
Proceedings Paper:
Wright, Steven A. orcid.org/0000-0001-7133-8533 and Jarvis, Stephen A. (2015) Quantifying the Effects of Contention on Parallel File Systems. In: Proceedings - 2015 IEEE 29th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2015. 29th IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2015, 25-29 May 2015 IEEE , IND , pp. 932-940.

https://doi.org/10.1109/IPDPSW.2015.8

Quantifying the Effects of Contention on Parallel File Systems

Steven A. Wright, Stephen A. Jarvis
Performance Computing and Visualisation
Department of Computer Science
University of Warwick, UK
Email: steven.wright@warwick.ac.uk

***Abstract*—As we move towards the Exascale era of supercomputing, node-level failures are becoming more commonplace; frequent checkpointing is currently used to recover from such failures in long-running science applications. While compute performance has steadily improved year-on-year, parallel I/O performance has stalled, meaning checkpointing is fast becoming a bottleneck to performance. Using current file systems in the most efficient way possible will alleviate some of these issues and will help prepare developers and system designers for Exascale; unfortunately, many domain-scientists simply submit their jobs with the default file system configuration.**

In this paper, we analyse previous work on finding optimality on Lustre file systems, demonstrating that by exposing parallelism in the parallel file system, performance can be improved by up to 49×. However, we demonstrate that on systems where many applications are competing for a finite number of object storage targets (OSTs), competing tasks may reduce optimal performance considerably. We show that reducing each job's request for OSTs by 40% decreases performance by only 13%, while increasing the availability and quality of service of the file system. Further, we present a series of metrics designed to analyse and explain the effects of contention on parallel file systems. Finally, we re-evaluate our previous work with the Parallel Log-structured File System (PLFS), comparing it to Lustre at various scales. We show that PLFS may perform better than Lustre in particular configurations, but that at large scale PLFS becomes a bottleneck to performance. We extend the metrics proposed in this paper to explain these performance deficiencies that exist in PLFS, demonstrating that the software creates high levels of self-contention at scale.

***Keywords*-Data storage systems; File servers; File systems; High performance computing; Optimization; Performance analysis; Supercomputers;**

I. Introduction

Historically, the optimisation of I/O performance has been the responsibility of application developers, configuring their own software to achieve the best performance – a responsibility that has often been ignored. Software solutions to achieving better performance, such as custom-built MPI-IO drivers that target specific file systems, are often not installed by default. For instance, on the systems installed at the Open Compute Facility (OCF) at the Lawrence Livermore National Laboratory (LLNL), an optimised Lustre-specific driver is not installed despite the software being readily available with most implementations of the Message Passing Interface (MPI) library.

Many previously published works show optimised middlewares outperforming MPI-IO in its default configuration considerably, and make no attempt to compare the software to an optimised MPI-IO installation [1]–[3]. In this paper, an MPI library with a custom Lustre driver is built and utilised on the Cab machine (see Section III for details) at LLNL. The research presented by Behzad et al. [4], [5] suggests that performance can be improved by as much as two orders of magnitude using this Lustre-specific driver with tuned parameters.

In this paper we perform a parameter sweep in order to find a more optimal Lustre configuration for a small test with IOR. Through this, we demonstrate a performance improvement of up to 49×. While performance is vastly improved by tweaking the Lustre settings, optimal performance for one application can reduce the quality of service (QoS) provided to other users on a large shared system. This paper investigates these potential issues, demonstrating that with the introduction of contention, optimal performance may be reduced considerably. Further, we demonstrate that reducing the demand on resources increases QoS at relatively little expense to performance. Finally, this paper compares the approach of Behzad et al. to the Parallel Log-structured File System (PLFS) from the Los Alamos National Laboratory (LANL) and EMC^2, showing that PLFS at scale acts much like several contended jobs.

Specifically this paper makes the following contributions:

- We utilise the Lustre MPI driver and an exhaustive parameter search to find an optimal Lustre configuration for a small I/O intensive test application, demonstrating a 49× performance improvement over the default MPI-IO performance. We then demonstrate the effect of this optimised configuration on a contended file system. Specifically, we show that with four simultaneous I/O intensive applications, each using the previously discovered *optimal* configuration, the performance for each task is decreased by a factor of 4. Reducing the amount of resources demanded by each task is shown to improve the system's availability, while having a minimal effect on each job's performance;
- We introduce a series of metrics designed to predict the possible effect of additional contention on Lustre

file systems. With these equations we can predict the average load of the file system's Object Storage Targets (OSTs) and thus quantify the level of contention being experienced by competing jobs. Additionally, we use a benchmark to predict how a single OST behaves under contention. Our analysis suggests that for three simultaneous tasks or more, OST contention begins to produce a noticeable performance overhead;

- Finally, we re-evaluate our previous work with PLFS in order to explain the deficiencies experienced at scale [6]. By extending our contention metrics to describe PLFS-based applications, we demonstrate that at scale PLFS creates high levels of self-contention on Lustre file systems.

The remainder of this paper is structured as follows: Section 2 summarises related work in the area of I/O optimisation and parallel file systems; Section 3 outlines the system used in the experiments in this paper; Section 4 describes our search for an optimal Lustre configuration for a simple I/O benchmark application; Section 5 shows how contended jobs may affect the performance achieved when using previously discovered tuned Lustre configurations, providing metrics and results that demonstrate that when contended, jobs can request lesser resources to achieve a similar level of performance; Section 6 extends this analysis to PLFS, showing that at scale PLFS introduces heavy self-contention; Section 7 concludes this paper.

II. Related Work

As supercomputers have grown in compute power, so too have they grown in complexity, size and component count. With the push towards Exascale computing (estimated by 2022 at the time of writing [7]), the explosion in machine size will result in an increase in component failures. To combat these reliability issues, long running scientific simulations require checkpointing to reduce the impact of a node failure. Periodically, during a time consuming calculation, the system's state is written out to persistent storage so that in the event of a crash, the application can be restarted and computation can be resumed with a minimal loss of data. Writing out this data to a parallel file system is fast becoming a bottleneck to science applications at scale.

Just as MPI has become the *de facto* standard for the development of parallel applications, so too has MPI-IO become the preferred method for coordinating parallel I/O [8]. The ROMIO implementation [9], common to OpenMPI [10], MPICH2 [11] and various other vendor-based MPI solutions [12], [13], offers a series of potential optimisations. Using an abstract API such as MPI-IO allows optimisations to be made to particular implementations. Specifically, ROMIO implements collective buffering [14] and data-sieving [15] in order to increase the potential bandwidth available. Through understanding the I/O demands of an application, and tuning which of these optimisations to make use of, modest performance improvements can be achieved [16], [17].

In addition to file system agnostic optimisations, ROMIO provides an Abstract Device I/O Interface (ADIO) [18] for developers to implement custom drivers for their file systems. Many supercomputers today utilise the Lustre file system [19], which splits files into blocks that are then distributed across a number of object storage targets (OSTs). While Lustre does provide a POSIX-compliant interface, the `ad_lustre` ADIO driver allows users to specify MPI-IO hints in order to control the data layout of a file [20].

Using the configuration parameters made available by the `ad_lustre` ADIO driver, Behzad et al. [4], [5] have demonstrated a $100\times$ increase in performance over using the system's stock configuration options, something that is all too often left unchanged. Similar performance improvements have been reported by Lind [21] and You et al. [22] suggesting that configuring the Lustre file system correctly prior to its use can produce a considerable boost in performance.

However, in a system with a small number of OSTs and a large number of concurrent jobs, optimising performance in the manner suggested by Behzad [5], Lind [21] and You [22] may induce large amounts of contention due to overlapping OSTs. It is this issue that this paper aims to address.

Another approach to handling substandard I/O in parallel applications has been to design *virtual file systems*. PLFS [1] is one such example that was developed at LANL and 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 [23].

Although PLFS has been shown to provide increases in performance that are in-line with those of using `ad_lustre` with optimal configuration options, it has also been shown that at large scale, PLFS may harm performance, such that it performs worse than even an unoptimised MPI-IO installation (using the UNIX file system, POSIX-compliant driver `ad_ufs`) [6].

This paper analyses the potential performance of the Lustre ADIO driver over the standard UFS driver in similar way to previous studies [4], [5], [21], [22]. However, none of these works consider the impact of background I/O load upon file system performance – this is an important consideration on large multi-user systems and this paper addresses this concern. We also focus on the work of Bent et al. [1] demonstrating how using PLFS at scale behaves much like a heavily contented file system.

Cab		
Processor		Intel Xeon E5-2670
CPU Speed		2.6 GHz
Cores per Node		16
Memory per Node		32 GB
Nodes		1,200
Interconnect		QLogic TrueScale 4× QDR InfiniBand
File System		Lustre 2.4.2
I/O Servers		32
Theoretical Bandwidth		≈30 GB/s
	Storage	Metadata
Number of Disks	4,800	30 (+2)[a]
Disk Size	450 GB	147 GB
Spindle Speed	10,000 RPM	15,000 RPM
RAID Configuration	Level 6 (8 + 2)	Level 1+0

Table I: Configuration for the *lscratchc* Lustre File System installed at LLNL.

[a]The MDS used by OCF's *lscratchc* file system uses 32 disks: two configured in RAID-1 for journalling data, 28 disks configured in RAID-1+0 for the data volume itself and a further two disks to be used as hot spares.

III. Computing Platforms

For the experiments in this paper, the results were all collected on the Cab supercomputer installed in the OCF at LLNL. Cab is a Cray-built Xtreme-X cluster with 1,200 batch nodes, each containing two oct-core Xeon E5-2670 processors clocked at 2.6 GHz. An InfiniBand fat-tree connects each of the nodes. Cab is connected to LLNL's islanded I/O network, which provides Lustre (version 2.4.2) file systems to a number of large clusters. For all experiments, the Intel compiler (version 13.0) and OpenMPI (version 1.4.3) were used. The experiments with PLFS were performed using OpenMPI 1.4.3 built with the PLFS 2.0.1 ADIO driver. More information can be found in Table I.

IV. Effective Use of Uncontended Parallel File Systems

As we have previously demonstrated [6], [17], the large parallel file systems connected to some of the most powerful supercomputers in the world are currently being under utilised – partially due to a lack of available software drivers, partially due to lack of optimisation in the applications themselves. The work presented by Behzad et al. [5] shows how using the Lustre-specific MPI-IO driver (`ad_lustre`) distributed with most MPI implementations can lead to performance improvements of up to 100× over the default installation. The authors utilise a genetic algorithm to search the parameter space for an optimal configuration [5], varying the stripe factor (the number of OSTs to use), the stripe size, the number of collective buffering nodes and the collective buffer size (as well as some HDF-5 specific options).

Using the same approach as Behzad et al. (but with reduced complexity due to the absence of a genetic algorithm), we performed a parameter sweep on a small IOR execution running on 64 nodes ($64 \times 16 = 1,024$ cores). The configuration used for IOR can be found in Table II. The collective buffer size was set to the default value (16 MB) and each node contributed one collective buffer process, meaning there was a total of 64 buffering processes. To reduce the search space, a linear search was conducted with a stripe count between 1 and 160 (as there is a 160 OST limit in the Lustre version 2.4.2, which is used on OCF machines) and a stripe size between 1 and 256 MB.

Option	Value
API	MPI-IO
Write file	On
Read file	Off
Block Size (bytes)	4 MB
Transfer Size (bytes)	1 MB
Segment Count	100

Table II: IOR configuration options for experiments.

The results of this parameter search are shown in Figure 1. Using the default Lustre configuration (stripe count = 2, stripe size = 1 MB), the application achieves an average bandwidth of 313 MB/s. Through varying the stripe size, performance can be increased from this baseline bandwidth up to 395 MB/s, and by varying the stripe count performance can be increased much further, up to a maximum of 4,075 MB/s.

Figure 1 shows that through varying both parameters the maximum bandwidth is found when using 160 stripes of size 128 MB; performance increases from the baseline 313 MB/s, up to 15,609 MB/s, representing a 49× improvement in write performance. This result largely echoes those reported by previously [5], where the greatest performance is usually found by striping across the maximum number of OSTs and writing stripes that are a multiple of the application's I/O block size.

That the optimal performance is found when exploiting the maximum amount of available parallelism may seem obvious, but on many systems, it is simply not possible to achieve this without a rebuilt software stack. Exploiting a larger proportion of the available file servers and storage targets may be optimal on a quiet system, however when there are many tasks requiring I/O simultaneously, the performance may decrease to the OST contention,

V. Quantifying the Performance of Contended File Systems

On a multi-user system, with limited resources, using a large percentage of the OSTs available may be detrimental to the rest of the system. The *lscratchc* file system used in this paper exposes 480 OSTs to the user. The assignment of OSTs to files is done at file creation time, with targets assigned at random (based on current usage, to maintain an approximately even capacity). This suggests that three jobs,

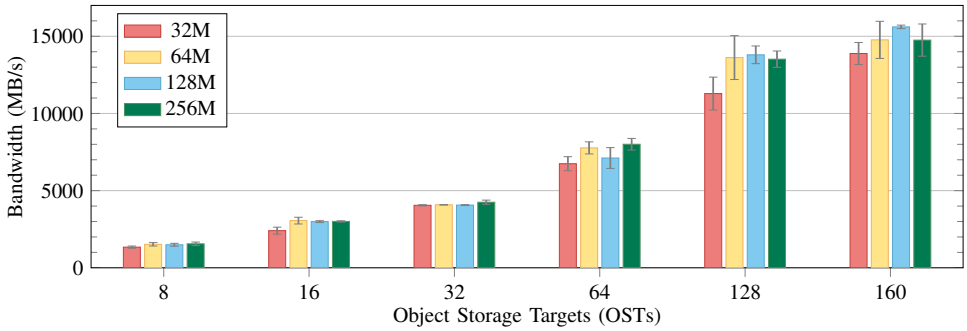

Figure 1: Write bandwidth achieved over 1,024 processors by varying both the stripe count and the stripe size.

each using 160 OSTs, would fully occupy the file system if the assignment had no overlaps. However, as OSTs are assigned randomly, for two jobs (of which the first uses $160/480$ of the available OSTs) approximately one third of the OSTs assigned to the second job will also be in use by the first job.

$$D_{\text{inuse}}(n) = D_{\text{inuse}}(n-1) + \left(r_j - \frac{D_{\text{inuse}}(n-1)}{D_{\text{total}}} r_j \right) \quad (1)$$

If each job ($j \in \{1 \ldots n\}$) requests r_j OSTs, the total number of OSTs in use (D_{inuse}) after each job starts is described by Equation 1, where $D_{\text{inuse}}(0) = 0$. Each time a new job starts, the number of OSTs in use increases by the size of the request, minus the average number of OST collisions that occur. If each job is requesting the same number of resources (R) – which may be the case if a parameter sweep has determined that the optimal configuration is when the maximum number of OSTs are used – then the number of OSTs in use can be simplified to:

$$D_{\text{inuse}} = D_{\text{total}} - \left(D_{\text{total}} \times \left(1 - \frac{R}{D_{\text{total}}} \right)^n \right) \quad (2)$$

With these two equations the average load of each OST (D_{load}) can be calculated, for any particular workload, by taking the number of stripes requested in total, and dividing it by the number of OSTs in use. A load of 1 would imply that each OST is, on average, only in use by a single job, whereas a higher number would indicate that there are a number of collisions on some OSTs, potentially resulting in a job switching overhead.

$$D_{\text{req}} = R \times n \quad (3)$$

$$D_{\text{load}} = \frac{D_{\text{req}}}{D_{\text{inuse}}} \quad (4)$$

Table III demonstrates this for the *lscratchc* file system where each job is requesting the previously discovered optimal number of stripes (160). With 10 simultaneous

$D_{\text{total}} = 480$, $R = 160$			
Jobs	D_{inuse}	D_{req}	D_{load}
1	160.00	160	1.00
2	266.67	320	1.20
3	337.78	480	1.42
4	385.19	640	1.66
5	416.79	800	1.92
6	437.86	960	2.19
7	451.91	1120	2.48
8	461.27	1280	2.78
9	467.51	1440	3.08
10	471.68	1600	3.39

Table III: The average number of OSTs in use and their average load based on the number of concurrent I/O intensive jobs with 160 stripes requested by each job.

$D_{\text{total}} = 480$, $R = 64$			
Jobs	D_{inuse}	D_{req}	D_{load}
1	64.00	64	1.00
2	119.47	128	1.07
3	167.54	192	1.15
4	209.20	256	1.22
5	245.31	320	1.30
6	276.60	384	1.39
7	303.72	448	1.48
8	327.22	512	1.57
9	347.59	576	1.66
10	365.25	640	1.75

Table IV: The average number of OSTs in use and their average load based on the number of concurrent I/O intensive jobs with 64 stripes requested by each job.

I/O intensive jobs each using 160 OSTs, an average of 4 collisions will occur on each OST, though a small subset of OSTs may well incur all 10 potential collisions (and some may incur none), reducing the performance of the file system for every job. Table IV shows that by reducing the size of the stripe requests to 64, the OST load is decreased significantly, possibly avoiding many of the bottlenecks associated with OST contention.

In order to ascertain how the OSTs in the *lscratchc* file system behave under contention, a study was undertaken using a custom-written benchmark that creates a split communicator that therefore allows each process to read and

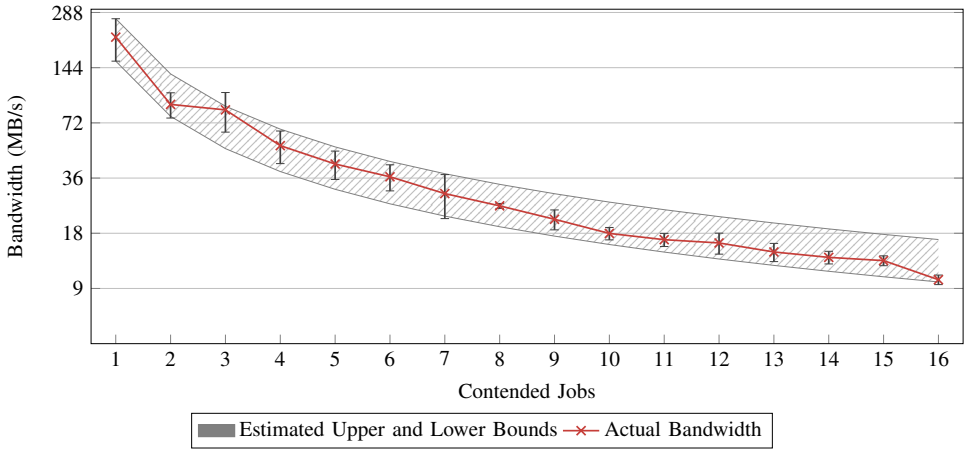

Figure 2: The performance per-processor of the *lscratchc* file system under contention, with the ideal upper and lower bounds.

write its own file in a single MPI application. The benchmark opens a number of files, with the same Lustre configuration (a single 1 MB stripe). Using the `stripe_offset` MPI hint, the OST to use is specified such that every rank writes to its own file that is stored on the same target. Figure 2 shows the per-process bandwidth achieved with a varying number of contended file writes.

In Figure 2, the shaded area indicates idealised scaling behaviour, where the upper and lower limits are calculated from the 95% confidence intervals from the single job experiment and scaled linearly; as *lscratchc* is already a shared-user file system, there is some variance in performance with no *forced* contention. The graph shows that, as the number of jobs is increased, the performance diverges from the top of the ideal scaling line, illustrating the performance degradation associated with high OST load.

To investigate this on the whole file system, a job was submitted to Cab that created four identical IOR executions each running simultaneously with the configuration stated in Table II. Each task utilised 64 nodes (1,024 processes) and thus the total job consumed 4,096 cores, and the MPI hints were specified according to the previously discovered optimal values (Figure 1). As can be seen in Figure 3, each individual application achieved approximately 4,500 MB/s – a 3.44× reduction from the peak value (15,609 MB/s) seen in Figure 1.

Using the mean of five experiments, Table V and Figure 4 demonstrate how reducing the number of stripes per job increases the OST availability to the rest of the system while having a minimal effect on performance. Using as few as 32 stripes per file, the average bandwidth achieved by each of the four applications is 3,500MB/s, but by Equation 2, only 115 OSTs will be in use in the average case, providing an average OST load of ≈1.1.

Furthermore, Table V shows that when using a stripe

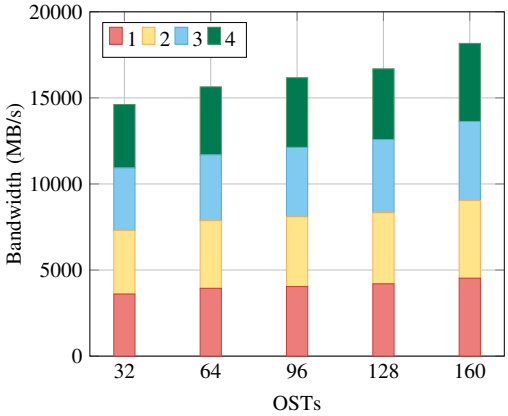

Figure 4: Graphical representation of the data in Table V, showing optimal performance at 160 stripes per file, but very minor performance degradation at just 32 stripes per file.

count of 160, there are 42 OSTs that are being contended by 3 of the 4 jobs and there are 7 OSTs being contended by all 4. By reducing the demand to 64 stripes, the performance is reduced by ≈14% while the number of OSTs in use is reduced by ≈37%, leaving more resources available for a larger number of tasks, while also reducing the number of collisions significantly. Although it is unlikely that many simultaneous jobs will request such a large number of OSTs each, we seek only to demonstrate a worst case scenario and demonstrate the potentially harmful nature of auto tuning without consideration for the QoS of a shared file system.

Although the optimal performance on *lscratchc*, with four competing tasks, is still found using the maximum number of stripes allowed, the bandwidth achieved is almost a quarter of the previously achieved maximum. On file systems where there are less OSTs (such as those used by Behzad et al. [5]), any job contention will decrease the achievable

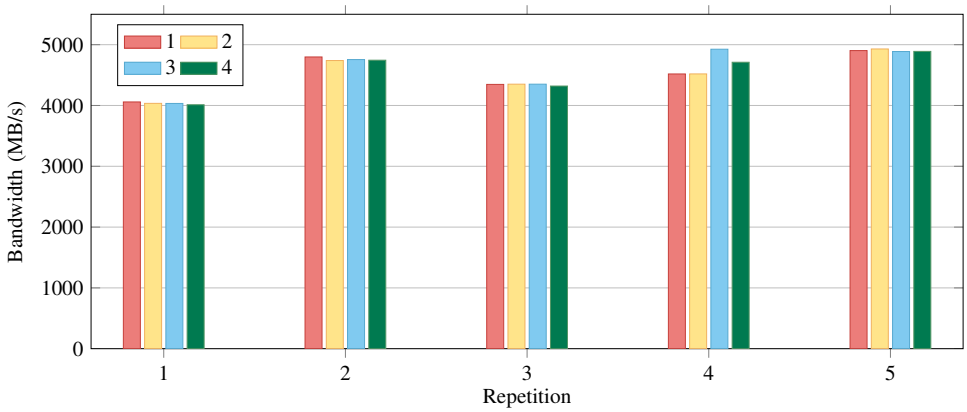

Figure 3: Performance of each of 4 tasks over 5 repetitions where all tasks are contenting the file system.

R	Average Bandwidth	Total Bandwidth	D_{req}	OST Usage 1	2	3	4	Predicted D_{inuse}	Predicted D_{load}	Actual D_{inuse}	Actual D_{load}
32	3654.06	14616.24	128	103.2	11.2	0.8	0.0	115.76	1.11	115.20	1.11
64	3910.51	15642.03	256	172.6	35.8	3.4	0.4	209.20	1.22	212.20	1.21
96	4042.98	16171.92	384	199.4	76.4	9.8	0.6	283.39	1.36	286.20	1.34
128	4172.17	16688.66	512	211.6	111.4	22.4	2.6	341.18	1.50	348.00	1.47
160	4541.37	18165.46	640	191.8	147.0	41.8	7.2	385.19	1.66	387.80	1.65

Table V: Average and total bandwidth achieved across four tasks for a varying stripe size request, along with values for the average number of tasks competing for 1, 2, 3 and 4 OSTs respectively.

$D_{\text{total}} = 160$, $R = 128$			
Jobs	D_{inuse}	D_{req}	D_{load}
1	128.00	128	1.00
2	153.60	256	1.67
3	158.72	384	2.42
4	159.74	512	3.21
5	159.95	640	4.00
6	159.99	768	4.80
7	160.00	896	5.60
8	160.00	1024	6.40
9	160.00	1152	7.20
10	160.00	1280	8.00

Table VI: Predicted OST usage and average load for the Stampede I/O setup described in [5] with contended jobs.

performance and may be detrimental to the rest of the system. To demonstrate this further, the equations presented in this paper have been applied to the configuration of the Stampede supercomputer [5]. Table VI shows our predicted OST load for Stampede's file system using the optimal stripe count found by Behzad et al. for the VPIC-IO application (128 stripes on a file system with 58 OSSs and 160 OSTs). Our analysis demonstrates that with only three simultaneous tasks with a similar I/O demand, the OSTs on Stampede could be in use by as many as two or three simultaneous tasks in the average case. This may potentially cause a significant performance degradation.

VI. The Influence of PLFS on Contention

In our previous work PLFS was shown to produce a noticeable performance increase on LLNL systems under certain conditions [6], [17]. However, this paper has already demonstrated that the performance gap is reduced when using a tuned Lustre-specific MPI-IO driver. Furthermore, some of our previous work has shown that at scale, PLFS performs worse than even the unoptimised UFS MPI-IO file system driver (`ad_ufs`) [6].

Figure 5 shows the performance of the *lscratchc* file system when running the IOR problem described in Table II on Cab. PLFS operates by creating a separate data and index file for each rank, in directories controlled by a hashing function; this increases the number of file streams available and consequently increases the number of Lustre stripes in use. As the files are written by PLFS through POSIX file system calls, each file is created with the system default configuration of two 1 MB stripes per file (unless otherwise specified using the `lfs` control program).

It should be noted that as PLFS creates a large number of files, with randomly placed stripes, there is a larger variance in PLFS performance. An execution running with 256 processes will create 256 data files, requiring 512 stripes. Experimentally, this produces an average OST load of 1.58 and a bandwidth between 3,329.9 MB/s and 11,539.4 MB/s (average 7,126.9 MB/s). Conversely, through the Lustre driver, the variance is much lower as at most 160 stripes will be created with no collisions between OSTs. Due to

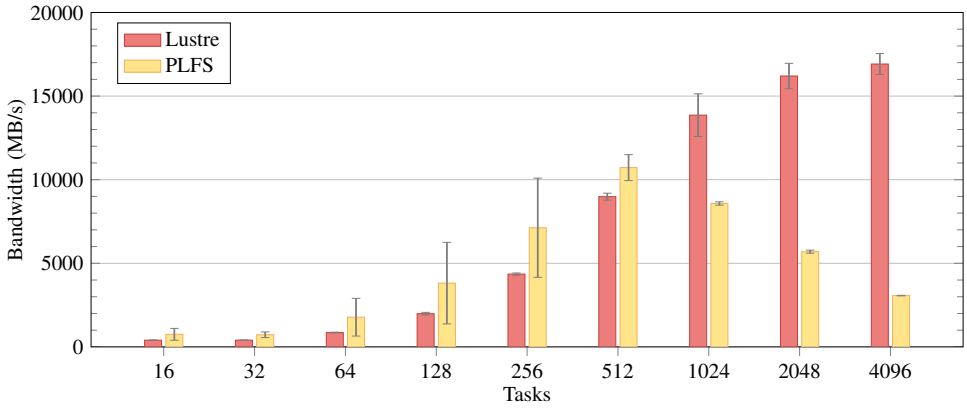

Figure 5: Achieved write bandwidth achieved for IOR through an *optimised* Lustre configuration and through the PLFS MPI-IO driver.

	ad_lustre		ad_plfs	
Processors	B/W	95% CI	B/W	95% CI
16	403.75	(390.73, 416.77)	752.96	(398.41, 1107.51)
32	404.71	(393.09, 416.34)	727.33	(558.95, 895.70)
64	857.35	(832.82, 881.88)	1776.70	(648.90, 2904.50)
128	1987.51	(1908.24, 2066.78)	3814.62	(1375.19, 6254.05)
256	4354.98	(4288.69, 4421.27)	7126.88	(4159.66, 10094.10)
512	8985.14	(8777.61, 9192.66)	10723.42	(9947.06, 11499.77)
1024	13859.58	(12582.68, 15136.47)	8575.13	(8474.06, 8676.21)
2048	16200.16	(15441.57, 16958.74)	5696.41	(5604.86, 5787.97)
4096	16917.11	(16291.58, 17542.64)	3069.05	(3052.82, 3085.28)

Table VII: Numeric data for Figure 5, showing the performance of IOR through Lustre and PLFS.

background machine noise it is difficult to know what the load is on each OST at any given time, but generally PLFS performs better when the number of OSTs experiencing a high number of collisions is minimised.

Table VIII shows the number of OST collisions in the PLFS backend directory for the 512 cores case – the highest core count for which PLFS produces a speed-up over the `ad_lustre` driver. At 512 cores, the performance of PLFS reaches its peak before Lustre begins to provide better performance. Although PLFS has been designed to target unaligned accesses, the scaling issues in PLFS will still be present at scale due to the number of files being created. However, unaligned accesses may be detrimental to the performance of Lustre.

Using the equations introduced earlier to quantify the number of OST collisions for contended jobs, but amending the equations to deal with the contention created by PLFS we can analyse the average OST load for PLFS. To modify the equations, we instead treat each rank as a separate task with 2 stripes (i.e. $R = 2$) and set the number of tasks (n) to the number of ranks in use. This mimics the behaviour of PLFS on a single application run (where n files are created, instead of a single file).

$$D_{\text{inuse}} = D_{\text{total}} - \left(D_{\text{total}} \times \left(1 - \frac{2}{D_{\text{total}}} \right)^{n} \right) \quad (5)$$

	Experiment				
Collisions	1	2	3	4	5
0	121	135	122	116	129
1	134	126	134	129	133
2	97	88	85	94	82
3	49	55	56	45	54
4	21	22	21	20	28
5	6	6	6	12	2
6	1	1	2	1	1
7	0	0	0	0	1
8	0	0	0	1	0
D_{inuse}	429	433	426	418	430
D_{load}	2.39	2.36	2.40	2.45	2.38
BW (MB/s)	12062.68	10469.38	10234.97	9768.07	11081.99

Table VIII: Stripe collision statistics for PLFS backend directory running with 512 processors.

$$D_{\text{load}} = \frac{2 \times n}{D_{\text{inuse}}} \quad (6)$$

With these equations we can show that, while PLFS may provide a small-scale fix, it inevitability overwhelms a Lustre file system at higher core counts. Using Equations 5 and 6, at 512 cores on the *lscratchc* file system there is an average of 2.4 tasks using each OST; by 688 cores, there are 3 tasks per OST (which is shown in Figure 2 to still provide "good" performance). At 2,048 and 4,096 cores, the number of collisions reaches 8.53 and 17.06

Collisions	Experiment 1	2	3	4	5
0	0	0	0	0	0
1	0	0	0	0	0
2	1	0	0	0	0
3	0	0	1	0	0
4	0	0	0	0	0
5	0	1	1	1	0
6	1	2	2	2	4
7	2	4	2	10	2
8	8	7	5	3	7
9	9	10	13	16	15
10	15	13	21	18	18
11	26	18	30	21	25
12	33	38	34	37	29
13	48	46	36	33	37
14	45	48	38	40	48
15	28	33	45	51	46
16	51	49	32	44	46
17	42	42	46	41	36
18	30	35	34	29	33
19	44	46	39	34	29
20	28	21	27	20	25
21	24	18	21	22	26
22	17	14	14	12	17
23	10	9	14	11	10
24	6	12	4	8	9
25	1	3	8	11	7
26	5	5	8	9	5
27	4	5	4	3	2
28	0	0	1	1	3
29	2	1	0	2	0
30	0	0	0	0	0
31	0	0	0	0	0
32	0	0	0	1	0
33	0	0	0	0	0
34	0	0	0	0	0
35	0	0	0	0	1
D_{inuse}	480	480	480	480	480
D_{load}	17.07	17.07	17.07	17.07	17.07
BW (MB/s)	3042.06	3077.16	3083.26	3084.89	3057.90

Table IX: Stripe collision statistics for PLFS backend directory running with 4,096 processors.

respectively, which begins to saturate the file system and decreases performance not just for the host application, but for all other users of the file system too. Tables VIII and IX show the observed collision statistics for two of these application configurations. Table IX specifically shows that at 4,096 cores, most OSTs are being used for the stripe data of between 10 and 23 files and in one of the five repeated experiments, a single OST is being used by 35 competing ranks, placing a large overhead on its potential performance.

VII. Conclusion

In this paper it has been shown that current-day I/O systems perform much better than some literature suggests [1]–[3]. However, this level of performance can only be achieved if the file system is being used correctly. Behzad et al. [4], [5] suggest that on Lustre file systems, a good level of performance can be found by adjusting particular file layout settings. They show that, with an optimised configuration, the file system scales almost linearly. However, due to restrictions in the version of Lustre used for the experiments in this paper, the maximum number of OSTs that can be used is only 160; this suggests that when problems are scaled up to hundreds of thousands of nodes (as may be the case for Exascale), the I/O performance will not scale. Particular versions of Lustre already scale beyond this OST limit [24], but they are not currently being used by some of the biggest supercomputing centres (such as the OCF at LLNL).

We have utilised an exhaustive search algorithm to perform a parameter sweep to find an optimal configuration for the Lustre file system connected to the Cab supercomputer. After finding an optimal configuration for a small IOR problem, this paper analyses the effect of I/O contention on the achievable bandwidth. A series of metrics have also been proposed that aim to quantify the contention that is created by several jobs competing for this shared resource. Section V shows that when jobs are run with the optimal configuration on a contented resource, performance drops considerably, and using less resources vastly improves system availability with a minor performance degradation.

We have previously explored the work of LANL and EMC^2 in creating PLFS, which has been shown to provide significant speed-ups at mid-scale [6], [17]. However, this paper shows that PLFS may be harmful to performance on Lustre file systems at large-scale. The work in this paper provides an explanation for the performance degradation experienced previously, within the framework of the Lustre contention metrics provided.

Using the equations given, the load of each OST can be calculated for both competing I/O intensive applications and for PLFS-based applications. With the results from these equations, various file system purchasing decisions can be made; for instance, the number of OSTs can be increased in order to reduce the OST load for a theoretically "average" I/O workload. Furthermore, the benefits PLFS may have on an application can be calculated based on the scale at which it will be run, as well as on the number of OSTs available for the task.

While, at the time of writing, the I/O backplanes in modern day systems are being under-utilised, with the correct configuration options and some optimisation by application developers, acceptable performance can be achieved with relatively little effort. Making these changes to applications and file system configurations will not only improve current scientific applications, but will also benefit future systems and inform future I/O system developers on how to best proceed towards Exascale-class storage.

References

[1] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A Checkpoint Filesystem for Parallel Applications," in *Proceedings of the 21st ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'09)*. Portland, OR: ACM, New York, NY, November 2009, pp. 21:1–21:12.

[2] J. Cope, M. Oberg, H. M. Tufo, and M. Woitaszek, "Shared Parallel Filesystems in Heterogeneous Linux Multi-Cluster Environments," in *Proceedings of the 6th LCI International Conference on Linux Clusters*. Chapel Hill, NC: Linux Clusters Institute, NM, 2005, pp. 1–21.

[3] Y. Chanel, "Study of the Lustre File System Performances Before its Installation in a Computing Cluster," http://upcommons.upc.edu/pfc/bitstream/2099.1/5626/1/49964.pdf (accessed June 20, 2011), 2008.

[4] B. Behzad, S. Byna, S. M. Wild, and M. Snir, "Improving Parallel I/O Autotuning with Performance Modeling," Argonne National Laboratory, Argonne, IL, Tech. Rep. ANL/MCS-P5066-0114, January 2014.

[5] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir, "Taming Parallel I/O Complexity with Auto-tuning," in *Proceedings of the 25th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)*. Denver, CO: IEEE Computer Society, Washington, DC, November 2013, pp. 1–12.

[6] S. A. Wright, S. D. Hammond, S. J. Pennycook, I. Miller, J. A. Herdman, and S. A. Jarvis, "LDPLFS: Improving I/O Performance without Application Modification," in *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium Workshops & PhD Forum (IPDPSW'12)*. Shanghai, China: IEEE Computer Society, Washington, DC, 2012, pp. 1352–1359.

[7] Department of Energy, "Department of Energy Exascale Strategy – Report to Congress," United States Department of Energy, Washington, DC 20585, Tech. Rep., June 2013.

[8] Message Passing Interface Forum, "MPI: A Message Passing Interface Standard Version 2.2," *High Performance Computing Applications*, vol. 12, no. 1–2, pp. 1–647, 2009.

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

[10] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," *Lecture Notes in Computer Science (LNCS)*, vol. 3241, pp. 97–104, September 2004.

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

[12] G. Alamási, C. Archer, J. G. Castaños, C. C. Erway, P. Heidelberger, X. Martorell, J. E. Moreira, K. Pinnow, J. Ratterman, N. Smeds, B. Steinmacher-burrow, W. Gropp, and B. Toonen, "Implementing MPI on the BlueGene/L Supercomputer," *Lecture Notes in Computer Science (LNCS)*, vol. 3149, pp. 833–845, August–September 2004.

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

[14] B. Nitzberg and V. M. 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.

[15] 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.

[16] 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.

[17] S. A. Wright, S. D. Hammond, S. J. Pennycook, R. F. Bird, J. A. Herdman, I. Miller, A. Vadgama, A. H. Bhalerao, and S. A. Jarvis, "Parallel File System Analysis Through Application I/O Tracing," *The Computer Journal*, vol. 56, no. 2, pp. 141–155, February 2013.

[18] R. Thakur, W. Gropp, and E. 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 (FRONTIERS'96)*. Annapolis, MD: IEEE Computer Society, Los Alamitos, CA, October 1996, pp. 180–187.

[19] P. Schwan, "Lustre: Building a File System for 1,000-node Clusters," in *Proceedings of the Linux Symposium*. Ottawa, Ontario, Canada: The Linux Symposium, July 2003, pp. 380–386.

[20] P. M. Dickens and J. Logan, "A High Performance Implementation of MPI-IO for a Lustre File System Environment," *Concurrency and Computation: Practice & Experience*, vol. 22, no. 11, pp. 1433–1449, August 2010.

[21] B. Lind, "Lustre Tuning Parameters," in *Proceedings of the 2013 Lustre User Group (LUG'13)*. San Diego, CA: OpenSFS, LUG, April 2013, pp. 1–12.

[22] H. You, Q. Liu, Z. Li, and S. Moore, "The Design of an Auto-Tuning I/O Framework on Cray XT5 System," in *Proceedings of the 2011 Cray Users Group Conference (CUG'11)*. Fairbanks, AK: Cray Inc., May 2011, pp. 1–10.

[23] M. Polte, J. F. Lofstead, J. Bent, G. Gibson, S. A. Klasky, Q. Liu, M. Parashar, K. Schwan, and M. Wolf, "... And Eat It Too: High Read Performance in Write-Optimized HPC I/O Middleware File Formats," in *Proceedings of the 4th Annual Workshop on Petascale Data Storage (PDSW'09)*. Portland, OR: ACM, New York, NY, November 2009, pp. 21–25.

[24] O. Drokin, "Lustre File Striping Across a Large Number of OSTs," in *Proceedings of the 2011 Lustre User Group (LUG'13)*. Orlando, FL: OpenSFS, LUG, April 2011, pp. 1–17.