
Proceedings Paper:
Wright, Steven A. orcid.org/0000-0001-7133-8533, Hammond, Simon D., Pennycook, Simon J. et al. (1 more author) (2011) Light-weight parallel I/O analysis at scale. In: Computer Performance Engineering - 8th European Performance Engineering Workshop, EPEW 2011, Proceedings. 8th European Performance Engineering Workshop, EPEW 2011, 12-13 Oct 2011 Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). , GBR, pp. 235-249.

https://doi.org/10.1007/978-3-642-24749-1_18

Light-weight Parallel I/O Analysis at Scale

S. A. Wright, S. D. Hammond, S. J. Pennycook, and S. A. Jarvis

Performance Computing and Visualisation
Department of Computer Science
University of Warwick, UK
{saw, sdh, sjp, saj}@dcs.warwick.ac.uk

Abstract. Input/output (I/O) operations can represent a significant proportion of the run-time when large scientific applications are run in parallel. Although there have been advances in the form of file-format libraries, file system design and I/O hardware, a growing divergence exists between the performance of parallel file systems and compute processing rates.

In this paper we utilise RIOT, an input/output tracing toolkit being developed at the University of Warwick, to assess the performance of three standard industry I/O benchmarks and mini-applications. We present a case study demonstrating the tracing and analysis capabilities of RIOT at scale, using MPI-IO, Parallel HDF-5 and MPI-IO augmented with the Parallel Log-structured File System (PLFS) middle-ware being developed by the Los Alamos National Laboratory.

Keywords. Input/Output, Message Passing Interface, Parallel I/O, Parallel Log-structured File System, Profiling

1 Introduction

The substantial growth in supercomputer machine size – over two orders of magnitude in terms of processing element count since 1993 – has created machines of extreme computational power and scale. As a result, users of these machines have been able to create increasingly sophisticated and complex computational simulations, advancing scientific understanding across multiple domains. Historically, industry and academia have focused on the development of scalable parallel algorithms – the cornerstone of large parallel applications. 'Performance' has become a measure of the number of calculation operations that can be performed each second.

One of the consequences of this focus has been that some of the vital contributors to application run-time have been developed at a much slower rate. One such area has been that of input/output (I/O) – typically seen as somewhat of a black-box which creates a need to read data at the start of a run and write state information on completion.

As well as reading and writing state information at the beginning and end of computation, the use of checkpointing is becoming common-place – where the

system state is periodically written to persistent storage so that, in the case of an application fault, the computation can be reloaded and resumed. The cost of checkpointing is therefore a slowdown at specific points in the application in order to achieve some level of resilience. As we look to the future, the size of multi-petaflop clusters looks set to bring reliability challenges from just the sheer number of components – checkpointing will become a vital initial tool in addressing these problems. Understanding the cost of checkpointing and what opportunities might exist for optimising this behaviour presents a genuine opportunity to improve the performance of parallel applications at significant scale.

The Message Passing Interface (MPI) has become the *de facto* standard for managing the distribution of data and process synchronisation in parallel applications. The MPI-2 [13] standard introduced MPI-IO, a library of functions designed to standardise the output of data to the file system in parallel. The most widely adopted MPI-IO implementation is ROMIO [21] which is used by both OpenMPI [9] and MPICH2 [10], as well as a number of vendor-based MPI solutions [1, 5].

In addition to the standardisation of parallel I/O through MPI, many file format libraries exist to further abstract low-level I/O operations such as data formatting from the application. Through the use of libraries such as HDF-5 [11], NetCDF [17] and Parallel NetCDF [12], information can be shared between multiple applications using the standardised formatting they create. Optimisations can also be made to a single library, creating improvements in the data throughput of many dependant applications. Unfortunately this has, in part at least, created a lack of responsibility on the part of the code designers to investigate the data storage operations used in their applications. The result has been poor I/O performance that does not utilise the full potential of expensive parallel disk systems.

In this paper we utilise the RIOT input/output toolkit (referred to throughout the remainder of this paper by the recursive acronym RIOT) introduced in [26] to demonstrate the I/O behaviours of three standard benchmarks at scale. RIOT is a collection of tools specifically designed to enable the tracing and subsequent analysis of application input/output activity. This tool is able to trace parallel file operations performed by the ROMIO layer and relate these to their underlying POSIX file operations. We note that this style of low-level parameter recording permits analysis of I/O middleware, file format libraries and application behaviour, all of which are assessed in a case study in Section 4.

The specific contributions of this work are the following:

- We extend previous work in [26] to demonstrate RIOT working at scale on a 261.3 TFLOP/s Intel Westmere-based machine located at the Open Computing Facility (OCF) at the Lawrence Livermore National Laboratory (LLNL);
- We apply our tracing tool to assessing the I/O behaviour of three standard industry benchmarks – the block-tridiagonal (BT) solver application from NASA's Parallel Benchmark Suite, the FLASH-IO benchmark from

the University of Chicago and the Argonne National Laboratory and IOR, a high-performance computing (HPC) file system benchmark which is used during procurement and file system assessment [19, 20]. The variety of configurations (including the use of MPI-IO, parallel HDF-5 [11] and MPI-IO utilising collective buffering hints) available makes the results obtained from our analysis of interest to a wider audience;

- We utilise RIOT to produce a detailed analysis of HDF-5 write behaviour for the FLASH-IO benchmark, demonstrating the significant overhead incurred by data read-back during parallel writes;
- Finally, through I/O trace analysis, we provide insight into the performance gains reported by the Parallel Log-structured File System (PLFS) [4, 16] – a novel I/O middleware being developed by the Los Alamos National Laboratory (LANL) to improve file write times. This paper builds upon [26] to show how file partitioning reduces the time POSIX write calls spend waiting for access to the file system.

The remainder of this paper is structured as follows: Section 2 outlines previous work in the fields of I/O profiling and parallel I/O optimisation; Section 3 describes how parallel I/O is performed with MPI and HDF-5, and how RIOT captures the low-level POSIX operations; Section 4 contains a case study describing the use of RIOT in assessing the parallel input/output behaviours of three industry I/O benchmarking codes; finally, Section 5 concludes the paper and outlines opportunities for future work.

2 Related Work

The assessment of file system performance, either at procurement or during installation and upgrade, has seen the creation of a number of benchmarking utilities which attempt to characterise common read/write behaviour. Notable tools in this area include the BONNIE++ benchmark, developed for benchmarking Linux file systems, as well as the IOBench [25] and IOR [19] parallel benchmarking applications. Whilst these tools provide a good indication of potential maximum performance, they are rarely indicative of the true behaviour of production codes due to the subtle nuances that production grade software contains. For this reason, a number of mini-application benchmarks have been created which extract file read/write behaviour from larger codes to ensure a more accurate representation of performance. Examples include the Block Tridiagonal solver application from NASA's Parallel Benchmark Suite [2] and the FLASH-IO [18] benchmark from the University of Chicago – both of which are used in this paper.

Whilst benchmarks may provide a measure of file system performance, their use in diagnosing problem areas or identifying optimisation opportunities within large codes is limited. For this activity profiling tools are often required which can record read/write behaviour in parallel. One approach, which aims to ascertain the performance characteristics of production-grade scientific codes, is to

intercept communications between the application and the underlying file system. This is the approach taken by RIOT, Darshan [6], developed at the Argonne National Laboratory, and the Integrated Performance Monitoring (IPM) suite of tools [8], from the Lawrence Berkeley National Laboratory (LBNL).

Darshan has been designed to record file accesses over a prolonged period of time, ensuring each interaction with the file system is captured during the course of a mixed workload. [6] culminates in the intention to monitor I/O activity for a substantial amount of time on a production BlueGene/P machine in order to generate analysis which may help guide developers and administrators in tuning the I/O back-planes used by large machines.

Similarly, IPM [22] uses an interposition layer to catch all calls between the application and the file system. This trace data is then analysed in order to highlight any performance deficiencies that exist in the application or middleware. Based on this analysis, the authors were able to optimise two I/O intensive applications, achieving a four-fold improvement in run-time. In contrast to both Darshan and IPM, RIOT focuses only on the POSIX function calls that are a direct result of using MPI-IO functions. As a result of this restriction, the performance data generated relates only to the parallel I/O operations performed, allowing users to obtain a greater understanding of the behaviour of various I/O intensive codes, as well as assessing the inefficiencies that may exist in any given middleware or MPI implementation.

As a result of previous investigations, a variety of methods have been introduced to improve the performance of existing codes or file systems. The development of middleware layers such as the Parallel Log-structured File System (PLFS) [4] and Zest [14] has, to an extent, bridged the gap between processor and I/O performance. In these systems multiple parallel writes are written sequentially to the file system with a log tracking the current data. Writing sequentially to the file system in this manner offers potentially large gains in write performance, at the possible expense of later read performance [15].

In the case of Zest, data is written sequentially using the fastest path to the file system available. There is however no read support in Zest; instead, it serves to be a transition layer caching data that is later copied to a fully featured file system at a later non-critical time. The result of this is high write throughput but no ability to restart the application until the data has been transferred and rebuilt on a read capable system.

In a similar vein to [23] and [24], in which I/O throughput is vastly improved by transparently partitioning a data file (creating multiple, independent, I/O streams), PLFS uses file partitioning as well as a log-structured file system to further improve the potential I/O bandwidth. An in-depth analysis of PLFS is presented in Section 4.5.

3 Warwick RIOT

The enclosed area in Figure 1 represents the standard flow of parallel applications. When conducting a parallel I/O operation using MPI, the application

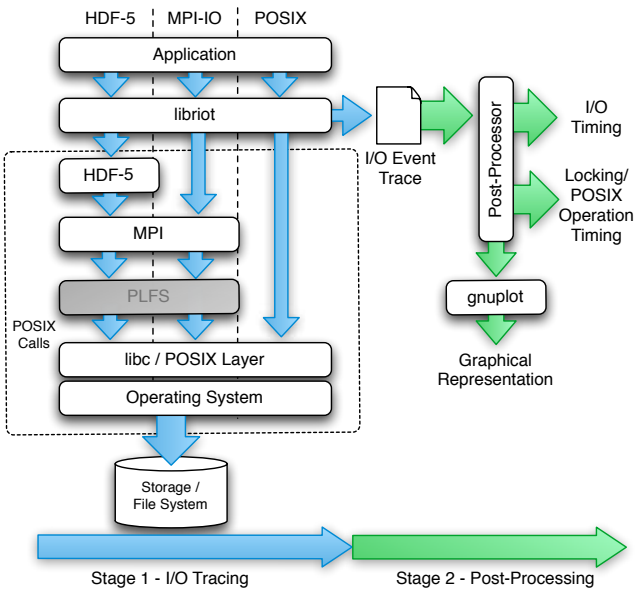


Fig. 1: RIOT tracing and analysis workflow

will make a call to the MPI-IO library, which will then provide inter-process communication and issue POSIX file operations as needed. When using an I/O middleware such as HDF-5, the application first calls the HDF-5 library functions, which will in turn will call a collection of MPI functions. PLFS can be configured to be used as a ROMIO file system driver, which will sit between the MPI library and the POSIX library in the application stack.

Tracing of I/O behaviour in RIOT is conducted using a two stage process. In the first stage (shown on the left in Figure 1), the tracing library is dynamically loaded and linked immediately prior to execution by the operating systems linker. `libriot` then overloads and intercepts calls to MPI-IO and POSIX file functions. The captured events are then performed by the library, where each function is timed and logged for later processing. The library makes use of function interposition to trace activity instead of requiring code modification or application recompilation. RIOT is therefore able to operate on existing application binaries and remain compiler or implementation language agnostic.

When the application being traced has completed, RIOT uses an operating system-level termination hook to write traced I/O information to disk – the delay of logging (by storing events in memory as opposed to flushing to disk) helps to prevent any distortion of application I/O behaviour which may result through the output of information whilst the application is being observed.

In the second stage, a post-execution analysis of the I/O trace is conducted (shown on the right in Figure 1). At this point I/O events are processed with ag-

gregated statistics such as file operation count, total bytes written/read, number of locks *etc.* being generated.

Throughout this paper we make a distinction between effective MPI-IO and POSIX bandwidths – in these studies, "MPI-IO bandwidths" refer to the data throughput of the MPI functions on a per MPI-rank basis, "POSIX bandwidths" relate to the data throughput of the POSIX read/write operations as if performed serially, called directly by the MPI middleware.

4 Case Study

We previously reported on the use of RIOT using a maximum of 256 processing elements on the Minerva cluster located at the Centre for Scientific Computing at the University of Warwick [26]. In this paper we utilise the Sierra supercomputer located at LLNL. Sierra is a 261.3 TFLOP/sec machine consisting of 1,849 nodes each comprising of dual hex-core Intel X5660 "Westmere-EP" processors running at 2.8 GHz. The machine offers 22,188 processor-cores each with a minimum of 2 GB of system memory per-core. Three storage paths are provided, each utilising the Lustre parallel file system with between 755 TB and 1.3 PB of storage available. Bandwidth to each of the file systems ranges from 10 GB/sec up to a maximum of 30 GB/sec. All applications and code compiled for this case study were built using the GNU 4.3.4 compiler and OpenMPI 1.4.3. Both IOR and FLASH-IO utilise the parallel HDF-5 version 1.6.9 library.

4.1 Input/Output Benchmarks

We provide analysis for three I/O benchmarks utilising a range of configurations using the Sierra supercomputer. First we demonstrate the bandwidth tracing ability of RIOT, providing commentary on the divergence between MPI-IO and POSIX effective bandwidths. This difference in bandwidth represents the significant overhead incurred when using blocking collective write operations from the MPI library. We then analyse one simple solution to poor write performance provided by the ROMIO file system layer. Collective buffering creates an aggregator on each node, reducing on-node contention for the file system by sending all data through one process. We then monitor two middleware layers to show how they affect the write bandwidth available to the application. The benchmarks used in this study (described below) have been selected since they all have input/output behaviour which is either extracted directly from a larger parallel application (as is the case with FLASH-IO and BT) or have been configured to be representative of the read/write behaviour used by several scientific applications.

The applications used in this study are:

– **IOR** [19, 20]: A parameterised benchmark that performs I/O operations through both the HDF-5 and MPI-IO interfaces. In this study it has been configured to write 256 MB per process to a single file in 8 MB blocks. Runs have been performed on a range of configurations, utilising between 1 and

128 compute nodes. Its write performance through both MPI-IO and HDF-5 are assessed.

- **FLASH-IO**: This benchmark replicates the checkpointing routines found in FLASH [7, 18], a thermonuclear star modelling code. In this study we use a $24 \times 24 \times 24$ block size per process, causing each process to write approximately 205 MB to disk through the HDF-5 library.
- **BT** [2, 3]: An application from the NAS Parallel Benchmark (NPB) suite has also been used in this study, namely the Block-Tridiagonal (BT) solver application. There are a variety of possible problem sizes but for this study we have used the C and D problem classes, writing a data-set of 6 GB and 143 GB respectively. This application requires the use of a square number of processes (*i.e.*, 4, 9, 16), therefore we have executed this benchmark on 1, 4, 16, 64, 256, 1,024 and 4,096 processors. Performance statistics were collected for BT using MPI-IO with and without collective buffering enabled, and also using the PLFS ROMIO layer.

Five runs of each configuration were performed, and the data contained throughout this paper is the mean from each of these runs, in an effort to reduce the influence of other users jobs.

4.2 MPI-IO and POSIX Bandwidth Tracing

First we demonstrate the MPI-IO and POSIX bandwidth traces for the three selected benchmarks. Figure 2 shows the significant gap between MPI bandwidth and POSIX bandwidth for each of the three benchmarks when using MPI-IO and parallel HDF-5. It is important to note that *effective* bandwidth refers to the total amount of data written divided by the total time spent writing as if done serially. Since the MPI-IO functions used are collective blocking operations, we can assume they are executed in parallel, therefore the *perceived* bandwidth is the effective bandwidth multiplied by the number of processor cores. As the POSIX write operations are performed in a non-deterministic manner, we cannot make any assumption about the perceived bandwidth; it suffices to say it is bounded by effective bandwidth multiplied by the processor count and the MPI perceived bandwidth.

Figure 2 also demonstrates a large performance gap between the BT mini-application and the other two benchmarks. This is largely due to the use of collective buffering ROMIO hints utilised by BT. The performance of HDF-5 in both IOR and FLASH-IO is also of interest. The POSIX write performance is much close to the MPI-IO performance in both IOR with HDF-5 and FLASH-IO. Despite this, the overall MPI-IO performance is lower when using HDF-5. The performance gap between HDF-5 and MPI-IO is analysed in Section 4.4. For the remainder of this study we concentrate on both BT and FLASH-IO as these better emulate the write behaviour found in other scientific applications.

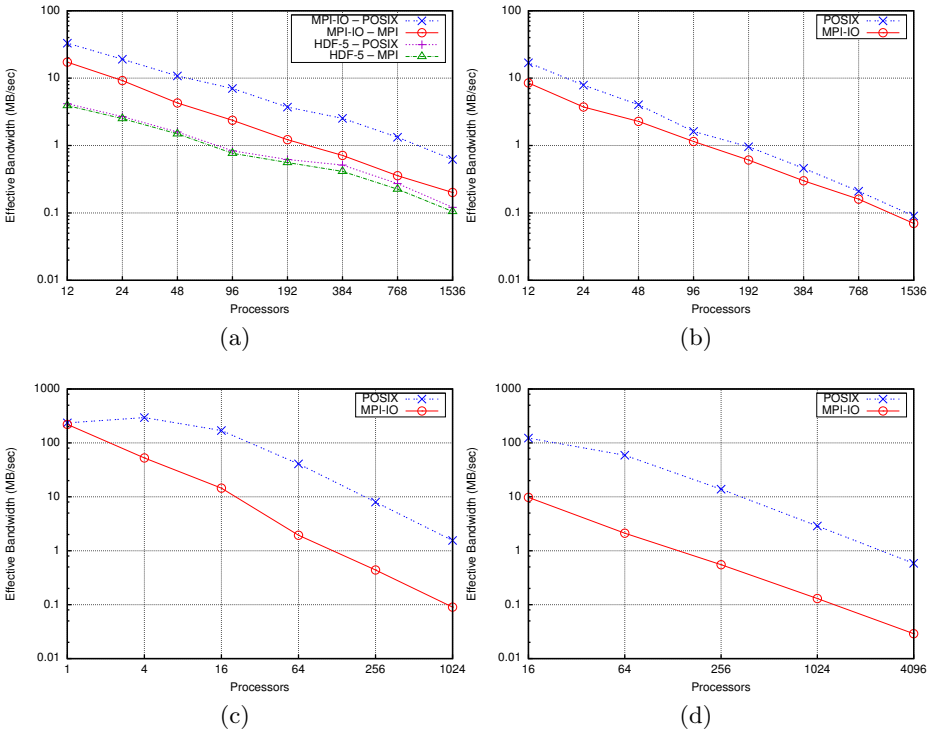


Fig. 2: Effective MPI and POSIX Bandwidths for (a) IOR (with both MPI-IO and HDF5), (b) FLASH-IO, (c) BT class C, and (d) BT class D

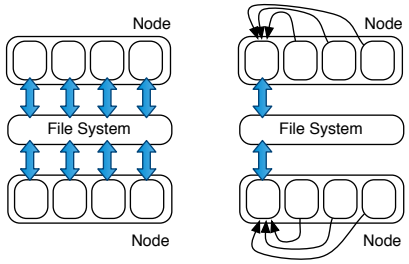
4.3 Collective Buffering in ROMIO

As stated previously, BT makes use of the collective buffering ROMIO hint. This causes MPI to assign a set of "aggregator" processes that perform the I/O required. This is shown in Figure 3, where each process writes to the file system in (a) and each process communicates the data to an aggregator in (b). Using an aggregator on each node reduces the on-node contention for the file system and allows the file server to perform node-level file locking, instead of requiring on-node POSIX file locking, as is the case without collective buffering.

Table 1 demonstrates the bandwidth achieved by BT with and without collective buffering enabled. It is interesting to note the POSIX performance on 4 processor cores with and without collective buffering is very similar. As soon as the run is increased to 16 processes, the POSIX bandwidth is significantly reduced as there is now more on-node competition for the file system. Furthermore, the operating system cannot buffer the writes effectively as the behaviour of the second compute node is unknown.

	Processor Cores			
	4	**16**	**64**	**256**
MPI-IO				
With collective buffering	70.86	23.63	4.45	1.24
Without collective buffering	9.90	3.46	1.34	0.54
POSIX				
With collective buffering	490.08	293.30	109.71	21.94
Without collective buffering	329.69	6.63	4.85	3.49

Table 1: MPI-IO and POSIX write performance (MB/sec) for BT on class C, with and without collective buffering



(a) (b)

Fig. 3: (a) Each process writes to the file system and (b) each process communicates with a single aggregator process on each node

4.4 Analysis of HDF-5

In [26] we demonstrated the performance of FLASH-IO on the Minerva cluster located at the Centre for Scientific Computing at the University of Warwick. Minerva utilises two GPFS servers, backed by an array of 2 TB hard drives, connected through QLogic 4X QDR Infiniband. Table 2 demonstrates a similar analysis for FLASH-IO at increased scale on the Sierra supercomputer. Whilst we previously reported a significant locking overhead, this becomes negligible when using the Lustre file system. However, there remains a large read-back overhead when using HDF-5 through MPI-IO. In the worst case POSIX reads account for nearly half the total MPI write time. As the problem is scaled, POSIX reads still make up 20% of the overall MPI write time.

Whilst HDF-5 allows easy application integration, due to the standardised formatting, it creates a significant overhead for parallel writes. When data is being written periodically for checkpointing or visualisation purposes, this overhead creates a significant slow-down in application performance. This analysis motivates opportunities for HDF-5 optimisations, including reducing or eliminating the read-back behaviour and enabling some form of node-level write aggregation to reduce locking overheads.

	Processor Cores							
	12	**24**	**48**	**96**	**192**	**384**	**768**	**1536**
MB written	2812.48	5485.68	11037.27	23180.15	43191.78	86729.59	179202.26	365608.30
MPI write calls	636	1235	2436	4836	9634	19233	38432	76832
POSIX write calls	6050	11731	23584	49764	91707	184342	382608	783068
POSIX read calls	5824	11360	22856	48000	89328	179437	370904	757060
Locks requested	12000	24000	48000	96000	192000	384000	768000	1536000
MPI write time (sec)	331.14	1471.30	4832.43	20215.00	70232.87	288633.74	1145889.19	4942746.26
POSIX write time (sec)	166.54	696.58	2737.14	14298.17	44869.00	190337.91	845969.92	3909161.40
POSIX read time (sec)	139.40	503.34	1462.00	5850.12	16782.82	66895.21	235866.21	908076.69
Lock time (sec)	5.95	14.02	28.80	57.80	172.80	385.65	1176.30	4328.96

Table 2: Detailed breakdown of MPI-IO and POSIX timings for FLASH-IO writes

4.5 Analysis of PLFS Middleware

Whilst HDF-5 has been shown to decrease parallel write performance, the Parallel Log-structured File System from LANL has been demonstrated to improve performance in a wide variety of circumstances, including for production applications from LANL and the United Kingdom's Atomic Weapons Establishment (AWE) [4]. PLFS is an I/O interposition layer designed primarily for checkpointing and logging operations.

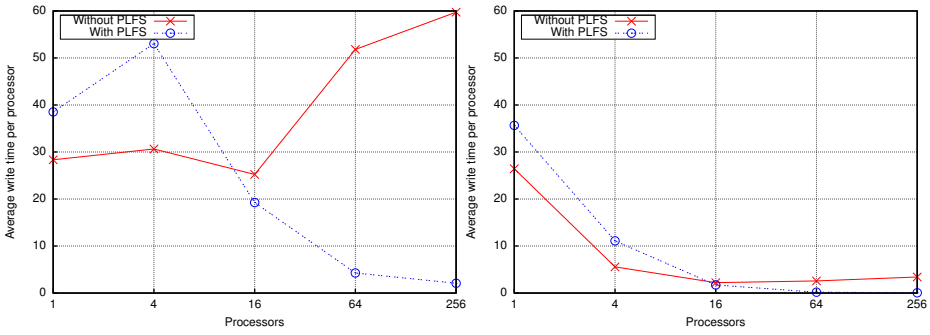
PLFS works by intercepting MPI-IO calls through a ROMIO file system driver, and translates the operations from n-processes writing to 1 file, to n-processes writing to n-files. The middleware creates a view over the n-files, so that the calling application can view and operate on these files as if they were all concatenated into a single file. The use of multiple files by the PLFS layer helps to significantly improve file write times as multiple, smaller files can be written simultaneously. Furthermore, improved read times have also been demonstrated when using the same number of processes to read back the file as were used in its creation [16].

Figures 4(a) and 4(b) present the average total MPI-IO and POSIX write time per process for the BT benchmark when running with and without the PLFS ROMIO file system driver. Note that as previously, POSIX bandwidth in this table refers to the bandwidth of POSIX operations called from MPI-IO and hence are higher due to the additional processing required by MPI. It is interesting to note that the write time when using PLFS is generally larger or comparable with the same run not utilising PLFS when using a single node (as is the case with 1 and 4 processes) due to the on-node contention for multiple files.

The effective MPI-IO and POSIX bandwidths are shown in Table 3. Whilst previously we have seen POSIX write bandwidth decrease at scale, PLFS partially reverses this trend as writes can be flushed to the cache as they are not waiting on surrounding writes. The log structured nature of PLFS also increases the bandwidth as data can be written in a non-deterministic sequential manner, with a log file keeping track of the data ordering. For a BT class C execution

	Processor Cores				
	1	**4**	**16**	**64**	**256**
MPI-IO					
Standard	220.64	52.58	13.89	1.97	0.45
PLFS	164.21	29.81	21.07	23.72	12.18
POSIX					
Standard	235.51	294.80	169.56	40.78	7.98
PLFS	177.34	142.51	235.44	538.13	437.88

Table 3: Effective MPI-IO and POSIX bandwidth (MB/sec) for BT class C using MPI-IO and PLFS



(a) (b)

Fig. 4: (a) MPI-IO write time per processor and (b) POSIX write time per processor

on 256 processors, PLFS increases the bandwidth from 115.2 MB/sec perceived bandwidth up to 3,118.08 MB/sec, representing a 27-fold increase in write performance.

Figure 5 demonstrates that during the execution of BT on 256 processors, concurrent POSIX write calls wait much less time for access to the file system. As each process is writing to its own unique file, each process has access to its own unique file stream, reducing file system contention. This results in each POSIX write call completing much more quickly as the data can be flushed to the cache. For non-PLFS writes we see a stepping effect where all POSIX writes are queued and complete in a non-deterministic order. Conversely, PLFS writes do not exhibit this stepping behaviour as the writes are not waiting on other processes to complete.

The average time per POSIX write call is shown in Table 4. As the number of processes writing increases, data written in each write decreases. However when using standard MPI-IO, the time to write the data increases due to contention. When using the PLFS ROMIO file system driver, the write time decreases due to the transparent partitioning of the data files.

	Processor Cores				
	1	**4**	**16**	**64**	**256**
Number of POSIX writes	13040	440	480	480	880
Standard MPI-IO					
Total time in POSIX write (s)	27.617	22.264	38.459	159.348	816.373
Time per write (s)	0.002	0.051	0.080	0.332	0.928
MPI-IO with PLFS					
Total time in POSIX write (s)	35.669	44.383	27.554	12.055	14.815
Time per write (s)	0.003	0.101	0.057	0.025	0.017

Table 4: Total time in POSIX writes and average time per POSIX write for BT class C

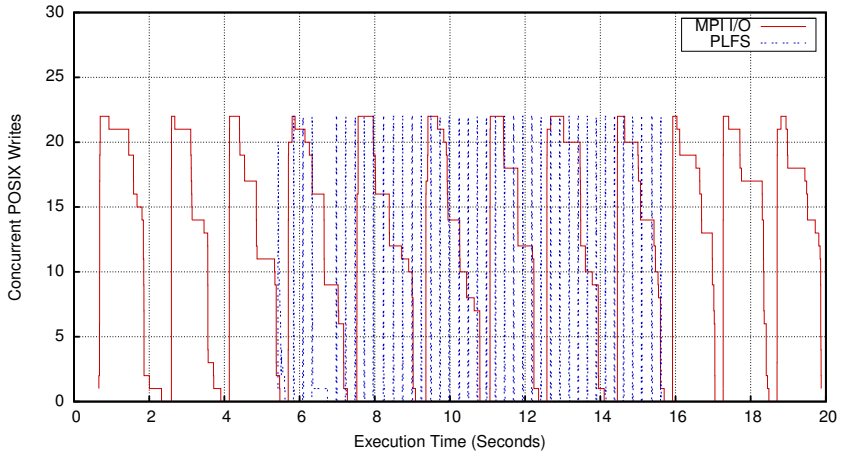


Fig. 5: Concurrent POSIX write calls for BT through MPI-IO and PLFS

5 Conclusions

Parallel I/O operations continue to represent a significant bottleneck in large-scale parallel scientific applications. This is, in part, because of the slower rate of development that parallel storage has witnessed when compared to that of micro-processors. However, other causes relate to limited optimisation at code level as well as the use of complex file formatting libraries. The situation is that contemporary applications can often exhibit poor I/O performance because code developers lack an understanding of how their code utilises I/O resources and how best to optimise for this.

In this paper we utilise the RIOT toolkit to intercept, record and analyse information relating to file reads, writes and locking operations within three standard industry I/O benchmarks and mini-applications. We presented a case study demonstrating RIOT's ability to:

– Calculate effective MPI-IO write bandwidths as well as produce bandwidths for POSIX file system calls originating from MPI-IO at increased scale. The comparison of these two figures demonstrates the slow-down in per-process write speed which results from the use of MPI. Typically these overheads arise because of file system contention, collective negotiation between MPI ranks for lock ownership and the calculation of offsets at which reads or writes take place;
– Provide detailed write behaviour analysis through the interception of read, write and locking operations included aggregated read/write time and the time spent obtaining or releasing per-file locks. Our results demonstrate the significant overhead incurred when performing HDF-5-based writes in the FLASH-IO benchmark. The nature of this analysis allows light-weight, non-intrusive detection of potential bottlenecks in I/O activity providing a first point at which application designers can begin optimisation;
– Compare low-level file system behaviour. In the last section of our case study we were able to investigate the low-level improvement which results in the use of PLFS middleware when executing the BT benchmark. PLFS is designed specifically to reduce file system and parallel overheads through the interposition of MPI file operations to re-target n-to-1 operations to n-to-n operations. Tracing of these runs using RIOT was able to demonstrate improvement in MPI-IO bandwidth due to the improvement in parallel POSIX bandwidth.

5.1 Future Work

This work builds upon preliminary work in [26] to show RIOT operating at scale on the 261.3 TFLOP/s Sierra supercomputer. Future studies are planned, applying this method of I/O tracing to larger, full-science applications. We expect these to exhibit increased complexity in their read/write behaviour resulting in increased contention and stress on the parallel file system. Further work with our industrial sponsors is also expected to use RIOT in the on-going assessment of parallel file system software and I/O-related middleware including the use of Lustre, GPFS, PLFS and alternatives.

Furthermore, future work is expected to include in-depth analysis of various I/O configurations, such as that utilised by BlueGene systems. It is expected that the performance characteristics seen on these systems will introduce additional complexities in measuring and visualising the I/O behaviour.

Acknowledgements

This work is supported in part by The Royal Society through their Industry Fellowship Scheme (IF090020/AM). Access to the LLNL Open Computing Facility 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 and Meghan Wingate at the Los Alamos National Laboratory for their expert PLFS advice and support.

References

1. Almási, G., et al.: Implementing MPI on the BlueGene/L Supercomputer. In: Proceedings of the 10th International European Conference on Parallel and Distributed Computing (Euro-Par'04). pp. 833–845 (2004)
2. Bailey, D.H., et al.: The NAS Parallel Benchmarks. International Journal of High Performance Computing Applications 5(3), 63–73 (1991)
3. Bailey, D.H., et al.: The NAS Parallel Benchmarks. Tech. Rep. RNR-94-007, NASA Ames Research Center (March 1994)
4. Bent, J., et al.: PLFS: A Checkpoint Filesystem for Parallel Applications. In: Proceedings of the ACM/IEEE International Conference on Supercomputing Conference (SC'09) (November 2009)
5. Bull: BullX Cluster Suite Application Developer's Guide (April 2010)
6. Carns, P., et al.: 24/7 Characterization of Petascale I/O Workloads. In: Proceedings of the IEEE International Conference on Cluster Computing and Workshops (CLUSTER'09). pp. 1–10 (September 2009)
7. Fryxell, B., et al.: FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes. The Astrophysical Journal Supplement Series 131(1), 273 (2000)
8. Fuerlinger, K., Wright, N., Skinner, D.: Effective Performance Measurement at Petascale Using IPM. In: Proceedings of the IEEE 16th International Conference on Parallel and Distributed Systems (ICPADS'10). pp. 373–380 (December 2010)
9. Gabriel, E., et al.: Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In: Proceedings of the 11th European PVM/MPI Users' Group Meeting. pp. 97–104 (September 2004)
10. Gropp, W.: MPICH2: A New Start for MPI Implementations. In: Kranzlmller, D., Volkert, J., Kacsuk, P., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface, Lecture Notes in Computer Science, vol. 2474, pp. 37–42. Springer Berlin / Heidelberg (2002)
11. Koziol, Q., Matzke, R.: HDF5 – A New Generation of HDF: Reference Manual and User Guide. Tech. rep., National Center for Supercomputing Applications, Champaign, Illinois, USA (1998)
12. Li, J., et al.: Parallel netCDF: A High-Performance Scientific I/O Interface. In: Proceedings of the ACM/IEEE International Conference on Supercomputing (SC'03) (November 2003)
13. Message Passing Interface Forum: MPI2: A Message Passing Interface Standard. High Performance Computing Applications 12(1–2), 1–299 (1998)
14. Nowoczynski, P., Stone, N., Yanovich, J., Sommerfield, J.: Zest Checkpoint Storage System for Large Supercomputers. In: Proceedings of the 3rd Annual Workshop on Petascale Data Storage (PDSW'08). pp. 1–5 (November 2008)
15. Polte, M., et al.: Fast Log-based Concurrent Writing of Checkpoints. In: Proceedings of the 3rd Annual Workshop on Petascale Data Storage (PDSW'08). pp. 1–4 (November 2008)

16. Polte, M., et al.: ... 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). pp. 21–25 (November 2009)
17. Rew, R.K., Davis, G.P.: NetCDF: An Interface for Scientific Data Access. IEEE Computer Graphics and Applications 10(4), 76–82 (1990)
18. Rosner, R., et al.: Flash Code: Studying Astrophysical Thermonuclear Flashes. Computing in Science & Engineering 2(2), 33–41 (2000)
19. Shan, H., Antypas, K., Shalf, J.: Characterizing and Predicting the I/O Performance of HPC Applications using a Parameterized Synthetic Benchmark. In: Proceedings of the ACM/IEEE International Conference on Supercomputing (SC'08) (November 2008)
20. Shan, H., Shalf, J.: Using IOR to Analyze the I/O Performance for HPC Platforms. In: Cray User Group Conference (CUG'07). Seattle, WA, USA (May 2007)
21. Thakur, R., Lusk, E., Gropp, W.: ROMIO: A High-Performance, Portable MPI-IO Implementation. Tech. Rep. ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory (1997)
22. Uselton, A., et al.: Parallel I/O Performance: From Events to Ensembles. In: Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS'10). pp. 1–11 (April 2010)
23. Wang, Y., , Wang, Y., Kaeli, D.: 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) (September–October 2003)
24. Wang, Y., Kaeli, D.: Profile-guided I/O Partitioning. In: Proceedings of the 17th Annual International Conference on Supercomputing (ICS'03). pp. 252–260 (June 2003)
25. Wolman, B., Olson, T.: IOBENCH: A System Independent IO Benchmark. ACM SIGARCH Computer Architecture News 17(5), 55–70 (1989)
26. Wright, S.A., Pennycook, S.J., Hammond, S.D., Jarvis, S.A.: RIOT – A Parallel Input/Output Tracer. In: Proceedings of the 27th Annual UK Performance Engineering Workshop (UKPEW'11). pp. 25 – 39 (July 2011)