The Importance of Temporal Behavior when Classifying Job IO Patterns Using Machine Learning Techniques

Eugen Betke[1] and Julian Kunkel[2]

[1] DKRZ – betke@dkrz.de
[2] University of Reading–j.m.kunkel@reading.ac.uk

Abstract. Every day, supercomputers execute 1000s of jobs with different characteristics. Data centers monitor the behavior of jobs to support the users and improve the infrastructure, for instance, by optimizing jobs or by determining guidelines for the next procurement. The classification of jobs into groups that express similar run-time behavior aids this analysis as it reduces the number of representative jobs to look into. It is state of the practice to investigate job similarity by looking into job profiles that summarize the dynamics of job execution into one dimension of statistics and neglect the temporal behavior.
In this work, we utilize machine learning techniques to cluster and classify parallel jobs based on the similarity in their temporal IO behavior to highlight the importance of temporal behavior when comparing jobs. Our contribution is the qualitative and quantitative evaluation of different IO characterizations and similarity measurements that work toward the development of a suitable clustering algorithm.
We explore IO characteristics from monitoring data of one million parallel jobs and cluster them into groups of similar jobs. Therefore, the time series of various IO statistics is converted into features using different similarity metrics that customize the classification. We discuss conventional ML techniques that are applied to job profiles and contrast this with the analysis of time series data where we apply the Levenshtein distance as a distance metrics. While the employed Levenshtein algorithms aren't yet optimal, the results suggest that temporal behavior is key to identify related pattern.

Keywords: IO fingerprinting, performance analysis, monitoring

1 Introduction

Scientific large-scale applications of different domains have different needs for IO and, thus, exhibit a variety of access patterns on storage. Even re-running the same simulation may lead to different behavior. We can distinguish between a temporal behavior, i.e., the operations performed over time such as long read phases, bursty IO pattern, and concurrent metadata operations, and spatial

access pattern of individual processes of the application as they can be, e.g., sequential or random.

On different supercomputers, the same IO patterns may result in different application runtimes depending on the nature of the access pattern. For example, machines equipped with burst buffers [1, 9] may significantly reduce application runtimes by absorbing bursty IO traffic. IO congestion and file system performance degradation can occur when several IO intensive jobs are running on the same machine at the same time.

In our environment at DKRZ, the raw monitoring data of a job is captured in form of a time series of nine metrics per node, each metrics sampled at five seconds intervals. When comparing the time series of such metrics between two jobs, the key question is how do we define the similarity between multiple time series. From the user support side, we might be interested in grouping similar suboptimal jobs and aim to provide one recipe to optimize all that exhibit such a behavior. Similarly, we might be interested to optimize the pattern for a single IO phase. We may be interested to ignore computation time and focus on IO phases only. Regardless of the segment of the time series we look at, we naively would consider an IO pattern to be identical if the time series for all metrics of one job is identical to those of another job.

Utilizing time series data of a job for clustering if difficult as it firstly, depends on runtime, the number of nodes, the gathered metrics, and possibly number of file systems; secondly, the temporal IO behavior of parallel jobs depends on the conditions of the cluster it is executed. For various reasons, even re-running the same job may lead to variations in execution time and, thus, observed statistics. Moreover, variants of workflows may lead to slight variations of behavior that might be relevant for a data analyst.

In this article, we discuss and demonstrate the benefit of utilizing time series data in contrast to profiles. First, we briefly discuss related work in Section 2. Next, we describe our previous work and the monitoring system used in Section 3. Our approach is described in Section 4. As jobs are of different length, a similarity metrics must be able to handle time series of different length. Two classes of approaches are investigated: 1) we generate job profiles and apply existing ML techniques to cluster data; 2) we create a string from the time series and we apply the Levenshtein distance which indicates the number of changes that need to be made between two job strings. The experimental conditions for our evaluation are described in Section 5. To evaluate these approach, we perform a qualitative analysis in Section 6 discussing the statistics about the generated clusters and a quantitative evaluation in Section 7 where we search jobs similar to a given job. Finally the paper is concluded in Section 8.

2 Related work

There are many tracing and profiling tools that are able to record IO information [6]. Most of them focus on individual jobs, and only a few of them apply machine learning for data analysis, in particular across jobs. As the purpose of

applications is computation and, thus, IO is just a byproduct, applications often spend less than 10% time with IO.

The Ellexus tools[3] include the Mistral tool which purpose is to report on and resolve IO performance issues when running complex Linux applications on high performance compute clusters. Darshan [2, 3] is an open source IO characterization tool for post-mortem analysis of HPC applications' IO behavior. Its primary objective is to capture concise but useful information with minimal overhead. This is accomplished by eschewing end-to-end tracing in favor of compact statistics such as elapsed time, access sizes, access patterns, and file names for each file opened by an application. Darshan can be used not just to investigate the IO behavior of individual applications but also to capture a broad view of system workloads for use by facility operators and IO researchers.

There are approaches that monitor record storage behavior and aim to identify inefficient applications in a cluster. TOKIO [7] integrates logs from various sources to allow an analysis of data. It allows to find certain inefficient access patterns in the data.

The LASSi tool [8] was developed for detecting victim and aggressor applications. To identify such applications, LASSi calculates metrics from Lustre job-stats and information from the job scheduler. The correlation of these metrics can help to identify applications that cause the file system to slow down. In the LASSi workflow this is a manual step, where a support team is involved in the identification of applications during file system slow down. LASSi's indicates that the main target group are system maintainers. Understanding LASSi reports may be challenging for ordinary HPC users, who do not have knowledge about the underlying storage system.

In [5], the authors utilized probes to detect file system slow-down. A probing tool measures file system response times by periodically sending metadata and read/write requests. An increase of response times correlates to the overloading of the file system. This approach allows the calculation of a slow-down factor identification of the slow-down time period. This approach is able to detect a file system slow-down, but cannot detect the jobs that cause the slow-down.

HiperJOBVIZ [**?**] is a visual analytic tool for visualizing the resource allocations of data centers for jobs, users, and usage statistics. It provides an overview of the current resource usage and a detailed view of the resource usage via multi-dimensional representation of health metrics. TimeRadar [4] is a part of the tool, which summaries the resource usage via radar charts, creating a kind of comprehensible profile for different user groups.

In contrast to existing approaches, the approach discussed in this paper focuses on the analysis of job data and investigates clustering strategy to group similar jobs.

---

[3] `https://www.ellexus.com/products/`

[4] `https://idatavisualizationlab.github.io/HPCC/TimeRadar`

3 Preliminary Work

The German Climate Computing Center (DKRZ) maintains a monitoring system that gathers various statistics from the Mistral HPC system. Mistral has 3,340 compute nodes, 24 login nodes, and two Lustre file systems (lustre01 and lustre02) that provide a capacity of 52 Petabyte.

Raw monitoring data. Figure 1 illustrates a generic example for raw monitoring data. In the example the data is captured on 2 nodes, on 2 file systems, for 2 metrics, and at 9 time points t_i.

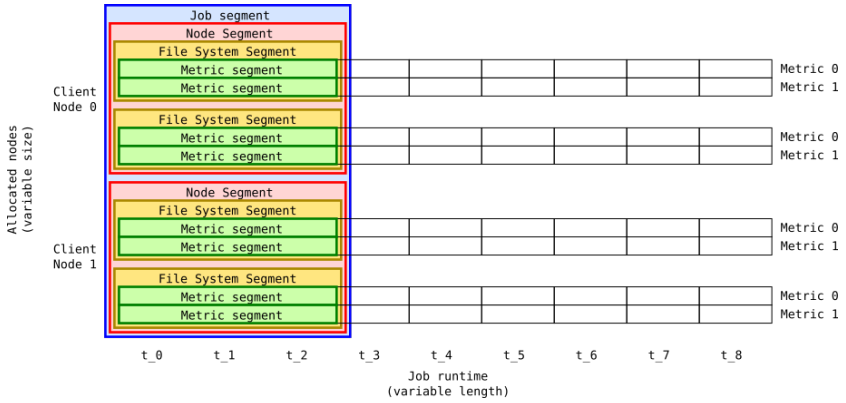

Fig. 1: A generic example of 4-dimensional raw monitoring data (Node × File System × Metric × Time) and different levels of segmentation (colored boxes).

Segmentation. We split the time series of each IO metric into equal-sized time intervals (segments) and computes a mean performance for each segment. This stage preserves the performance units (e.g., Op/s, MiB/s) for each IO metric. The generic example in Figure 1 creates segments out of three successive time points just for illustration purposes. Actually, we convert raw monitoring data to 10 minutes segments, which we found is a good trade-off to represent the temporal behavior of the application while it reduces the size of the time series. Depending on aggregation function, segments can be created of metrics, of file systems, of nodes, or even over all dimensions.

Categorization. Next, to get rid of the units, and to allow calculations between different IO metrics, we introduced a categorization pre-processing step that takes into account the performance of the underlying HPC system and assigns an unitless ordered category to each metric segment. We use a three category system, which contains the LowIO=0, HighIO=1 and CriticalIO=4 categories. The category split points are based on the observed file system usage and the

score values assigned to each category represent their weigth. We investigated both concepts in our previous work [4] This node-level data can then be used to compute job-statistics by aggregating across dimensions such as time, file systems, and nodes.

In summary, this data representation has the following key advantages for data analysis. The ordered categories make the calculations between different metrics feasible, which is not possible with raw data. Furthermore, the domains are equally scaled and compatible, because the values are between 0 and 4, and a value has a meaning. Besides, the resulting data representation is much smaller compared to the raw data. This allows us to apply compute-intensive algorithms to large datasets. Finally, irrelevant data is hidden by the LowIO category and doesn't distract from significant parts of jobs.

In our previous work, we computed three high-level Job-IO-metrics per job that aid users to understand job profiles: **Job-IO-Balance** indicates how IO load is distributed between nodes during job runtime. **Job-IO-Utilization** shows the average IO load during IO-phases but ignores computation phases. **Job-IO-Problem-Time** is the fraction of job runtime that is IO-intensive; it is approximated by the fractions of segments that are considered IO intensive.

We will use them in job profiles as well to capture some temporal behavior.

4 Methodology

The goal of this article is to research the impact of the temporal dimension when applying clustering strategies on a large number of jobs. Therefore, we compare job-profiles that neglect the temporal dimension and time series of different length represented as strings.

Generally, machine learning algorithms expect a fixed number of features. Thus, the time series that are retrieved on the node-level needs to be pre-processed. The application of a "specific algorithm" can be understood as a number of successive processing steps on data. Roughly speaking, there are three basic steps that we apply: data pre-processing including coding, similarity computation, and clustering. We call one of such a combination a *clustering stack*. The pre-processing converts the dynamic-sized monitoring data which depends on the number of captured IO metrics, allocated nodes, and application runtime into a suitable representation for the clustering algorithm. Then the clustering is applied. Finally, the clustering result needs to be assessed, i.e., how suitable is this strategy for our IO statistics and use cases? In the following, we have dedicated a section to each step discussing potential alternatives.

Data pre-processing The 4-dimensional data (Node $\times$ File System $\times$ Metric $\times$ Time) from our monitoring system is too fine-grain for mass analysis. To be able to analyse millions of jobs, we must reduce the dimensionality. Depending on reduction techniques, the result of the data-preprocessing is either a dataset of feature vectors for general-purpose algorithms, or a set of codings for custom clustering algorithms.

We decided to distinguish how the different dimensions of a job are reduced and aggregated (if at all); for example, we may summarize a metric over the node dimension and then compute the mean across time to obtain a profile for each metric and file system. Regardless of this decision, we first convert the time series into segments of 10 minutes. Hence, for a job and for each of our nine client-side recorded IO metrics, we obtain a coarse-grained time series. To simplify the interpretation of results and the choice distance metrics, it is beneficial to have the same unit for all features which is why we use our category classification which creates a unitless order. For instance, when reduced by node, file system, and across metrics, a point may represent the mean value for the job for each 10 minute runtime.

Coding Segmented data contains a numeric floating point value for each data, which can be too much information for the analysis. Therefore, we introduce two condensed data representations called binary and hexadecimal coding. Additionally, we introduce zero-aggregation, that is an operation that aggregates continuous zero segments to one zero segment.

Binary coding represents monitoring data as a sequence of numbers, where each number stands for a specific file system usage depending on the activities. In this coding approach each conceivable combination of activities has an unique number. In our implementation, we use a 9-bit number to represent each segment where each bit represents a metric. The approach maps the three categories to the following two states: The LowIO category is mapped to the non-active (compute intense) state (0), and HighIO and CriticalIO categories are mapped to the active state (1). On one side, by doing this, we lose information about performance intensity, but on other side, this simplification allows a more comprehensible comparison of job activities.

Using this kind of coding we can compute a number for each segment, that describes unambiguously the file system usage, e.g., a situation where intensive usage of md_read (Code=16) and read_bytes (Code=32) occur at the same time and no other significant loads are registered is coded by the value 48. Coding is reversible, e.g., when having value 48, the computation of active metrics is straightforward.

To reduce the 4-dimensional data, we reduce that structure to two dimensions (segments metrics) by aggregating other dimensions by summing up the score values. Additionally, sequences of zero segments can be reduced to just one zero segment to neglect the length of an application's IO phase. For presentation purposes, in the resulting table we leave zero scores. An example encoded job before and after the reduction of zero segments is shown here:

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

Hexadecimal coding preserves monitoring data for each metric and each segment. As the name suggests, the value of a segment is converted into a hexadecimal

number. The numbers are obtained in two steps. Firstly, the dimension reduction aggregates the file system and the node dimensions and computes a mean value for each metric and segment, which lies in interval [0,4]. Secondly, the mean values are quantized into 16 levels – 0 = [0,0.25), 1 = [0.5,0.75), ... , 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]
```

Similarity We use euclidean distance to determine the similarity between two job profiles. For time series, we use Levenshtein distance that is the number of operations (inserts/deletes/changes) required to convert one coding in another.

Clustering In the last step, similar jobs need to be grouped in clusters. To handle millions of jobs, the algorithm must be performant. We developed two strategies that meet the requirement, one based on widely used general-purpose algorithms, and a custom algorithm.

ClusteringTree algorithm. As we do not know how many different classes of jobs are in the dataset, a traditional k-means classification turned out to be not productive in our experiments. Therefore, we explored the usage of agglomerative clustering, however, with it's complexity of $\geq O(N^2)$, it wasn't applicable to our dataset. Thus, we simplified the application into this algorithm. This algorithm involves three steps: (1) Agglomerative clustering of a small dataset and labeling data, (2) training of a decision tree model, and (3) clustering with the decision tree of the remaining jobs.

SimplifiedDensity algorithm. Clusters are formed around centroids. That are job codings that form clusters by attracting similar jobs. All jobs in a cluster fulfill only one condition, the similarity (SIM) to the centroid has to be larger than the user defined value. The algorithm takes a non-assigned job and iterates through existing clusters looking if the similarity to the cluster centroid is larger than the user defined values. The job is assigned to the first cluster, where the condition is fulfilled. If there is no such a cluster, the job forms a new cluster and becomes a centroid of this cluster.

Clustering stacks There are various combinations of the different strategies possible. For simplicity, we refer to one clustering stack just as algorithm. During our research, we explored various combinations out of the possible combinations. The paths are visualized Figure 2 and discussed further in the following section.

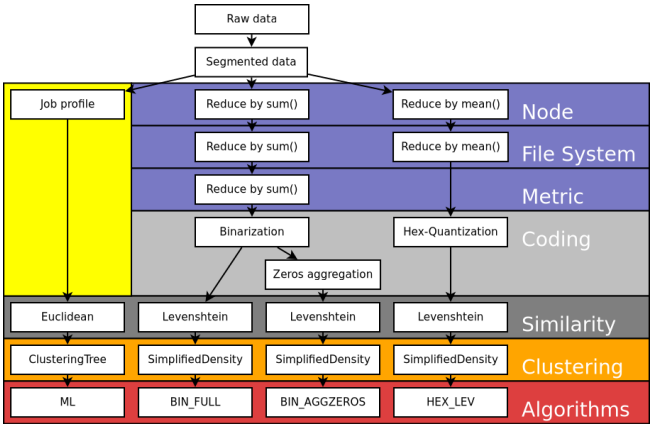

Fig. 2: Algorithms and their actual clustering stacks.

4.1 Algorithms

ML To apply existing clustering algorithms, first, a job-profile is created in the pre-processing. The 4d time series can be transformed into the required fixed size input format accepted by the general-purpose ML clustering algorithms. In the preprocessing step, the MinMaxScaler scales the features to values between 0 and 1 using MinMax normalization. Therefore, the highest distance between two points can be at most $\epsilon_{\max} = d^{1/d}$, where d is the dimension of the dataset.

We explored two job profiles: IO-metric and IO-duration. The **IO-metric job profile** utilizes three features, Job-IO-Balance, Job-IO-Utilization, and Job-IO-Problem-Time (as defined in [4]). After the data pre-processing, we obtain a set of 3-dimensional data points with a domain between 0 and 1. The maximum distance between any two jobs ($\epsilon_{\max}$) is 1.44.

The **IO-duration job profile** contains the fraction of runtime, a job spent doing the individual IO categories leading to 27 columns. The columns are named according to the following scheme: metric_category, e.g, bytes_read_0 or md_file_delete_4. The first part is the one of the nine metric names and the second part is the category number (LowIO=0, HighIO=1 and CriticalIO=4). These columns are used for machine learning as input features. There is a constraint for each metric (metric_0 + metric_1 + metric_4 = 1), that makes 9 features redundant, because they can be computed from the other features. So we have to deal with 18 features; $\epsilon_{\max}$ is 1.17.

In experiments, we observed that the agglomerative clustering algorithm that is used in this work can handle around 10,000 jobs in a reasonable amount of time as the complexity is N^2. With the following additional classification steps, we are able to cluster 1,000,000 samples:

1. Clustering and labeling 10,000 jobs with agglomerative clustering algorithm.
2. Training of a decision tree model with data from the previous step.
3. Predict labels of 1,000,000 jobs with the trained decision tree model.

BIN_ALL and BIN_AGGZEROS For these algorithms, we encode the time series of 9 metrics into one time series that is then assessed using Levensthein distance. The similarity between two jobs is determined by the following formula:

$$\text{similarity}\,(\text{job}_A, \text{job}_B) = 1 - \frac{\text{levenshtein}\,(\text{coding}_A, \text{coding}_B)}{\max\,(length_A,\, \text{length}_B)} \tag{1}$$

It computes the number of operations (changes/deletes/inserts) divided by the length of the longest sequence, and subtracted from the value one. According to this equation, the similarity between the following two jobs is 73 percent:

```
jobA: [1:5:0:0:0:0:0:0:96:96:96:96:96:96:96], 'length': 15
jobB: [0:0:0:0:0:0:0:0:0:96:96:96:96:96:98], 'length': 15
```

As a variation of this approach, we investigated also the case where consecutive zero-sequences are reduced to a single zero segment. This allows us to focus on IO intensive parts of the job. The example below shows reduced codings from the previous example. Note, that this operation has no effect on the job length and similarity computation.

The similarity between the following two codings is 53 percent:

```
jobA: [1:5:0:96:96:96:96:96:96:96], 'length': 15
jobB: [0:96:96:96:96:96:98], 'length': 15
```

HEX_LEV This similarity function works on the same principle as the BIN algorithms, with the difference that instead of a single pre-reduced time series per job, it computes the similarity between all 9 metrics of two different jobs first and then compute the mean.

This adaption allows to apply Levenshtein-based similarity on hexadecimal coding as follows:

$$\text{similarity}\,(\text{job}_A, \text{job}_B) = 1 - \frac{\sum_{m \in Metric} \text{levenshtein}\,(\text{coding}_{A,m}, \text{coding}_{B,m})}{N \cdot L_B}, \text{ with } L_B \geq L_A \tag{2}$$

4.2 Assessment

Lastly, the quality of the obtained clusters must be assessed. Overall, we will assess their suitability using quantitative metrics such as the number of generated clusters and their sizes and qualitatively by manually exploring clusters of relevant jobs. We want to emphasize that our goal is to find similar jobs. Unfortunately, it is not feasible to analyze all of them qualitatively with reasonable effort and there are no tools that can assess the cluster quality automatically. For the qualitative analysis, we start by looking into a job that is given to user support, then similar jobs need to be found. In the same cluster, we expect the sequences to be similar. If not, the clustering algorithm is not effective.

5 Experimental Setup

5.1 Data

This section describes the job data extracted from Mistral, originally we gathered 1 million jobs from a period of 203 days. Mostly jobs are allowed to run up to 8-hours, leading to time series with up to 48 segments. The general procedure for monitoring data shorter than 10 minutes, that occur inevitable in short jobs and in the last job segment, if job runtime is not divisible by 10 minutes is the following: We compute the mean performance, assume the run time of 10 minutes with this mean performance, and create one 10 minutes segment out of it. From the perspective of this work, analysis of non-IO-intensive jobs (jobs with zero in all segments) is irrelevant, these jobs can be grouped into one class easily. For that reason, we detect zero-jobs early and remove them from the dataset; these are about 40% of jobs.

The number of zero-jobs is different for hexadecimal and absolute mode codings. For BIN algorithms we create 583,000 codings and for HEX algorithms 444,000 codings. The reason is the quantization to HEX coding, which firstly computes mean performance values for all segments, and then quantizes them to 16 levels. Hereby, some segments can be quantized to zeros, if the mean value becomes sufficiently low. Therefore, it may happen that some jobs fall into the zero-job category if all segments are quantized to zeros. It can not happen in BIN coding, because it preserves all the active segments, so that no job may change the category. Interestingly, it affects around 14% of jobs.

5.2 Test environment

For the performance tests, we allocate a compute node on Mistral supercomputer. It is equipped with 2x Intel® Xeon® CPU E5 2680 v3 @ 2.50GHz, 64GB DDR4 RAM. For clustering of job profiles, we use the agglomerative clustering algorithm, decision trees, and the MinMaxScaler from the sklearn 0.22.1 library and python 3.8.0. For clustering of binary and hexadecimal codings we a clustering algorithm implemented in Rust and run it on a single core.

5.3 Algorithm parameters

ML. We explored our discussed job profiles: IO-metric and IO-duration. For both datasets we explore $\epsilon \in [0.03, 0.06, 0.09, 0.1, 0.2, 0.3]$.

BIN/HEX. We conduct experiments with BIN_ALL, BIN_AGGZEROS, and HEX_LEV algorithms, varying the $SIM \in [0.1, 0.3, 0.5, 0.7, 0.9, 0.95, 0.99]$ parameter and capturing clustering progress each time after clustering 10,000 jobs.

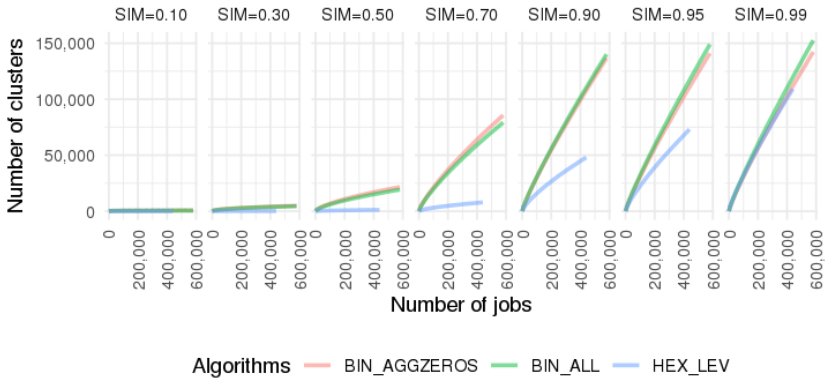

Fig. 3: Clustering progress.

6 Evaluation

ML The jobs within clusters have indeed a similar job profile, the time series and, therefore, the binary coding differs significantly. For example, a cluster can contain sequences with different IO behavior like in Table 1. Obviously, the approach don't work stable enough. We omit further details.

BIN/HEX In the introduced algorithms, the user-defined similarity (SIM) defines the closeness a job must fulfill to the cluster centroid to be assigned to the cluster. It is expected that low SIM values produce a small number of large but noisy clusters and a high SIM value produces a large amount of small but clean clusters. Although an optimal SIM value is depending on use case and dataset, a parameter exploration may provide important hints to find a good value and achieve optimal cluster qualities.

Figure 3 shows the number of clusters created when clustering an increasing total number of jobs for different SIM values; each point represents the number for an analyzed number of jobs in increments of 10,000 jobs. For all algorithms, we can see that with an increase in SIM value, the number of clusters created increases, and the number of total clusters created slows down the more jobs have been processed as jobs are allocated to existing clusters. For a SIM of 99%, BIN and HEX_LEV can barely group jobs together.

To understand the aggregation behavior better, alternative visualizations are investigated. In Figure 4, the number of clusters created for a given similarity

Job-IO-Utilization	Job-IO-Problem-Time	Job-IO-Balance	Binary coding
4	1	0.44	118
4	1	0.45	368:368:368:368:368:368:374:368:368:368
4	1	0.46	496:496

Table 1: IO-metrics job profiles

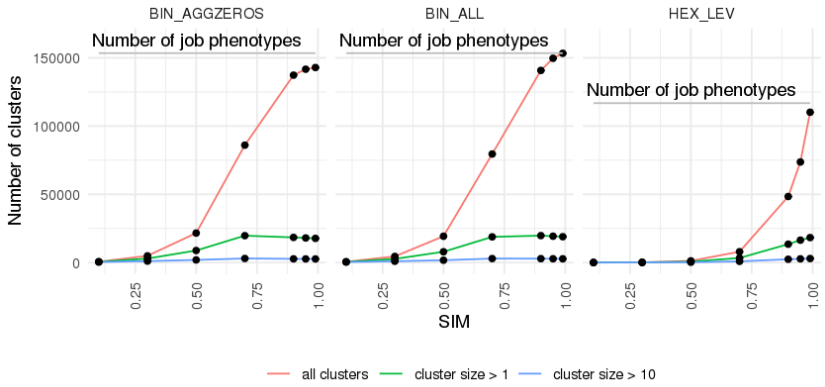

Fig. 4: Similarity value exploration.

value is plotted. The red line approximates the overall number of clusters, the green line shows how many contain at least two jobs and the blue line shows how many of them contain at least 10 jobs. On the red line we can observe increasing number of cluster with increasing SIM value, but we can also see on the green line that for the BIN algorithms the number of cluster with two jobs decreases after SIM ≥ 0.7. The maximum number of clusters is equivalent to the number of jobs; it is visualized by the gray line. Coding with 100% similarity are of the same job phenotype, i.e., they have exactly the same length and IO behavior.

This kind of investigation could help a user to find the right SIM for a particular use case. A user can read off the line generalization capabilities of the algorithm with the particular SIM value. The less clusters are created, the more job phenotype they contain in average. The green line shows the point where the algorithms begin to create job clusters with 1 job only. In some use case, this might be an unwanted behavior.

7 Use Case: Investigating an IO-Intensive Job

The demonstration in this section shows how this approach can be used to identify a cluster of IO-intensive jobs similar to an existing job.

Based on the parameter investigation, we choose the sim value by the following criteria. The BIN algorithms work best for SIM ≥ 0.7, and the HEX algorithm requires a higher SIM value, hence we chose 0.9. A further increase of the SIM value doesn't make significant improvements in our experiments.

Firstly, we determined an IO intensive job that we use to identify similar jobs. The IO intensive metric of the selected job is visualized in Figure 5. Other metrics contain only zero segments or negligible IO. We can see that this job reads data over the whole runtime. At beginning, only a subset of the nodes is reading most of the data, later more nodes participate in the reading. The amount of

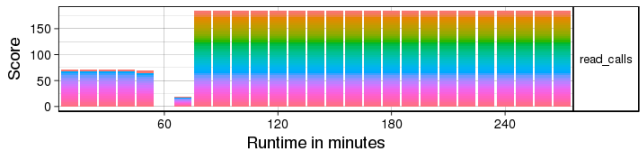

Fig. 5: IO intensive metric of one high IO intensity job running on 46 nodes. Other metrics have negligible IO and are omitted. Score is the sum of all nodes stacked by the node. A color represents the contribution of one of the nodes.

	SIM	Cluster size	Number of job types
BIN_ALL	0.7	27	17
BIN_AGGZEROS	0.7	8	8
HEX_LEV	0.9	209	189

Table 2: Cluster statistics.

Binary coding	Type
192:192:192:192:192:192:196:192:192:192:192:192:192:192:192:192:192:192:192:192:192:192:192:192:64:64:64:64:64	job
192:192:192:192:192:192:192:192:192:192:192:192:454:230:192:192:192:192:192:192:192:192:192:192:192:192:192	centroid

(a) BIN_ALL

Binary coding	Type
192:192:192:192:192:192:196:192:192:192:192:192:192:192:192:192:192:192:192:192:192:192:192:192:64:64:64:64:64	job
511:238:192:510:192:224:228:192:192:192:192:192:192:192:192:192:192:192:192:192:192:192:192:192:64:64:64:64:64	centroid

(b) BIN_AGGZEROS

Hexadecimal coding (partially) read_calls	Type
3:3:8:8:8:5:6:8:8:8:8:8:8:8:8:8:8:8:8:8:8:8:8:8:8:8:8:8:8	job
8:8:8:8:8:2:6:8:8:8:8:8:8:8:8:8:8:8:8:8:8:8:8:8:8:8:8:8:8	centroid

(c) HEX_LEV

Table 3: Job and the cluster centroid. Other jobs in the clusters are similar.

transmitted data is not large, but the amount of read calls is exceptionally high and may potentially degrade the file system performance.

The SIM value selection strategy can vary from use case to use case. As criteria we choose a SIM value that creates a moderate number of clusters (around 50% of job phenotypes) and keeps its generalization capabilities (the number of clusters with more than 1 job is considerable). For the BIN algorithms we chose a SIM of 0.7, and the HEX algorithm SIM of 0.9.

In the following, we investigate the cluster that contains this job for the different algorithms. The number of jobs found in the cluster are listed in Table 2. It shows that all algorithms find relatively small clusters. In Table 3, we can see that the jobs are relatively close to the cluster centroid. All other jobs in the clusters appear to be subjectively similar (not shown in the table). Thus, we conclude the approach generally works.

8 Summary

In this article, we applied clustering strategies to job-profile and time series of IO metrics. We conducted a short quantitative analysis to understand generalization capabilities of the algorithms and to select the parameters and conducted a qualitative analysis, i.e., manual inspection of the data to assess the quality of the approach.

After a series of experiments with general purpose algorithms, the outcome didn't meet our expectations. The investigation of resulting clusters shows that they are noisy. One problem might be the devised approach to use a clustering and a classification algorithm. It is likely that the reason is that the temporal behavior is compressed too much into the job-profile neglecting the important information.

On binary coding, the Levenshtein-based algorithms produce better clusters, especially with zero aggregation enabled. But the results are not sufficient for short jobs. Codings like [0:6:0:0] and [0:388:174:0] have the same Levenshtein distance to the centroid [0:388:0:0] but have different IO behavior.

Using the hexadecimal coding instead of binary coding leads to qualitative better results with the price that a higher similarity must be chosen. Presumably one reason is that hexadecimal coding sequences are nine times longer, which provides better conditions for the Levenshtein similarity.

Despite the suboptimal results of the algorithms when inspecting clusters, the final experiment actually shows that all the developed algorithms can actually be applied to identify jobs similar to a given job. The definition of similarity differs between these algorithms and may make them applicable to specific use cases. More research is needed to understand the needs of users and data center staff, and to define the appropriate similarity levels. We believe that the temporal pattern plays a key role in the definition of similarity as our comparison shows. In the future, we intend to refine the algorithms to account for different definitions of similarity.

References

1. Betke, E., Kunkel, J.: Benefit of DDN's IME-FUSE for I/O Intensive HPC Applications. In: Yokota, R., Weiland, M., Shalf, J., Alam, S. (eds.) High Performance Computing. pp. 131–144. Springer International Publishing, Cham (2018)
2. Carns, P.: Darshan. In: High performance parallel I/O. pp. 309–315. Computational Science Series, Chapman & Hall/CRC (2015)
3. Carns, P., Harms, K., Allcock, W., Bacon, C., Lang, S., Latham, R., Ross, R.: Understanding and improving computational science storage access through continuous characterization. ACM Transactions on Storage (TOS) **7**(3), 8 (2011)
4. Eugen Betke, J.K.: Semi-automatic Assessment of I/O Behavior by Inspecting the Individual Client-Node Timelines — An Explorative Study on 10^6 Jobs. In: 2014 43rd International Conference on Parallel Processing Workshops. ISC Events (2020)

5. Kunkel, J., Betke, E.: Tracking User-Perceived I/O Slowdown via Probing. In: High Performance Computing: ISC High Performance 2019 International Workshops, Frankfurt/Main, Germany, June 20, 2019, Revised Selected Papers. Lecture Notes in Computer Science, Springer (07 2019)
6. Kunkel, J., Betke, E., Bryson, M., Carns, P., Francis, R., Frings, W., Laifer, R., Mendez, S.: Tools for Analyzing Parallel I/O. In: Yokota, R., Weiland, M., Shalf, J., Alam, S. (eds.) High Performance Computing: ISC High Performance 2018 International Workshops, Frankfurt/Main, Germany, June 28, 2018, Revised Selected Papers. pp. 49–70. No. 11203 in Lecture Notes in Computer Science, ISC Team, Springer (01 2019). https://doi.org/https://doi.org/10.1007/978-3-030-02465-9_4
7. Lockwood, G.K., Wright, N.J., Snyder, S., Carns, P., Brown, G., Harms, K.: TOKIO on ClusterStor: Connecting standard tools to enable holistic I/O performance analysis. Tech. rep., Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States) (2018)
8. Sivalingam, K., Richardson, H., Tate, A., Lafferty, M.: LASSi: Metric based I/O analytics for HPC. CoRR **abs/1906.03884** (2019), http://arxiv.org/abs/1906.03884
9. Wang, T., Oral, S., Wang, Y., Settlemyer, B., Atchley, S., Yu, W.: Burstmem: A high-performance burst buffer system for scientific applications. In: 2014 IEEE International Conference on Big Data (Big Data). pp. 71–79 (2014)