Predicting performance of non-contiguous I/O with machine learning

Predicting Performance of Non-Contiguous I/O with Machine Learning

Julian Kunkel[1], Michaela Zimmer[2], and Eugen Betke[2]

[1] DKRZ
[2] University of Hamburg *

Abstract. Data sieving in ROMIO promises to optimize individual non-contiguous I/O. However, making the right choice and parameterizing its buffer size accordingly are non-trivial tasks, since predicting the resulting performance is difficult. Since many performance factors are not taken into account by data sieving, extracting the optimal performance for a given access pattern and system is often not possible. Additionally, in Lustre, settings such as the stripe size and number of servers are tunable, yet again, identifying rules for the data-centre proves challenging indeed. In this paper, we 1) discuss limitations of data sieving, 2) apply machine learning techniques to build a performance predictor, and 3) learn and extract best practices for the settings from the data. We used decision trees as these models can capture non-linear behavior, are easy to understand and allow for extraction of the rules used. Even though this initial research is based on decision trees, with sparse training data, the algorithm can predict many cases sufficiently. Compared to a standard setting, the decision trees created are able to improve performance significantly and we can derive expert knowledge by extracting rules from the learned tree. Applying the scheme to a set of experimental data improved the average throughput by 25-50% of the best parametrization's gain. Additionally, we demonstrate the versatility of this approach by applying it to the porting system of DKRZ's next generation supercomputer and discuss achievable performance gains.

Keywords: MPI, non-contiguous I/O, parallel I/O

1 Introduction

With MPI 2, an I/O interface has been standardized which promises to improve performance for parallel applications. Among the supported features, it explicitly supports non-contiguous I/O – one API call accesses multiple file regions, and, with collective I/O, multiple processes can coordinate their file accesses. The standard explicitly allows an implementation to exploit its knowledge about concurrent operations; for example, by scheduling the I/O calls intelligently. Since

* We want to express our gratitude to the „Deutsches Zentrum für Luft- und Raumfahrt e.V.“ as responsible project agency and to the „Bundesministerium für Bildung und Forschung“ for the financial support under grant `01 IH 11008 A-C`.

there are many factors influencing performance in a supercomputer, extracting the best performance is anything but trivial. The available optimizations offer a selection of parameters to be adapted to target machine and specific workload, and through a wrong choice, performance may be degraded.

It is very difficult for users to estimate how an I/O pattern will perform under a given set of optimization parameters; they therefore typically try various parameters, which resembles a limited brute force approach. While there are some rules of thumb and expert knowledge, e.g. "data sieving helps for small data accesses", they need to be adjusted for each system. DKRZ will install the first phase of its next generation supercomputer HLRE3 this year, providing more than 2 Petaflop/s and 30 Petabyte storage capacity. However, we struggle in the data center to determine good defaults for certain Lustre parameters such as number of servers and stripe size, as they are very specific to system and application. Even the knowledge of specialists often merely helps to direct the exploration of the complex parameter space for data sieving and stripe size. It would be helpful if expert knowledge could be automatically inferred from observations. To alleviate this, in the long run, our research strives to provide a tool that will be aware of system capabilities as well as its performance history, using all to suggest the best parameter set for the task at hand.

Our main contributions are: 1) The evaluation of decision trees to capture and predict non-contiguous performance behavior. 2) The semi-automatic extraction of expert knowledge from the measurements.

This paper is structured as follows: Related work regarding I/O research and machine learning is discussed in Section 2. Section 3 presents performance results of experiments with several relevant parameters that are currently missing in data sieving. The overall machine learning approach is introduced in Section 4. In Section 5, the accuracy of the predictor is investigated and interesting results are shown. We apply the approach to the porting system for DKRZ's next generation system in Section 6, to evaluate whether we can extract best practices for this test system; this would permit use of the strategy on the full system as well. Section 7 concludes the paper and discusses future steps.

2 Related Work

Widely used concepts to improve I/O performance are non-blocking I/O, data pre-fetching and write-behind. ROMIO [1], a common MPI-IO implementation, offers collective I/O and data sieving with the promise to speed up performance. *Data sieving* optimizes independent sparse non-contiguous I/O; by accessing larger file regions and discarding unwanted data, it avoids seeks on hard disk drives and improves performance, especially for very small accesses. While holes in the access pattern can just be read and discarded for reads, it is not as easy for writes because they require reading the whole region, modifying changed data and writing it back. Traditional file systems that offer POSIX semantics require locking to avoid conflicts with concurrent writes to an overlapping region. Ching et al.[2] implemented ListIO for PVFS in MPI which supports access to multiple file regions in one request and, thus, does not need such a read-modify-write cycle. Over the last years, there has not been much research into further optimizing non-contiguous I/O.

Optimizing collective I/O has been investigated more deeply. The basic idea of coordinated I/O is that the processes exchange information about accessed file regions; then they compute a schedule assigning responsibility for specific file regions and defining the access order; finally, data is exchanged amongst those processes that ultimately perform the I/O. There are many variations to this basic process: The Two-Phase protocol as discussed by Thakur et.al [1] iterates over communication and I/O phases – in each phase, a maximum amount of data is accessed. Multiphase-I/O [3,4] iteratively increases locality, and Orthrus [5] offers several strategies to optimize either for file or process locality. One difficulty with these approaches is that they require careful analysis and tuning of parameters.

Monitoring and analysis of system state and performance data is important to optimize HPC systems; tools include Vampir [6] and Darshan [7]. While they help in the analysis, they cannot set parameters.

There are several research projects which try to integrate machine learning into the analysis and optimization cycle. One of the first is the work of Madhyastha and Reed [8], comparing classification of I/O access patterns by feed-forward neural networks and by hidden Markov models. Higher level application I/O patterns are inferred and looked up in a table to determine the file system policy to set for the next accesses. The table, however, has to be supplied by an administrator implementing his heuristics. `Magpie`, a system by Barham et al. [9], traces events under Windows, merging them according to pre-defined schemas specifying event relationships. Their causal chains are reconstructed and clustered into models for the various types of workload observed. Deviations will point to anomalies deserving human attention. Classifying new traces according to these models yields insights into their actual and expected performance, leading to various applications such as capacity planning and on-line latency tuning, as described in [10] and [11]. Behzad et al. [12] offer a framework that uses genetic algorithms to auto-tune select parameters of a stack consisting of HDF5, MPI and Lustre. But its monolithic view of the system disregards the relations between the layers as well as the users' individual requirements, setting optimizations but once per application run.

All of these systems have in common the need for human intervention to benefit from the results or to apply the solutions to the problems identified. The SIOX framework [13,14] aims to implement a holistic approach covering the full cycle of monitoring, analysis, machine learning of the adequate settings and their automatic enactment.

3 Limits of Non-Contiguous I/O

First, we discuss the handling of non-contiguous I/O with data sieving. From the user perspective, the bytes to access are defined by the MPI file view: MPI

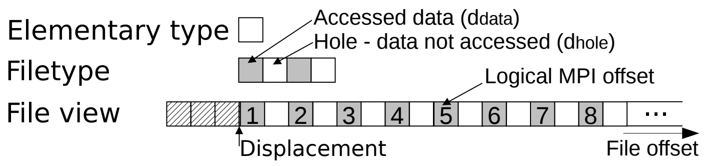


Fig. 1. Example non-contiguous access pattern in which every other elementary data type is accessed.

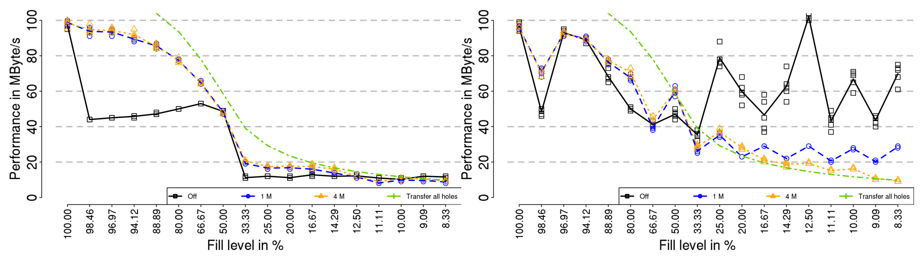
data types are used to describe the elementary data type (etype) the file consists of (e.g. a structure), and the file type specifies which of those are to be accessed by the current process. A programmer usually accesses data at the granularity of etypes. An offset (the so-called displacement) is added to the beginning of the file to account for e.g. file meta-data. An example file view and mapping of its data to bytes in the file is illustrated in Figure 1. Here, the etype could be a 4-Byte integer; the file type covers four contiguous regions (first etype is occupied, then a hole, ...) and allows access to every other etype. We will refer to the number of bytes one contiguous region occupies as d_{data}, and to the hole size as d_{hole}. The data sieving algorithm is parameterized by its *state* (s: on, off) and a *buffer size* (s_{buffer}) which defines the granularity of data access for reading and writing. It will access data at this size, starting from the bytes needed to be accessed next that are not contained in the current buffer.

3.1 Experiments

To demonstrate the suboptimality and to illustrate the difficulties in parameterizing the current data sieving strategy, we conducted several experiments. The `mpipattern` benchmark has been created to measure performance for arbitrary file views and MPI hints. In the following experiments[3], this benchmark uses a file pattern similar to Figure 1; the etype is always an integer and we vary d_{data} between 1 KiB and 16 MiB, the data sieving options (s, s_{buffer}) and the *fill level* $f := \frac{d_{\text{data}}}{d_{\text{data}}+d_{\text{hole}}} = \frac{d_{\text{data}}}{d_{extent}}$.

The experiments have been conducted on our 20 node cluster: 10 I/O nodes are each equipped with an Intel Xeon E3-1275@3.4 GHz, 16 GByte RAM and one Seagate Barracuda 7200.12. Nodes are interconnected with Gigabit Ethernet and the performance of one HDD is about 100 MiB/s. The I/O nodes run CentOS 6.5 and Lustre 2.5. On one additional compute node, a single `mpipattern` process is run which reports the observed performance. In a production environment, multiple users and applications access the shared storage; this may lead to high fluctuations in observable performance. For a first discussion, this effect is ignored; during the measurement, the whole cluster has been blocked to ensure exclusive access to the I/O servers. The test file is pre-created with 8 GiB of data; between runs, we clear the Linux cache. In the following, we limit our discussion to read calls.

[3] Experimental data is taken from Schmidtke's thesis [15].



(a) $d_{\text{data}} = 16\,\text{KiB}$ **(b)** $d_{\text{data}} = 256\,\text{KiB}$

Fig. 2. Independent I/O performance for a variable hole size and two block sizes measured with one client for different data sieving options.

Overall, out of 198 different configurations, the best result is achieved without data sieving in 56 cases, and in 59 and 54 cases with $s_{\text{buffer}} = 1\,\text{MiB}$ and $s_{\text{buffer}} = 4\,\text{MiB}$, respectively. A 100 MiB buffer never achieved best performance, even for larger datasets and holes. In 29 cases, performance of all settings was similar (relative performance within 95% of the best). Figure 2a and Figure 2b show the observed performance for $d_{\text{data}} = 16$ and 256 KiB, respectively. The lines in the figures represent the performance with data sieving turned off, or its buffer set to 1 MiB or 4 MiB of data. Additionally, a theoretic line is given: it is based on the maximum network performance (117 MiB/s) and assumes all holes would be read; e.g., for a fill level of 10%, user data is transferred at 11.7 MiB/s.

For small blocks, data sieving performs better, because it avoids seeks and the system benefits from the Linux read-ahead mechanism. The effects of read-ahead can be seen by comparing performance from fill levels f of 100% and 98% ($d_{\text{hole}} = 160$ Byte). With a low fill level and thus large holes, data sieving actually slows down the operations; the reason is that it actually reads the full buffer size for every required data block even though we only need the first 16 KiB of data. When accessing 256 KiB of data, this strategy also explains the effect starting at $f = 16\%$: The 4 MiB buffer is much slower than the 1 MiB buffer. However, a user would expect that larger buffers may increase performance but never decrease it.

Moreover, for the large access granularity, turning data sieving off is beneficial starting with $f = 33\%$. For larger accesses, data sieving extracts similar performance in many cases. Several interesting effects can be seen: performance of $f = 98\%$ ($d_{\text{hole}} = 8\,\text{KiB}$) is slower; with $f = 66\%$ ($d_{\text{hole}} = 128\,\text{KiB}$), performance converges; and with smaller fill levels, some values attain much better performance. The 128 KiB hole can be explained by Linux read-ahead mechanisms: normally, another 128 KiB block of data is fetched, which is available in any case. We are striping in 1,048,576 Byte blocks; due to the layout, every single 256 KiB access is covered completely by one Lustre object storage target (OST). In these cases, with data sieving, another performance pattern can be observed.

Note that for a 1 MiB buffer, each I/O involves one additional OST, all of the requested data of which is discarded.

Presumably, the reason for the zig-zag pattern for small fill levels without data sieving is the OST-centric read-ahead in Lustre. Several patterns lead to sequential access of data on a subset of servers, such as for 25% and 12.5%; here, data is read from every (or every other) OST in a sequential fashion. The `read_ahead_stats` from `/proc` reveal that per 256 KiB access, about 0.03 and 0.7 cache misses occur for the very good and bad cases, respectively. The `osc_read` shows about 0.77 to 1.75 operations per access; thus, some patterns trigger more operations than others.

3.2 Performance Factors

There are many factors involved in the performance of non-contiguous I/O that can be classified into the applications' spatial and access pattern, the behavior of file system client and parallel file system, and hardware characteristics. Each individual I/O operation comes with some overhead for the system call and transferring and processing the triggered I/O request within the parallel file system. If data is not cached, the operation is dominated on the server side by the latency of the block storage. Aggregating multiple non-contiguous accesses into one operation alleviates these costs but may transfer irrelevant data from block devices and across the network, and thus benefit depends on the throughput of these components. The file's data distribution (stripe size in Lustre and number of servers) has a big impact on performance, as it should be avoided to involve too many I/O servers with very small requests. Therefore, the alignment of the accessed data on the file system's server is important. An additional factor is the cost for distributed locking needed for writes. The operating system's and file system's pre-fetching mechanisms can transform some read patterns automatically into beneficial sequential access patterns without explicitly requesting large chunks at application level.

The decision whether or not to merge a consecutive operation with the current operation depends on the knowledge of these factors; the best choice may fuse certain blocks and process others individually. As none of these factors are explicitly included in ROMIO's data sieving, and the buffer sizes can only be changed when opening the file, this approach is hard to tune for users and the achieved performance is often suboptimal. Therefore, machine learning may be a suitable technique to analyze the data.

4 Methodology

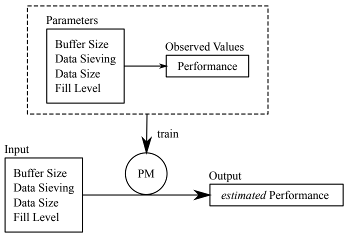
Every approach to optimization will consist of three basic steps: Identifying the task's fixed parameters, choosing the best set of variable parameters and suggesting or enacting them. While we aim to perform the machine learning with the execution of the application (online), in this study, we measured the performance and investigate the machine learning offline to evaluate the accuracy.

In our use case, the fixed parameters consist of the access pattern, specified by a sequence of (offset, size) tuples (cf. Figure 1). As this may constitute a sequence of finite but unbounded length, we use a simple first abstraction, computing only the total *size* d_{data} of each data type and its *fill level* f. Further research will target more accurate representations and characteristics of the resulting parameter spaces.

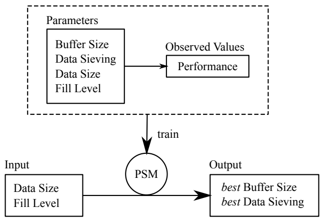
Our variable parameters are the *state* of data sieving (*on*, *off*) and the *buffer size* s_{buffer} used for it. Our optimization criterion is the performance p, the average (arithmetic mean) throughput achieved under the `mpipattern` benchmark. The data to be used in training and validation was gathered by running the benchmark five times per parameter set, then the performance's arithmetic mean is computed for each configuration. The relevant variable parameters and target labels are stored in a CSV file. The evaluation is conducted by loading the observed performance data into the statistics tool R. We then create the models offline and compare their performance to the best achievable performance. Since the observed performance data volume is small (CSV files of roughly 100 KiB), the time needed for machine learning is negligible in the analysis.

We use standard machine learning techniques to extract knowledge from the data. For our first method to evaluate, we chose Classification And Regression Trees (CART)[16], as implemented in the open source library Shark [17], the statistics tool R and the language python. Our first step is to create a predictor for the performance to be expected from a given set of fixed and variable parameters. This Performance Model (PM) is trained on a number of samples, allowing it to estimate a performance value for any given parameter set (see Figure 3a). For this model, the CSV file contains: *size, fill level, state, buffer size* and *arithmetic mean performance.* We train the model using a subset of rows in the CSV file (the training set) and predict the performance for the validation set. Since the mean performance of the data is available for the validation set, we can determine the error.

As not all machine learning algorithms are suited to regression, we transformed this task into a classification problem which allows for a full comparison later on. For this, we form classes by quantizing the performance space into intervals, similar to the "shingles" used in the `R` package `lattice`; parameter sets are classified by mapping them to the interval "class" covering the achieved performance p. For every parameter set thus classified, a representative of the pertinent interval is then chosen as a performance estimate. Since our interval partition is ignorant of the true performance's distribution, we chose the intervals' middle points as representatives to facilitate error bound assertions. Using the median of the values classed within each interval might decrease actual errors as it better approximates clusters within the interval, though. With this set-up, however, uniform intervals are imprecise in the lower ranges, while small relative variations in the higher ranges will mean several classes displacement. We therefore vary interval length with the absolute values they cover: Given a relative error limit ϵ and the maximum performance measured $p_{\max}$, we define $l := \epsilon \cdot p_{\max}$. Between 0 and l, we choose uniform interval lengths $|I_i| := l \cdot 2\epsilon$;



(a) Performance Model



(b) Parameter Setting Model

Fig. 3. PM provides a performance estimate, whereas PSM provides the corresponding “tunable” variable parameters to achieve it.

above l, we increase them stepwise by $|I_i| := |I_{i-1}| \cdot (1 + 2\epsilon)$. This implies the quantization mean error $err_q := \frac{1}{4n} \sum_{i=0}^{n} (\Delta I_i)$, where ΔI_i is the interval size of the class that belongs to the i-th instance.

We imagine an implementation could automatically tune the data sieving parameters online by estimating the performance for all sets of variable parameters and picking the values expected to perform best. This strategy has the advantage that we can validate the prediction accuracy online by comparing estimation with measurement, and disable the predictor if the results differ significantly from the observation.

However, this strategy requires us to assess performance for many different settings. Instead, we chose a complementing strategy to directly predict the variable settings for a given set of fixed parameters; we call this the Parameter Setting Model (PSM) Figure 3b. Since the performance data is still available to us, we can also quantify the efficiency of this model in our evaluation. Note that in an implementation, the PSM could use the performance prediction of the PM to check its correctness.

5 Evaluation

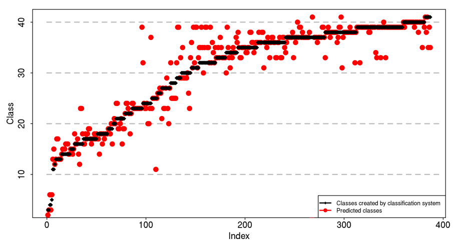
To assess the quality of the machine learning algorithms, we created simpler models and use them as a baseline: A very naive prediction for a sample would be the arithmetic mean performance. In our experiments, the mean performance is 54.7 MiB/s, which leads to an average error of 28.5 MiB/s. Experimenting with different linear models based on the fixed and variable settings led to a model with a mean error of 12.7 MiB/s.

5.1 Validation

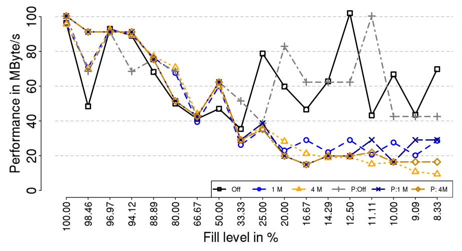
A series of k-fold cross-validation tests (Table 1) shows that on our data set, the CART classifier performs better than the our baseline. Unless noted otherwise, all results cited in this section have been generated with the following parameters: size of training set = size of validation set = 387 instances. Classification parameter: $\epsilon = 0.05$, $p_{max} = 109.554$.

k	Performance errors			Class errors		
	min	mean	max	min	mean	max
2	6.74	6.80	6.87	1.46	1.59	1.72
4	5.19	6.25	6.92	0.94	1.34	1.72
8	4.67	5.66	6.77	0.87	1.19	1.62

Table 1. Prediction errors in MB/s and class errors for training sets under k-fold cross-validation. Values for k=3..7 lie in between.



(a) CART prediction (trained by 387 instances).



(b) Performance prediction for $d_{\text{data}} = 256\,\text{KiB}$.

Fig. 4. Quality of PM performance prediction.

Figure 4a shows the observed and predicted classes when using half the training data. The graph is sorted by the true performance class (black dots) and the red dots show the predicted classes. The actual performance prediction for $d_{\text{data}} = 256\,\text{KiB}$ are presented in Figure 4b. Often, a predicted performance matches one of the nearby observed values; the reason is that the original data point is not contained in the training set and thus the model learns from nearby values and uses them as approximation. Clearly, the sensitivity of the pattern and thus major performance differences are impossible to predict accurately if instances are missing.

5.2 Investigating Training Set Size

We are working towards a self-optimizing system that stops the optimization process as soon as some convergence criterion is fulfilled, e.g. the learning rate is negligible or the error rate small enough. This bypasses the need for rules or a static formula to calculate an optimal training set size, allowing us to replace learning algorithms and apply this approach to a variety of problems without interdependency with our data acquisition scheme.

Nevertheless, we have investigated the prediction accuracy of PM under various training set sizes, using a variant ("inverse") k-fold cross-validation where one fold is used for training and the remaining $k-1$ for validation instead of the other way round. The results of the CART classifier are shown in Figure 5. The 774-instances case validates the overall scheme: The CART classifier was both trained and validated with the whole data set, yielding a prediction mean

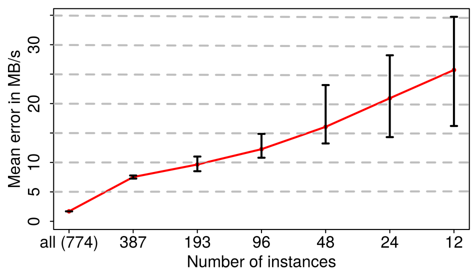


Fig. 5. Mean prediction error of PM by training set size under inverse k-fold cross-validation. Class prediction errors show very similar behavior.

error of 1.7 MB/s. Deducting the quantization error of 1.31 MB/s due to our assignment to classes leaves the real CART classifier error at 0.39 MB/s.

Beginning with the full data set, we reduce the number of training instances by a factor of 1/2 in each iteration step until the CART classifier stops producing reasonable predictions. By training with 24 instances, we can recognize the first learning progress: the maximum prediction error drops beneath the raw data's standard deviation. 96 instances were sufficient to outperform the naive approach, and after 387 instances, we could observe a considerable stagnation of the learning rate (cf. Table 1). Moving to random forests[18] yielded very similar results, not justifying the additional computational cost incurred.

The potential benefit of the approach can be assessed by applying the strategy to the experimental data. Assuming the user had parameterized the data sieving for all experiments in the same way, the average performance benefit of choosing the optimal values instead is given in Table 2. As a default, setting data sieving to 1 MiB would yield the best result, as even optimal parameter settings outperform it by at most 7.6 MB/s. But even here, our CART-driven PSM with training and test set sizes of 387 instances each could improve performance by 1.9 MB/s.

5.3 Decision Rules

By classifying only into three classes (slow, average, fast), the CART classifier applied to the complete data set of mean performance values creates a tree of 221 nodes; the first 4 levels are shown in Figure 6. The following analysis relies on the fact that the test cases cover the selected parameter space equally. Based

Default Choice	CART PSM, 387 Inst.	Best Choice
Off	4.2 MB/s	9.6 MB/s
1 MiB	1.9 MB/s	7.6 MB/s
4 MiB	6.9 MB/s	12.2 MB/s
100 MiB	6.9 MB/s	12.2 MB/s

Table 2. Average performance improvements that can be achieved with the PSM-learned and best choices for s_{buffer}, compared to one default choice.

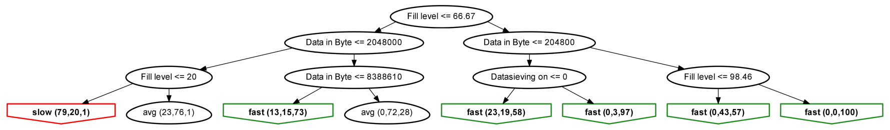


Fig. 6. First three levels of the CART classifier rules for three classes slow, avg, fast ($[0, 25]$, $(25, 75]$, > 75 MB/s). The dominant label is assigned to the leaf nodes – the probability for each class is provided in brackets.

on the figure, some rules can be derived: e.g., the left-most path tells us that of non-contiguous accesses with $f < 20\%$ and $d_{\text{data}} = 2$ MiB, 79% will show slow and 20% average behavior. Other rules thus created include: in most cases, sparse access to larger blocks is slow; sequential access to small blocks is fast (due to read-ahead); for $f > 2/3$ and $d_{\text{data}} < 200$ KiB, data sieving is beneficial for almost all accesses; in the case of $f < 2/3$ and $2\,\text{MiB} < d_{\text{data}} < 8\,\text{MiB}$, performance is mostly fast – surprisingly, larger accesses achieve merely average performance. Experts in the field typically know the first rules, but the last two statements are interesting.

6 Learning Best Practices for DKRZ

DKRZ runs a test system to prepare for their next supercomputer that will be installed in Q1 of 2015. We conducted measurements on this porting system to study whether our methodology can be applied to learn appropriate Lustre settings. The test system consists of 20 compute nodes and a Lustre 2.5 file system hosted by one ClusterStor 6000 enclosure (SSU) from Seagate with two OSS servers and 84 HDDs. All nodes are interconnected with FDR-Infiniband.

The following measurements are conducted with our NCT library which is currently in development and offers POSIX-compatible calls for non-contiguous access. Amongst other strategies, it implements the ROMIO algorithm for data sieving, allowing for an analysis similar to the one discussed before: A single process performs reads or writes on a previously created 10 GB file, varying hole size and access granularity. As opposed to the results discussed so far, this evaluation is conducted on the file system shared amongst all users. To gain comparable results, the client cache is cleaned between the runs, and several repetitions are measured. If a value differs more than 20% from the average of all others measured so far for this configuration, an additional run is executed after which this procedure is repeated, resulting in up to 10 measurements for cases with high variation.

Overall, 408 configurations of hole and block size were measured for up to 8 combinations of user controllable settings (one or two Lustre servers, 128 KiB or 2 MiB stripe size, data sieving with 4 MiB or off). 240 of these were run with all 8 settings; of the remainder, 84 more cases each were only evaluated for 128 KiB and 2 MiB stripe sizes, respectively. For validation purposes, the two

Data sieving			Off				On			
Server count			1		2		1		2	
Stripe size			128 KiB	2 MiB	128 KiB	2 MiB	128 KiB	2 MiB	128 KiB	2 MiB
Sieving	Server #	Stripe								
Off	1	128 KiB	-	0	0	0	31	31	33	31
		2 MiB	2	-	0	0	31	32	34	32
	2	128 KiB	54	57	-	3	43	43	35	37
		2 MiB	56	59	2	-	39	45	42	37
On	1	128 KiB	114	115	109	93	-	5	8	19
		2 MiB	104	103	90	83	0	-	3	14
	2	128 KiB	112	112	104	107	65	71	-	8
		2 MiB	112	111	104	96	56	69	2	-

Table 3. Frequency in which a setting of the row is better by 20% (at least 5 MB/s) than the one of a column, out of 240 hole/size configurations.

settings of stripe sizes were also evaluated for one server, which should not make a difference.

First, we look at the frequency at which a particular setting is superior to another. Table 3 shows the number of times the configuration in a row achieves at least 20% and 5 MB/s better performance than the one in the column. A few unexpected cases are observable: when only one server is used, the variation in stripe sizes should lead to similar performance results. However, e.g. without data sieving, the 2 MiB stripe size is in 2 (out of 240) cases better than the one with 128 KiB which is presumably due to fluctuations on the shared storage resource. Without data sieving, it can be seen that typically one server achieves less performance than two; with data sieving, there are some cases in which one server significantly outperforms two. In the sampled configurations, turning data sieving on is usually superior to turning it off in about 100 cases, the naive I/O is better in about 35 cases. While the configurations are similar to the ones measured on our test cluster, the amount of data accessed is typically small compared to the fast interconnect and storage system, which explains why data sieving dominates the naive approach. With 20% tolerance, there are a few cases in which the stripe size is relevant; reducing tolerance to 10%, this number rises to about 50. Nevertheless, we expect that this number will grow much higher on DKRZ's final system than in this preliminary experiment.

6.1 Applying Machine Learning

In the following analysis, used the scikit-learn Python library with its DecisionTreeClassifier (with its entropy criterion) on the CSV file to learn the decision tree and extract knowledge. The triple (sieving, server count, stripe size) of the best possible choice for each configuration is encoded as an integer and learned. Note that we treat all configurations equally – in a real system, each would be weighted based on the probability of observing it, making sure that frequent access patterns will be well optimized.

Looking at some statistics of the achieved performance allows us to quantify the optimization potential: The best observed performances for a single run are up to 800 MB/s and 350 MB/s for read and write operations, respectively. Over the 240 configurations, an average performance of 213 MB/s is observable. The average performance over all configurations, choosing the best setting for each, is 293 MB/s; choosing the worst for each, it is 146 MB/s. Creating a decision tree of depth 1 yields the rule *if (write) select (data sieving=on, servers=2, stripe=128 KiB) else select (data sieving=on, servers=1, stripe=128 KiB)*. Following even this simple rule reduces the gap in average performance compared to the best per-case choice possible to only 16.6 MB/s.

In practice, we will not normally have all settings sampled for a given configuration, resulting in missing values similar to our case with 408 configurations. Using pruned trees with reduced height however, as in this evaluation, rules may still suggest settings that have not been measured so far, and if this recommendation is followed, the sampled portion of the parameter space will grow in the long run. Using all values, the average performance over all measured configurations and settings is 244.7 MB/s. The best setting for each configuration achieves an average performance of 357.7 MB/s, and the worst choice of 179.9 MB/s. Table 4 lists the average performance loss of a given default choice when compared to the best available choices.

When averaging test run performance, two scenarios may apply: Computing centers desire a continually saturated job queue, where the mean achieved over a fixed time is of interest. Users, who typically have a fixed workload to be completed, regard the mean derived from the total time to completion as more important. The first is given by the arithmetic mean, while the harmonic mean yields the second. Another interpretation of the arithmetic columns is the expected performance when picking a random experiment, while the harmonic performance defines the average throughput expected when executing all experiments with the given setting. The arithmetic mean favors fast execution,

Default Choice			Best	Worst	Arithmethic Mean			Harmonic Mean	
Servers	Stripe Size	Sieving	Freq.	Freq.	Rel.	Abs.	Loss	Rel.	Abs.
1	128 KiB	Off	20	35	58.4%	200.1	102.1	9.0%	0.09
1	2 MiB	Off	45	39	60.7%	261.5	103.7	9.0%	0.09
2	128 KiB	Off	87	76	69.8%	209.5	92.7	8.8%	0.09
2	2 MiB	Off	81	14	72.1%	284.2	81.1	8.9%	0.09
1	128 KiB	On	79	37	64.1%	245.6	56.7	15.2%	0.16
1	2 MiB	On	11	75	59.4%	259.2	106.1	14.4%	0.15
2	128 KiB	On	80	58	68.7%	239.6	62.6	16.2%	0.17
2	2 MiB	On	5	74	62.9%	258.0	107.3	14.9%	0.16

Table 4. Tunable settings: Expected performance of a user's default choice vs. the per-case optimal setting (absolute in MB/s, relative and performance loss in MB/s compared to the best choice) using arithmetic and harmonic mean. The number of cases in which a setting is the worst or best choice out of all 408 configurations is listed for reference.

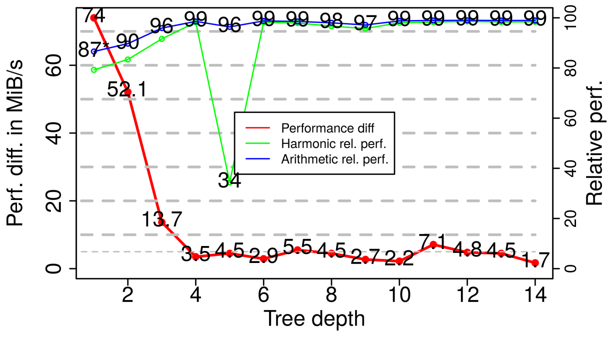


Fig. 7. Performance difference between learned and the best choices, by maximum tree depth, for the DKRZ porting system.

the harmonic mean is restricted by slow performance. With one server, 2 MiB stripe size and activated data sieving, for instance, 64.1% of the best possible performance is expected for any run and the arithmetic mean performance loss compared to the optimal settings is 57 MB/s (achieving 245.6 MB/s). However, when executing all experiments with this setting, only 15.2% of the best performance is expected resulting in an average harmonic mean performance of only 0.16 MiB/s. This low harmonic mean is due to experiments with large holes and small amounts of data achieving a performance below 1 MB/s. By choosing the optimal tunable settings, the achievable performance can thus be significantly increased, which further underlines the relevance of this early study.

Figure 7 shows the average performance loss between machine learning and the per-case optimum, based on the depth of the tree learned. The figure also includes the relative performance achieved, compared as harmonic and arithmetic means. Even at a very low height, the tree proves very efficient, achieving more than 87% of the arithmetic mean performance and 79% of the harmonic[4]. This is much better than all possible fixed defaults (72% arithm. mean and 15% harmonic mean performance at best). Therefore, the trees avoid suboptimal choices efficiently. One exception is the tree with a depth of 5: it suggests several slow settings, resulting in a relative harmonic mean performance of 34%.

A tree of level four (shown as in Figure 8) achieves good performance (about 3.5 MB/s average gap) at a reasonable size; it can be expected to achieve 99% of the potential performance (arithmetic and harmonic). The leaves are the choices based on the access pattern. The number of instances in which this choice is the best is given in the leaf for convenience, followed by the second best choice. Interestingly, in most cases, both differ only in a single parameter, i.e., either number of servers, data sieving or stripe size. Given an access pattern, this tree

[4] Note that for a tree of depth one, 80 choices are made for which no measurement is available; these values are excluded from the calculation of the average performance. For bigger trees, less than a handful of choices are not quantifiable. Therefore, we believe this comparison to be fair.

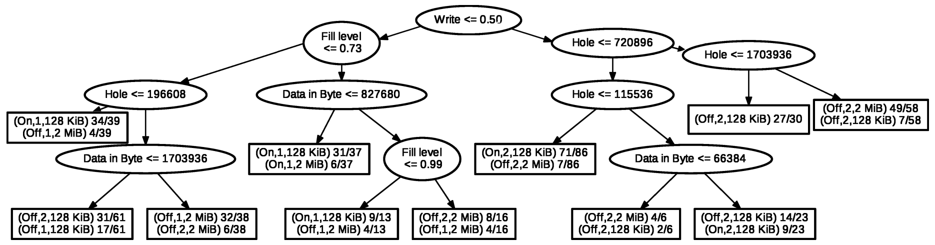


Fig. 8. Decision tree for DKRZ test system with height 4. In the leaf nodes, the settings (Data sieving, server number, stripe size) and number of instances of the best and second best choice are shown.

allows users (or a library) to select appropriate and efficient settings. Also, using machine learning and extracting rules from such a tree proved far less time consuming and error-prone than studying the measurement results by hand.

7 Conclusions and Future Work

Even constrained to the few parameters governing data sieving, optimizing HPC I/O is anything but trivial. We have discussed the challenges and limitations faced when optimizing non-contiguous access using data sieving, and used Classification and Regression Trees to create a predictor for the I/O performance resulting from a given parameter set. Evaluating this predictor under various training set sizes, we found it a fairly accurate indicator of the performance to be expected. We created another model that will choose the parameter set promising the highest performance, achieving significant improvements over the best default settings and increasing the average I/O performance by several MiB/s. While the decision trees reproduced known heuristics correctly, we also harvested interesting insights from them, yielding best practices for data sieving on our system.

Future work will focus on automatically generating simple rules-of-thumb from the extensive decision trees. Integrating our findings with the SIOX system will allow us to harness this knowledge for optimization as well as for active learning during phases of low utilization. Thus, sparse training data can be supplemented to greatly improve predictor accuracy and overall effectiveness. Since data sieving does not incorporate the important performance factors, observed performance behaves unpredictably in many cases, leading to suboptimal accuracy of the CART when using sparse training data. The parameters discussed in this paper are system dependent, but not affected by the file type and pattern, marking them as candidates for machine learning. We are currently working on an adaptive data sieving algorithm relying on this, and researching more accurate representations and characteristics of the resulting parameter spaces. Finally, future efforts will further explore ML techniques and their applicability, as well as the effects of selective data acquisition and active learning.

References

1. Thakur, R., Gropp, W., Lusk, E.: Data Sieving and Collective I/O in ROMIO. In: FRONTIERS '99: Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation, Washington, DC, USA, IEEE Computer Society (1999) 182
2. Ching, A., Choudhary, A., Coloma, K., Liao, W.k., Ross, R., Gropp, W.: Noncontiguous I/O Accesses Through MPI-IO. In: Proceedings of the 3st International Symposium on Cluster Computing and the Grid. CCGRID, Washington, DC, USA, IEEE Computer Society (2003) 104–
3. Singh, D.E., Isaila, F., Calderon, A., Garcia, F., Carretero, J.: Multiple-Phase Collective I/O Technique for Improving Data Access Locality. In: Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing. PDP, Washington, DC, USA, IEEE Computer Society (2007) 534–542
4. Singh, D.E., Isaila, F., Pichel, J.C., Carretero, J.: A Collective I/O Implementation Based on Inspector–Executor Paradigm. The Journal of Supercomputing **47**(1) (2009) 53–75
5. Zhang, X., Ou, J., Davis, K., Jiang, S.: Orthrus: A framework for implementing efficient collective i/o in multi-core clusters. In: Supercomputing, Springer (2014) 348–364
6. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The Vampir Performance Analysis Tool-Set. In: Tools for High Performance Computing, Proceedings of the 2nd International Workshop on Parallel Tools, Springer (2008) 139–155
7. Argonne National Laboratory: Darshan. http://www.mcs.anl.gov/project/darshan-hpc-io-characterization-tool
8. Madhyastha, T., Reed, D.: Learning to Classify Parallel Input/Output Access Patterns. Parallel and Distributed Systems, IEEE Transactions on **13**(8) (August 2002) 802–813
9. Barham, P., Donnelly, A., Isaacs, R., Mortier, R.: Using Magpie for Request Extraction and Workload Modelling. Proceedings of the 6th Symposium on Opearting Systems Design and Implementation **6** (2004) 259–272
10. Barham, P., Isaacs, R., Mortier, R., Narayanan, D.: Magpie: Online modelling and performance-aware systems. In: Proceedings of the 9th conference on Hot Topics in Operating Systems. Volume 9. (2003)
11. Isaacs, R., Barham, P., Bulpin, J., Mortier, R., Narayanan, D.: Request extraction in magpie: events, schemas and temporal joins. In: Proceedings of the 11th workshop on ACM SIGOPS European workshop. EW 11, New York, NY, USA, ACM (2004)
12. Behzad, B., Huchette, J., Luu, H.V.T., Aydt, R., Byna, S., Yao, Y., Koziol, Q., Prabhat: A framework for auto-tuning hdf5 applications. In: Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing. HPDC '13, New York, NY, USA, ACM (2013) 127–128
13. Kunkel, J., Zimmer, M., Hübbe, N., Aguilera, A., Mickler, H., Wang, X., Chut, A., Bönisch, T., Lüttgau, J., Michel, R., Weging, J.: The SIOX Architecture – Coupling Automatic Monitoring and Optimization of Parallel I/O. In: Supercomputing. Number 8488 in Lecture Notes in Computer Science, Berlin, Heidelberg, Springer (06 2014) 245–260

14. Zimmer, M., Kunkel, J., Ludwig, T.: Towards Self-optimization in HPC I/O. In: Supercomputing. Number 7905 in Lecture Notes in Computer Science, Berlin, Heidelberg, Springer (06 2013) 422–434
15. Schmidtke, D.: Analyse und Optimierung von nicht-zusammenhängende Ein-/Ausgabe in MPI (04 2014)
16. Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J.: Classification and Regression Trees. Wadsworth & Brooks, Pacific Grove, California (1984)
17. Igel, C., Heidrich-Meisner, V., Glasmachers, T.: Shark. Journal of Machine Learning Research **9** (2008) 993–996
18. Breiman, L.: Random forests. Mach. Learn. **45**(1) (October 2001) 5–32