RIOT – A Parallel Input/Output Tracer

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

Performance Computing and Visualisation
Department of Computer Science
University of Warwick, UK
{saw, sjp, sdh, 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. The effect is often a bottleneck when any form of file-system interaction is required.

In this paper we present RIOT – an input/output tracing toolkit being developed at the University of Warwick for dynamic attachment to parallel applications. The two-stage tracing process includes a lightweight library to record I/O events and an in-depth post-execution analysis tool to extract performance metrics such as MPI-IO bandwidth, effective POSIX/file-system bandwidth, duration of individual or aggregated time spent in obtaining or releasing file locks and temporal information relating to parallel file activity.

We present a case study on the use of RIOT for three standard industry I/O benchmarks: the BT-IO micro-application from NASA's Parallel Benchmark suite, FLASH-IO, a benchmark which replicates the checkpointing operations of the FLASH thermonuclear star modelling code and IOR, an industry standard I/O benchmark using HDF-5 and MPI-IO. Furthermore, we utilise RIOT to assess these codes when running with the Parallel Log-structured File System (PLFS) middleware developed by the Los Alamos National Laboratory.

Keywords. Input/Output, Message Passing Interface, Parallel I/O.

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 from across multiple domain areas. 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. The use of parallel file libraries has helped to a lesser extent, in that data formats have been standardised enabling optimisation by library developers but, in part at least, this has created a lack of responsibility on the part of application developers to investigate optimising their data storage operations.

Another aspect of I/O which has become popular as the scale of machines has grown has been the use of checkpointing – the periodic writing of application state 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.

In this paper we present the RIOT input/output toolkit (referred to throughout the remainder of this paper by the recursive acronym RIOT) – 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 message passing interface input/output (MPI-IO) layer [16] and relate these to underlying POSIX operations including file locking, unlocking and low-level data reads or writes. Through timing these operations and aggregating them to the respective MPI-IO operations, effective application bandwidths can be obtained as well as an analysis of the percentage of time spent in individual low-level kernel calls. 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 applying our tool to a set of industry standard I/O benchmarks.

The specific contributions of this work are the following:

- The presentation of RIOT – a new MPI-IO tracing and profiling toolkit. We describe its functionality, implementation strategy and flexibility in analysing the performance characteristics of parallel applications;

- We apply our tracing tool to assessing the I/O behaviour of three standard industry benchmarks – the block-triangular input/output (BT-IO) 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 [14, 15]. Although these benchmarks represent small sections of code extracted from larger parallel applications, they are able to accurately recreate the parallel input/output behaviours, making the results obtained from our analysis of interest to a wider audience;

- We execute these selected benchmarks using a variety of configurations including use of the low-level Hierarchical Data Format 5 (HDF-5) library [8] commonly used by parallel scientific applications and the Parallel Log-structured File System (PLFS) [3, 11] – a novel I/O middleware being developed by the Los Alamos National Laboratory to improve file write times;

- Finally, we compare the run-time of applications with and without RIOT to assess the overheads and implications of using our tool for the investigation of parallel applications at scale. These investigations show overheads that are typically less than 10% of the initial application run-time demonstrating the extremely low penalty that RIOT is able to offer.

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 RIOT and highlights its core features; 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 [20] and IOR [14] 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 BT-IO benchmark from NASA's Parallel Benchmark Suite [2] and the FLASH-IO [12] application from the University of Chicago – both of which are used in this paper.

Whilst benchmarks may provide a measure of file-system performance they are rarely useful for diagnosing problem areas or scope for optimisation within large codes. For this activity profiling tools are often required which must assess read/write behaviour in parallel. One such tool is the *Ex*tensible *Per*formance *T*oolkit (ExPerT) developed by Eckart *et al.* in [5]. This framework is able to benchmark, trace and profile applications within a unified environment providing interfaces for parsing benchmark data and presenting this graphically. The tool presented in this paper differs from ExPerT in that is it much more limited in scope and therefore has lower overhead. The dynamic instrumentation of RIOT also requires no code modification for tracing to be possible.

Another approach, which aims to ascertain the performance characteristics of production-grade scientific codes, is to intercept communications between the application and underlying file-system. This is the approach taken by RIOT and also by Darshan [4], from Argonne National Laboratory, and the Integrated Performance Monitoring (IPM) tool [7], from 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. [4] culminates in the intention to monitor the I/O 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, [17] uses an interposition layer to catch all calls between the application and the file system. This large amount of data is then analysed in order to highlight any performance deficiencies that exist in the application or middleware. Based on the analysis of this data, the authors are able to optimise two I/O intensive applications, achieving a four-fold improvement in run-time.

Much work into improving the performance of parallel I/O systems has been conducted as a means of improving the performance of existing codes or file-systems. This has lead to the development of middleware layers such as the Parallel Log-structured File System (PLFS) [3] and Zest [9] . 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 expense of later read performance [10].

In the case of Zest, data is written sequentially using the fastest path to the file-system available. There is 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 non-critical time. The result of this is high write throughput but no ability to restart until the data has been moved and rebuilt elsewhere.

In a similar vein to the work in [18] and [19], 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. A more in-depth analysis of PLFS is presented in Section 4.5.

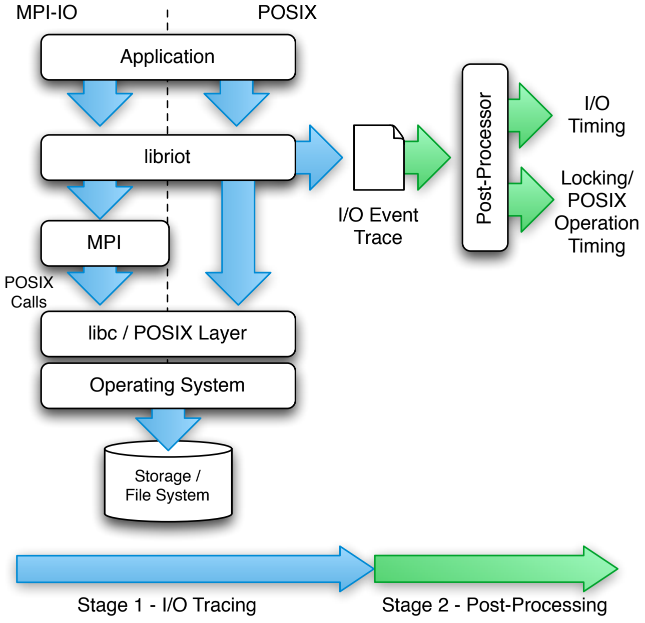

Fig. 1: RIOT tracing and analysis workflow

3 Warwick RIOT

Tracing of I/O behaviour in RIOT is conducted using a two stage process. In the first stage (shown as blue arrows in Figure 1), the tracing library is preloaded and linked immediately prior to execution by the operating systems linker. This enables MPI-IO and POSIX file functions to be intercepted via the tool. Calls to these functions are intercepted, passing control to the RIOT tracing library which is able to record the I/O event in memory before being passed either onto the MPI run-time or the appropriate POSIX function as required. As such the library makes use of function interposition to trace activity as opposed to requiring code modification or recompilation. RIOT is therefore able to operate on existing application binaries and remain compiler or implementation language agnostic. Calls made to the MPI file routines eventually end up being passed to the POSIX layer for reading/writing on the file-system. RIOT is able to intercept these and aggregate them to the respective MPI-IO operation so that the low-level file operations being executed are able to be benchmarked and traced separately. 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 in serial, called directly by the MPI middleware.

When the application being traced has completed, RIOT uses an operating system-level termination hook to write summarised 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 as green arrows in Figure 1). At this point I/O events are processed with aggregated statistics such as total bytes written/read, number of locks *etc.* being generated. As the trace also encodes temporal information in the reported I/O events, an analysis of activity over time can also be conducted – sample outputs are demonstrated in Section 4.4.

4 Case Study

We report on the use of RIOT when applied to three industry standard benchmarks, running on the newly commissioned Minerva cluster being installed at the Centre for Scientific Computing at the University of Warwick. Minerva is a 258-node IBM iDataPlex cluster in which each node comprises dual hex-core Intel X5650 "Westmere-EP" processors running at 2.66 GHz. The machine offers a total of 3096 processor-cores with a minimum of 2 GB of system memory per-core. Communications between nodes are conducted via a non-blocking array of QLogic TrueScale 4X QDR InfiniBand switches. 100 TB of IBM GPFS [13] parallel scratch space is provided by 2 storage servers. 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

In order to assess the tracing and analysis capabilities of RIOT, three I/O benchmarks have been executed on Minerva with the tracing library enabled. The benchmarks (described below) have input/output behaviour which is either extracted directly from a larger parallel application (as is the case with FLASH-IO and BT-IO) or has been configured to be representative of the read/write behaviour used by several scientific applications.

The applications used in this study are:

- **IOR** [14, 15]: 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 16 and 256 cores. Its read and write performance through both MPI-IO and HDF-5 are assessed.

	BT-IO		**FLASH-IO**		**IOR (MPI-IO)**		**IOR (HDF-5)**	
	MPI-IO	POSIX-IO	MPI-IO	POSIX-IO	MPI-IO	POSIX-IO	MPI-IO	POSIX-IO
16	21.41	159.65	3.47	7.66	8.43	21.97	7.60	17.75
32	N/A	N/A	1.93	3.49	3.68	11.75	3.63	6.92
64	4.95	84.90	0.90	1.51	1.92	4.36	0.84	1.89
128	N/A	N/A	0.50	0.77	1.00	2.03	0.53	0.85
256	1.11	24.48	0.26	0.37	0.56	1.13	0.26	0.48

Table 1: Effective MPI-IO and POSIX write bandwidths for each of the three benchmarks (MBytes (written)/sec)

- **BT-IO** [1, 2]: An application from the NAS Parallel Benchmark (NPB) suite has also been used in this study, namely the Block-Tridiagonal (BT) application. There are a variety of possible problem sizes but for this study we have used the C problem class, writing a data-set of 3 GB. This application requires the use of a square number of processes (*i.e.*, 4, 9, 16). In order to align BT with the other benchmarks in this study, runs were performed on configurations where the square number was also a power of 2. Performance statistics were collected for BT using MPI-IO and also using the PLFS ROMIO layer.
- **FLASH-IO**: This benchmark replicates the checkpointing routines found in FLASH [6, 12], a thermonuclear star modelling code. It should be noted that FLASH-IO has no read back capabilities and therefore was not used to assess parallel read operations. 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.

The benchmarking and tracing of these codes was conducted on the Minerva machine prior to general user availability. The effect of this early access has been that executions and benchmarked results are highly repeatable between subsequent runs. The benchmarked data contained throughout this paper is therefore the mean taken from three executions since jobs experience low degrees of variability.

4.2 MPI-IO and POSIX Bandwidth Tracing

In the first demonstration of RIOT we utilise tracing data to benchmark the effective bandwidth rates of MPI-IO operations and the low-level POSIX operations which the library intercepts. This is performed by tracing the total time spent in MPI-IO file routines as well as independently timing the child calls to POSIX read, write and lock activities. The difference between the MPI-IO and POSIX bandwidths provides an estimate of the overheads that the use of MPI-IO routines create. The write and read bandwidths for each of the three benchmarks are demonstrated in Table 1 and Table 2 respectively. It should be noted that the *effective* POSIX bandwidth is the number of bytes written by

	BT-IO		**IOR (MPI-IO)**		**IOR (HDF-5)**	
	MPI-IO	POSIX-IO	MPI-IO	POSIX-IO	MPI-IO	POSIX-IO
16	21.41	159.65	12.46	17.74	14.41	14.64
32	N/A	N/A	6.08	9.36	7.66	7.73
64	10.23	181.98	3.20	5.55	3.07	3.08
128	N/A	N/A	1.44	2.47	1.90	1.90
256	4.83	165.03	0.69	1.16	0.85	0.85

Table 2: Effective MPI-IO and POSIX read bandwidths for each of the three benchmarks (MBytes (read)/sec)

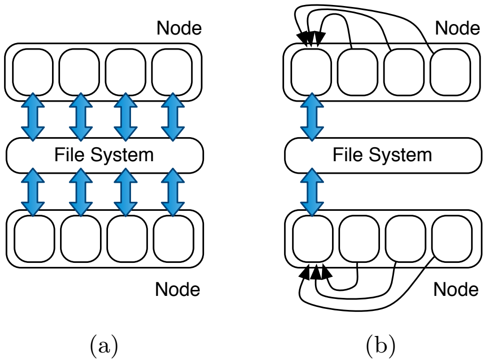

Fig. 2: (a) Each process writes to the file-system and (b) node-level write aggregation with each process communicating with a single aggregator process on each node

all processes divided by the total time spent in POSIX write calls by all processes throughout the execution. The (*perceived*) POSIX bandwidth achieved by the application – that is the total bytes written by all processes divided by the maximum time taken by any single process – is therefore higher than the quoted value due to execution in parallel. MPI-IO bandwidth is the average bandwidth achieved by any given process during calls to MPI file operations.

It should be noted that the high bandwidth achieved by BT-IO is largely due to the applications write behaviour and write caching. BT-IO uses node-level write aggregation to decrease the number of processes accessing the I/O back-plane (illustrated in Figures 2(a) and 2(b)). In order to assess the read and write performance of BT with all processors writing data, the experiments were repeated using scattered node allocations (*i.e.*, all MPI ranks were performing I/O). Table 3 demonstrates performance that is much more in-line with the other benchmarks used in this study.

	Write				**Read**			
	Packed		**Scattered**		**Packed**		**Scattered**	
	MPI-IO	POSIX-IO	MPI-IO	POSIX-IO	MPI-IO	POSIX-IO	MPI-IO	POSIX-IO
16	21.41	159.65	5.74	12.39	21.41	159.46	13.96	52.32
64	4.95	84.90	0.84	1.72	10.23	181.98	3.27	28.98
256	1.11	24.48	0.24	0.83	4.83	165.03	0.78	16.25

Table 3: Effective MPI-IO and POSIX write bandwidths for BT-IO using both packed and scattered node configurations (MBytes (written)/sec)

	Processor Cores				
	16	**32**	**64**	**128**	**256**
`MPI_File_write()` Call Count	836	1637	3237	6435	12834
POSIX `write()` Call Count	8308	16605	33173	66331	132626
Locks Requested	7984	15992	31984	63992	127984
Total Time in MPI Writes (secs)	1112.22	3992.01	17212.95	61908.37	238024.80
Total Time in POSIX Writes (secs)	503.96	2213.92	10243.53	40008.50	169246.40
Total Time in Lock Operations (secs)	125.07	316.61	2363.96	7038.46	14546.15
Total Time in Unlock Operations (secs)	0.15	0.55	1.36	2.98	6.15
Percentage of MPI time in POSIX	45.31	55.46	59.51	64.63	71.10
Percentage of MPI time in Locks	11.25	7.93	13.73	11.37	6.11
Percentage of MPI time in Unlocks	0.01	0.01	0.00	0.00	0.00

Table 4: Aggregated statistics for FLASH-IO writes through the HDF-5 library

4.3 Detailed Application Analysis: FLASH-IO Write Behaviour

Detailed in Table 4 are the aggregated statistics for the MPI-IO function calls, and underlying POSIX function calls, experienced when running the FLASH-IO benchmark. As is demonstrated, the HDF-5 output library relies heavily on advisory file locking to ensure the consistency of its data files. In the best case the added overhead experienced by acquiring file locks accounts for 6% of the MPI write time. In the worst case this is nearer 12%. Whilst locking is often required to ensure consistency between nodes, when writing out checkpoint information (as is the case in FLASH-IO), file locking does not need to be performed as each process writes to its own section of the file, which is not being concurrently read.

Furthermore, POSIX write operations may account for only 50% of the total time spent in MPI write calls. As the application is scaled to 256 cores, 71% of the MPI write time is spent performing POSIX write calls. The write calls also take longer to complete due to the I/O contention experienced, resulting in a slower overall write bandwidth.

4.4 Temporal Analysis: IOR Benchmark

As the operations recorded by the RIOT tracing library are timestamped, graphical representations of the I/O can be produced. In Figure 3 we demonstrate the

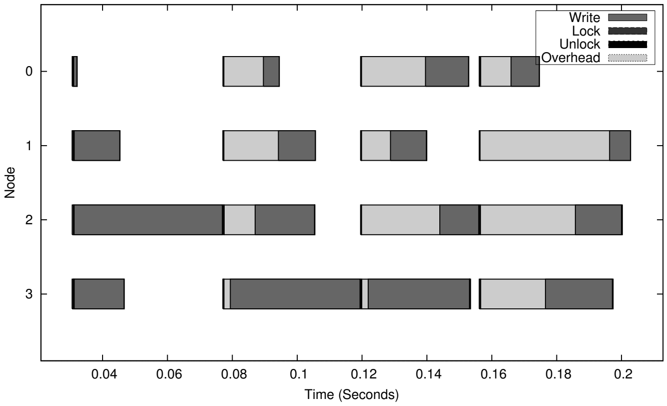

Fig. 3: Graphical representation of the write behaviour of IOR through HDF-5

	Processor Cores		
	16	**64**	**256**
MPI-IO Bandwidth no PLFS (Mbytes/sec)	13.96	0.84	0.24
MPI-IO Bandwidth with PLFS (Mbytes/sec)	17.27	3.51	0.49
Improvement (%)	23.71	317.86	104.17
POSIX Bandwidth no PLFS (Mbytes/sec)	12.39	1.72	0.83
POSIX Bandwidth with PLFS (Mbytes/sec)	72.07	36.60	8.86
Improvement (%)	481.68	2027.90	967.47

Table 5: Traced write bandwidth (MBytes/second) for BT-IO through PLFS

I/O behaviour for a small 4-core run of the IOR benchmark utilising the HDF-5 library. Although the POSIX write operations are performed in parallel, at the disk-level the operations are serialised by the file server – increasing the write time for all but one of the processes. The overhead associated with MPI-IO, and in particular HDF-5, is also a significant factor in the write performance.

Through the use of temporal analysis, using RIOT, application developers can visualise the behaviour of their codes, highlighting hot spots of I/O slowdown. Figure 3 clearly demonstrates significant overhead in later MPI calls which, in this case results from read calls performed by the HDF-5 library. It is also clear from the graphic that significant amounts of time are wasted by processes 0 and 1 whilst waiting for the blocking collective MPI operations to complete. Use of non-blocking collectives may provide a significant improvement in performance. Furthermore, file segmentation may result in an improvement in parallel POSIX write operations.

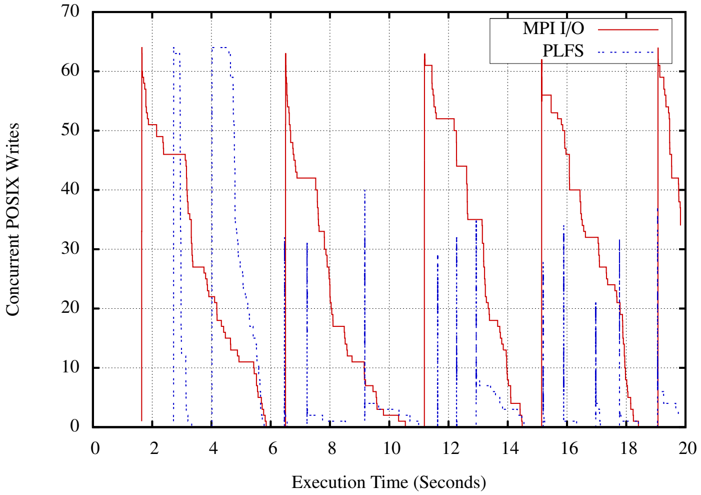

Fig. 4: Concurrent POSIX write calls for BT-IO through MPI-IO and PLFS.

4.5 Analysis of PLFS Middleware

The Parallel Log-structured File System (PLFS) is an I/O interposition layer developed by the Los Alamos National Laboratory (LANL) to improve the performance of parallel I/O operations. Although the middleware is intended primarily for checkpointing and logging operations, it has been shown to provide performance improvements in a wide variety of circumstances including production applications from LANL [3].

PLFS works by intercepting MPI-IO calls through the ROMIO MPI 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 [11].

Table 5 presents MPI-IO and POSIX traced write-bandwidth for the BT-IO benchmark when using PLFS middleware. 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 also interesting to note that whilst MPI-IO bandwidth drops dramatically, POSIX bandwidth reduces in a much more stable manner. This indicates that the write performance of MPI-IO is heavily affected by additional overheads such as file locking and concurrent access to the same file as process count increases.

	Processor Cores				
	16	32	64	128	256
FLASH-IO					
Output Time (w/o tracer) (secs)	86.43	155.40	295.34	570.04	1033.39
Output Time (with tracer) (secs)	85.91	162.65	312.37	563.79	1081.87
Percentage Change (%)	-1.01	+4.66	+5.76	-1.09	+4.69
IOR (HDF-5)					
Output Time (w/o tracer) (secs)	48.42	181.62	405.27	684.28	1211.98
Output Time (with tracer) (secs)	53.01	180.73	426.32	700.58	1317.74
Percentage Change (%)	+9.48	-0.49	+5.19	+2.38	+8.73
IOR (MPI-IO)					
Output Time (w/o tracer) (secs)	33.74	72.56	126.20	239.75	466.15
Output Time (with tracer) (secs)	31.58	70.67	135.18	259.85	463.87
Percentage Change (%)	-6.40	-2.60	+7.12	+8.38	-0.49
BT-IO					
Output Time (w/o tracer) (secs)	71.05	–	120.31	–	108.76
Output Time (with tracer) (secs)	70.70	–	121.31	–	104.23
Percentage Change (%)	-0.40	–	+0.80	–	-4.17

Table 6: Time to output for write benchmarks with and without RIOT tracing enabled

Figure 4 demonstrates that during execution of BT-IO, the number of concurrent POSIX write calls is generally lower when using PLFS. As each process is writing to its own unique file, each process has access to its own file stream, reducing file-system contention; this results in each POSIX write call completing much more quickly, reducing the number of concurrent calls waiting for access to the file-system.

It is interesting to note that despite the Minerva system having only two GPFS servers for parallel scratch space, a significant improvement in I/O performance is able to be achieved through the use of PLFS. This indicates that the system bottleneck at smaller core counts may be due to limited parallelism rather than limited capacity in the file system servers.

4.6 Tracing Run-time Overhead

Table 6 presents the run-time I/O overhead associated with the use of RIOT during parallel execution. Note that the overhead generated from these runs is less than 10% in all cases demonstrating a minimal addition to application runtime. Whilst the figures reported show a small increase in I/O time, this overhead is the result of timing and processing the system call – the times reported by RIOT include only the time taken for the system call to complete, and ignore the additional processing overhead.

A worst case slow-down of 9% experienced at application level demonstrates the low overhead achieved by RIOT, making it a viable tool for I/O analysis in a production environment.

5 Conclusions

Parallel input/output 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, but 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 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 present the RIOT toolkit – a lightweight I/O tracing library and a post-execution analysis framework. RIOT utilises MPI-IO and POSIX function interposition to intercept, benchmark and record information relating to file reads, writes and locking operations. Unlike a number of existing toolkits, RIOT specifically targets the profiling of POSIX calls from within MPI allowing users to obtain statistics relating to genuine file-system bandwidth and perceived MPI-bandwidth which includes the overhead associated with collective MPI file activity.

In the latter half of this paper we presented a case study in which our tool was applied to three standard industry I/O benchmarks. Using tracing data obtained from runs up to 256-cores we were able to demonstrate RIOT's ability to:

- Calculate effective MPI-IO read and write bandwidths as well as produce bandwidths for POSIX file-system calls originating from MPI-IO. The comparison of these two figures demonstrates the slow-down in per-process read-/write speed which results from the use of MPI. Typically these overheads arise because of collective negotiation between MPI ranks for lock ownership and the calculation of offsets at which reads or writes take place;

- Provide detailed read/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 were able to demonstrate the significant time spent in locking for HDF-5-based writes in the FLASH-IO benchmark (exceeding 11% of MPI file write times). The nature of this analysis allows lightweight, non-intrusive detection of potential bottlenecks in input/output activity providing a first point at which application designers can begin optimisation;

- Record and graphically represent file activity over time. Through the recording and post-execution analysis of trace logs, RIOT is able to graphically represent an application's file activity in parallel over time;

- 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-IO 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 is the first publication to use the new RIOT toolkit to assess parallel input/output activity. 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 set-ups, such as that utilised by BlueGene systems. It is expected that the performance characteristics seen on much larger systems will introduce additional complexities in measuring and visualising the I/O behaviour.

Acknowledgements

Access to the Minerva machine was provided by the Centre for Scientific Computing at the University of Warwick. Thanks go to Matt Ismail and Tony Arber for allowing early access to the machine.

References

1. Bailey, D., et al.: The NAS Parallel Benchmarks. Tech. Rep. RNR-94-007, NASA Ames Research Center (March 1994)
2. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The NAS Parallel Benchmarks. International Journal of High Performance Computing Applications 5(3), 63–73 (1991)
3. Bent, J., Gibson, G.A., Grider, G., Mcclelland, B., Nowoczynski, P., Nunez, J., Polte, M., Wingate, M.: PLFS: A Checkpoint Filesystem for Parallel Applications. In: Proceedings of the ACM/IEEE International Conference on Supercomputing Conference (SC'09) (November 2009)
4. Carns, P., Latham, R., Ross, R., Iskra, K., Lang, S., Riley, K.: 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)
5. Eckart, B., He, X., Ong, H., Scott, S.L.: An Extensible I/O Performance Analysis Framework for Distributed Environments. In: Proceedings of the 15th International European Conference on Parallel and Distributed Computing (Euro-Par'09). pp. 57–68 (August 2009)

6. Fryxell, B., Olson, K., Ricker, P., Timmes, F.X., Zingale, M., Lamb, D.Q., MacNeice, P., Rosner, R., Truran, J.W., Tufo, H.: FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes. The Astrophysical Journal Supplement Series 131(1), 273 (2000)
7. 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)
8. 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)
9. 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)
10. Polte, M., Simsa, J., Tantisiriroj, W., Gibson, G., Dayal, S., Chainani, M., Uppugandla, D.: 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)
11. Polte, M., Lofstead, J., Bent, J., Gibson, G., Klasky, S.A., Liu, Q., Parashar, M., Podhorszki, N., Schwan, K., Wingate, M., Wolf, M.: ... 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)
12. Rosner, R., Calder, A., Dursi, J., Fryxell, B., Lamb, D., Niemeyer, J., Olson, K., Ricker, P., Timmes, F., Truran, J., et al.: Flash Code: Studying Astrophysical Thermonuclear Flashes. Computing in Science & Engineering 2(2), 33–41 (2000)
13. Schmuck, F., Haskin, R.: GPFS: A Shared-Disk File System for Large Computing Clusters. In: Proceedings of the First USENIX Conference on File and Storage Technologies (FAST'02). pp. 231–244 (January 2002)
14. 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)
15. 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)
16. 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)
17. Uselton, A., Howison, M., Wright, N., Skinner, D., Keen, N., Shalf, J., Karavanic, K., Oliker, L.: 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)
18. 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)
19. 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)
20. Wolman, B., Olson, T.: IOBENCH: A System Independent IO Benchmark. ACM SIGARCH Computer Architecture News 17(5), 55–70 (1989)