Advance Access publication on 24 May 2012 doi:10.1093/comjnl/bxs044

Parallel File System Analysis Through Application I/O Tracing

S.A. WRIGHT[1,*], S.D. HAMMOND[2], S.J. PENNYCOOK[1], R.F. BIRD[1], J.A. HERDMAN[3], I. MILLER[3], A. VADGAMA[3], A. BHALERAO[1] AND S.A. JARVIS[1]

[1]*Performance Computing and Visualisation, Department of Computer Science, University of Warwick, Coventry, UK*
[2]*Scalable Computer Architectures, Center for Computing Research, Sandia National Laboratories, Albuquerque, NM, USA*
[3]*Supercomputing Solution Centre, UK Atomic Weapons Establishment, Aldermaston, Reading, UK*
**Corresponding author: steven.wright@warwick.ac.uk*

Input/Output (I/O) operations can represent a significant proportion of the run-time of parallel scientific computing applications. Although there have been several advances in file format libraries, file system design and I/O hardware, a growing divergence exists between the performance of parallel file systems and the compute clusters that they support. In this paper, we document the design and application of the RIOT I/O toolkit (RIOT) being developed at the University of Warwick with our industrial partners at the Atomic Weapons Establishment and Sandia National Laboratories. We use the toolkit to assess the performance of three industry-standard I/O benchmarks on three contrasting supercomputers, ranging from a mid-sized commodity cluster to a large-scale proprietary IBM BlueGene/P system. RIOT provides a powerful framework in which to analyse I/O and parallel file system behaviour—we demonstrate, for example, the large file locking overhead of IBM's General Parallel File System, which can consume nearly 30% of the total write time in the FLASH-IO benchmark. Through I/O trace analysis, we also assess the performance of HDF-5 in its default configuration, identifying a bottleneck created by the use of suboptimal Message Passing Interface hints. Furthermore, we investigate the performance gains attributed to the Parallel Log-structured File System (PLFS) being developed by EMC Corporation and the Los Alamos National Laboratory. Our evaluation of PLFS involves two high-performance computing systems with contrasting I/O backplanes and illustrates the varied improvements to I/O that result from the deployment of PLFS (ranging from up to 25× speed-up in I/O performance on a large I/O installation to 2× speed-up on the much smaller installation at the University of Warwick).

Keywords: high performance computing; input/output; MPI; checkpointing; file systems

Received 2 December 2011; revised 14 February 2012
Handling editor: Irfan Awan

1. INTRODUCTION

The substantial growth in the size of supercomputers—over two orders of magnitude in terms of processing elements since 1993—has created machines of extreme computational power and scale. As a result, users have been able to develop 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 and their deployment on increasingly sophisticated hardware, creating a perception that supercomputer *performance* is synonymous with the number of floating-point operations that can be performed each second.

One of the consequences of this has been that some of the vital contributors to application run-time have developed at a much slower rate. One such area is that of input and output (I/O), typically required to read data at the start of a run and write state information on completion.

As we advance towards exa-scale computing, the increasing number of compute components will have huge implications for system reliability. As a result, checkpointing—where the system state is periodically written to persistent storage so that, in the case of a hardware or software fault, the computation can be restored and resumed—is becoming common place. The cost of checkpointing is a slow-down at specific points

in the application in order to achieve some level of resilience. Understanding the cost of checkpointing, and the opportunities that might exist for optimizing this behaviour, presents a genuine opportunity to improve the performance of parallel applications at scale.

The Message Passing Interface (MPI) has become the *de facto* standard for managing the distribution of data and process synchronization in parallel applications. The MPI-2 [1] standard introduced MPI-IO, a library of functions designed to standardize the output of data to the file system in parallel. The most widely adopted MPI-IO implementation is ROMIO [2], which is used by both OpenMPI [3] and MPICH2 [4], as well as by a number of vendor-based MPI solutions [5, 6].

In addition to the standardization of parallel I/O through MPI, many file format libraries exist to further abstract low-level I/O operations (e.g. data formatting) from the application. Libraries such as HDF-5 [7], NetCDF [8] and Parallel NetCDF [9] allow applications to output data in a standardized format, enabling information to be more easily utilized by multiple parallel applications. Optimizations can also be made to a single library, creating improvements in the data throughput of many dependent applications.

Unfortunately this has, in part at least, encouraged code designers to treat these libraries as a black-box, instead of investigating and optimizing the data storage operations required by their applications. The result has been poor I/O performance that does not utilize the full potential of expensive parallel disk systems.

In this paper, we document the design and application of the RIOT I/O Toolkit (referred to throughout the remainder of this paper by the recursive acronym RIOT), first introduced in [10], to demonstrate the I/O behaviours of three standard benchmarks at scale on a variety of contrasting high-performance computing (HPC) systems. RIOT is a collection of tools specifically designed to enable the tracing and subsequent analysis of application I/O 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 recording of low-level parameters permits the analysis of I/O middleware, file format libraries, application behaviour and even the underlying file systems utilized by large clusters.

Specifically, this paper makes the following contributions:

(i) We present RIOT, an I/O tracer designed to intercept the file functions in the MPI-2 standard, as well as the low-level system calls triggered by the MPI-IO library. Our tool records not only the timing information, but also information relating to how much data is written and the file offset to which it is written. We also introduce a post-processing tool capable of generating statistical summaries and graphical representations of an application's parallel I/O activities;

(ii) Using RIOT, we analyse the I/O behaviour of three industry-standard benchmarks: the Block-Tridiagonal (BT) solver, from the NAS Parallel Benchmark (NPB) Suite; the FLASH-IO benchmark, from the University of Chicago and the Argonne National Laboratory (ANL); and IOR, a HPC file system benchmark that is used during procurement and file system assessment [11, 12]. Our analysis employs three contrasting platforms: a mid-size commodity cluster located at the University of Warwick, a large-scale capacity resource housed at the Open Computing Facility (OCF) at the Lawrence Livermore National Laboratory (LLNL) and a proprietary IBM BlueGene/P (BG/P) system installed at the Daresbury Laboratory in the UK;

(iii) Through using RIOT, we demonstrate the significant overhead associated with file locking on a small-scale installation of IBM's General Parallel File System (GPFS) and contrast this to a larger GPFS installation, as well as to a large-scale Lustre installation. We provide an analysis of both the locking semantics of the contrasting file systems as well as the different hardware configurations;

(iv) RIOT is the first tool, to our knowledge, to show the relationship between POSIX and MPI function calls and thus allow developers to analyse the POSIX file behaviour that is a direct result of MPI-IO calls. In Section 6.1, we utilize this ability to visualize the performance of the FLASH-IO benchmark, demonstrating a significant slow-down in performance due to the use of suboptimal MPI hints. We optimize the application's behaviour using MPI hint directives and achieve more than a 2× improvement in the write bandwidth;

(v) Finally, through an I/O trace analysis, we provide insight into the performance gains reported by the Parallel Log-structured File System (PLFS) [13, 14]—a novel I/O middleware being developed by EMC Corporation and the Los Alamos National Laboratory (LANL) to improve file write times. We show how improvements in the I/O performance can be demonstrated on relatively small parallel file systems and how large gains can be achieved on much larger installations. We also offer some insight into why this is the case and how PLFS reduces file system contention, improving the achievable bandwidth.

The remainder of this paper is structured as follows. Section 2 gives an overview of related work in the area of parallel I/O analysis and optimization. Section 3 outlines the design and implementation of RIOT. Section 4 describes the experimental set-up used in this paper, ranging from the configuration of the machines used to the three applications employed in this study. Section 5 demonstrates the low overhead of our tool, as well as RIOT operating on three different HPC systems, presenting the initial results for the three codes on the contrasting platforms. Section 6 presents a comparison of the different file systems in use on the machines used in our study. We present an analysis of the HDF-5 middleware library and we investigate PLFS, offering insight into the gains reported in [13] and finally,

Section 7 concludes the paper and offers potential avenues for future work.

2. BACKGROUND AND RELATED WORK

2.1. I/O benchmarking

The assessment of the file system performance, either during procurement or during system installation and upgrade, has resulted in a number of benchmarking utilities which attempt to characterize common read/write behaviour. Notable tools in this area include the IOR [11] and IOBench [15] parallel benchmarking applications. While these tools provide a good indication of *potential* performance, they are rarely indicative of the true behaviour of production codes. For this reason, a number of mini-application benchmarks have been created that extract file read/write behaviour from larger codes to ensure a more accurate representation of the application I/O. Examples include the BT solver application from the NPB Suite [16] and the FLASH-IO [17] benchmark from the University of Chicago—both of which are employed in this paper.

2.2. System monitoring and profiling tools

While benchmarks may provide a measure of file system performance, their use in diagnosing problem areas or identifying optimization opportunities within large codes is limited. For this activity, monitoring or profiling tools are required to either sample the system's state or record the file read and write behaviour of parallel codes in real-time.

The tools `iotop` and `iostat` both monitor a single workstation and record a wide range of statistics ranging from the I/O busy time to the CPU utilization [18]. `iotop` is able to provide statistics relevant to a particular application, but this data is not specific to a particular file system mount point. `iostat` can provide more detail that can be targeted to a particular file system, but does not provide application-specific information. These two tools are targeted at single workstations, but there are many distributed alternatives, including Collectl [19] and Ganglia [20].

Ganglia and Collectl both operate using a daemon process running on each compute node and therefore require some administrative privileges to install and operate correctly. Data about the system's state is sampled and stored in a database; the frequency of sampling therefore dictates the overhead incurred on each node. The I/O statistics generated by the tools focus only on low-level POSIX system calls and the load on the I/O backend and, therefore, much of the data will be inclusive of the calls made by other running services and applications. Furthermore, the data generated by these applications may not include information relating to file locks, or the file offsets currently being operated on. These applications, therefore, have a limited use for application analysis and optimization, since they do not provide an appropriate amount of application-specific data (Table 1). It is for this reason that many large multi-science HPC laboratories [e.g. ANL, Lawrence Berkeley National Laboratory (LBNL)] have developed alternative tools.

Application profiling tools can be used to generate more detailed information about a particular execution on a function-by-function basis. The data produced relates only to the specific application being traced along with its library (e.g. MPI) and low-level system calls.

Using function interpositioning, an intermediate library can intercept communications between the application and the underlying file system to analyse the behaviour of I/O intensive applications. Intercepting the POSIX and MPI file operations is the approach taken by RIOT; Darshan [21], developed at ANL and the Integrated Performance Monitoring (IPM) suite of tools [22], from LBNL.

Darshan has been designed to record file accesses over a prolonged period of time, ensuring that each interaction with the file system is captured during the course of a mixed workload. As described in [21], the aim of this work is to monitor I/O activity for a substantial amount of time on a production BG/P machine in order to guide developers and administrators in tuning the I/O backplanes used by large machines.

Similarly, IPM [23] 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. On the basis of this analysis, the authors were able to optimize two applications, achieving a 4× improvement in the I/O performance.

TABLE 1. Feature comparison between a collection of cluster I/O monitoring tools (Ganglia and Collectl) and application I/O profiling tools (Darshan, IPM and RIOT).

	Ganglia	Collectl	Darshan	IPM	RIOT
Monitoring level	System	System	Application	Application	Application
Monitoring style	Sampled	Sampled	Continuous	Continuous	Continuous
Syscall monitoring	Limited	Limited	POSIX-IO	POSIX-IO	POSIX-IO
MPI monitoring	None	None	MPI-IO	Complete	MPI-IO
Statistics collection	Counters	Counters	Counters	Counters	Full trace
Statistics reported	Per-node	Per-node	Per-rank	Per-rank	Per-rank

Darshan and IPM both collect data using counters to record the I/O statistics. In contrast, RIOT records all I/O events in the memory and thus provides users with a full trace of file activities. As a result, RIOT's post-processing tools can relate individual POSIX operations to their parent MPI-IO function calls, permitting analysis not only of the application's I/O behaviour but also of the underlying POSIX file behaviour induced by the use of MPI-IO. This data can then be used to analyse the performance of a particular piece of middleware. Furthermore, RIOT is able to highlight any deficiencies that may exist within a ROMIO file system driver or provide guidance as to which MPI hints may benefit a particular application or file system.

Table 1 summarizes the features of each of the tools described above.

2.3. Distributed file systems

The I/O backplane of high-performance clusters is generally provided by a distributed file system. The two most widely used file systems are IBM's GPFS [24] and the Lustre File System [25], both of which are analysed in this study. While both ultimately serve the same purpose, their architectures are somewhat different.

A Lustre installation consists of a number of *Object Storage Servers* (OSS) and a single, dedicated Metadata Server (MDS). Conversely, GPFS uses a number of I/O servers, and distributes metadata over each of them. While the MDS in Lustre uses its own hard drives to store metadata (e.g. directory tree, file structure), GPFS can be configured to store this data either to the same disks as the raw data, or to higher performance metadata-specific disks, depending on the configuration.

2.4. Virtual file systems

In addition to distributed file systems, a variety of virtual file systems have been produced to further improve performance. PLFS [13] and Zest [26] have both been shown to improve the file write 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 [27].

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 as 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 [28, 29], 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. Through our tracing tools, we offer an in-depth analysis of the benefits offered by PLFS (Section 6.3).

We previously introduced RIOT in [10, 30]. In this paper, we significantly extend this work as follows:

(i) We utilize a custom I/O benchmark designed specifically to assess the impact of using RIOT on I/O intensive applications. We demonstrate that the performance overheads incurred through the use of RIOT are minimal, thus making it an appropriate tool for tracing large and long-running production-grade codes;
(ii) We demonstrate the first applications of RIOT on a proprietary BlueGene system, IBM's highly scalable, low-power massively parallel HPC platform;
(iii) We present a detailed application of RIOT in a comparative study of the I/O performance of a mid-range and a large-scale commodity cluster, and also the IBM BG/P. The contrasting I/O configuration of these three platforms are extensively evaluated, offering insight into potential future designs and also demonstrating the versatility of RIOT;
(iv) We complement our previous analysis of collective buffering [30] with an assessment of the use of dedicated I/O aggregators, such as the dedicated I/O nodes found in the BG/P. Our results show that collecting the data from many nodes prior to committing the data to the file system can lead to exceptional I/O performance;
(v) We utilize RIOT's ability to visualize file write behaviour to analyse the write pattern used by HDF-5 with data-sieving enabled. This data clearly demonstrate how a single MPI write operation is decomposed into a series of smaller POSIX lock, read, write, unlock cycles;
(vi) Through a new case study, we demonstrate that a RIOT trace can be used to detect performance bottlenecks in an application's I/O. By disabling data-sieving (a problem highlighted by RIOT) and enabling collective buffering in the FLASH-IO benchmark, we increase the achievable bandwidth of this industry-standard benchmark by 2× on two of our three test systems.

3. RIOT OVERVIEW

The left-hand side of Fig. 1 depicts the usual flow of I/O in parallel applications; generally, applications either utilize the MPI-IO file interface directly, or use a third party library such as HDF-5 or NetCDF. In both cases, MPI is ultimately used to perform the read and write operations. In turn, the MPI library calls upon the MPI-IO library which, in the case of both OpenMPI and MPICH, is the ROMIO implementation [2]. The ROMIO file system driver [31] then calls the system's POSIX file operations to read and write the data to the file system. In this paper, we utilize the PLFS ROMIO file system driver during our experiments.

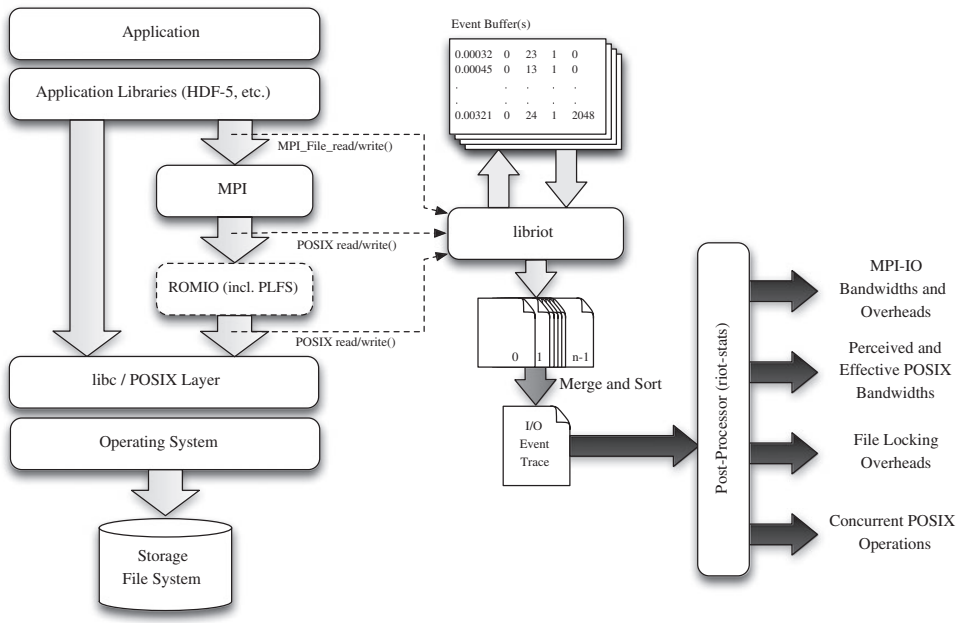

FIGURE 1. Tracing and analysis workflow using RIOT.

RIOT is an I/O tracing tool that can be utilized either as a dynamically loaded library (via run-time pre-loading and linking) or as a static library (linked at compile time). In the case of the former, the shared library uses function interpositioning to place itself in the library stack immediately prior to execution. When compiled as a dynamic library, RIOT redefines several functions from the POSIX system-layer and MPI libraries—when the running application makes calls to these functions, control is instead passed to handlers in the RIOT library. These handlers allow the original function to be performed, timed and recorded into a log file for each MPI rank. By using the dynamically loadable `libriot`, application recompilation is avoided; RIOT is therefore able to operate on existing application binaries and remain compiler and implementation language agnostic.

For situations where dynamic linking is either not desirable or is only available in a limited capacity (such as in the BG/P system used in this study), a static library can be built. A compiler wrapper is used to compile RIOT into a parallel application using the `-wrap` functionality found in the Linux linker.

As shown in Fig. 1, `libriot` intercepts I/O calls at three positions. In the first instance, MPI-IO calls are intercepted and redirected through RIOT, using either the PMPI interface, or dynamic or static linking. In the second instance, POSIX calls made by the MPI library are intercepted, and in the final instance any POSIX calls made by the ROMIO file system interface are caught and processed by RIOT.

Traced events in RIOT are recorded in a buffer stored in main memory. While the size of the buffer is configurable, our experiments have led us to set the buffer size at 8 MB. This allows ∼340 000 operations to be stored before needing to be flushed to the disk. This delay of logging (by storing events in memory) may have a small affect on the compute performance (since the memory access patterns may change), but storing trace data in memory helps to prevent any distortion of the application's I/O performance. In the event that the buffer becomes full, the data is written out to disk and the buffer is reset. This repeats until the application has terminated.

Time coherence is maintained across multiple nodes, by overloading the `MPI_Init` function to force all ranks to wait on an `MPI_Barrier` before each resetting their respective timers. This allows us to order events even after nodes have experienced a degree of time drift.

After the recording of an application trace is complete, a post-execution analysis phase can be conducted (shown on the right-hand side of Fig. 1).

During execution RIOT builds a file lookup table and for each operation only stores the time, the rank, a file identifier, an operation identifier and the file offset. After execution, these log files are merged and time-sorted into a single master log file, as well as a master file database. Using the information stored, RIOT can:

(i) produce a complete run-time trace of the application's I/O behaviour;
(ii) demonstrate the file-locking behaviour of a particular file system;
(iii) calculate the effective POSIX bandwidth achieved by MPI to the file system;

(iv) visualize the decomposition of an MPI file operation into a series of POSIX operations;
(v) demonstrate how POSIX operations are queued and then serialized by the I/O servers.

Throughout this paper, we make a distinction between effective MPI-IO and POSIX bandwidths—*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 and called directly by the MPI library. We make this distinction due to the inability to accurately report the perceived POSIX bandwidth due to the non-deterministic nature of parallel POSIX operations. The perceived POSIX bandwidth is therefore bounded below by the *perceived* MPI bandwidth (since the POSIX bandwidths must necessarily be at least as fast as the MPI bandwidths), and is bounded above by the *effective* POSIX bandwidth multiplied by the number of ranks (assuming a perfect parallel execution of each POSIX operation).

4. EXPERIMENTAL SET-UP

In this paper, we demonstrate RIOT working on three distinct HPC systems. We utilize the recently installed Minerva supercomputer, located at the University of Warwick's Centre for Scientific Computing, the Sierra cluster from the OCF at LLNL and, finally, the IBM BG/P proprietary system housed at the Daresbury Laboratory in the United Kingdom.

Minerva and Sierra are built from similar components, utilizing commodity Intel Xeon dual-socket hex-core Westmere-EP processors, clocked at 2.66 and 2.80 GHz, respectively. The interconnect between Minerva nodes is QLogic's TrueScale 4X QDR InfiniBand, offering a theoretical maximum bandwidth of 32 Gb/s. Each Minerva node is connected through InfiniBand to its two I/O nodes. Similarly, Sierra compute nodes are also connected via InfiniBand to the 24-node storage system utilized in this study—the I/O backplane used in these experiments is part of LLNL's 'islanded I/O' network, whereby many large I/O systems are shared between multiple clusters in the computing facility.

The BG/P system used for the experiments in this paper is a single cabinet, consisting of 1024 compute nodes. Each node contains a single quad-processor compute card clocked at 850 MHz. The BlueGene features dedicated networks for point-to-point communications and MPI collective operations. File system and complex operating system calls (such as timing routines) are routed over the MPI collective tree to specialized higher-performance login or I/O nodes enabling the design of the BlueGene compute node kernel to be significantly simplified to reduce the background compute noise. For these experiments, we utilize a compute-node to I/O node ratio of 1:32; however, differing ratios are provided by IBM to support varying levels of workload I/O intensity.

As well as variation in compute architecture and size, the machines selected utilize different file systems for their I/O backends. Minerva employs IBM's GPFS, whereas the file system used for our experimentation on Sierra is formatted for use with Lustre. The BG/P system also uses GPFS, but does so with faster hard disks and a higher density of I/O servers than Minerva.

A detailed specification of the three machines utilized in this study can be found in Table 2.

4.1. I/O benchmarks

For this study, we have selected three applications which are representative of a broad range of high-performance applications.

We employ a standard benchmark (IOR) that can be customized to recreate typical, or problematic I/O behaviours, as well as being customized to use either an MPI-IO interface or the HDF-5 application library.

Two additional applications have also been chosen (FLASH-IO, BT) that recreate the I/O behaviour of much larger codes but with reduced compute time and less configuration required than the parent version. This permits the investigation of system configurations that may have an impact on the I/O performance of the larger codes, without requiring considerable machine resources.

The three applications used in this study are the following:

(i) *IOR* [11, 12]: A parametrized benchmark that performs I/O operations through both the HDF-5 and MPI-IO interfaces. In this study, IOR has been configured to write 256 MB per process to a single file in 8 MB blocks. IOR's write performance through both MPI-IO and HDF-5 are assessed.
(ii) *FLASH-IO* [32, 33]: This benchmark replicates the checkpointing routines found in FLASH [17, 34], 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 $\sim$205 MB to disk through the HDF-5 library.
(iii) *BT* [16, 35]: An application from the NPB Suite which has been configured by NASA to replicate I/O behaviours from several important internal production codes. A variety of possible problem sizes are available but our focus is the C problem class ($162 \times 162 \times 162$), writing a data-set of $\sim$6.4 GB.

We note that since all three machines are production platforms, and results are subject to variation, all data is derived from five runs; where appropriate the mean is reported.

TABLE 2. Benchmarking platforms used in this study.

	Minerva	Sierra	BG/P	
Processor	Intel Xeon 5650	Intel Xeon 5660		PowerPC 450
CPU speed	2.66 Ghz	2.8 Ghz		850 Mhz
Cores per node	12	12		4
Nodes	258	1849		1024
Interconnect	QLogic TrueScale 4X QDR InfiniBand		3D torus	Collective tree
File system	GPFS	Lustre		GPFS
I/O servers/OSS	2	24		4
Theoretical bandwidth	∼4 GB/s	∼30 GB/s		∼6 GB/s
Storage disks				
Number of disks	96	3600	110	35
Disk size	2 TB	450 GB	147 GB	500 GB
Disk speed	7200 RPM	10 000 RPM	15 000 RPM	7200 RPM
Bus type	Nearline SAS	SAS	Fibre channel	S-ATA
Raid level	6 (8 + 2)	6 (8 + 2)		5 (4 + 1)
Metadata disks				
Number of disks	24	30 (+2)[a]		N/A
Disk size	300 GB	147 GB		N/A
Disk speed	15 000 RPM	15 000 RPM		N/A
Bus type	SAS	SAS		N/A
Raid level	10	10		N/A

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

5. PERFORMANCE ANALYSIS

5.1. RIOT performance analysis

We first use an I/O benchmark specifically designed to assess the overheads that our toolkit may introduce at run-time. Since RIOT requires all I/O functions to be interposed, we utilize a custom benchmark designed to perform a known set of read and write operations over a series of files. Write sizes of 256 KB, 1, 4 and 16 MB per process were performed 100 times for both read and write operations.

Table 3 shows the time taken to perform 100 operations, each 4 MB in size (other file sizes demonstrate similar effects and so the results are therefore not shown), in three configurations: (i) without RIOT, (ii) with RIOT configured to only trace POSIX file calls and, (iii) with RIOT performing a full trace of MPI and POSIX file activity. Our benchmark reports timings for the six MPI-IO functions that we believe to be the most commonly used in scientific codes. It should be noted that we would expect the overhead of RIOT associated with other MPI-IO functions to be approximately the same, since the function interposition and timing routines would be equivalent.

In Table 3, we see that RIOT adds minimal overhead to an application's run-time, although it is particularly difficult to precisely quantify this overhead since the machines employed operate production workloads. In some cases, the overheads appear negligible (or indeed present themselves as slight improvements over the original run-time). We account for this by the fact that the machines and their I/O backplanes are shared resources and are therefore subject to a small (≃10%) variation in run-time performance. Despite this variability (which is a feature of any large-scale production system), the low variation between run-times with and without RIOT loaded demonstrates the minimal impact that our toolkit has on application run-time. This is a key feature of our design and is an important consideration for profiling activities associated with large codes that may already take considerable lengths of time to run in their own right.

5.2. I/O subsystem performance

We next use RIOT to trace the write behaviour of the three codes in five different configurations. Figure 2 shows the perceived bandwidth achieved by each of the benchmarks on the three different systems. It is interesting to note that the two HDF-5-based codes (FLASH-IO and IOR through HDF-5) perform significantly worse than the other two codes on both Minerva and Sierra. The parallel HDF-5 library, by default, attempts to utilize data-sieving in order to transform many discontinuous small writes into a single much larger write. To do this, a large region (containing the target file locations) is locked and read into memory. The small changes are then made to the block in memory, and the data is then written back out

TABLE 3. Average time (s) to perform a hundred 4 MB operations: without RIOT, with only POSIX tracing and with complete MPI and POSIX RIOT tracing.

		Minerva			Sierra			BG/P		
Function	Tracing level	24	48	96	24	48	96	32	64	128
MPI_File_write	None	44.00	84.54	150.91	44.75	60.32	119.72	71.31	120.26	262.96
	POSIX	44.22	93.05	150.72	46.78	67.07	113.41	68.99	113.52	256.70
	Complete	44.36	84.66	155.48	46.33	70.72	123.89	69.00	116.13	256.89
MPI_File_write_all	None	26.39	50.62	100.96	36.25	71.13	129.69	71.46	101.24	140.40
	POSIX	26.17	51.95	101.08	38.59	68.44	131.63	70.59	100.93	135.17
	Complete	26.99	50.66	99.95	38.11	64.32	127.30	70.70	100.09	139.02
MPI_File_write_at_all	None	12.91	25.83	57.00	37.52	70.68	127.12	61.37	69.50	96.83
	POSIX	11.81	28.27	55.39	36.64	63.91	126.87	61.48	70.87	98.30
	Complete	13.20	27.08	54.51	36.27	73.06	135.99	60.78	72.06	96.96
MPI_File_read	None	7.26	12.22	25.81	2.77	5.41	15.04	47.74	56.97	182.20
	POSIX	6.73	11.99	25.10	5.83	6.66	19.57	48.45	57.27	190.51
	Complete	7.05	12.46	26.10	2.36	5.00	16.42	48.41	57.60	179.90
MPI_File_read_all	None	21.09	26.03	40.66	15.81	21.56	49.47	55.92	65.89	203.63
	POSIX	19.85	27.17	42.31	18.63	24.73	58.97	56.27	64.52	211.50
	Complete	20.67	26.62	41.24	16.99	25.74	63.82	58.97	68.57	209.91
MPI_File_read_at_all	None	2.21	3.61	6.36	4.41	4.56	5.15	36.73	37.89	41.23
	POSIX	2.33	3.79	5.87	4.14	4.72	5.56	36.45	39.97	44.06
	Complete	2.14	3.70	5.60	4.83	4.71	5.31	38.53	40.63	45.33
Average overhead of RIOT (%)		0.48	1.15	0.31	2.39	4.23	5.95	0.54	0.74	0.08

to persistent storage in a single write operation. While this offers a large improvement in performance for small unaligned writes [36], many HPC applications are constructed to perform larger sequential file operations.

The use of file locks will help to maintain file coherence but, as RIOT is able to demonstrate, when writes do not overlap, the locking, reading and unlocking of file regions may create a significant overhead—this is discussed further in Section 6.1.

Also of note in Fig. 2d is that BT performs significantly better on the BG/P after 256 processors are used (note the logarithmic scale). Owing to the architecture of the machine and the relatively small amount of data that each process writes at this scale, the data is flushed very quickly to the I/O node's cache and this gives the illusion that the data has been written to disk at speeds in excess of 1 GB/s. For much larger output sizes the same effect is not seen, since the writes are much larger and therefore cannot be flushed to the cache at the same speed. We note that while the I/O performance of Minerva and Sierra plateau quite early, the I/O performance of the BG/P system does not. While a commodity cluster using MPI will often use ROMIO hints such as *collective buffering* [37] to reduce the contention for the file system, the BG/P performs what could be considered 'super' collective buffering, where 32 nodes send all of their I/O traffic through a single I/O node. This results in the exceptional scaling behaviour observed in Fig. 2d. As the output size and the number of participating nodes increase, contention begins to affect performance as Fig. 2a demonstrates.

The write performance on each of the commodity clusters is roughly equivalent to the write speed of a single disk. When we consider that these systems consist of hundreds (or thousands) of disks, configured to write in parallel, it is clear that the full potential of the hardware is not being realized. If we analyse the effective bandwidth of each of the codes (i.e. the total amount of data written divided by the total time taken by all nodes), it becomes apparent that data is being written very slowly onto the individual disks. Figures 3 and 4 and Table 4 show the effective MPI and POSIX bandwidths achieved by each of the codes on our three machines. The POSIX bandwidth is significantly higher than the MPI bandwidth, demonstrating a large overhead in MPI. However, even the POSIX bandwidth does not approach the potential achievable bandwidth of the machine.

We believe that much of this slow-down can be attributed to two main problems: (i) disk seek time and (ii) file system contention. In the former, since data is being accessed simultaneously from many different nodes and users, the file server must constantly seek for the information that it requires. In the latter case, since the reads and writes to a single file must maintain some degree of consistency, contention for a single file can become prohibitive.

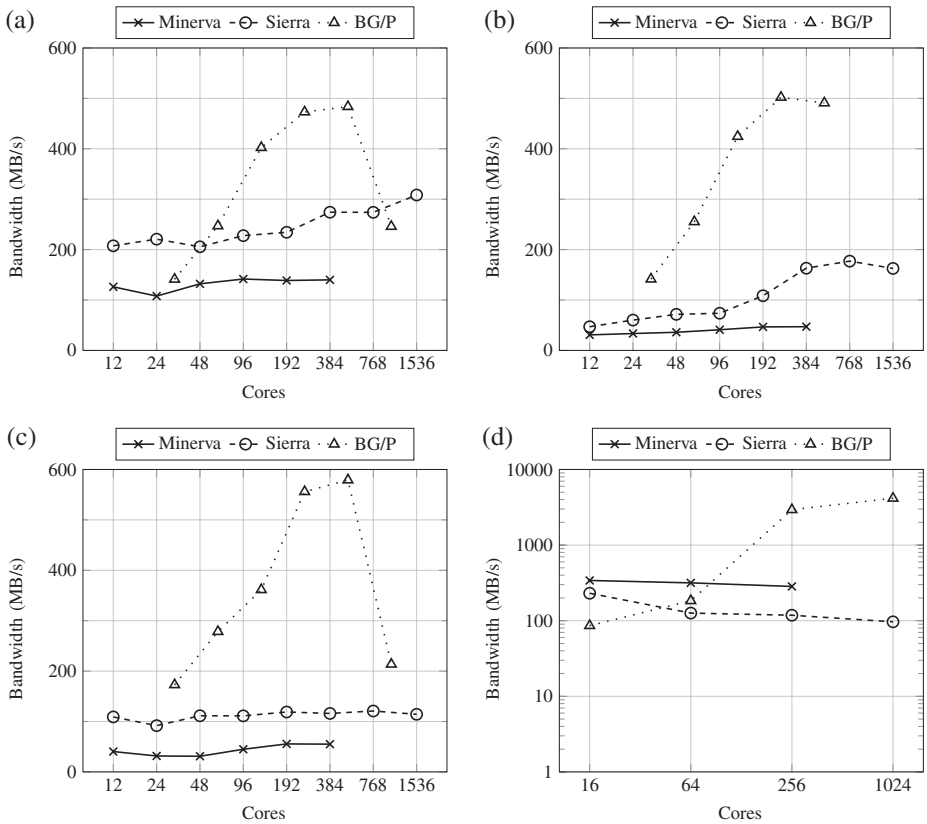

FIGURE 2. Perceived MPI write bandwidth for (**a**) IOR (through MPI-IO), (**b**) IOR (through HDF-5), (**c**) FLASH-IO (through HDF-5) and (**d**) BT Class C (through MPI-IO), measured using RIOT.

From the results presented in Figs 2–4 and Table 4, it is clear that Sierra generally has a much quicker I/O subsystem than Minerva. However, the BG/P's file system far outperforms both clusters when scaled. The unusual interconnect and architecture that it uses allows its compute nodes to flush their data to the I/O nodes cache quickly, allowing computation to continue. Similarly, when the writes are small, Minerva can be shown to outperform Sierra, mainly due to the locality of its I/O backplane. However, when HDF-5 is in use on Minerva, the achievable bandwidth is much lower than that of the other machines due to file-locking and the poor read performance of its hard disk drives.

6. FILE SYSTEM COMPARISON

As highlighted in Table 2, the I/O subsystem utilized by Minerva is much smaller than that used by Sierra. Each machine utilizes a different file system, providing us with an opportunity to not only compare the hardware differences between the machines, but also to compare the semantics of the differing file systems and MPI-IO ROMIO drivers. Furthermore, there exists a clear opportunity to find ways in which Minerva's I/O system could be better utilized to provide a higher level of service to users. Using RIOT, we undertake a comparative study of IBM's GPFS and the Lustre File System. We also explore the HDF-5 middleware library, as well as the effect of the PLFS virtual file system on the contrasting systems.

6.1. GPFS *vs.* Lustre

Through our experiments with FLASH-IO and IOR, both through HDF-5, a large performance gap has been identified between using the HDF-5 file format library and performing I/O directly via MPI-IO. Although a slight slow-down may be expected, since there is an additional layer in the software stack to traverse, the decrease in performance is quite large (up to a 50% slow-down). Figure 5 shows the percentage of the MPI file write time spent in each of the four main contributing POSIX file functions.

For the Minerva supercomputer, at low core counts, there is a significant overhead associated with file-locking (Fig. 5a). In the worst case, on a single node, this represents an approximate slow-down of 30% in performance. The reason for the use of file-locking in HDF-5 is that data-sieving is often utilized to write small unaligned blocks in much larger blocks. The penalty for this is that data must be read into memory prior to writing; this behaviour can prove to be a large overhead for many applications, where the writes may perform much better

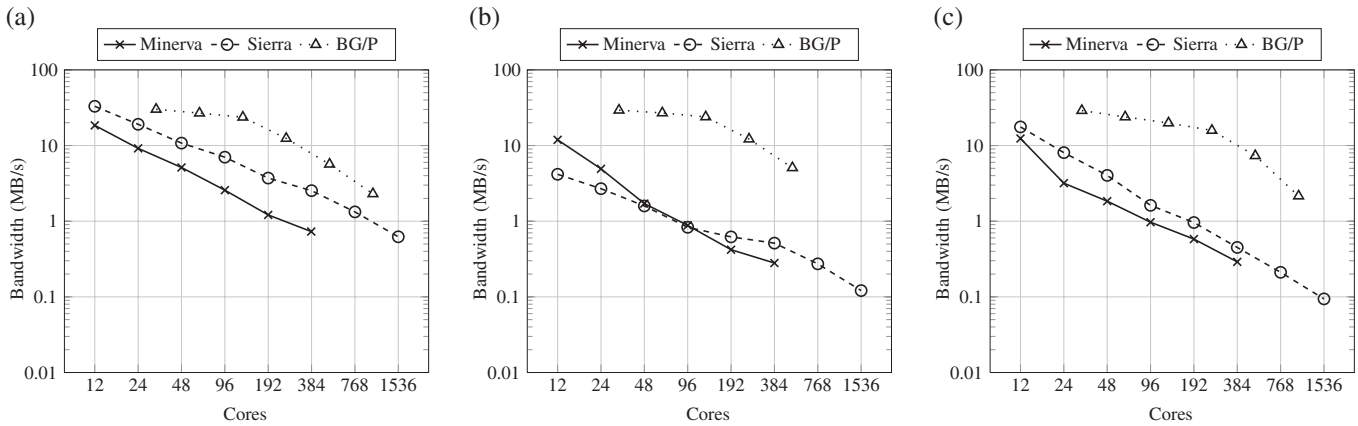

FIGURE 3. Effective POSIX bandwidth for (**a**) IOR (through MPI), (**b**) IOR (through HDF-5) and (**c**) FLASH-IO (through HDF-5), measured using RIOT.

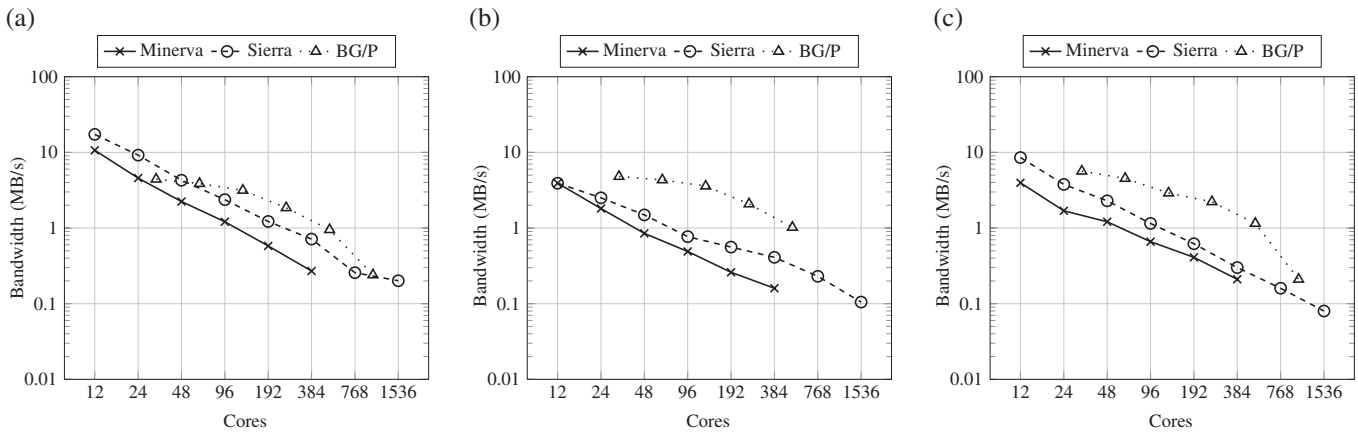

FIGURE 4. Effective MPI bandwidth for (**a**) IOR (through MPI), (**b**) IOR (through HDF-5) and (**c**) FLASH-IO (through HDF-5), measured using RIOT.

TABLE 4. Effective POSIX and MPI bandwidths (MB/s) achieved on the three platforms for BT.

		Processors			
		16	64	256	1024
Minerva	POSIX	159.65	84.90	24.48	
	MPI	21.41	4.95	1.11	
Sierra	POSIX	169.56	40.78	7.98	1.55
	MPI	14.49	1.94	0.44	0.09
BG/P	POSIX	132.10	49.06	21.10	8.75
	MPI	84.15	2.88	8.58	3.45

were data-sieving to be disabled. Figure 5c shows how the BG/P does not perform data-sieving and therefore there is no overhead associated with reading data into memory. However, due to the use of dedicated I/O nodes, the compute nodes spend ∼80% of their MPI write time waiting for the I/O nodes to complete.

In contrast to Minerva, the same locking overhead is not experienced by Sierra; however, up to 20% of the MPI write time is spent waiting for other ranks. It is also of note that Minerva's I/O subsystem is backed by relatively slow 7200 RPM 2 TB Nearline SAS hard drives; Sierra, on the other hand, uses much quicker 10 000 RPM 450 GB SAS enterprise-class hard disk drives, providing a much smaller seek time, a much greater bandwidth and various other performance advantages (e.g. greater rotational vibration tolerance). As a consequence of this, a single Sierra I/O node can service a read request much more quickly than one of Minerva's, providing an overall greater level of service.

Despite Sierra having 12 times more I/O servers, nearly 40 times more disks (which also spin faster and are connected through a faster bus connection), its performance is not significantly greater (as demonstrated in Fig. 2). One explanation for this is that, ultimately, reads and writes to a single file must be serialized, in part at least, by the I/O servers.

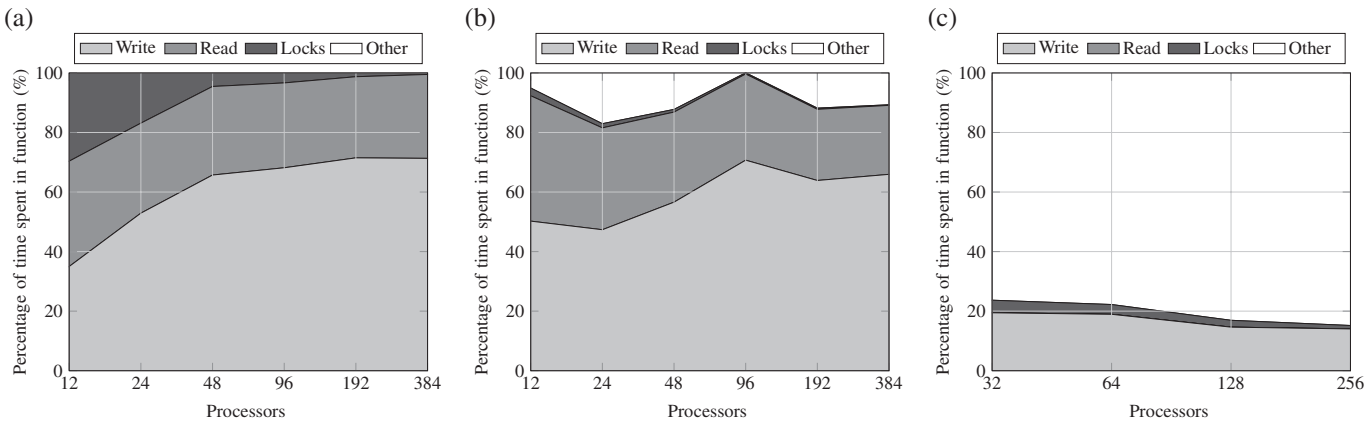

FIGURE 5. Percentage of time spent in POSIX functions for FLASH-IO on (**a**) Minerva, (**b**) Sierra and (**c**) BG/P, measured using RIOT.

We next explore two ways to improve the performance of I/O intensive codes. First, we utilize RIOT to analyse why we experience such a slowdown with HDF-5 based applications on the Minerva and Sierra supercomputers; secondly, we use RIOT to analyse the behaviour of the PLFS virtual file system, developed at LANL, to gain more understanding into how a log-based file system and transparent file partitioning can offer such impressive improvements in the achievable bandwidth.

6.2. Middleware analysis: HDF-5

Using RIOT's tracing and visualization capabilities, the execution of a small run of the FLASH-IO benchmark (using a $16 \times 16 \times 16$ grid size and only two processors) is investigated. Figure 6 shows the composition of a single MPI-IO write operation in terms of its POSIX operations. Rank 0 (Fig. 6a) spends the majority of its write time performing read, lock and unlock operations, whereas Rank 1 (Fig. 6b) spends much of its time performing only lock, unlock and write operations. Since Rank 1 writes to the end of the file, increasing the end-of-file pointer, there is no data for it to read in during data-sieving; Rank 0, on the other hand, will always have data to read (since there will be zeroes present between its current position and the new end-of-file pointer).

Also of interest is the fact that both processors are splitting one large write into five lock, read, write, unlock cycles. This is indicative of using data-sieving, with the default 512 KB buffer, to write ∼2.5 MB of data. When performing a write of this size, where all the data is 'new', data-sieving may be detrimental to performance. To test this hypothesis, we located the `MPI_Info_set` operations in the FLASH-IO source code (used to set the MPI-IO hints) and added additional operations to disable data-sieving. We then performed the same experiment in order to visualize the write behaviour with data-sieving disabled. Figure 7 shows that now the MPI-IO write operation is consumed by a single write operation, and the time taken to perform the write is shorter than that found in Fig. 6.

Using the problem size benchmarked previously ($24 \times 24 \times 24$), we performed further executions of FLASH-IO on Minerva and Sierra using between 1 and 32 compute nodes (12 to 384 processors) in three configurations: first, in its original configuration; secondly, with data-sieving disabled and, finally, with collective buffering enabled and data-sieving disabled. Figure 8a demonstrates the resulting improvement on Minerva, showing a 2× increase in the write bandwidth over the unmodified code. Better performance is observed when using collective buffering. On Sierra (Fig. 8b), we see a similar improvement in performance (∼2× improvement in bandwidth). We also see that, on a single node (12 processor cores), performing only data-sieving is slightly faster than using collective buffering and beyond this we see that using collective buffering increases the bandwidth by between 5 and 20%.

It should be noted that this result does not mean that data-sieving will always decrease performance. In the case where data in an output file is being updated (rather than a new output file generated), using data-sieving to make small differential changes may improve performance.

6.3. Virtual file system analysis: PLFS

PLFS is an I/O interposition layer designed primarily for checkpointing and logging operations. PLFS intercepts MPI-IO calls through a ROMIO file system driver and transparently translates them 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 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 [14].

Table 5 presents the average perceived and effective MPI-IO and POSIX bandwidths achieved by the BT benchmark when

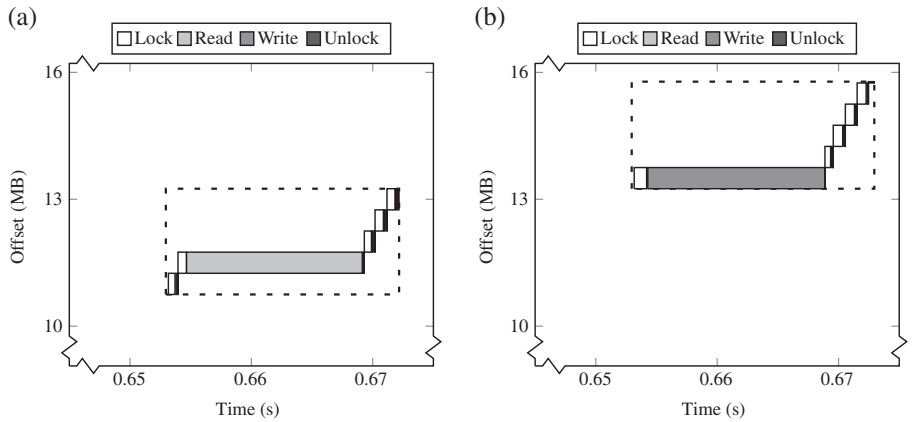

FIGURE 6. Composition of a single, collective MPI write operation on MPI ranks 0 and 1 ((**a**) and (**b**), respectively) of a two-processor run of FLASH-IO, called from the HDF-5 middleware library in its default configuration.

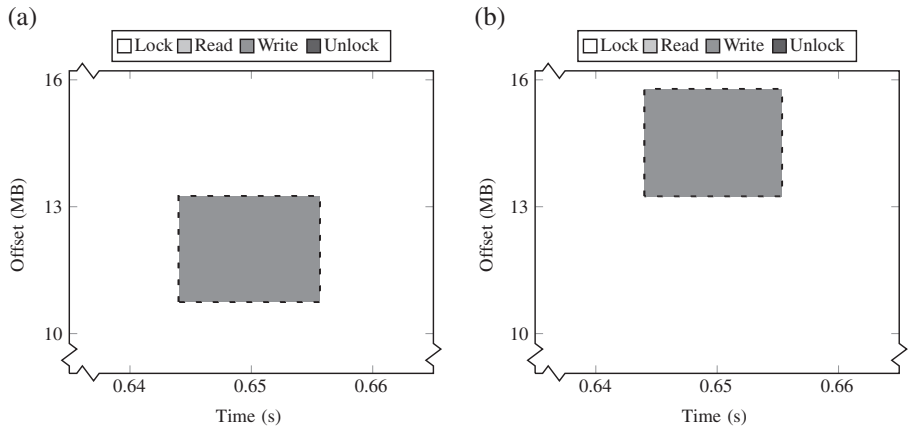

FIGURE 7. Composition of a single, collective MPI write operation on MPI ranks 0 and 1 ((**a**) and (**b**), respectively) of a two processor run of FLASH-IO, called from the HDF-5 middleware library after data-sieving has been disabled.

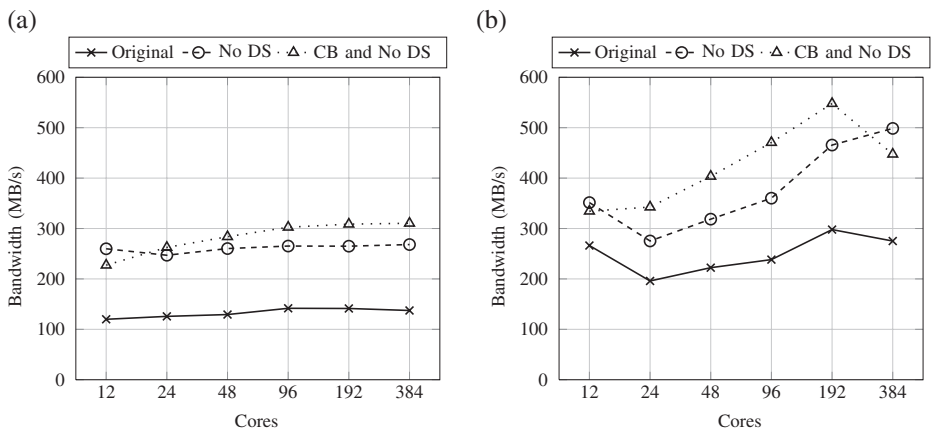

FIGURE 8. The perceived bandwidth for the FLASH-IO benchmark in its original configuration (Original), with data-sieving disabled (No DS) and with collective buffering enabled *and* data-sieving disabled (CB and No DS) on (**a**) Minerva and (**b**) Sierra, as measured by RIOT.

TABLE 5. Perceived and effective bandwidth (MB/s) for BT Class C through MPI-IO and PLFS, measured using RIOT.

		MPI-IO			PLFS		
		16	64	256	16	64	256
Perceived bandwidth	Minerva	223.36	53.76	61.44	276.32	224.64	125.44
	Sierra	222.24	126.08	115.20	337.12	1518.08	3118.08
Effective POSIX bandwidth	Minerva	12.39	1.72	0.83	72.07	36.60	8.86
	Sierra	169.56	40.78	7.98	235.44	538.13	437.88
Effective MPI bandwidth	Minerva	13.96	0.84	0.24	17.27	3.51	0.49
	Sierra	13.89	1.97	0.45	21.07	23.72	12.18

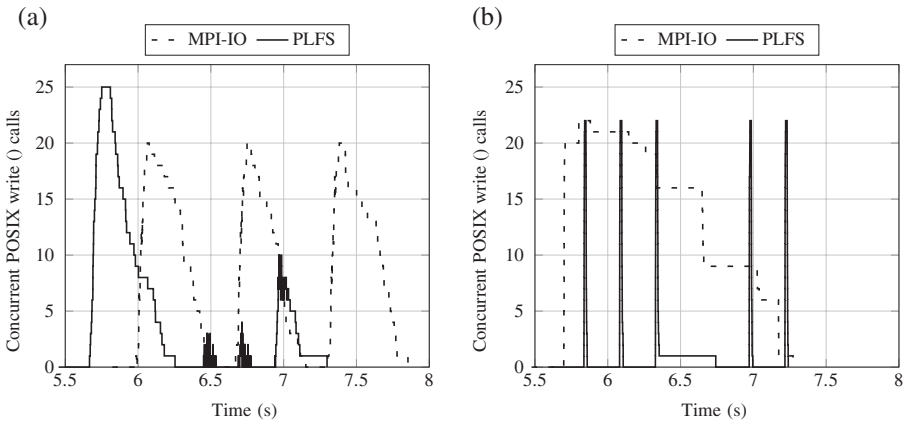

FIGURE 9. Concurrent `write()` operations for BT Class C on 256 processors on (**a**) Minerva and (**b**) Sierra.

running with and without the PLFS ROMIO file system driver. Note that, as previously, effective bandwidth in this table refers to the bandwidth of the operations as if called serially and hence are much lower than the perceived bandwidths.

When not using PLFS we see that the effective POSIX write bandwidth decreases as the applications are scaled. PLFS partially reverses this trend as writes are no longer dependent on operations performed by other processors and can therefore be flushed to the file server's cache much more quickly. 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 on 256 processors, PLFS increases the bandwidth from 115.2 MB/s perceived bandwidth up to 3118.08 MB/s on the Sierra cluster, representing a 27-fold increase in write performance.

Much smaller gains are seen on Minerva, but due to its rather limited I/O hardware, this is to be expected. There are fewer I/O servers to service read and write requests on Minerva and as a result there is much less bandwidth available for the compute nodes.

Figure 9 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, it has access to its own unique file stream, reducing file system contention. For non-PLFS writes, we see a stepping effect where all POSIX writes are queued and completed in a serialized, non-deterministic manner. Conversely, on larger I/O installations, PLFS writes do not exhibit this stepping behaviour, and on smaller I/O installations they exhibit this behaviour to a much lesser extent, as the writes are not waiting on other processes to complete.

7. 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 with that of microprocessors. Other causes include limited optimization at the code level and the use of complex file formatting libraries. Contemporary applications can often exhibit poor I/O performance because code developers lack an understanding of how their code utilizes I/O resources and how best to optimize for this.

In this paper, we document the design, implementation and application of RIOT, a toolkit with which these issues might be addressed. We demonstrate RIOT's ability to intercept, record and analyse information relating to file reads, writes and locking operations within three standard industry I/O benchmarks. RIOT has been utilized on two commodity clusters as well an IBM BG/P supercomputer.

The results generated by our tool illustrate the difference in performance between the relatively small I/O subsystem installed on the Minerva cluster and the much larger Sierra I/O backplane. Although there is a large difference in the size and complexity of these I/O systems, much of the performance differences originate from the contrasting file systems that they use. Furthermore, through using the BG/P located at Daresbury Laboratory, we have demonstrated that exceptional performance can be achieved on small I/O subsystems where dedicated I/O aggregators are used, allowing data to be quickly flushed from the compute node to an intermediate node.

RIOT provides the opportunity to:

(i) calculate not only the bandwidth perceived by a user, but also the effective bandwidth achieved by the I/O servers. This has highlighted a significant overhead in the MPI library, showing that the POSIX write operations to the disk account for little over half of the MPI write time. It has also been shown that much of the time taken by MPI is consumed by file-locking behaviours and the serialization of file writes by the I/O servers;

(ii) demonstrate the significant overhead associated with using the HDF-5 library to store data grids. Through our analysis, we have shown that on a small number of cores, the time spent acquiring and releasing file locks can consume nearly 30% of the file write time. Furthermore, on small-scale, multi-user I/O systems, reading data into memory, in order to perform data-sieving, can prove very costly;

(iii) visualize the write behaviour of the MPI when data-sieving is in use, showing how large file writes are segmented into many 512 KB lock, read, write, unlock cycles. Through adjusting the MPI hints to disable data-sieving, we have shown that on some platforms, and for some applications, data-sieving may degrade performance;

(iv) analyse the performance gains resulting from PLFS. In this paper, we have demonstrated a 25× speed-up on the Sierra supercomputer through using PLFS. The increased number of individual file streams allows an I/O server to better handle many concurrent write requests. Even on the much smaller Minerva cluster, PLFS was able to yield almost a 2× speed-up.

Next we plan to utilize the log files produced by RIOT to create an automated benchmark generator. We believe that RIOT can be used to create synthetic I/O benchmarks with which I/O configuration options for the host or file system can be quickly assessed. We also believe this offers an opportunity for many laboratories to release I/O benchmarks that recreate the I/O operations in classified, production-grade applications.

ACKNOWLEDGEMENTS

Access to the Minerva supercomputer was provided by the Centre for Scientific Computing at the University of Warwick. We are grateful to Scott Futral, Todd Gamblin, Jan Nunes and the Livermore Computing Team for access to, and help in using, the Sierra machine located at LLNL. We are also indebted to John Bent at EMC Corporation, and Meghan Wingate McClelland at LANL for their expert advice and support concerning PLFS. The authors would also like to thank the High Performance Computing team, and especially Colin Morey, at the Daresbury Laboratory (UK) for access to the IBM BG/P system.

FUNDING

This work is supported in part by The Royal Society through their Industry Fellowship Scheme (IF090020/AM). The performance modelling research is also supported jointly by AWE and the TSB Knowledge Transfer Partnership grant number KTP006740. Access to the LLNL OCF is made possible through collaboration with the UK Atomic Weapons Establishment under grants CDK0660 (The Production of Predictive Models for Future Computing Requirements) and CDK0724 (AWE Technical Outreach Programme). Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

REFERENCES

[1] Message Passing Interface Forum (1998) MPI2: a message passing interface standard. *High Perf. Comput. Appl.*, **12**, 1–299.

[2] ANL/MCS-TM-234 (1997) *ROMIO: A High-Performance, Portable MPI-IO Implementation*. Argonne, IL.

[3] Gabriel, E. *et al*. (2004) Open MPI: goals, concept, and design of a next generation MPI implementation. *Lect. Notes Comput. Sci. (LNCS)*, **3241**, 97–104.

[4] Gropp, W. (2002) MPICH2: a new start for MPI implementations. *Lect. Notes Comput. Sci. (LNCS)*, **2474**, 7–42.

[5] Almási, G. *et al*. (2004) Implementing MPI on the BlueGene/L supercomputer. *Lect. Notes Comput. Sci. (LNCS)*, **3149**, 833–845.

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

[7] Koziol, Q. and Matzke, R. (1998) *HDF5—A New Generation of HDF: Reference Manual and User Guide*. Champaign, IL, USA.

[8] Rew, R.K. and Davis, G.P. (1990) NetCDF: an interface for scientific data access. *IEEE Comput. Graph. Appl.*, **10**, 76–82.

[9] Li, J., Liao, W., Choundhary, A., Ross, R., Thakur, R., Gropp, W., Latham, R., Siegel, A., Gallagher, B. and Zingale, M. (2003) Parallel netCDF: A High-performance Scientific I/O Interface. *Proc. ACM/IEEE Int. Conf. Supercomputing (SC'03)*, Pheonix, AZ, November, pp. 39–50. ACM, New York, NY.

[10] Wright, S.A., Pennycook, S.J., Hammond, S.D. and Jarvis, S.A. (2011) RIOT—A Parallel Input/Output Tracer. *Proc. 27th Annual UK Performance Engineering Workshop (UKPEW'11)*, Bradford, UK, July, pp. 25–39. The University of Bradford, Bradford, UK.

[11] Shan, H., Antypas, K. and Shalf, J. (2008) Characterizing and Predicting the I/O Performance of HPC Applications using a Parameterized Synthetic Benchmark. *Proc. ACM/IEEE Int. Conf. Supercomputing (SC'08)*, November. IEEE Press, Piscataway, NJ, Austin, TX.

[12] LBNL-62647 (2007) *Using IOR to Analyze the I/O Performance for HPC Platforms*. Lawrence Berkeley National Laboratory, Berkeley, CA.

[13] Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M. and Wingate, M. (2009) PLFS: A Checkpoint Filesystem for Parallel Applications. *Proc. ACM/IEEE Int. Conf. Supercomputing Conf. (SC'09)*, Portland, OR, November, pp. 21:1–21:12. ACM, New York, NY.

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

[15] Wolman, B. and Olson, T. (1989) IOBENCH: a system independent IO benchmark. *ACM SIGARCH Comput. Archit. News*, **17**, 55–70.

[16] Bailey, D.H. *et al.* (1991) The NAS parallel benchmarks. *Int. J. High Perf. Comput. Appl.*, **5**, 63–73.

[17] Rosner, R. *et al.* (2000) Flash code: studying astrophysical thermonuclear flashes. *Comput. Sci. Eng.*, **2**, 33–41.

[18] Layton, J. (2012) *HPC storage—getting started with I/O profiling*. http://hpc.admin-magazine.com/Articles/HPC-Storage-I-O-Profiling (accessed 2 February 2012).

[19] (2011) *Collectl*. http://collectl.sourceforge.net (accessed 16 January 2012).

[20] (2011) *Ganglia monitoring system*. http://ganglia.sourceforget.net (accessed 16 January 2012).

[21] Carns, P., Latham, R., Ross, R., Iskra, K., Lang, S. and Riley, K. (2009) 24/7 Characterization of Petascale I/O Workloads. *Proc. IEEE Int. Conf. Cluster Computing and Workshops (CLUSTER'09)*, New Orleans, LA, September, pp. 1–10. IEEE Computer Society, Los Alamitos, CA.

[22] Fuerlinger, K., Wright, N. and Skinner, D. (2010) Effective Performance Measurement at Petascale Using IPM. *Proc. IEEE 16th Int. Conf. Parallel and Distributed Systems (ICPADS'10)*, Shanghai, China, December, pp. 373–380. IEEE Computer Society, Washington, DC.

[23] Uselton, A., Howison, M., Wright, N.J., Skinner, D., Keen, N., Shalf, J., Karavanic, K.L. and Oliker, L. (2010) Parallel I/O Performance: From Events to Ensembles. *Proc. IEEE Int. Symp. Parallel Distributed Processing (IPDPS'10)*, Atlanta, GA, April, pp. 1–11. IEEE Computer Society, Los Alamitos, CA.

[24] Schmuck, F. and Haskin, R. (2002) GPFS: A Shared-Disk File System for Large Computing Clusters. *Proc. 1st USENIX Conf. File and Storage Technologies (FAST'02)*, Monterey, CA, January, pp. 231–244. USENIX Association Berkeley, CA.

[25] Schwan, P. (2003) *Lustre: building a file system for 1000-node clusters*. http://lustre.org (accessed 23 October 2011).

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

[27] Polte, M., Simsa, J., Tantisiriroj, W., Gibson, G., Dayal, S., Chainani, M. and Uppugandla, D.K. (2008) Fast Log-based Concurrent Writing of Checkpoints. *Proc. 3rd Annual Workshop on Petascale Data Storage (PDSW'08)*, Austin, TX, November, pp. 1–4. IEEE Computer Society, Los Alamitos, CA.

[28] Wang, Y. and Kaeli, D. (2003) Source Level Transformations to Improve I/O Data Partitioning. *Proc. 1st Int. Workshop on Storage Network Architecture and Parallel I/Os (SNAPI'03)*, New Orleans, LA, September–October, pp. 27–35. ACM, New York, NY.

[29] Wang, Y. and Kaeli, D. (2003) Profile-Guided I/O Partitioning. *Proc. 17th Annual Int. Conf. Supercomputing (ICS'03)*, San Francisco, CA, June, pp. 252–260. ACM, New York, NY.

[30] Wright, S.A., Hammond, S.D., Pennycook, S.J. and Jarvis, S.A. (2011) Light-weight parallel I/O analysis at scale. *Lect. Notes Comput. Sci. (LNCS)*, **6977**, 235–249.

[31] Thakur, R., Gropp, W. and Lusk, E. (1996) An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. *Proc. 6th Symp. Frontiers of Massively Parallel Computation (FRONTIERS'96)*, Annapolis, MD, October, pp. 180–187. IEEE Computer Society, Los Alamitos, CA.

[32] Zingale, M. (2011) *FLASH I/O benchmark routine—parallel HDF 5*. http://www.ucolick.org/~zingale/flash_benchmark_io/ (accessed 21 February 2011).

[33] Argonne National Laboratory (2011) *Parallel I/O benchmarking consortium*. http://www.mcs.anl.gov/research/projects/pio-benchmark/ (accessed 21 February 2011).

[34] Fryxell, B., Olson, K., Ricker, P., Timmes, F., Zingale, M., Lamb, D., MacNeice, P., Rosner, R., Truran, J. and H.Tufo (2000) FLASH: an adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *Astrophys. J. Suppl. Ser.*, **131**, 273.

[35] RNR-94-007 (1994) *The NAS Parallel Benchmarks*. NASA Ames Research Center, Moffet Field, CA.

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

[37] Nitzberg, B. and Lo, V. (1997) Collective Buffering: Improving Parallel I/O Performance. *Proc. 6th IEEE Int. Symp. High Performance Distributed Computing (HPDC'97)*, Portland, OR, August, pp. 148–157. IEEE Computer Society, Washington, DC.