
White Rose Research Online

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk
Universities of Leeds, Sheffield and York

Deposited via The University of York.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/136615/

Version: Accepted Version

Proceedings Paper:
Wright, S. A., Hammond, S. D., Pennycook, S. J. et al. (2012) LDPLFS: Improving I/O performance without application modification. In: Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2012. 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2012, 21-25 May 2012 , CHN, pp. 1352-1359.

https://doi.org/10.1109/IPDPSW.2012.172

Reuse
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.



LDPLFS: Improving I/O Performance Without Application Modification

S. A. Wright*, S. D. Hammond†, S. J. Pennycook*, I. Miller‡, J. A. Herdman‡, S. A. Jarvis*

*Performance Computing and Visualisation
Department of Computer Science
University of Warwick, UK
Email: steven.wright@warwick.ac.uk

†Scalable Computer Architectures/CSRI
Sandia National Laboratories
Albuquerque, NM

‡Supercomputing Solution Centre
UK Atomic Weapons Establishment
Aldermaston, UK

***Abstract*—Input/Output (I/O) operations can represent a significant proportion of run-time when large scientific applications are run in parallel and at scale. In order to address the growing divergence between processing speeds and I/O performance, the Parallel Log-structured File System (PLFS) has been developed by EMC Corporation and the Los Alamos National Laboratory (LANL) to improve the performance of parallel file activities. Currently, PLFS requires the use of either (i) the FUSE Linux Kernel module; (ii) a modified MPI library with a customised ROMIO MPI-IO library; or (iii) an application rewrite to utilise the PLFS API directly.**

In this paper we present an alternative method of utilising PLFS in applications. This method employs a dynamic library to intercept the low-level POSIX operations and retarget them to use the equivalents offered by PLFS. We demonstrate our implementation of this approach, named LDPLFS, on a set of standard UNIX tools, as well on as a set of standard parallel I/O intensive mini-applications. The results demonstrate almost equivalent performance to a modified build of ROMIO and improvements over the FUSE-based approach. Furthermore, through our experiments we demonstrate decreased performance in PLFS when ran at scale on the Lustre file system.

***Index Terms*—Data Storage Systems, File Systems, High Performance Computing, I/O**

I. Introduction

Historically, much of the high performance computing (HPC) industry has focused on the development of methods to improve compute-processing speeds, creating a tendency to measure performance in terms of floating-point operations per second (FLOP/s). The result has been that many other contributors to application performance have developed at a slower rate. An example of this are input and output (I/O) activities which are required to read and store application data and checkpoints. As the performance of I/O systems continues to diverge substantially from compute performance, a number of projects have been initiated to look for software- and hardware-based solutions to address this concern.

One recent project of note is the Parallel Log-structured File System (PLFS) which is being actively developed by EMC Corporation, the Los Alamos National Laboratory (LANL) and their academic and industrial partners [1]. To date, PLFS has been reported to yield large gains in both application read and write performance through the utilisation of two well known principles for improving parallel file system performance: (i) through the use of a *log-structured file system* – where write operations are performed sequentially to the disk regardless of intended file offsets (keeping the offsets in an index structure instead); and (ii) through the use of *file partitioning* – where a write to a single file is instead transparently transposed into a write to many files, increasing the number of available file streams.

Currently PLFS can be deployed in one of three ways: (i) through a File System in Userspace (FUSE) mount point, requiring installation and access to the FUSE Linux Kernel module and its supporting drivers and libraries; (ii) through an MPI-IO file system driver built into the Message Passing Interface (MPI) library; or (iii) through the rewriting of an application to use the PLFS API directly. These methods therefore require either the installation of additional software, recompilation of the MPI application stack (and, subsequently, the application itself) or modification of the application's source code. In HPC centres which have a focus on reliability, or which lack the time and/or expertise to manage the installation and maintenance of PLFS, it may be seen as too onerous to be of use.

In this paper we present an alternative method of using PLFS that avoids the need to rewrite applications, obtain specific file/system access permissions, or modify the applica-

tion stack. Such a method will allow HPC centres to quickly and simply assess the impact of PLFS on their applications and systems. We call this solution 'LDPLFS' since it is dynamically linked (using the Linux linker `ld`) immediately prior to execution, enabling calls to POSIX file operations to be transparently retargeted to PLFS equivalents. This solution requires only a simple environment variable to be exported in order for applications to make use of PLFS – existing compiled binaries, middleware and submission scripts require no modification.

Specifically, this paper makes the following contributions:

- We present LDPLFS, a dynamically loadable library designed to retarget POSIX file operations to functions on PLFS file containers. We demonstrate its use with standard UNIX tools, providing users with an alternative method for extracting raw data from PLFS structures without the need for a FUSE file system;
- We demonstrate the performance of LDPLFS in parallel with respect to: the FUSE mounted alternative, the PLFS ROMIO file system driver and standard MPI-IO file operations without PLFS. Our study shows performance that is near identical to the PLFS ROMIO driver and greater than the PLFS FUSE file system, without the need for FUSE specific permissions;
- Finally, we utilise LDPLFS at scale on the 260 TFLOP/s Sierra cluster located at the Lawrence Livermore National Laboratory (LLNL), utilising two mini-applications designed for file system performance analysis. We show how LDPLFS can improve the performance of applications without requiring any modification to the system's environment or an application's source code. We also demonstrate that on the Lustre file system used by Sierra, PLFS can harm an application's performance at scale, most likely due to a bottleneck being created by the metadata server (MDS).

The remainder of this paper is organised as follows: Section 2 outlines related work in the area of I/O optimisation; Section 3 discusses the experimental setup used in this study, the mechanics of LDPLFS, and an analysis of its performance compared to the FUSE and ROMIO alternatives; Section 4 contains a case study demonstrating the performance gains of LDPLFS in parallel using a number of I/O intensive mini-applications; finally, Section 5 concludes the paper and outlines future work.

II. Background and Related Work

Just as the Message Passing Interface (MPI) has become the *de facto* standard for the development of parallel applications, so too has MPI-IO become the preferred method for handling I/O in parallel [2]. The ROMIO implementation [3] – utilised by OpenMPI [4], MPICH2 [5] and various other vendor-based MPI solutions [6], [7] – offers a series of potential optimisations.

Firstly, *collective buffering* has been demonstrated to yield a significant speed-up, initially on applications writing relatively small amounts of data [8], [9] and more recently on densely packed nodes [10]. These improvements come in the first instance due to larger "buffered" writes which better utilise the available bandwidth and in the second instance due to the aggregation of data to fewer ranks per node, reducing on-node file system contention.

Secondly, *data-sieving* has been shown to be extremely beneficial when utilising file views to manage interleaved writes within MPI-IO [9]. In order to achieve better utilisation of the file system, a large block of data is read into memory before small changes are made at specific offsets. The data is then written back to the disk in a single block. This decreases the number of seek and write operations that need to be performed at the expense of locking a larger portion of the file.

Another approach shown to produce large increases in write bandwidth is the use of so called *log-structured* file systems [11]. When performing write operations, the data is written sequentially to persistent storage regardless of intended file offsets. Writing in this manner reduces the number of expensive seek operations required on I/O systems backed by magnetic disks. In order to maintain file coherence, an index is built alongside the data so that it can be reordered when being read. In most cases this offers a large increase in write performance at the expense of poor read performance.

In the Zest implementation of a log-structured file system [12], the data is written in this manner (via the fastest available path) to a temporary staging area that has no read-back capability. This serves as a transition layer, caching data that is later copied to a fully featured file system at a non-critical time.

As well as writing sequentially to the disk, *file partitioning* has also been shown to produce significant I/O improvements. In [13] and [14], an I/O profiling tool is utilised to guide the transparent partitioning of files written and read by a set of benchmarks. Through segmenting the output into several files spread across multiple disks, the number of available file streams is increased, reducing file contention on the I/O backplane. Furthermore, file locking incurs a much smaller overhead as each process has access to its own unique file.

PLFS from LANL [1] combines file partitioning and a log-structure to improve I/O bandwidth. In an approach that is transparent to an application, a file access from n processors to 1 file is transformed into an access of n processors to n files. The authors demonstrate speed-ups of between $10\times$ and $100\times$ for write performance. Furthermore, due to the increased number of file streams, they report an increased read bandwidth when the data is being read back on the same number of nodes used to write the file [15]. Whilst the log-structured nature of the file system usually decreases the read performance, the use of file partitioning has a much greater effect in this respect on large I/O systems.

As discussed, PLFS can currently be used in one of three ways, each with advantages and disadvantages.

Firstly, the use of the FUSE kernel module and device allows PLFS to be utilised by any user, but due to passing data

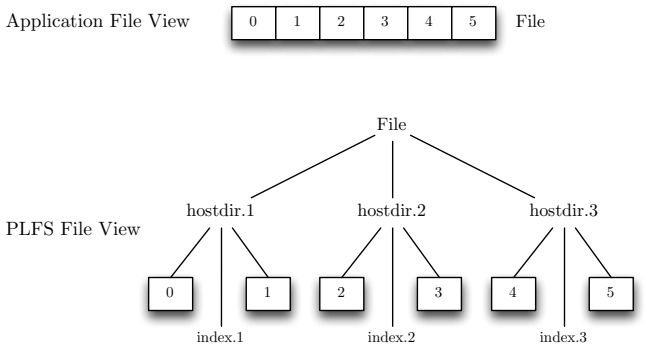


Fig. 1: An applications view of a file and the underlying PLFS container structure.

```
int open(const char *filename, int flags, mode_t mode);
int plfs_open(Plfs_fd *fd, const char *filename, int flags,
        pid_t pid, mode_t mode, Plfs_open_opt *open_opt);

ssize_t write(int fd, const void *buf, size_t count);
ssize_t plfs_write(Plfs_fd *plfsfd, const void *buf,
        size_t count, off_t offset, pid_t pid);

ssize_t read(int fd, void *buf, size_t count);
ssize_t plfs_read(Plfs_fd *plfsfd, void *buf,
        size_t count, off_t offset);
```

Listing 1: Open, Read and Write functions from the POSIX and PLFS API.

in and out of the kernel, may produce the worst performance of the three options. It may also introduce several known security issues, such as non-privileged access to block devices, or privilege escalation. Furthermore, the installation of FUSE also requires administrative privileges which may not be available on shared systems (such as the Sierra supercomputer used in Section IV).

Secondly, PLFS can be used by building a modified version of the MPI library to include the PLFS ROMIO MPI-IO driver. This method strikes a balance in performance between the FUSE module and an application rewrite. However, recompilation of the MPI library may introduce performance issues in other areas, since optimisations found in the system's MPI installation may not be present in a modified build.

Finally, an application can be rewritten to use the PLFS API directly. This has the potential to produce the greatest performance at the expense of application redevelopment and recompilation. However, neither a modified MPI library or an application rewrite will allow users to view PLFS files as if they were single files. Any applications that do not use MPI or the PLFS API will not be able to load PLFS containers. This may make the output of visualisation dumps, through PLFS, inaccessible to the user.

In this paper we aim to offer an alternative approach that operates much like the MPI-IO ROMIO driver without the need to recompile the application stack. Furthermore, our approach can take advantage of advanced MPI-IO features, such as collective buffering and data-sieving, that are not available when using the PLFS API directly.

III. Performance Analysis

A. LDPLFS

PLFS is a virtual file system that makes use of file partitioning and a log-structure to improve the performance of parallel file operations. Each file within the PLFS mount point appears to an application as though it is a single file; PLFS, however, creates a container structure, with a data file and an index for each process or compute node. This provides each process with its own unique file stream, increasing the available bandwidth, as file writes do not need to be serialised [10]. Figure 1 demonstrates how a six rank (two processes per rank) execution would view a single file and how it would be stored within the PLFS backend directory.

LDPLFS is a dynamic library specifically designed to interpose POSIX file functions and retarget them to PLFS equivalents. By utilising the Linux loader, LDPLFS overloads many of the POSIX file symbols (*e.g.* `open`, `read`, `write`), causing an augmented implementation to be executed at runtime[1]. This allows existing binaries and application stacks to be used without the need for recompilation. For systems where dynamic linking is either not available or is only available in a limited capacity (such as on an IBM BlueGene system), a static LDPLFS library can be compiled and, through the use of the `-wrap` functionality found in some compilers, can be linked at compile time.

Due to the difference in semantics between the POSIX and PLFS APIs, LDPLFS must perform two essential bookkeeping tasks. Firstly, LDPLFS must return a valid POSIX file descriptor to the application, despite PLFS utilising an alternative structure to store file properties. Secondly, as the PLFS API requires an explicit offset to be provided, LDPLFS must maintain a file pointer for each PLFS file. Listing 1 shows three POSIX functions and their PLFS equivalents.

When a file is opened from within a pre-defined PLFS mount point, a PLFS file descriptor (`Plfs_fd`) pointer is created and the file is opened with the `plfs_open` function (using default settings for `Plfs_open_opts` and the value of `getpid()` for `pid_t`). In order to return a valid POSIX file descriptor (`fd`) to the application, a temporary file (in our case `/dev/random`) is also opened. The file descriptor of the temporary file is then stored in a look-up table and related to the `Plfs_fd` pointer. Future POSIX operations on a particular `fd` will then either be passed onto the POSIX API, or if a look-up entry exists, the PLFS library.

In order to provide the correct file offset to the PLFS functions, a file pointer is maintained through `lseek()` operations on the temporary POSIX file descriptor. When a POSIX operation is to be performed on a PLFS container,

[1]Note that although LDPLFS makes use of the `LD_PRELOAD` environmental variable in order to be dynamically loaded, other libraries can also make use of the dynamic loader (by appending multiple libraries into the environmental variable), allowing tracing tools to be used in alongside LDPLFS.

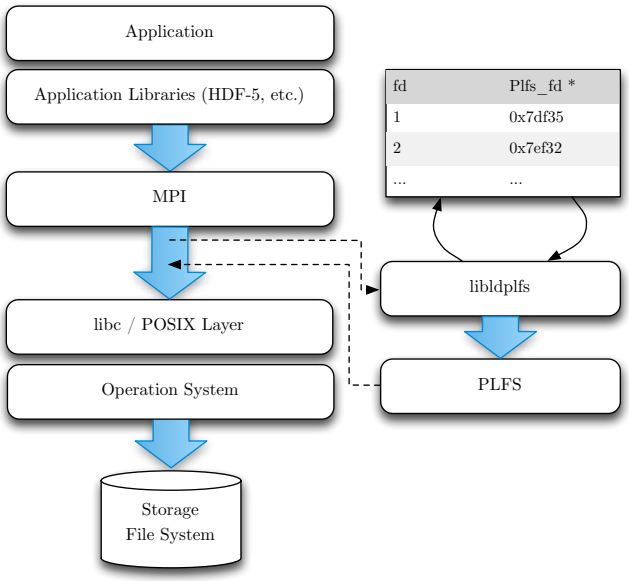


Fig. 2: The control flow of LDPLFS in an applications execution.

the current offset of the temporary file is established (through a call to `lseek(fd, 0, SEEK_CUR)`), a PLFS operation is performed (again using `getpid()` where needed), and then finally, the temporary file pointer is updated (once again through the use of `lseek()`). Figure 2 shows the control flow of an application when using LDPLFS.

B. Testing Platform

LDPLFS has been tested and utilised on two production-grade supercomputers: *Minerva*, located at the Centre for Scientific Computing (CSC) at the University of Warwick, and *Sierra*, located at the Open Computing Facility (OCF) at LLNL. Both machines consist of dual Intel "Westmere" hex-core processors, clocked at 2.66 GHz and 2.8 GHz respectively, and a QLogic QDR InfiniBand interconnect.

Minerva consists of 258 nodes and has a peak LINPACK performance of approximately 30 TFLOP/s. The I/O back-end for Minerva uses IBM's General Parallel File System (GPFS) [16] and consists of two servers. 96 disks, configured in RAID-6, are used for data storage and 24 disks, configured in RAID-10, are used for the storage of metadata. The theoretical peak bandwidth is approximately 4 GB/s, but the performance is heavily constrained by the relatively small number and slow speed of the 2 TB hard disk drives.

Sierra is a much larger machine, with 1,849 compute nodes. It has a LINPACK performance of 260 TFLOP/s and is backed by LLNL's "islanded I/O" (where many file systems are shared by multiple machines). For this study we utilise the *lscratchc* Lustre file system [17], using 24 I/O servers and a dedicated MDS. The system uses 3,600 hard disk drives, running at 10,000 RPM. The theoretical peak bandwidth of the file system is approximately 30 GB/s, with the limiting factor being the InfiniBand interconnect to the file servers.

	Minerva	**Sierra**
Processor	Intel Xeon 5650	Intel Xeon 5660
CPU Speed	2.66 GHz	2.8 GHz
Cores per Node	12	12
Nodes	258	1,849
Interconnect	QLogic TrueScale 4X QDR	InfiniBand
File System	GPFS	Lustre
I/O Servers / OSS	2	24
Theoretical Bandwidth	∼4 GB/s	∼30 GB/s
Storage Disks		
Number of Disks	96	3,600
Disk Type	2 TB	450 GB
Disk Speed	7,200 RPM	10,000 RPM
Bus Type	Nearline SAS	SAS
Raid Level	6 (8 + 2)	6 (8 + 2)
Metadata Disks		
Number of Disks	24	30 (+2) [a]
Disk Type	300 GB	147 GB
Disk Speed	15,000 RPM	15,000 RPM
Bus Type	SAS	SAS
Raid Level	10	10

[a] Sierra's MDS uses 32 disks; two configured in RAID-1 for journalling data, 28 disks configured in RAID-10 for the data volume itself and a further two disks to be used as hot spares.

TABLE I: Benchmarking platforms used in this study.

The specification for each machine is summarised in Table I.

C. Performance Analysis

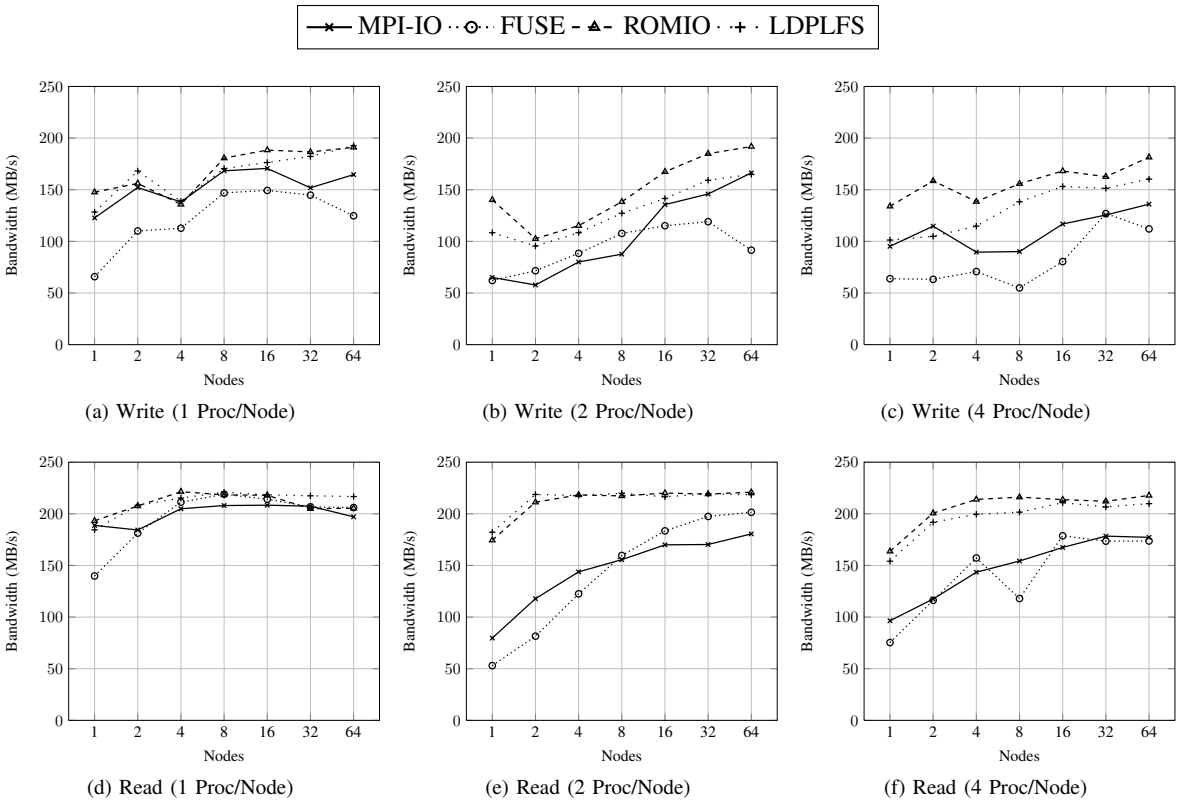
Our initial assessment of LDPLFS is conducted on Minerva. We utilise the MPI-IO Test application [18] from LANL, writing a total of 1 GB per process in 8 MB blocks. Collective blocking MPI-IO operations are employed with tests utilising PLFS through the FUSE kernel library, the ROMIO PLFS driver and LDPLFS. In all cases we use OpenMPI version 1.4.3 and PLFS version 2.0.1. We compare the achieved bandwidth figures against those from the default MPI-IO library without PLFS.

Tests have been conducted on 1, 2, 4, 8, 16, 32 and 64 compute nodes utilising 1, 2 and 4 processors per node[2]. We note that each run is conducted with collective buffering enabled and in its default configuration[3] in order to provide better performance with minimal configuration changes. The node-wise performance should remain largely consistent, while the number of processors per node is varied – in each case there remains only one process on each node performing the file system write. As the number of processors per node is increased, an overhead is incurred because of the presence of on-node communication and synchronisation.

Figure 3 demonstrates promising results, showing that LDPLFS performs almost as well as PLFS through ROMIO and significantly better than FUSE (up to 2×) in almost all cases. It is interesting to note that on occasion, LDPLFS performs better than the PLFS ROMIO driver. This performance difference may be due to a combination of background

[2] Due to machine usage limits, using all 12 processors per node would limit our results to a maximum of 16 compute nodes, decreasing the scalability of our results.

[3] The default collective buffering behaviour is to allocate a single aggregator per *distinct* compute node.



(a) Write (1 Proc/Node) (b) Write (2 Proc/Node) (c) Write (4 Proc/Node)

(d) Read (1 Proc/Node) (e) Read (2 Proc/Node) (f) Read (4 Proc/Node)

Fig. 3: Benchmarked MPI-IO bandwidths on FUSE, ROMIO, LDPLFS and standard MPI-IO (without PLFS).

load on the file system, optimisations in the standard MPI-IO routines and the reduced overhead incurred by LDPLFS over the PLFS ROMIO driver equivalent. On the Minerva cluster, FUSE performs worse than standard MPI-IO by 20% on average for parallel writes. While the overhead of FUSE is addressed in [1], the I/O set-up used in that study is much larger than that used by Minerva, and makes use of custom optimised hardware.

D. Standard UNIX Tools

One of the current difficulties associated with the practical use of PLFS is the complexity associated with managing PLFS containers. Since FUSE treats a PLFS mount point as a self-contained file system, using the files in any application is trivial. However, when using either of the alternative solutions for PLFS, applications must either use MPI or be rewritten for PLFS. PLFS files appear inside the "backend" directory as directories with hundreds of files. Visualising data or post processing the information output becomes difficult in this scenario; this is one of the problems LDPLFS aims to address. As LDPLFS operates at the POSIX call level, it can be used with any standard UNIX tools as well as parallel science and engineering applications.

Table II presents the performance of several standard UNIX tools operating on a PLFS container of 4 GB in size. Note that the file copy (`cp`) times correspond to copying a file from a PLFS container to a standard UNIX file and vice versa. These

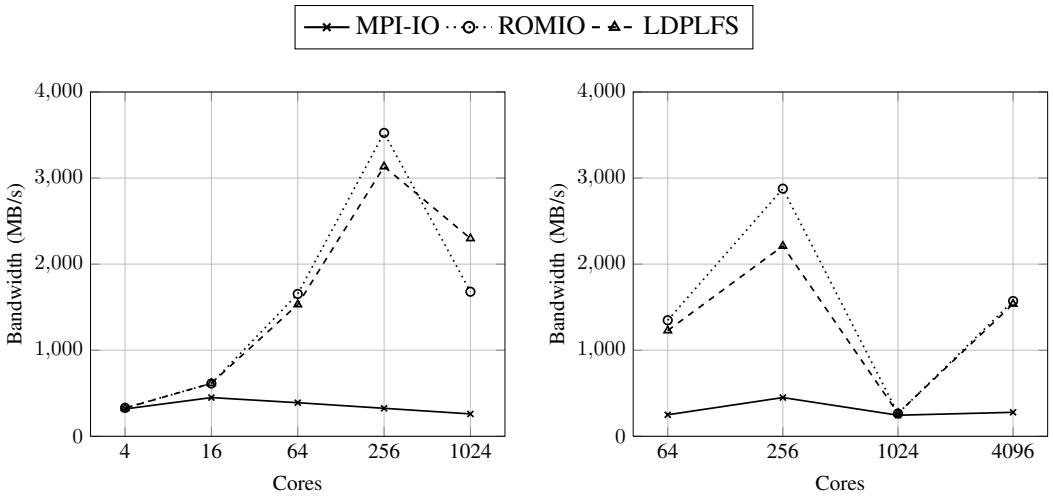
	PLFS Container	**Standard UNIX File**
`cp` (read)	100.713	114.279
`cp` (write)	107.587	
`cat`	25.186	25.433
`grep`	130.662	128.863
`md5sum`	26.970	26.781

TABLE II: Time in seconds for UNIX commands to complete using PLFS through LDPLFS, and without PLFS.

can be compared to a single time for copying from and to a standard UNIX file.

Since each of these commands are serial applications, each command was executed on the login node of Minerva. It is promising to see that the time for each of the commands to complete is largely the same for both standard UNIX files and PLFS container structures. These results show that PLFS is marginally faster when copying to or from a PLFS file than a normal UNIX file. We attribute this improvement in performance to the increased number of file streams available, improving the bandwidth achievable from the file servers.

Our results position LDPLFS as a viable solution to improving the performance of I/O in parallel, as well as showing that there is no substantial performance hit when using LDPLFS to interact with PLFS mount points using serial (non-MPI) applications. We next demonstrate the performance of LDPLFS at much larger scale, using a set of I/O intensive mini-applications.



(a) Problem Class C

(b) Problem Class D

Fig. 4: BT benchmarked MPI-IO bandwidths using MPI-IO, as well as PLFS through ROMIO and LDPLFS.

IV. Case Study

Figure 3 shows that the performance of PLFS on Minerva is approximately $2\times$ greater than that of MPI-IO without PLFS in parallel. Because of the relatively small I/O set-up employed by Minerva, we do not believe it is possible to achieve the same levels of speed-up seen in [1], where a high-end PanFS I/O solution is used. In order to better demonstrate how PLFS and LDPLFS perform on a much more substantial I/O set-up, we have used two applications to benchmark the *lscratchc* file system attached to Sierra.

For this study we utilise the Block Tridiagonal solver application (BT) from the NAS Benchmark Suite [19], [20] and the FLASH-IO [21], [22] mini-application. For BT we use the C problem class ($162 \times 162 \times 162$), writing a total of 6.4 GB of data during an execution, and the D problem class ($408 \times 408 \times 408$), writing a total of 136 GB of data. The application is strong scaled – as the number of processor cores is increased, the global problem size remains the same, with each process operating over a smaller sub-problem. For the C problem class, the global problem size is relatively small, and can only be scaled to 1,024 processors before the local problem size becomes too small to operate on correctly. Conversely for the D program class, the global problem size is so large that on less than 64 processors, the execution time becomes prohibitive. For this reason we use between 4 and 1,024 processor cores for the C problem class, and between 64 and 4,096 processor cores for the D problem class.

Figures 4 and 5 show the achieved bandwidth for the two mini-applications in their default configurations using the system's pre-installed OpenMPI version 1.3.4 library (without PLFS), as well as with the system's OpenMPI library augmented with LDPLFS, and finally with the ROMIO PLFS file system driver compiled into a customised build of OpenMPI version 1.4.3. The performance of PLFS through the two methods is largely the same, with a slight divergence for BT.

Since LDPLFS retargets POSIX file operations transparently and uses various structures in memory to maintain file consistency, a change in the local problem size may effect the LDPLFS performance due to the memory access patterns changing and additional context switching. Furthermore, write caching can produce a large difference in performance – where data is small enough to fit in cache, the write of that data to disk can be delayed.

Write caching is most prevalent in the BT application where, at large-scale, small amounts of data are being written by each process during each parallel write step. For the C problem class (Figure 4(a)), 6.4 GB of data is written in 20 separate MPI write calls, causing approximately 300 KB of data to be written by each process at each step. When writing to a single file, the file server must make sure that writes are completed before allowing other processes to write to the file. This causes each write command to wait on all other processes, leading to relatively poor performance. Conversely, through PLFS, each process writes to its own file, therefore allowing the write to be cleared to cache almost instantly.

In Figure 4(b), the performance rapidly decreases at 1,024 cores, where each process is writing approximately 136 MB, in 20 steps. We believe these writes (of approximately 7 MB each) are marginally too large for the system's cache, causing performance that is equivalent to vanilla MPI-IO. However, when using 4,096 cores, each write is less than 2 MB per process, writing only 34 MB per process during the execution. This causes the write caching effects seen in Figure 4(a) to reappear.

FLASH-IO is a synthetic benchmark that recreates the checkpointing behaviour of the FLASH thermonuclear simulation code [23], [24]. In this study we weak scale the problem, with a local problem size of $24 \times 24 \times 24$. This causes each process to write approximately 205 MB to the disk, through the HDF-5 library [25]. Runs were conducted on between 1 node and 256 nodes, using all 12 processors each time, thus utilising up to 3,072 processors. We note that as the number

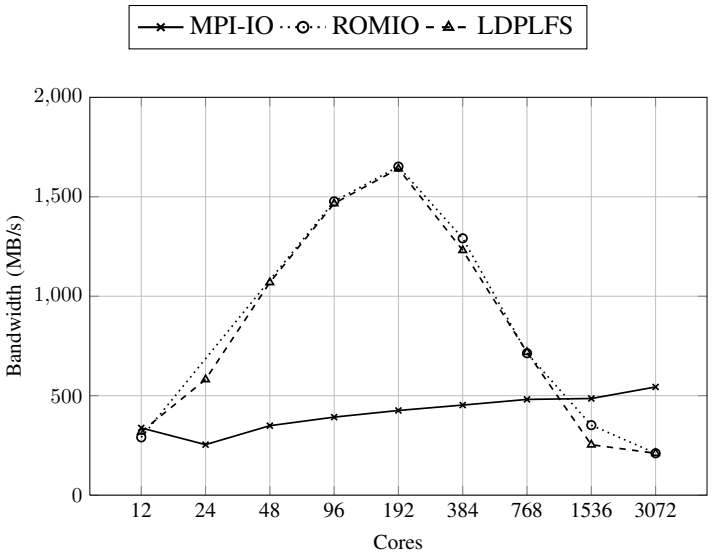


Fig. 5: FLASH-IO benchmarked MPI-IO bandwidths using MPI-IO, as well as PLFS through ROMIO and LDPLFS.

of compute nodes is increased, so too is the output file size. Since each process is writing the same total amount of data, over the same number of time steps, caching effects will be less prevalent in weak scaled problems.

Interestingly, Figure 5 shows that as the core count is increased on FLASH-IO, the write speed of MPI-IO gently increases up to approximately 550 MB/s. However, when using PLFS we see a sharp increase in write speed until 192 cores (or 16 nodes), at which point the average write speed reaches approximately 1,650 MB/s, before decreasing to 210 MB/s at 3,072 cores. A possible explanation for this is that since the Lustre file system uses a dedicated MDS, as the number of processors is increased, the performance plateaus and then decreases due to the MDS becoming a bottleneck in the system. Since PLFS operates using multiple files per processor (at least one for the data and one for the index), it uses many more files as the problem is scaled, potentially putting a large load on the MDS. This bottleneck is less evident in the BT mini-application due to the small write sizes, which facilitates simple write caching. On a file system like GPFS, where metadata is distributed, these performance decreases may not materialise.

V. Conclusion

File I/O operations have, in many cases, been one of the last aspects considered during application optimisation. In this paper we have presented LDPLFS – a dynamic, runtime loadable plug-in for PLFS which offers the opportunity to accelerate file read and write activities without modification to the machine's environment or an application's source code.

Specifically we have demonstrated the performance of our LDPLFS solution in comparison to PLFS using the FUSE Linux kernel module, PLFS using the ROMIO MPI-IO file system driver and the original MPI-IO operations without PLFS. In this comparison LDPLFS is able to offer approximately equivalent performance to using PLFS through the ROMIO file system driver and improved performance over FUSE.

LDPLFS not only allows end-users to improve their applications I/O performance, but also allows users to quickly evaluate the benefits of PLFS on their system before undertaking the task of library rebuilds or code modifications to use PLFS natively.

In the second part of this paper we used LDPLFS at scale to accelerate the I/O operations of the FLASH-IO and BT mini-applications. We have shown that ROMIO with PLFS and LDPLFS can offer significant improvements in I/O performance – up to 20× – when compared to the original unmodified applications. Furthermore, we have demonstrated that while PLFS may seem like a quick-fix solution to improving I/O performance, its use can actually harm performance at scale and under certain conditions, due to the overhead of managing hundreds or thousands of files in parallel.

LDPLFS is a solution which requires only two small pieces of software to be built with no system administrator actions. The library is loadable from only a single environment variable, yet is able to offer significant improvement in parallel I/O activity. In our work with industry partners, such a solution helps to address concerns which may arise over the security model of FUSE and the significant investment associated with the recompilation of applications using a custom MPI-IO/ROMIO middleware. LDPLFS therefore straddles the gap between offering improved application performance and the effort associated with the installation of traditional PLFS.

A. Future Work

In future work we intend to create an alternative implementation of PLFS that can operate on IBM BlueGene systems. We feel that this platform is of interest to many research laboratories and therefore PLFS could help improve the performance of its unusual I/O setup (where all I/O operations are "function shipped" to dedicated I/O nodes).

Through utilising an alternative implementation of PLFS, we aim to investigate the low-level performance effects of a log-based file system and file partitioning in isolation. Furthermore, we aim to make our implementation much more customisable, in order to correct the negative effects seen at scale in Figure 5. We also intend to model the performance of our implementation in order to aid auto-optimisation of parameters, as well as assess the benefits of PLFS on future I/O backplanes without requiring extensive benchmarking.

We hope to use our performance model to highlight systems where PLFS may have a negative effect on performance, where perhaps using just file partitioning or a log-based file system will provide greater performance.

ACKNOWLEDGMENTS

This work is supported in part by The Royal Society through their Industry Fellowship Scheme (IF090020/AM). The performance modelling research is also supported jointly by AWE and the TSB Knowledge Transfer Partnership, grant number KTP006740.

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

The authors would like to thank Matt Ismail and Tony Arber in the Centre for Scientific Computing at the University of Warwick for access to the Minerva supercomputer.

Access to the LLNL OCF is made possible through collaboration with the UK Atomic Weapons Establishment under grants CDK0660 (The Production of Predictive Models for Future Computing Requirements) and CDK0724 (AWE Technical Outreach Programme).

We are grateful to Scott Futral, Todd Gamblin, Jan Nunes and the Livermore Computing Team for access to, and help in using, the Sierra machine located at LLNL. We are also indebted to John Bent at EMC Corporation, and Meghan Wingate McClelland at the Los Alamos National Laboratory for their expert advice and support concerning PLFS.

REFERENCES

[1] J. Bent *et al.*, "PLFS: A Checkpoint Filesystem for Parallel Applications," in *Proceedings of the ACM/IEEE Supercomputing Conference (SC'09)*. Portland, OR: ACM, New York, NY, November 2009, pp. 21:1–21:12.

[2] Message Passing Interface Forum, "MPI2: A Message Passing Interface Standard," *High Performance Computing Applications*, vol. 12, no. 1–2, pp. 1–299, 1998.

[3] R. Thakur, E. Lusk, and W. Gropp, "ROMIO: A High-Performance, Portable MPI-IO Implementation," Mathematics and Computer Science Division, Argonne National Laboratory, Tech. Rep. ANL/MCS-TM-234, 1997.

[4] E. Gabriel *et al.*, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Recent Advances in Parallel Virtual Machines and Message Passing Interface*, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds., vol. 3241. Springer-Verlag, Berlin, September 2004, pp. 97–104.

[5] W. Gropp, "MPICH2: A New Start for MPI Implementations," *Lecture Notes in Computer Science (LNCS)*, vol. 2474, pp. 7–42, 2002.

[6] G. Almási *et al.*, "Implementing MPI on the BlueGene/L Supercomputer," in *European Conference on Parallel and Distributed Computing 2004 (Euro-Par'04)*, D. L. M. Danelutto, M. Vanneschi, Ed., vol. 3149. Pisa, Italy: Springer-Verlag, Berlin, August–September 2004, pp. 833–845.

[7] Bull, *BullX Cluster Suite Application Developer's Guide*, Les Clayes-sous-Bois, Paris, April 2010.

[8] B. Nitzberg and V. Lo, "Collective Buffering: Improving Parallel I/O Performance," in *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing (HPDC'97)*. Portland, OR: IEEE Computer Society, Washington, DC, August 1997, pp. 148–157.

[9] R. Thakur, W. Gropp, and E. Lusk, "Data-Sieving and Collective I/O in ROMIO," in *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation (FRONTIERS'99)*. Annapolis, MD: IEEE Computer Society, Los Alamitos, CA, February 1999, pp. 182–191.

[10] S. A. Wright, S. D. Hammond, S. J. Pennycook, and S. A. Jarvis, "Light-weight Parallel I/O Analysis at Scale," *Lecture Notes in Computer Science (LNCS)*, vol. 6977, pp. 235–249, October 2011.

[11] M. Polte *et al.*, "Fast Log-based Concurrent Writing of Checkpoints," in *Proceedings of the 3rd Annual Petascale Data Storage Workshop (PDSW'08)*. Austin, TX: IEEE Computer Society, Los Alamitos, CA, November 2008, pp. 1–4.

[12] P. Nowoczynski, N. Stone, J. Yanovich, and J. Sommerfield, "Zest Checkpoint Storage System for Large Supercomputers," in *Proceedings of the 3rd Annual Petascale Data Storage Workshop (PDSW'08)*. Austin, TX: IEEE Computer Society, Los Alamitos, CA, November 2008, pp. 1–5.

[13] Y. Wang and D. Kaeli, "Profile-guided I/O Partitioning," in *Proceedings of the 17th Annual International Conference on Supercomputing (ICS'03)*. San Francisco, CA: ACM, New York, NY, June 2003, pp. 252–260.

[14] Y. Wang and D. Kaeli, "Source Level Transformations to Improve I/O Data Partitioning," in *Proceedings of the 1st International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI'03)*. New Orleans, LA: ACM, New York, NY, September 2003, pp. 27–35.

[15] M. Polte *et al.*, "... And Eat It Too: High Read Performance in Write-Optimized HPC I/O Middleware File Formats," in *Proceedings of the 4th Annual Petascale Data Storage Workshop (PDSW'09)*. Portland, OR: ACM, New York, NY, November 2009, pp. 21–25.

[16] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST'02)*. Monterey, CA: USENIX Association Berkeley, CA, January 2002, pp. 231–244.

[17] P. Schwan, "Lustre: Building a File System for 1000-node Clusters," http://lustre.org (accessed October 23, 2011), 2003.

[18] J. Nunez and J. Bent, "MPI-IO Test User's Guide," http://institutes.lanl.gov/data/software/ (accessed February 21, 2011), 2011.

[19] D. H. Bailey *et al.*, "The NAS Parallel Benchmarks," NASA Ames Research Center, Tech. Rep. RNR-94-007, March 1994.

[20] D. H. Bailey *et al.*, "The NAS Parallel Benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[21] M. Zingale, "FLASH I/O Benchmark Routine – Parallel HDF 5," http://www.ucolick.org/~zingale/flash_benchmark_io/ (accessed February 21, 2011), 2011.

[22] Argonne National Laboratory, "Parallel I/O Benchmarking Consortium," http://www.mcs.anl.gov/research/projects/pio-benchmark/ (accessed February 21, 2011), 2011.

[23] B. Fryxell *et al.*, "FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes," *The Astrophysical Journal Supplement Series*, vol. 131, no. 1, p. 273, 2000.

[24] R. Rosner *et al.*, "Flash Code: Studying Astrophysical Thermonuclear Flashes," *Computing in Science & Engineering*, vol. 2, no. 2, pp. 33–41, 2000.

[25] Q. Koziol and R. Matzke, *HDF5 – A New Generation of HDF: Reference Manual and User Guide*, Champaign, Illinois, USA, 1998.