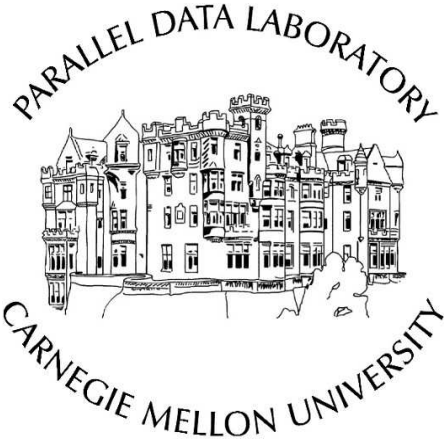


# Data-intensive file systems for Internet services: A rose by any other name ...

Wittawat Tantisiriroj Swapnil Patil Garth Gibson

{wtantisi, swapnil.patil , garth.gibson} @ cs.cmu.edu

CMU-PDL-08-114

October 2008

**Parallel Data Laboratory**
Carnegie Mellon University
Pittsburgh, PA 15213-3890

**Acknowledgements:** We would like to thank several people who made significant contributions in improving this paper. Sam Lang and Rob Ross helped us with all PVFS issues; they graciously answered many questions about PVFS internals, provided pointers for performance debugging, and sent us quick bug fixes and patches to keep us moving forward. Julio Lopez, the Yahoo!'s M45 team and Hadoop team made inroads that helped us evaluate our work on the M45 cluster. Justin Betteridge, Le Zhao, Jamie Callan, Shay Cohen, Noah Smith, U Kang and Christos Faloutsos gave us their scientific applications and data-sets for experimental evaluation. We also thank Dave Andersen and Greg Ganger for valuable feedback about our work. This material is based upon research sponsored in part by the Department of Energy under Award Number DE-FC02-06ER25767, the Petascale Data Storage Institute (PDSI) and by the Los Alamos National Lab under Award Number 54515-001-07, the CMU/LANL IRHPIT initiative. We also thank the members and companies of the PDL Consortium (including APC, Cisco, EMC, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft Research, NetApp, Oracle, Seagate, Symantec, and VMWare) for their interest and support.

**Keywords:** Hadoop, HDFS, PVFS, Internet services, parallel file systems

**Abstract**

*Data-intensive distributed file systems are emerging as a key component of large scale Internet services and cloud computing platforms. They are designed from the ground up and are tuned for specific application workloads. Leading examples, such as the Google File System, Hadoop distributed file system (HDFS) and Amazon S3, are defining this new purpose-built paradigm. It is tempting to classify file systems for large clusters into two disjoint categories, those for Internet services and those for high performance computing.*

*In this paper we compare and contrast parallel file systems, developed for high performance computing, and data-intensive distributed file systems, developed for Internet services. Using PVFS as a representative for parallel file systems and HDFS as a representative for Internet services file systems, we configure a parallel file system into a data-intensive Internet services stack, Hadoop, and test performance with microbenchmarks and macrobenchmarks running on a 4,000 core Internet services cluster, Yahoo!'s M45.*

*Once a number of configuration issues such as stripe unit sizes and application buffering sizes are dealt with, issues of replication, data layout and data-guided function shipping are found to be different, but supportable in parallel file systems. Performance of Hadoop applications storing data in an appropriately configured PVFS are comparable to those using a purpose built HDFS.*

# 1 Introduction

*"What's in a name? that which we call a rose*
*By any other name would smell as sweet."*

**– Shakespeare in 'Romeo and Juliet' (II, ii, 1-2)**

## 1.1 Motivation

Recent trends suggest a rapid growth in the use of large server infrastructure like data centers and cloud computing platforms for scalable services [3, 35, 15]. Internet services, like Google and Yahoo!, are using their computing infrastructure to run applications that compute on massive amounts of input data. And in order to finish running these applications fast, computations are distributed over a large cluster. With the success of such *data-intensive applications*, Internet services are being explored for more diverse applications compelled by their parallel programming models, manageability and reliability [7, 8, 11].

At the core of data-intensive applications is a distributed file system also running on the large server cluster. Leading Internet services have designed and implemented file systems "from-scratch" to provide high performance for their anticipated application workloads and usage scenarios. Leading examples of such *Internet services file systems*, as we will call them, include the Google file system (GoogleFS) [17], Amazon Simple Storage Service (S3) [4] and the open-source Hadoop distribute file system (HDFS) [23].

Another style of computing at a comparable scale and with a growing market place [24] is high performance computing (HPC). Like Internet applications, HPC applications are often data-intensive and run in parallel on large clusters (supercomputers). These applications use *parallel file systems* for highly scalable and concurrent storage I/O. Examples of parallel file systems include IBM's GPFS [30], Sun's LustreFS [26], and the open source Parallel Virtual file system (PVFS) [28].

It is commonly believed that HPC systems use specialized infrastructure, that their file systems are designed for vastly different data access patterns, and that they cannot support Internet services workloads efficiently. Most oft-cited HPC examples assume the use of low latency networks for distributed shared memory computations of a physical phenomena. In fact, many HPC clusters use commodity compute, storage and network infrastructure. Furthermore, an increasing number of scientific applications are being programmed to be like Internet services applications. The goal of this paper is to compare Internet services file systems and parallel file systems, specifically *can we use modern parallel file systems in the place of custom Internet services file systems?*

Several factors motivate the use of existing parallel file systems for Internet services. First, most parallel file systems are designed to efficiently handle a wide variety of workloads, especially concurrent file access, while Internet services file systems are not. Second, they offer different performance versus reliability tradeoff, such as RAID-5, mirroring and failover pairs. Finally, parallel file systems are maturing; soon the venerable NFS, through a standardization effort known as parallel NFS (pNFS) [18], will become a parallel file system.

In this paper, we compare and contrast the Parallel Virtual File System (PVFS), a representative for parallel file systems, and the Hadoop Distributed File System (HDFS), a representative for Internet services file systems. We built a non-intrusive shim layer to plug PVFS in to the open-source Hadoop Internet services stack [20]. This enables Hadoop applications to use PVFS for persistent storage without any changes to the PVFS source.

Our shim layer enables PVFS to offer the same benefits that HDFS offers to the Hadoop data-processing framework through three key features:

- Exposing data layout for function shipping – HDFS is optimized for applications that process massive amount of data using the Hadoop/Mapreduce abstraction [14, 20]. A goal of this abstraction is to minimize the transfer of large amounts of input data by shipping computation to nodes that store the input data. The Hadoop framework achieves this collocation using file data layout exposed by HDFS. PVFS also maintains file layout information. Our shim layer extracts layout maps from PVFS to the Hadoop framework.

- Readahead buffering – To avoid the overhead of synchronous small reads, HDFS clients prefetch large amount of data and pipeline the data transfer in smaller units. Because clients in PVFS are stateless and do not cache data, all requests for data are synchronously sent to the server, irrespective of the amount requested. Our shim layer implements a readahead buffer that enables PVFS to match the transfer efficiency of HDFS.

- Replication for fault tolerance – HDFS provides high availability by storing three copies (by default) of a file. It uses a "rack-aware" replication policy to ensure data availability in face of machine and rack failures. PVFS relies on storage hardware to provide fault tolerance. Our shim layer emulates HDFS-like replication by writing all data, on behalf of the PVFS clients, to three different servers.

We evaluate the performance of PVFS and HDFS by running microbenchmarks and macrobenchmarks, comprised of a suite of four data-intensive applications, on the 4,000 core Yahoo! M45 cluster. Our experiments demonstrate that PVFS performs at least as good as HDFS for most workloads including data-intensive Hadoop applications that benefit from the data layout. The major exception to this is `sort`, which is a write-intensive workload. In such workloads, HDFS writes one copy unstriped locally and two striped widely while our unmodified PVFS writes all three remotely. With limited network bandwidth this can cause a 2:3 ratio in completion time. Moreover, PVFS outperforms HDFS for workloads doing concurrent writes to the same file because HDFS does not support concurrent writes. For instance, a "parallel" file copy operation using PVFS is more than four times faster than HDFS on 16 nodes.

In the next section, we present an overview of real-world parallel file systems and Internet services file systems. In §3, we discuss the design and implementation details of PVFS and HDFS. §4 describes our implementation and cluster configuration in the context of the Hadoop Internet services stack. Experimental results from microbenchmarks and macrobenchmarks are described in §5 and §6. Finally we summarize our contributions and opportunities for future work in §7.

## 2 Real, Large-scale Storage Systems

Over the last decade, the use of large compute clusters has fueled the resurgence of scalable file systems. We present a brief overview of parallel file systems and Internet services file systems that are used in real-world clusters.

### 2.1 Parallel file systems

Most users want a Unix file system that can scale to handle very large files, that delivers very fast aggregate throughput, increasingly high rates of concurrent writing and metadata operations, and very large numbers of files. The leading parallel file systems all plug into the VFS interface

and strive for Unix/POSIX semantics although the widespread deployment of NFS, which has very weak semantics for concurrent access to shared files, has taught users to tolerate some divergence from exact emulation of historical single-CPU UNIX file system.

There are two classes of parallel file systems, shared-disk file systems and object-based file systems. *Shared-disk file systems* use a pool of block-level storage that is shared in a scalable manner and is distributed across many machines under common administration. These file systems allow all clients to access the shared storage directly without going through an intermediate server. Some examples include IBM's General Purpose file system (GPFS) [30], Redhat Global File System (GFS) [16, 32], and SGI's Cluster XFS [12]. *Object-based file systems* use separate servers to store the data and metadata; data servers store the actual file data and metadata servers store all file metadata like directories, access permissions and timestamps. To access a file, clients contact the metadata server to find which data servers it should contact and then directly accesses the file on these data servers. Sun's Lustre file system [26], Panasas's PanFS [34] and PVFS [28] are leading examples of object-based file systems.

In the near future, NFSv4.1 (or pNFS) will offer parallel file system capability, supporting variants of both classes [18]. A key feature being added to NFSv4.1 is layout delegation, a description of the mapping of files to storage (blocks, objects or subfiles) that NFS servers can lease to clients to allow direct access [25]. The possibility that these layout delegations could be used by applications to make scheduling decisions, as is done in MapReduce on Internet service file systems, inspired us to ask how well a parallel file system might perform as an Internet services file system.

### 2.2 Internet services file systems

Google's software and the closely related open-source Hadoop software are revolutionizing the Internet services community by building scalable systems infrastructure for data intensive applications. A key component of these systems is their distributed file system, called Google file system (GoogleFS) or Hadoop distributed file system (HDFS), that is built from scratch to provide high availability in face of component failures and to deliver a high performance for data-intensive applications [17, 23]. In particular, the file systems provide the right abstraction for the Mapreduce data processing framework that divides a computation into many tasks which are scheduled to run on many nodes in a manner that minimizes transferring massive amounts of input data-sets [14].

GoogleFS attributes that its basic architecture closely resembles the CMU Network attached secure disks (NASD) design, which also inspired the design of many modern parallel file systems, particularly object-based systems (§2.1). Files in GoogleFS are split into 64MB chunks that are placed on different chunkservers. All the filesystem metadata, including chunk layout, is stored on a master server. Clients contact the master to get the chunk layout information of a file and then perform all file mutations directly on the chunks stored on the chunkservers. GoogleFS is built to deliver high performance for applications that mutate files through appends. It uses "atomic record appends" and relaxed consistency semantics to enable concurrent appends to a single file. The other key feature of GoogleFS is that it uses chunk replication to detect, tolerate and recover from component failures in the cluster. It uses techniques for replica consistency, automatic re-replication and inexpensive snapshots. More details about the GoogleFS internals can be found in the 2003 paper by Ghemawat et. al. [17].

The Hadoop project, led by Yahoo! and the Apache Foundation, offers an open source implementation of Google's Mapreduce model [20]. This framework is widely used to experiment and run large-scale computing services; even, Google is supporting it in its joint project with the National Science Foundation (NSF) and IBM to promote research and development in Internet-scale computing [27]. The file system support for Hadoop is provided by the Hadoop distributed file

| | **Hadoop Distributed file system (HDFS)** [23] | **Parallel Virtual file system (PVFS)** [28] |
|---|---|---|
| **Deployment model** | Compute and storage on one node (beneficial to Hadoop/Mapreduce model where computation is moved closer to the data) | Separate compute and storage nodes (easy manageability and incremental growth); also used by Amazon S3 [4] |
| **Concurrent writes** | Not supported – allows only one writer at time | Guarantees POSIX sequential consistency for non-conflicting writes, i.e. optimized writes to different regions of a file |
| **Small file operations** | Not optimized for small files; but client-side buffering will only send a write to a server once | Lack of client-side caching incurs high I/O overhead for small file ops (beta version include small file packing optimizations) |
| **Append mode** | Write once semantics that does not allow file rewrites or appends | Not supported yet, but can modify it to use "fetch-add" semantics by writing to the end-of-file offset that is stored as a file's extended attribute |
| **Buffering** | "Data staging" to buffer writes until the amount of data is worth the chunk size (64MB) | No client-side caching |
| **Data layout** | Exposes mapping of chunks to datanodes to Hadoop applications | PVFS maintains stripe layout information as extended attributes (pNFS has support for layout delegations) |
| **Replication** | 3 replicas of data using rack-aware replica placement policy | No file system level support, redundancy through RAID on/across storage devices (modified PVFS to emulate replication) |
| **Compatibility** | API designed for the requirements of data-intensive applications | Supports UNIX I/O interface and (most) POSIX semantics |

**Table 1. Comparing various facets of PVFS and HDFS design.**

system (HDFS) [23]. While HDFS draws inspiration from GoogleFS, it has significantly different file access semantics, especially for concurrent access and file mutations (details discussed in §3).

Amazon S3 is a simple file system where files are written only once and are accessed using the web services API like REST and SOAP [4]. These files are stored in a flat space using a key-value abstraction, where the filename is the key and the actual file content is the value. Amazon objects are similar to NASD's definition of objects, but more restricted in the way they are written. They are perhaps closer to EMC's Centera for digital information regulation compliance [9].

# 3 HDFS and PVFS – A Design Primer

Both HDFS and PVFS are open-source file systems that are widely used in real production clusters and are well-supported by an active developer base. HDFS is the core storage subsystem for the Hadoop Internet services stack. HDFS is deployed on production systems in Internet services such as Yahoo! and Facebook, and on experimental systems such as the 4000-core Yahoo! M45 cluster [35]. PVFS is a user-level parallel file system that is built for high performance access to large data sets. Currently PVFS is used at numerous HPC sites such as Argonne National Lab, Ohio Supercomputer Center and Sandia National Labs. Both file systems are deployed as user-level

services without any changes to the underlying local file system or other kernel components; this makes it feasible to experiment with HDFS and PVFS configurations in a production Internet services platform.

HDFS and PVFS have a similar high-level design – they store file data and file metadata on two different types of servers. All the file system metadata is stored on a *metadata server* ("namenode" in HDFS). This includes the namespace, data location and access permissions. All metadata operations may be handled by a single metadata server, but a cluster will configure multiple metadata servers as primary-backup failover pairs. *Data servers* are responsible for storing all file data on persistent storage. In PVFS, a data server is called a "I/O server", and HDFS calls it a "datanode". Any installation will have many data servers, which can run on any node, although HDFS typically uses compute client machines also as data servers and PVFS typically does not.

Files are divided into pieces, called chunks in HDFS and stripe units in PVFS, that are stored on different data servers. Both chunk size and stripe unit size are configurable parameters used by metadata servers to distribute a file. By default, HDFS uses 64MB chunks and PVFS uses 64KB stripe units, although in practice PVFS systems use a larger stripe unit (1-4 MB) to achieve higher data transfer bandwidth.

To perform a data access operation, a client first asks the metadata server which data servers it should contact. Clients then send their read and write requests directly to the data servers. data I/O operations are never performed through the metadata server. Clients cache the data server layout information it receives from a metadata server to interact repeatedly and directly with the data servers.

We now contrast the design and implementation of HDFS and PVFS (summarized in Table 1). Occasionally, we also reference unique characteristics of other parallel file systems and Internet services file systems.

### 3.1 Storage deployment architecture

The storage and compute capabilities of a cluster are organized in two ways. A cluster can either collocate storage and compute on the same node or separate storage nodes from compute nodes. Both HDFS and GoogleFS use the former approach with 2-6 disks attached locally to a server with a commodity processor and few gigabytes of RAM [13]. This "disk-per-node" model is well suited for the Hadoop/Mapreduce data processing abstraction that seeks to collocate a compute task on a node that stored the input data for that task. This model is an attractive cost-effective high bandwidth approach for a cluster that desires private storage on each node.

PVFS typically uses the other approach of keeping storage servers separate from the compute infrastructure. This model pools storage nodes to enable highly parallel storage I/O and is widely used in high performance computing clusters [19]. Separating storage nodes also helps build optimized reliability and manageability solutions [34]. Compared to the disk-per-node model, this approach has marginally higher capital costs but it comes at a much lower operational complexity, which is becoming a growing bane for users of large computing infrastructure.

In fact, this model is also adopted by some Internet services, most notably Amazon's web services platforms. In Amazon's compute cloud infrastructure the compute instances (called EC2) are separate from the underlying storage infrastructure (called EBS and S3) [2, 4, 3]. This allows the EBS storage system to provide features for high availability, seamless data volume migration and data backups without being dependent on EC2.

### 3.2 File access semantics

The semantics of most Internet services file systems, including HDFS, GoogleFS and Amazon S3, are optimized for their anticipated workloads. Files in HDFS have write-once-read-many (WoRm) semantics and strictly have one writer at any time. HDFS does not allow changes to a file once it is created, written, and closed. These semantics favor Hadoop/MapReduce applications that typically read and process data from a huge file. The lack of support for concurrent write sharing simplifies the data consistency semantics of HDFS. WoRm semantics have been previously seen in the Bullet file server [33], in archival file systems like Venti [29] and more recently in Amazon S3 [5]. Amazon S3 provides a key-value abstraction that uses the filename as a key and stores file contents as a value. If a key already exists, S3 will overwrite the existing value of that key with a new one. In contrast, GoogleFS is optimized for mutating files by appending the data to the file [17]. Instead of supporting random writes, GoogleFS provides relaxed consistency semantics and an "atomic" append operation to support concurrent appends to a single file. Current version of HDFS does not support appending writes to files; there are plans to add it in a future release.

Most parallel file systems support a wide variety of file operations, especially for highly concurrent file access. PVFS provides high throughput for large file accesses using "non-conflicting write" semantics. A write operation is non-conflicting with another write if the two operations are writing to non-overlapping regions. (However, interleaved write requests can still be non-conflicting.) If two clients are performing concurrent, non-conflicting write operations to a file, then all other clients will see the data from the writers after their respective writes have completed. But if two clients concurrently write data to the same region of a file (conflicting write), the result is undefined. Thus PVFS adheres to the POSIX consistency semantics by guaranteeing sequential consistency in case of non-conflicting writes.

Like HDFS, the current PVFS release does not support appending writes to a file. Appends can be implemented in PVFS using a "fetch-and-add" style operation using extended attributes. PVFS supports extended attributes for applications that require the file system to provide custom metadata types. For appends, the End-of-File (EoF) offset of a file can be stored as an extended attribute. PVFS clients can "fetch" the EoF offset and "add" data starting at that offset [1].

### 3.3 Huge file operations

HDFS architecture and semantics enable writes for high throughput (instead of low latency) batch processing [23]. It enables streaming writes through "write staging" at the clients. Clients send writes to a datanode only when they have data worth the HDFS chunk size (64MB). Initially, clients buffer all write operations by redirecting them to a temporary file in memory. Once filled, the clients flush their buffer to the datanode responsible for storing that file chunk. If the file is closed when the buffer is not full, clients are forced to flush the buffer to the chunk's respective datanode.

On the other hand, PVFS does not have any client-side caching and sends all its writes directly to a I/O server. This approach is not optimal for small write workloads, however, small file I/O continues to remain uncommon among data-intensive applications. Such applications do not benefit from caching because they operate on large files that are too big to fit in memory. Moreover, by not performing any client-side caching, PVFS does not require complex cache consistency protocols, especially on file `close()`, for concurrent write operations to the same file. Typically in large systems that support concurrent writes to a file, cache consistency and synchronization mechanisms often become a potential source of bottleneck. Since HDFS allows only one writer per file, its client-side write staging does not encounter any chunk inconsistency issues. PVFS does not use

any caching on the I/O servers either; it relies on the buffer cache of the underlying file system on the server.

### 3.4 Data layout and function shipping

Both HDFS and PVFS distribute chunks of a file on to different nodes. As we show in our evaluation, the choice of data distribution policy can have a significant impact on the applications running on Hadoop/Mapreduce.

HDFS uses a random chunk layout policy to map chunks of a file on to different datanodes. At file create time, the namenode randomly selects a datanode to store a chunk. This random chunk selection may often lead to sub-optimal file layout that is not uniformly load balanced. The namenode is responsible to maintain the chunk to datanode mapping which is used by clients to access the desired chunk.

In particular, HDFS exposes this layout information to the Hadoop/Mapreduce applications. Hadoop uses this chunk layout to determine how to schedule computation tasks to datanodes. Each task is assigned to a chunk and Hadoop tries to assign a task to a datanode that holds the appropriate input chunk. In fact, the job scheduler assigns tasks to nodes in a manner that achieves load balancing across all the nodes, and makes up for the non-uniform chunk placement due to HDFS's random chunk placement policy. The HDFS file system API provides a function call (`getFileBlockLocations()`) that provides a layout of each chunk using a array of chunk location information (the server's hostname), offset of the first byte of the chunk and the current size of the chunk.

Most parallel file systems also stripe the data across multiple servers using various policies. However, by default, PVFS uses a round robin policy to stripe a file across multiple I/O servers uniformly to achieve load balanced distribution. This uniform distribution enables PVFS to support highly parallel operations. PVFS stores a file's stripe distribution information as an extended attribute of that file. But unlike HDFS, it does not expose this layout to the Hadoop application by default. We implemented a mechanism that queries PVFS for this layout information and exposes to the Hadoop/Mapreduce framework (details in §4.1).

Current version of parallel NFS (pNFS or NFS v 4.1) has in-built support to expose data layout information [25]. The key feature being added to pNFS is a layout delegation that provides a description of the mapping of files to storage (blocks, objects or subfiles). The NFS servers can lease this mapping to the clients to allow direct access to the files. The possibility that these layout delegations could be used by applications to make scheduling decisions, as done in Hadoop/Mapreduce on Internet services file systems, would help in improving the current pNFS design.

### 3.5 Handling failures through replication

Failures are common in large clusters and any large system must detect, tolerate and recover from failures [31, 17]. HDFS uses replication to maintain at least three copies (one primary and two replicas) of every chunk. Applications that require more copies can specify a higher replication factor typically at file create time. All copies of a chunk are stored on different datanodes using a "rack-aware" replica placement policy. The first copy is always written to the local storage of a datanode to lighten the load on the network. To handle machine failures, the second copy is distributed at random on different datanodes on the same rack as the datanode that stored the first copy. This improves network bandwidth utilization because inter-rack communication is faster than cross-rack communication which often goes through intermediate network switches. To

maximize data availability in case of a rack failure, HDFS stores a third copy distributed at random on datanodes in a different rack.

In HDFS, datanodes do the work of replicating the chunk. HDFS uses "replication pipelining" where a datanode that receives a chunk sends the chunk to the datanode that stores the next copy [23]. The list of datanodes that will store copies of any chunk are determined and maintained by the namenode.

Most parallel file systems, including PVFS, rely on hardware based reliability solutions like per I/O server RAID inside logical storage devices and RAID 0 across nodes. Some exceptions include Panasas's PanFS that uses RAID-5 across nodes [34] and Sun's LustreFS that configures its metadata servers as active-active failover pairs [26]. We modify PVFS to emulate the HDFS-style three replicas by forcing the client to propogate each write to three data servers (described in §4.1).

### 3.6 Usability and API

HDFS is explicitly designed for applications that operate on large input data sets and that are built to use the write-once semantics. This is similar to GoogleFS where applications are expected to mutate files by appending rather than writing at random offsets [17]. This philosophy of enforcing "rules" for application writers is perhaps easy to adopt in an "enterprise" setup but it will fail to capture a wider audience.

Most parallel file systems, including PVFS, offer standard UNIX I/O interface and most POSIX semantics. This eases the adoption of parallel file systems without any significant changes to the applications. PVFS diverges from POSIX semantics only in the case of conflicting writes described earlier in §3.2.

## 4 Experimental Evaluation

In this section, we describe our modifications to the Hadoop Internet services stack to plug in PVFS. We also describe the configuration of two clusters that we used to run microbenchmarks and macrobenchmarks.

### 4.1 Prototype Implementation

Figure 1 shows our modifications to the Hadoop Internet services stack that allow data-intensive Hadoop applications to use PVFS instead of HDFS. We implemented a *shim layer* that uses Hadoop's extensible abstract file system API (`org.apache.hadoop.fs.FileSystem`) to use PVFS for all file I/O operations. Prior systems like the Kosmos filesystem (KFS) [10] and Amazon S3 [22] have used this file system API to build backend stores for Hadoop applications.

Hadoop directs all file system operations to the shim layer that forwards each request to the PVFS user-level library. Our implementation does not make any code changes to PVFS other than one configuration change, increasing the default 64KB stripe size to match the HDFS chunk size of 64MB, during PVFS setup. The shim layer is implemented using the Java Native Interface (JNI) API to allow the Java-based Hadoop applications make calls to the C-based PVFS library. Not counting comments, our shim layer implementation consists of less than 3,000 line of Java code. The shim layer has three key components that are used by Hadoop applications.

*Readahead buffering* – While applications can be programmed to request data in any size, the Hadoop framework uses 4KB as the default amount of data accessed in each file system call. Instead of performing such small reads, HDFS prefetches the entire chunk (of default size 64MB)

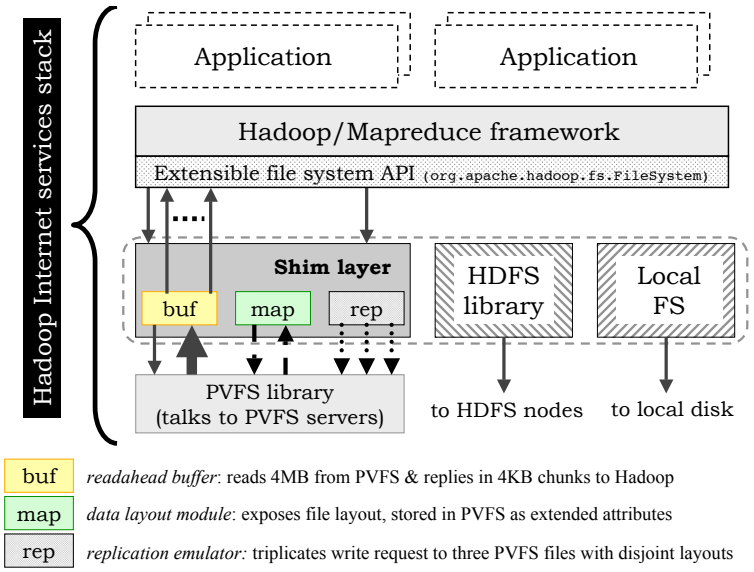


**Figure 1. Hadoop-PVFS Shim Layer** *The shim layer allows Hadoop to use PVFS in place of HDFS. This layer has three responsibilities: to perform read-ahead buffering, to expose the data layout to Hadoop applications and to emulate replication.*

---

asynchronously and then synchronously reads a buffer at a time from the result of the chunk prefetch. This "read ahead" mechanism is important for reducing file system overhead. We modified the shim to provide similar buffering for PVFS. For every 4KB request made by Hadoop, the shim layer synchronously reads a larger buffer (which is a configurable parameter) from PVFS but without the asynchronous prefetch of the whole chunk.

*Data layout module* – The Hadoop/Mapreduce job scheduler distributes computation tasks across many nodes in the cluster. Although not mandatory, it prefers to assign tasks to those nodes that store input data required for that task. This requires the Hadoop job scheduler to be aware of the file's layout information. Fortunately, as a parallel file system, PVFS has this information at the client, and exposes the file striping layout as an extended attribute of each file. Our shim layer matches the HDFS API for the data layout by querying the appropriate extended attributes as needed.

*Replication emulator* – Although the public release of PVFS does not support triplication, our shim enables PVFS to emulate HDFS-style replication by writing, on behalf of the client, to three data servers with every application write. Note that it is the client that sends the three write requests to different servers, unlike HDFS which uses pipelining among its servers. Our approach was motivated by the simplicity of emulating replication at the client instead of making non-trivial changes to the PVFS server implementation. Planned work in PVFS project includes support for replication techniques [1].

| | HDFS | vanilla PVFS | PVFS with shim layer | |
|---|---|---|---|---|
| **Features** (in HDFS and PVFS shim) | | | | |
| Readahead buffer (4MB) | **yes** | no | **yes** | **yes** |
| File layout information | **yes** | no | no | **yes** |
| `grep` **performance** (over a 64GB data-set on 32 nodes) | | | | |
| Read throughput (MB/s) | 579.4 | 244.9 | 358.5 | 597.1 |
| Read network traffic (GB) | 3.71 | 70.1 | 67.7 | 7.2 |
| Write network traffic (GB) | 2.68 | 70.3 | 68.0 | 7.1 |
| Avg CPU utilization | 43% | 27% | 27% | 43% |
| Completion time (min:sec) | 1:45 | 4:08 | 2:57 | 1:46 |

**Figure 2. Benefits of using readahead buffering and layout information** – *By using both these techniques, PVFS with its shim layer matches the performance of HDFS for the read-intensive Hadoop application (`grep`) executed on a 32-node configuration.*

### 4.2 Experimental Setup

We performed our experiments on two clusters: a small cluster for microbenchmarks and a big cluster for running real applications. Our small cluster, called the *SS cluster*, consists of 20 nodes, each containing a dual-core 3GHz Pentium D processor, 4GB of memory, and one 7200 rpm SATA 180 GB Seagate Barracuda disk with 8MB buffer DRAM size. Nodes are directly connected to a HP Procurve 2848 using Gigabit Ethernet backplane and have 100 $\mu$second node to node latency. All machines run the Linux 2.6.24.2 kernel (Debian release) and use the ext3 file system to manage its disk.

For large scale testing, we use the Yahoo! *M45 cluster*, a 4000-core cluster used to experiment with ideas in data-intensive scalable computing [35]. It makes available about 400 nodes, of which we typically use about 50-100 at a time, each containing two quad-core 1.86GHz Xeon processors, 6GB of memory, and four 7200 rpm SATA 750 GB Seagate Barracuda ES disk with 8MB buffer DRAM size. Because of the configuration of these nodes, only one disk is used for a PVFS I/O server. Nodes are interconnected using a Gigabit Ethernet switch hierarchy. All machines run the Redhat Enterprise Linux Server OS (release 5.1) with the 2.6.18-53.1.13.el5 kernel and use the ext3 file system to manage its disks.

Our experiments were performed on the Hadoop stack version 0.17.1 (that includes HDFS), PVFS version 2.7.1 and Sun Java SE Runtime Environment version 1.6.0. Both HDFS and PVFS were configured to use a single dedicated metadata server and data server on each compute client.

### 4.3 Baseline performance

To understand the baseline performance of PVFS and HDFS, we use `grep` on a large data set using the Hadoop/Mapreduce framework. In this experiment, 32 clients run the `grep` program to search for a pattern on an input data-set comprised of 32 files, each consisting of two billion 100-byte records and striped over 32 data servers. Note that `grep` is a highly read-intensive benchmark that reads the entire data-set to search for the desired pattern. We configured Hadoop to use both HDFS and vanilla PVFS.

Figure 2 shows that vanilla PVFS, without the shim layer, is more than a factor of two

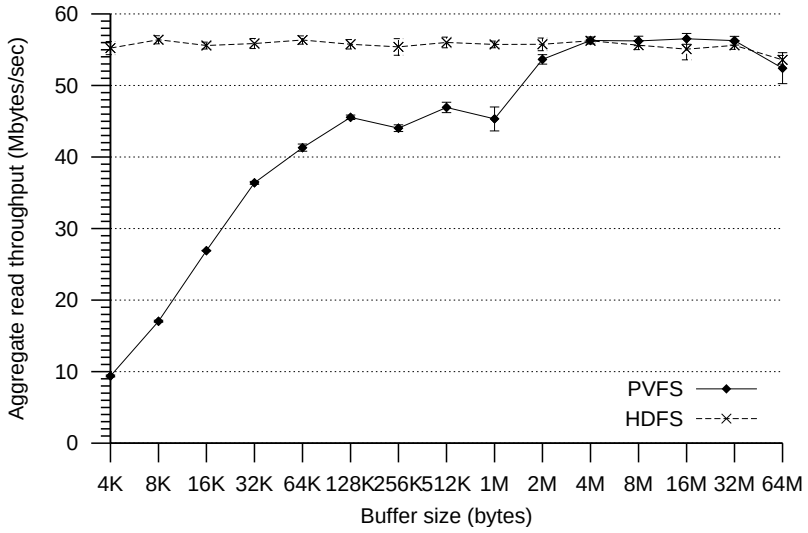


**Figure 3. Single client read performance with varying readahead buffer** – *By using a readahead buffer, the PVFS shim delivers a read throughput comparable to HDFS's in-built prefetching mechanism that reads the entire 64MB chunk for all requests. For the rest of the evaluation, PVFS uses a 4MB readahead buffer and HDFS uses a 128KB buffer (as per the Hadoop setup guide [21]).*

---

slower than HDFS. Hadoop-on-HDFS benefits significantly from the readahead buffering and the file layout information. Vanilla PVFS does not have either of these features; we next perform the same experiment using the PVFS shim layer that does readahead and exposes the file layout information. Figure 2 shows that using a 4MB readahead buffer alone enables PVFS to improve the application's completion time by 25%. And by doing both readahead and exposing PVFS's file layout information, Hadoop-on-PVFS matches the performance of Hadoop-on-HDFS. In the rest of the evaluation, all PVFS I/O operations use the shim layer.

The size of the readahead buffer in the PVFS shim is an important design parameter. To choose the appropriate buffer size, we performed an experiment where a single client sequentially reads a 1GB file, striped over 16 data servers. Starting from the beginning of the file, a client application reads 4KB (a default value) in each request until it reaches the end of file. We configured Hadoop to use the default PVFS and HDFS configurations with no replication (single copy); PVFS uses 64MB stripe units with no read-ahead buffering and replication.

If a client's Hadoop application requests 4KB of data, PVFS without readahead buffering fetches only 4KB from a server. By performing such small reads, PVFS's read throughput is dominated by high message processing overhead and network latency; as shown in Figure 3, PVFS reads data at less than 10 MB/s (for the 4KB buffer size). Unlike PVFS, HDFS reads the entire 64MB chunk asynchronously and sends it to the application as soon as the first 4KB arrives. Figure 3 shows how PVFS's read performance improves with larger readahead buffers. The readahead performance peaks when the shim layer is configured to use a 4MB buffer; using even bigger buffers does not yield any higher read throughput because the server's local file system (ext3) is saturated and is delivering its peak read throughput. Our evaluation uses a 4MB buffer for the Hadoop-PVFS shim and a 128KB buffer for HDFS, which is the recommended value for Hadoop clusters [21].

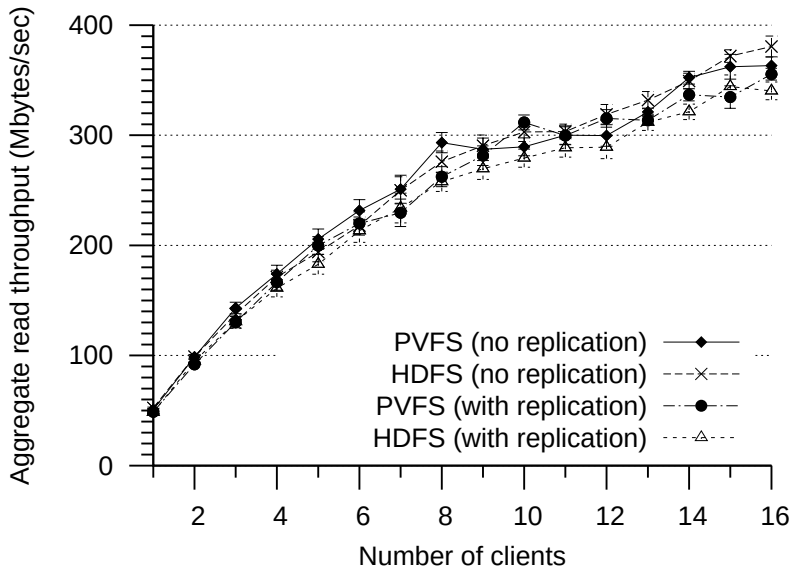


**Figure 4. Clients reading concurrently from different files** – *PVFS's read performance is similar to HDFS.*

---

# 5 Performance of Microbenchmarks

In this section, we report the results of PVFS and HDFS performance using microbenchmarks that represent common use cases in data-intensive applications. All microbenchmark experiments were done using the 20-node SS cluster and the applications use the file systems directly without using Hadoop framework.

## 5.1 Concurrent reads from different files

This experiment represents the most common use case of Internet services, where multiple clients are each reading a different large file that is spread over many servers. We use 1-16 clients, each reading a different 1GB files (striped on 16 data servers) sequentially from start to end, 4KB at a time. Figure 4 shows that PVFS performs as well as HDFS both in a replicated and a non-replicated configuration.

During this experiment, we discovered a bug in the way PVFS chose its first data server for the chunks. PVFS used the current time as a seed to choose a server; but when the client applications were started concurrently, they always used the same first server (and layout pattern). We fixed it such that each client can randomly choose the first server and used the fixed codebase for all our experiments.

## 5.2 Concurrent reads from a single file

This experiment is similar to the previous microbenchmark except that instead of reading different files, all clients are reading different (non-overlapping) regions from the same file. For a configuration of $N$ clients reading from 64*N megabyte file, each client is reading a unique $1/N^{th}$ chunk of the file.

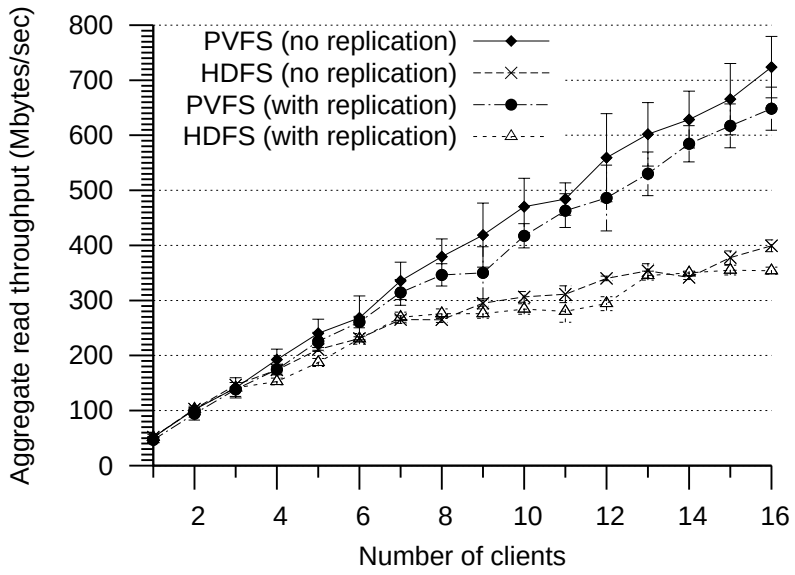


**Figure 5. Clients reading concurrently from one single file** – *PVFS read throughput increases linearly with more clients. Unlike HDFS's random file layout policy, which leads to contention at the servers, PVFS uses a uniform round robin file layout that spreads the read requests uniformly across all its I/O servers.*

---

Figure 5 shows that PVFS does significantly better with increasing number of clients. Its read performance increases linearly because adding more clients does not affect reading files that are striped uniformly using a round robin pattern. HDFS's random file layout policy creates hot-spots at some data servers that results in I/O contention on those servers.

Hadoop is able to mask this sub-optimal performance of HDFS's non uniform file layout by scheduling tasks in a manner that that achieves load balancing across all nodes. To understand this more, we repeated the above experiment by using HDFS and PVFS with the Hadoop framework. In this experiment, we use a "dummy" map task that just reads the data; we do not perform any computation and have no reduce tasks in the system. Figure 6 shows that HDFS benefits significantly from Hadoop's intelligent load-balanced task scheduling. HDFS delivers a read throughput of 525 MB/s, which is 66% more than using HDFS without Hadoop (in Figure 5).

### 5.3 Concurrent writes to different files

This microbenchmark measures the write performance when multiple clients are concurrently writing to separate files that are spread over 16 data servers. This is common in the "reduce" phase of a Mapreduce job, when the nodes running the "reduce" task all generate and write different output files.

Figure 7 shows the write performance with and without replication. As expected, more copies require more write operations, and both HDFS and PVFS slow down. With a replication factor $N$ (i.e. $N$ copies), PVFS writes to $N$ different servers at an aggregate speed of $1/N^{th}$ the single copy configuration and HDFS writes $N-1$ copies remotely. With the default replication setting (i.e., keeping three copies), HDFS writes the first copy locally and the other two copies on different

| Reading a 16GB file (with Hadoop's load balancing) | HDFS | PVFS |
|---|---|---|
| **Read throughput (MB/s)** | **524.9** | **509.7** |
| Read network traffic (MB) | 234.8 | 187.8 |
| Write network traffic (MB) | 207.5 | 155.4 |
| Completion time (min:sec) | 0:31 | 0:32 |

**Figure 6. Performance of reading a single file with Hadoop** – *Hadoop uses a load-balanced scheduling mechanism to mask the non uniform chunk layout resulting from HDFS's random chunk placement policy. Using HDFS with Hadoop delivers a read throughput, of 525 MB/s, that is two-thirds more than HDFS without Hadoop (which is about 350 MB/s, in Figure 5).*

servers.

With no replication (i.e., a single copy), HDFS's policy of writing the first copy locally allows it to scale linearly in write bandwidth. One the other hand, PVFS with a single copy will send all writes to a server. In case of fewer number of clients (or writers), this allows PVFS to benefit from shifting all the file I/O work to the otherwise idle servers; but this benefit is diminished when many clients begin to send data over the network and saturate all the data servers. In practice, it is hard to imagine HDFS or PVFS being deployed in a production systems without enabling replication.

### 5.4 Concurrent writes to a single file

Multiple clients writing concurrently to the same file is one of the common access patterns in high performance computing. While this "concurrent writing" scenario is well supported by most parallel file systems, Internet services file systems including GoogleFS and HDFS do not optimize for this use case. HDFS semantics only allow a single process to write to a file at any time.

However, we believe that many data-intensive workloads may benefit from the support for concurrent writing to the same file. One such example is copying an existing large file. Figure 8 shows the performance of clients copying a large 16 GB file, striped on 16 data servers, from its source to the destination. Both PVFS and HDFS have enabled their default replication scheme of keeping 3 copies of each file, so all writes to the remote destination will happen thrice for PVFS and twice for HDFS. HDFS does not allow multiple clients to write to the same file, so it can only use a single client to perform this file copy. PVFS enables concurrent writes to non-overlapping regions of the file and it can support a highly parallel copy where $N$ clients write $1/N$ each of the big file to the destination. Using all 16 nodes to copy different chunks of the file, PVFS outperforms HDFS by completing the copy more than four times faster than HDFS (Figure 8).

## 6 Performance of Real Applications on Hadoop/Mapreduce

In this section, we compare the performance of PVFS with HDFS using a suite of four data-intensive applications that ran on the 400-node M45 cluster using the Hadoop/Mapreduce framework with replication enabled (three copies). For each application, we report aggregate numbers for all I/O throughput, CPU statistics and network traffic. We describe Hadoop/Mapreduce execution model along with the different I/O operations that are measured in our experiments.

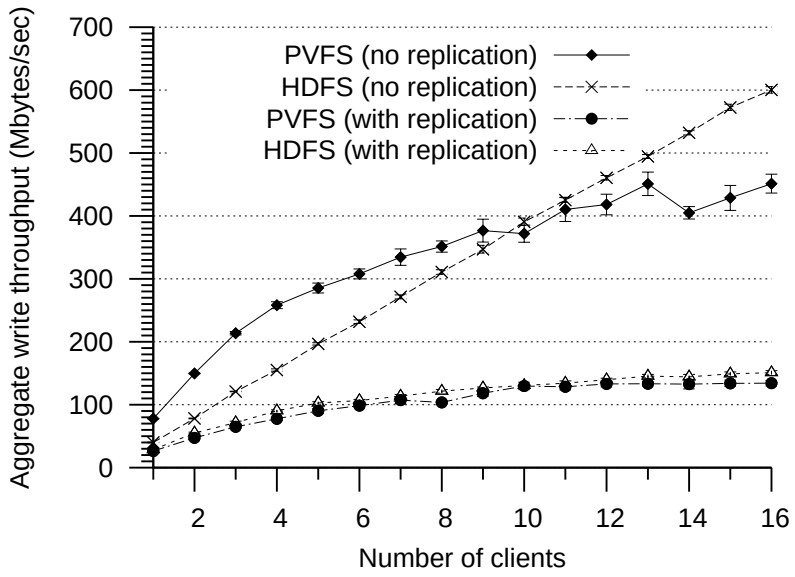


**Figure 7. Clients writing concurrently to separate files** – *Keeping three copies of each file requires both HDFS and PVFS to perform more write operations. All PVFS writes are sent to the data servers, HDFS sends all but the first write to remote servers. By performing its first write locally, HDFS with no replication sees a linear scale up in write performance.*

---

The Hadoop job scheduler divides an application job into multiple tasks that operate on an input file that is striped into chunks by HDFS. By default, the job scheduler assigns one task per chunk. It tries to assign the "map" task to a node that also stores the respective chunk (such maps are called *local maps*). If the scheduler cannot make local assignments, the "map" task is assigned to a remote node; these maps are known as *remote maps*. Once the "map" task is assigned, the node reads the input data in chunks, performs computation on each chunk and writes the output to intermediate files stored on local disks. It repeats this process until all the input has been processed; we measure the throughput of reading the input data. These intermediate files are transferred to the local storage of nodes that run "reduce" tasks using a daemon process. The "reduce" nodes read the data from their local storage, apply the appropriate reduce function and write the final output using HDFS or PVFS. This process repeats until all the output have been written; we measure the throughput of writing the output data.

For each application, we also report the amount of data sent and received over the network. This includes the data transfers when the input is read by the "map" tasks, when the intermediate file moved from the "maps" to the "reduce" tasks and when the output written by the "reduce" tasks. We also measure the completion time of each application (with the time it took to run the map and reduce functions) along with the average per-node CPU utilization.

Hadoop's implementation also uses various optimizations proposed by Google's Mapreduce. Hadoop runs backup tasks that help minimize a job's response time by re-executing the "straggler" tasks that have failed to complete. Another optimization is that Hadoop uses the data layout to assign tasks in a manner that load-balances the file I/O performed in reading the input data. Hadoop schedules tasks to be run out of order to overcome the potential drawbacks manifesting from the HDFS's non-uniform chunk placement (observed in the previous section).

| Copying a 16GB file | HDFS | PVFS |
|---|---|---|
| Throughput (MB/s) | 24.6 | 105.5 |
| Read network traffic (GB) | 49.7 | 59.0 |
| Write network traffic (GB) | 48.1 | 59.2 |
| **Completion time (min:sec)** | **10:50** | **2:31** |

**Figure 8. Using copy to quantify the benefits of concurrent writes to a file** – *PVFS enables highly concurrent writes to non-overlapping regions and uses N clients to copy $1/N$ each. By not allowing multiple writers to a file, HDFS can only go as fast as a single client can.*

### 6.1 Application suite

The first two applications – *grep* and *sort* – are the most common workloads of Internet services that have been previously used to evaluate the Mapreduce model [14, 36]. The last two applications are real data-intensive applications that are using the Hadoop framework for data processing.

**`grep` benchmark** – The *grep* utility scans through a large file, searching for a rare pattern "aaa" (that occurs only than 5985 times).Our input data-set comprised of 50 files, each consisting of two billion 100-byte records. The input data consists of a set of arbitrarily generated strings. In this experiment, we launch 1500 tasks (one task per input chunk) on 50 nodes and the job scheduler tries to assign each task to a node that stores the respective 64MB chunk. Due to the read-intensive nature of *grep*, we modified this benchmark to only perform the "map" phase and terminate without doing any "reduce" phase.

**`sort` benchmark** – The *sort* utility sorts a file of two billion 100-byte records on a key represented by the first 10-bytes from each record. The input data set has 50 files that are assigned to 1500 tasks, one per input chunk. Unlike *grep*, this application writes as much it reads, thus generating a write-intensive workload in the "reduce" phase.

**NELL prototype** – The NELL (Never-Ending Language Learning) prototype system classifies noun phrases according to lexical categories such as "city" and "country" [6]. To accomplish its goals, NELL computes the co-occurrence statistics of the candidate noun phrases with positive and negative contextual patterns (e.g. "cities such as _" and "countries such as _") in a large collection of web pages. When populating two categories at once, there are on the order of $10^5$ phrases (e.g. "cities such as New York" or "countries such as New York") whose occurrence counts need to be computed over a collection of 8 million web pages. The authors of this suite commonly run their application on a 100 node Hadoop cluster using a 37GB input data set. It takes them less than 10 minutes to complete the execution.

**Pagerank application** – This application performs graph analysis (topic sensitive reversed Pagerank) based on link structure of 100 million web pages along with the reading difficulty level of each site. In general, typical web pages are hard for lower level language learners to read. To help determine a group of web pages that is easier to read, this application uses readability level information of each web page, along with link structure of web pages in its link analysis. The result would then rank web pages based on the likelihood that a person can reach out to web pages with low readability levels, i.e. web pages that are easy to read. This application runs on 50 nodes using 8GB input data set stored in HDFS (or PVFS) along with the 100MB file which contains readability level information that is stored locally on each node.

| | `grep` benchmark | | `sort` benchmark | | NELL suite | | Pagerank app | |
|---|---|---|---|---|---|---|---|---|
| | **HDFS** | **PVFS** | **HDFS** | **PVFS** | **HDFS** | **PVFS** | **HDFS** | **PVFS** |
| **Experimental setup** (number of nodes, input data-set size and number of Hadoop "maps") | | | | | | | | |
| Number of nodes | 50 | 50 | 50 | 50 | 100 | 100 | 50 | 50 |
| Input data set (GB) | 100 | 100 | 100 | 100 | 37 | 37 | 8 | 8 |
| Number of local maps | 1455 | 1435 | 1486 | 1512 | 617 | 680 | 182 | 237 |
| Number of remote maps | 47 | 86 | 23 | 15 | 108 | 44 | 68 | 14 |
| Number of backup maps | 97 | 110 | 86 | 120 | 22 | 34 | 0 | 0 |
| **Aggregate performance** (file I/O throughput, amount of network traffic and execution time) | | | | | | | | |
| Input (read) rate (MB/s) | 816.56 | 879.83 | 403.51 | 366.65 | 78.46 | 75.95 | 28.48 | 35.46 |
| Output (write) rate (MB/s) | n/a | n/a | 175.17 | 110.67 | 11.5 | 9.84 | 0.10 | 0.11 |
| Data sent (GB) | 4.09 | 12.43 | 365.81 | 479.32 | 4.26 | 3.03 | 9.63 | 9.19 |
| Data received (GB) | 3.94 | 12.4 | 356.87 | 468.99 | 4.47 | 3.24 | 10.43 | 8.73 |
| Per-node CPU utilization | 39% | 39% | 20% | 12% | 31% | 30% | 2% | 2% |
| Run time (min:sec) | 1:56 | 1:52 | 13:55 | 20:06 | 8:13 | 8:35 | 72:27 | 69:52 |
| *% of time for "map" tasks* | 100% | 100% | 31.35% | 22.32% | 98.62% | 98:03% | 6.66% | 6.15% |
| *% of time for "reduce" tasks* | n/a | n/a | 68.65% | 77.78% | 1.38% | 1.97% | 93.34% | 93.85% |

**Figure 9. Comparing HDFS and PVFS performance for real applications executed through the Hadoop/Mapreduce abstraction** – *PVFS performance is comparable to HDFS for all four data-intensive applications. Both `grep` and NELL, are read-intensive benchmarks that spend almost all their time reading data for the "map" tasks. The Pagerank application is also read-intensive and the algorithm is designed to use only one "reduce" task. This single "reduce" task takes a long time to complete and, hence, accounts for more than 93% of the total execution time and biases the average per-node CPU utilization (without this "reduce" the utilization was about 25%). In contrast, `sort` writes as much data as it reads. Because we enable replication (i.e., 3 copies), PVFS sends writes a file to three different servers but HDFS sends writes to two servers; by writing the first copy locally, `sort` using HDFS is faster than running `sort` on PVFS. We confirmed this by running PVFS with two copies and HDFS with three copies, and both had similar completion times.*

### 6.2 Results

Figure 9 shows that the performance of PVFS is on par with HDFS for our four data-intensive applications. Both *grep* and NELL are read-intensive workloads that process large amounts of input data to generate miniscule output information. These applications achieve similar read throughput, but have a very low write throughput because of the lack of any large output files. PVFS and HDFS run at almost identical speeds when both file systems are keeping three copies each. Note that the *grep* benchmark has been modified to only perform the "map" tasks; hence there are no "reduce" tasks to write any data.

In case of *sort*, PVFS is about one-third slower than HDFS. Unlike *grep*, the *sort* application writes as much as it reads. That is, by keeping three copies, write operations are three times slower than reads; by writing one copy locally, HDFS does better than PVFS. We confirmed this phenomenon by running PVFS with two copies and HDFS with three copies. PVFS with two copies delivers an output rate of 178 MB/second and completes the application run in 13 minutes and 38 seconds – both are comparable to HDFS with three copies (one local and two remote).

The Pagerank application is a compute intensive workload that reads about 8GB of input data and writes very small output. The algorithm in this application is designed to use only one reduce task. This single "reduce" task takes a long time to complete and, hence, accounts for more than 93% of the total execution time and biases the average per-node CPU utilization (without this "reduce" the utilization was about 25%).

Our knowledge about different Internet services applications is limited to the handful of papers published by Google, Yahoo and other Internet services. While our current evaluation is only the tip of the iceberg, it covers four popular categories of large applications, and in future, we would like to experiment with real Internet applications or benchmarks.

# 7 Conclusion and Future Work

Internet services are building file systems from-scratch that can deliver scalable performance for their anticipated data-intensive applications workload. High performance computing has applications at similar scale that use parallel file systems for their demanding storage I/O requirements. This paper explores the relationship between modern parallel file systems, represented by PVFS, and purpose-built Internet services file systems, represented by HDFS, in the context of their design and performance. We show that PVFS can perform comparable to HDFS in the Hadoop Internet services stack. Our evaluation is done at a large scale on Yahoo!'s M45 cluster with microbenchmarks and real data-intensive applications.

The biggest difference between PVFS and HDFS is the redundancy scheme for handling failures. HDFS keeps three copies of file data on different servers in the cluster. Users of most parallel file systems do not want to pay for the 200% overhead of triplication when they can use RAID (single or double protection) with overheads of 10-25%, so they stripe RAID 0 over data servers that concentrate many disks into RAID systems. This may be changing however as at least one parallel file system uses mirroring as a concurrent write optimization and can do RAID across nodes [34], and many more are planning to offer higher levels of replication; our shim implementation emulates file triplication for PVFS.

PVFS and HDFS have contrasting file access semantics. HDFS uses "write-once" semantics, allowing only a single writer of a file at any time. PVFS provides UNIX file system API and supports most POSIX semantics; in particular, its ability to support concurrent writers to a file enables high parallelism. Many data-intensive applications could benefit from a parallel file system that supports much broader range of workloads for large data sets.

Another big difference is that Internet services file systems, both HDFS and GoogleFS, expose the file distribution information to the Mapreduce. Hadoop job schedulers use this information for load-balanced job distribution masking the potential sub-optimal chunk distribution, random rather than striped, by HDFS. PVFS and other parallel file systems also expose their file layout mapping. Layout map is available in PVFS and can be exposed to Hadoop using our shim layer. The emerging NFSv4.1 (or pNFS) standard may also be able to make file layout available to client applications. For write intensive workloads, HDFS's optimization to write one copy of the file locally, avoiding network traffic, can be impactful.

On balance, we believe that parallel file systems could be made available for use in Hadoop, while delivering promising performance for diverse access patterns. These services can benefit from parallel file system specializations for concurrent writing, faster metadata and small file operations. With a range of parallel file systems to choose from, Internet services can select a system that better integrates their local data management tools. In future, we plan to investigate the "opposite" direction; that is, how could we use Internet services file systems for HPC applications.

## References


[1] Private Communication with Rob Ross and Sam Lang, PVFS developers. Argonne Natonal Lab.

[2] Amazon-EBS. Amazon Elastic Block Storage (Amazon EBS). http://www.amazon.com/s3.

[3] Amazon-EC2. Amazon Elastic Compute Cloud (Amazon EC2). http://www.amazon.com/ec2.

[4] Amazon-S3. Amazon Simple Storage Service (Amazon S3). http://www.amazon.com/s3.

[5] Amazon-S3-Docs. Amazon Simple Storage Service (S3) Developer Guide. http://docs.amazonwebservices.com/AmazonS3/2006-03-01/.

[6] Betteridge, J., Carlson, A., Hong, S. A., Jr., E. R. H., Law, E. L. M., Mitchell, T. M., and H.Wang, S. Toward Never Ending Language Learning. In *Extended Abstract submitted to AAAI 2008*.

[7] Bryant, R. E. Data-Intensive Supercomputing: The Case for DISC. Tech. Rep. CMU-CS-07-128, Carnegie Mellon University, May 2007.

[8] CCC. Big-Data Computing Study Group at the Computing Community Consortium. http://www.cra.org/ccc/bigdata.php.

[9] Centera. EMC Centera Family. http://www.emc.com/products/family/emc-centera-family.htm.

[10] Cloudstore. Cloudstore distributed file system (formerly, Kosmos file system). http://kosmosfs.sourceforge.net/.

[11] CluE. Cluster Exploratory (CluE) Program at NSF. http://www.nsf.gov/pubs/2008/nsf08560/nsf08560.htm.

[12] CXFS. SGI InfiniteStorage Shared CXFS. http://www.sgi.com/products/storage/software/cxfs.html.

[13] Dean, J. Experiences with MapReduce, an Abstraction for Large-Scale Computation. Slides from Keynote talk at PACT 2006 on September 18, 2006 in Seattle WA.

[14] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)* (San Francisco CA, Dec. 2004).

[15] Gartner. Gartner Says Cloud Computing Will Be As Influential As E-business. http://www.gartner.com/it/page.jsp?id=707508, 2008.

[16] GFS. Red Hat Global File System. http://www.redhat.com/gfs.

[17] Ghemawat, S., Gobioff, H., and Lueng, S.-T. Google File System. In *Proc. of 19th ACM Symposium on Operating Systems Principles (SOSP '03)* (Rochester NY, Oct. 2003).

[18] Gibson, G., and Corbett, P. pNFS Problem Statement. Internet Draft, July 2004.

[19] Gibson, G. A., Welch, B. B., Nagle, D. B., and Moxon, B. C. ObjectStorage: Scalable Bandwidth for HPC Clusters. In *Proc of the Cluster World Conference and Expo* (San Jose CA, 2003).

[20] Hadoop. Apache Hadoop Project. http://hadoop.apache.org/.

[21] Hadoop-Docs. The Hadoop Cluster Setup. http://hadoop.apache.org/core/docs/current/cluster_setup.html.

[22] Hadoop-S3. Using the Amazon S3 backend on Hadoop. http://wiki.apache.org/hadoop/AmazonS3.

[23] HDFS. The Hadoop Distributed File System: Architecture and Design. http://hadoop.apache.org/core/docs/r0.16.4/hdfsdesign.html.

[24] IDC. HPC Server Market Maintains Strong Growth. http://www.idc.com/getdoc.jsp?containerId=prUS20975507, Dec. 2007.

[25] IETF. NFS v4.1 specifications. http://tools.ietf.org/wg/nfsv4/.

[26] Lustre. Lustre File System. http://www.lustre.org.

[27] NSF. NSF Partners With Google and IBM to Enhance Academic Research Opportunities. http://www.nsf.gov/news/news_summ.jsp?cntn_id=111186.

[28] PVFS2. Parallel Virtual File System, Version 2. http://www.pvfs2.org.

[29] Quinlan, S., and Dorward, S. Venti: a new approach to archival storage. In *Proc. of the FAST '02 Conference on File and Storage Technologies* (Monterey CA, Jan. 2002).

[30] Schmuck, F., and Haskin, R. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proc. of the FAST '02 Conference on File and Storage Technologies* (Monterey CA, Jan. 2002).

[31] Schroeder, B., and Gibson, G. A. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *Proc. of the FAST '07 Conference on File and Storage Technologies* (San Jose CA, Feb. 2007).

[32] Soltis, S. R., Ruwart, T. M., and O'Keefe, M. T. The Global File System. In *Proc. of the 5th NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies* (College Park MA, Sept. 1996).

[33] van Renesse, R., Tanenbaum, A. S., and Wilschut, A. The Design of a High-Performance File Server. In *Proc of the 9th International Conference on Distributed Computing Systems (ICDCS)* (Newport Beach CA, 1989).

[34] Welch, B., Unangst, M., Abbasi, Z., Gibson, G., Mueller, B., Small, J., Zelenka, J., and Zhou, B. Scalable Performance of the Panasas Parallel File System. In *Proc. of the FAST '08 Conference on File and Storage Technologies* (San Jose CA, Feb. 2008).

[35] Yahoo! Yahoo! Reaches for the Stars with M45 Supercomputing Project. http://research.yahoo.com/node/1884.

[36] Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R., and Stoica, I. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)* (San Diego CA, Nov. 2008).