *iod* daemons has not proved fruitful for enhancing performance. However, the current PVFSP implementation can log requests and still run at nearly full speed, so it might prove enlightening to timestamp the request logs with microsecond resolution. For example, some aspect of the implementation may be causing request processing to become synchronized over time, so that periods of intense activity are interleaved with periods of little activity. Some such mechanism may be responsible for the decrease in read throughput as file size is increased.

Finally, it might be useful to actually deploy PVFSP with request logging enabled, so that the logs could be mined to extract the most common request types and sizes under actual use conditions. It has been this author's observation that real data about the distribution of requests from Sandia's large-scale applications is sorely lacking. Knowing how applications actually behave might be a useful guide to the most important aspects of file system implementation.

6 References

Brightwell, R., T. Hudson, R. Riesen, and A. B. Maccabe, 1999. *The Portals 3.0 Message Passing Interface*. SAND99-2959. Sandia National Laboratories, Albuquerque, NM.

Ligon, III, W.B., and R. B. Ross, 2001. PVFS: Parallel Virtual File System, in *Beowulf Cluster Computing with Linux*, Thomas Sterling, ed. pp. 391-430. MIT Press.

Stevens, W. R., 1998. *UNIX Network Programming, Volume 1: Networking APIs; Sockets and XTI* — 2nd ed. Prentice Hall PTR, Upper Saddle River, NJ.

Tomkins, J. L. and W. J. Camp, 2003. *Architectural Requirements for the Red Storm Computing System*. SAND2003-3564. Sandia National Laboratories, Albuquerque, NM.

Appendix

This appendix gives in tabular form the results presented in Section 4. The standard deviation is calculated from the results for three runs of each test.

File Size

Table A1 Aggregate throughput for various file sizes, using a single file, 64 clients, and 1 MiB request size.

	Aggregate Throughput (MB/s)							
	Write				Read			
Total File Size (GiB)	PVFS		PVFSP		PVFS		PVFSP	
	mean	sd	mean	sd	mean	sd	mean	sd
2	606.2	12.4	501.4	23.9	1416.6	3.9	1113.8	7.2
4	637.0	12.2	502.8	9.0	1428.5	2.0	1124.9	16.8
8	627.3	18.6	514.7	13.7	1435.2	4.4	1132.2	5.7
16	649.2	12.8	518.2	14.5	537.2	8.5	522.9	22.7
32	629.6	7.1	457.7	7.6	476.8	7.9	387.7	4.4
64	606.6	3.8	449.7	22.8	308.7	27.9	324.4	7.2

Table A2 Aggregate throughput for various file sizes, using one file per client, 64 clients, and 1 MiB request size

	Aggregate Throughput (MB/s)							
	Write				Read			
Total File Size (GiB)	PVFS		PVFSP		PVFS		PVFSP	
	mean	sd	mean	sd	mean	sd	mean	sd
2	541.2	39.9	490.9	22.6	1413.1	2.7	1115.3	9.7
4	583.9	15.8	502.9	13.0	1421.9	5.4	1123.0	6.1
8	588.9	3.7	509.7	7.8	1427.0	1.8	1125.5	5.9
16	600.0	30.2	533.3	11.7	453.6	4.1	396.3	4.2
32	595.4	28.2	516.5	12.7	412.1	0.2	379.3	4.3
64	603.8	14.7	524.3	7.1	398.1	5.7	370.8	2.5

Client Scaling

Note that for this series of tests, the single-file case and the one-file-per-client case are the same when there is only one client. The one-client case was run only as part of the single-file series, and the results reused as part of the one-file-per-client presentation.

Table A3 Aggregate throughput for various numbers of clients, using a single file, 2 GiB total file size, and 1 MiB request size.

	Aggregate Throughput (MB/s)							
Number of Clients	Write				Read			
	PVFS		PVFSP		PVFS		PVFSP	
	mean	sd	mean	sd	mean	sd	mean	sd
1	127.6	0.2	118.4	0.3	134.3	3.0	130.8	0.4
2	236.5	2.1	214.3	6.9	264.8	5.9	205.9	3.4
4	413.6	16.8	348.2	21.4	390.9	2.0	374.0	4.4
8	516.2	9.7	385.9	12.9	575.2	2.9	608.4	18.8
16	541.7	21.4	491.7	18.7	1069.9	10.2	944.4	3.4
32	573.4	4.6	522.3	4.6	1341.9	16.8	1093.9	16.5
64	590.7	18.9	468.6	10.5	1415.4	7.7	1124.0	8.8
128	513.6	12.3	485.2	9.2	1456.8	1.2	1023.5	3.5

Table A4 Aggregate throughput for various numbers of clients, using one file per client, 2 GiB total file size, and 1 MiB request size.

	Aggregate Throughput (MB/s)							
Number of Clients	Write				Read			
	PVFS		PVFSP		PVFS		PVFSP	
	mean	sd	mean	sd	mean	sd	mean	sd
1	127.6	0.2	118.4	0.3	134.3	3.0	130.8	0.4
2	242.2	0.8	224.9	2.0	264.8	5.2	207.1	1.5
4	426.7	0.8	368.9	5.2	386.2	1.5	390.3	4.3
8	495.4	1.3	460.6	4.9	547.3	31.6	726.2	3.5
16	521.7	10.2	521.8	5.7	1063.0	27.6	966.0	21.4
32	553.3	25.5	532.5	12.1	1345.8	5.3	1075.3	15.2
64	555.3	18.2	484.9	17.6	1411.5	7.1	1089.9	7.3
128	446.1	9.4	435.6	29.1	1446.7	4.1	905.6	9.5

Table A5 Aggregate throughput for various numbers of clients, using a single file, 32 GiB total file size, and 1 MiB request size.

	Aggregate Throughput (MB/s)							
Number of Clients	Write				Read			
	PVFS		PVFSP		PVFS		PVFSP	
	mean	sd	mean	sd	mean	sd	mean	sd
1	128.3	0.2	118.6	0.1	121.7	0.4	104.3	0.3
2	237.4	0.8	225.5	1.1	216.4	1.3	146.4	6.2
4	401.4	2.6	371.2	0.7	313.2	5.5	175.5	2.3
8	446.4	3.8	435.2	20.7	241.5	10.0	201.5	3.3
16	487.7	5.0	488.2	47.9	310.2	8.4	288.1	6.3
32	523.1	35.6	469.5	58.6	368.5	56.0	360.7	26.8
64	473.9	33.2	490.2	14.8	428.8	56.8	410.3	11.9
128	498.3	6.6	538.8	8.5	500.6	4.8	506.8	84.0

Table A6 Aggregate throughput for various numbers of clients, using one file per client, 32 GiB total file size, and 1 MiB request size.

	Aggregate Throughput (MB/s)							
Number of Clients	Write				Read			
	PVFS		PVFSP		PVFS		PVFSP	
	mean	sd	mean	sd	mean	sd	mean	sd
1	128.3	0.2	118.6	0.1	121.7	0.4	104.3	0.3
2	230.9	0.9	223.9	0.8	215.4	0.5	172.2	1.5
4	410.7	1.5	369.1	1.8	285.6	0.9	294.1	4.0
8	449.3	4.3	475.9	2.2	252.4	11.0	406.5	5.8
16	464.8	2.7	531.1	2.2	306.2	13.9	386.2	4.0
32	484.9	3.1	547.9	1.7	402.4	14.1	364.1	2.6
64	486.9	11.1	532.0	12.4	413.6	3.9	380.3	1.9
128	497.9	2.6	492.7	30.7	408.2	6.1	368.2	4.1

Client Buffer Size

Table A7 Aggregate throughput for various client request sizes, using a single file, 64 clients, and 2 GiB total file size.

Client Request Size (KiB)	Aggregate Throughput (MB/s)							
	Write				Read			
	PVFS		PVFSP		PVFS		PVFSP	
	mean	sd	mean	sd	mean	sd	mean	sd
4	201.2	5	181.9	21.9	3.8	0	342.3	0.3
8	310.9	17.1	297.4	21.8	6.6	0	582.1	18.1
16	405.7	9.7	379.5	11.1	13.2	0.1	920.2	23.0
32	477.4	16.7	458.5	7.9	26.1	0.1	1163.9	9.9
64	506.1	5.0	531.1	2.8	50.0	0.5	1329.9	10.9
128	506.5	11.4	499.1	18.8	90.6	1.0	1279.3	70.2
256	491.9	8.6	426.5	68.7	135.6	0.8	1291.4	5.3
512	504.6	9.3	432.1	75.2	1170	15.5	1288.1	19.7
1024	500.2	7.2	532.2	11.7	1372.3	49.0	1121.9	7.2
2048	529.5	9.0	571.6	21.8	1436.3	3.5	1052.4	187
4096	533.2	2.2	639.1	9.5	1435.6	14.2	979.5	160

Table A8 Aggregate throughput for various client request sizes, using one file per client, 64 clients, and 2 GiB total file size.

Client Request Size (KiB)	Aggregate Throughput (MB/s)							
	Write				Read			
	PVFS		PVFSP		PVFS		PVFSP	
	mean	sd	mean	sd	mean	sd	mean	sd
4	143.8	15.4	120.0	7.1	3.8	0.0	231.1	1.1
8	223.8	4.8	197.9	5.3	7.5	0.0	440.4	2.8
16	299.9	6.3	290.0	5.1	14.9	0.1	745.5	5.9
32	348.9	10.2	376.9	4.2	29.4	0.4	1086.7	15.0
64	399.8	12.5	447.4	1.4	57.5	0.1	1295.4	22.4
128	407.4	10.5	458.4	3.2	122.8	1.3	1310.9	8.9
256	419.6	13.6	472.2	0.9	233.3	2.0	1275.6	21.3
512	450.8	6.3	496.9	0.9	1247.8	8.5	1310.8	14.9
1024	470.3	20.6	491.8	19.8	1356.6	18.2	1116.9	2.7
2048	479.3	20.1	541.1	4.5	1406.9	32.6	1151.9	14.8
4096	482.8	4.8	595.7	4.3	1449.0	33.1	1083.9	15.3

Table A9 Aggregate throughput for various client request sizes, using a single file, 64 clients, and 32 GiB total file size.

Client Request Size (KiB)	Aggregate Throughput (MB/s)							
	Write				Read			
	PVFS		PVFSP		PVFS		PVFSP	
	mean	sd	mean	sd	mean	sd	mean	sd
4	302.2	6.7	200.4	2.4	3.0	0.2	99.5	1.5
8	395.7	31.3	318.8	17.7	6.0	0.2	230.8	7.0
16	559.7	12.8	428.3	8.4	12.3	0.1	247.0	22.7
32	649.9	13.2	514.7	9.2	21.5	0.9	195.1	22.0
64	696.6	9.0	573.8	9.8	31.7	1.0	197.8	8.0
128	667.6	5.9	568.1	4.8	53.8	1.4	243.7	9.2
256	633.9	19.6	511.8	34.7	74.8	3.2	261.0	45.1
512	602.8	7.9	483.2	9.4	339.3	9.7	354.6	42.0
1024	643.5	4.1	511.9	13.0	463.3	13.2	444.6	21.6
2048	693.2	18.7	554.8	19.2	486.8	11.9	441.4	14.7
4096	732.8	11.0	667.6	7.2	574.9	17.5	522.3	6.7

Table A10 Aggregate throughput for various client request sizes, using one file per client, 64 clients, and 32 GiB total file size.

Client Request Size (KiB)	Aggregate Throughput (MB/s)							
	Write				Read			
	PVFS		PVFSP		PVFS		PVFSP	
	mean	sd	mean	sd	mean	sd	mean	sd
4	186.4	3.0	119.8	1.6	3.7	0.0	138.0	7.7
8	263.3	4.2	193.3	1.7	7.0	0.0	244.1	12.0
16	357.8	4.8	277.7	5.1	13.0	0.0	235.8	20.1
32	432.4	4.3	365.6	5.7	22.8	0.1	297.0	2.8
64	518.5	10.9	433.7	2.8	36.1	0.2	340.1	4.7
128	541.6	4.6	446.4	2.7	78.5	1.7	357.7	9.6
256	573.7	11.5	459.9	2.2	170.8	0.2	372.0	4.3
512	595.8	1.1	478.2	6.0	408.3	5.2	376.9	5.3
1024	608.7	2.8	506.1	6.0	413.1	1.6	379.8	8.8
2048	657.4	6.3	540.7	1.1	416.4	1.7	384.1	7.5
4096	680.1	2.0	607.0	0.9	418.3	1.8	385.0	9.5

Distribution

1 Los Alamos National Laboratory
Attn: Gary A. Grider
P.O. Box 1663
Los Alamos, NM 87545

1 Lawrence Livermore National Laboratory
Attn: Tyce T. McLarty
P.O. Box 808
Livermore, CA 94551-0808

1 Argonne National Laboratory
Attn: Robert B. Ross
9700 S. Cass Avenue
Argonne, IL 60439

1	MS	0321	R. W. Leland, 9220
1		0801	R.W. Mason, 9320
1		0801	M. R. Sjulin, 9330
1		0806	L. Stans, 9336
1		0807	J. P. Noe, 9328
1		0822	C. J. Palakos, 9326
1		0822	J. E. Sturtevant, 9326
1		0822	D. R. White, 9227
1		0823	J. D. Zepper, 9324
1		1109	S. M. Kelly, 9224
1		1109	J. L. Tomkins, 9220
1		1110	R. A. Klundt, 9223
1		1110	A. B. MacCabe, 9223
1		1110	N. D. Pundit, 9223
1		1110	R. E. Riesen, 9223
1		1110	S. Tideman, 9223
1		1110	H. L. Ward, 9223
1		1110	D. W. Doerfler, 9224

1	MS	9018	Central Technical Files, 8945-1
2		0899	Technical Library, 9616