The Journal of High-Performance Storage (JHPS)
Issue 2 - 2021-08-09

Preface

The Journal of High-Performance Storage (JHPS) is a new open-access journal edited by storage experts that unite key features of journals fostering openness and trust in storage research. In particular, JHPS offers open reviews, living papers, digital replicability, and free open access.

The editing team is proud to announce the publication of the second JHPS issue. It contains two articles regarding the important topic of performance analysis. Since our last publication, we hardened the manuscript central[1] that we use to manage submissions. With the ISSN 2748-7814 assigned, the journal is also now recognized as a serial publication. Now that we are confident in the effectiveness of the established workflows and tools, our goal is to foster the adoption of the journal and to refine the workflows for digital replicability.

We thank all authors, reviewers, and readers.

Cordially,
Julian Kunkel, Jean-Thomas Acquaviva, Suren Byna, Adrian Jackson, Ivo Jimenez, Anthony Kougkas, Jay Lofstead, Glenn K. Lockwood, Carlos Maltzahn, George S. Markomanolis, Lingfang Zeng
JHPS Editors

[1] https://jhps.vi4io.org/central/

Note that this PDF is a compendium of the articles. It includes the preface and TOC and the version of all articles at the release date (newer versions of individual articles may exist due to our "living paper" concept). Unfortunately, the embedded links in the PDF of individual articles are lost. We recommend you visit the webpage of the Journal of High-Performance Storage, Issue 2 to download the individual and latest version of an article.

Contents

Holistic I/O Activity Characterization Through Log Data Analysis of Parallel File Systems and Interconnects **3**

A Workflow for Identifying Jobs with Similar I/O Behavior Utilizing Time Series Analysis **31**

Holistic I/O Activity Characterization Through Log Data Analysis of Parallel File Systems and Interconnects

Reviews

Yuichi Tsujita
RIKEN Center for Computational Science, Kobe, Japan
✉ *yuichi.tsujita@riken.jp*

Yoshitaka Furutani
Fujitsu Limited, Tokyo, Japan

Hajime Hida
Fujitsu Limited, Tokyo, Japan

Keiji Yamamoto
RIKEN Center for Computational Science, Kobe, Japan

Atsuya Uno
RIKEN Center for Computational Science, Kobe, Japan

Abstract

The computing power of high-performance computing (HPC) systems is increasing with a rapid growth in the number of compute nodes and CPU cores. Meanwhile, I/O performance is one of the bottlenecks in improving HPC system performance. Current HPC systems are equipped with parallel file systems such as GPFS and Lustre to cope with the huge demand of data-intensive applications. Although most of the HPC systems provide performance tuning tools on compute nodes, there is not enough chance to tune I/O operations on parallel file systems, including high speed interconnects among compute nodes and file systems. We propose an I/O performance optimization framework that utilizes log data of parallel file systems and interconnects in a holistic way for improving the performance of HPC system, including effective use of I/O nodes and parallel file systems. We demonstrated our framework at the K computer with two I/O benchmarks for the original and the enhanced MPI-IO implementations. The analysis by using the framework revealed the effective utilization of parallel file systems and interconnects among I/O nodes in the enhanced MPI-IO implementation, thus paving the way towards holistic I/O performance tuning framework in the current HPC systems.

Keywords: Holistic log data analysis, K computer, FEFS, Lustre, Tofu, MPI-IO

1 Introduction

HPC systems have been facing the performance gap between computing power and I/O performance. Parallel file systems such as GPFS [SH02] and Lustre [Lus] provide vast amounts of storage capacity with high I/O bandwidth to bridge the gap. Most of the I/O optimization research efforts have addressed to improve I/O performance of their implementations in an empirical way using I/O benchmarks rather than analyses of I/O activities on target parallel file systems and interconnect data transfers [BLH+13, BBR+16, TMV+16, VdSB+18, OVW+19]. With an increase in the number of compute nodes and target I/O nodes, it is quite difficult to tune an implementation only through such benchmark runs. Holistic log analysis has been proposed for investigating I/O performance bottlenecks or I/O performance tuning of applications [LWS+18, WSL+18, YJM+19]. Such analysis collects log data about file system activities in addition to I/O performance results of applications. As Yang et al. remarked in their paper [YJM+19], a storage interconnect is another contention point on HPC systems. To this end, they have extended their framework named Beacon to monitor performance counters of InfiniBand network switches.

Within this context, an interconnect is one of the important points to tune not only inter-node communications in applications but also I/O accesses towards underlying storage systems. A profiling tool named Tofu PA [IOIkM12] was provided in the K computer, which acquired statistical information called Tofu PA information regarding communication in the Tofu interconnects [AIH+12] on compute nodes used, with the purpose to tune communications among compute nodes. However, there were no tools to get the Tofu PA information of Tofu interconnects among I/O nodes and I/O activities of its parallel file systems. Similar to other HPC platforms, we conducted I/O benchmark runs to evaluate and tune performance of I/O subsystems in an empirical way in the K computer. For investigating I/O performance bottlenecks and further I/O performance improvements, a well-balanced I/O workload among compute nodes, I/O nodes, and parallel file systems is required to optimize I/O operations. Without knowing the status of I/O nodes and parallel file systems, it is quite difficult to tune I/O operations in HPC applications.

It is expected that the utilization of statistics log data of I/O subsystems such as file system servers and interconnects provides quite useful metrics for I/O performance tuning by examining statistics of I/O request operations or data packet transfers through interconnects. In this context, we have proposed a framework that monitors data transfers on Tofu interconnects on I/O nodes and I/O activities of parallel file systems with the help of log data collected in the system administration in our workshop paper [TFH+20]. To our best knowledge, this is the first work to utilize data transfer information of Tofu interconnects on I/O nodes among the HPC systems using Tofu interconnects in tuning I/O operations. The framework consists of analysis functions for several components: log data collected by *fluentd* [flu], a PostgreSQL database that keeps a large amount of executed job information (JOB-DB), and information about compute and I/O nodes (node information table).

Given a unique ID for each job (JOB-ID), the analysis function of the framework provides us data such as averaged values of essential I/O activities on OSSes used, bandwidth utilization of Tofu interconnects on I/O nodes, and heat-maps about I/O throughput of OSTs used from the log data with the help of the JOB-DB and the node information table. In this paper, we demonstrated how such analyzed data could be used for further performance improvements by examining I/O bottlenecks and unbalanced situations in I/O workload among I/O nodes used in our enhancements in collective MPI-IO implementations named "EARTH on K" [THI14, THK+18]. We have already conducted empirical benchmark evaluations in performance improvements for our enhancement work in the K computer through studies in [THI14, THK+18]. Since only the result obtained from the benchmark evaluations is I/O performance, we had difficulties in tuning the implementation. Once we have introduced the framework for the evaluations, we have noticed which subsystem is bottleneck during the

optimizations.

Compared with our previous paper [TFH+20], we have made further analysis for additional optimizations in an enhanced MPI-IO with and without two-phase I/O in order to show their impact in each underlying I/O subsystems. We have found additional explicit differences in metrics that were given by the framework in not only write operations but also read operations. In addition, we propose scoring scheme in the framework so that we can easily find optimal optimization candidates.

Rest of this paper is organized as follows. In Sec. 2, we discuss related work. A system overview of the K computer including its file I/O subsystems, in which we conducted implementation and evaluation of the proposed framework, is explained in Sec. 3. In Sec. 4, we present the proposed analysis framework. We also explain the "EARTH on K" in Sec. 5, where we briefly present its advanced functions relative to the original MPI-IO. In Sec.6, we report experimental evaluations for the proposed framework at the K computer, and we discuss the usefulness of the proposed framework through examinations about performance improvements achieved by the "EARTH on K" at the K computer. Finally, we conclude the paper in Sec. 7.

2 Related Work

I/O bottlenecks in various applications were studied in [XCD+12, SRC+12]. These studies showed numerous characteristics in terms of I/O access patterns performed by applications on HPC systems using Lustre file systems. I/O monitoring at storage system level was studied in [KZH+14,UW13,MBC+17]. For example, Kunkel et al. proposed a monitoring and analysis framework to suggest and apply performance optimizations automatically [KZH+14]. It assisted locating and diagnosing performance problems. Separately, Uselton and Write [UW13] proposed extended monitoring about metrics available from Lustre using Lustre Monitoring Tool [LMT]. They characterized I/O patterns with their own metric named File System Utilization using obtained metrics. Madireddy et al. conducted I/O system analysis using operation log data, and they demonstrated I/O characterization of each job through correlation between I/O patterns of each job and I/O subsystem activities [MBC+17]. They also discussed the influence of monitoring intervals in system performance. Multi-platform study using system logs of file systems was reported in [LWG+15]. Their cross-platform analysis with I/O behavior collection by Darshan [DAR] showed wide varieties of insights about I/O subsystem operations through comparison among the several HPC systems. Our framework also supports similar functions compared with the above studies. Compared with the above researches, our case also focuses on behavior of interconnects among I/O nodes in the target file system.

Log data collection and analysis for performance tuning were conducted in server-side analysis [LGMV14, LGMV16, XBV+16, PBLT19]. For example, Liu et al. proposed a framework to identify I/O activities automatically using trace log data from file system servers [LGMV14, LGMV16]. Separately, Xu et al. proposed their I/O profiling framework named LIOProf to track I/O activities of on Lustre file system servers including client-side statistics recorded on servers [XBV+16]. Using those metrics, they demonstrated optimization effect in collective MPI-IO implementation. A detailed study in production runs was conducted in [PBLT19] by analyzing server-side log data of parallel file systems to draw new insights about I/O characteristics. However, abovementioned studies did not focus on behavior of interconnects. Our proposed framework supports interconnect monitoring in performance bottleneck investigation to tune I/O optimization with server-side log data of a local file system although the framework has not been used in production runs at the K computer.

Interconnects are also one of the key components in HPC systems. Monitoring data transfers of interconnects tells us a hot-spot of traffic congestion for instance, and such ap-

proaches succeeded in analysis of application activities and performance impact associated with the traffic condition [ZGL16, KGP+18, CJH+19, YJM+19]. Zimmer et al. demonstrated their monitoring framework to collect detailed stats information using their daemon program named I/O Router Congestion Daemon with monitoring performance counter of Gemini interconnects in the Titan at the Oak Ridge Leadership Computing Facility [ZGL16]. Kumar et al. utilized performance counter of Gemini interconnect in the Titan too [KGP+18]. They analyzed and characterized errors and traffic congestion on Gemini interconnects. Chunduri et al. succeeded in execution time predictions of applications by analyzing traffic congestion information obtained from performance counters of Aries interconnects in the Theta at the Argonne Leadership Computing Facility [CJH+19]. They introduced a machine learning approach in their prediction. However, the above studies were not sufficient to characterize I/O activities on parallel file systems in HPC systems. Within this context, Yang el al. pointed out that storage interconnect is another contention point on HPC systems [YJM+19]. They have extended their I/O monitoring framework to monitor performance counters of InfiniBand network switches.

Recently, holistic I/O monitoring has been proposed in many research works [LWS+18, WSL+18, YJM+19]. For example, Lockwood et al. proposed a holistic I/O monitoring framework named TOKIO [LWS+18]. It consisted of several components for monitoring, analysis, and visualization for administrators and users. Separately, Wang et al. proposed a monitoring and analysis framework named IOMiner [WSL+18], in which Darshan [DAR] was introduced to collect I/O performance metrics. Application users can easily identify root cause of poor I/O performance with the framework from vast amounts of log data associated with I/O subsystems. Yang et al. proposed a monitoring framework named Beacon [YJM+19]. This framework provided a collection of monitoring tools for Metadata Servers (MDSes) and Object Storage Servers (OSSes) and analysis functions, including some visualization interface. Their extended monitoring for InfiniBand networks covered to find network contention in I/O operations. The above studies are similar to our study regarding holistic approach to characterize I/O activities.

On the other hand, our study addresses holistic I/O activity analysis through log data analysis of Tofu interconnects and parallel file systems including associated I/O nodes. The uniqueness of this framework is a holistic analysis approach using data transfer status on the Tofu interconnects among I/O nodes and associated I/O activity traces at parallel file systems.

3 K computer and Its File System Monitoring

3.1 Overview of the K computer

The K computer finished its operation for about seven years in August 2019. The system had two-layered file systems, a local file system (LFS) and eight volumes of a global file system (GFS), as shown in Figure 3.1. The LFS was a scratch high-performance storage space which was used during computations, while the GFS was used to store programs and data with high redundancy. An enhanced Lustre named Fujitsu Exabyte File System (FEFS) [SSK12] that was based on Lustre version 1.8 was equipped to build both file systems. The K computer consisted of 82,944 compute nodes and 5,184 I/O nodes, where every system rack consisted of 96 compute nodes and six I/O nodes. Every compute node and I/O node were connected through the Tofu interconnect in a six-dimensional (6D) mesh/torus network represented by X, Y, Z, A, B, and C. Tofu links of X, Z, and B were connected in a torus configuration, while those of Y, A, and C were connected in a mesh configuration. However, torus configuration of the Z-link was only available in I/O accesses through I/O nodes because I/O nodes were included only in I/O accesses. In other cases, the Z-link was used in a mesh configuration for inter-node communications by application jobs.

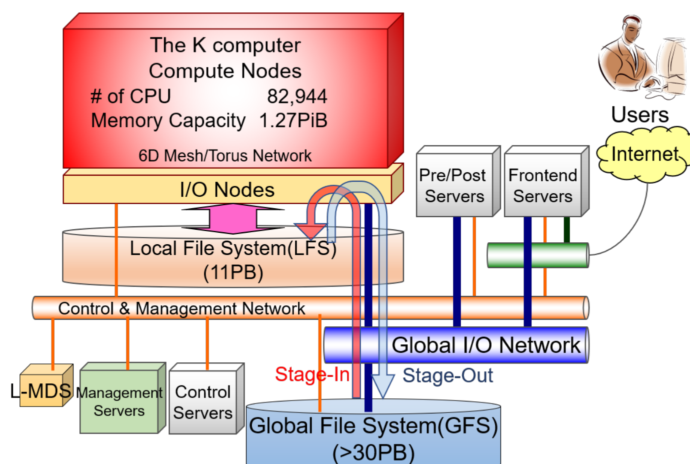


Figure 3.1: System configuration of the K computer.

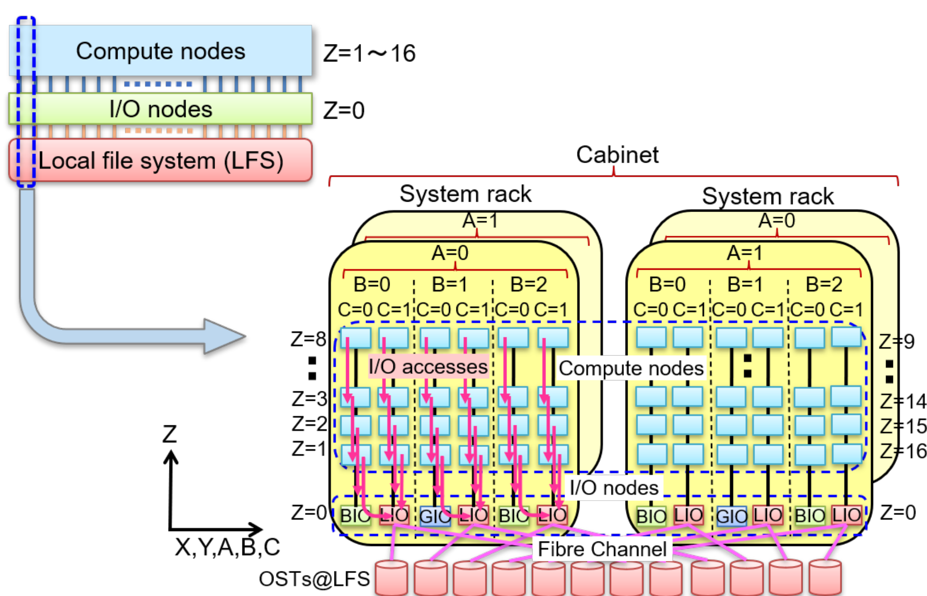


Figure 3.2: Subset of system racks of the K computer and I/O accesses from compute nodes towards the LFS.

Figure 3.2 depicts the configuration of a subset of system racks and I/O accesses from compute nodes towards the LFS. Each cabinet consists of two system racks, which have two groups separated by the A-link position (A=0 and 1) consisting of 48 compute nodes each. Every compute node was located on Z-link positions ranging from Z=1 to 8 and from Z=9 to 16 in the groups of A=0 and 1, respectively, while I/O nodes were on Z=0 in both groups. Boot-I/O nodes (BIOs) were responsible for system software start-up and Global I/O nodes (GIOs) were the gateways in accessing the GFS. The LFS was accessible from compute nodes through OSSes running on the local-I/O nodes (LIOs). Every node including I/O nodes consisted of Tofu network router (TNR) [AIH+12] where each TNR had 10 communication links ($X+$, $X-$, $Y+$, $Y-$, $Z+$, $Z-$, A, $B+$, $B-$, and C) to construct a 6D mesh/torus network.

The number of available OSTs at the LFS was uniquely configured based on the assigned compute node layout according to the I/O zoning scheme [Sum11]. I/O zoning scheme was introduced in order to mitigate I/O interference on OSTs and I/O nodes among jobs by assigning I/O nodes and OSTs on the same Z-link with compute nodes used. In Fig. 3.2, I/O

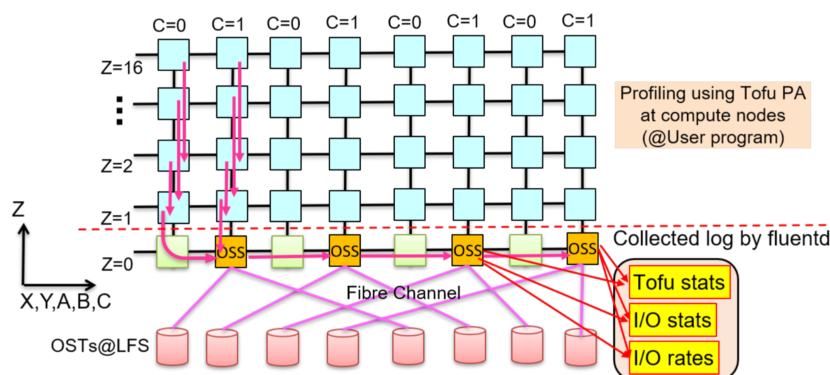


Figure 3.3: Log collection from I/O nodes.

accesses from compute nodes at the system rack are illustrated on the left side. I/O nodes on the same Z-links were configured to work with compute nodes that issued I/O requests, and those I/O nodes take part in data transfers during I/O accesses on the LFS. It is noted that I/O paths from compute nodes were automatically routed to the corresponding I/O nodes (either of BIO, GIO, and LIO) on the same Z-link, then routed to a target OSS running on an LIO.

Performance profiling tools including Tofu PA addressed to tune performance of compute nodes and communications among compute nodes. It is noted that similar profiling tools are available at our current HPC system, the supercomputer Fugaku [The] (hereinafter, Fugaku). The tools succeeded to leverage high levels of computing potential of the K computer, especially in tuning applications utilizing the large number of compute nodes in terms of data transfer status of each node in addition to CPU and memory utilization. The only way to tune I/O operations had been benchmark evaluations because there was not any I/O profiling tool for users to profile activities of I/O nodes and parallel file systems. Therefore, it was quite difficult in I/O operation tuning using only the existing profiling tools.

3.2 Log collection for monitoring the LFS

We have addressed to extract I/O activity information of the LFS in order to investigate operation status of the LFS for not only finding malfunctions but also performance tuning. In this context, we conducted to collect log data from servers associated with the LFS during the K computer operation, as shown in Figure 3.3. We have deployed *fluentd* to collect performance metrics associated with I/O operations from 5,184 I/O nodes including 2,592 LIOs which also acted as OSSes for the LFS.

The proposed analysis framework utilized the following three log data collection groups from large amounts of collected information by *fluentd*.

- `Tofu stats`: Data transfer status metrics of I/O nodes on each Tofu interconnect link (the number of transferred packets, amount of transferred data size, and others)
- `I/O stats`: Statistics of I/O requests obtained from `/proc/fs/lustre/ost/OSS/ost_io/stats` on every OSS
- `I/O rates`: Amount of size in read and write operations on every OST

Only the `I/O stats` was collected at one minute intervals, while the rest were collected at ten minute intervals. We have selected the ten minute intervals for the I/O-related monitoring as trial in a conservative manner not to affect I/O node activities for stable production runs from our empirical study. We conducted the trial monitoring in the last few months of the K computer operation. Limited storage space for the I/O related log-collection was another

reason for the ten-minutes intervals. In the last few months of the K computer operation, we already collected huge amounts of recorded information on the log-collection server from other high-priority components of the K computer for a long time.

The `Tofu stats` consisted of the following packet processing metrics of the ten Tofu links, which were obtained from the TNR of each I/O node through the Tofu PA information in each ten minute interval:

- Cycle counts until target transfer buffer was available in packet transfers
- Amount of transferred data size

It is noted that the cycle counts in the `Tofu stats` corresponded to congestion status since unavailability of transfer buffers in packet processing closely corresponds to packet transfer congestion. We conducted to retrieve those metrics from the TNR at every I/O node during the K computer operation for about a few months until the end of the K computer operation.

The `I/O stats` consisted of the same statistics with an original Lustre, where we especially focused on the three statistics, `req_qdepth`, `req_active`, and `req_waittime`. These statistics provided the status of I/O requests coming from compute nodes through I/O nodes. For instance, a large value in both `req_qdepth` and `req_waittime` indicated very busy status of OSSes or idle status of OSSes waiting for the next operation due to heavy load of an MDS before I/O accesses. Such situation was not suitable for effective I/O operations. Since `req_active` indicated the number of active threads for I/O operations, high numbers in `req_active` indicated a very good condition in terms of I/O accesses.

The `I/O rates` provided I/O throughput status at each OST over time. Collected I/O throughput information showed I/O behavior on each OST such as how much I/O bandwidth was achieved in each OST or how about I/O load balancing was for instance. Due to the reasons described above, we examined I/O activities in the two I/O benchmark runs at ten minute intervals as trial, where each I/O benchmark run took around ten minutes so that we could observe I/O activities of each I/O benchmark run. Minimization of monitoring interval time is our future work in Fugaku.

3.3 Database for executed jobs

A database to store job information named JOB-DB was built on a PostgreSQL database server to collect and refer to job information executed in the K computer. The JOB-DB kept compute nodes used, compute node layout, elapsed time, and start and finish times of job execution, which were associated with a JOB-ID, for instance. Therefore, we could refer to information about a target job from the JOB-DB by specifying a JOB-ID.

4 Analysis Framework for I/O Activities

As described in Sec. 3.2 and Sec. 3.3, we had monitoring and log collection environment for each component in the K computer operation. However, there was not any environment to have holistic I/O activity analysis for the purpose of investigation and performance tuning. Considering the complexity in I/O subsystems and I/O software stacks in a large scale of HPC systems such as the K computer, we have built an analysis framework in cooperation with the existing monitoring components described in Sec. 3.2 and Sec. 3.3. Since the framework needs to work together with several components such as the JOB-DB built on a PostgreSQL database and log data collected by *fluentd*, we have conducted to build the framework using *Python*. Figure 4.1 depicts an overview of the implemented analysis framework, which is connected with associated log data collected by *fluentd* and the JOB-DB to analyze I/O activities on I/O nodes and the LFS. Given a target JOB-ID, the framework retrieves information of the JOB-ID such as 6D mesh/torus network positions of compute nodes used and

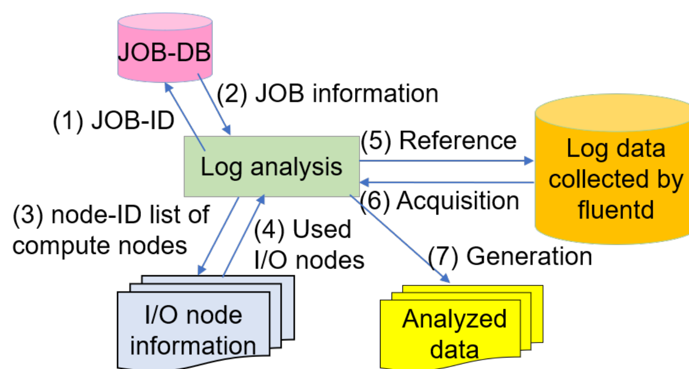


Figure 4.1: Functional overview of implemented analysis framework.

system racks used from the JOB-DB. Such information about compute nodes used and system racks is utilized to find I/O nodes used including LIOs from the I/O node table because the assigned I/O node layout is automatically configured by the shape of assigned compute nodes. Besides, start and finish times of the target job obtained from the JOB-DB are used to pick up essential information associated with the JOB-ID from a large amount of log data collected by *fluentd*.

Once the framework collects all essential information, its log analysis function figures out and gives the following information for the given JOB-ID:

- Maximum waiting time of each interconnect at each I/O node used (T^{max}_{wait}) from `Tofu stats` log collection
- Peak bandwidth utilization ratio of the interconnects relative to the theoretical bandwidth during job execution (R_{BW}) from `Tofu stats` log collection
- I/O throughput in both write and read operations on each OST used from `IO rates` log collection

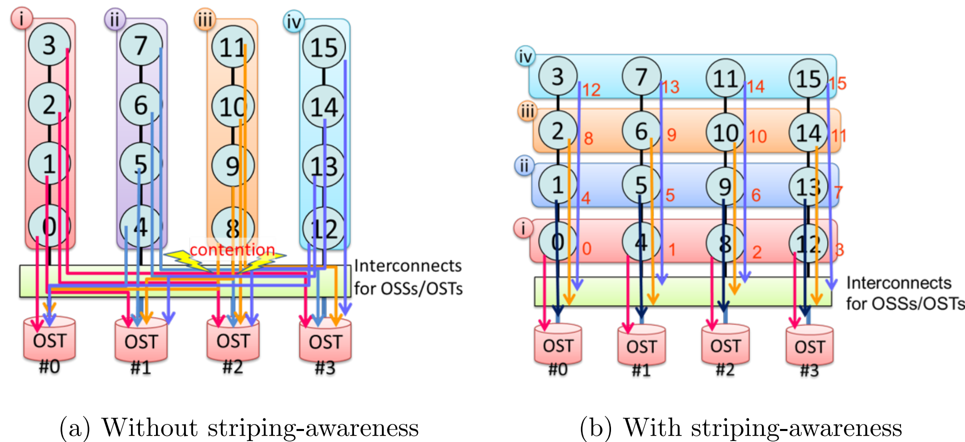
The former two performance values were calculated by using the packet transfer metrics obtained from the TNR of Tofu links used, which were obtained from `Tofu stats` log collection. The function converts the cycle counts obtained from the TNR into time values in the unit of a second for the T^{max}_{wait}. While the R_{BW} was obtained by dividing the peak bandwidth in Tofu links of the job with the theoretical bandwidth. Note that the peak bandwidth was obtained by dividing transferred packet size with elapsed time of the specified job. In the proposed framework, we used the R_{BW} to examine the effectiveness in packet transfers associated with I/O operations.

While the I/O performance values were obtained by dividing an amount of data size in read and write operations with a monitoring interval time (600 seconds in the current configuration) in each snapshot in order to know I/O throughput at each OST used. Once the analysis function is executed, data are stored in the CSV format and associated heat-map image data are stored in the PNG format.

5 Enhanced MPI-IO Implementation: EARTH on K

In order to examine effectiveness of the proposed framework, we have conducted to evaluate MPI-IO benchmark runs. In this context, we have picked up our enhanced MPI-IO implementations in addition to the original MPI-IO implementation at the K computer.

MPI-IO is an I/O interface including parallel I/O in the MPI standard [MPI]. An MPI library for the K computer supports MPI-IO functions for the FEFS using MPI-IO implementation named ROMIO [TGL99]. Two-phase I/O optimization in ROMIO improves collective



(a) Without striping-awareness (b) With striping-awareness

Figure 5.1: Aggregator layouts with and without striping awareness.

MPI-IO performance in accessing non-contiguous data layouts in each process by rearrangement to form large data access space as much as possible in each process performing I/O (aggregator). Although two-phase I/O optimization of ROMIO improves collective MPI-IO performance, the implementation on the K computer uses an old implementation of ROMIO which is not optimized for Lustre. Therefore, the original MPI-IO implementation is not suitable for the FEFS to achieve high I/O performance.

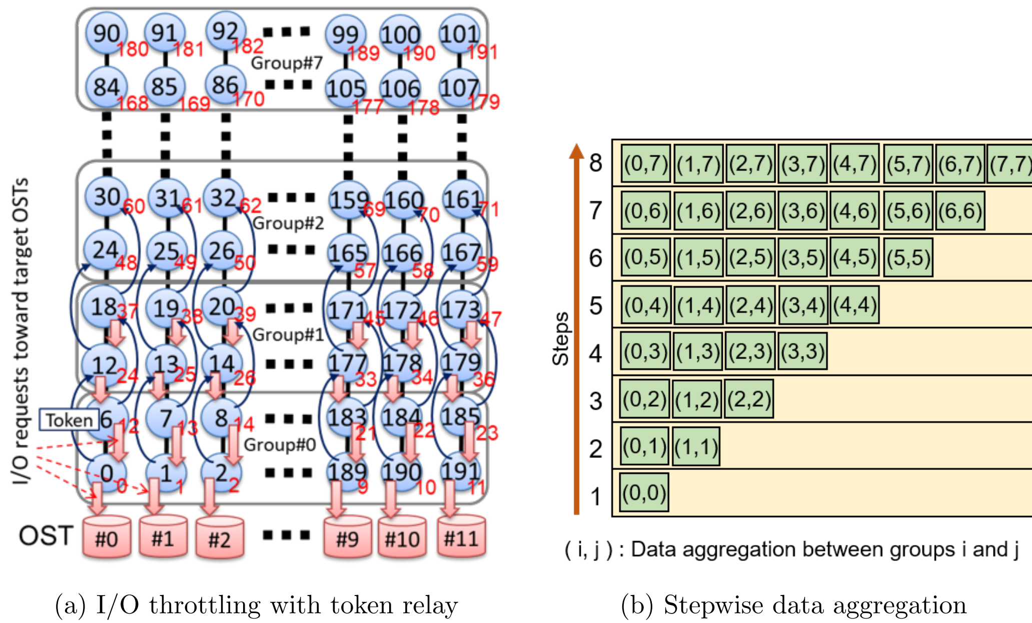
The current ROMIO with the improved two-phase I/O for Lustre [Lus08] has the potential to improve performance on the FEFS. Our enhanced MPI-IO implementation named "EARTH on K" (hereinafter, EARTH) has been developed for the K computer based on the improved two-phase I/O with topology-aware performance optimizations for collective MPI-IO at the FEFS. We have already reported performance improvements by using the EARTH in some conference papers [THI14,THK+18], there is not any evidences what kind of improvements has been achieved in underlying interconnects among I/O nodes or OST activities. In this context, we have investigated some advanced functions of this implementation with the proposed framework.

Compared with the original MPI-IO, EARTH has advanced optimizations controlled by the following three parameters represented by `agg`, `req`, and `rr`, respectively:

- `agg`: Striping-aware aggregator layout
- `req`: I/O throttling and associated stepwise data aggregation with a given number of I/O requests per step
- `rr`: Round-robin aggregator layout among compute nodes

ROMIO deploys one aggregator in each compute node, and its layout is dependent on MPI rank layout among compute nodes. In HPC systems, users have been focusing on MPI rank layout for communication performance among compute nodes. However, such optimizations are not always suited for aggregator layout with respect to interconnects among compute nodes and I/O nodes or layout of OSSes/OSTs of a Lustre file system. In such layout, contention in data transfer happens on network paths among compute nodes and a Lustre file system.

The striping-aware aggregator layout mitigates data transfer congestion by suitable aggregator layout. Figure 5.1 shows aggregator layouts with and without striping awareness in accessing four OSTs by 16 aggregators, where numbers in circles represent MPI ranks and the numbers ranging from i to iv represent the order of striping accesses. By placing aggregators in an MPI rank order as shown in Figure 5.1a, we may face contention in a path towards target OSTs. On the other hand, a striping-aware layout shown in Figure 5.1b, renumbering the



(a) I/O throttling with token relay (b) Stepwise data aggregation

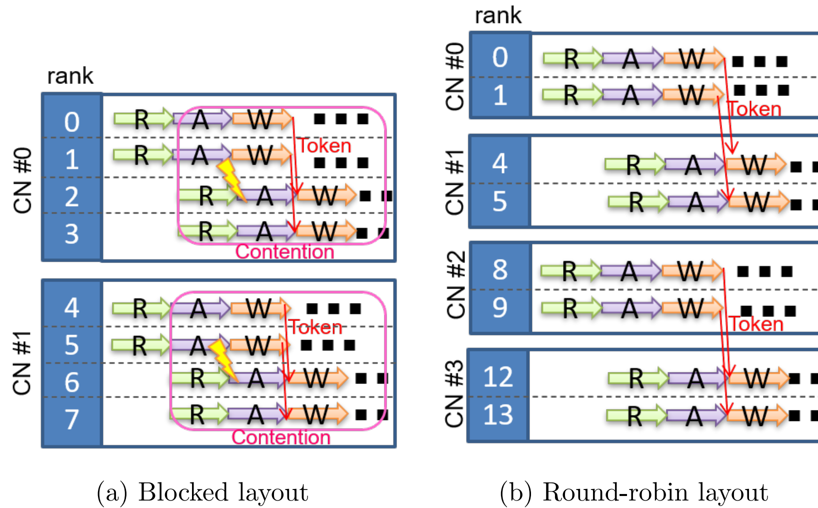
Figure 5.2: I/O request throttling with stepwise data aggregation.

order of aggregators in the red-colored numbers, eliminates data transfer congestion on every I/O path because I/O flows of every I/O path towards a target OST are evenly distributed for a striping access pattern against OSTs.

Figure 5.2a illustrates I/O throttling scheme operated by 192 aggregators accessing 12 OSTs, where we assume every process acts as an aggregator. Numbers in circles represent MPI ranks, and red-colored numbers neighboring to the circles are the aggregator layout orders configured by the striping-aware aggregator layout. Two-phase I/O consists of repetitive operations of data aggregation on every aggregator and I/O accesses by aggregators. We may have I/O request contention on OSTs if we have I/O accesses simultaneously from all aggregators. The I/O throttling scheme shown in this figure alleviates I/O request contention on OSTs by issuing I/O requests from aggregators in each group in a stepwise manner from the younger number group by relaying tokens from an aggregator that finishes I/O accesses to a corresponding aggregator in the next group. The number of groups can be tuned at runtime through `MPI_Info_set()` or an environment variable. It is also noted that EARTH also supports I/O request throttling even if we disable two-phase I/O in collective MPI-IO.

Stepwise data aggregation shown in Figure 5.2b is another optimization associated with the I/O throttling. Compared with simultaneous data aggregation by all the processes, we can eliminate congestion in data transfers among compute nodes by stepwise data aggregations issued from younger number group. In this figure, we represent data aggregation between the groups numbered by i and j as (i, j), which is equivalent to (j, i). In the first step (step=1), processes in the first group numbered as 0 initiates data transfers to aggregators on every group numbered from 0 to 7, described by $(0, k)$, where $k =$ 0-7. However, in the first step, only the $(0, 0)$ is carried out because other groups are not ready in data aggregation. In the next step (step=2), the second group numbered as 1 initiates data aggregation of $(1, k)$, where $k =$ 0-7, and only data aggregations of $(0, 1)$, which is equivalent to $(1, 0)$, and $(1, 1)$ complete in this step. Finally in the last step (step=8), the last group numbered as 7 initiates data aggregations of $(7, k)$, where $k =$ 0-7. Then the remaining aggregations denoted by $(k, 7)$, where $k =$ 0-7, complete in this step.

Figure 5.3 shows examples of blocked and round-robin aggregator layouts, where we have eight aggregators from 16 processes deployed among four compute nodes in a blocked layout. Horizontal colored arrows represented by R, A, and W stand for read, data aggregation, and



(a) Blocked layout (b) Round-robin layout

Figure 5.3: Aggregator layouts with blocked and round-robin manners, utilizing I/O throttling and stepwise data aggregation in write operations.

write phases in two-phase I/O during collective write operations, respectively. As shown in this figure, I/O throttling scheme relays tokens among aggregators. When we have a blocked aggregator layout as shown in Figure 5.3a, four processes in the two compute nodes (*CN #0* and *CN #1*) work as aggregators, which are from 0 to 7 in MPI ranks. As a result, we may have contention within the same compute node in performing each phase of two-phase I/O. It is also noted that such layout leads to high I/O workload in each node compared with other compute nodes without aggregators. Meanwhile, the round-robin aggregator layout in Figure 5.3b can distribute I/O workload evenly among compute nodes, and this layout also prevents aggregators from I/O access contention within the same compute node by reducing the number of aggregators in the same compute node.

Although the above enhancements outperformed the original version in an empirical study using I/O benchmark runs [THI14, THK+18], there was not any investigations about the performance impact of those optimizations in I/O nodes or underlying file systems. One of the main reasons is the lack of tools to characterize optimization effects in data transfers among I/O nodes and I/O accesses against the LFS at the K computer. By using the proposed framework, we examined their advanced features at the K computer in the following section.

6 Experimental Evaluation

We conducted to examine functionalities of the proposed framework at the K computer through two I/O benchmark runs, IOR [IOR] and HPIO [CCkL+06], about the original MPI-IO implementation and EARTH. Although the K computer was already dismantled, we believe that results and experiences obtained from the evaluations provide useful hints for current HPC systems including Fugaku. Although IOR supports two file creation modes, accessing file per rank and shared access to a single file, we utilized the shared access mode with collective MPI-IO. Meanwhile, HPIO supports I/O accesses for non-contiguous data layout, and we performed collective MPI-IO for the data layout. For both benchmark runs, we initiated 12,288 processes on 3,072 compute nodes forming a logical 3D layout of $8 \times 12 \times 32$ in order to eliminate I/O interference from other jobs. According to the 3D layout of assigned compute nodes, 192 OSTs were assigned for parallel I/O, and we set 192 as a stripe count to use all available OSTs. We set 256 MiB and 64 MiB in stripe size in the IOR run and the HPIO run, respectively.

In both benchmark runs, every processes worked as aggregators with ascending order

layout in MPI ranks from zero in the original MPI-IO under default configuration. On the one hand, 6,144 processes were chosen to be aggregators in the EARTH case in order to examine performance impact of aggregator layout among compute nodes according to optimization configuration of the EARTH. In this paper, `original` stands for the original MPI-IO, while a combination of the three optimization parameters, `agg`, `rr`, and `req`, indicates MPI-IO of the EARTH. Concerning the EARTH use case, `agg=1` stands for striping-aware aggregator layout and `rr=1` denotes round-robin aggregator layout among compute nodes. A zero value in each case stands for deactivation in the corresponding layout optimization. The last parameter `req` with a number describes the number of I/O requests going to each OST per step in I/O throttling and stepwise data aggregation except that `req=0` denotes deactivation of I/O throttling and stepwise aggregation.

In the IOR benchmark run, we conducted collective MPI-IO without two-phase I/O. In this paper, we describe collective MPI-IO with and without two-phase I/O by giving "T:" and "N:" at the beginning of the parameter configuration notation such as `T:original` and `N:original`, respectively.

6.1 Benchmark configuration

We conducted to evaluate collective MPI-IO in the two benchmark runs, IOR and HPIO with the proposed framework. In both cases, we enabled two-phase I/O implemented in ROMIO.

6.1.1 IOR

The following command was executed in write operations to generate a shared file of 3 TiB (= 256 MiB × 12,288) per iteration:

```
$ ior -i 5 -a MPIIO -c -U hints_info -k -m -vvv -w -t 256m -b 256m \
-o ${TARGET_DIR}/test-IOR.dat -d 0.1
```

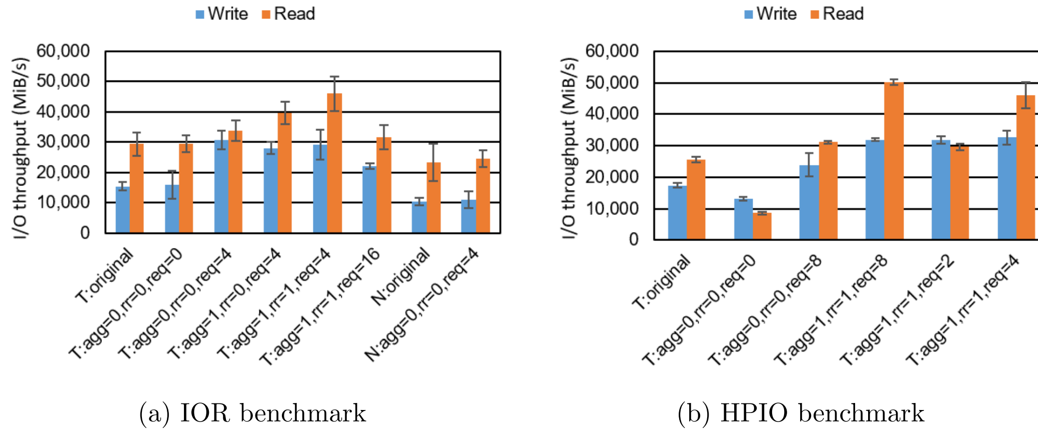
We performed read operations with the same command changing "-w" by "-r", followed by write operations with the above command in every optimization parameter configuration. "`hints_info`", is a file describing some hints associated with I/O operations such as the number of processes per node and so forth. A target file (`test-IOR.dat`) was generated in the directory (`${TARGET_DIR}`) with 192 stripe count. We carried out collective MPI-IO with and without two-phase I/O by enabling or disabling "`romio_cb_write`" and "`romio_cb_read`" through the "`hints_info`."

6.1.2 HPIO

We executed the following command for write operations to generate a shared file of about 2.1 TiB (≈ (5,992 B + 256 B) × 12,288 × 30,729 - 256 B) per iteration, followed by read operations in non-contiguous access pattern on a target file with specifying the number of processes per node (`-H cb_config_list=*:4`) and parameter to tune the number of aggregators to be 6,144 (=192 × 32):

```
$ hpio -n 12288 -n 0010 -r 6 -B -s 5992 -c 30729 -p 256 -m 01 -O 11 -f 0 \
  -S 0 -a 0 -g 2 -H cb_config_list=*:4 -H romio_lustre_co_ratio=32 \
  -d ${TARGET_DIR} -w 1
```

Note that the option, "`cb_config_list`" was available only for the EARTH case, and thus all processes worked as aggregators in the original case as we explained. The target file was generated in the directory `${TARGET_DIR}` with 192 stripe count. We conducted the collective MPI-IO only with two-phase I/O because collective MPI-IO without two-phase I/O was time-consuming under the non-contiguous access pattern and it was difficult to perform in a limited machine time.



(a) IOR benchmark (b) HPIO benchmark

Figure 6.1: Benchmark results of the original MPI-IO and EARTH with several optimization configurations by using (a) IOR and (b) HPIO.

6.2 Benchmark results

Figure 6.1 shows averaged I/O throughput values with standard deviations for the IOR and HPIO benchmarks.

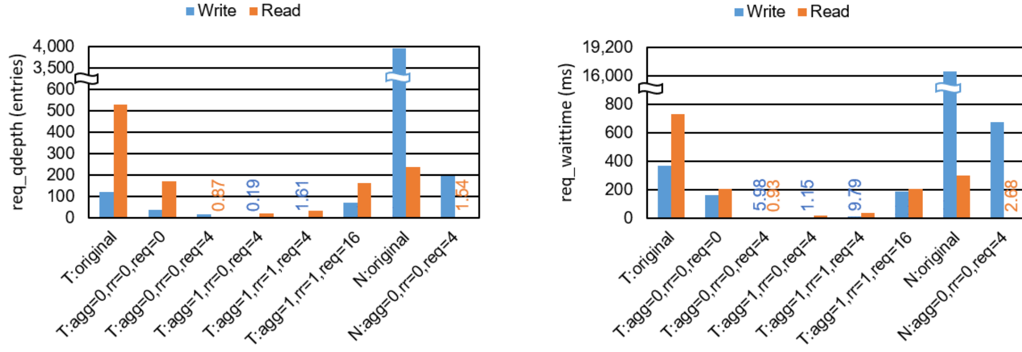
The original MPI-IO operations with and without two-phase I/O represented by `T:original` and `N:original` showed poor performance in both read and write operations in the IOR runs. The same situation was observed for the `T:original` case in the HPIO runs. EARTH with full optimization in aggregator layout, I/O request throttling, and stepwise data aggregation outperformed other cases by setting four requests per step (`T:agg=1,rr=1,req=4`) in the IOR runs and eight requests per step (`T:agg=1,rr=1,req=8`) in the HPIO runs. However, performance was degraded by changing the number of requests per step or deactivating aggregator layout optimization. In addition, the EARTH case using only I/O throttling without two-phase I/O (`N:agg=0,rr=0,req=4`) could not improve I/O performance compared to the original case (`N:original`) in the IOR runs.

Although we learned optimization effects through such empirical benchmark runs in our previous research papers [THI14, THK+18], it was not clear about the performance impact of the optimization configuration on I/O nodes, Tofu links among I/O nodes, and the LFS. We report investigations of each component using the proposed framework in the following subsections.

6.3 I/O request status at file system servers

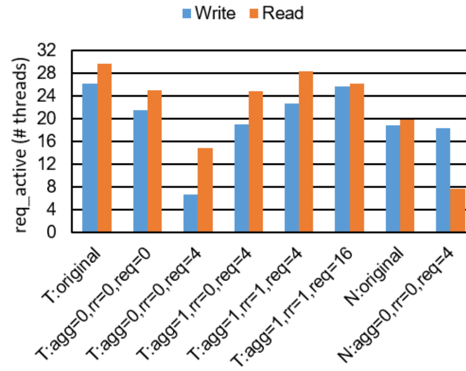
Figure 6.2 shows the mean values of `req_qdepth`, `req_waittime`, and `req_active` from `I/O stats` log collection during I/O operations at the IOR benchmark run. As shown in Figure 6.2a, the two original cases with or without two-phase I/O, `T:original` and `N:original`, had the largest number of requests in a request queue in each I/O operations with or without two-phase I/O. Figure 6.2b shows that those cases also took the longest time to proceed requests in each I/O operations with or without two-phase I/O. Additionally, Figure 6.2c shows the highest number of I/O threads in the original case with two-phase I/O. Note that the maximum number of threads at each OSS of the LFS was 32 at the K computer. Through these results, we determined that the two original cases, `T:original` and `N:original`, were not suited for I/O request processing at OSSes.

While the EARTH use case with good I/O performance (`T:agg=1,rr=1,req=4`) showed small number of requests in the queue, as shown in Figure 6.2a. Figure 6.2b also shows the fact that this case took quite short times to process I/O requests. Additionally, Figure 6.2c shows many I/O threads were active in this case.



(a) `req_qdepth`: The vertical axis is expanded from 600 to 3,500

(b) `req_waittime`: The vertical axis is expanded from 800 ms to 16 s



(c) `req_active`

Figure 6.2: Mean stats values obtained from OSSes using our analysis framework during the IOR benchmark run, where numbers represent very small values.

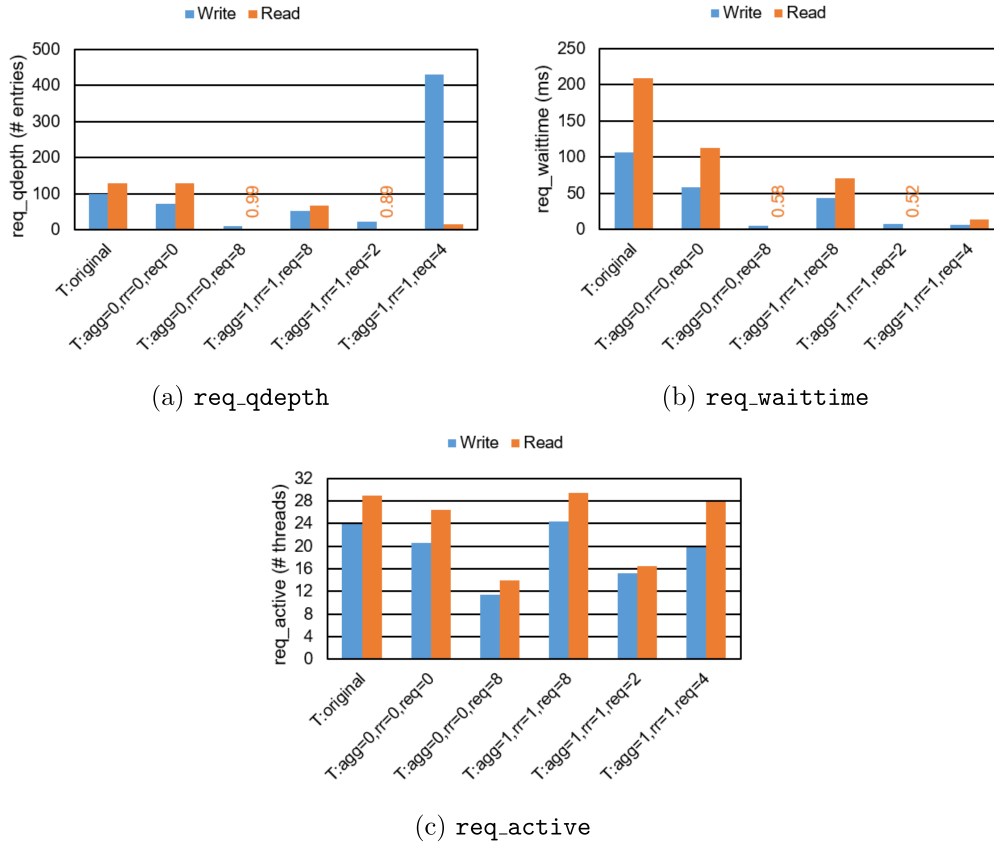
Figure 6.3 shows the same statistics obtained in the HPIO benchmark run. Similar to the IOR run, the original use case was not good compared with the EARTH use case with good optimization configuration indicated by `T:agg=1,rr=1,req=8`.

6.4 Bandwidth utilization and waiting times in data transfers on Tofu interconnects of I/O nodes

Figure 6.4 shows mean values of (a) R_{BW} and (b) T_{wait}^{max} on Tofu links of I/O nodes used. Concerning bandwidth utilization shown in Figure 6.4a, the original MPI-IO use case showed the lowest utilization, while the full set of EARTH optimizations such as `T:agg=1,rr=1,req=4` led to higher levels of bandwidth utilization relative to other cases. While the use cases without two-phase I/O, `N:original` and `N:agg=0,rr=0,req=4`, showed larger number of requests in queue, especially in write operations. By considering effectiveness in data transfers among I/O nodes via Tofu interconnects, a higher utilization was preferable. Within this context, the above optimized case was suitable for I/O optimization.

Figure 6.4b shows that the enhanced implementation without aggregator layout optimization indicated by `T:agg=0,rr=0,req=4` took the longest times. It is also noted that this case also performed the lowest bandwidth utilization in write operations, as shown in Figure 6.4a. It is notable that the lack of aggregator layout optimization in the EARTH case led to a negative impact in data transfers on Tofu interconnects among I/O nodes.

In a similar way, Figure 6.5 shows bandwidth utilization ratios and waiting times in data transfers on Tofu links of I/O nodes used at the HPIO benchmark run. The EARTH



(a) `req_qdepth` (b) `req_waittime`

(c) `req_active`

Figure 6.3: Mean stats values obtained from OSSes using our analysis framework during the HPIO benchmark run, where numbers represent very small values.

use case with the best configuration (`T:agg=1,rr=1,req=8`) also outperformed other cases in Figure 6.5a. This case also minimized waiting times in both read and write operations among the EARTH use cases in Figure 6.5b.

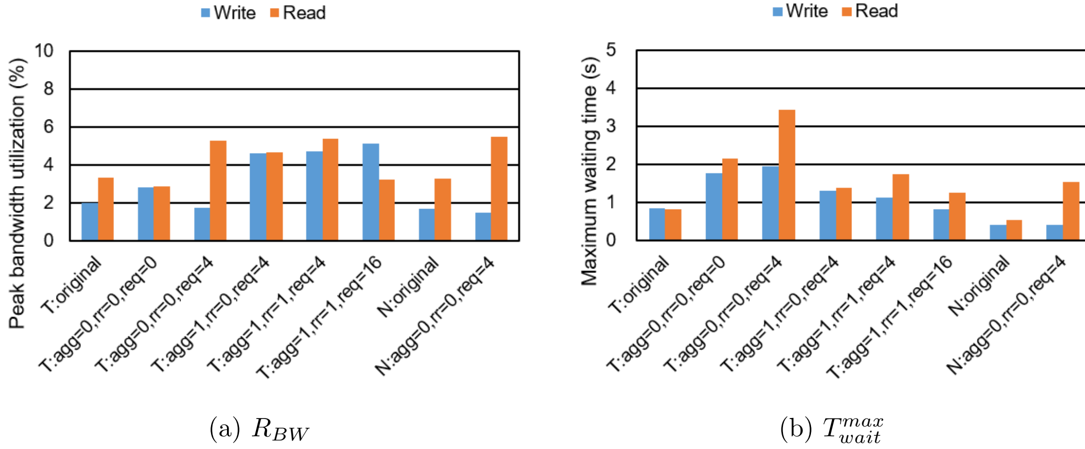
6.5 Load balancing in I/O throughput at OSTs

Figure 6.6 shows write throughput heat-maps ranging from 0 to 160 MiB/s among the 192 OSTs used during the IOR benchmark runs. Horizontal and vertical axes ranging from 0 to 15 and from 0 to 11, indicate subjected relative 2D positions of OSTs used from the logical 3D layout of the K computer.

In the original MPI-IO use case in Figure 6.6a, we can see performance gaps among the left and right sides separated by the dotted line. Figure 6.6b also shows performance gaps among OSTs used because of imbalanced aggregator layout although the EARTH was used. Both cases were not suitable configurations because total I/O performance was limited by the slowest OSTs in parallel I/O. Meanwhile, the most optimized case in Figure 6.6c shows a well-balanced situation in write throughput among OSTs used. Within the context of parallel I/O characteristics, this case was suitable to achieve I/O performance for the benchmark run.

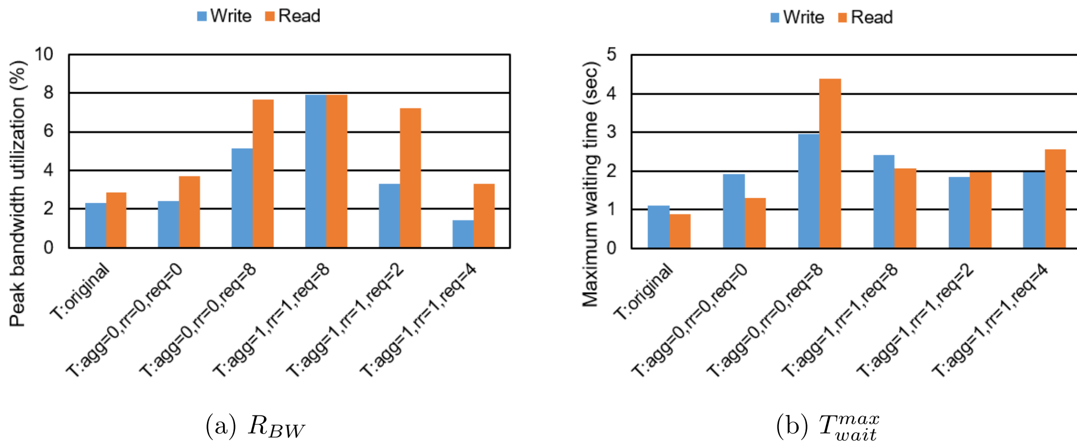
On the other hand, Figure 6.6d and Figure 6.6e show poor I/O throughput in collective write operations in the original and the EARTH use cases without two-phase I/O, respectively. These poor performance situations were due to contention in I/O task assignment to I/O threads by huge number of concurrent I/O accesses from all 12,288 processes as we observed in Figure 6.2a and Figure 6.2b.

Figure 6.7 shows read throughput heat-maps ranging from 0 to 160 MiB/s among the 192 OSTs used at the IOR benchmark runs. Imbalanced bandwidth situations in the original



(a) R_{BW} (b) T_{wait}^{max}

Figure 6.4: Mean values for (a) R_{BW} and (b) T_{wait}^{max} on the Tofu interconnects among I/O nodes used during the IOR benchmark run.

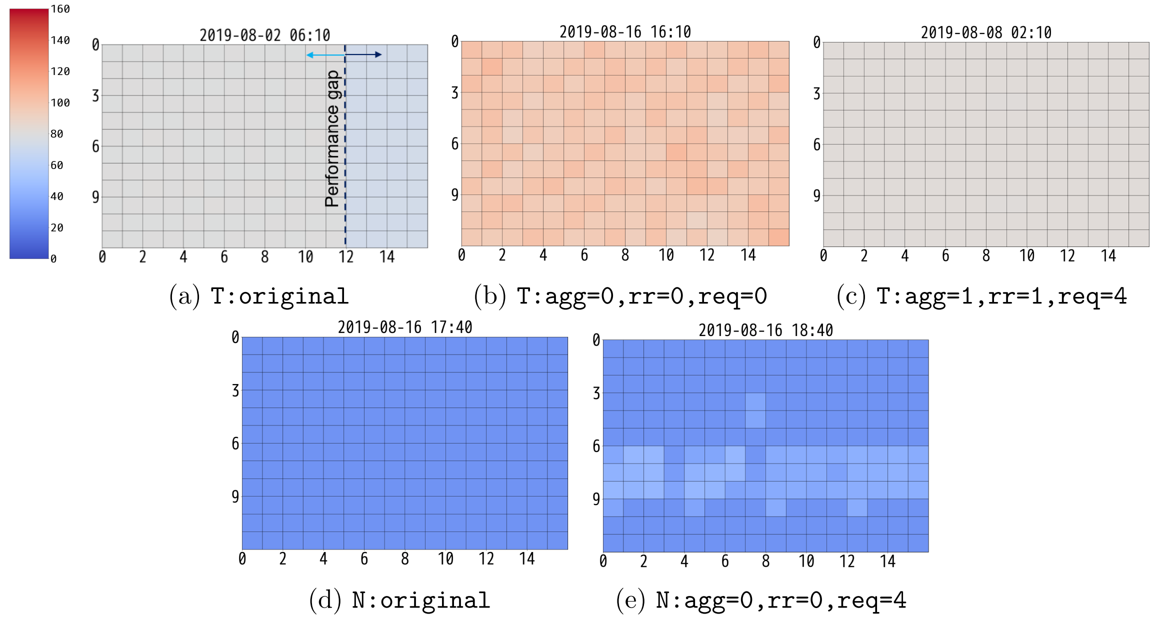


(a) R_{BW} (b) T_{wait}^{max}

Figure 6.5: Mean values for (a) R_{BW} and (b) T_{wait}^{max} on the Tofu interconnects among I/O nodes used during the HPIO benchmark run.

MPI-IO use case and the EARTH use case without any optimizations were observed in Figure 6.7a and Figure 6.7b, respectively. Meanwhile, well-balanced situations were achieved in the EARTH use case with an optimal optimization configuration, which led to high performance collective I/O, as shown in Figure 6.7c.

Write and read throughput heat-maps ranging from 0 to 160 MiB/s at the HPIO run are also shown in Figure 6.8 and Figure 6.9, respectively. In Figure 6.9, the EARTH use case with insufficient configuration (`T:agg=0,rr=0,req=8`) showed lower performance compared with the original MPI-IO use case. Meanwhile, a full set of the three optimizations in the EARTH use case (`T:agg=1,rr=1,req=8`) achieved the highest I/O throughput at every OST used. Read throughput heat-maps in Figure 6.9 show the highest I/O throughput in the insufficient optimization configuration (`T:agg=0,rr=0,req=8`), followed by the original MPI-IO use case and the full optimization configuration case. However, the insufficient configuration case performed the longest waiting time in the Tofu interconnects among the I/O nodes used, as shown in Figure 6.5b, and thus high I/O bandwidth could not be achieved in this configuration.



(a) `T:original` (b) `T:agg=0,rr=0,req=0` (c) `T:agg=1,rr=1,req=4`

(d) `N:original` (e) `N:agg=0,rr=0,req=4`

Figure 6.6: Write throughput heat-maps ranging from 0 to 160 MiB/s about the 192 OSTs used during the IOR benchmark run.

Optimization	`I/O stats`			`Tofu stats`		`I/O rates`	Overall
configuration	`req_qdepth`	`req_waittime`	`req_active`	R_{BW}	T_{wait}^{max}	OST_{mean}	score
T:original	7	7	1	7	2	5	4.83
T:agg=0,rr=0,req=0	5	4	4	6	7	1	4.50
T:agg=0,rr=0,req=4	1	1	8	4	8	7	4.83
T:agg=1,rr=0,req=4	2	2	5	2	5	6	3.67
T:agg=1,rr=1,req=4	3	3	3	1	6	4	3.33
T:agg=1,rr=1,req=16	6	5	2	3	4	2	3.67
N:original	8	8	6	8	1	8	6.50
N:agg=0,rr=0,req=4	4	6	7	5	3	3	4.67

Table 6.1: Scores of the IOR benchmark run, where lesser is better in each score number.

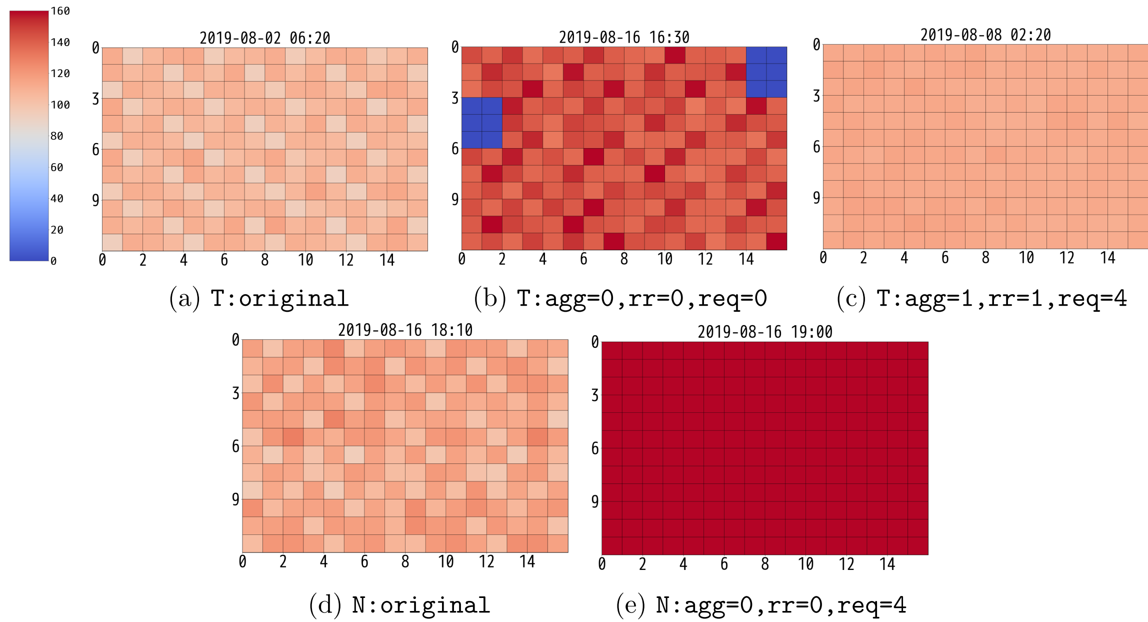
6.6 Overall evaluation

We conducted the overall evaluation based on the abovementioned results in each target metric. From the results in write and read operations in each benchmark run, we obtained mean values of the following metrics:

- Three metrics of `I/O stats`: (`req_qdepth`, `req_waittime`, and `req_active`)
- Two metrics of `Tofu stats`: (R_{BW} and T_{wait}^{max})
- Mean OST I/O bandwidth from `I/O rates` (OST_{mean})

It is preferable to have low values in the two of the three metrics in `I/O stats`, `req_qdepth` and `req_waittime`. While having high value is preferable in the `req_active`. Concerning the two metrics in the `Tofu stats`, high value is desirable in R_{BW}, while low value is suitable in T_{wait}^{max}. High value is preferable in OST_{mean} from `I/O rates`. We gave them ranks from 1 in the order from the best one among the evaluated optimizations in each metric according to the above context. Finally we obtained a stats score as a mean value of the ranks.

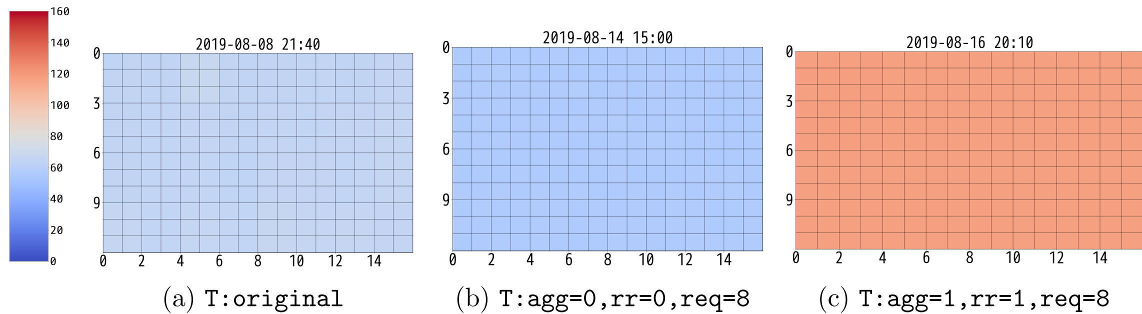
Table 6.1 summarizes the scores of the IOR benchmark run. We can see that the case of `T:agg=1,rr=1,req=4` shows the best overall score (3.33) among the evaluated optimization parameter sets. Although we already examined that this case was the best in the IOR run, we observed another insight that the bast case achieved such balanced situation among I/O subsystems from the score.



(a) `T:original` (b) `T:agg=0,rr=0,req=0` (c) `T:agg=1,rr=1,req=4`

(d) `N:original` (e) `N:agg=0,rr=0,req=4`

Figure 6.7: Read throughput heat-maps ranging from 0 to 160 MiB/s about the 192 OSTs used during the IOR benchmark run.



(a) `T:original` (b) `T:agg=0,rr=0,req=8` (c) `T:agg=1,rr=1,req=8`

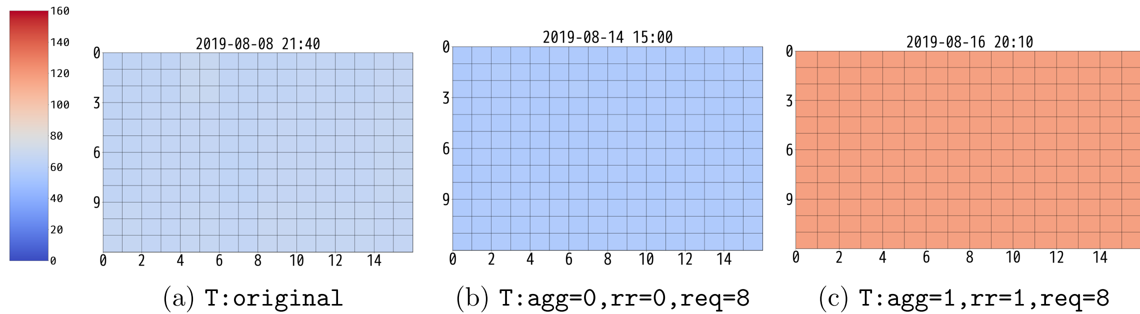
Figure 6.8: Write throughput heat-maps ranging from 0 to 160 MiB/s about the 192 OSTs used at the HPIO benchmark run.

Meanwhile, the scores of the HPIO benchmark run are summarized in Table 6.2. From this table, we can see that the case of `T:agg=1,rr=1,req=8` achieves the best score (2.67) among the evaluated optimization parameter configurations. The best case showed balanced situation among the I/O subsystems as well as the IOR benchmark run. We can easily observe the best optimization configuration with the balanced situation using the scoring scheme.

7 Conclusions

We built a holistic log data analysis framework to characterize I/O activities at the LFS and data transfers through the Tofu interconnects of I/O nodes in I/O optimization at the K computer. The proposed framework utilized the bandwidth status of the Tofu links among I/O nodes used and performance metrics of log data generated at the LFS and I/O nodes. The holistic analysis of data transfer activities on the Tofu links among I/O nodes and I/O activities on the LFS provided useful information in I/O performance tuning.

The two I/O benchmark runs showed notable differences in I/O activities at the LFS and data transfers through the Tofu links among I/O nodes between the original MPI-IO and the enhanced one named EARTH. The EARTH with the optimal optimization configuration showed a high number of active threads on OSSes with short waiting times in I/O request



(a) T:original (b) T:agg=0,rr=0,req=8 (c) T:agg=1,rr=1,req=8

Figure 6.9: Read throughput heat-maps ranging from 0 to 160 MiB/s about the 192 OSTs used at the HPIO benchmark run.

Optimization configuration	I/O stats req_qdepth	req_waittime	req_active	Tofu stats R_{BW}	T_{wait}^{max}	I/O rates OST_{mean}	Overall score
T:original	5	6	2	5	1	5	4.00
T:agg=0,rr=0,req=0	4	5	4	4	2	6	4.17
T:agg=0,rr=0,req=8	1	1	6	2	6	1	2.83
T:agg=1,rr=1,req=8	3	4	1	1	4	3	2.67
T:agg=1,rr=1,req=2	2	2	5	3	3	4	3.17
T:agg=1,rr=1,req=4	6	3	3	6	5	2	4.17

Table 6.2: Scores of the HPIO benchmark run, where lesser is better in each score number.

operations in comparison with the original MPI-IO. The EARTH case also showed high scores in bandwidth utilization of the Tofu links and waiting times for data transfers on the Tofu links in addition to high I/O bandwidth on OSTs. Such obtained profiling information provided insights to understand why the EARTH gained I/O performance relative to the original MPI-IO. We had an unknown issue in performance gaps among different optimization configurations of the EARTH. The framework also informed us how much the impact in I/O activities at the LFS and bandwidth utilization of the Tofu links of I/O nodes among several optimization configurations of the EARTH not only individual examinations in the three log data collections but also overall scoring scheme. By using the framework, we obtained the same answer about the optimal optimization configuration in the two I/O benchmark runs compared with the I/O bandwidth values obtained only from benchmark runs. Compared with traditional evaluation only using I/O benchmarks, our framework can provide more insights about the I/O activities in each I/O subsystem such as high speed interconnects and activities on the target file systems.

Our future work is building a similar framework in Fugaku, with more sophisticated organization of the database to cover all essential metrics from collected log data with a fine-grained monitoring interval. Although the system configuration of Fugaku is different from the K computer, the enhanced Tofu interconnects called TofuD [AKO+18] in Fugaku supports the same metrics used in the proposed framework. This means that we can monitor Tofu data transfer packet status through TNRs of TofuD. Unfortunately, we did not have any chance to investigate real application jobs with the proposed framework in the K computer because the implementation and the evaluation were done as trial only for the last few months before the K computer termination. Such evaluation would be our future work if we have chance to deploy the similar framework in Fugaku. The proposed analysis framework with some enhancements for Fugaku is expected to be useful for I/O performance tuning by monitoring I/O workloads of I/O nodes and file systems and data transfers on TofuD interconnects. According to some confidential issues with vendors, we cannot describe anything about the framework in Fugaku.

We did not have any enhanced works about the proposed framework in other HPC platforms unfortunately. However, we consider that the framework can be easily enhanced in other

HPC platforms since the framework was built by open source environment such as *Python*. Although the proposed framework was partially lack of generality by using logs of Tofu and FEFS, which were specific subsystems in Fujitsu's machine, we can enhance it by having an abstract layer on top of an underlying system-dependent layer. Other interconnects such as Gemini [ARK10, PVB+13] or Aries [Arib, Aria] from Cray also provide the similar hardware counters, and they have been used in log analysis studies [CJH+19, AAB+18, ZGL16, PVB+13]. Such abstract layer will cover all the system-dependent layer to provide metrics about data transfers on interconnects. Besides, metrics extracted from FEFS were ones available in Lustre because FEFS was an enhanced file system based on Lustre. Therefore it would be easy to utilize the same metrics in other HPC platforms equipped with Lustre file systems. Within this context, enhancements on other HPC platforms would be another challenge.

Acknowledgment

This research used computational resources of the K computer provided by the RIKEN Center for Computational Science.

We thank the reviewers Lingfang Zeng, Suren Byna, Anthony Kougkas, and George Markomanolis for their constructive feedback.

References

[AAB+18] Ville Ahlgren, Stefan Andersson, Jim Brandt, Nicholas P. Cardo, Sudheer Chunduri, Jeremy Enos, Parks Fields, Ann Gentile, Richard Gerber, Joe Greenseid, Annette Greiner, Bilel Hadri, Yun (Helen) He, Dennis Hoppe, Urpo Kaila, Kaki Kelly, Mark Klein, Alex Kristiansen, Steve Leak, Mike Mason, Kevin Pedretti, Jean-Guillaume Piccinali, Jason Repik, Jim Rogers, Susanna Salminen, Mike Showerman, Cary Whitney, and Jim Williams. Cray system monitoring: Successes, requirements, and priorities. In *2018 Cray User Group Meeting (CUG)*, 2018.

[AIH+12] Yuichiro Ajima, Tomohiro Inoue, Shinya Hiramoto, Yuzo Takagi, and Toshiyuki Shimizu. The Tofu interconnect. *IEEE Micro*, 32(1):21–31, 2012.

[AKO+18] Yuichiro Ajima, Takahiro Kawashima, Takayuki Okamoto, Naoyuki Shida, Kouichi Hirai, Toshiyuki Shimizu, Shinya Hiramoto, Yoshiro Ikeda, Takahide Yoshikawa, Kenji Uchida, and Tomohiro Inoue. The Tofu Interconnect D. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 646–654. IEEE, 2018.

[Aria] Aries Hardware Counters. https://pubs.cray.com/bundle/Aries_Hardware_Counters_S-0045-40/page/Aries_Hardware_Counters.html.

[Arib] Aries Network Overview. https://pubs.cray.com/bundle/Aries_Hardware_Counters_S-0045-40/page/Aries_Hardware_Counters.html.

[ARK10] Robert Alverson, Duncan Roweth, and Larry Kaplan. The gemini system interconnect. In *2010 18th IEEE Symposium on High Performance Interconnects*, pages 83–87, 2010.

[BBR+16] Wahid Bhimji, Debbie Bard, Melissa Romanus, David Paul, Andrey Ovsyannikov, Brian Friesen, Matt Bryson, Joaquin Correa, Glenn K. Lockwood, Vakho Tsulaia, Suren Byna, Steve Farrell, Doga Gursoyz, Chris Daley, Vince Beckner, Brian Van Straalen, David Trebotich, Craig Tull, Gunther Weber, Nicholas J.

Wright, Katie Antypas, and Prabhat. Accelerating science with the nersc burst buffer early user program. In *2016 Cray User Group Meeting (CUG)*, 2016.

[BLH+13] Babak Behzad, Huong Vu Thanh Luu, Joseph Huchette, Surendra Byna, Prabhat, Ruth Aydt, Quincey Koziol, and Marc Snir. Taming parallel I/O complexity with auto-tuning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC'13. ACM, 2013.

[CCkL+06] Avery Ching, Alok Choudhary, Wei keng Liao, Lee Ward, and Neil Pundit. Evaluating I/O characteristics and methods for storing structured scientific data. In *Proceedings 20th IEEE International Parallel and Distributed Processing Symposium*, page 49. IEEE Computer Society, April 2006.

[CJH+19] Sudheer Chunduri, Elise Jennings, Kevin Harms, Christopher Knight, and Scott Parker. A generalized statistics-based model for predicting network-induced variability. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 59–72. IEEE Computer Society, nov 2019.

[DAR] DARSHAN. http://www.mcs.anl.gov/research/projects/darshan/.

[flu] fluentd. https://www.fluentd.org/.

[IOIkM12] Keiichi Ida, Yasuyuki Ohno, Shunsuke Inoue, and kazuo Minami. Performance profiling and debugging on the K computer. *Fujitsu Sci. Tech. J.*, 48(3):331–339, July 2012.

[IOR] IOR. https://github.com/hpc/ior.

[KGP+18] Mohit Kumar, Saurabh Gupta, Tirthak Patel, Michael Wilder, Weisong Shi, Song Fu, Christian Engelmann, and Devesh Tiwari. Understanding and analyzing interconnect errors and network congestion on a large scale HPC system. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN'18, pages 107–114. IEEE, June 2018.

[KZH+14] Julian M. Kunkel, Michaela Zimmer, Nathanael Hübbe, Alvaro Aguilera, Holger Mickler, Xuan Wang, Andriy Chut, Thomas Bönisch, Jakob Lüttgau, Roman Michel, and Johann Weging. The SIOX architecture – coupling automatic monitoring and optimization of parallel I/O. In *Supercomputing, 29th International Conference, ISC2014, Leipzig, Germany, June 22-26, Proceedings*, pages 245–260. Springer, 2014.

[LGMV14] Yang Liu, Raghul Gunasekaran, Xiaosong Ma, and Sudharshan S. Vazhkudai. Automatic identification of application I/O signatures from noisy server-side traces. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 2014)*, pages 213–228. USENIX, 2014.

[LGMV16] Yang Liu, Raghul Gunasekaran, Xiaosong Ma, and Sudharshan S. Vazhkudai. Server-side log data analytics for I/O workload characterization and coordination on large shared storage systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'16. ACM, 2016.

[LMT] Lustre Monitoring Tool. https://github.com/LLNL/lmt.

[Lus] Lustre. https://www.lustre.org/.

[Lus08] Lustre. Lustre ADIO collective write driver. Technical report, Lustre, September 2008.

[LWG+15] Huong Luu, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Mr Prabhat, Suren Byna, and Yushu Yao. A multiplatform study of I/O behavior on petascale supercomputers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 33–44. ACM, 2015.

[LWS+18] Glenn K. Lockwood, Nicholas J. Wright, Shane Snyder, Philip Carns, George Brown, and Kevin Harms. Tokio on clusterstor: Connecting standard tools to enable holistic I/O performance analysis. In *2018 Cray User Group Meeting (CUG)*, 2018.

[MBC+17] S. Madireddy, P. Balaprakash, P. Carns, R. Latham, R. Ross, S. Snyder, and S. M. Wild. Analysis and correlation of application I/O performance and system-wide I/O activity. In *Proceedings of the 2017 International Conference on Networking, Architecture, and Storage (NAS)*, pages 1–10. IEEE, 2017.

[MPI] MPI Forum. https://www.mpi-forum.org/.

[OVW+19] Sarp Oral, Sudharshan S. Vazhkudai, Feiyi Wang, Christopher Zimmer, Christopher Brumgard, Jesse Hanley, George Markomanolis, Ross Miller, Dustin Leverman, Scott Atchley, and Veronica Vergara Larrea. End-to-end I/O portfolio for the summit supercomputing ecosystem. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'19. ACM, 2019.

[PBLT19] Tirthak Patel, Suren Byna, Glenn K. Lockwood, and Devesh Tiwari. Revisiting I/O behavior in large-scale storage systems: The expected and the unexpected. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'19, pages 65:1–65:13. ACM, 2019.

[PVB+13] Kevin Pedretti, Courtenay Vaughan, Richard Barrett, Karen Devine, and K. Scott Hemmert. Using the cray gemini performance counters. In *2013 Cray User Group Meeting (CUG)*, 2013.

[SH02] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02. USENIX Association, 2002.

[SRC+12] S. Saini, J. Rappleye, J. Chang, D. Barker, P. Mehrotra, and R. Biswas. I/O performance characterization of Lustre and NASA applications on Pleiades. In *19th International Conference on High Performance Computing (HiPC)*, pages 1–10, 2012.

[SSK12] Kenichiro Sakai, Shinji Sumimoto, and Motoyoshi Kurokawa. High-performance and highly reliable file system for the K computer. *Fujitsu Sci. Tech. J.*, 48(3):302–309, 2012.

[Sum11] Shinji Sumimoto. An overview of Fujitsu's Lustre based file system. In *Lustre User Group 2011*, 2011.

[TFH+20] Yuichi Tsujita, Yoshitaka Furutani, Hajime Hida, Keiji Yamamoto, and Atsuya Uno. Characterizing I/O optimization effect through holistic log data analysis of parallel file systems and interconnects. In *High Performance Computing - ISC*

High Performance 2020 International Workshops, Frankfurt/Main, Germany, June 21-25, 2020, Revised Selected Papers, volume 12321 of *Lecture Notes in Computer Science*, pages 177–190. Springer, 2020.

[TGL99] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, 1999.

[The] The supercomputer Fugaku. https://www.r-ccs.riken.jp/en/fugaku/.

[THI14] Yuichi Tsujita, Atsushi Hori, and Yutaka Ishikawa. Locality-aware process mapping for high performance collective MPI-IO on FEFS with Tofu interconnect. In *Proceedings of the 21th European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 157:157–157:162. ACM, 2014. Challenges in Data-Centric Computing.

[THK+18] Yuichi Tsujita, Atsushi Hori, Toyohisa Kameyama, Atsuya Uno, Fumiyoshi Shoji, and Yutaka Ishikawa. Improving collective MPI-IO using topology-aware stepwise data aggregation with I/O throttling. In *Proceedings of HPC Asia 2018: International Conference on High Performance Computing in Asia-Pacific Region, January 28-31, 2018*, pages 12–23. ACM, 2018.

[TMV+16] François Tessier, Preeti Malakar, Venkatram Vishwanath, Emmanuel Jeannot, and Florin Isaila. Topology-aware data aggregation for intensive I/O on large-scale supercomputers. In *Proceedings of the First Workshop on Optimization of Communication in HPC*, COM-HPC'16, page 73–81. IEEE Press, 2016.

[UW13] Andrew Uselton and Nicholas Wright. A file system utilization metric for I/O characterization. In *2013 Cray User Group Meeting*, 2013.

[VdSB+18] Sudharshan S. Vazhkudai, Bronis R. de Supinski, Arthur S. Bland, Al Geist, James Sexton, Jim Kahle, Christopher J. Zimmer, Scott Atchley, Sarp Oral, Don E. Maxwell, Veronica G. Vergara Larrea, Adam Bertsch, Robin Goldstone, Wayne Joubert, Chris Chambreau, David Appelhans, Robert Blackmore, Ben Casses, George Chochia, Gene Davison, Matthew A. Ezell, Tom Gooding, Elsa Gonsiorowski, Leopold Grinberg, Bill Hanson, Bill Hartner, Ian Karlin, Matthew L. Leininger, Dustin Leverman, Chris Marroquin, Adam Moody, Martin Ohmacht, Ramesh Pankajakshan, Fernando Pizzano, James H. Rogers, Bryan Rosenburg, Drew Schmidt, Mallikarjun Shankar, Feiyi Wang, Py Watson, Bob Walkup, Lance D. Weems, and Junqi Yin. The design, deployment, and evaluation of the coral pre-exascale systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC'18. IEEE Press, 2018.

[WSL+18] Teng Wang, Shane Snyder, Glenn Lockwood, Philip Carns, Nicholas Wright, and Suren Byna. IOMiner: Large-scale analytics framework for gaining knowledge from I/O logs. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 466–476, 2018.

[XBV+16] Cong Xu, Suren Byna, Vishwanath Venkatesan, Rober Sisneros, Omkar Kulkarni, Mohamad Chaarawi, and Kalyana Chadalavada. LIOProf: Exposing lustre file system behavior for I/O middleware. In *2016 Cray User Group Meeting*, May 2016.

[XCD+12] B. Xie, J. Chase, D. Dillow, O. Drokin, S. Klasky, S. Oral, and N. Podhorszki. Characterizing output bottlenecks in a supercomputer. In *Proceedings of 2012*

International Conference for High Performance Computing, Networking, Storage and Analysis, SC'12, pages 1–11. IEEE, 2012.

[YJM+19] Bin Yang, Xu Ji, Xiaosong Ma, Xiyang Wang, Tianyu Zhang, Xiupeng Zhu, Nosayba El-Sayed, Haidong Lan, Yibo Yang, Jidong Zhai, Weiguo Liu, and Wei Xue. End-to-end I/O monitoring on a leading supercomputer. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'19, pages 379–394. USENIX, 2019.

[ZGL16] Christopher Zimmer, Saurabh Gupta, and Ver´onica G. Vergara Larrea. Finally, a way to measure frontend I/O performance. In *2016 Cray User Group Meeting (CUG)*, 2016.

Reviews

This section is optional for reviewers and shows their assessment that lead to the acceptance of the original manuscript. Reviewers may or may not update their review for a major update of the paper, the exact trail is available in GitHub repository of this article. The reviews are part of the article and validate the acceptance. Please check the details on the JHPS webpage.

Reviewer: Lingfang Zeng, Date: 2021-04-16

Overall summary and proposal for acceptance What makes the reviewer deeply influenced in this manuscript is that the experiment is very rich. The main shortcomings of the manuscript are: (1) lack of discussion on the optimization scheme and technical details (which also leads to the lack of support/explanation in the Section "Experimental Evaluation": what are the specific measures / improvements to achieve the performance optimization?) (2) The technical schemes and experimental platform are out of date and lack of novelty. (3) The unique software and hardware scheme makes the technical schemes of this draft unrepeatable, which is not in line with the purpose of JHPS. It is suggested that the draft be rejected. Others: (1) Page 4, Section 3 "K computer and Its File System Monitoring" – The system was retired (about two years). In the era of Exascale Supercomputers and intelligent computing, always new architecture and technology can attract readers. (2) Page 8, Section 5 "Enhanced MPI-IO Implementation: EARTH on K", "Its advanced functions are summarized in the following three key optimization parameters described by agg, req, and rr, respectively:" – This is the main contribution of this draft, but the details are too few. Although the reviewer understands the relevant technical details, the readers may not be sure. On the whole, the contribution of this draft is too little, lack of novelty. (3) Page 10, Section 6 "Experimental Evaluation", Subsection 6.1 "Benchmark configuration" – K computer has been running for many years. It has run many typical applications and must have collected a lot of real log information. Therefore, I suggest using real application log and system log information instead of being generated by benchmark.

Scope Yes. Its topic fits the JHPS.

Significance Minor.

Readability Yes.

Presentation It's clear and easy to understand.

References Yes.

Correctness There are some sound.

Reviewer: Suren Byna, Date: 2021-05-09

Overall summary In contrast to existing I/O performance analysis efforts, this paper collects interconnection information and detailed parallel file system information on I/O nodes. This additional information allows authors to analyze I/O performance using OSS stats files and interconnect bandwidth.

Some of my concerns were related to the overhead for tracing and storing the detailed information, the impact of other concurrent jobs on the system affecting the bandwidth, and a comparison with existing logs.

It seems that the Tofu and I/O stats have been extracted periodically and stored in a separate database. It was unclear to me if that periodic access cost anything? How large was the stats database?

In the analysis shown (OSS stats, bandwidth utilization, and load balance of I/O to OSTs), were there other jobs running concurrently along with the IOR and HPIO jobs that are being studied? What was the impact of them?

Since the results shown here are mainly for a decommissioned system, is the stats collection and storing continuing on current production systems? What was the impact of this study? While various plots were shown for the same configurations across the evaluation section, how do they (bandwidth utilization, waiting time in data transfer, OSS stats, etc.) correlate with the I/O throughput?

The last point regarding comparison with existing logs, such as Darshan and Darshan's extended tracing (DXT), could we get the information needed for the analysis shown in this paper? If not, what information is unavailable? It would be good to describe and compare.

Scope Yes. Its topic fits the JHPS.

Significance Minor.

Readability Yes.

Presentation It's easy to read the paper; I'd suggest some reorganizing in the introduction to motivate the problem before jumping into the Tofu PA log collection and information.

References Yes.

Correctness The exploration and evaluation are correct.

Reviewer: Anthony Kougkas, Date: 2021-05-16

Overall summary The paper presents a holistic log data analysis framework to characterize I/O activities at the LFS and data transfers through the Tofu interconnects of I/O nodes in I/O optimization at the K computer. Further, the paper presents a comparison between vanilla MPI-IO and EARTH, an optimized one, and how the obtained profiling information gave insights to understand why the EARTH demonstrated higher I/O performance relative to the original MPI-IO. The analysis framework allowed the authors to uncover how much I/O activities at the LFS and bandwidth utilization of the Tofu links of I/O nodes impacted the performance of EARTH. Extensive results were presented and analyzed.

I enjoyed reading about this work. The paper is informative on issues stemmed by poor monitoring and analysis of I/O activities in a large computing environment. This reviewer found the paper focusing on the K computer a bit restricting in drawing conclusions in general. A lot of the proposed framework is specific to the architecture (e.g., LIO, GIO, BIO are all particulars of the K computer) and I am not convinced that the audience can extract generally useful methodologies on I/O tracking. However, the paper reads well, has a decent motivation section, and the extensive results presented are only positive. I am recommending a minor revision with the following three suggestions: a) the authors should invest some time bringing the manuscript in published quality through detailed proofreading. b) add a subsection (or a clear paragraph somewhere early in the paper) discussing in detail the intellectual contributions. c) add a discussions and considerations section (before the evaluation maybe) where the authors can connect their methodology in a more general architecture (i.e., how could we achieve the same depth of collected information for another HPC machine? What

parts of this work are specific to K computer and what are not? Is the framework developed open sourced? What are the caveats of the proposed analysis?)

Strengths

- The paper addresses issues in an area (i.e., I/O activity monitoring) that needs more investigation by the community.
- The paper has done a great work presenting the methodology the authors followed and described the architecture of K computer in great detail making it easy to follow.
- The paper went to great lengths to present detailed performance analysis of MPI-IO and EARTH.

Weaknesses

- The paper is more of an experience paper. There is nothing general in the findings but solely focuses on the K computer. Even though this was a production machine, it does not represent all HPC architectures. There is little (or no) discussion as to how the reader can replicate their work for a general architecture.
- The paper needs a good proofreading, possibly multiple passes. While it is not poorly written, there are many areas where grammar and syntax can be polished

Reviewer: George Markomanolis, Date: 20210-05-17

Overall summary The authors present their work, an I/O performance optimization framework that uses log data of parallel file systems and interconnects in a holistic way. They discuss the Tofu PA profiling tool on the K computer for the Tofu interconnects. They present how all the framework works and its usefulness for specific cases. One main disadvantage of this paper is that this is a work for a decommissioned system, the K supercomputer. The collection of statistics every 1 or 10 minutes depending on the type of the stats can not correlate with every I/O intensive application. The installed ROMIO version on the K computer is not optimized for the Lustre, thus a newer version was used and the comparison between these two is not fair. There are presented some results from their approach and their metrics are explained. They do achieve better I/O performance with their tuning. The manuscript is more experimental, sometimes tuning parameters becomes more known case but the holistic approach on such system is not a common topic. I would like to see more explanations about the duration of the logs collection as collecting every a few minutes is a really specific I/O pattern. I assume it is not possible to have more diverse applications as the K computer is not available anymore. I would also like some discussion regarding overhead from your tool.

Scope Yes. It fits.

Significance Minor.

Readability Yes but it could be improved

Presentation It is good but a proofreading would help.

References Yes.

Correctness Yes, although the fact that the research is not reproducible because of the decommissioned system is always an issue.

A Workflow for Identifying Jobs with Similar I/O Behavior Utilizing Time Series Analysis

Reviews

Julian Kunkel
Georg-August-Universität Göttingen
Göttingen, Germany

Eugen Betke
ECMWF
Reading, UK
✉ *eugen.betke@ecmwf.int*

Abstract

One goal of support staff at a data center is to identify inefficient jobs and to improve their efficiency. Therefore, a data center deploys monitoring systems that capture the behavior of the executed jobs. While it is easy to utilize statistics to rank jobs based on the utilization of computing, storage, and network, it is tricky to find patterns in 100,000 jobs, i.e., is there a class of jobs that aren't performing well. Similarly, when support staff investigates a specific job in detail, e.g., because it is inefficient or highly efficient, it is relevant to identify related jobs to such a blueprint. This allows staff to understand the usage of the exhibited behavior better and to assess the optimization potential.

In this article, our goal is to identify jobs similar to an arbitrary reference job. In particular, we describe a methodology that utilizes temporal I/O similarity to identify jobs related to the reference job. Practically, we apply several previously developed time series algorithms and also utilize the Kolmogorov-Smirnov-Test to compare the distribution of the metrics. A study is conducted to explore the effectiveness of the approach by investigating related jobs for three reference jobs. The data stem from DKRZ's supercomputer Mistral and include more than 500,000 jobs that have been executed for more than 6 months of operation. Our analysis shows that the strategy and algorithms are effective to identify similar jobs and reveal interesting patterns in the data. It also shows the need for the community to jointly define the semantics of similarity depending on the analysis purpose.

Keywords: performance analysis, monitoring, time series, job analysis

1 Introduction

Supercomputers execute 1000s of jobs every day. Support staff at a data center have two goals. Firstly, they provide a service to users to enable them the convenient execution of their applications. Secondly, they aim to improve the efficiency of all workflows – represented as batch jobs – in order to allow the data center to serve more workloads.

In order to optimize a single job, its behavior and resource utilization must be monitored and then assessed. Rarely, users will liaise with staff and request a performance analysis and optimization explicitly. Therefore, data centers deploy monitoring systems and staff must pro-actively identify candidates for optimization. Monitoring and analysis tools such as TACC Stats [EBB+14], Grafana [Cha19], and XDMod [SWD+18] provide various statistics and time-series data for job execution.

The support staff should focus on workloads for which optimization is beneficial, for instance, the analysis of a job that is executed once on 20 nodes may not be a good return of investment. By ranking jobs based on their utilization, it is easy to find a job that exhibits extensive usage of computing, network, and I/O resources. However, would it be beneficial to investigate this workload in detail and potentially optimize it? For instance, a pattern that is observed in many jobs bears potential as the blueprint for optimizing one job may be applied to other jobs as well. This is particularly true when running one application with similar inputs, but also different applications may lead to similar behavior. Knowing details about a problematic or interesting job may be transferred to similar jobs. Therefore, it is useful for support staff (or a user) that investigates a resource-hungry job to identify similar jobs that are executed on the supercomputer.

It is non-trivial to identify jobs with similar behavior from the pool of executed jobs. Re-executing the same job will lead to slightly different behavior, a program may be executed with different inputs or using a different configuration (e.g., number of nodes). Job names are defined by users; while a similar name may hint to be a similar workload, finding other applications with the same I/O behavior would not be possible.

In the paper [BK21], we developed several distance measures and algorithms for the clustering of jobs based on the time series and their I/O behavior. These distance measures can be applied to jobs with different runtimes and the number of nodes utilized, but differ in the way they define similarity. They showed that the metrics can be used to cluster jobs, however, it remained unclear if the method can be used by data center staff to explore similar jobs effectively. In this paper, we refine these algorithms slightly, include another algorithm, and apply them to rank jobs based on their temporal similarity to a reference job.

We start by introducing related work in Section 2. In Section 3, we describe briefly the data reduction and the algorithms for similarity analysis. We also utilize the Kolmogorov-Smirnov-Test to illustrate the benefits and drawbacks of the different methods. Then, we perform our study by applying the methodology to three reference jobs with different behavior, therewith, assessing the effectiveness of the approach to identify similar jobs. In Section 5, the reference jobs are introduced and quantitative analysis of the job pool is made based on job similarity. In Section 6, the 100 most similar jobs are investigated in more detail, and selected timelines are presented. The paper is concluded in Section 7.

2 Related Work

Related work can be classified into distance measures, analysis of HPC application performance, inter-comparison of jobs in HPC, and I/O-specific tools.

The ranking of similar jobs performed in this article is related to clustering strategies. Levenshtein (Edit) distance is a widely used distance metric indicating the number of edits needed to convert one string to another [Nav01]. The comparison of the time series using

various metrics has been extensively investigated. In [KS18], an empirical comparison of distance measures for the clustering of multivariate time series is performed. 14 similarity measures are applied to 23 data sets. It shows that no similarity measure produces statistically significant better results than another. However, the Swale scoring model [MP07] produced the most disjoint clusters.

The performance of applications can be analyzed using one of many tracing tools such as Vampir [WBW+17] that record the behavior of an application explicitly or implicitly by collecting information about the resource usage with a monitoring system. Monitoring systems that record statistics about hardware usage are widely deployed in data centers to record system utilization by applications. There are various tools for analyzing the I/O behavior of an application [KBB+19].

For Vampir, a popular tool for trace file analysis, in [WBW+17] the Comparison View is introduced that allows them to manually compare traces of application runs, e.g., to compare optimized with original code. Vampir generally supports the clustering of process timelines of a single job, allowing to focus on relevant code sections and processes when investigating many processes.

Chameleon [BM18] extends ScalaTrace for recording MPI traces but reduces the overhead by clustering processes and collecting information from one representative of each cluster. For the clustering, a signature is created for each process that includes the call-graph. In [HDRFV20], 11 performance metrics including CPU and network are utilized for agglomerative clustering of jobs, showing the general effectiveness of the approach.

In [RÖE+18], a characterization of the NERSC workload is performed based on job scheduler information (profiles). Profiles that include the MPI activities have shown effective to identify the code that is executed [DSB13]. Many approaches for clustering applications operate on profiles for compute, network, and I/O [EVGB15, LLK+20, BKW+20]. For example, Evalix [EVGB15] monitors system statistics (from proc) in 1-minute intervals but for the analysis, they are converted to a profile removing the time dimension, i.e., compute the average CPU, memory, and I/O over the job runtime.

PAS2P [MPW+12] extracts the I/O patterns from application traces and then allows users to manually compare them. In [WKD+18], a heuristic classifier is developed that analyzes the I/O read/write throughput time series to extract the periodicity of the jobs – similar to Fourier analysis. The LASSi tool [TSMS+19] periodically monitors Lustre I/O statistics and computes a ”risk” factor to identify I/O patterns that stress the file system. In contrast to existing work, our approach allows a user to identify similar activities based on the temporal I/O behavior recorded by a data center-wide deployed monitoring system.

3 Methodology

The purpose of the methodology is to allow users and support staff to explore all executed jobs on a supercomputer in order of their similarity to the reference job. Therefore, we first need to define how a job's data is represented, then describe the algorithms used to compute the similarity, and, the methodology to investigate jobs.

3.1 Job Data

On the Mistral supercomputer at DKRZ, the monitoring system [BK20] gathers in ten seconds intervals on all nodes nine I/O metrics for the two Lustre file systems together with general job metadata from the SLURM workload manager. The results are 4D data (time, nodes, metrics, file system) per job. The distance measures should handle jobs of different lengths and node count. In the open-access article [BK21], we discussed a variety of options from 1D job-profiles to data reductions to compare time series data and the general workflow and pre-

processing in detail. We will be using this representation. In a nutshell, for each job executed on Mistral, they partitioned it into 10 minutes segments[1] and compute the arithmetic mean of each metric, categorize the value into NonIO (0), HighIO (1), and CriticalIO (4) for values below 99-percentile, up to 99.9-percentile, and above, respectively. The values are chosen to be 0, 1, and 4 because we arithmetically derive metrics: naturally, the value of 0 will indicate that no I/O issue appears; we weight critical I/O to be 4x as important as high I/O. This strategy ensures that the same approach can be applied to other HPC systems regardless of the actual distribution of these statistics on that data center. After the mean value across nodes is computed for a segment, the resulting numeric value is encoded either using binary (I/O activity on the segment: yes/no) or hexadecimal representation (quantizing the numerical performance value into 0-15) which is then ready for similarity analysis. By pre-filtering jobs with no I/O activity – their sum across all dimensions and time series is equal to zero, the dataset is reduced from 1 million jobs to about 580k jobs.

Binary coding Binary coding is a data reduction technique that represents monitoring data as a sequence of numbers, where each number stands for a specific I/O behavior. It is applied on the hypercube, produced by the previous pre-processing steps (segmentation and categorization). This hypercube has four dimensions (nodes, metrics, file systems and time) and is filled with categories. The data is reduced in two steps. First, reduction of two dimensions (nodes and file systems) and aggregation by the sum() function creates a new two-dimensional data structure (metrics and time) filled with integer values. Second, reduction of a dimension (metrics) and mapping of integer values to a unique number (behavior mapping) creates a one dimensional data structure (time) filled with I/O behavior. Behavior mapping is a function that maps 9 metrics to 9-bit numbers where each bit represents activity of a metric. If the value of a metric is LowIO, then the bit is set to 0 (compute intense state), else (for values HighIO and CriticalIO) it is set to 1 (IO intense state). The following example shows a 15 segment long binary coding:

```
jobA (after coding): [1:5:0:0:0:0:0:0:96:96:96:96:96:96:96], 'length':15
```

Hexadecimal coding Hexadecimal coding is a data reduction technique, that can be applied to the hypercube to obtain further data reduction. After the first two pre-processing steps (segmentation and categorization), a four dimensional hypercube (nodes, metrics, file systems and time) stores performance categories. Hexadecimal coding operates in two steps. First, it aggregates two dimensions (nodes and file systems) by the mean() function. What remains are two dimensions (metrics and time) and the mean values between 0 an 4. Second, the mean values are quantized to 16 levels – 0 = [0,0.25), 1 = [0.25,0.5), . . . , F = [3.75, 4]. The following example shows a five segment long hexadecimal coding:

```
jobB: 'length': 6, 'coding':
  'metric_read' : [0:2:2:2:9],
  'metric_write'   : [0:0:0:0:0],
  ...,
  'metric_md_other': [0:0:0:F:F]
```

3.2 Algorithms for Computing Similarity

We reuse the B and Q algorithms developed in [BK21]: B-all, B-aggz(eros), Q-native, Q-lev, and Q-phases. They differ in the way data similarity is defined; either the time series

[1] We found in preliminary experiments that 10 minutes provide sufficient resolution while reducing compute time and noise, i.e., the variation of the statistics when re-running the same job.

is encoded in binary or hexadecimal quantization, the distance measure is the Euclidean distance or the Levenshtein distance. B-all determines similarity between binary codings by means of Levenshtein distance. B-aggz is similar to B-all, but computes similarity on binary codings where subsequent segments of zero activities are replaced by just one zero. Q-lev determines similarity between quantized codings by using Levenshtein distance. Q-native uses a performance-aware similarity function, i.e., the distance between two jobs for a metric is $\frac{|m_{job1} - m_{job2}|}{16}$. There are various options for how a longer job is embedded in a shorter job, for example, a larger input file may stretch the length of the I/O and compute phases; another option can be that more (model) time is simulated. In this article, we consider these different behavioral patterns and attempt to identify situations where the I/O pattern of a long job is contained in a shorter job. Therefore, for jobs with different lengths, a sliding-windows approach is applied which finds the location for the shorter job in the long job with the highest similarity. Q-phases extracts phase information and performs a phase-aware and performance-aware similarity computation. The Q-phases algorithm extracts I/O phases from our 10-minute segments and computes the similarity between the most similar I/O phases of both jobs. In this paper, we add a similarity definition based on Kolmogorov-Smirnov-Test that compares the probability distribution of the observed values which we describe in the following.

Kolmogorov-Smirnov (KS) algorithm. For the analysis, we perform two preparation steps. Dimension reduction by computing means across the two file systems and by concatenating the time series data of the individual nodes (instead of averaging them). This reduces the four-dimensional dataset to two dimensions (time, metrics). The reduction of the file system dimension by the mean function ensures the time series values stay in the range between 0 and 4, independently of how many file systems are present on an HPC system. Unlike the previous similarity definitions, the concatenation of time series on the node dimension preserves the individual I/O information of all nodes while it still allows comparison of jobs with a different number of nodes.

For the analysis we use the kolmogorov-smirnov-test 1.1.0 Rust library from the official Rust Package Registry "cargo.io". The similarity function calculates the mean inverse of reject probability p_{reject} computed with the ks-test across all metrics M: $sim = \frac{\sum_m 1 - p_{\text{reject}(m)}}{|M|}$.

3.3 Methodology

Our strategy for localizing similar jobs works as follows:

- A user[2] provides a reference job ID and selects a similarity algorithm.
- The system iterates over all jobs of the job pool, computing the similarity to the reference job using the specified algorithm.
- It sorts the jobs based on the similarity to the reference job.
- It visualizes the cumulative job similarity allowing the user to understand how job similarity is distributed.
- The user starts the inspection by looking at the most similar jobs first.

The user can decide about the criterion when to stop inspecting jobs; based on the similarity, the number of investigated jobs, or the distribution of the job similarity. For the latter, it is interesting to investigate clusters of similar jobs, e.g., if there are many jobs between 80-90% similarity but few between 70-80%.

[2]This can be support staff or a data center user that was executing the job.

For the inspection of the jobs, a user may explore the job metadata, search for similarities, and explore the time series of a job's I/O metrics.

4 Reference Jobs

For this study, we chose several reference jobs with different compute and I/O characteristics:

- Job-S: performs post-processing on a single node. This is a typical process in climate science where data products are reformatted and annotated with metadata to a standard representation (so-called CMORization). The post-processing is I/O intensive.
- Job-M: a typical MPI parallel 8-hour compute job on 128 nodes that write time series data after some spin up.
- Job-L: a 66-hour 20-node job. The initialization data is read at the beginning. Then only a single master node writes constantly a small volume of data; in fact, the generated data is too small to be categorized as I/O relevant.

The segmented timelines of the jobs are visualized in Figure 4.1 – remember that the mean value is computed across all nodes on which the job ran. This coding is also used for the Q algorithms, thus this representation is what the algorithms will analyze; B algorithms merge all timelines together as described in [BK21]. The figures show the values of active metrics ($\neq 0$); if few are active, then they are shown in one timeline, otherwise, they are rendered individually to provide a better overview. For example, we can see in Figure 4.1a, that several metrics increase in Segment 6. We can also see an interesting result of our categorized coding, the write bytes are bigger than 0 while write calls are 0[3]. Figure 4.2 summarizes hexadecimal codings of Job-S and Job-M to histograms: A specific histogram contains the metric of each node at every timestep – without being averaged across the nodes. Essentially, this data is used to compare jobs using Kolmogorov-Smirnov-Test. The metrics at Job-L are not shown as they have only a handful of instances where the value is not 0, except for write_bytes: the first process is writing out at a low rate. In Figure 4.1c, the mean value is mostly rounded down to 0 except for the first segment as primarily Rank 0 is doing I/O.

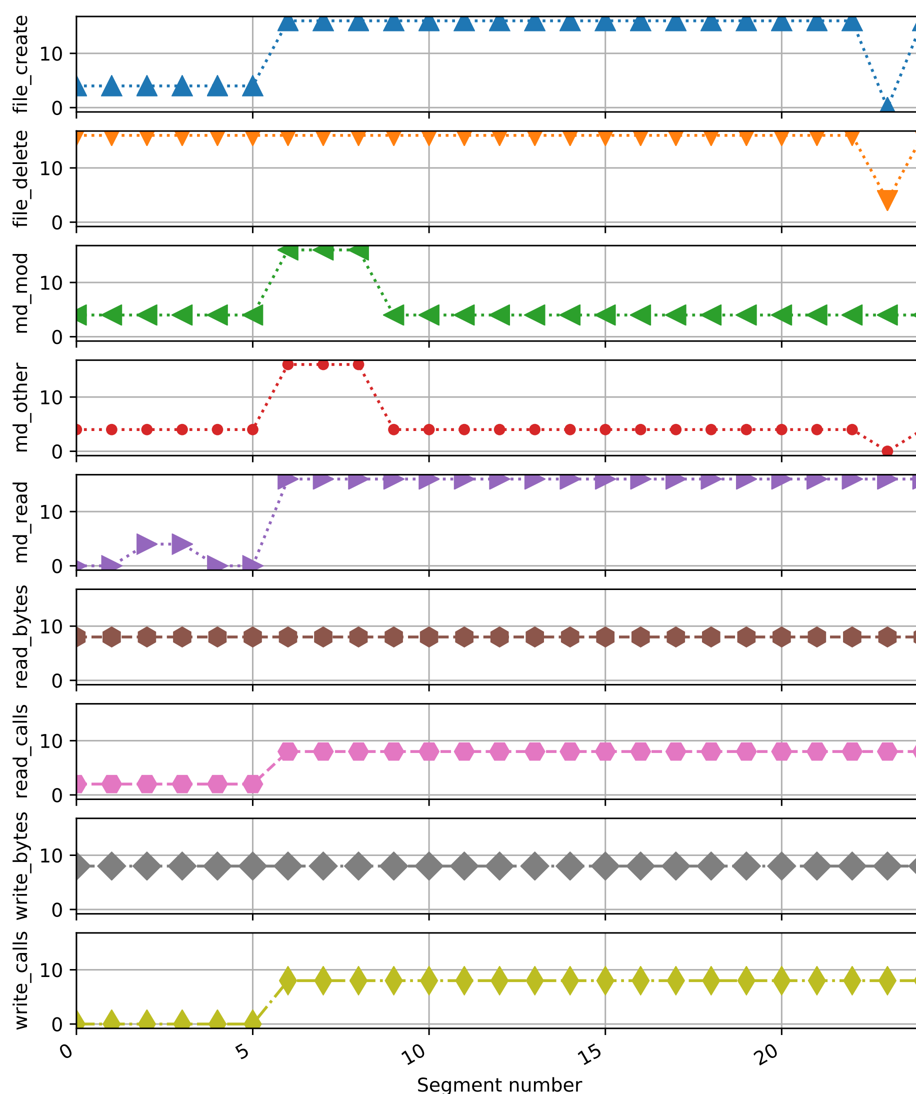
5 Evaluation

In the following, we assume a reference job is given (we use Job-S, Job-M, and Job-L), and we aim to identify similar jobs. For each reference job and algorithm, we created CSV files with the computed similarity to all other jobs from our job pool (worth 203 days of production of Mistral). During this process, the runtime of the algorithm is recorded. Then we inspect the correlation between the similarity and number of found jobs. Finally, the quantitative behavior of the 100 most similar jobs is investigated.

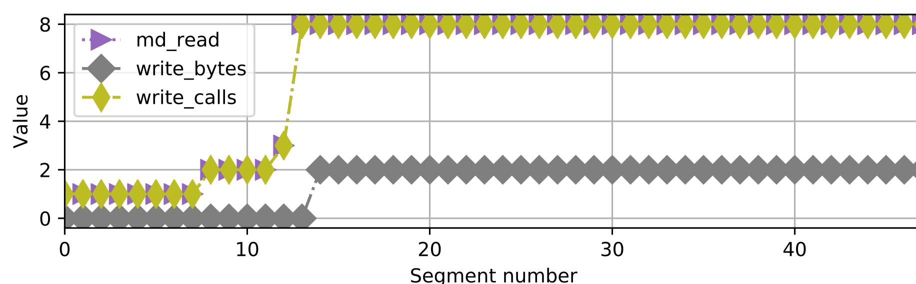
5.1 Performance

To measure the performance for computing the similarity to the reference jobs, the algorithms are executed 10 times on a compute node at DKRZ which is equipped with two Intel Xeon E5-2680v3 @2.50GHz and 64GB DDR4 RAM. A boxplot for the runtimes is shown in Figure 5.1. The runtime is normalized for 100k jobs, i.e., for B-all it takes about 41 s to process 100k jobs out of the 500k total jobs that this algorithm will process. Generally, the B algorithms are fastest, while the Q algorithms often take 4-5x as long. Q_phases is slow for Job-S and

[3] The reason is that a few write calls transfer many bytes; less than our 90%-quantile, therefore, write calls will be set to 0.

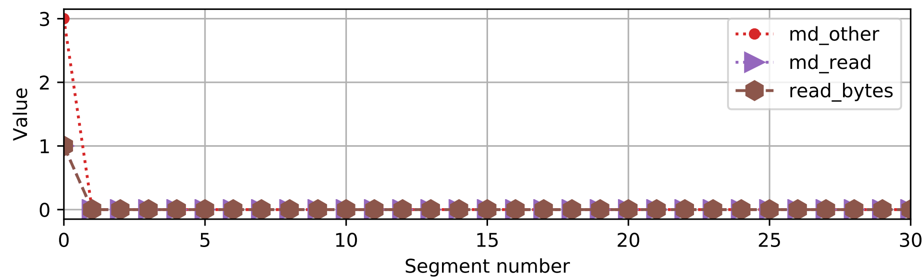


(a) Job-S (runtime=15,551 s, segments=25)



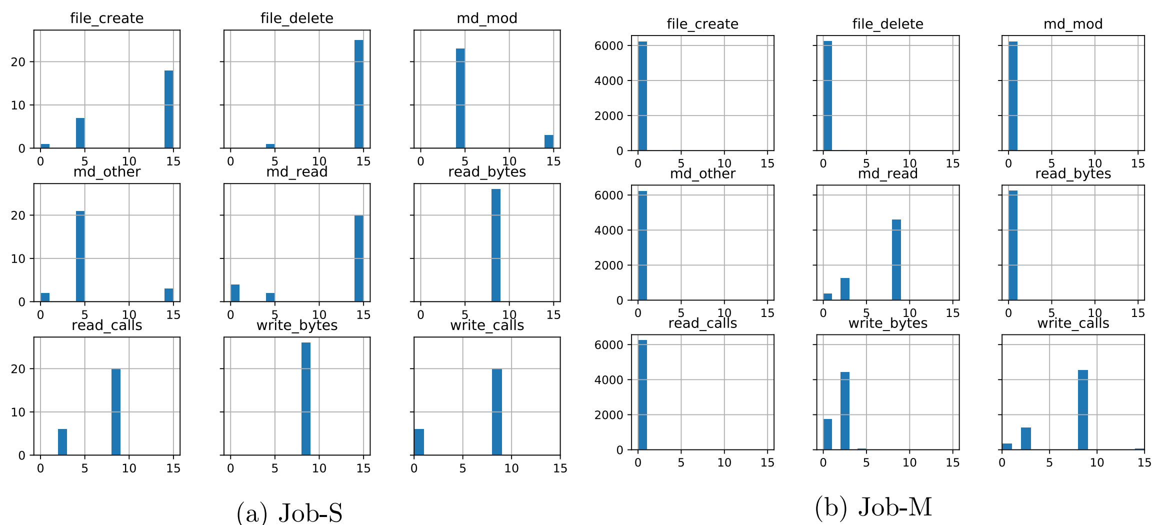
(b) Job-M (runtime=28,828 s, segments=48)

Figure 4.1: Reference jobs: segmented timelines of mean I/O activity



(c) Job-L (first 30 segments of 400; remaining segments are zero)

Figure 4.1: Reference jobs: segmented timelines of mean I/O activity



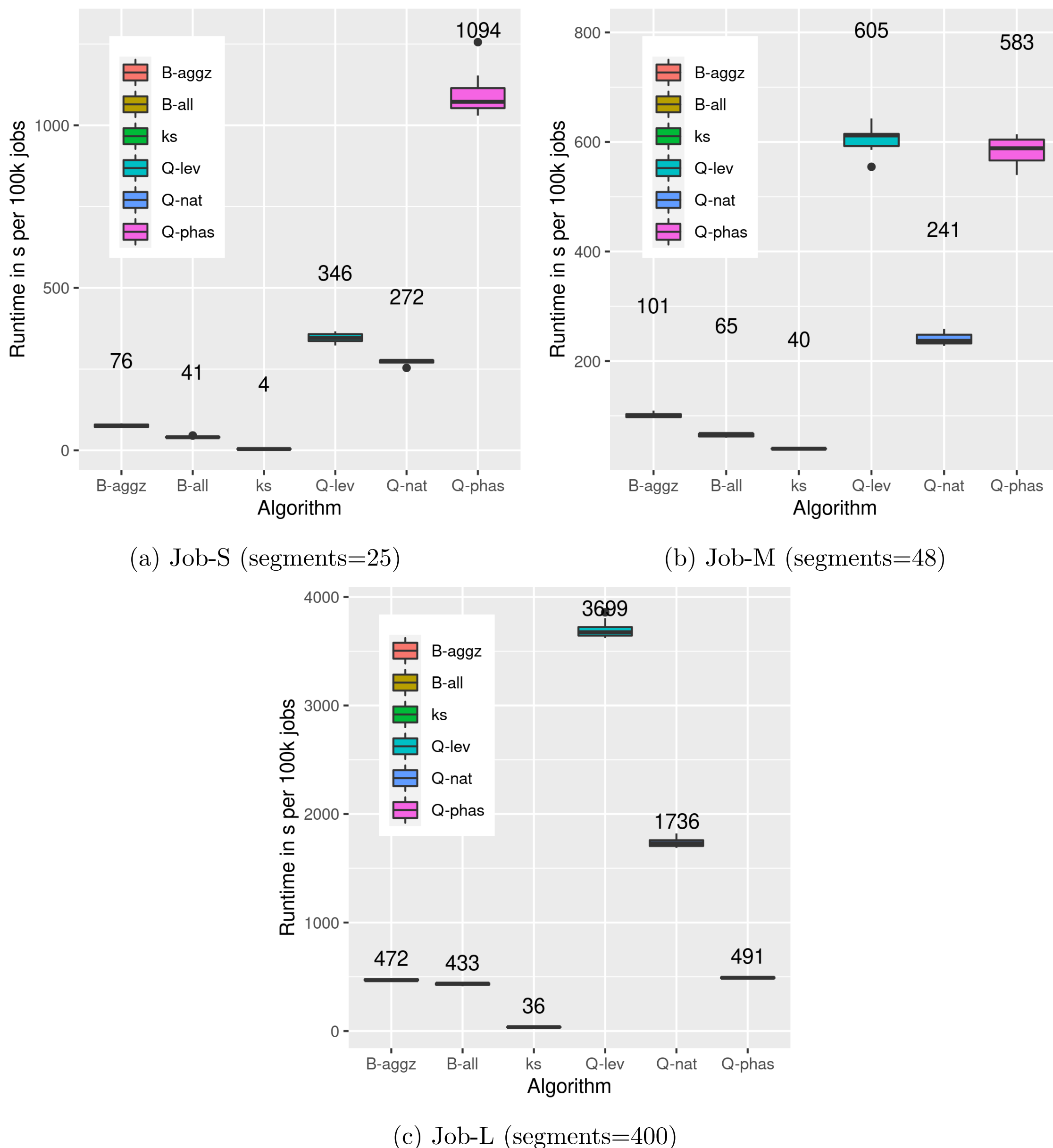
(a) Job-S (b) Job-M

Figure 4.2: Reference jobs: histogram of I/O activities

Job-M while it is fast for Job-L. The reason is that just one phase is extracted for Job-L. The Levenshtein-based algorithms take longer for longer jobs – proportional to the job length as it applies a sliding window. The KS algorithm is faster than the others by 10x, but it operates on the statistics of the time series. Note that the current algorithms are sequential and executed on just one core. They could easily be parallelized, which would then allow for an online analysis.

5.2 Quantitative Analysis

In the quantitative analysis, we explore the different algorithms how the similarity of our pool of jobs behaves to our reference jobs. The support team in a data center may have time to investigate the most similar jobs. Time for the analysis is typically bound, for instance, the team may analyze the 100 most similar jobs and rank them; we refer to them as the Top 100 jobs, and *Rank i* refers to the job that has the i-th highest similarity to the reference job – sometimes these values can be rather close together as we see in the histogram in



(a) Job-S (segments=25)

(b) Job-M (segments=48)

(c) Job-L (segments=400)

Figure 5.1: Runtime of the algorithms to compute the similarity to reference jobs

Figure 5.2 for the actual number of jobs with a given similarity. As we focus on a feasible number of jobs, we crop it at 100 jobs (total number of jobs is still given). It turns out that both B algorithms produce nearly identical histograms, and we omit one of them. In the figures, we can see again a different behavior of the algorithms depending on the reference job. Especially for Job-S, we can see clusters with jobs of higher similarity (e.g., at Q-lev at SIM=75%) while for Job-M, the growth in the relevant section is more steady. For Job-L, we find barely similar jobs, except when using the Q-phases and KS algorithms. Q-phases find 393 jobs that have a similarity of 100%, thus they are indistinguishable, while KS identifies 6880 jobs with a similarity of at least 97.5%. Practically, the support team would start with Rank 1 (most similar job, e.g., the reference job) and walk down until the jobs look different, or until a cluster of jobs with close similarity is analyzed.

5.2.1 Inclusivity and Specificity

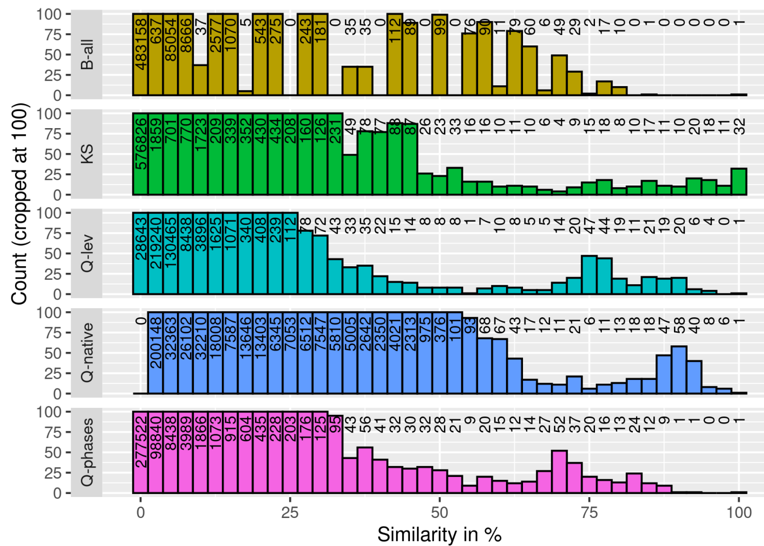
When analyzing the overall population of jobs executed on a system, we expect that some workloads are executed several times (with different inputs but with the same configuration) or are executed with slightly different configurations (e.g., node counts, timesteps). Thus, potentially our similarity analysis of the job population may just identify the re-execution of the same workload. Typically, the support staff would identify the re-execution of jobs by inspecting job names, which are user-defined generic strings[4].

To understand if the analysis is inclusive and identifies different applications, we use two approaches with our Top 100 jobs: We explore the distribution of users (and groups), runtime, and node count across jobs. The algorithms should include different users, node counts, and across runtime. To confirm the hypotheses presented, we analyzed the job metadata comparing job names which validate our quantitative results discussed in the following.

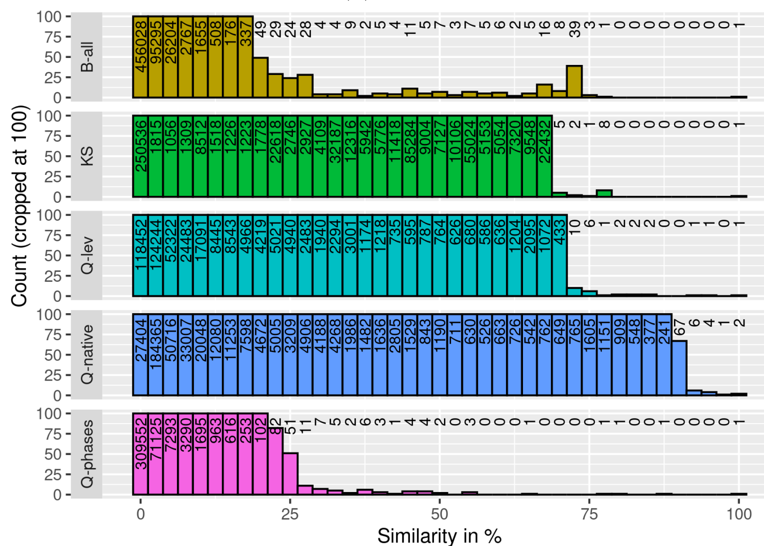
User distribution. To understand how the Top 100 are distributed across users, the data is grouped by user ID and counted. Figure 5.3 shows the stacked user information, where the lowest stack is the user with the most jobs and the topmost user in the stack has the smallest number of jobs. For Job-S, we can see that about 70-80% of jobs stem from one user, for the Q-lev and Q-native algorithms, the other jobs stem from a second user while B algorithms include jobs from additional users (5 in total). For Job-M, jobs from more users are included (13); about 25% of jobs stem from the same user; here, Q-lev, Q-native, and KS include more users (29, 33, and 37, respectively) than the other three algorithms. For Job-L, the two Q algorithms include (12 and 13) a bit more diverse user community than the B algorithms (9) but Q-phases cover 35 users. We didn't include the group analysis in the figure as user count and group ID are proportional, at most the number of users is 2x the number of groups. Thus, a user is likely from the same group and the number of groups is similar to the number of unique users.

Node distribution. Figure 5.4 shows a boxplot for the node counts in the Top 100 – the red line marks the reference job. All algorithms reduce over the node dimensions, therefore, we naturally expect a big inclusion across the node range as long as the average I/O behavior of the jobs is similar. For Job-M and Job-L, we can observe that indeed the range of nodes for similar jobs is between 1 and 128. For Job-S, all 100 top-ranked jobs use one node. As post-processing jobs use typically one node and the number of post-processing jobs is a high proportion, it appears natural that all Top 100 are from this class of jobs, which is confirmed by investigating the job metadata. The boxplots have different shapes which is an indication that the different algorithms identify a different set of jobs – we will analyze this later further.

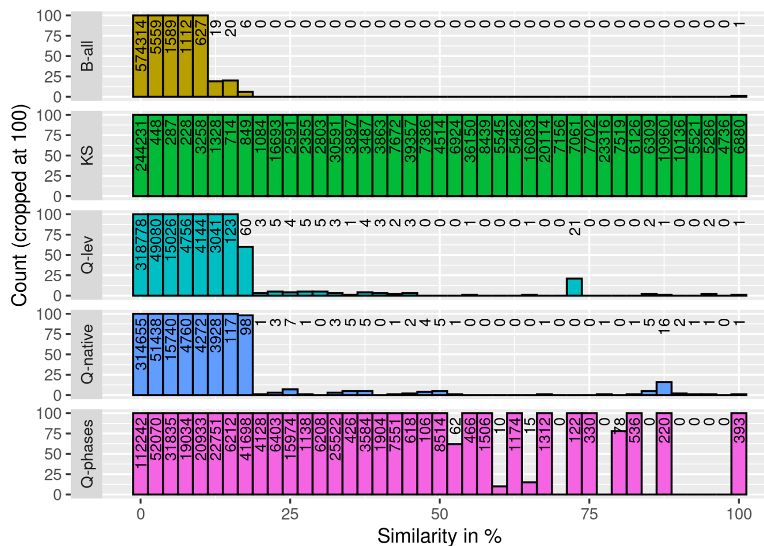
[4] As they can contain confidential data, it is difficult to anonymize them without perturbing the meaning. Therefore, they are not published in our data repository.



(a) Job-S

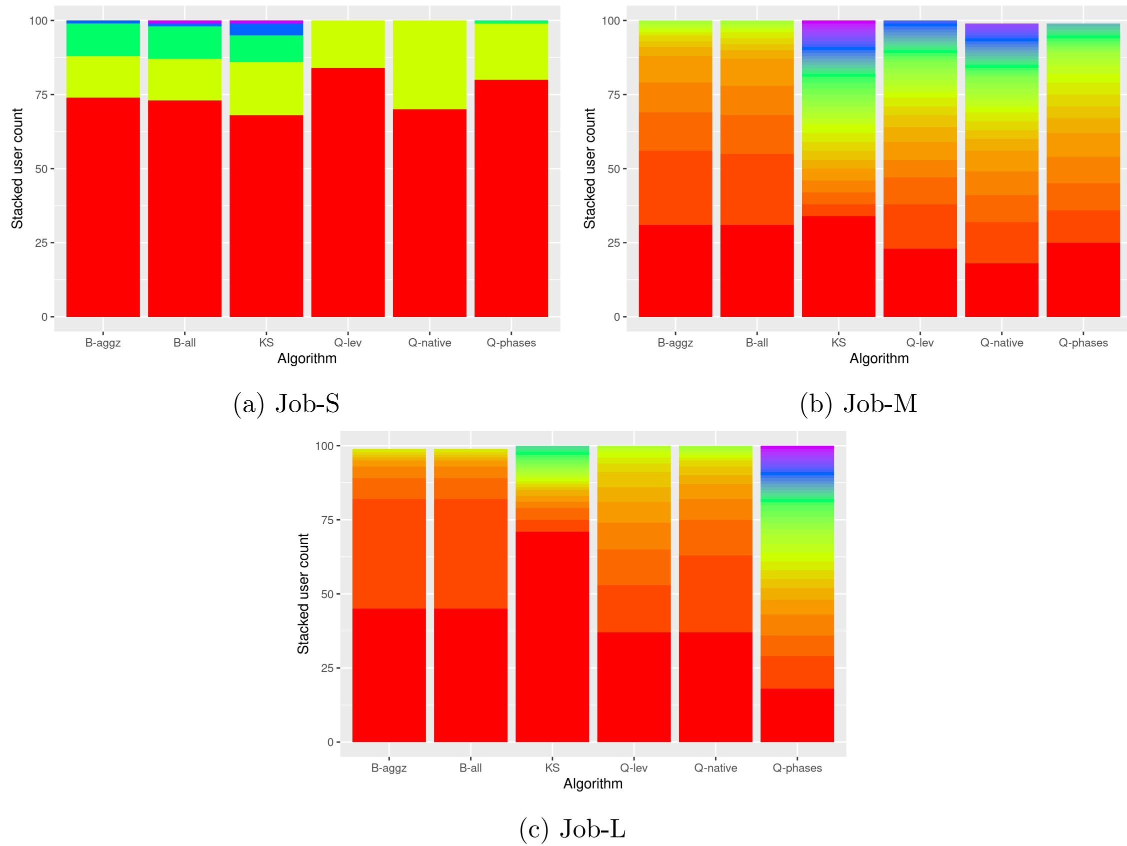


(b) Job-M



(c) Job-L

Figure 5.2: Histogram for the number of jobs (bin width: 2.5%, numbers are the actual job counts). B-aggz is nearly identical to B-all and therefore omitted.



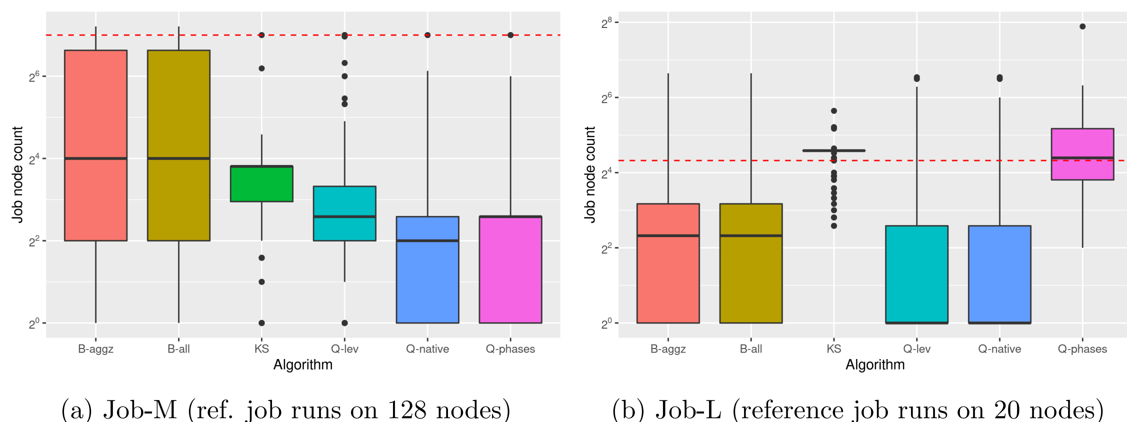
(a) Job-S

(b) Job-M

(c) Job-L

Figure 5.3: User information for all 100 top-ranked jobs. Each color represents a specific user for the given data.

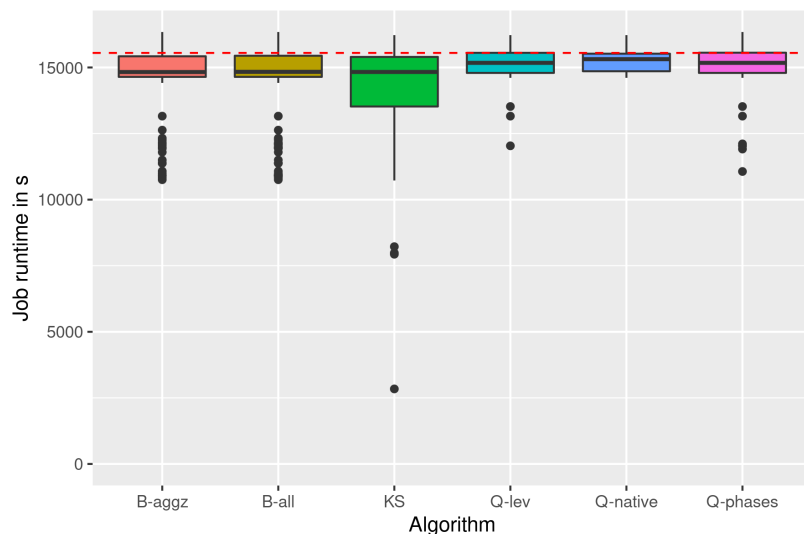
Runtime distribution. The job runtime of the Top 100 jobs is shown using boxplots in Figure 5.5. While all algorithms can compute the similarity between jobs of different lengths, the B algorithms and Q-native penalize jobs of different lengths preferring jobs of very similar length. For Job-M and Job-L, Q-phases and KS are able to identify much shorter or longer jobs. For Job-L, for Q-phases and KS, the job itself isn't included in the chosen Top 100 (see Figure 5.2c, 393 jobs have a similarity of 100%) which is the reason why the job runtime isn't shown in the figure itself. Also, as there are only few jobs of similar lengths to Job-L and the B-* algorithms penalize job-length differences, the Top 100 similar jobs have a significant difference in job length.



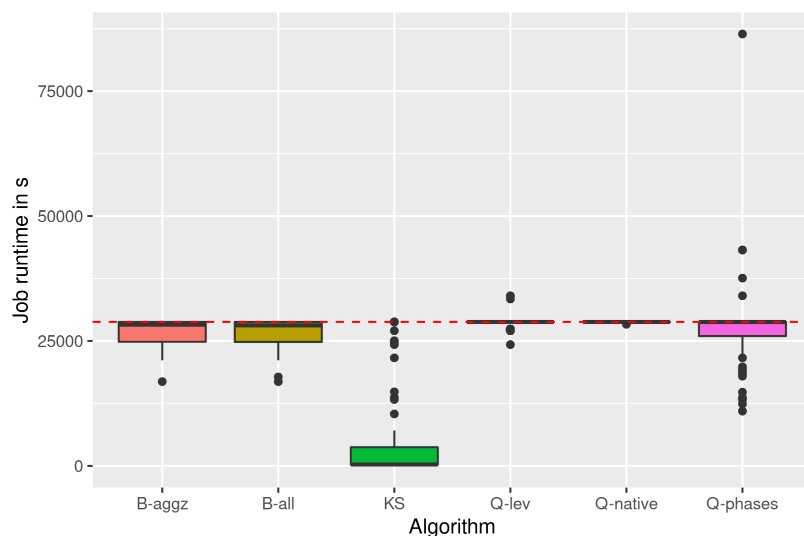
(a) Job-M (ref. job runs on 128 nodes)

(b) Job-L (reference job runs on 20 nodes)

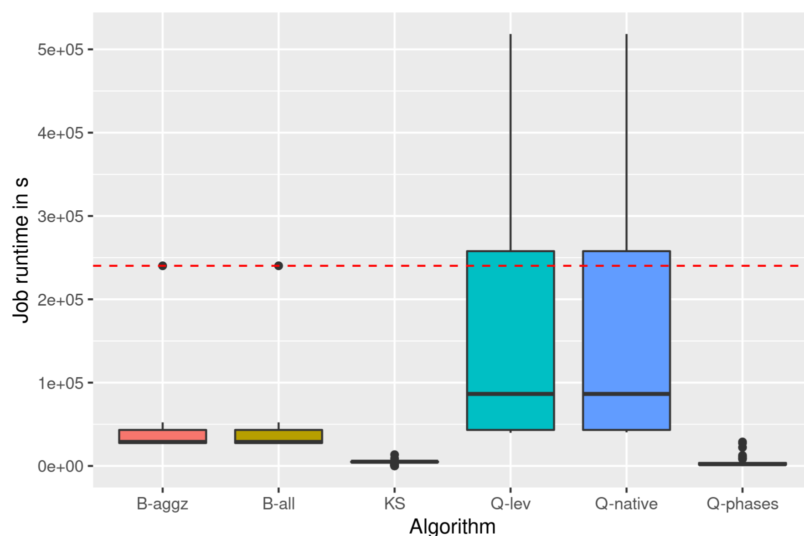
Figure 5.4: Distribution of node counts for Top 100 (for Job-S always nodes=1)



(a) Job-S ($job = 15,551s$)

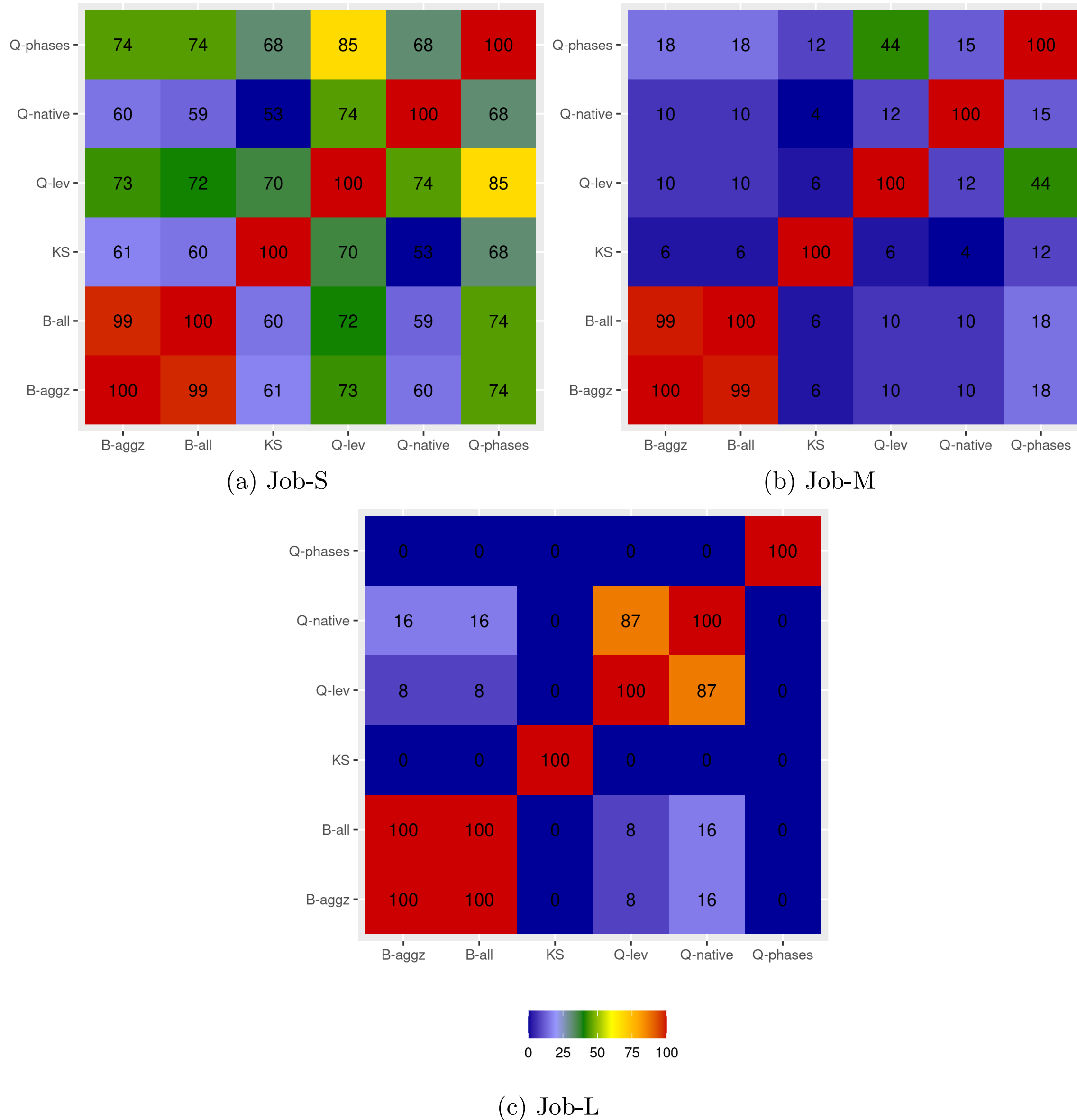


(b) Job-M ($job = 28,828s$)



(c) Job-L ($job = 240ks$)

Figure 5.5: Distribution of runtime for all 100 top-ranked jobs



(a) Job-S

(b) Job-M

(c) Job-L

Figure 5.6: Intersection of the 100 top-ranked jobs for different algorithms

5.2.2 Algorithmic differences

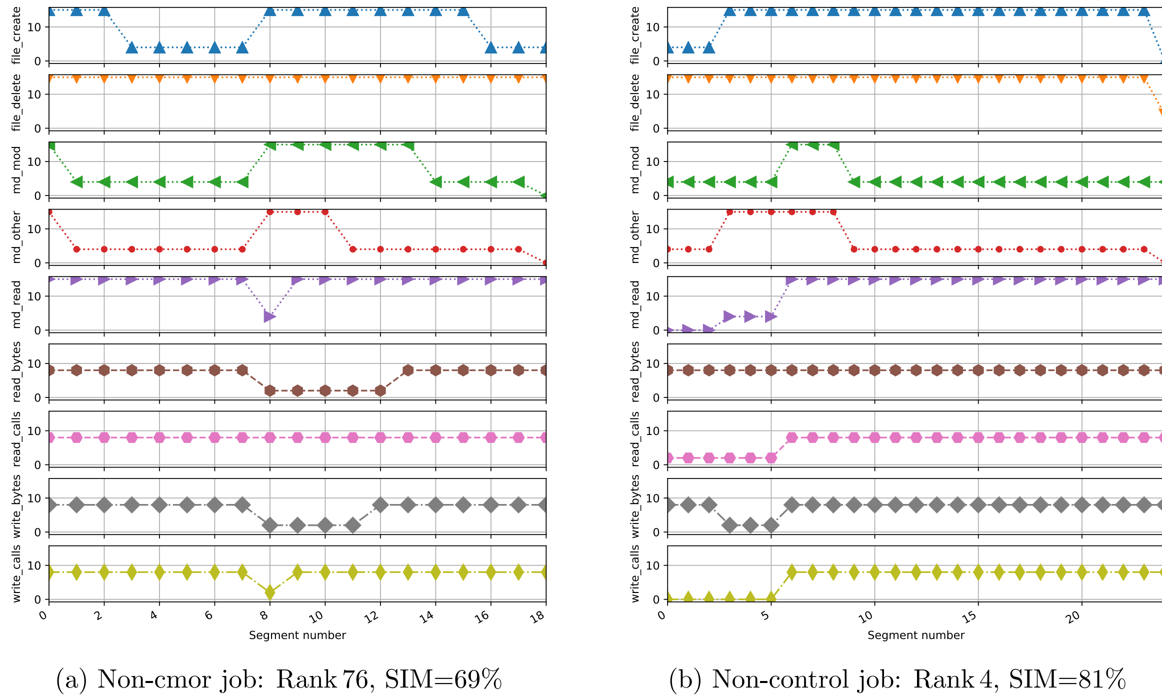
To verify that the different algorithms behave differently, the intersection for the Top 100 is computed for all combinations of algorithms and visualized in Figure 5.6. B-all and B-aggz overlap with at least 99 ranks for all three jobs. While there is some reordering, both algorithms lead to a comparable set. All algorithms have a significant overlap for Job-S. For Job-M, however, they lead to a different ranking, and Top 100, particularly KS determines a different set. Generally, Q-lev and Q-native are generating more similar results than other algorithms. From this analysis, we conclude that one representative from B is sufficient as it generates very similar results while the other algorithms identify mostly disjoint behavioral aspects.

6 Assessing Timelines for Similar Jobs

To verify the suitability of the similarity metrics, for each algorithm, we carefully investigated the timelines of each of the jobs in the Top 100. We subjectively found that the approach

B-aggz	B-all	Q-lev	Q-native	Q-phases	KS
38	38	33	26	33	0

Table 6.1: Job-S: number of jobs with “control” in their name in the Top-100



(a) Non-cmor job: Rank 76, SIM=69%

(b) Non-control job: Rank 4, SIM=81%

Figure 6.1: Job-S: jobs with different job names when using B-aggz

works very well and identifies suitable similar jobs. To demonstrate this, we include a selection of job timelines and selected interesting job profiles. These can be visually and subjectively compared to our reference jobs shown in Figure 4.1. For space reasons, the included images will be scaled down making it difficult to read the text. However, we believe that they are still well suited for a visual inspection and comparison.

6.1 Job-S

This job represents post-processing (CMORization) which is a typical step. It is executed for different simulations and variables across timesteps. The job name suggests that it is applied to the control variable. In the metadata, we found 22,580 jobs with “cmor” in the name of which 367 jobs mention “control”.

The B and KS algorithms identify one job whose name doesn’t include “cmor”. All other algorithms identify only “cmor” jobs and 26-38 of these jobs are applied to “control” (see Table 6.1) – only the KS algorithm doesn’t identify any job with control. A selection of job timelines on control variables is given in Figure 6.2. The single non-cmor job and a high-ranked non-control cmor job is shown in Figure 6.1. While we cannot visually see many differences between these two jobs compared to the control job, the algorithms indicate that jobs processing the control variables are more similar as they are more frequent in the Top 100 jobs. For Job-S, we found that all algorithms work well and, therefore, omit further timelines.

6.2 Job-M

Inspecting the Top 100 for this reference job is highlighting the differences between the algorithms. All algorithms identify a diverse range of job names for this reference job in the

Top 100. Firstly, the same name of the reference job appears 30 times in the whole dataset. Additional 932 jobs have a slightly modified name. So this job type isn't necessarily executed frequently and, therefore, our Top 100 is expected to contain other names. All algorithms identify only the reference job but none of the other jobs with the identical name but 1 (KS), 2 (B-* and Q-native) to 3 (Q-lev and Q-phases) jobs with slightly modified names. Some applications are more prominent in these sets, e.g., for B-aggzero, 32 jobs contain WRF (a model) in the name. The number of unique names is 19, 38, 49, and 51 for B-aggzero, Q-phases, Q-native, and Q-lev, respectively.

When inspecting their timelines, the jobs that are similar according to the B algorithms (see Figure 6.4) subjectively appear to us to be different. The reason lies in the definition of the B-* similarity, which aggregates all I/O statistics into one timeline. The other algorithms like Q-lev (Figure 6.5) and Q-native (Figure 6.6) seem to work as intended: While jobs exhibit short bursts of other active metrics even for low similarity, we can eyeball a relevant similarity particularly for Rank 2 and Rank 3 which have the high similarity of 90+%. For Rank 15 to Rank 100, with around 70% similarity, a partial match of the metrics is still given. The KS algorithm working on the histograms ranks the jobs correctly on the similarity of their histograms. However, as it does not deal with the length of the jobs, it may identify jobs of very different length. In Figure 6.3, we see the 3rd ranked job whose profile is indeed quite similar but the time series differs but it is just running for 10min (1 segment) on 10 nodes. Remember, for the KS algorithm, we concatenate the metrics of all nodes together instead of averaging it in order to explore if node-specific information helps the similarity.

6.3 Job-L

The B algorithms find a low similarity (the best 2nd ranked job is 17% similar), the inspection of job names (14 unique names) leads to two prominent applications: bash and xmessy with 45 and 48 instances, respectively. In Figure 6.7, it can be seen that the found jobs have little in common with the reference job.

The Q-lev and Q-native algorithms identify a more diverse set of applications (18 unique names and no xmessy job). Q-native Figure 6.8 finds long jobs with only little activity which, therefore, is similar to our reference job. The Q-phases algorithm finds 85 unique names but as there is only one short I/O phase in the reference job, it finds many (short) jobs with 100% similarity as seen in Figure 6.9. The KS algorithm is even more inclusive having 1285 jobs with 100% similarity; the 100 selected ones contain 71 jobs ending with t127, which is a typical model configuration. As expected, the histograms mimic the profile of the reference job, and thus, the algorithm does what it is expected to do.

7 Conclusion

We conducted a study to identify similar jobs based on timelines of nine I/O statistics. Therefore, we applied six different algorithmic strategies developed before and included this time as well a distance metric based on the Kolmogorov-Smirnov-Test. The quantitative analysis shows that a diverse set of results can be found and that only a tiny subset of the 500k jobs is very similar to each of the three reference jobs. For the small post-processing job, which is executed many times, all algorithms produce suitable results. For Job-M, the algorithms exhibit a different behavior. Job-L is tricky to analyze, because it is compute-intense with only a single I/O phase at the beginning. Generally, the KS algorithm finds jobs with similar histograms which are not necessarily what we subjectively are looking for. We found that the approach to compute similarity of reference jobs to all jobs and ranking these was successful to find related jobs that we were interested in. The Q-lev and Q-native work best according to our subjective qualitative analysis. Typically, a related job stem from

the same user/group and may have a related job name, but the approach was able to find other jobs as well. The pre-processing of the algorithms and distance metrics differ leading to alternative definitions of similarity. The data center support/user must define how to define similarity to select the algorithm that suits best. Another consideration could be to identify jobs that are found by all algorithms, i.e., jobs that meet a certain (rank) threshold for different algorithms. That would increase the likelihood that these jobs are very similar and what the user is looking for.

Our next step is to foster a discussion in the community to identify and define suitable similarity metrics for the different analysis purposes.

Acknowledgment

We thank the reviewers for their comments and suggestions.

References

[BK20] Eugen Betke and Julian Kunkel. The Importance of Temporal Behavior when Classifying Job IO Patterns Using Machine Learning Techniques. In *High Performance Computing: ISC High Performance 2020 International Workshops, Revised Selected Papers*, number 12151 in Lecture Notes in Computer Science, pages 191–205. Springer, 06 2020.

[BK21] Eugen Betke and Julian Kunkel. Classifying Temporal Characteristics of Job I/O Using Machine Learning Techniques. *Journal of High Performance Computing*, 01 2021.

[BKW+20] Jiwoo Bang, Chungyong Kim, Kesheng Wu, Alex Sim, Suren Byna, Sunggon Kim, and Hyeonsang Eom. HPC Workload Characterization Using Feature Selection and Clustering. In *Proceedings of the 3rd International Workshop on Systems and Network Telemetry and Analytics*, pages 33–40, 2020.

[BM18] Amir Bahmani and Frank Mueller. Chameleon: Online clustering of mpi program traces. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1102–1112. IEEE, 2018.

[Cha19] Nicolas Chan. A Resource Utilization Analytics Platform Using Grafana and Telegraf for the Savio Supercluster. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*, pages 1–6. 2019.

[DSB13] Orianna DeMasi, Taghrid Samak, and David H Bailey. Identifying HPC codes via performance logs and machine learning. In *Proceedings of the first workshop on Changing landscapes in HPC security*, pages 23–30, 2013.

[EBB+14] Todd Evans, William L Barth, James C Browne, Robert L DeLeon, Thomas R Furlani, Steven M Gallo, Matthew D Jones, and Abani K Patra. Comprehensive resource use monitoring for HPC systems with TACC stats. In *2014 First International Workshop on HPC User Support Tools*, pages 13–21. IEEE, 2014.

[EVGB15] Joseph Emeras, Sébastien Varrette, Mateusz Guzek, and Pascal Bouvry. Evalix: classification and prediction of job resource consumption on HPC platforms. In *Job Scheduling Strategies for Parallel Processing*, pages 102–122. Springer, 2015.

[HDRFV20] Mohamed S Halawa, Rebeca P Díaz Redondo, and Ana Fernández Vilas. Unsupervised KPIs-Based Clustering of Jobs in HPC Data Centers. *Sensors*, 20(15):4111, 2020.

[KBB+19] Julian Kunkel, Eugen Betke, Matt Bryson, Philip Carns, Rosemary Francis, Wolfgang Frings, Roland Laifer, and Sandra Mendez. Tools for Analyzing Parallel I/O. In *High Performance Computing: ISC High Performance 2018 International Workshops, Frankfurt/Main, Germany, June 28, 2018, Revised Selected Papers*, number 11203 in Lecture Notes in Computer Science, pages 49–70. Springer, 01 2019.

[KS18] Hassan Khotanlou and Amir Salarpour. An Empirical Comparison of Distance Measures for Multivariate Time Series Clustering. *International Journal of Engineering*, 31(2):250–262, 2018.

[LLK+20] Zhengchun Liu, Ryan Lewis, Rajkumar Kettimuthu, Kevin Harms, Philip Carns, Nageswara Rao, Ian Foster, and Michael E Papka. Characterization and identification of HPC applications at leadership computing facility. In *Proceedings of the 34th ACM International Conference on Supercomputing*, pages 1–12, 2020.

[MP07] Michael D Morse and Jignesh M Patel. An efficient and accurate method for evaluating time series similarity. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007.

[MPW+12] Sandra Méndez, Javier Panadero, Alvaro Wong, Dolores Rexachs, and Emilio Luque. A new approach for Analyzing I/O in parallel scientific applications. *Computer Science & Technology Series*, 2012.

[Nav01] Gonzalo Navarro. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 33(1):31–88, 2001.

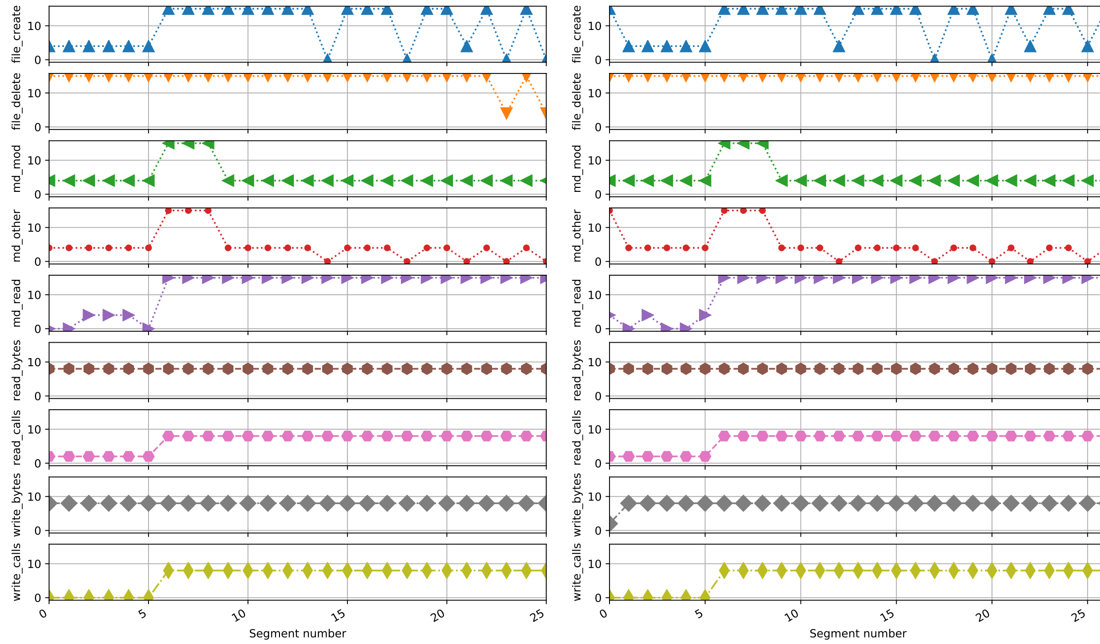
[RÖE+18] Gonzalo P Rodrigo, P-O Östberg, Erik Elmroth, Katie Antypas, Richard Gerber, and Lavanya Ramakrishnan. Towards understanding HPC users and systems: a NERSC case study. *Journal of Parallel and Distributed Computing*, 111:206–221, 2018.

[SWD+18] Nikolay A Simakov, Joseph P White, Robert L DeLeon, Steven M Gallo, Matthew D Jones, Jeffrey T Palmer, Benjamin Plessinger, and Thomas R Furlani. A Workload Analysis of NSF's Innovative HPC Resources Using XDMoD. *arXiv preprint arXiv:1801.04306*, 2018.

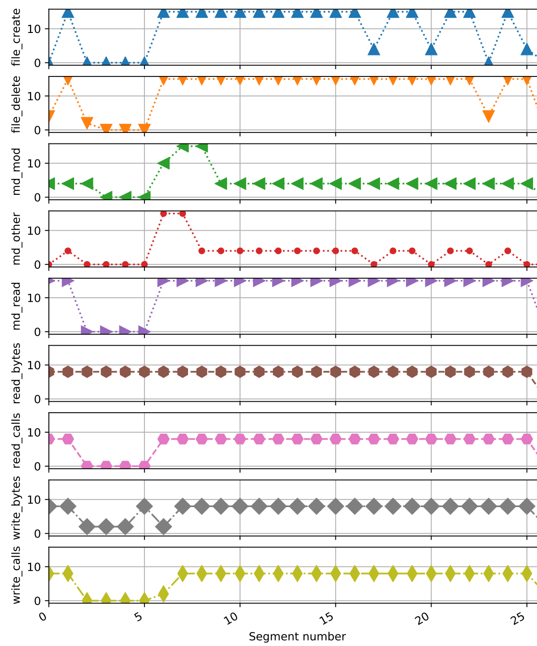
[TSMS+19] Andrew Turner, Dominic Sloan-Murphy, Karthee Sivalingam, Harvey Richardson, and Julian Kunkel. Analysis of parallel I/O use on the UK national supercomputing service, ARCHER using Cray's LASSi and EPCC SAFE. 10 2019.

[WBW+17] Matthias Weber, Ronny Brendel, Michael Wagner, Robert Dietrich, Ronny Tschüter, and Holger Brunst. Visual Comparison of Trace Files in Vampir. In *Programming and Performance Visualization Tools*, pages 105–121. Springer, 2017.

[WKD+18] Joseph P White, Alexander D Kofke, Robert L DeLeon, Martins Innus, Matthew D Jones, and Thomas R Furlani. Automatic Characterization of HPC Job Parallel Filesystem I/O Patterns. In *Proceedings of the Practice and Experience on Advanced Research Computing*, pages 1–8. 2018.

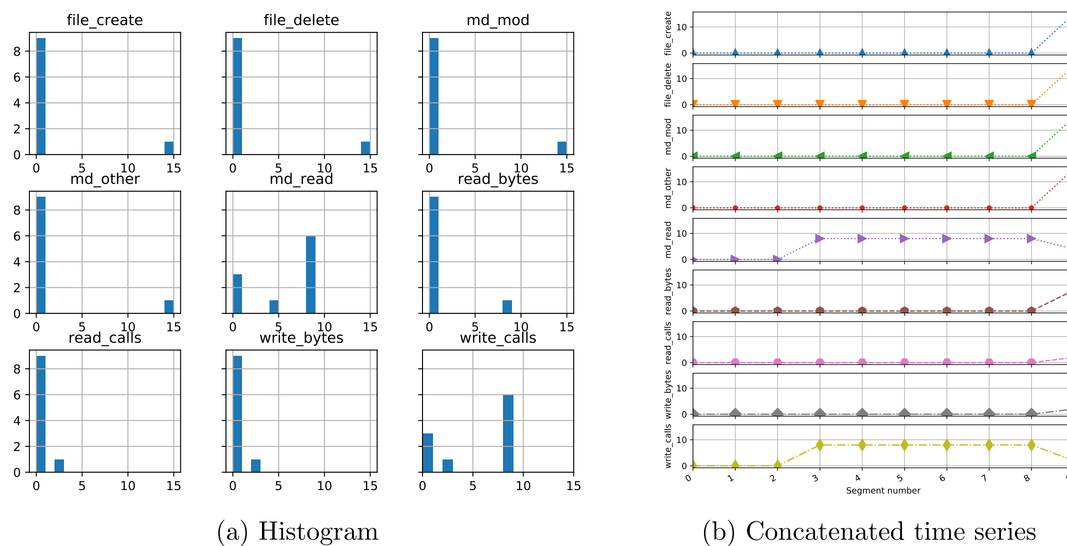


(a) Rank 2, SIM=96% (b) Rank 15, SIM=90%



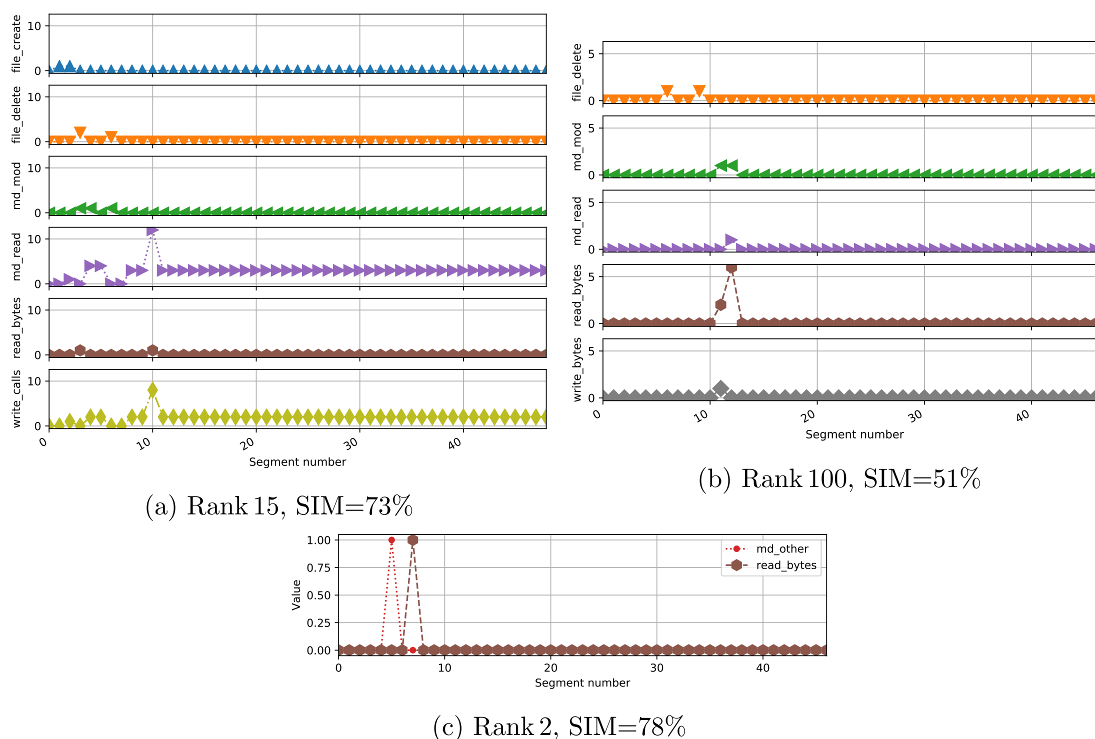
(c) Rank 100, SIM=79%

Figure 6.2: Job-S with Q-Lev, selection of similar jobs



(a) Histogram (b) Concatenated time series

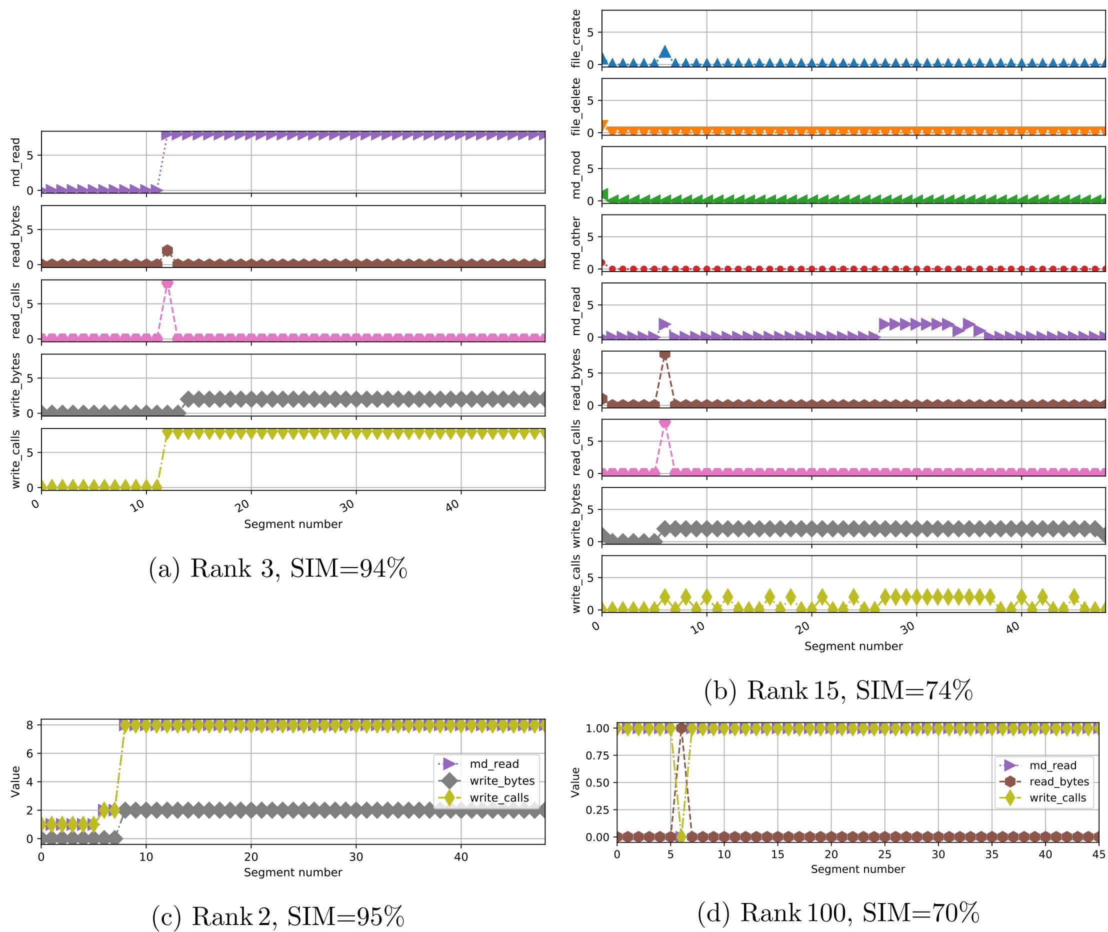
Figure 6.3: Job-M with KS, for Rank 3, SIM=78%



(a) Rank 15, SIM=73% (b) Rank 100, SIM=51%

(c) Rank 2, SIM=78%

Figure 6.4: Job-M with Bin-Aggzero, selection of similar jobs



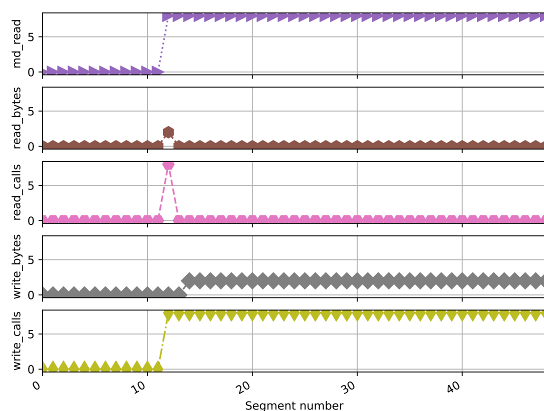
(a) Rank 3, SIM=94%

(b) Rank 15, SIM=74%

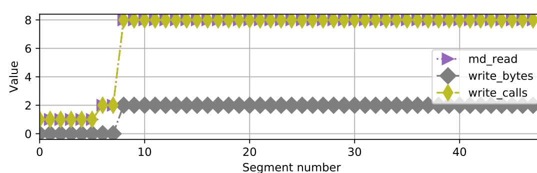
(c) Rank 2, SIM=95%

(d) Rank 100, SIM=70%

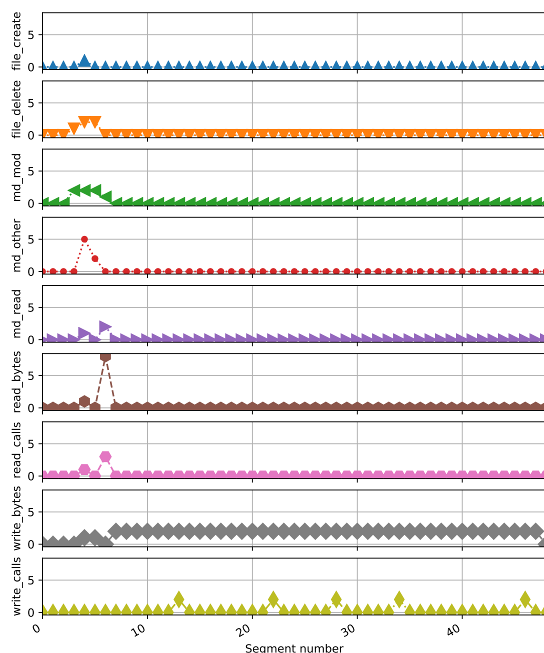
Figure 6.5: Job-M with Q-lev, selection of similar jobs



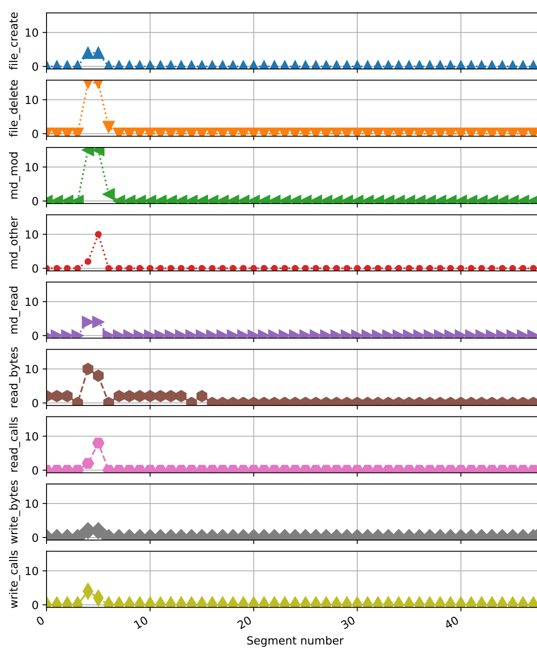
(a) Rank 2, SIM=99%



(b) Rank 3, SIM=97%

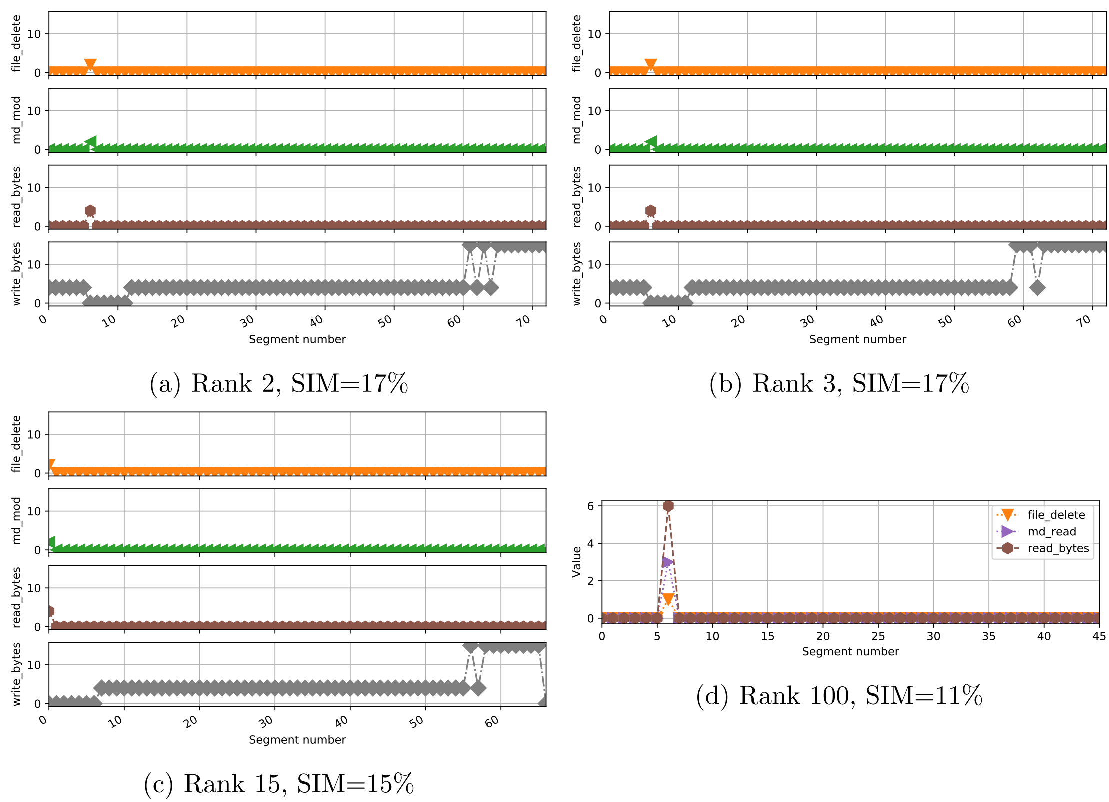


(c) Rank 15, SIM=91%



(d) Rank 100, SIM=88%

Figure 6.6: Job-M with Q-native, selection of similar jobs



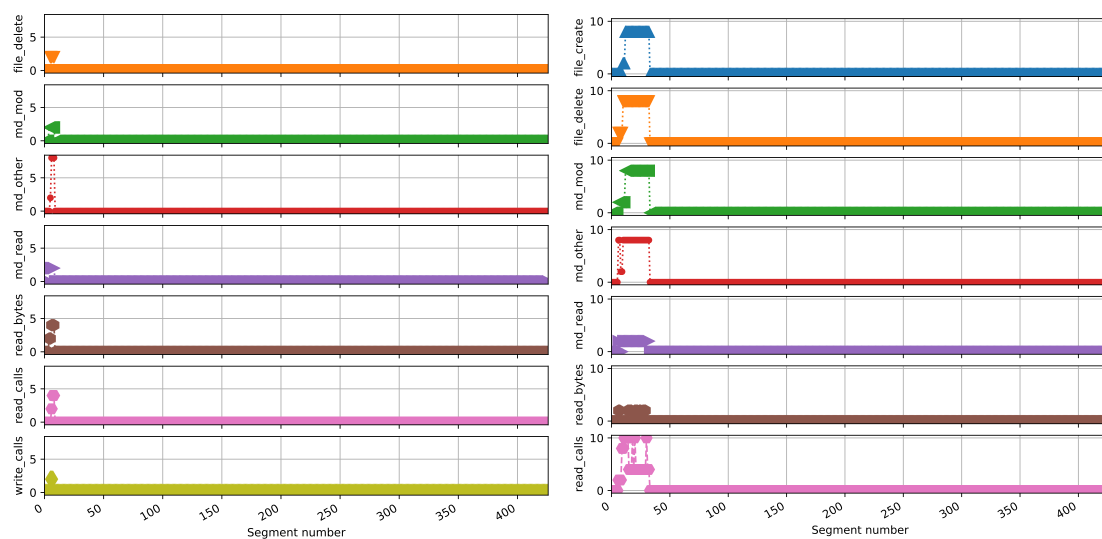
(a) Rank 2, SIM=17%

(b) Rank 3, SIM=17%

(c) Rank 15, SIM=15%

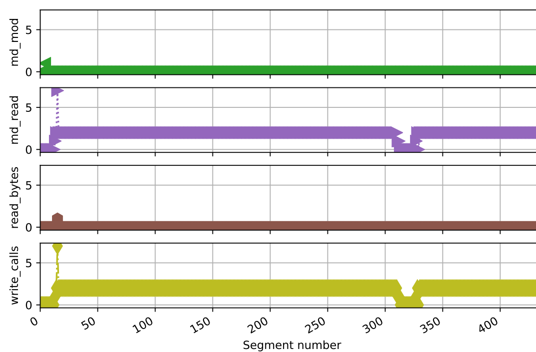
(d) Rank 100, SIM=11%

Figure 6.7: Job-L with B-aggzero, selection of similar jobs



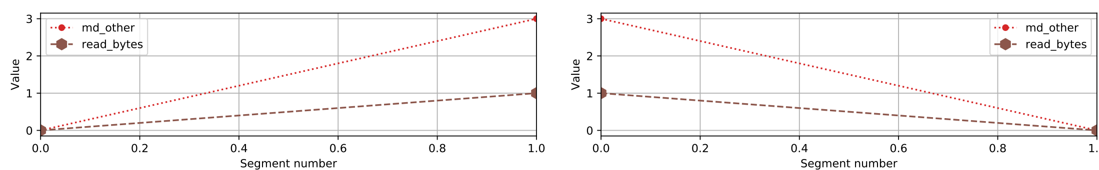
(a) Rank 2, SIM=94%

(b) Rank 3, SIM=93%



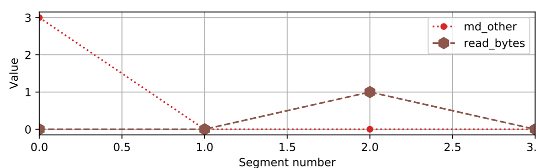
(c) Rank 15, SIM=87%

Figure 6.8: Job-L with Q-native, selection of similar jobs



(a) Rank 2, SIM=100%

(b) Rank 3, SIM=100%



(c) Rank 100, SIM=100%

Figure 6.9: Job-L with Q-phases, selection of similar jobs

8 Reviews

This section is optional for reviewers and shows their assessment that lead to the acceptance of the original manuscript. Reviewers may or may not update their review for a major update of the paper, the exact trail is available in GitHub repository of this article. The reviews are part of the article and validate the acceptance. Please check the details on the JHPS webpage.

Reviewer: Feiyi Wang, Date: 2020-07-07

Overall summary and proposal for acceptance The paper proposed the method of identifying jobs having an I/O pattern similar to a referenced job. The work could be an important contribution to the HPC community. The work takes three reference jobs and then finds out the jobs similar to the reference jobs from job pools. Jobs have been collected for six months in production operation, ca. 500K jobs. Overall, the paper should be accepted, provided that the authors answer the comments.

Reviewer: Garcia-Blas Javier, Date: 2021-05-17

Overall summary and proposal for acceptance The paper presents an experimental study of a methodology employed for identifying similarities between concurrent running applications in clusters, using the I/O pattern of them. The methodology aims to enable the adaptation of the system, allowing the optimization of the I/O subsystem. Additionally, this paper provides a novel method for calculating the similarity of I/O patterns.

The paper is well addressed and the topic interesting. The authors compared a significant amount of data to cope with the problem.

Yes, I recommend this paper to be accepted.

Scope The paper fits the workshop's topics.

Significance Major.

Readability Good.

Presentation The paper presentation is good. The language and the paper structure is clear.

References Yes

Correctness

Reviewer: Gorini Stefano Claudio, Date: 2021-05-18

Overall summary and proposal for acceptance The paper is presenting a methodology for identifying similarities in concurrent job running similar I/O pattern leveraging Kolmogorov-Smirnov algorithm to reduce the dimension to be considered (factor 2 simplification).The goal of optimizing the I/O subsystem could give fantastic benefit to a user that is willing to optimise his code. Considering the trend these days where data oriented workflows are predominant this work is definitively interesting.

The paper is well structured and the authors analyzed a conspicuous dataset giving the article a solid ground.

I do recommend this paper to be accepted.

Scope It is definitively in the scope of the workshop.

Significance Major

Readability Well written with perfect English and perfect structure.

Presentation Clear structure and optimal presentation following a very well set logical structure.

References Yes

Correctness Based on my knowledge I would say yes it is correct.