Tools for Analyzing Parallel I/O

Julian M. Kunkel,[1] Eugen Betke,[2] Matt Bryson,[3] Philip Carns,[4] Rosemary
Francis,[5] Wolfgang Frings,[6] Roland Laifer,[7] Sandra Mendez[8]

1: University of Reading, 2: German Climate Computing Center (DKRZ),
3: University of California, Santa Cruz, 4: Argonne National Laboratory, 5: Ellexus
Ltd, 6: Jülich Supercomputing Centre (JSC), 7: Karlsruhe Institute of Technology
(KIT), 8: Leibniz Supercomputing Centre (LRZ)

Abstract. Parallel application I/O performance often does not meet
user expectations. Additionally, slight access pattern modifications may
lead to significant changes in performance due to complex interactions
between hardware and software. These issues call for sophisticated tools
to capture, analyze, understand, and tune application I/O.

In this paper, we highlight advances in monitoring tools to help address
these issues. We also describe best practices, identify issues in measure-
ment and analysis, and provide practical approaches to translate parallel
I/O analysis into actionable outcomes for users, facility operators, and
researchers.

1   Introduction

The efficient use of I/O systems is of prime interest for data-intensive appli-
cations, since storage systems are increasing in size and the use case of a sin-
gle system is so diverse, especially in the scientific community. As computing
centers grown in size, and the high-performance computing (HPC) community
approaches exascale [1, 2], it has become increasingly important to understand
how these systems are operating and how they are being used. Additionally,
understanding system behavior helps light the path for future storage system
development and allows the purveyors of these systems to ensure performance is
adequate to allow for work to continue unimpeded. While industry systems are
typically well understood, shared storage systems in the HPC community are
not as well understood. The reason is their sheer size and the concurrent usage
by many users, which typically submit disparate workloads. Most applications
achieve only a fraction of theoretically available performance. Hence, optimiza-
tion and tuning of available knobs for hardware and software are important. This
requires that the user to understand the I/O behavior of the interaction between
application and system, since it determines the runtime of the applications.

However, measuring and assessing observed performance are already non-
trivial tasks and raise various challenges around hardware, software, deployment,
and management. This paper describes the current state of the practice with
respect to such tools.

The paper is structured as follows: First, an introduction to concepts in performance analysis is given in Section 2. Various tools and concepts are described in Section 3. In Section 4 small example studies illustrate how these tools could be used to identify inefficient system behavior. In Section 5 we discuss common issues and sketch potential roads to overcome these issues. In Section 6 we summarize our conclusions.

2 Introduction to Performance Analysis

In computer science, *performance analysis* refers to activity that fosters understanding in timing and resource utilization of applications. Understanding resource utilization includes understanding runtime behavior of the application and the system. For parallel applications, the concurrent computation, communication, and parallel I/O increase the complexity of the analysis.

In industry, the process of system and application tuning is often referred to as *performance engineering*. Software engineers design special methods to embed performance engineering into the application development. With these approaches, performance is considered explicitly during the application design and its implementation. Tools and methodologies such as *Computer-aided software engineering* (CASE) serve the developer in the phases of the software life-cycle, namely, requirement engineering, analysis, design, coding, documentation, and testing. Such tools try to encourage software engineers to incorporate the performance relevant aspects early in the development cycle. Unfortunately, the research and processes in industry are not integrated in state-of-the-art HPC application development, although a few tools do assist in the development of parallel applications. Instead, a closed loop of performance tuning is applied that optimizes programs after a running version exists.

So far, we discussed the situation from the point of view of the user or application. For facility operators or file system administrators, the I/O performance of the whole system or of one parallel file system is the primary concern: Does the system provide the expected or degraded performance, is the system is efficiently used, or do some applications create such a high load that other applications see a significant performance impact?

2.1 Closed Loop of Performance Tuning

The localization of a performance issue on an existing system is a process in which a hypothesis is supported by measurement and theoretic considerations. Measurement is performed by executing the program while monitoring runtime behavior of the application and the system. In general, tuning is not limited to source code; it can be applied to any system. The typical iterative optimization process is the *closed loop of performance tuning* consisting of the phases:

Measurement of performance in an experiment. The environment, consisting of hardware and software including their configuration, is chosen; and the appropriate input (i.e., problem statement) is decided. Since monitoring is limited

to instances of program input, optimizations made for a particular configuration might degrade performance on a different setup or program input. Typically, the measurement itself perturbs the system slightly by degrading performance. Picking the appropriate measurement tools and granularity can reveal the relevant behavior of the system.

Analysis of obtained empirical performance data to **identify issues** and optimization potential in the source code and on the system. Particularly, hot spots—code regions where execution requires significant portions of runtime (or system resources)—are identified. Then, the optimization potential of the hot spots is assessed based on potential performance gains and the estimated time required to modify the current solution.

Changing a few code lines to improve runtime by 5% is more efficient than recoding the whole input/output of a program, especially if I/O might account for only 1% of the total runtime. However, care must be taken when the potential is assessed; depending on the overall runtime, a small improvement might be valuable. From the view of the computing facility, decreasing by 1% the runtime of a program which runs for millions of CPU hours yields a clear benefit by saving operational costs in form of 10,000 CPU hours (about 1.5 CPU years).

Generation of alternatives. Based on the insight gained by the analysis, alternative implementation and tuning options are explored, and system modifications are considered that may mitigate the observed performance issue. This is actually the hardest part of the tuning because it requires that the behavior of the new system can be predicted or estimated. In practice, however, multiple potential options often are evaluated; and, based on the results, the best one is chosen. With increasing experience and knowledge of the person tuning the system, the number of options is reduced as the future behavior can be better anticipated.

Implementation. At the end of a loop the current system is modified; that is, one of the performance relevant layers is adjusted, to realize the potential improvement of the new design. The system is re-evaluated in the next cycle until the time required to change the system outweighs potential improvements or potential gains are too small because the performance measured is already near-optimal. In practice, however, in most cases the efficiency of the current solution is not estimated; instead, the current runtime is considered to be potentially saved.

These phases are then repeated until the desired performance is achieved or until further tuning is not cost effective.

2.2 Measurement

In the closed loop, data is collected that characterizes the application run and system utilization. Many types of data can be collected. For example, the operating system provides a rich set of interesting characteristics such as memory,

network, I/O, and CPU usage. These characterize the activity of the whole system, and sometimes usage can even be assigned to individual applications.

The semantics of this data can be of various kinds. Usually, a *metric* defines the measurement process and the way subsequent values are obtained. For example, *time* is a simple metric that indicates the amount of time spent in a program, function or hardware command. The *throughput* of a storage system, that is, the amount of data transferred per second, is another metric.

One way of managing performance information is to store *statistics*, for example, absolute values such as number of function invocations, utilization of a component, average execution time of a function, or floating-point operations performed. Statistics of the activity of a program are referred to as a **profile**. A profile aggregates events by a given metric, for example by summing up the inclusive duration of function calls. Many tools exist that generate profiles for applications.

In contrast to a profile, a **trace** records events of a program together with a timestamp. Thus, it provides the exact execution chronology and allows analysis of temporal dependencies. External metrics such as hardware performance can be integrated into traces as well. Tracing of behavior produces much more data, potentially degrading performance and distorting attempts of the user to analyze observation. Therefore, in an initial analysis, often only profiles are recorded. A combination of both approaches can be applied to reduce the overhead while still offering enough information for the analysis. Events that happen during a timespan can be recorded periodically as a profile for an interval, thus allowing analysis of temporal variability. By generating profiles for disjoint code regions, behavior of the different program phases can be assessed. Another approach is to enable tracing conditionally, for example by capturing the coarse-grained I/O behavior of the application with statistics and starting tracing when observed performance drops.

The performance data must be correlated with the interesting application's behavior and source code. Depending on the measurement process, assigning information to the cause can be impossible. For example, a statistic cannot reveal the contribution of concurrent activity. Some low-level tools exploit compiler-generated debug symbols to localize the origin of triggered events in the source code.

Several approaches can be used to measure the performance of a given application. A *monitor* is a system that collects data about the program execution. Approaches can be classified based on *where*, *when* and *how* runtime behavior is monitored. A monitor might be capable of recording activities within an application (e.g., function calls), across used libraries, activities within the operating system such as interrupts, or it may track hardware activities; in principle, data can be collected from all components or layers. For I/O analysis, monitors rely on software to measure the state of the system. Data from available hardware sensors is usually queried from the software on demand. Hardware monitors are too expensive, complicated, and inflexible to capture program activity in detail, yet some metrics such as network errors may be provided by the hardware itself.

File system administrators typically measure on the file system server side. Many measurements are independent from the file system type, for example, the load on the servers or the summarized throughput on network adapters or on storage subsystems. Some statistics, however, are only available with special file system types (e.g., Lustre, Spectrum Scale, BeeGFS, NFS).

2.3 Preparation of Applications

Although the Linux operating system offers various statistics in the `/proc` file system, this information is often insufficient for I/O analysis. Usually, changes are made to the program under inspection in order to increase analysis capabilities; the activity that alters a program is called *instrumentation*. Popular methods are to modify source code, to relink object files with patched functions, overriding dynamic library calls at execution time with *LD_PRELOAD*, or to modify machine code directly [3]. During execution, such a modified program invokes functions of the monitoring environment to provide additional information about the program execution and the system state. This instrumentation functionality could also be supported directly by the (operating) system, and hence one could collect performance data without modifying the application.

Since a software monitor requires certain resources to perform its duty (those can be considered as overhead), monitoring an application perturbs the original execution. Observed data must be kept in memory and might be flushed to disk if memory space does not suffice. Additionally, computation is required to update the performance data. The overhead depends on the characteristics of the application and system: it might perturb behavior of the instrumented application so much that an assessment of the original behavior is impossible. In I/O analysis, particularly storing the profile or trace in memory and flushing it to secondary storage incurs considerable overhead that causes additional I/O.

Several techniques can be used to combat potential overhead (and thus application perturbation) in I/O instrumentation. Some tools constrain their instrumentation to summary statistics [4] or compressed representations [5] to minimize overhead. Others may automatically filter activity if an event is fired too often or if the overhead of the measurement system itself grows too high. If filtering still incurs too much overhead, then interesting functions can be manually instrumented, i.e., by inserting calls to the monitoring interface by hand.

Additionally, a selective activation of the monitor can significantly reduce the amount of recorded data. A monitor could sample events at a lower frequency, reducing the overhead and the trace detail level on the same extent.

2.4 Analysis of Data

Users analyze the data recorded by the monitoring system in order to localize optimization potential. Performance data can be recorded during program execution and assessed after the application has finished; this approach of *post-mortem* analysis is also referred to as *offline* analysis. An advantage of this approach is that data can be analyzed multiple times and compared with older

results. Another approach is to gather and assess data *online*, while the program runs. In this approach, feedback is provided immediately to the user, who could adjust settings to the monitoring environment depending on the results.

Because of the vast amount of data, sophisticated tools are required in order to localize performance issues of the system, correlate them with application behavior, and identify the source code causing them. Tools operate either manually (i.e., the user must inspect the data) or automatically. A *semi-automatic tool* could give hints to the user where abnormalities or inefficiencies are found. Tool environments that localize and tune code automatically, without user interaction, are on the wishlist of all programmers. Because of the system and application complexity, however, such tools are only applicable for a very small set of problems. Usually, tools offer analysis capability in several *views* or *displays*, each relevant to a particular type of analysis.

At best a system-wide monitoring of all applications can be conducted to reveal issues; in other words, all applications running on a supercomputer are constantly monitored in a non-intrusive fashion while additional analysis is triggered upon demand. In order to better understand the behavior of a single application, separate analysis runs may be conducted, since it is important to reduce the complexity of scientific software in order to find the cause of the behavior.

The performance analysis is usually done in an ad hoc manner because of the nature of the logs produced by these large scale machines: they are machine specific, and despite having similar attributes, do not share a single format for their trace behavior. This situation extends past HPC storage, and applies more broadly to HPC in general and, additionally, to individual application traces.

Because of this difference in log/trace format, system analysis is usually done per system. Such analysis can produce useful results on the behavior of a particular system, but sheds no light on how it compares to other similar systems, HPC or otherwise [6–9]. Additionally, programmer effort is wasted analyzing each system when the generated analytics are the same or similar for each system. Because of the difference in traces and techniques, these are not typically comparable, leading to most systems being analyzed in a vacuum, never compared to one another.

3 Tools

This section gives an overview of existing tools.

3.1 Darshan

Darshan [10, 11] is an open source I/O characterization tool for post mortem analysis of HPC applications' I/O behavior. Its primary objective is to capture concise but useful information with minimal overhead. Darshan accomplishes this by eschewing end-to-end tracing in favor of compact statistics such as elapsed time, access sizes, access patterns, and file names for each file opened

by an application. These statistics are captured in a bounded amount of memory per process as the application executes. When the application shuts down, it is reduced, compressed, and stored in a unified log file. Utilities included with Darshan can then be used to analyze, visualize, and summarize the Darshan log information. Because of Darshan's low overhead, it is suitable for system-wide deployment on large-scale systems. In this deployment model, Darshan can be used not just to investigate the I/O behavior of individual applications but also to capture a broad view of system workloads for use by facility operators and I/O researchers. Darshan is compatible with a wide range of HPC systems.

Darshan supports several types of instrumentation via software modules. Each module provides its own statistical counters and function wrappers while sharing a common infrastructure for reduction, compression, and storage. The most full-featured modules provide instrumentation for POSIX, MPI-IO and standard I/O library function calls, while additional modules provide limited PNetCDF and HDF5 instrumentation. Other modules collect system information, such as Blue Gene runtime system parameters or Lustre file system striping parameters. The Darshan eXtended Tracing (DXT) module can be enabled at runtime to increase fidelity by recording a complete trace of all MPI-IO and POSIX I/O operations.

Darshan uses *LD_PRELOAD* to intercept I/O calls at runtime in dynamically linked executables and link-time wrappers to intercept I/O calls at compile time in statically linked executables. For example, to override POSIX I/O calls, the GNU C Library is overloaded so that Darshan can intercept all the read, write and metadata operations. In order to measure MPI I/O, the MPI libaries must be similarly overridden. This technique allows an application to be traced without modification and with reasonably low overhead.

3.2 Vampir

Vampir[1] [12] is an open source graphical tool for post mortem performance analysis of parallel systems. It supports off-line analysis of parallel software (MPI, OpenMP, multi-threaded) and hardware-accelerated (CUDA and OpenCL) applications. The analysis engine allows a scalable and efficient processing of large amounts of data. Vampir uses the infrastructure of Score-P[2] for instrumenting applications. Score-P stores events in a file, which can be analysed by Vampir and converted to different views; for example, events can be presented on a time axis or compressed to different statistics. Some views have elaborate filters and zoom functions that can provide an overview but can also show details. Effective usage of Vampir requires a deep understanding of parallel programming. Although the program enables one to capture and analyze sequences of POSIX I/O operations, it gives little or no information about the origin or evaluation of I/O.

---

[1] `http://www.paratools.com/Vampir`
[2] `http://www.vi-hps.org/projects/score-p/`

3.3 Mistral/Breeze

Mistral is commercial command-line tool from Ellexus[3] used to report and resolve I/O performance issues of misbehaving complex Linux applications on HPC clusters. It has real-time monitoring abilities and can change the I/O behavior of applications in order to delay I/O operations and prevent overloading of shared storage. Rules for monitoring and throttling I/O are stored as plain text in configuration files called contracts, which can be modified at runtime by privileged users for systemwide changes and by users application-wide. A sophisticated logging mechanism registers monitoring and throttling events and stores them in files; but with an appropriate plug-in, the logging information can be redirected to any location, for example, in a central database such as Elasticsearch or InfluxDB so that the results can be viewed with Grafana.[4] Administrators can use Mistral to identify applications that are running with bad I/O patterns and harming performance of shared file systems. It can also be used outside of production to run quality assurance tests on complex applications prior to deployment. The key idea behind Mistral is that the information collected is configurable and easily aggregated so that Mistral can be run at scale. Mistral supports POSIX and MPI (MPICH, MVAPICH, OpenMPI) I/O interfaces.

Ellexus Breeze is a user-friendly, self-explained, and well-documented off-line analysis tool with command-line, GUI, and HTML reporting modes. All the information gathered during the application runtime is presented in a comprehensive format and can be of great help to developers. The detailed information about environment can help support teams reproduce and understand problems. The most valuable piece of information is the list of application dependencies so that users and administrators can get a list of every file, library, program, and network location used by an application. Breeze also includes a breakdown of how each program accessed each file so that performance issues such as inefficient metadata access can be found and resolved.

The analysis tool is delivered with a tool called *"trace-program.sh"* that can capture MPI-IO and POSIX function calls and information about the environment and store them in binary trace files. It uses $LD\_PRELOAD$ to wrap original I/O function and, therefore, works only with dynamically linked I/O libraries. Breeze uses either a proprietary binary trace format or plain text output files reported similar to the popular diagnostic tool *strace*. The binary format has some performance advantages and can be decoded to a human readable representation by the included *decode-trace.sh* tool.

Another feature is the ability to compare two different application runs, thereby identifying changes in an application's behaviour and providing valuable feedback to the application's developers.

---

[3] https://www.ellexus.com/products/
[4] https://grafana.com/

3.4 SIOX

SIOX [13] is a highly modular instrumentation, analysis, and profiling framework. It contains an instrumentation tool *"siox-inst"*, a trace reader *"siox-trace-reader"*, and a set of plug-ins and wrappers.

It has wrappers for MPI, POSIX, NetCDF, and HDF5 interfaces that contain reimplementations of the original I/O functions. Inside a reimplemented function is a call to the original function or syscall, and instrumentation code, that generates an activity after each execution. Activities in SIOX are structures that contain various information about the calls. The wrappers can be dynamically linked to an application by using *LD_PRELOAD* and the creation of wrappers during link-time.

Extreme modular design is a key feature of SIOX. The tools *siox-inst* and *siox-trace-reader* can be considered as pure plug-in infrastructures. In other words, there is no functionality inside until some plug-ins and wrappers are loaded. Usage of different sets of plug-ins and wrappers may result in "new" tools that exactly fit the problem. There is no restriction on the number of wrappers and plug-ins that can be loaded simultaneously, so that the functionality of SIOX can be easily extended to perform complex tasks. It has been used to research various aspects such as triggering tracing depending on unusual system behavior, the creation of replayers for recorded behavior, the mutation of I/O calls rerouting targets or replacing system calls, and online analysis. Online analysis can be done by *siox-inst*, by collecting activities from the wrappers and forwarding them to the registered plug-ins. Off-line analysis is based on both tools. Most of the SIOX plug-ins use plug-in interfaces that are supported by *siox-inst* and *siox-trace-reader*, and consequentially these plug-ins can be used by both tools. SIOX has also been coupled with OpenTSDB and Grafana to support online monitoring [14].

By using an instrumented version of FUSE, we found that the complexity of I/O tracers can be reduced and one can monitor `mmap()` of applications. Note that because of the popularity of other tools, SIOX is primarily maintained as a research vehicle.

3.5 PIOM-MP

PIOM-MP (formerly known as PAS2P-IO [15, 16]) represents the MPI application's I/O behavior by using I/O phases. A phase is a consecutive sequence of similar access patterns into the logical view of a file. Because HPC scientific applications show a repetitive behavior, `m` phases will exist in the application. PIOM-MP identifies applications' phases with their access patterns and weights. By using phases of the applications, the analysis focuses on the functional model of the applications.

The I/O phases are used as units for analyzing the performance and scalability of parallel applications. Our approach [17] depicts the global access pattern (spatial and temporal) at the `MPI-IO` and `POSIX-IO` level. Figure1 shows different components of `PIOM-MP`. It comprises three modules: `PIOM-MP` Tracer, `PIOM-MP`
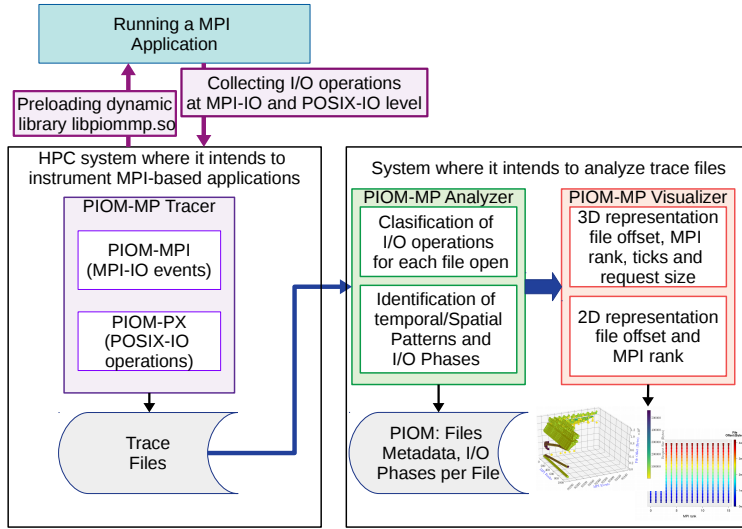
Fig. 1. PIOM-MP modules

Analyzer, and `PIOM-MP` Visualizer. The first module must be in the HPC system in which the parallel application is executed; the other modules can be in a different system.

With PIOM-MP, the user can define the relation between the phase's pattern and the I/O system configuration by using the concept of the I/O requirement in order to explain the I/O performance in a specific system [18]. Furthermore, the user can extrapolate the phases for another number of MPI processes and workloads to evaluate the parallel application I/O scalability.

3.6 Additional User-Level Tools

The Linux kernel monitors various statistics and provides them in the `/proc` file system. These statistics are updated with a certain frequency—typically 1 second—and can be queried by various Linux tools to monitor system and application behavior. Since these counters are incremented, they can be used to derive certain statistics for any interval such as the amount of data accessed during the execution of an application. For example, *collectl*[5] can be used to capture and analyse Lustre throughput or metadata statistics or throughput on an InfiniBand network. Other similar tools include Ganglia and Nagios.

Additional user-level tools can be executed to get a basic understanding of what the application is doing. For example, one can compare the capacity and node quotas before and after job execution to check how much permanent data and how many files have been created. Some tools work only for special file system or network types and need to be executed on client nodes where the application is running.

---

[5] collectl, `http://collectl.sourceforge.net/`

Additional tools of note are the Integrated Performance Monitoring for HPC[6], IOSIG [19], RIOT [20], ScalaIOTrace [5], and Linux blktrace.

3.7 Further Administrative Tools

Most preassembled parallel file system appliances provide a monitoring system with a GUI. It displays various aspects of the servers such as load and I/O activity for current and previous time slots. Typically, it also shows a number of file-system-specific statistics. In many cases, the monitoring systems are only accessible by system administrators, and hence only a few HPC sites expose such information to their users.

However, statistics can be gathered from either each client node or the server nodes. Correlation of a parallel application with the triggered activities on I/O servers is non-trivial since these are shared. With Lustre, *jobstats* [21] provide a way to easily collect I/O statistics of batch jobs with little overhead, for example, the number of open operations or the total amount of written data. To activate *jobstats*, a system administrator selects the environment variable that holds the batch job ID on the client nodes. Lustre clients send the content of this variable with the standard Lustre protocol to the servers, and the servers sum up all I/O activity on this content. An example of how these statistics can be easily made available for users is described in [22]. The Lustre Monitoring Tool (LMT) [23] is also available for collecting server-side time series statistics from Lustre file systems, although data produced with LMT does not distinguish traffic from different applications.

The first tool that combined actual client and server traces of I/O into one view to show a comprehensive timeline of activities was PIOviz [24].

3.8 Tools for Unifying Trace Formats

The diversity of file formats for profiles and tracing is approached by tracing formats such as OTF [25], TAU, and EXTRAE by providing converters between the formats. Since the formats differ slightly, sometimes some information is lost. Besides such generic trace formats, various specialized trace formats exist.

Work is being done at the University of California, Santa Cruz, to produce a tool that provides field per record access to traces with the ability to convert them to another trace schema format on the fly. By giving the ability to translate traces directly to other formats, there is the expectation that analysis will not be done ad hoc for different storage systems. This enables the creation of a standard set of tools for analyzing the behavior of parallel file systems without converting typically large traces.

While the database community has allowed programmable data presentation for some time [26], through views and computed columns, these techniques have not worked their way into trace analysis, despite many system traces being stored in database formats, such as `Parquet`, a Spark SQL queryable columnar

---

[6] IPM, https://github.com/nerscadmin/IPM

format [27]. Presenting data in a uniform format, similar to a trace, is essential but does not require the overhead of a database, since most analysis is done in place, usually in a time series format.

The trace unification tool being developed borrows techniques from NoDB systems as well as from the database community at large. Because of the sequential nature and read-only analysis properties of trace analysis, our aim is a way to enable low-latency translation from one format to another. This approach can be used to unify traces based on the information they share and allow for the information to be analyzed directly, either from a storage system trace analysis tool that we plan to create or from an API that we plan to add in future work.

This system ingests minimal information about each file in order to allow for field-per-record addressing, which is a good interface for analysis tools to access trace data. This mechanism of trace addressing is used to create field and record primitives, which serve as the basis for our schema translation language. By allowing for records and fields to be used as primitives, a simple mechanism for traces to be unified is provided. This is achieved with a simple translation language based on s-expressions that allows for simple translation to occur, with more functionality coming in future work.

Other projects, such as TOKIO [28, 29], have explored the possibility of synthesizing normalized data on demand via modular libraries rather than standardizing on an at-rest trace format. This approach is well-suited to integration of vendor instrumentation tools that utilize proprietary formats or are otherwise difficult to modify. It also enables the integration of multiple data sources simultaneously for holistic analysis and correlation.

4   Example Studies

This section gives examples about how the aforementioned tools can be used to identify performance issues. First, we describe some performance issues that can arise.

System administrators typically try to identify users with the highest I/O activity because of the huge impact that can be realized by helping such users reduce their I/O. Often users are not aware of what they are doing; for example, a user may forget to remove debugging output, and the application will write many times more data than normal. Sometimes this is sufficient to crash the storage completely.

Many sites have different options for storing data: local SSDs on the cluster nodes, burst buffers, different parallel file systems, and so forth. Frequently, application runtime can be improved by selecting an appropriate storage device. For example, with the application OpenFoam [30], one usually can use local disks to store scratch data. Doing so increases the scalability of the application and reduces the load on the central parallel file system, thus helping accelerate other applications. While these optimizations are simple to apply, users commonly store data in the wrong place and hence increase the load on the file system inadvertently.

With Lustre *jobstats* and a simple perl script [22], one can display all jobs with I/O activity above a high-water mark, for example, jobs that have done more than 5 million open operations. Huge amounts of metadata operations should be omitted because each operation requires communication with the responsible server. Lots of small read and write operations have a similar effect and additionally might cause numerous slow seeks on the file system disks. Again, *jobstats* can be used to find jobs doing a huge number of read or write operations with a small average I/O size. For more detailed information about job I/O patterns one would need to combine the file system metrics with application-level tracing from a tool such as Mistral.

Metadata operations are often a source of performance degradation. Common issues include trawling the file system looking for a file, which results in lots of failed stat or open operations. Programs also commonly check the existence of a file before opening it, producing an extra stat for every open. A better approach would be to try to open the file and to fail gracefully if it is not there. Users also can significantly slow metadata servers by opening a file every time it is written to. This usually happens when the open operation has been placed inside a for loop.

In general, parallel file systems show better performance with large sequential read/write operations than with small random operations (huge IOPS rates). Therefore, applications should read or write data in huge chunks. One area of HPC that is known for small I/O operations is the life-sciences industry [31] where applications often use one-byte reads and writes and almost always use 32 kB reads and writes. Although small I/O operations may be justifiable when the algorithms employed in mapping genomes lend themselves to the generation of lots of small files of around 4 kB, often the I/O operations are far smaller, leaving room for improvement.

Another easy improvement with parallel file systems can be achieved by setting the appropriate stripe count. If many files are used and if they are not accessed at the same time by many tasks, a stripe count of 1 is appropriate. On the other hand, if files are shared by many tasks or if only a few tasks use huge files, increasing the stripe count usually improves performance.

With MPI-IO, the MPI library, underlying libraries, and its adaption for the underlying file system can make a huge difference. For example, some vendors provided MPI versions with activated Lustre support providing further optimizations. With the main optimization each collective buffering node collects the I/O for one OST, in other words, small I/O buffers are collected into bigger buffers, and there are no locking conflicts between different nodes which try to access the same area of data. Thus, selecting the right number of collective buffering nodes for a given application and problem size might improve its runtime.

4.1 I/O Performance Analysis at the Application level

The interaction between the I/O system and the application pattern can report poor performance in some HPC systems. In order to identify the root cause of the problem, the I/O pattern along the I/O path needs to be analyzed. Because

the I/O system spans between user (compute nodes) and administrator domains, sometimes it is very difficult to coordinate work to find a possible solution.

On the one hand, administrators have several monitoring tools and commands to control, identify, and solve problems. Usually, these tools are not accessible by the users, however. Problems on this level are related mainly to parallel file systems, the network, and storage devices.

On the other hand, users have several performance analysis tools that, depending on the problem, can be used as profiling or tracing to identify or analyze the I/O. At this level, I/O issues are related mainly to application pattern and I/O libraries. The user needs to understand the impact that a specific access pattern can have on the performance. Achieving such understanding is not simple, however, because a real application can use several files and the user must evaluate each file to weigh the impact of each one, which not always is related directly with file size. An appropriate tool to start with for I/O performance analysis is Darshan, which provides I/O time and throughput for each file opened by a parallel application.

To illustrate this, Figure 2a shows the I/O time and the relation with total time for a strong scaling case that was obtained using Darshan. Two files are selected for analyzing the application scalability: (1) the `mglet_fieldgrid.h5` (red line) is the file that the user considers more important for the application because move a total of 400GiB, which after optimizations shows a scalable behavior; and (2) a small file that corresponds to ghost cells (`gc_stencils.h5` in black line) that is avoiding scaling to the application. Figure 2b shows the offset for the ghost file, whose size is 2.9 G. Concurrent accesses can be observed in Figure 2c for all the MPI processes in lowest and highest offsets, which serialize the I/O as can be seen in Figure 2d. The application presents an I/O imbalance observed for the first 1024 MPI processes. The I/O pattern represented in Figure 2b is similar for the different cases showed in Figure 2a in which there are several rewriting operations that are moving more data into the file system as the number of MPI processes increases. In this case, one must redesign the I/O pattern in order to remove the problem.

4.2 Online Monitoring

DKRZ maintains a monitoring system that gathers various statistics from 3,340 client nodes, 24 login nodes, and Lustre servers. The monitoring system is realized mainly by open source components such as Grafana, OpenTSDB,[7], and Elasticseach but also includes a self-developed data collector. Additionally, the monitoring system obtains various information from the Slurm workload manager. A schematic overview is provided in Figure 3.

The data is aggregated and visualized by a Grafana web interface, which is available to all DKRZ users. The information is structured in three sections: login nodes, user jobs, and queue statistics.

---

[7] Consists of various additional components from the Hadoop Stack

(a) I/O Problem

(b) `gc_stencils.h5` file offset

(c) The `stencils` file temporal pattern
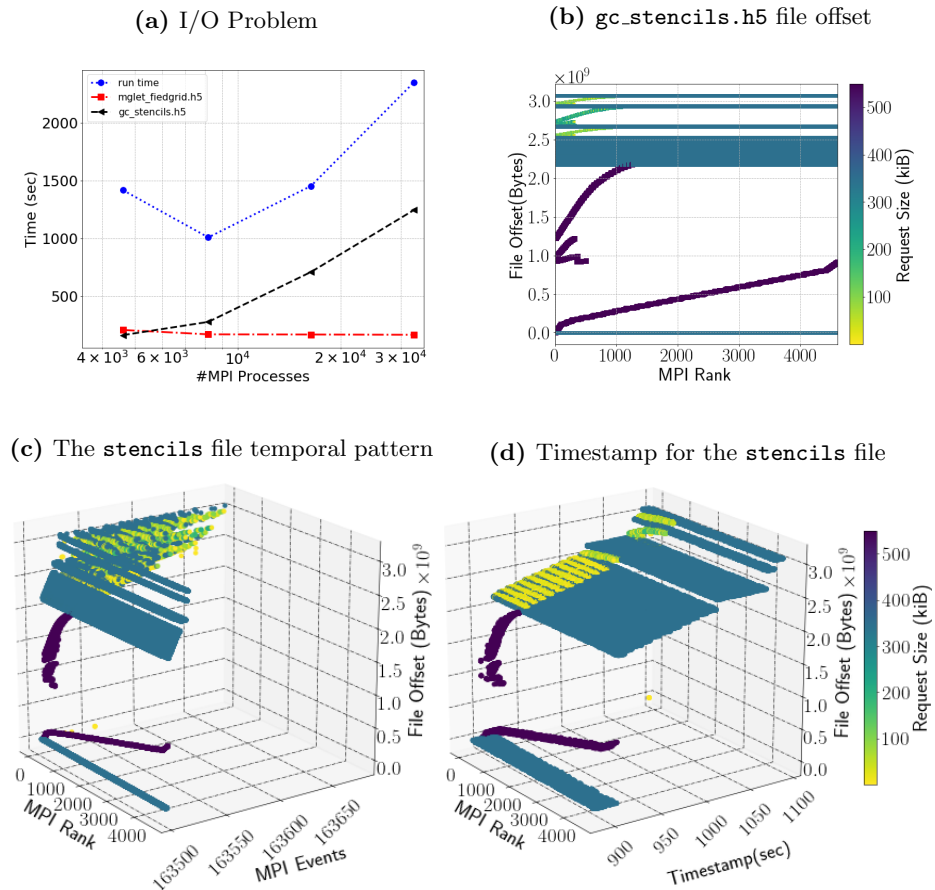
(d) Timestamp for the `stencils` file

Fig. 2. Example of I/O performance analysis. (a) Identifying the file with more impact on run time using Darshan. (b) Analyzing the I/O pattern with a focus on the `gc_stencils.h5` file offset for 4608 MPI processes using PIOM-MP. (c) The `gc_stencils.h5` file temporal pattern based on the application logical using MPI events as ticks. (d) The `gc_stencils.h5` file temporal pattern using system timestamps.

In the first place, a monitoring service gives the users an overview of the current state of the system, namely, the current load of login nodes and number of used nodes on Slurm partitions. For each single machine, a detail view also provides information about system load, memory consumption, and Lustre statistics, as well as historical data. Job monitoring is enabled by default in a coarse-grained mode but can be modified by a Slurm parameter. When enabled, the monitoring system gathers information about CPU usage and frequency, memory consumption, Lustre throughput, and network traffic for each client node. Statistics about the usage of *Mistral* nodes (Figure 4) show the current
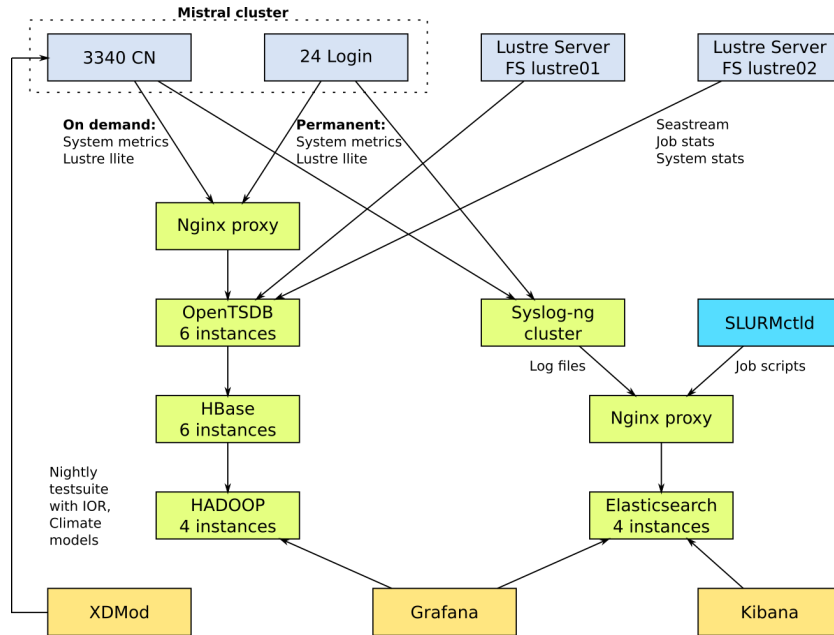
Fig. 3. DKRZ-monitoring

Fig. 4. Statistics about the usage of Mistral nodes

state and history of node allocation, queue status, and waiting time of both Slurm partitions. Additionally, DRKZ runs xdmod on the client nodes for viewing historical job information as well as real-time scientific application profiling.

4.3 Online Monitoring with LLview

In 2004 the Jülich Supercomputing Centre (JSC) began to develop LLview [32], an interactive graphical tool to monitor job scheduling for various resource managers such as SLURM or LoadLeveler. In its stand-alone and web-based clients LLview provides information about currently running jobs and how they are mapped on the individual nodes of an HPC cluster, using an intuitive visual representation of the underlying cluster configuration. In addition, it includes statistical information on completed jobs as well as a prediction for scheduled future jobs [33]. LLview is client-server based. Internally it uses an XML schema named LML [34] as an extensible abstraction layer to encode monitor data. The data itself is obtained from various server and application interfaces and subsequently transferred to the clients of LLview. LML is a simple, extensible, and independent description of node- and job-related information. A subset of the LLview dashboard elements are also integrated in Parallel Tools Platform, an extension to Eclipse to develop, run, and debug parallel programs on a remote HPC system [35].

Recently, a number of extensions to LLview have been designed and implemented that allow users to acquire, store, and display job performance metrics such as node CPU load, memory usage, interconnect activity, and I/O activity. In practice at JSC these have greatly facilitated the analysis of HPC codes that either show unsatisfactory performance or behave incorrectly. With regard to metrics about I/O, the subject of this paper, LLview uses performance data generated with the GPFS mmpmon command [36] within the framework of the IBM Spectrum Scale (GPFS) file system. At JSC, all GPFS clients running on cluster compute nodes are configured to frequently write (every 1–2 minutes) such mmpmon data to a log file that itself resides in a local GPFS files system. The LLview server components collect the I/O performance data (e.g., number of bytes written/read, number of open/close calls) from these files and match it to user jobs that are running on these nodes. This mapping is possible because node sharing is not applied on JSC systems. The job-based data is then stored in an internal database, from which individual performance reports are generated.

Since the minimum update frequency of LLview is not more than once a minute, JSC has integrated this to help users monitor job behavior on a coarse level and to help detect performance issues while jobs are running on the system. Within a web portal (*LLview jobreports*) access is provided to timeline-based performance data of current and recently finished jobs on the system, ranging two weeks into the past. In addition to interactive charts displaying the various performance data, the portal provides PDF reports for all jobs, which can be collected and archived by users to document their production runs. These extensions of LLview have proven useful as a first step of performance analysis and code tuning, thanks to the integrated coarse view on the most essential performance metrics of user jobs. In this way it greatly facilitates the detection of situations where deeper performance analysis and tuning should be started with additional tools such as Darshan or SIOX (Section 3).

5 Challenges in Analyzing I/O

The following list represents issues that we see as a community that must be overcome to ensure effective and efficient monitoring in future HPC systems.

- **Ability to monitor I/O at all levels**: The involvement of many layers in the I/O stack that are not instrumented limits the ability to identify issues. Today, parallel file systems typically have disks, storage controllers, storage servers, and clients. All are connected with different network types. Hardware and firmware or software can cause performance degradation at each layer. Above the parallel file systems are MPI-IO, MPI libraries, and upper-level libraries such as HDF5. In many cases administrative power and deep knowledge are required in order to capture data from certain layers. In an ideal case, one should be able to easily investigate the I/O at each level.
- **Adaption of tools to new programming models**: The HPC application community has long been (and still is) dominated by MPI simulations written in Fortran or C. However, other paradigms such as deep learning and big data use their own programming models, languages, and runtime environments. Parallel I/O analysis tools must adapt to keep pace as these models are more widely adopted in HPC.
- **Adaption to new hardware technologies**: Analyzing parallel I/O at the moment basically means analyzing I/O access to a parallel file system, but a number of emerging technologies call for a broader interpretation of I/O. Future systems may include nonvolatile memory (which may be accessed as a memory device or a file system) and multiple tiers of storage that hold the simulation working set, campaign data, or archival data. The performance of each subsystem is best interpreted in the context of the complete storage hierarchy. These new technologies must provide means to systematically report performance data for the analysis.
- **Integrating analysis data**: Data analysis of performance data is often an afterthought, done after the application was executed (post mortem). Projects such as TOKIO [8] are working toward integrating data from multiple components of the storage system, but this remains a pressing problem. Each storage system component is typically designed by a specialized vendor and is serviced by its own instrumentation framework. Aligning and synthesizing data from job schedulers, applications, file systems, storage devices, and network devices remain a significant challenge.
- **Feature replication and sustainability**: Many of the technologies included in this survey serve overlapping purposes and implement their own variation of tasks such as function interception, time series indexing, and visualization. Increased standardization and code sharing could potentially reduce the engineering effort involved in each project while at the same time increasing portability.

---

[8] http://www.nersc.gov/research-and-development/storage-and-i-o-technologies/tokio/

- **Performance assessment**: Monitoring and recording performance data are part of the process. However, it is often hard for experts to assess whether the observed behavior and performance is acceptable or not. End users typically do not understand what a runtime of X seconds or an achieved performance of Y GiB/s means. Guidance for assessing the quality is a necessary step.
- **Correlation of I/O monitoring with other metrics**: In parallel applications it is often essential to observe more than only one performance metric such as I/O activity in an isolated manner. Rather, bringing I/O activity in correlation with other metrics such as CPU activity, memory usage, and interconnect usage is needed in order to identify root causes of faulty behavior and to distinguish between causes and effects.
- **User guidance**: A gap remains between what can be learned from expert I/O analysis and what can be readily adapted by end users. The data produced by today's I/O analysis tools requires significant expertise to interpret, and little guidance is provided as to what I/O performance a user should expect in a given application scenario.
- **Small developer community**: I/O tools for HPC suffer from the fact that the HPC community is small and I/O is often neglected. Massive amounts of traces are being generated from user applications and other sources outside the HPC community but not covered by tools. We have to make the base of tools (at least) useful outside the HPC community, for example, to CERN [37,38], in order to speed the development.

We believe that the **standardization of monitoring APIs and infrastructure** is a necessary step facing the overall community. At the moment no standard API exists to ease the reporting of data; instead, vendors and tool developers build their own solution. A standard would be beneficial to several issues mentioned in this list, and might help encourage hardware and software developers to provide the necessary data. Indeed, we must standardize our approaches to system analysis, both to allow comparison between like systems and to avoid the programming overhead of repeatedly creating the same analytics code again and again. By creating a standardized method of analyzing storage systems, as well as other system traces, we hope to light the path ahead to exascale and help computing centers deal with massive growth and data-driven analysis.

Additionally, by creating a general-purpose method of trace analysis, we can provide optimizations and features that would be too costly to develop for individual tools or systems.

6 Conclusions

In this paper, we have provided an overview of the state-of-the-art in I/O monitoring tools, motivated by our own involvement in the development of the described tools. We then gave examples of how monitoring can be used to reveal I/O issues; because of space limitations, this list is intended as an appetizer

for the reader to study the tools and mentioned papers further. Furthermore, we discussed challenges we see as a community that are currently not well addressed and must be resolved further in future research, development, and engineering. One step forward for the community is to work on standardization.

Acknowledgments

References

1. Benson, T., Anand, A., Akella, A., Zhang, M.: Understanding data center traffic characteristics. SIGCOMM Comput. Commun. Rev. **40**(1) (January 2010) 92–99
2. Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hill, K., Hiller, J., et al.: Exascale computing study: Technology challenges in achieving exascale systems. Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep **15** (2008)
3. Shende, S., Malony, A.D., Ansell-bell, R.: Instrumentation and measurement strategies for flexible and portable empirical performance evaluation. In: International Conference on Parallel and Distributed Processing Techniques and Applications. PDPTA (2001) 1150–1156
4. Carns, P., Latham, R., Ross, R., Iskra, K., Lang, S., Riley, K.: 24/7 characterization of petascale I/O workloads. In: Proceedings of 2009 Workshop on Interfaces and Architectures for Scientific Data Storage, IEEE (2009)
5. Vijayakumar, K., Mueller, F., Ma, X., Roth, P.C.: Scalable I/O tracing and analysis. In: Proceedings of the 4th Annual Workshop on Petascale Data Storage, ACM (2009) 26–31
6. Adams, I., Madden, B., Frank, J., Storer, M.W., Miller, E.L.: Usage behavior of a large-scale scientific archive. In: Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC12). (2012)
7. Adams, I.F., Storer, M.W., Miller, E.L.: Analysis of workload behavior in scientific and historical long-term data repositories. ACM Transactions on Storage **8**(2) (2012)
8. Wang, F., Xin, Q., Hong, B., Brandt, S.A., Miller, E.L., Long, D.D.E., McLarty, T.T.: File system workload analysis for large scale scientific computing applications. In: Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies, College Park, MD (April 2004) 139–152
9. Grawinkel, M., Nagel, L., Mäsker, M., Padua, F., Brinkmann, A., Sorth, L.: Analysis of the ECMWF storage landscape. In: 13th USENIX Conference on File and Storage Technologies (FAST 15), Santa Clara, CA, USENIX Association (2015) 15–27

10. Carns, P., Harms, K., Allcock, W., Bacon, C., Lang, S., Latham, R., Ross, R.: Understanding and improving computational science storage access through continuous characterization. ACM Transactions on Storage (TOS) **7**(3) (2011) 8

11. Carns, P.: Darshan. In: High performance parallel I/O. Computational Science Series, Chapman & Hall/CRC (2015) 309–315

12. Knüpfer, A., Rössel, C., Mey, D.a., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., Nagel, W.E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S., Tschüter, R., Wagner, M., Wesarg, B., Wolf, F. In: Score-P: A joint performance measurement run-time infrastructure for eriscope, Scalasca, TAU, and Vampir. Springer Berlin Heidelberg, Berlin, Heidelberg (2012) 79–91

13. Kunkel, J., Zimmer, M., Hübbe, N., Aguilera, A., Mickler, H., Xuan Wang, A.C., Thomas Bönisch, J.L., Michel, R., Weging, J.: The siox architecture -– coupling automatic monitoring and optimization of parallel i/o. (2014)

14. Betke, E., Kunkel, J.: Real-time I/O-monitoring of HPC applications with SIOX, Elasticsearch, Grafana and FUSE. In: High Performance Computing: ISC High Performance 2017 International Workshops, DRBSD, ExaComm, HCPM, HPC-IODC, IWOPH, IXPUG, P3MA, VHPC, Visualization at Scale, WOPSSS. Number 10524 in Lecture Notes in Computer Science, Springer (2017) 158–170

15. Mendez, S., Rexachs, D., Luque, E.: Modeling parallel scientific applications through their input/output phases. In: Cluster Computing Workshops (CLUSTER WORKSHOPS), 2012 IEEE International Conference on. (Sept 2012) 7–15

16. Mendez, S., Panadero, J., Wong, A., Rexachs, D., Luque, E.: A New approach for analyzing I/O in parallel scientific applications. In: CACIC 12, Congreso Argentino de Ciencias de la Computación. (2012) 337–346

17. Gomez-Sanchez, P., Mendez, S., Rexachs, D., Luque, E.: PIOM-PX: A framework for modeling the I/O behavior of parallel scientific applications. In: High Performance Computing - ISC High Performance 2017 International Workshops, DRBSD, ExaComm, HCPM, HPC-IODC, IWOPH, IXPUG, P^3MA, VHPC, Visualization at Scale, WOPSSS, Frankfurt, Germany, June 18-22, 2017, Revised Selected Papers. (2017) 160–173

18. Mendez, S., Rexachs, D., Luque, E.: Analyzing the parallel I/O severity of MPI applications. In: Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, Madrid, Spain, May 14-17, 2017. (2017) 953–962

19. Yin, Y., Byna, S., Song, H., Sun, X.H., Thakur, R.: Boosting application-specific parallel I/O optimization using IOSIG. In: Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012), IEEE Computer Society (2012) 196–203

20. Wright, S.A., Hammond, S.D., Pennycook, S.J., Bird, R.F., Herdman, J., Miller, I., Vadgama, A., Bhalerao, A., Jarvis, S.A.: Parallel file system analysis through application I/O tracing. The Computer Journal **56**(2) (2012) 141–155

21. Intel(2011-2017), Oracle(2010-2011): Lustre Software Release 2.x, Operations Manual. Chapter 12.2. `http://doc.lustre.org/lustre_manual.pdf`

22. Lustre-Community: Lustre Monitoring and Statistics Guide. Chapters 6.2.7 and 6.2.8. `http://wiki.lustre.org/Lustre_Monitoring_and_Statistics_Guide`

23. Uselton, A.: Deploying Server-side File System Monitoring at NERSC. In: Proceedings of the 2009 Cray User Group. (2009)

24. Ludwig, T., Krempel, S., Kuhn, M., Kunkel, J., Lohse, C.: Analysis of the MPI-IO optimization levels with the PIOViz Jumpshot enhancement. In: European Parallel

Virtual Machine/Message Passing Interface Users' Group Meeting, Springer (2007) 213–222

25. Eschweiler, D., Wagner, M., Geimer, M., Knüpfer, A., Nagel, W.E., Wolf, F.: Open Trace Format 2: The next generation of scalable trace formats and support libraries. In: PARCO. Volume 22. (2011) 481–490

26. Smith, I.: Guide to using sql: Computed and automatic columns. Rdb Jornal (2008)

27. Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., Zaharia, M.: Spark SQL: Relational data processing in Spark. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. SIGMOD '15, New York, NY, USA, ACM (2015) 1383–1394

28. Lockwood, G.K., Yoo, W., Byna, S., Wright, N.J., Snyder, S., Harms, K., Nault, Z., Carns, P.: UMAMI: a recipe for generating meaningful metrics through holistic I/O performance analysis. In: Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems, ACM (2017) 55–60

29. Lockwood, G.K., Snyder, S., Brown, G., Harms, K., Carns, P., Wright, N.J.: TOKIO on ClusterStor: Connecting standard tools to enable holistic I/O performance analysis. In: Proceedings of the 2018 Cray User Group. (2018)

30. Jasak, H., Jemcov, A., Tukovic, Z., et al.: OpenFOAM: A C++ library for complex physics simulations. In: International workshop on coupled methods in numerical dynamics. Volume 1000., IUC Dubrovnik, Croatia (2007) 1–20

31. Ellexus, Alces-Flight: Maximising hpc performance on AWS public cloud. `https://www.ellexus.com/wp-content/uploads/2017/07/white_paper_alces_ellexus.pdf`

32. Frings, W., Karbach, C.: LLview: Graphical monitoring of batch system controlled cluster. `http://www.fz-juelich.de/jsc/llview/` (2004-2018)

33. Karbach, C.: A highly configurable and efficient simulator for job schedulers on supercomputers. PARS-Mitteilungen **30**(1) (Apr 2013) 25–36

34. Karbach, C.: LML: Large-scale system Markup Language. `http://llview.fz-juelich.de/LML/OnlineDocumentation/lmldoc.html` (2013)

35. Watson, G.R., Frings, W., Knobloch, C., Karbach, C., Rossi, A.L.: Scalable control and monitoring of supercomputer applications using an integrated tool framework. In: 2011 40th International Conference on Parallel Processing Workshops. (Sept 2011) 457–466

36. IBM: Monitoring GPFS I/O performance with the mmpmon command. `https://www.ibm.com/support/knowledgecenter/STXKQY_5.0.0/com.ibm.spectrum.scale.v5r00.doc/bl1adv_mmpmonch.htm` (2018)

37. Peters, A., Sindrilaru, E., Adde, G.: Eos as the present and future solution for data storage at cern. Journal of Physics: Conference Series **664**(4) (2015)

38. Peters, A.J., Janyst, L.: Exabyte scale storage at cern. Journal of Physics: Conference Series **331**(5) (2011)