Semi-automatic Assessment of I/O Behavior by Inspecting the Individual Client-Node Timelines—An Explorative Study on 10^6 Jobs

Eugen Betke[1(✉)] and Julian Kunkel[2]

[1] DKRZ, Hamburg, Germany
betke@dkrz.de
[2] University of Reading, Reading, UK
j.m.kunkel@reading.ac.uk

Abstract. HPC applications with suboptimal I/O behavior interfere with well-behaving applications and lead to increased application runtime. In some cases, this may even lead to unresponsive systems and unfinished jobs. HPC monitoring systems can aid users and support staff to identify problematic behavior and support optimization of problematic applications. The key issue is how to identify relevant applications? A profile of an application doesn't allow identifying problematic phases during the execution but tracing of each individual I/O is too invasive.

In this work, we split the execution into segments, i.e., windows of fixed size and analyze profiles of them. We develop three I/O metrics to identify three relevant classes of inefficient I/O behaviors, and evaluate them on raw data of 1,000,000 jobs on the supercomputer Mistral. The advantages of our method is that temporal information about I/O activities during job runtime is preserved to some extent and can be used to identify phases of inefficient I/O.

The main contribution of this work is the segmentation of time series and computation of metrics (Job-I/O-Utilization, Job-I/O-Problem-Time, and Job-I/O-Balance) that are effective to identify problematic I/O phases and jobs.

1 Introduction

Modern HPC systems are processing many thousands of jobs every day. Some of them can misbehave for some reasons (e.g., due to poor programming practices, I/O intensive tasks, or bugs) and can slow down the whole system performance and affect other jobs that are running on the same system in parallel. This bad behavior must be identified and brought under control. Before we can think about what to do with these jobs, we need to find a way to detect them.

It is important to detect inefficient I/O patterns. Monitoring systems are employed to solve this problem. However, the amount of time needed by humans to identify inefficient usage grows with the system size and the runtime of jobs.

© Springer Nature Switzerland AG 2020
P. Sadayappan et al. (Eds.): ISC High Performance 2020, LNCS 12151, pp. 166–184, 2020.
https://doi.org/10.1007/978-3-030-50743-5_9

To overcome this, the monitoring system must provide tools aiding the analysis. It needs to produce more compact representation of data providing meaningful metrics and allow for deeper analysis.

There are a variety of data-intensive parallel applications that run on HPC-systems solving different tasks, for example, climate applications. Depending on the application, we can observe different data and metadata characteristics such as parallel/serial I/O, check-pointing behavior, or I/O bursts in write/read phases. Efficient patterns are critical for I/O performance of file systems and application runtime. Checking every application manually is not possible for the support. We believe that focusing on relevant jobs is important, hence we need meaningful metrics tailored to parallel jobs and are sensitive to specific I/O behaviors.

After the related work section, the theoretical part follows. Then, we evaluate the approach on a real HPC system.

2 Related Work

There are many tracing and profiling tools that are able to record I/O information [6]; we will discuss a selection of them in more detail in the following. The issue of performance profiles is that they remove the temporal dimension and make it difficult to identify relevant I/O phases. As the purpose of interesting applications is the computation and I/O is just a byproduct, applications often spend less than 10% time with I/O. Tracing tools, however, produce too much information that must be reduced further.

The Ellexus tools[1] include `Breeze`, a user-friendly offline I/O profiling software, an automatic I/O report generator `Healthcheck`, and command line tool `Mistral`[2] which purpose is to report on and resolve I/O performance issues when running complex Linux applications on high performance compute clusters. Mistral is a small program that allows you to monitor application I/O patterns in real time, and log undesirable behaviour using rules defined in a configuration file called a contract. Ellexus tools support POSIX and MPI (MPICH, MVAPICH, OpenMPI) I/O interfaces.

Darshan [2,3] 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

[1] https://www.ellexus.com/products/.

[2] Not to confuse with the DKRZ supercomputer Mistral!.

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-I/O 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-I/O 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 libraries must be similarly overridden. This technique allows an application to be traced without modification and with reasonably low overhead.

LASSi tool [7] was developed for detecting, the so called, victim and aggressor applications. An aggressor can steal I/O resources from the victim and negatively affect its runtime. To identify such applications, LASSi calculates metrics from Lustre job-stats and information from the job scheduler. One metric category shows file system load and another category describes applications I/O behavior. The correlation of these metrics can help to identify applications that cause the file system to slow down. In the LASSi workflow this is a manual step, where a support team is involved in the identification of applications during file system slow down. Manual steps are disadvantageous when processing large amounts of data and must be avoided in unsupervised I/O behavior identification. LASSi's indicates that the main target group are system maintainers. Understanding LASSi reports may be challenging for ordinary HPC users, who do not have knowledge about the underlying storage system.

The Ellexus tool set includes, `Breeze`, an offline I/O profiling software, an automatic I/O report generator `Healthcheck`, and command line tool `Mistral`, which purpose is to report on and resolve I/O performance issues when running complex Linux applications on high performance compute clusters. Mistral is a small download that allows you to monitor application I/O patterns in real time, and log undesirable behaviour using rules defined in a configuration file called a contract. Another powerful feature of Mistral is the ability to control I/O for application individually. Ellexus tools currently support POSIX and MPI (MPICH, MVAPICH, OpenMPI) I/O interfaces.

Another branch of research goes towards I/O prediction. Some methods work with performance data from storage systems, application side and hybrids. Application runtime prediction, efficient scheduling, I/O performance improvement. The methods work in a dynamically changing environment. They didn't tell much about the application.

The discussed limitations are well known, and many projects investigate new solutions for I/O assessment of behaviour.

In [5], the authors utilized probes to detect file system slow-down. A probing tool measures file system response times by periodically sending metadata and read/write requests. An increase of response times correlates to the overloading of the file system. This approach allows the calculation of a slow-down factor identification of the slow-down time period.

In [4], the authors run HPC applications in monitored containers. Depending on metric values captured during application runtime, the I/O management can increase or decrease the number of containers, or even take them offline, if insufficient resources are available.

In [8], a performance prediction model is developed by developers that aims to improve job runtime estimation for better job scheduling. The authors use the property of static iterative scientific code to produce near constant I/O burst, when considered over a longer period of time.

3 Methodology

The methodology of this work relies on (1) the segmentation of I/O traces for jobs, i.e., the generation of performance profiles for fixed length time windows. This operation results in a set of segments over job runtime that (2) are analyzed individually and aggregated on node level or job level. (3) Finally, the development of metrics for scoring the segments, i.e., the mapping from segment data to meaningful scores. The thresholds for those metrics can be semi-automatically determined and learned. In this section, we introduce the methodology in a generic manner, without giving any numbers or using metrics. We apply and evaluate the approach on a real HPC system in Sect. 5.

3.1 Segmentation and Timeline Aggregation

Let us assume the following as a starting situation. A data collector runs on all compute nodes, captures periodically metrics, and sends them to a centralized database. Database stores each metric as a time series together with information like node name, file system, job ID.

As the resolution of the sampling is too fine-grained (the default sampling interval is 5 s), we split the timeline obtained on a client node into segments of equal length.

To illustrate the approach, consider the fictive example: a job runs on 4 nodes and a monitoring system collects data for 4 different metrics at time points t_X, with $0 \leq X < 9$. By grouping 3 samples of each metric into one segment, we obtain 3 segments.

Node and job segments are collections of metric segments that aggregate this information for each node or for each job. The example is illustrated in Fig. 1. A segment can be related to an individual metric (green), a node (red), or the job data (blue).

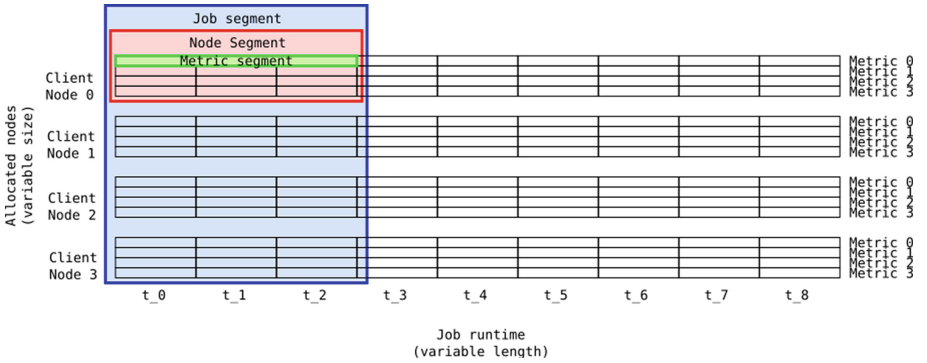


Fig. 1. Monitoring data structure and segmentation. In the example, 4 metrics are captured on 4 client nodes at time points t_i. Three sequential samples are aggregated to metric segments (green box). Node and job segments are collections of metric segments (red and blue boxes). (Color figure online)

3.2 Training

The training step produces statistics, which describe the overall I/O performance of the HPC system. Ideally, the analyzed dataset should contain peak performance values, achievable on an HPC, for all metrics. Similar performance values form categories (e.g., low, medium, and high performance).

There are several alternative ways to form categories: by manual selection, by using statistics like quantiles, and by using machine learning algorithms. We tried all the three mentioned methods, but quantiles worked robustly for our purpose. Furthermore, it allows to determine the percentage of jobs that the support team can investigate. For example, for the one million jobs investigated in this study (covering a period of 3 month), DKRZ could inspect 1000 - 10k jobs closer, hence looking at the 0.1% of jobs that are most I/O demanding.

We want to take a closer look at the computation of quantiles. Tabe 1 illustrates the idea. First of all, we define two quantiles `qX` and `qY`, and use them to determine the limits for each metric individually (in our case X = 99 and Y = 99.9). For simplification, we use the same quantiles for all metrics. After definition of the limit, the metric segments can be categorized and we count the number of segments that falls into each category in the following way:

LowIO	smaller than `qX`	$c_{0,X} = count(value(\text{metric}_X) \leq \text{limit}_{0,X})$
HighIO	between `qX` and `qY`	$c_{1,X} = count(\text{limit}_{0,X} > value(\text{metric}_X) \leq \text{limit}_{1,X})$
CriticalIO	larger than `qY`	$c_{2,X} = count(value(\text{metric}_X) > \text{limit}_{1,X})$

Table 1. Generic limits and category statistics.

Metric	Limits		Number of occurrences		
Name	qY	qX	LowIO	HighIO	CriticalIO
$metric_0$	$limit_{0,0}$	$limit_{1,0}$	$c_{0,0}$	$c_{1,0}$	$c_{2,0}$
$metric_1$	$limit_{0,1}$	$limit_{1,1}$	$c_{0,1}$	$c_{1,1}$	$c_{2,1}$
...					
$metric_N$	$limit_{0,N}$	$limit_{1,N}$	$c_{0,N}$	$c_{1,N}$	$c_{2,N}$

3.3 Scores

Our categories are labeled manually. The scoring strategy is based on the following considerations:

Since, `LowIO` represents low I/O utilization, it gets a score of 0. This category will be mostly ignored in derived metrics. `HighIO` contains no outliers but may generate a mixed workload or be inefficient and needs to be taken into account. Therefore, it gets a score of 1. `CriticalIO` is a weight factor, larger than HighIO. We suggest to compute CriticalIO/HighIO, and to take the smallest value for Z (this is summarized in Table 2a).

Table 2. Summary of the scoring

Category name	MScore
`LowIO`	0
`HighIO`	1
`CriticalIO`	Z

(a) *Category scores*

Score name	Definition
`MScore`	= category scores
`NScore`	$\sum$ `MScore`
`JScore`	$\sum$ `NScore`

(b) *Segment scores*

Based on the individual metrics scores, further scores are derived. The node score is the sum of all individual metrics scores for a segment, i.e., it indicates if there is an I/O issue at all in this segment and on this node. The job level aggregation is the sum of the node score (see Table 2b).

3.4 Job Assessment

Once the system is trained and a configuration file with the statistics generated, a single job can be analyzed and assessed automatically. To understand the behavior of the job I/O, we exploit the knowledge about the timeline and analyze the temporal and spatial I/O behavior of the segments in coarse-grained fashion. This is achieved by introducing new metrics that reduce the complexity into relevant scores that show potential for optimization: the Job-I/O-Problem-Time, Job-I/O-Utilization, and Job-I/O-Balance. These values must be considered together.

Job-I/O-Problem-Time. This metric is the fraction of job runtime that is I/O-intensive; it is approximated by the fractions of segments that are considered problematic (JScore > 1). I/O problem time is the amount of problematic, I/O-intensive job segments (IOJS) divided by the total number of job segments (JS) (see Eq. (1)).

$$\text{Job-I/O-Problem-Time} = \frac{\text{count (IOJS)}}{\text{count (JS)}} \tag{1}$$

Job-I/O-Utilization. While most phases may not do any I/O, these might have extraordinary I/O activity during such phases. Large jobs with a large number of I/O accesses can induce slow down on the file system for other jobs. To identify such jobs, we compute a metric that shows the average load during I/O-relevant phases.

The first step identifies I/O-intensive job segments (IOJS), i.e., JScore > 1, and counts occurrences $N = count(IOFS)$. Assume, the `max_score() function returns the highest metric score of all metrics in a job segment.` Then, the quotient of the `max_score()`'s sum and N is I/O utilization for one particular file system. For handling several file systems, we compute a sum of the resulting values and obtain Job-I/O-Utilization (see Eq. (2)).

$$\text{Job-I/O-Utilization} = \sum_{FS} \frac{\sum_{j \in IOJS} \text{max\_score}(j)}{N} \tag{2}$$

Since, Job-I/O-Utilization considers only I/O intensive job segments, the condition `max_score()` ≥ 1 is always true. Thus, Job-I/O-Utilization is defined for a job iff the job has at least some relevant I/O activity. Job-I/O-Utilization values are always ≥ 1.

For a conventional mean-score computation, we would probably apply the `mean_score()` function to a job segment, instead of `max_score()`, to obtain a mean value of all metric scores in a job segment. This would provide a conventional mean value, as we would expect it. Although such a value might be more intuitive, the following considerations show that it is not robust enough for our purpose. Monitoring data (in particular historical data) may be incomplete or incompatible, e.g., when some metrics are not captured due a collector malfunction or when monitoring system changes after. As a consequence, conventional mean values for complete and incomplete job data may diverge quite substantially from one another, even for jobs with similar I/O performance. For illustration, consider a job segment with only one active metric segment, e.g., with score = 4, and others with scores = 0. The mean value would be smaller, if data for all 13 metrics are available as if only 8 metrics are present. This would adversely affect the result, assigning higher values to incomplete data. In this context of this work, this would be interpreted as higher I/O load. To prevent such a miss-calculation, we compute mean value of job segment max values. This method is independent of the number of metrics and fulfills our requirements. Even if one metric segment works with high performance, the whole job seg-

ment can be considered as loaded. This works as a perfect complement for the balance metrics.

Job-I/O-Balance. The balance metric indicates how I/O load is distributed between nodes during job runtime. Here again, we consider only I/O-intensive job segments ($IOJS$), i.e., JScore > 1 but divide them with the maximum score obtained on any single node. A perfect balance is 1.0 and a balance where 25% of nodes participate in I/O is 0.25.

For each job segment j, with $j \in IOJS$, we compute:

1. NScore for each node segment
2. Mean and max values of NScores
3. Job-I/O-Balance(j) for a job segment, i.e., the quotient of mean and max values

The overall Job-I/O-Balance is the mean value of all Job-I/O-Balance(j) values, with $j \in IOJS$ (see Eq. (3)).

$$\text{Job-I/O-Balance} = \text{mean}\left(\left\{\frac{\text{mean\_score (j)}}{\text{max\_score (j)}}\right\}_{j \in \text{IOJS}}\right) \tag{3}$$

3.5 Example

Assume a 4-node job with two I/O intensive job segments s_{j_0} and s_{j_5}. Furthermore, assume, the job assesses two file systems fs_1 and fs_2. We compute Job-I/O-Utilization, Job-I/O-Problem-Time and Job-I/O-Balance metrics in Eqs. (4) to (6) for generic data illustrated in Fig. 2.

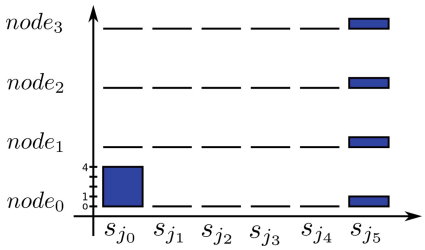


Fig. 2. Segment timeline. $s_{j_0}, s_{j_5} \in IOJS$ are I/O-intensive job segments.

$$\begin{aligned}
\text{max}_0 &= \text{max_score}(s_{j_0}) &&= 4 \\
\text{max}_1 &= \text{max_score}(s_{j_5}) &&= 1 \\
\text{U}_{fs1} &= \text{mean}(\{\text{max}_0, \text{max}_1\}) &&= 2.5 \\
\text{U}_{fs2} &= \text{mean}(\{\text{max}_0, \text{max}_1\}) &&= 2.5 \\
\text{Job-I/O-Utilization} &= \text{U}_{fs1} + \text{U}_{fs2} &&= 5
\end{aligned} \tag{4}$$

$$\begin{aligned} \mathrm{N_{IOJS}} &= 2 \\ \mathrm{N_{JS}} &= 6 \\ \text{Job-I/O-Problem-Time} &= \frac{\mathrm{N_{IOJS}}}{\mathrm{N_{JS}}} \qquad \approx 0.33 \end{aligned} \tag{5}$$

$$\begin{aligned} b_0 &= \text{balance}(s_{j_0}) \qquad = 0.25 \\ b_1 &= \text{balance}(s_{j_5}) \qquad = 1 \\ \text{Job-I/O-Balance} &= \text{mean}(\{b_0, b_1\}) \qquad = 0,625 \end{aligned} \tag{6}$$

4 Data Exploration

DKRZ uses Slurm workload manager for scheduling jobs on Mistral on shared and non-shared partitions. The monitoring system of DKRZ [1] does not capture data on shared Slurm partitions, because it can not assign this data unambiguously to jobs. The problem hides in the (in-house) data collector, more precise, in the usage of proc files as its main data source. The point is that shared partitions can run two or more jobs on a compute node. Job activities can change the I/O counters in the proc files, but the changes can not be traced back to jobs. This kind of monitoring makes observation of individual jobs not feasible. In contrast, a non-shared partition, where only one job is allowed to run, does not suffer from this problem. Monitoring system assumes that all changes in proc files are a result of activities done by a currently running job.

This section deals with job data statistics of 1,000,000 job data downloaded from DKRZ's monitoring system. These data cover a time period of 99 days (from 2019–05–16 until 2019–08–23).

4.1 Job Data

In our experiments, the monitoring system periodically collects various metrics (according to a capture interval) including I/O metrics. The resulting time series is collected for each client node and then assigned to a parallel (SLURM) job. Ultimately, the job data has a 3-dimensional structure: $Metric \times Node \times Time$. Metrics used in our investigation are listed in Table 3a and 3b.

To reduce the overhead of the data acquisition and storage space, metadata and I/O metrics are selected in the following way: Similar metadata operations are combined into three different counters: read, modification and other accesses. Then, create and unlink counters are captured separately as these operations are performance critical. The exact group compositions and metric names are listed in Table 3a.

For I/O, we capture a set of counters: The `read_*` and `write_*` counters provide the basic information about file system access performed by the application. We also include the `osc_read_*`, `osc_write_*` that represent the actual data transfer between the node (Lustre client) and each server[3]. The metrics are listed in Table 3b.

4.2 Analysis Tool

The analysis tool is a product of our continuous research on monitoring data analysis. It requires an initial training, based on a relatively small job dataset, before it can be used for automatic job assessment. Therefore, in the first step, it downloads job data from a system-wide monitoring database and creates statistics about I/O performance on the HPC system. In the second step, these statistics are used for assessing individual jobs. The workflow is illustrated in Fig. 3.

Table 3. Data collectors run on all compute node and capture periodically thirteen I/O metrics (emphasized by **bold font**) and send them to a centralized database. These I/O metrics are computed from around thirty constantly growing proc counters in **`/proc/fs/lustre/llite/lustre*-*/stats`**. (Note: Lustre can reset counters at any time point.)

md_read	= getattr + getxattr + readdir + statfs + listxattr + open + close
md_mod	= setattr + setxattr + mkdir + link + rename + symlink + rmdir
md_file_create	= create
md_file_delete	= unlink
md_other	= truncate + mmap + ioctl + fsync + mknod

(a) *Metadata metrics: data collector form groups of related metadata proc counters, compute sums, and assign the sums to corresponding metadata metrics.*

read_bytes	**osc_read_bytes**
read_calls	**osc_read_calls**
write_bytes	**osc_write_bytes**
write_calls	**osc_write_calls**
Application's I/O requests.	*Lustre client I/O requests.*

(b) *Data metrics: data collectors assign selected data related proc counter values directly to corresponding data metrics (proc counter names are omitted).*

4.3 Data Statistics

About 5.3% of data is empty. For these jobs neither data, nor metadata exist. We suppose these jobs are canceled, before Slurm is able to allocate nodes. After this filtering, 947445 job data are available.

[3] The Lustre client transforms the original file system accesses—made by the application—to Lustre specific accesses, for instance by utilizing the kernel cache. This can have a significant impact on I/O performance, when many small I/O accesses are created but coalesced.

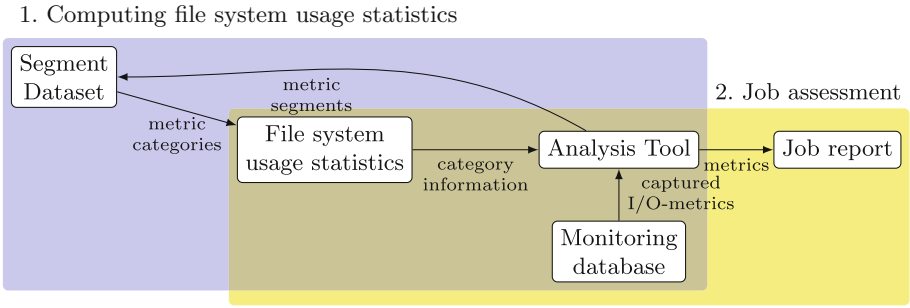
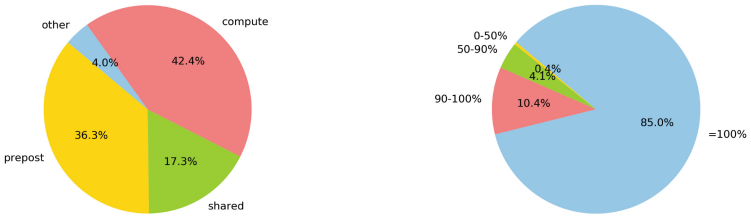


Fig. 3. Analysis tool workflow



(a) Percetage of Slurm usage *(b) Missed data*

Fig. 4. Statistics about Slurm jobs analysed.

Fig. 5. Ordered job runtime (blue line) and 10 min threshold (red line). (Color figure online)

All nodes have access to two file systems; as both deliver similar performance values, a differentiation is not necessary. Therefore, in the course of the paper, we will summarize both partitions to one big partition, called "compute". The nodes of these partitions are reserved exclusively for a job. The monitoring system relies on the assumption that all I/O activities registered on these nodes refers to the running job. Conversely, other partitions can be shared by several jobs. Since the monitoring system captures node related data, monitoring data from these partitions can not be assigned unambiguously to a job. Thus, data from "shared", "prepost", and other small partitions is filtered out. A further filtering

criteria is exit state of jobs. We analyze data only from successfully completed jobs. Statistics for completed jobs for Mistral's large partitions are shown in Fig. 4a. After filtering, 338,681 job data remain for analysis.

The next statistic describes the runtime of the successfully completed jobs. Below the red line are about 45% of jobs that are shorter than 10 min. As these jobs consume only 1.5% of available node hours, we do not expect to find significant I/O loads in there. Figure 5 illustrates the runtime of the remaining jobs, including the 10 min threshold (red line).

During our experiments, we encounter a problem with incomplete data. Sometimes, individual metrics, and occasionally, data from complete nodes are missing. The statistics are shown in Fig. 4b. The reasons can be, that some counters are not available during collector initialization, collectors can crash, or the database is overloaded and is not able to record data. For 4.5% of the jobs less than 90% of data is available, in 10.4% data is complete from 90% to 100%, and in the remaining 85.1% all data is available. It is not harmful for the training to lack some data as metric scores can be computed on partially available data. We believe the approach is sufficiently robust to process such data, but for assessment of individual jobs the results won't be perfectly accurate if they omitted some I/O phases.

5 Evaluation

This section uses our methodology to identify I/O-intensive applications on the Mistral supercomputer by doing a step-by-step evaluation of real data. Therewith, we validate that the strategy and metrics will allow us to identify I/O critical jobs and I/O segments within. The segment size used in the experiments is 10 min.

5.1 Limits

There is no perfect recipe for finding the best quantiles that meets everyone's needs, because file system usage and goals may be different. In our case, identification of outlier jobs requires quantiles in the upper range. We can see this in

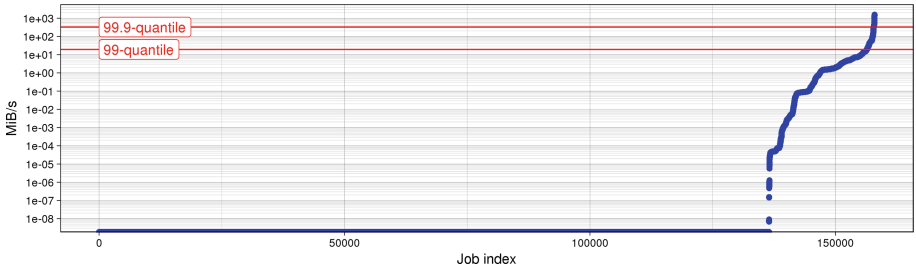


Fig. 6. Training data (subset) for `read_bytes` metric, and q99%- and q99.9%-quantiles (red lines). (Color figure online)

the example of `read_calls` segments in Fig. 6. The most blue dots are located close to 0 Op/s, which means that there is low or no I/O activity in most segments. We separated them by the 99-quantile (lower red line). The remaining high activity segments are significant for identification of high I/O load. The more of them are located in a job, the higher is the probability that this job causes a high I/O load. Additionally, the 99.9-quantile (the upper red line) separates high and critical activity segments. This separation defines segments with an exceptionally high I/O load. Generally speaking, the quantiles choice in this work is based on observations of file system usage on Mistral and rough ideas of what we want to achieve. We suspect it is transferable to other HPC systems, but this point was not investigated and requires a further study.

For limit calculation we use a 30 days training set consisting of 72,000 jobs. Their segmentation results in around 152,000,000 metrics segments. The resulting limits are listed in Table 5.

5.2 Categorization

In the next step, the limits are used for categorization of all job data (about 660 million metric segments) (Table 4). The result of categorization is shown in Table 5.

Table 4. Category scores for Mitral evaluation.

Category name	MScore	Justification for Mscore value
`LowIO`	0	Ignore this category in mathematical expressions
`HighIO`	1	Consider this category in mathematical expressions
`CriticalIO`	4	CriticalIO is at least four times higher than HighIO

The first observation is, that there are less `osc_read_*` and `osc_write_*` metrics reported than for other metrics. The reason for that is the file system changed from Lustre 2.7.14 to Lustre 2.11.0. Unfortunately, since Lustre 2.8, the proc files do not offer the `osc_read_*` and `osc_write_*` metrics anymore. We did not know that and captured incomplete data. (Fortunately, other sources provide this information and we can fix that in the future.) This trifle makes no difference for this concept, as long as data represents typical file system usage. We assume that 17M metric segments form a representative training set and take this opportunity to show the robustness of the approach.

The second observation is that modification of metadata, deleting and creation of files are rare operations. For delete and modify operations, the 99%-quantile is zero, i.e., any segment that has one delete/modify operation, it is considered to be in the category HighIO.

5.3 Aggregation

The conversion of metrics value to the score allows the aggregation of job data on job, node, and metric levels and of incompatible metrics, like `md_delete`

Table 5. Category statistics for training with segments size of 600 s.

Metric		Limits		Number of occurrences		
Name	Unit	q99	q99.9	LowIO	HighIO	CriticalIO
md_file_create	Op/s	0.17	1.34	65,829 K	622 K	156 K
md_file_delete	Op/s	0.00	0.41	65,824 K	545 K	172 K
md_mod	Op/s	0.00	0.67	65,752 K	642 K	146 K
md_other	Op/s	20.87	79.31	65,559K	763K	212 K
md_read	Op/s	371.17	7084.16	65,281 K	1,028 K	225 K
osc_read_bytes	MiB/s	1.98	93.58	17,317 K	188 K	30 K
osc_read_calls	Op/s	5.65	32.23	17,215 K	287 K	33 K
osc_write_bytes	MiB/s	8.17	64.64	16,935 K	159 K	26 K
osc_write_calls	Op/s	2.77	17.37	16,926 K	167K	27K
read_bytes	MiB/s	28.69	276.09	66,661 K	865 K	233 K
read_calls	Op/s	348.91	1573.45	67,014 K	360 K	385K
write_bytes	MiB/s	9.84	80.10	61,938 K	619 K	155 K
write_calls	Op/s	198.56	6149.64	61,860 K	662 K	174 K

and `read_bytes`. This is useful as it allows us to reduce the data for large jobs. Due to inability to aggregate, conventional dashboards contain many plots with detailed information, which, in turn, is hard to grasp and inconvenient to use. With uniform scoring aggregation it becomes an easy task. This is illustrated in Fig. 7. Data is aggregated from detailed view in Fig. 7a to reduced view in Fig. 7b, and finally to one single chart in Fig. 7c.

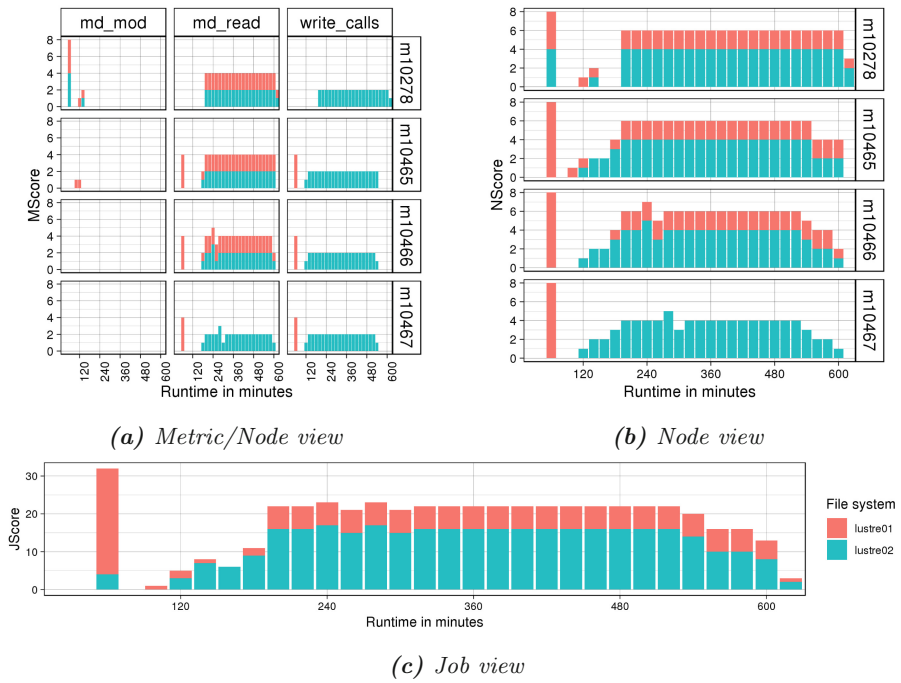
5.4 Metrics Calculation

Metrics calculation is the next logical step in our work. They describe specific I/O behavior by a meaningful number.

5.5 Job-I/O-Utilization (U)

The mean score metric filters non-I/O-intensive jobs out of the dataset. 41% jobs (151,777) have a Job-I/O-Utilization = 0. These jobs are of little interest to us, since they do not produce any noticeable load for our file system. The remaining 59% jobs (218,776) are selected for further investigations.

The distribution of Job-I/O-Utilization is shown in Fig. 8a. The utilization for one file system may be $U = 4$, if the file system is used to 100%. We can observe that for many jobs $U > 4$, which means these jobs are using two file systems at the same time. This may be a copy job that moves data from one file system to another.



(a) Metric/Node view *(b) Node view*

(c) Job view

Fig. 7. Segments visualization at different level of details.

5.6 Job-I/O-Balance (B)

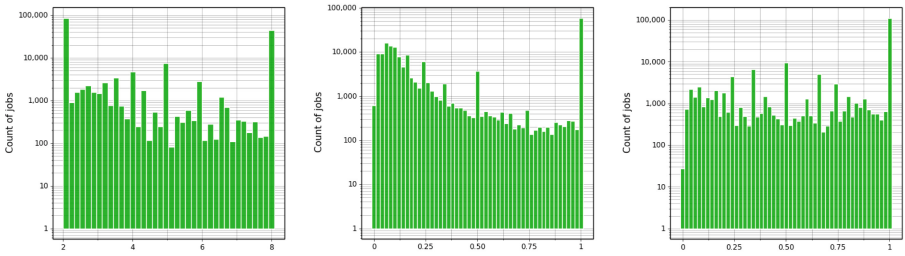
Jobs that are running on 1 node are always balanced. There are about 66,049 (30%) jobs of this kind. Job-I/O-Balance for the remaining 152,727 (70%) jobs are visualized in Fig. 8b. The picture shows that a vast amount of jobs are not using parallel I/O or doing it insufficiently. 17,323 of the jobs are balanced to 50% or more. 4,122 of them are highly optimized and are running with almost 100% optimization.

We have to keep in mind that during categorization, all negligible I/O (i.e., if JScore = 0) is filtered out. That means, the balance metric focuses on significant I/O sizes.

List of jobs ordered by Job-I/O-Balance in increased order gives an overview of jobs with the lowest I/O balance. A closer look at the first entries reveals that Jobs with a fixed number of I/O nodes have also a small I/O balance value, but they are far behind in the list.

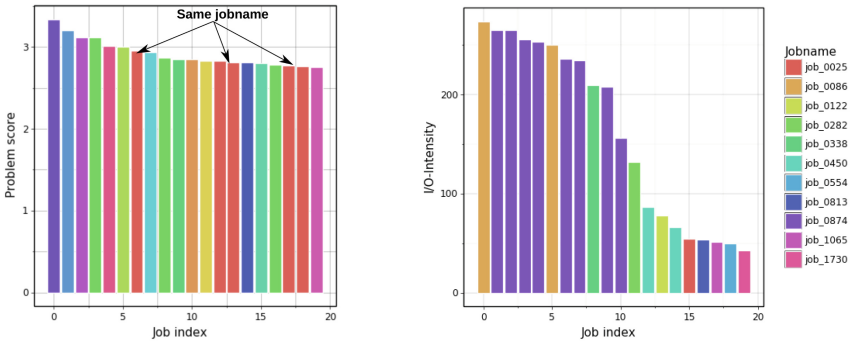
5.7 Job-I/O-Problem-Time (PT)

Surprisingly, we found that 142,329 (65%) jobs are pure I/O jobs, i.e., with Job-I/O-Problem-Time = 1. The other 76,447 (35%) jobs have a Job-I/O-Problem-Time < 1. The peaks in Fig. 8c at positions $1, 1/2, 1/3, 1/4, \ldots$



(a) Job-I/O-Utilization *(b) Job-I/O-Balance* *(c) Job-I/O-Problem-Time*

Fig. 8. Metric statistics



(a) $Problem\text{-}Score = (1 - B) \cdot PT \cdot U$ *(b)* $I/O\text{-}Intensity = B \cdot PT \cdot U \cdot total\_nodes$

Fig. 9. Penalty functions and the Top 20 jobs with a runtime >30 min. The color represents a unique job name. (Color figure online)

are mostly artifacts from short jobs. After filtering out jobs shorter than 2 h, they disappear, but peak at position 1 is still there.

6 Job Assessment

Job assessment is a semi-automated process. In the first step, penalty functions sort jobs according to user-defined requirements. Typically, a function is constructed such that each sub-optimal parameter increases its value. A job list can be sorted automatically by that value. The manual tasks in the second steps are visualization of top ranked jobs and actual assessment.

Based on our initial goals, we define two functions: (1) Problem-Score: for detection of potential inefficient file system usage and I/O-Intensity: for detection of high I/O loads. Both are defined and visualized in Fig. 9. The computation includes B, U, and PT metrics from the previous section and further parameters for computing a single value.

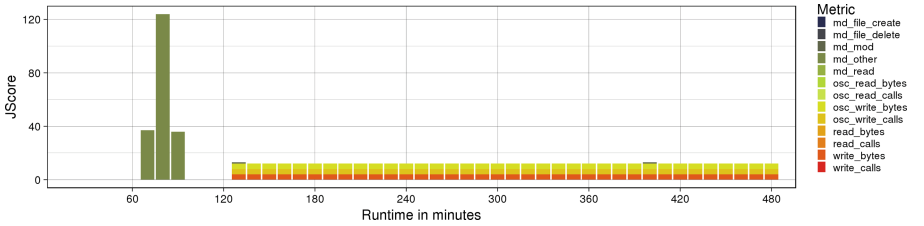


Fig. 10. Problem-Score ≈ 2.9: Nodes: 70; B: 0.05; PT:0.8; U: 7.5. First I/O phase: highly parallel metadata access; Second I/O phase: single node writes.

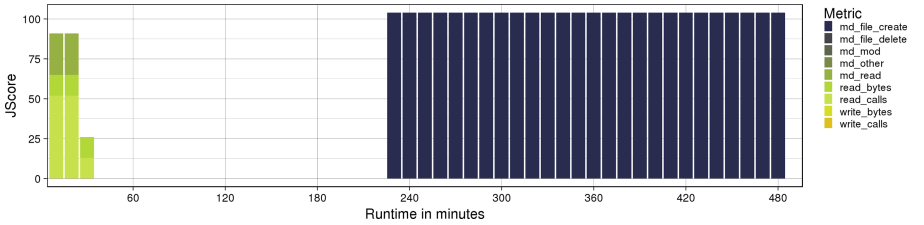


Fig. 11. I/O-Intensity ≈ 29.9; Nodes: 13; B: 1.0; PT: 0.6; U: 3.9.; First I/O phase: fully balanced metadata operations and reads on both file systems; Second I/O phase: fully balanced file create operations on both file systems.

6.1 Problem-Score

The Problem-Score is a product of all metrics, as defined by the penalty function in the Fig. 9a. For illustration, a 70-node job with Problem-Score ≈ 2.9 is visualized on node-level in Fig. 10. It represents a classic case of unoptimized single node I/O. In the picture, we see a short phase of metadata operations, and a 360 min long write phase. The node view (omitted, due to space restrictions) reveals also, that the short phase is fully balanced, and the long phase runs on a single node. The phases can be clearly identified by naked eye in the timeline.

When considering further jobs, we found other recurring and inefficient I/O patterns, e.g., partially or improperly balanced I/O. In all cases, different phases can be easily read from timelines, even if they are connected to each other or running in parallel.

6.2 I/O-Intensity

To identify applications that generate high I/O loads, we have also to consider the number of nodes. Here again, we use the same logic as before, i.e., when I/O load increases, I/O-Intensity must also increase. Now, high balance is a sign for load generation, and can be used directly in the function. All that is reflected in the penalty function in Fig. 9b.

A particularly interesting case is illustrated on job level in Fig. 11. This picture reveals that the job does I/O in two phases. Looking at the metric/node

level (omitted, due to space restrictions), we see that the job (1) operates on both file systems, (2) reads data in the first phase and creates files in the second phase, and (3) both phases are fully balanced. The file creation phase takes longer than 240 min (>50% of job runtime). This extreme behavior can degrade the performance of Lustre metadata servers, affect the runtime of parallel running jobs, and slow down metadata operations for other users. We suppose that users and developers of this application are not aware of that, and store information in different files for reasons of convenience.

This job could be discovered even if all `osc_*` are missing. Obviously, the design of the approach is robust enough to handle missing data.

7 Conclusion

In this work, we developed and evaluated an approach for characterization of I/O phases utilizing monitoring infrastructure widely available and compute derived metrics for phases of application execution. In our experiments, these metrics support the detection of I/O-intensive and problematic jobs.

In the pre-processing part, we split monitoring data into fixed size time windows (segments). Then, data of several thousands of jobs are used for computing statistics representing typical file system usage. Based on statistics and average segment performance, we are able to assign a score value for each segment. These segment scores are the basis for the next processing.

Working with categories and scores significantly simplifies mapping of common I/O behavior to meaningful metrics. We derived the metrics Job-I/O-Balance, Job-I/O-Problem-Time, and Job-I/O-Utilization. These metrics can be used in any mathematical calculation, or in direct comparison of jobs, or for deriving new metrics.

Visualization of the derived metrics is easier to understand than visualization of raw data, e.g., because raw data can have a different semantics, an arbitrary value with high peaks. For the ordinary users, it is not always obvious, if the performance of such values is good or bad. The categorization hides all the details from users.

In our experiments, we could identify applications with high potential to degrade file system performance and applications with inefficient file system usage profile. By investigating raw data, we could verify that the presented approach supports the analysis. In our opinion, this approach is suitable for most current state-of-the-art cluster environments that are able to monitor suitable file system usage counters.

Ultimately, we work toward automatic analysis and reporting tools. Our next step is the data reduction, e.g., the grouping of similar profiles.

References

1. Betke, E., Kunkel, J.: Footprinting parallel I/O – machine learning to classify application's I/O behavior. In: Weiland, M., Juckeland, G., Alam, S., Jagode, H. (eds.) ISC High Performance 2019. LNCS, vol. 11887, pp. 214–226. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34356-9_18
2. Carns, P.: Darshan. In: High performance parallel I/O. pp. 309–315. Computational Science Series, Chapman & Hall/CRC (2015)
3. Carns, P., et al.: Understanding and improving computational science storage access through continuous characterization. ACM Trans. Storage (TOS) **7**(3), 8 (2011)
4. Dayal, J., et al.: I/O containers: managing the data analytics and visualization pipelines of high end codes. In: 2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Ph.D. Forum, pp. 2015–2024, May 2013. https://doi.org/10.1109/IPDPSW.2013.198
5. Kunkel, J., Betke, E.: Tracking user-perceived I/O slowdown via probing. In: Weiland, M., Juckeland, G., Alam, S., Jagode, H. (eds.) ISC High Performance 2019. LNCS, vol. 11887, pp. 169–182. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34356-9_15
6. Kunkel, J.M., et al.: Tools for analyzing parallel I/O. In: Yokota, R., Weiland, M., Shalf, J., Alam, S. (eds.) ISC High Performance 2018. LNCS, vol. 11203, pp. 49–70. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02465-9_4
7. Sivalingam, K., Richardson, H., Tate, A., Lafferty, M.: Lassi: metric based I/O analytics for HPC. CoRR abs/1906.03884 (2019). http://arxiv.org/abs/1906.03884
8. Xie, B., et al.: Predicting output performance of a petascale supercomputer. In: Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2017, pp. 181–192. ACM, New York (2017). https://doi.org/10.1145/3078597.3078614, https://doi.org/10.1145/3078597.3078614