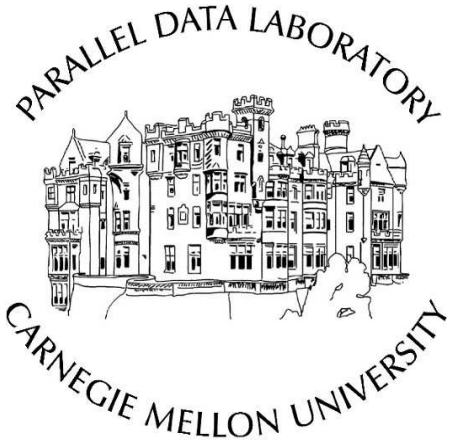

HPC Computation on Hadoop Storage with PLFS

Chuck Cranor, Milo Polte, Garth Gibson

CMU-PDL-12-115
November 2012

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

Abstract

In this report we describe how we adapted the Parallel Log Structured Filesystem (PLFS) to enable HPC applications to be able read and write data from the HDFS cloud storage subsystem. Our enhanced version of PLFS provides HPC applications with the ability to concurrently write from multiple compute nodes into a single file stored in HDFS, thus allowing HPC applications to checkpoint. Our results show that HDFS combined with our PLFS HDFS I/O Store module is able to handle a concurrent write checkpoint workload generated by a benchmark with good performance.

Acknowledgements: The work in this paper is based on research supported in part by the Los Alamos National Laboratory, under subcontract number 54515 and 153593 (IRHPIT), by the Department of Energy, under award number DE-FC02-06ER25767 (PDSI), by the National Science Foundation, under award number CNS-1042543 (PRObE), and by the Qatar National Research Fund, under award number NPRP 09-1116-1-172 (Qloud). We also thank the members and companies of the PDL Consortium (including Actifio, APC, EMC, Emulex, Facebook, Fusion-IO, Google, Hewlett-Packard, Hitachi, Huawei, IBM, Intel, LSI, Microsoft, NEC, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, VMware, Western Digital) for their interest, insights, feedback, and support.

Keywords: HPC Computing, Cloud Storage, PLFS, HDFS, PVFS

1 Introduction

High Performance Computing (HPC) applications are typically large environmental simulations that compute for a long time. These applications are launched across a large number of compute nodes using the Message Passing Interface (MPI) subsystem [7]. In addition to starting the HPC application on the compute nodes, MPI also provides group communication synchronization features such as barriers.

HPC applications protect themselves from failure by periodically pausing and concurrently writing their state to a checkpoint file in a POSIX-based distributed filesystem (DFS) such as Lustre [6], PanFS [13], PVFS [8] or GPFS [9]. Checkpoint I/O can be particularity challenging when all processes in the parallel application write to the same checkpoint file at the same time, an access pattern known as N-1 writing [2].

An increasing number of clusters are being configured for data analytics using the Apache Hadoop open source version of the Google Internet services tool suite (Google File System, BigTable, etc.) [4, 3]. Because HPC and Internet services analytics are both big data and big compute applications families, it is desirable to be able to mix and match applications on cluster infrastructure. Prior research has demonstrated that Hadoop applications can run efficiently on HPC cluster infrastructure [1]. However, cloud storage systems such as the Hadoop Distributed Filesystem (HDFS) [10] are not POSIX-based and do not support multiple concurrent writers to a file. In order to enable the convergence of HPC and Cloud computing on the same platform, we set out to provide a way to enable HPC applications to checkpoint their data to a Cloud filesystem, even if all processes write to the same file.

In this paper we describe how we adapted the Parallel Log Structured Filesystem (PLFS) to enable HPC applications to be able read and write data from the HDFS cloud storage subsystem. Our enhanced version of PLFS provides HPC applications with the ability to concurrently write from multiple compute nodes into a single file stored in HDFS, thus allowing HPC applications to checkpoint.

2 PLFS

PLFS [2] is an interposing filesystem that sits between HPC applications and one or more backing filesystems (normally DFSs). PLFS has no persistent storage itself. Instead of storing data, the PLFS layer's role is to transparently translate application-level I/O access pattern from their original form into patterns that perform well with modern DFSs. Neither HPC applications nor the backing filesystems need to be aware of or modified for PLFS to be used.

PLFS was developed because HPC parallel file systems suffer lock convoys and other congestion when a large number of different client machines are writing into the same region of a concurrently written file [2]. The key technique used in PLFS to improve DFS performance is to reduce the sharing of distributed filesystem resources by spreading the load out among the I/O nodes. This is achieved in three ways. First, all writes to PLFS are separated out into per-process log files so that each process writes to its own file rather than to a single shared file. Second, when multiple compute nodes are writing to the same file, each node's output stream is hashed to one of the backing filesystems, thus spreading a single file's data across the underlying filesystem volumes. Finally, individual files in a single directory are distributed among multiple configured backing filesystems using hashing. The first two mechanisms improve the performance of N-1 checkpointing, while the third mechanism improves N-N performance (N processes writing to one file each). The second and third mechanisms require multiple backing DFS volumes to be available in order to be effective, while the first mechanism works even if only a single backing volume is available. Of course when applications read data PLFS must collect the indexing information from all the write logs in order to reassemble the data being read.

Each file stored in a PLFS filesystem is mapped to one or more "container directories" in the underlying DFS filesystems. An example of this is shown in Figure 1. The figure shows a N-1 checkpoint file called

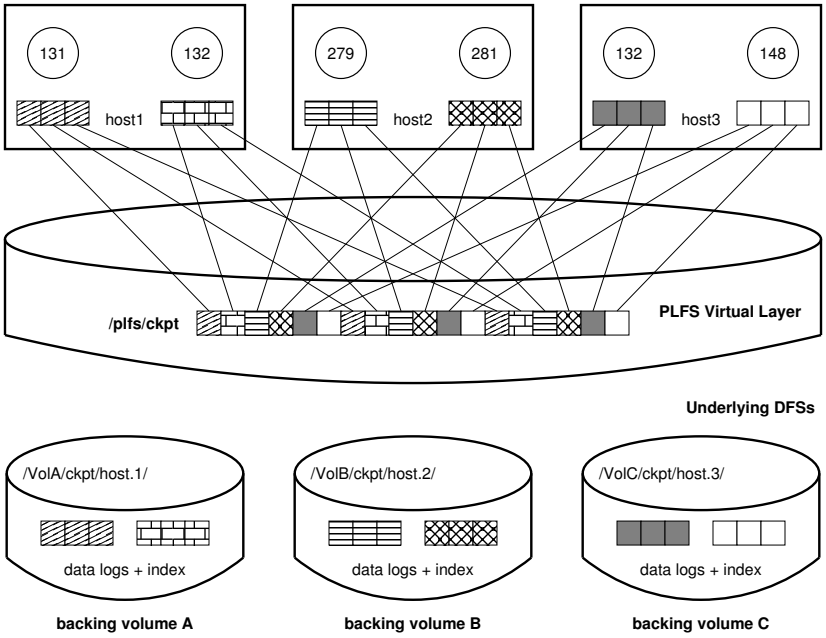

Figure 1: PLFS container structure

`/plfs/ckpt` being written into a PLFS file by six processes distributed across three compute nodes. The PLFS virtual layer shown is backed by three volumes of a DFS (e.g. PanFS). Each process writes blocks of its state to the checkpoint file in PLFS at a given offset. The blocks of data in the checkpoint file are strided, resulting in a process's state blocks being logically distributed throughout the checkpoint file.

PLFS stores the data in the `/plfs/ckpt` checkpoint file in container directories that it creates on the backing DFS volumes. PLFS assigns each host writing to the checkpoint file a backing volume to store data in. For example, writes from `host1` are mapped to backing volume `A`. Then within that volume, each process writing to the checkpoint file is allocated its own private data log file to write to. PLFS keeps indexing information for all writes so that it can reassemble all the data in the log files back into the virtual checkpoint file if it is read. Without the PLFS layer, the application would be storing all its data in a single checkpoint file on a single backing DFS volume. This would result in poor checkpoint I/O performance due to bottlenecks in the DFS. However, with PLFS, the checkpoint data is separated into per-process log files that are distributed across multiple backing volumes allowing the application to better take advantage of parallelism in the DFS. To use PLFS, HPC applications just need to be configured to store their checkpoint files in PLFS — no source code level modifications are required.

3 HDFS I/O Store Layer for PLFS

Cloud storage systems such as HDFS do not support concurrent writing into one file. Fortuitously, a key feature of PLFS is that it is log structured. Data written to PLFS files is broken up and separated into log files in a PLFS container directory. The log files are written once and indexed by PLFS for later reads. However, PLFS assumes that log data files, log index files, and directories can be accessed through a mounted filesystem using the standard POSIX I/O system calls. This will not work for storage systems that cannot be mounted as a POSIX filesystem and must be accessed through their own APIs. HDFS, designed only to be accessible through its Java-based API, is an example of one such storage system. PLFS must be enhanced in order to use these types of filesystems as backing store.

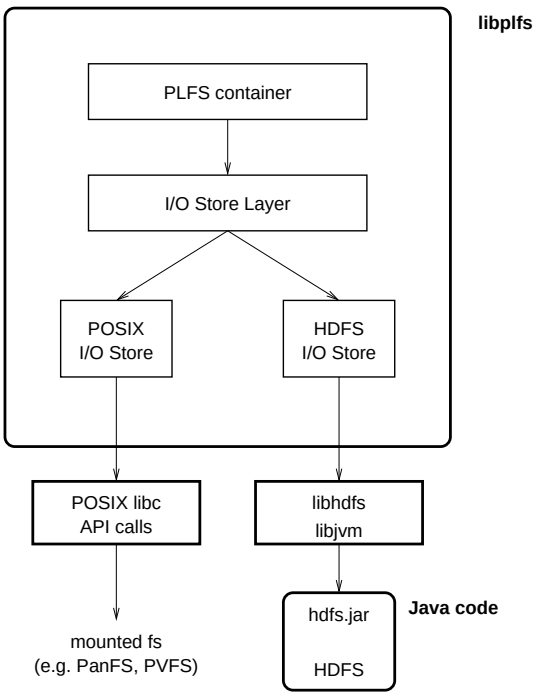

Figure 2: PLFS I/O Store architecture

To enable PLFS to use non-POSIX filesystems as backing store, we have added a new layer of abstraction to PLFS called the I/O Store. PLFS now directs all its backing store I/O calls through the new I/O Store interface. PLFS can now be configured with multiple I/O Stores. We initially wrote a POSIX I/O Store to support the traditional PLFS mode of operation. Next, we implemented an HDFS I/O Store to enable PLFS to be used with Cloud Storage.

Figure 2 shows the architecture of the I/O Store abstraction we have added to PLFS and how it works with both POSIX and HDFS filesystems. The PLFS code that manages containers is unchanged, except for the parts that access the backing store. Instead of calling directly into POSIX I/O system calls, we modified PLFS to call the I/O Store interface instead. The POSIX I/O Store passes I/O requests to the POSIX I/O system calls provided by libc. Our new HDFS I/O Store converts the PLFS I/O Store calls into HDFS API calls and passes them to libhdfs. The libhdfs library is a thin C JNI-based wrapper over the Java HDFS API. To use libhdfs, PLFS must be linked with a Java virtual machine (libjvm) so that it can make calls to HDFS Java routines stored in the the Hadoop hdfs.jar file.

3.1 HDFS I/O Store Implementation Issues

We encountered two types of implementation issues when designing and testing PLFS HDFS. First, we had to adapt the HDFS API to match the I/O Store interface, and there were cases where the semantics of the HDFS API did not line up well with the I/O store interface. Second, using the HDFS API requires us to pull in the entire Java virtual execution environment into our application and there were platform issues associated with that.

When adapting the HDFS API for PLFS we encountered three cases: HDFS APIs that closely match, HDFS APIs that can be made to match with a little help, and APIs that have semantics that HDFS cannot provide. HDFS APIs that closely match the I/O store interface include Java methods for operations such as `access`, `chmod`, `fsync`, `mknod` (no device files), `read/pread`, `rename`, `rmdir`, `unlink`, `write`, and

`utime`.

There were a number of areas where the HDFS API required help to match the I/O Store interface. These areas include:

File group/ownership: HDFS uses text-based user and group names, while PLFS uses integer-based UIDs and GIDs, so we had to use the local password and group file access functions to establish a mapping between these identities for API functions such as `chown`.

Object creation mode: The PLFS I/O Store interface follows the POSIX semantics of allowing an object's permission to be established when a file or directory is first created. HDFS does not provide this semantic, so creating an HDFS object with a given permission requires two HDFS operations: a create followed by a chmod operation.

Reading memory mapped files: HDFS does not support reading files using the `mmap` memory mapped file interface. Since PLFS only reads and never writes files with `mmap`, this interface can be emulated by doing a normal HDFS read into a a memory buffer allocated with `malloc`.

Directory I/O: HDFS does not have a POSIX-like `opendir`, `readddir`, `closedir` interface. We emulate this by caching the entire listing of a directory in memory when it is opened and performing `readdir` operations from this cache.

File truncation: HDFS cannot truncate files to smaller non-zero lengths. This is ok because PLFS only truncates files to size zero. HDFS cannot truncate files to size zero either, but this operation can be emulated by opening an file for writing. In this case HDFS discards the old file and creates a new zero length file.

There are several filesystem semantics that HDFS does not provide, but they do not prevent PLFS from operating. These semantics include opening a file in read/write mode, positional write (`pwrite`) operations, and symbolic link related operations. Since PLFS is a log structured filesystem, it does not need or use read/write mode or writing to any location of a file other than appending to the end of it. PLFS also does not require symbolic links in the backend (but symbolic links have been added to HDFS in version 0.23).

In addition to these issues there are two cases where the semantics provided by the HDFS I/O API are unusual. First, the HDFS API used to create directories (`hdfsCreateDirectory`) functions like the Unix "`mkdir -p`" command — it will create multiple levels of directories at the same time and will not fail if the directories path give already exists. Second, the HDFS API used to rename files and directories ("`hdfsRename`") operates more like the Unix "`mv`" command than the POSIX `rename` system call. Attempts to rename a file or directory to a name of a directory that already exists causes the object being renamed to be moved into the existing target directory rather than having the rename operation fail with a file exists error. This partly breaks PLFS code that handles concurrent file creation races: PLFS still works properly but it is not as efficient as it would be in the POSIX case.

Finally, the HDFS API's handling of errors is unusual because part of it is based around Java exceptions. When HDFS encounters a condition that generates a Java exception, the C-level libhdfs API returns -1 and does not set a meaningful error number. These kinds of exception error occur when trying to perform I/O on an open file that has been unlinked, a situation allowed in POSIX but not HDFS.

3.1.1 Platform Issues

There are two HDFS/Java platform issues that caused us problems. First, we discovered that the HDFS API deadlocks in a Java call if it is used by a child process after a fork operation. This caused problems with the PLFS FUSE program forking off its daemon process at startup time. The main PLFS FUSE process first

connects to HDFS to ensure that the specified backend is available, then it forks off a child process to act as a background daemon and then the parent process exits. This resulted in the PLFS/HDFS FUSE daemon process hanging in a HDFS Java API call the first time it tries to access backing store. We resolved this by moving all HDFS API usage to child processes.

The second HDFS/Java platform issue was a memory leak. The PLFS/HDFS FUSE daemon would grow in size under load until the kernel "out of memory" subsystem started killing userlevel processes. After extensive debugging, we discovered that the memory leak was due to an interaction between PLFS pthread management and HDFS/Java. PLFS parallelizes filesystem read operations by creating a set of threads to perform reads in parallel (using the HDFS/Java API) and then collecting the results as those threads terminate. This results in the HDFS/Java API receiving read calls from a constant stream of freshly allocated pthreads. It appears that HDFS/Java allocates and never releases memory for each new pthread that calls into the Java Virtual Machine. This is the source of the memory leak. We resolved this issue by inserting a thread pool between the HDFS instance of the PLFS I/O Store and the underlying HDFS/Java API calls.

Generally, the addition of a Java run-time system to the PLFS makes debugging more of a challenge due to the number of software systems required to provide PLFS file service. Bugs and bad interactions can occur in PLFS itself, in the FUSE library or kernel module, in the kernel itself, in HDFS code, or in the Java virtual machine.

4 Evaluation

To evaluate how the HDFS I/O Store module handles HPC workloads using a Cloud-based storage system, we ran a series of HPC benchmark runs using the open source File System Test Suite checkpoint benchmark from LANL[5]. Our results show that HDFS combined with our PLFS HDFS I/O Store module is able to handle the concurrent write checkpoint workload generated by the benchmark with good performance.

Our hardware testing platform is the PRObE Marmot cluster hosted by CMU[12]. Each node in the cluster has dual 1.6GHz AMD Opteron processors, 16GB of memory, Gigabit Ethernet, and a 2TB Western Digital SATA disk drive. For our tests we run the 64 bit version of the Ubuntu 10 Linux distribution.

4.1 File System Test Benchmark

The LANL filesystem checkpoint benchmark can generate many types for HPC checkpoint I/O patterns. For all of our tests, we configured the benchmark to generate a concurrently written N-1 checkpoint. Figure 3 illustrates how the checkpoint benchmark operates. All checkpoint file I/O is performed by a set of nodes that synchronize with each other using MPI barriers. In the first phase of the benchmark each node opens the freshly created checkpoint file for writing and then waits at a barrier until all nodes are ready to write. Once all nodes are ready, the benchmark starts concurrently writing the checkpoint data to the file. Each node then writes a fixed number of chunks of data to the checkpoint file (shown as an access unit in the figure). The access units are written to the checkpoint file in strides. Each stride contains one access unit from each node. Once a node has completed writing a stride, it seeks forward in the file to the next stride in order to write its next access unit. Each node continues writing to the checkpoint file until it has written the specified number of access units, then it waits at an MPI barrier until all the other nodes have completed writing the data. Once writing is complete and an MPI barrier reached, each node syncs its data to disk, closes the file, and then waits at a final barrier before finishing.

Before starting the read phase we terminate all processes accessing the underlying files so that we can unmount the filesystem in order to ensure that all freshly written data has been flushed from all the nodes' memory to avoid cached data from unfairly biasing our read performance. After the filesystem has been mounted and restarted, the benchmark reads the checkpoint in the same way it was written, however we shift

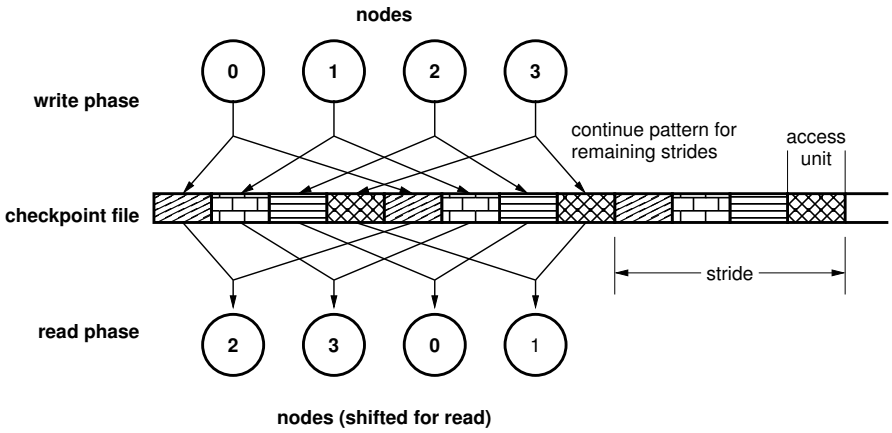

Figure 3: Checkpoint benchmark operation

the nodes around so that each node reads data that some other node wrote (rather than data that it just wrote). This ensures that the benchmark is not reading locally stored data (instead it has to use the network to obtain the data it wants to read).

For all our tests, we configured the benchmark to have each node generate 512MB of checkpoint data. Our test cluster consisted of 64 nodes, so the overall size of the checkpoint file is 32GB. We used 3 access unit sizes for our tests: 47001 bytes, 48KB, and 1MB. The 47001 size is a small, unaligned number observed at LANL in actual applications to be particularly problematic for file systems. The 48K access unit size is close to the 47001 size, but aligned to the system page size (48K is 12 4K pages of physical memory). The 1MB size is shows how the system performs with larger and more disk friendly access unit sizes.

4.2 Filesystem Organization

We used two backing filesystems for our tests: PVFS and HDFS. For PVFS we used the OrangeFS 2.4.8 distribution, and for HDFS we used the version that came with Hadoop 0.21.

PVFS is our baseline traditional HPC distributed filesystem. Each file in PVFS is broken down into stripe-sized chunks. We configured PVFS to use a stripe size of 64MB which is similar to the chunk size used by HDFS. The first stripe sized chunk of a file gets assigned to be stored on a node. Subsequent stripes are assigned using a round-robin policy across the storage nodes. Note that unlike HDFS, PVFS does not replicate data. Instead, PVFS is assumed to be running on top of a RAID-based underlying filesystem in order to provide protection from failure.

Under Hadoop, HDFS is normally not used with hardware RAID controllers. Instead HDFS is configured to write each block of data to three different servers for fault tolerance. For our benchmarking we used HDFS in two modes: HDFS3 and HDFS1. HDFS3 is normal HDFS with 3-way replication, while HDFS1 is HDFS with the replication factor hardwired to 1 (no replicas). HDFS3 sends three times as much data over the network for filesystem writes as compared to PVFS or HDFS1. We included HDFS1 results because that mode of operation is similar to what PVFS provides (no extra replicas). HDFS always writes the first copy of its data to local disk. If the replication factor is 3, then HDFS sends a second copy of the data to a node randomly chosen from the same rack. The third copy of the data is sent to a node in a different rack by the second node. For our experiments, all nodes were connected to the same switch, so the second and third nodes are chosen at random.

Both PLFS and PVFS can be used in one of two ways. First, both filesystem can be accessed through a

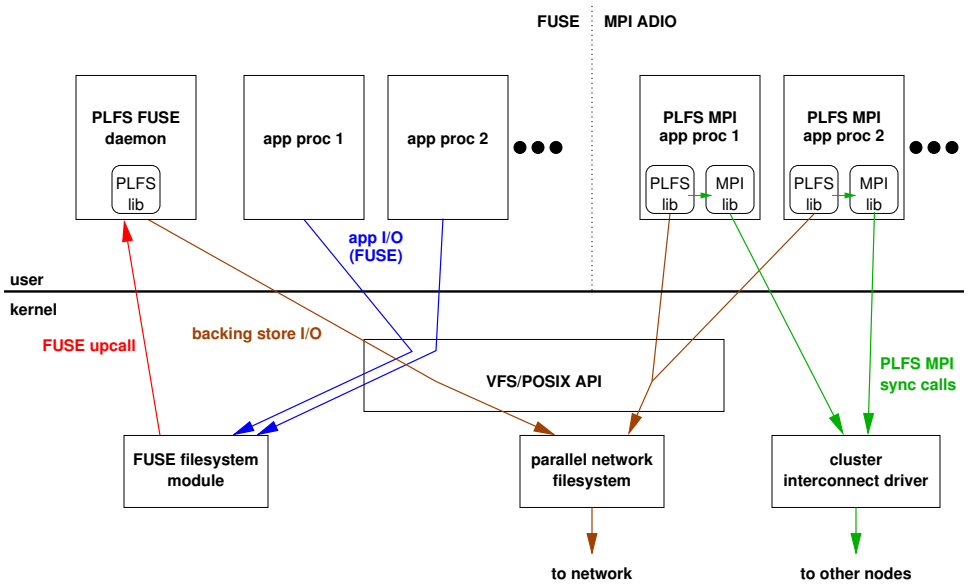

Figure 4: PLFS modes of operation

kernel mountpoint. PVFS provides a Linux kernel module that plugs into the the kernel virtual filesystem layer. PLFS provides the same function by linking with the Linux FUSE (Filesystem in Userspace) subsystem.

The other way both PLFS and PVFS can be used is as a library. For PVFS, this means linking your application with `libpvfs2.a`. The same applies for PLFS (link with `libplfs.a`). HPC applications often use the library approach to access their distributed filesystem in order to avoid the overhead of having to route their data through the kernel. The MPI system provides an abstract-device interface for parallel I/O called ADIO [11] so that HPC developers do not have to port their applications to new filesystem library APIs.

Figure 4 shows the PLFS implementation architecture for both the kernel/FUSE and library modes of operation. Applications using a FUSE kernel mount point have their I/O requests routed through the kernel's VFS layer back up to the user-level FUSE daemon process. This process uses the PLFS library to convert the application's file I/O requests to PLFS backing store container I/O requests and forwards those requests on to the network filesystem. PVFS kernel mounts operate in a similar way, except they use their own kernel VFS module rather than FUSE.

Applications using the library interface to a filesystem either access it directly using its API, or they access it indirectly using the MPI ADIO interface, as shown on the top right of the figure. For PLFS in this case, each application process gets its own instance of the PLFS library linked into it.

For our benchmark runs, we have results from PVFS and the HDFS I/O Store under PLFS using both kernel mount points and the library interface.

4.3 Results

Figure 5 shows the write bandwidths for PVFS and PLFS with the HDFS I/O Store module under kernel and library configurations using access unit sizes of 47001, 48K, and 1M. The results for each test have been averaged over 5 runs made on our 64 node cluster. The error bars indicate the standard deviation across the 5 runs.

The plot shows that the HDFS I/O Store module can support concurrent write workloads well under the worst case access unit size of 47001 bytes. To interpret the numbers, note that PVFS is one remote copy, HDFS1 is one local (no network) copy, and HDFS3 is one local and two remote copies. The HDFS1

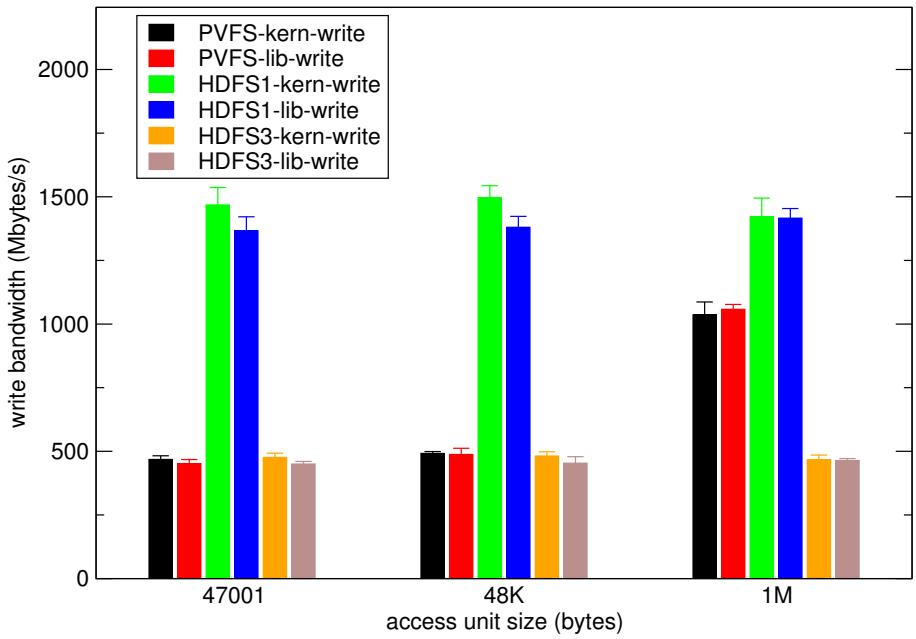

Figure 5: Write bandwidths

bandwidth is limited to around 1450 Mbytes/s by the speed of the local disk. HDFS3 achieves around 1/3 of the HDFS1 bandwidth due to the extra copying, network transits and disk writing. The PVFS bandwidth is limited by the synchronous network at 47001 bytes and increases as more efficient access unit sizes are used (1M).

Figure 6 shows the read bandwidths achieved by the benchmark after writing the checkpoint file. These results are also averaged over 5 runs with the standard deviation shown in the error bars. Note that nodes are shifted for readback so that no client reads the data it wrote, as shown in Figure 3. This means that for HDFS1 the data being read will always be on a remote node.

In the readback, both HDFS1 and HDFS3 do well. For small access unit sizes HDFS outperforms PVFS. This is because of the log structured writes that PLFS performs with HDFS. PVFS does not have log grouping of striped data on readback. For HDFS, reading from three copies with HDFS3 is around 20% slower than reading from one copy with HDFS1. This is because with three copies HDFS has a scheduling choice as to which of the three copies it reads, where as with HDFS1 it has no choice. The HDFS1 copy of the checkpoint file is perfectly balanced and HDFS scheduling cannot make that worse. With HDFS3 and two of the three copies being randomly placed, the HDFS scheduler can force itself into unbalanced I/O patters.

Figure 7 shows the total number of bytes served by each node during a read operation for one of the runs using a 1MB access unit size. The figure clearly shows that the HDFS1 I/O access pattern is almost perfectly balanced between the 64 nodes, where as the HDFS3 I/O pattern is unbalanced due to choices made by the HDFS I/O scheduler when choosing which of the three copies of the data to access.

For the unaligned 47001 access unit size versus the 48K access unit size, the main change is an improvement in the HDFS readback bandwidth under the kernel mount case. This is because unaligned readback through the kernel mount must go through the Linux buffer cache which stores file data in units of memory pages. In order to fill out the data outside of the 47001 access unit size to page boundaries, the kernel must fetch checkpoint data from neighboring blocks in the checkpoint file. This extra work to align the data to page sized boundaries for the buffer cache results in a 20% performance loss. The library case is not affected by this loss because it does not use the Linux kernel buffer cache.

The 47001 and 48K access unit sizes are small relative to efficient access units for disk and networks, so

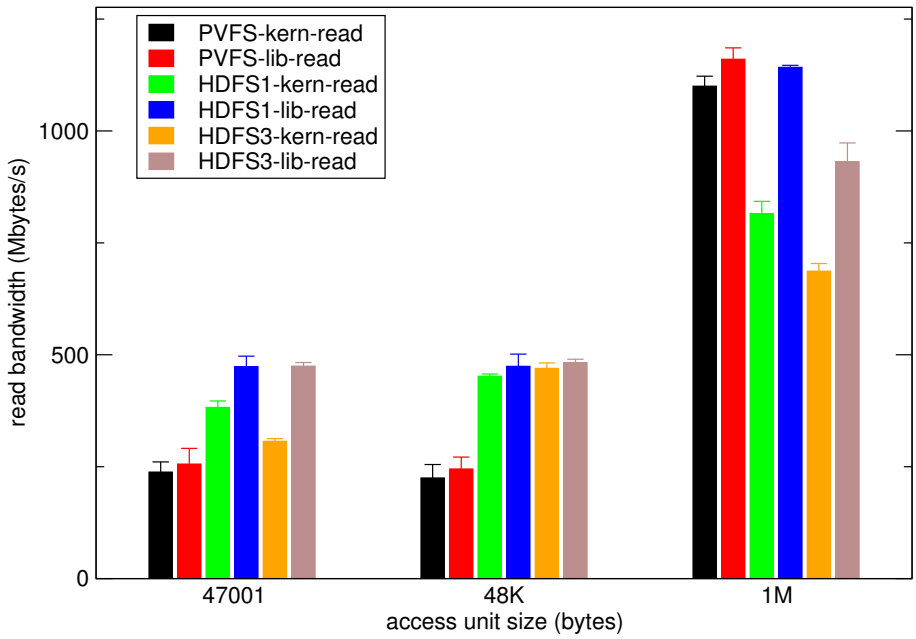

Figure 6: Read bandwidths

we also ran our tests with a 1MB access unit size. With 1MB, the HDFS write bandwidths are unaffected because they are operating under PLFS and PLFS uses log structured writes so it is disk bottlenecked. PVFS write gets much faster with larger access unit sizes because it does access sized transfers, so a larger access unit size results in fewer different transfers, allowing PVFS to go faster.

For readback with a 1MB access unit size, the PVFS bandwidth is about the same as the write bandwidth. For HDFS, reading 1MB does not got as fast as PVFS, though the library version of HDFS comes close. HDFS has the added overhead of an extra data copy between the the HDFS Java virtual machine and the PLFS code, and in the case of HDFS1-kernel the FUSE implementation may not be as efficient as the PVFS VFS kernel module. In addition to this, the HDFS3 bandwidth also suffers from the excess scheduler freedom (shown in Figure 7) relative to HDFS1.

The performance of the benchmark under HDFS when it is directly linked to the PLFS library should be close to the kind of performance HPC applications would see when using HDFS with the PLFS ADIO interface under MPI. In this case the write performance is unaffected because it is disk limited. The read performance for the 47001 access unit size gets better because it avoids alignment problems associated with the Linux buffer cache. Also the read performance for 1MB transfers gets more efficient by about third.

5 Conclusions

It might seem that HPC's use of concurrently written checkpoint files would be incompatible with the single-writer, immutable file semantics offered by Hadoop's HDFS Cloud file system. In fact, the mechanism needed to support concurrently written files, and most of the rest of the POSIX API suite commonly used by HPC, can be provided by HPC's PLFS checkpoint file system. In this paper we present an adaptation of PLFS's storage layer to the particulars of the HDFS client API and demonstrate the excellent performance PLFS and HDFS can provide for HPC applications.

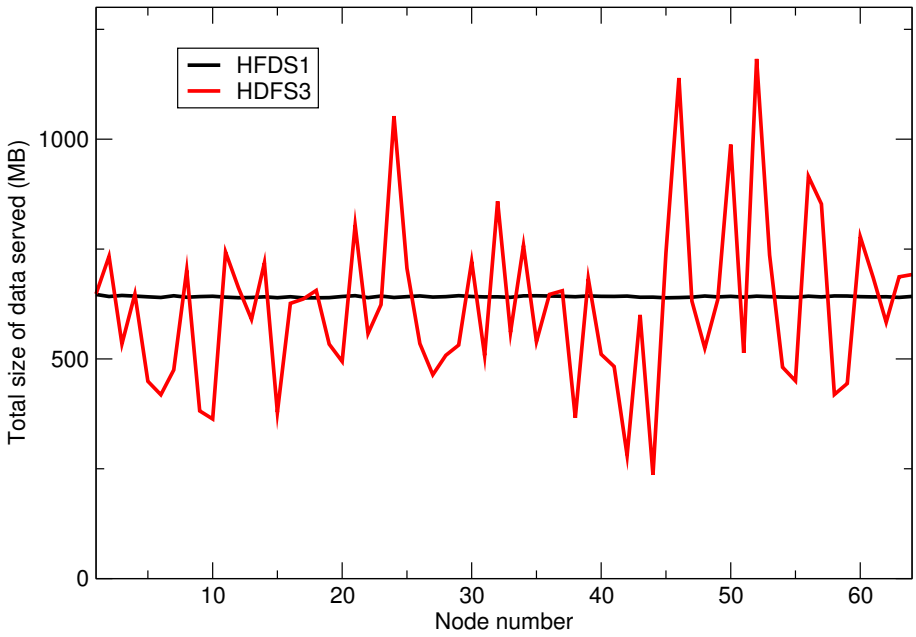

Figure 7: HDFS1 and HDFS3 I/O access patterns

References

[1] Rajagopal Ananthanarayanan, Karan Gupta, Prashant Pandey, Himabindu Pucha, Prasenjit Sarkar, Mansi Shah, and Renu Tewari. Cloud analytics: Do we really need to reinvent the storage stack? In *Proceedings of the 1st USENIX Workshop on Hot Topics in Cloud Computing (HOTCLOUD '2009)*, San Diego, CA, USA, June 2009.

[2] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. Plfs: a checkpoint filesystem for parallel applications. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *USENIX OSDI 2006*, Seattle WA, November 2006.

[4] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[5] G. Grider, J. Nunez, and J. Bent. LANL MPI-IO Test. http://institutes.lanl.gov/data/software/, July 2008.

[6] Sun Microsystems. Lustre file system, October 2008.

[7] MPI Forum. Message Passing Interface. http://www.mpi-forum.org/.

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

[9] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *USENIX FAST 2002*, Monterey CA, January 2002.

[10] Konstantin Shvachko, Hairong Huang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *MSST 2010*, Lake Tahoe NV, May 2010.

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

[12] The PRObE Project. Parallel Reconfigurable Observational Environment. http://www.nmc-probe.org/.

[13] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–17, Berkeley, CA, USA, 2008. USENIX Association.