Adaptive Request Scheduling for the I/O Forwarding Layer using Reinforcement Learning

Jean Luca Bez, Francieli Zanon Boito, Ramon Nou, Alberto Miranda, Toni Cortes, Philippe Navaux

Adaptive Request Scheduling for the I/O Forwarding Layer using Reinforcement Learning

Jean Luca Bez[1,3], Francieli Zanon Boito[2], Ramon Nou[3], Alberto Miranda[3], Toni Cortes[3,4], Philippe Navaux[1]

[1]Institute of Informatics, Federal University of Rio Grande do Sul (UFRGS) — Porto Alegre, Brazil
[2]LaBRI, Université de Bordeaux, Inria, CNRS, Bordeaux-INP — Bordeaux, France
[3]Barcelona Supercomputing Center (BSC) — Barcelona, Spain
[4]Universitat Politècnica de Catalunya — Barcelona, Spain

{jean.bez, navaux}@inf.ufrgs.br, francieli.zanon-boito@u-bordeaux.fr, {ramon.nou, alberto.miranda}@bsc.es, toni@ac.upc.edu

***Abstract*—I/O optimization techniques such as request scheduling can improve performance mainly for the access patterns they target, or they depend on the precise tune of parameters. In this paper, we propose an approach to adapt the I/O forwarding layer of HPC systems to the application access patterns by tuning a request scheduler. Our case study is the TWINS scheduling algorithm, where performance improvements depend on the time window parameter, which depends on the current workload. Our approach uses a reinforcement learning technique – contextual bandits – to make the system capable of learning the best parameter value to each access pattern during its execution, without a previous training phase. We evaluate our proposal and demonstrate it can achieve a precision of 88% on the parameter selection in the first hundreds of observations of an access pattern. After having observed an access pattern for a few minutes (not necessarily contiguously), we demonstrate that the system will be able to optimize its performance for the rest of the life of the system (years).**

***Index Terms*—Auto-tuning, Reinforcement Learning, Parallel I/O, I/O forwarding, I/O scheduling**

I. Introduction

Scientific applications impose ever-increasing performance requirements to the High-Performance Computing (HPC) field. These justify the continuous upgrades and deployment of new large-scale platforms. Such massive platforms rely on a shared storage infrastructure which is built over a dedicated set of servers, powered by a Parallel File System (PFS) such as Lustre [1], PVFS (OrangeFS) [2] or Panasas [3]. However, if all the compute nodes were to access the same shared file system servers concurrently, the contention would compromise the overall performance and cause interference [4], [5].

To alleviate this contention problem, the I/O forwarding technique [6] aims at reducing the number of clients concurrently accessing the file system servers. It defines a set of *I/O nodes* that are in charge of receiving I/O requests from the processing nodes and of forwarding them to the PFS. This technique is applied in several of the Top 500 machines[1], as detailed by Table I. Besides alleviating contention, I/O forwarding creates an additional layer between applications and file system, that can be used for optimizations such as request reordering, aggregation, and I/O request scheduling [5], [7].

[1]The June 2019 TOP500 list: https://www.top500.org/lists/2019/06/.

We call the "access pattern" the way the application perform I/O operations: spatiality (contiguous, 1D-strided, etc), request size, number of files, etc. I/O optimization techniques (including but not limited to the I/O forwarding layer) typically provide improvements for specific system configurations and application access patterns, but not for all of them. Moreover, they often depend on the right choice of parameters (for example, the size of the buffer for MPI-IO collective operations). That happens because they are designed to explore specific characteristics of systems and workloads [8]. That was demonstrated to be the case for request scheduling at different levels [9], [10]. Therefore, to successfully improve performance, it is essential to dynamically adapt the system to a changing workload. A caveat is that the I/O stack is typically stateless, and does not have information about the applications' behaviors until accesses are happening.

We propose a novel approach to adapt the I/O forwarding layer to the observed I/O workload. In our proposal, a *Council* periodically receives metrics on the access pattern collected by the I/O nodes. A reinforcement learning technique, contextual bandits [11], is used so that the system can learn the best choice to each access pattern. After observing a pattern enough times, the acquired knowledge will be used to improve its performance during the whole life of the system (years). By making the system capable of learning, instead of using a supervised technique, we eliminate the need for a previous training step, and without adding a high overhead (median $< 2\%$). That is essential as designing and executing a training set to represent the diverse set of applications that will run in a supercomputer, and the interactions between concurrent applications (over the shared I/O infrastructure), is difficult, error-

TABLE I
Some of the TOP 500 machines using I/O forwarding techniques

Rank	Supercomputer	Compute Nodes	I/O Nodes
3	Sunway TaihuLight	$40,960$	240
4	Tianhe-2A	$16,000$	256
6	Piz Daint	$6,751$	54
7	Trinity	$19,420$	576
12	Titan	$18,688$	432
13	Sequoia	$98,304$	768

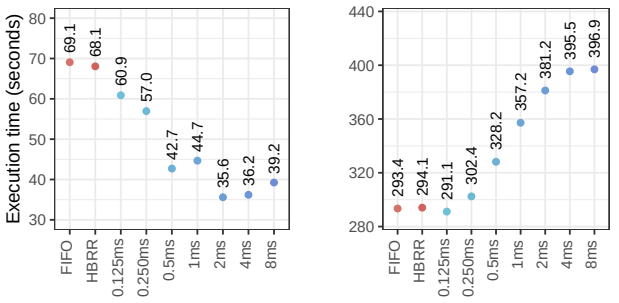
prone, and time-consuming. Finally, the continuous learning process enables the system to adapt to workload changes and system upgrades.

In this paper, we demonstrate our proposed technique by applying it to a study case: the TWINS request scheduling algorithm for the I/O forwarding layer [9]. Nevertheless, our approach can be used to tune other optimization techniques that depend on the access pattern. We present TWINS in Section II. We discuss the viability of learning and adapting in HPC machines in Section III. Section IV presents our reinforcement learning approach. The results are described in Section V. Finally, Section VI discusses related work and Section VII concludes this paper and addresses future work.

II. Case study: the TWINS scheduling algorithm

The main goal of TWINS [9] is to coordinate the I/O nodes' accesses to the shared PFS servers to mitigate contention. Each I/O node keeps multiple request queues, one per data server. During a configurable *time window*, requests are taken (in arrival order) from only one of the queues, to access only one of the servers. When the time window ends, the scheduler moves to the next queue following a round robin scheme. If $server\_i$ is the current server being accessed, but there are no requests to it, the scheduler will wait until either requests to $server\_i$ arrive or the time window ends, even if there are queued requests to other servers. To ensure each server is accessed at different orders by the I/O nodes, each node applies a translation rule to the server identifiers.

TWINS can increase performance by up to 48% over other schedulers available for the I/O forwarding layer (we compared it to FIFO and HBRR provided by the IOFSL framework [4]), as illustrated by Fig. 1(a). These high performance improvements are obtained transparently, without requiring any modifications in the applications or runtime systems (except the forwarding software). On the other hand, as depicted by Fig. 1(b), for other patterns performance can be decreased by TWINS. In both situations, the selection of the value for the time window duration parameter, which depends on the access pattern, is of paramount importance. Further details about TWINS and its performance evaluation are available in our previous work [9].



(a) `READ`, 8 forwarding nodes (b) `WRITE`, 2 forwarding nodes

Fig. 1. Impact of the window size on performance based on the execution time as perceived by the user (makespan). A total of 128 processes access a 4 GB shared file in 32 KB 1D-strided requests. Baseline algorithms are colored in red and TWINS results (with distinct window sizes) are in blue.

III. Motivation from Real-World HPC

In this paper, we propose an approach to adapt the I/O system to a changing workload by selecting appropriate values for parameters (such as the time window duration for TWINS), which depend on the current access pattern. The access pattern represents the way applications issue their I/O requests, i.e., aspects such as type of operation, file layout, sequentiality, size of requests, etc. In this section, we discuss the viability of applying such a solution for a real-world HPC workload.

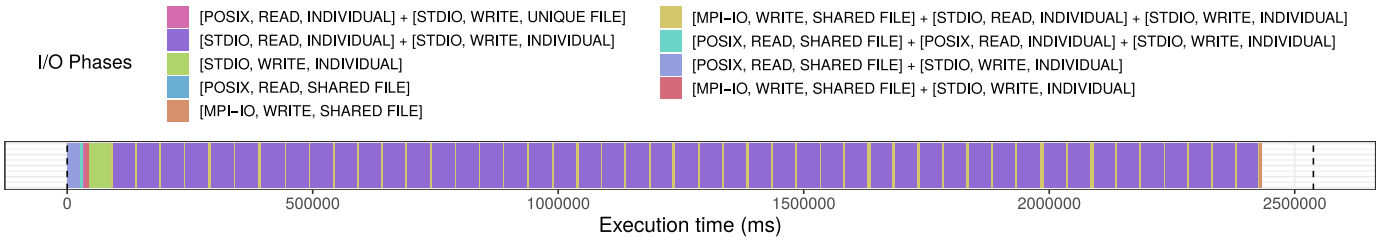
Fig. 2 illustrates the executions of two applications as reported by Darshan [12] traces. The x-axis represents the execution time, different colors represent different access patterns, and the boxes identify I/O phases. We use the concept of I/O phases to identify intervals where I/O operations are made using an access pattern. We infer this information from coarse-grained aggregated logs. Hence I/O operations are not necessarily happening throughout those entire periods, but we can be sure that those patterns characterize the I/O operations that are. The duration of each phase is defined by the interval between the first and the last I/O operations from a sequence of operations with the same access pattern.

Fig. 2(a) illustrates the I/O behavior the Ocean-Land-Atmosphere Model (OLAM) [13], executed in the Santos Dumont[2] supercomputer, at the National Laboratory for Scientific Computation (LNCC), in Brazil. We selected a job that used 240 processes and ran for 2433 seconds, of which 221 seconds were spent on I/O operations. OLAM processes read time-dependent input files, write per-process logs (purple), and periodically write to a shared-file with MPI-IO (yellow).

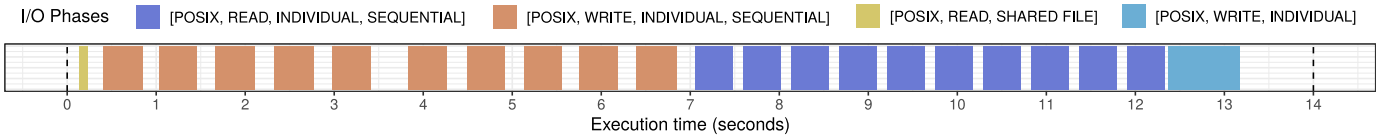
The second application, in Fig. 2(b), is anonymously identified as `2201660091`, job `1533518366532481378 4` from the Argonne Leadership Computing Facility I/O Data Repository [14]. This execution was randomly chosen from those who spent at least 30% of their time on I/O. We can see the application performs sequential writes to individual files in roughly the first half of the execution, and then it moves on to perform sequential reads from individual files.

The examples illustrates that HPC applications tend to present a consistent I/O behavior, with a few access patterns being repeated multiple times over an extensive period [15]. Therefore, one way of adapting the stateless I/O system would be to observe the current access pattern over some time and combine it with a tuning strategy. A supervised strategy (a decision tree, for example) would require a previous training step. However, considering I/O performance is sensitive to a large number of parameters, creating a training set to represent all applications (and all interactions of concurrent applications) and executing it would be difficult, error-prone, and time-consuming. Therefore the desired approach should be able to learn the best choice for each situation during the execution of applications, trying to avoid possible disturbances. The next section describes our proposal for such an approach.

[2]https://www.top500.org/system/178568



(a) The I/O phases of one execution of the OLAM application at the Santos Dumont Supercomputer (LNCC).



(b) Application `2201660091` (job `15335183665324813784`) running in the Intrepid supercomputer.

Fig. 2. Two real-world executions in HPC systems

IV. Adaptive I/O forwarding layer

We require a solution where the I/O forwarding layer learns the best choice for different situations while they are being observed, but without a prior training process due to its high cost and complexity. Therefore we approach this as a Reinforcement Learning (RL) problem [11]. We view it as a *k-armed bandit problem* [16], where at each step an agent takes one of the k possible actions (for TWINS each action is a different time window duration) and receives a reward (performance). The expected reward from an action is called its *value*. Since the value of an action a is *not* known beforehand, at the time step t we only have an estimate of it $Q_t(a)$, based on rewards obtained by taking that action in the past. The algorithm selects actions with the highest estimated values (*exploitation*), but also occasionally chooses other actions to attain better estimates of their values (*exploration*).

Nonetheless, the values (performance with each parameter) change as the access pattern changes. To avoid having to "reset" the learning process whenever that happens, we model our problem as a *contextual bandit* (or *associative search* task) [11]: we have multiple concurrent "instances" of the k-armed bandit, one for each different access pattern. At each iteration, only one of these instances will be active. This choice carries the assumption that taking an action does not have an impact on the following observation. Tuning the parameter will impact the performance of the current I/O phase and therefore slightly anticipate or delay the next I/O phase, but we consider this effect as negligible because the application primarily dictates the access pattern.

We implemented each of the concurrent armed bandit instances as an ε-*greedy* algorithm, that at step t takes the action a of the highest estimated value $Q_t(a)$, with probability $(1-\varepsilon)$, or with probability ε takes a randomly selected action. Value estimates use *incrementally computed sample averages*, i.e., after obtaining this step's reward R, the estimate for a is updated as ($N(a)$ is the number of times a has been taken):

$$Q_{t+1}(a) = Q_t(a) + \frac{1}{N(a)}[R - Q_t(a)] \tag{1}$$

Since the proposed mechanism will run in the intermediate I/O nodes, its lifetime will *not* be restrained to the jobs' execution times. Consequently, after observing an access pattern multiple times, it will be capable of consistently providing performance improvements by making the best decision.

A. Architecture of the proposed mechanism

Our case study TWINS, described in Section II, aims to achieve global coordination, thus although request scheduling happens independently in each I/O node, decisions about the window size should *not*, as multiple I/O nodes should use the same parameter value for it to make sense. Hence, we included an agent called *Council* to our solution, depicted by Fig. 3. On each I/O node, the *Announcer* is responsible for sending metrics about the observed access pattern to the *Council*, and receiving the chosen window size. The *Council*'s workflow consists of two steps: *detection* and *decision*. The *detection* phase is responsible for classifying the observed access pattern from the metrics. A new *decision* is only made if the metrics allow for a detection (for instance if there are enough requests).

Algorithm 1 details the *Council* interactive learning process. Once a set of metrics is received, the access pattern is detected, and the corresponding instance of the armed bandit

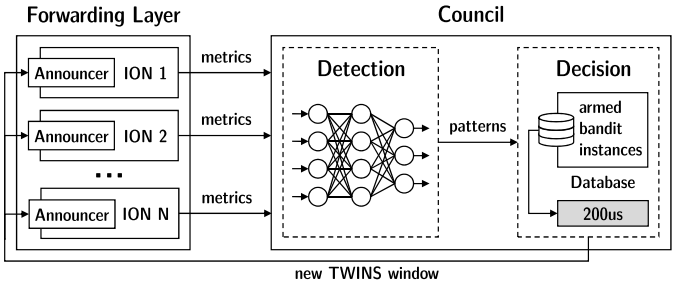


Fig. 3. The proposed architecture includes the *Announcers*, at the I/O nodes, and the centralized *Council* where detection and decision take place.

Algorithm 1 Council

Input:

ion_number is the number of connected I/O nodes

ε is the greedy exploration ratio

1: $S \leftarrow \varnothing$, $window \leftarrow default\_window$
2: **while** $true$ **do**
3: **while** $S.size() < ion\_number$ **do**
4: $host, metrics \leftarrow receiveMetrics()$
5: $reward \leftarrow getBandwidth(metrics)$
6: $pattern \leftarrow detectAccessPattern(metrics)$
7: $increment(N[pattern][window])$
8: $update(Q, N, pattern, window, reward)$ ▷ Eq. 1
9: ▷ Select the recommended window based on the pattern
10: $choices[host] = \underset{window}{\operatorname{argmax}} Q[pattern][window]$
11: $S.add(host)$
12: **end while**
13: ▷ Define the window size for all I/O nodes
14: **if** $random(0, 1) \leqslant \varepsilon$ **then**
15: $window \leftarrow randomWindow()$ ▷ Explore
16: **else**
17: $window \leftarrow mostOccurringInList(choices)$ ▷ Exploit
18: **end if**
19: $announceWindow(window)$
20: $S.reset()$
21: **end while**

is selected. Then the value estimate of the previously taken action is updated according to Equation 1, using the observed performance as reward. The estimates are used to determine the window size that is suitable for that I/O node (line 10). Upon making the centralized decision, the *Council* must decide between *exploration* or *exploitation*. For the latter, a consensus is required (line 17) between the possible divergent recommendations for each I/O node. In this work, we take the best for the majority. Investigating consensus policies for the particular case study of TWINS is subject to future work.

The centralized Council is required by our case study, but would not be if the tuned optimization tolerated different decisions being made by the I/O nodes. However, it accelerates the learning process, as parallel experiences are combined in the value estimates. It is also important to notice only the I/O nodes will interact with the Council, and not all the compute nodes. A compromise would be a hierarchical approach, where some of the ability of reaching global consensus is sacrificed for a decreased overhead. We investigate overhead and scalability of our solution in Section V-E.

B. Access pattern detection

Our solution requires a server-side classification of observed access patterns in order to identify an armed bandit instance to be used. Hence it is important for that classification to cover all aspects that have an impact on the behavior of the tuned optimization, otherwise the instances would not be able to learn properly. On the other hand, redundant classes slow down the learning, as multiple armed bandit instances cover the same behavior. Specifically to our case study, we classified the access pattern regarding the operation (read or write), spatiality (contiguous or 1D-strided), number of accessed files, and request size aspects, since our extensive performance evaluation of TWINS showed these to uniquely identify the different behaviors. The classification of the spatiality uses a neural network described in our previous work [17].

It is important to remember this is a server-side classification, where information from user-side libraries are not available and very little is known about the applications. When applying our proposal to other optimization techniques, the server-side access pattern detection must be adapted accordingly. If the set of relevant aspects is not known (which is often the case), a generic classification is required. In a previous work [18], we proposed such a classification strategy that covers all aspects. We represent access patterns as time series and use pattern matching to compare them. Our results have shown the same application access pattern to be represented by 1 to 3 patterns in the resulting knowledge base. Although that means having to observe it more times before learning, we still consider it to be a good strategy given the long life of the system (years), and considering the alternative is not adapting, and hence giving up of possible performance improvements.

V. Results and Discussion

In this section, we evaluate our proposal applied to the TWINS case study. We conduct an offline evaluation to show the learning ability (Section V-B), and an online one to show our architecture works in practice and leads to performance improvements (Section V-C). We also show the results of our approach with a real application (Section V-D) and evaluate its overhead and scalability (Section V-E). Section V-A discusses the methodology used for all experiments.

A. Experimental methodology

All experiments were carried out in clusters from the Grid'5000 platform [19] at Nancy: four PVFS2 servers in the Grimoire nodes, 32 clients and multiple IOFSL nodes in separated Grisou nodes. Each node of both clusters is powered by an Intel Xeon E5-2630 v3 processor (Haswell, 2.40 GHz, 2 CPUs/node, 8 cores/CPU) and 128 GB of memory. The parallel file system servers use a 600 GB HDD SCSI Seagate ST600MM0088. Nodes are connected by a 10 Gbps Ethernet interconnection, and there are four 10 Gbps links between the clusters. Both clusters were entirely reserved during the experiments to minimize network interference.

PVFS version 2.8.2 was deployed with default 64 KB stripe size and striping through all data servers. They write directly to their disks, bypassing caches, to ensure the scale of tests would be enough to see access pattern impact on performance. Clients are equally distributed among the I/O nodes, that communicate directly with the file system through the IOFSL dispatcher. The IOFSL daemon was launched with all its default parameters, aggregating up to 16 requests in dispatch, and using 4 to 16 threads.

Since the experiments in Sections V-B and V-C require metrics to guide our learning mechanism and determine the best scenario in the offline simulations, we used the MPI-IO

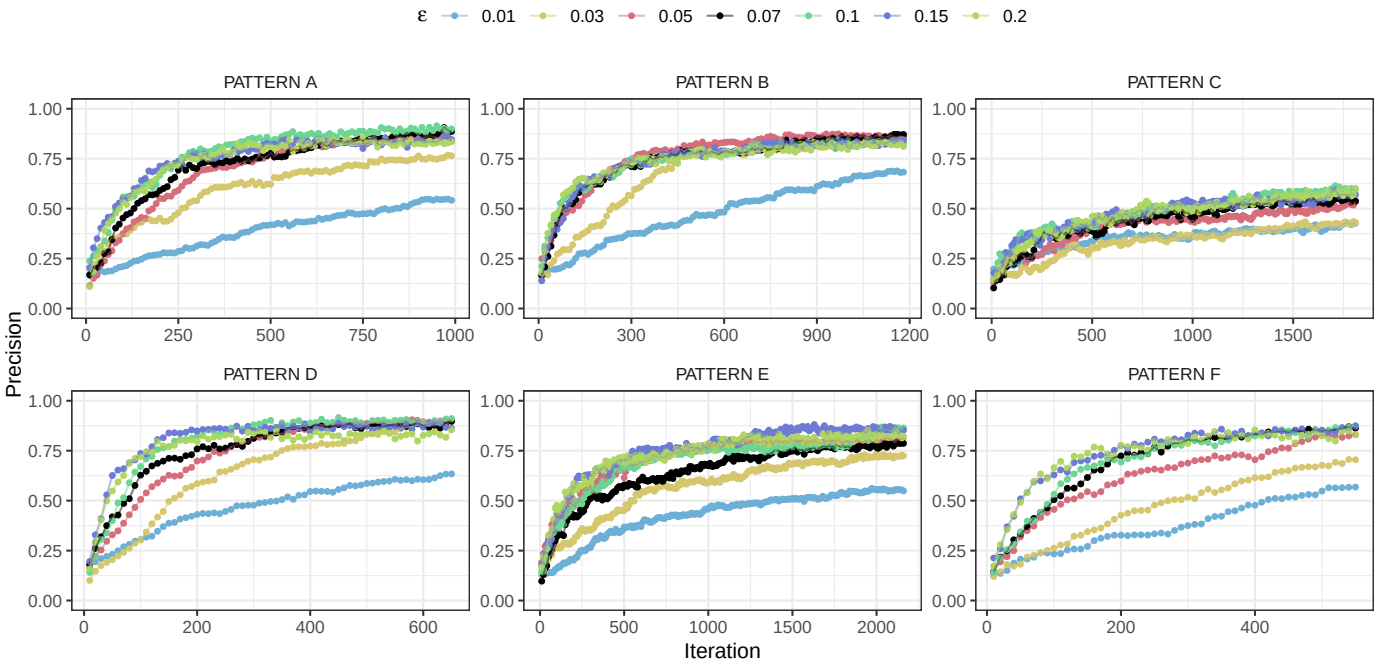


Fig. 4. Achieved precision, i.e., how often our approach chooses the correct window size, depicted in bins of 10 observations for simulations with different ε. Table III details the characteristics of the six patterns selected for this analysis. The x-axis of each plot is different because the simulations are limited by the amount of measurements obtained during the experiments described in Section V-A.

Test benchmarking tool [20]. We sought to cover the most common access patterns of HPC applications by varying the number of processes (128, 256, or 512), the file layout (shared-file or file-per-process), spatiality (contiguous or 1D-strided access), operation (reads or writes), and request sizes (32 or 256 KB – smaller than the stripe size or larger enough so that all servers are accessed). Each experiment writes/reads a total of 4 GB of data. To consider multiple deployment scenarios, we varied the number of available I/O nodes (1, 2, 4, or 8). These 144 different situations (we excluded the unusual 1D-strided file-per-process) were executed with seven different values for the time window parameter, for a total of $1,008$ experiments. Metrics were collected from all I/O nodes every second composing a dataset of over one million observations. The complete data set is publicly available at `jeanbez.gitlab.io/adaptative-io-scheduling`.

B. Offline evaluation

To evaluate our ability to learn the best parameter value without prior knowledge, we conducted simulations of the ε-greedy approach described in Section IV, assuming a perfect access pattern detection, separately for each of the $1,008$ scenarios. The algorithm has seven possible actions (window sizes) to choose from: 125µs, 250µs, 500µs, 1ms, 2ms, 4ms, and 8ms. All choices start with value estimates of zero. After deciding on an action, to determine its reward (performance), our simulation samples a data set of previously collected real measurements obtained with that window size in that scenario. The simulation duration is thus limited by the amount of measurements in the data set. We repeat each simulation 100 times to account for the sampling variability.

We group iterations into bins to calculate precision, i.e. to count how many times we chose the window duration that was previously observed to be the best for the access pattern. We then summarize the 100 simulations of each bin by their average. The bin size of 10 was chosen to aid the visualization of the trends. Table II compiles the precision difference from the first (untrained) to the last iterations. The precision improvements ranged between 26% and 582%.

TABLE II
PRECISION IMPROVEMENTS FOR ALL THE $1,008$ SCENARIOS.

Operation	Improvement (%)			
	Min.	Median	Mean	Max.
READ	26.37	184.06	216.10	582.35
WRITE	64.13	253.89	272.39	553.90
Total	26.37	226.82	244.24	582.35

From the 1,008 scenarios, we have selected six to illustrate the learning process. Fig. 4 depicts the precision, i.e., how often the correct value is selected, during simulations with different ε. The six patterns are detailed in Table III.

As the algorithm reaches better estimates for performance with different window durations, it selects the best value for the parameter more frequently, thus increasing precision. The smaller the ε (probability of exploration), the slower the convergence. On the other hand, in the long-term, an ε of 0.15, for instance, will choose the best action only at 85% of the times. An alternative would be to start with a high value for ε and decrease it over time. Table IV compiles the precision and performance achieved in the last bin (10

TABLE III
SELECTED PATTERNS CONCURRENTLY SIMULATED IN FIG. 5

Access Pattern	I/O Nodes	Processes	File Layout	Request Spatiality	Request Size	Op.
A	8	128	Shared	1D-strided	32KB	read
B	2	128	Shared	contiguous	32KB	write
C	8	512	Shared	contiguous	32KB	read
D	1	128	Shared	1D-strided	32KB	write
E	1	128	Individual	contiguous	32KB	write
F	4	128	Shared	1D-strided	32KB	read

TABLE IV
ACHIEVED PRECISION AND PERFORMANCE FOR THE SIX PATTERNS.

Access Pattern	A	B	C	D	E	F
Precision	0.88	0.88	0.49	0.87	0.59	0.59
Performance	0.99	0.96	0.96	0.97	0.98	0.92

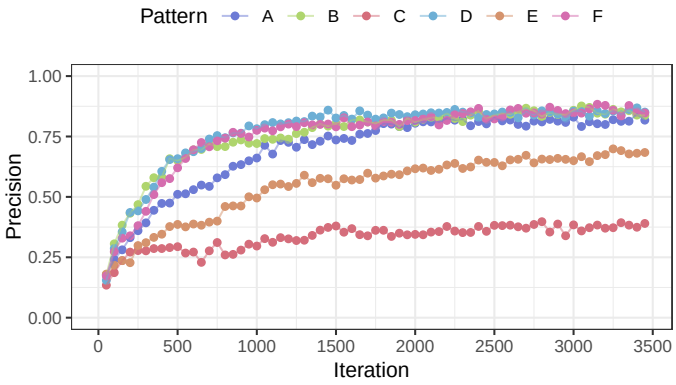


Fig. 5. Precision during the simulations of the six distinct concurrent access patterns detailed in Table III, using $\varepsilon = 0.15$.

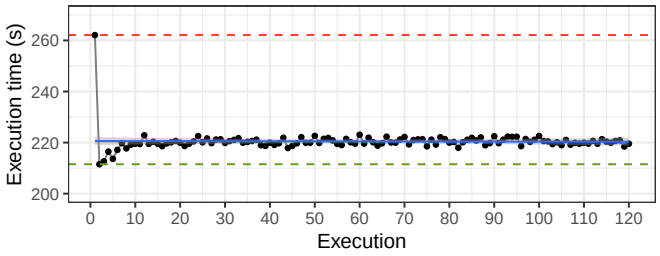
last iterations). Performance is normalized by what would be obtained if statically using the best possible window duration.

We can see our approach achieves better precision for some patterns than others. That is explained by the behavior of these patterns: situations where there is a clear better choice, with a large performance difference to other options, are easier to learn. On the other hand, if there are multiple parameter values that yield similar performance, the algorithm might converge to the "wrong" one. It is important to notice that selecting a value very similar to the best will still lead to good performance results: after learning, our approach is able to provide at least 92% of the performance of the best choice.

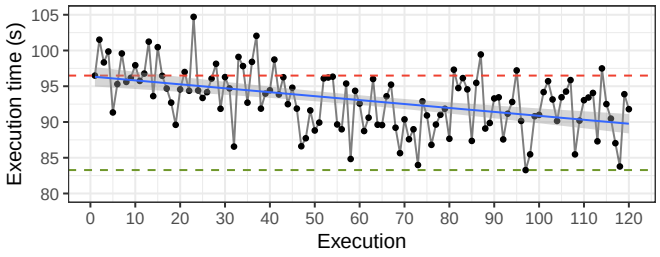
In practice, the different armed bandit instances will learn as the system observes different access patterns over time. To illustrate that, we executed the simulation with $\varepsilon = 0.15$, but this time at every iteration we randomly chose one of the six patterns to be observed. Fig. 5 presents the precision results and confirms that the bandit can indeed learn and reach similar precision as when observing each pattern separately.

C. Online evaluation

To test our proposal in practice, we performed an evaluation in the environment described in Section V-A, using four I/O



(a) WRITE phases

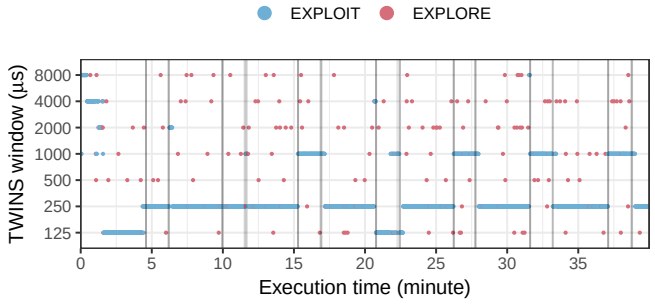


(b) READ phases

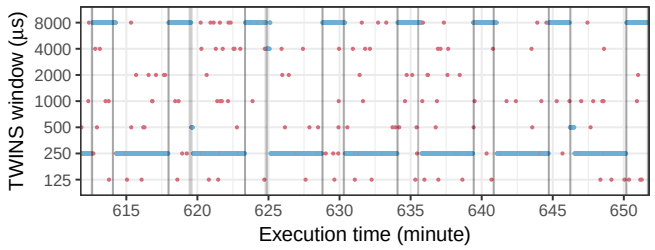
Fig. 6. Execution time of the benchmark during the learning process. The red dashed line shows the execution time of the first iteration, the green line indicates the lowest execution time, and the blue presents the trend using linear regression. The y-axis is different in each plot.

nodes. We executed a benchmark using 128 processes to write and then read a 4 GB shared file with 32 KB 1D-strided requests using MPI-IO. This scenario ("F" from the previous section), was chosen because of the impact of the value of the parameter on read and write performance, so it allows us to visualize results better. We executed it 120 times, while keeping track of benchmark execution time, of metrics observed in the council, and of selected window durations.

Fig. 6 shows the performance observed by the client — the execution time — throughout the 120 executions. The red dashed line shows the time for the first execution, the green line indicates the lowest time, and the blue line presents the trend obtained with a linear regression. Our approach was able to reach a choice for the write access pattern (Fig. 6(a)) still in the first execution of the benchmark (in the first $\approx 260s$), and yielded good results for the following ones. This learning process was not that fast for the read phases (Fig. 6(b)) for many reasons. First, they are shorter and consequently include less learning iterations. Secondly, it is important to notice consecutive write/read phases are separated by read/write ones, so in each phase there is a delay of at least one second (due to the interval in which metrics are reported) before detecting the new access pattern and acting accordingly (and that delay has a stronger impact on the shorter read phases than on the write ones). Third, the execution time of the read portion of the benchmark usually presents a higher variability, and that is reflected in the metrics observed by the council, which complicates the learning. Finally, the performance impact of bad decisions (sometimes made for exploration) is higher

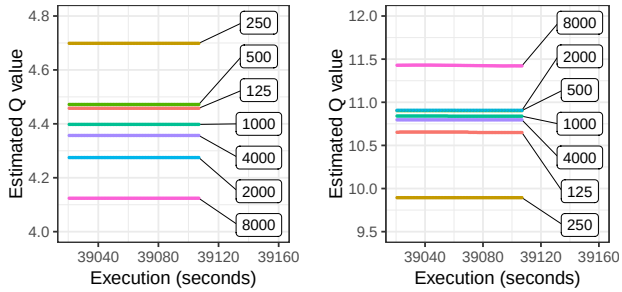


(a) First 40 minutes of execution



(b) Last 40 minutes of execution

Fig. 7. Elected window size during the first and the last 40min. of the online experiment. The gray lines separate the write (wide) and read (narrow) phases.



(a) WRITE (b) READ

Fig. 8. Value estimates for the actions (window durations) at the end of the online experiment. Each action (in µs) is shown by the side of its value.

for these read phases than for the write phases. Despite these adversities, we can see a descending trend for the read execution time as the system adapts. Read time decreases from $96.5s$ to $83.2s$, i.e., $\approx 14\%$.

Moreover, as discussed in the previous section, performance improvements are somewhat limited by the exploration phases. In these experiments, with $\varepsilon = 0.7$, the best value will be chosen 93% of the times. Decreasing ε over time would allow for better performance improvements. On the other hand, exploration can be important to adapt to changes in the system (if the best value for the parameter changes over time). The choice depends on the situation at hand.

Fig. 7 illustrates this exploration/exploitation process by showing the first and the last 40 minutes of the experiment. The vertical gray line denotes a shift in the detected access pattern (between read and write). We can see the algorithm stabilizes at the choice of 250µs as the window duration for write phases (the wide ones) from the second execution, while the choice as $8ms$ for read phases (the narrow ones) does not happen in the first 7 executions. This is clarified by Fig. 8, which shows the value estimates for the possible actions (window durations) towards the end of the experiment for the write (Fig. 8(a)) and the read (Fig. 8(b)) access patterns.

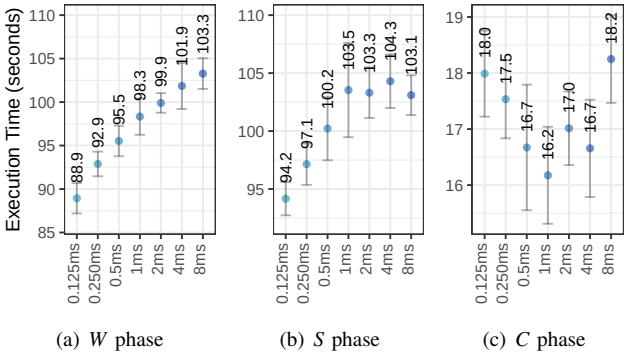
D. Results with MADspec

We also evaluated our approach using the MADspec I/O kernel (MADbench2 [21]). It allows testing the integrated performance of the I/O, communication, and calculation subsystems of massively parallel architectures under the stresses of a real scientific application. It is derived directly from a large-scale Cosmic Microwave Background (CMB) data analysis package, which calculates the maximum likelihood angular power spectrum of the Cosmic Microwave Background radiation from a noisy pixelized map of the sky and its pixel-pixel noise correlation matrix. The application has three component functions, each with different access patterns, named *S*, *W*, and *C*. Table V describes them. In our evaluation we used $N_p = 256$, $N_{pix} = 1280$, $N_{bin} = 80$, and $N_{gang} = 1$. The application uses the MPI-IO interface to issue its I/O operations synchronously to a single shared file.

TABLE V
I/O CHARACTERISTICS OF THE MADCODE.

Name	Input / Output
S	N_{bin} writes each of ${N_{pix}}^2$ bytes on N_p processors.
W	N_{bin} reads each of ${N_{pix}}^2$ bytes on N_p processors. N_{bin} writes each of ${N_{pix}}^2$ bytes on N_p/N_{gang} processors.
C	${N_{bin}}^2/N_{gang}$ reads each of ${N_{pix}}^2$ bytes on N_p/N_{gang} processors.

To evaluate the impact of the window choice for this application, we conducted an exploratory investigation measuring the execution time when using a fixed TWINS window size during the execution. Having those results as a baseline, we can evaluate the choices made by our approach. Fig. 9 presents the mean of five repetitions for each component using different window sizes. For the *W* and *S* phases, a small window size



(a) *W* phase (b) *S* phase (c) *C* phase

Fig. 9. Execution time of the *W*, *S*, and *C* components of MADspec with different window durations. The y-axes are different and do not start at zero.

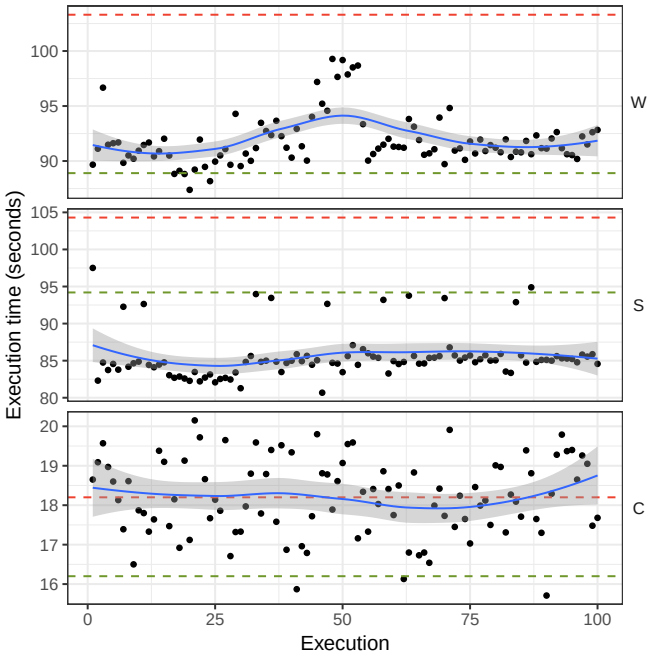


Fig. 10. Execution time for *W*, *S*, and *C* while adapting the TWINS window size. The dashed lines indicate the previously measured times without adaptation. In red, the worst window size, and in green, the best one for each scenario. The blue line represents the trend using a Local Polynomial Regression Fitting function with a 95% confidence level interval.

(125µs or 250µs) yields better results. On the other hand, for *C*, a larger window is better. For the latter, results are less conclusive due to higher variability.

Fig. 10 shows the execution time of the three phases when executed repeatedly while our approach works to adapt the TWINS window size. The red and the green dashed lines come from Fig. 9 and show to each case the result previously obtained with the worst and with the best values for the parameter, respectively. For the *W* component, the window choice gets closer to the best (though in some executions, due to exploration, we get higher times). For *S*, all times are below the previous best. This happens because S is a mixture of different shorter access patterns, and the adaptation mechanism is able to tune the parameter to them separately, which is better than using a static window for the whole phase. Finally, for *C*, due to the read pattern being shorter and more variable, there were not enough interactions to learn.

E. Analysis of overhead and time-to-decision

In our proposal, a separated *Announcer* thread in each I/O node interacts with the *Council*. This allows for requests to continue to be received and processed while metrics are asynchronously sent and the decision is made and broadcasted. Therefore, the overhead of our proposal is related to the cost of collecting and keeping metrics about the current access pattern. To quantify this overhead, we repeated all 144 experiments described in Section V-A using the proposed architecture but ignoring the decisions. This means that new values for the window parameter are chosen and announced, but TWINS continues to use the same window as before. Hence we can compare the execution time to a static solution.

Table VI summarizes the results for the 65 scenarios out of the 144, where we observed any overhead. Among these, the minimum observed overhead was 0.02%, the median, 1.8%, and the maximum of 32%. The latter was observed when a single I/O node was used by 512 processes to read contiguously from a shared file using with small requests (32 KB). Furthermore, 81.5% of those scenarios have an overhead of less than 5%. We conclude that in general, our proposal imposes a low overhead (median $< 2\%$).

TABLE VI
OVERALL OVERHEAD (%) OF OUR APPROACH FOR THE 144 SCENARIOS, EXCLUDING THE 79 ONES WHERE THE OVERHEAD WAS ZERO.

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.0199	0.8378	1.7780	3.8354	4.1218	32.2950

We also evaluate the *Council*'s ability to handle a large number of I/O nodes reporting metrics. In order to do that, we measure the total time to decision, which is measured between before the *Announcer* starts to send its metrics to right after it receives the new parameter value from the council. We executed an increasing number of *Announcer* processes (up to the largest number of I/O nodes in Table I) to report to the centralized *Council* every second. Results (the average of 60 measurements) are presented in Fig. 11.

The centralized decision-making agent is able to work under the heavier workload, with an expected degradation of the time to decision. In this work we used a rather naive communication strategy and a single-threaded prototype of the *Council*, so there is room for improvement. This time does *not* directly impose overhead because, as previously discussed, the adaptation mechanism happens asynchronously.

The time to decision is important, however, when selecting the adaptation frequency. It is essential to give the system enough time after adapting to observe an impact on performance. On the other hand, we want to make it as often as possible, so shorter I/O phases can still benefit. In the experiments from this paper, the *Announcer* sent metrics one second after the previous decision, and the time to decision was measured to be of tens of milliseconds (median of 131ms). That yielded good adaptation results. A degraded time to solution would mean a lower adaptation frequency.

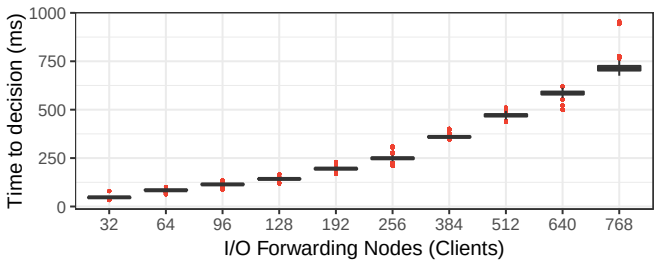


Fig. 11. Time to decision when I/O nodes are reporting metrics every second.

An alternative in this case would be to separate the I/O nodes in groups, and assign one *Council* per group. That would be a compromise where we sacrifice some ability of making global decisions. It is important to notice this discussion only concerns optimization techniques that require global decisions, since otherwise a centralized *Council* is not required.

F. Discussion on other aspects of the proposal

In our case study of TWINS, the continuous numerical parameter "window size" was represented as a set of different values to enable our bandit approach to be applied to this problem. We believe this is an appropriate strategy because when optimization techniques are proposed, they are typically experimented with a set of values for their parameters, as developers have an idea of what would be reasonable (for instance, for TWINS it is clear windows of several milliseconds or a few seconds would cause requests to starve). The real optimal values may lie between classes, but reaching the performance of the best among the observed classes is already an improvement over having no adaptation at all. Extrapolating from the case study, if the tuned optimization allows for it, it could be a good idea to provide fewer options to the armed bandit. That would accelerate learning and could be combined with a smaller value of ε without losing the ability to find the best option. Further, in Section V-B, we attested that having options with similar values slows down learning.

Regarding the choice of using the bandwidth as a reward, the caveat is that a low demand can result in a low bandwidth that is not related to the success of the optimization technique. Hence, that could carry noise to the learning process. This strategy is not a problem in our experiments because measurements have the same "intensity" of access. Furthermore, we ignored rewards when no I/O is being done, i.e., when the bandwidth is zero. To mitigate this problem in practice, we must either take into account load metrics (like the number of bytes requested) or apply some bandwidth normalization. Our server-side access pattern classification [18] approach solves the issue by accounting for the load.

VI. Related Work

The I/O forwarding layer has been the focus of research to improve its performance and transparently benefit applications. Vishwanath et al. [5] improved I/O performance of an IBM Blue Gene/P supercomputer by up to 38% by improving this layer. Their modifications allowed for asynchronous operations in the I/O nodes and included a simple FIFO request scheduler to coordinate accesses from the multiple threads. The same authors later optimized data movement between layers through a topology-aware approach [22], whereas Isaila et al. [23] proposed a two-level prefetching scheme. Ohta et al. [7] implemented a FIFO, and the quantum-based HBRR request schedulers for the IOFSL framework. The latter aims at reordering and aggregating requests. TWINS, on the other hand, was the first to aim at coordinating accesses to the data servers to avoid contention. Our previous work [9] showed it to improve performance and alleviate interference.

A number of parameters affect I/O performance, thus tuning the system requires a large number of experiments. Research has been directed to facilitate the configuration of the I/O stack [24]–[26]. Building models to represent the impact of parameters is a usual strategy, as done by McLay et al. [26] to optimize MPI-IO collective writes to Lustre. Nevertheless, that is unavoidably specific to the tested system, while we coveted a generic approach. The same disadvantage is present in decision tree proposals like the one by Boito et al. [10].

Behzad et al. [24] proposed an auto-tuning system that intercepts HDF5 calls at runtime and applies a genetic algorithm to tune a set of parameters like Lustre stripe size and count, and MPI-IO collective nodes and buffer size. Their approach decreases the number of experiments to be executed to tune the parameters, but it does not eliminate the profiling phase.

Focusing on block-level local storage, Nou et al. [27] used pattern matching to record known patterns and their performance with different disk schedulers. This work is close to ours in the sense that it does not require prior knowledge. However, our server-side patterns are diverse, with concurrency and variability caused by the network, i.e., the knowledge base would grow to a point where overhead, memory footprint, and slow convergence would make it unfeasible.

CAPES, the tuning system proposed by Li et al. [28], takes periodic measurements of a machine and train (online) a deep neural network that uses Q-learning to change parameters. They applied it to pick the congestion window size and the I/O rate limit, improving write performance by up to 45%. Their approach requires previous training that can take over 24 hours, while our system can learn and start to benefit from a new access pattern after minutes of it is first seen.

VII. Conclusion

Different I/O optimization techniques (including but not limited to the I/O forwarding layer) typically provide improvements for specific system configurations and application access patterns, but not for all of them. Moreover, they often require fine-tuning of parameters. In this paper, we focused on the I/O forwarding layer and proposed an approach to make it adapt to different access patterns. Our case study was TWINS, a request scheduler that provides improvements over other algorithms, but it is strongly dependent on selecting the proper window size parameter. Our approach has an agent named *Council* that periodically receives access and performance metrics observed by the I/O nodes, reported by a *Announcer*. Based on the detected access pattern, a contextual bandit is then used to learn the best window to each pattern during execution. This mechanism does not require previous training, which is a difficult, error-prone, and time-consuming task.

Our results for the offline evaluation of $1{,}008$ scenarios have shown that our approach is capable of reaching a precision of $\approx 88\%$ (and achieve the best option's performance) in the first hundreds of observations of a given access pattern. In our online evaluation, we observed it discovering the correct window sizes (based on our initial baseline) and showing improvements of up to 19.3% by avoiding window

sizes that could harm performance. Additional experiments with an application, the MADspec I/O workload, demonstrate the applicability of our learning mechanism by reducing the impact of a wrong window choice by up to 17% (for the "S" phase). Finally, the median overhead imposed by our proposal is inferior to 2%, and the time required to announce metrics and reach a decision is short enough to make adaptation viable.

The approach we proposed in this paper is not specific to tuning the TWINS window size parameter, and it can be applied to other scenarios. Future work will focus on extending our approach to other tunable parameters of the HPC I/O stack. A demonstration of our approach in a large scale real machine is also in our plans, although made difficult by the need of changing the I/O forwarding software. Finally, all the data, source-codes, and analysis conducted in this paper are available in the companion repository: `jeanbez.gitlab.io/adaptative-io-scheduling`.

Acknowledgment

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. It has also received support from the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), Brazil; the LICIA International Laboratory; NCSA-Inria-ANL-BSC-JSC-Riken Joint-Laboratory on Extreme Scale Computing (JLESC); the Spanish Ministry of Science and Innovation under the TIN2015–65316 grant; and the Generalitat de Catalunya under contract 2014–SGR–1051. This research received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 800144. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr). The authors acknowledge the National Laboratory for Scientific Computing (LNCC/MCTI, Brazil) for providing HPC resources of the SDumont supercomputer, which have contributed to the research results reported within this paper.

This project is partially funded by the European Union.

References

[1] SUN, "High-performance storage architecture and scalable cluster file system," Sun Microsystems, Inc, Tech. Rep., 2007. [Online]. Available: http://www.csee.ogi.edu/~zak/cs506-pslc/lustrefilesystem.pdf

[2] P. H. Carns *et al.*, "PVFS: A parallel file system for linux clusters," in *Proc. of the Extreme Linux Track: 4th Annual Linux Showcase and Conferenc*. USENIX Association, 2000, pp. 317–327.

[3] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the panasas parallel file system," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, ser. FAST'08. USENIX Conf. on File and Storage Technologies, 2008, pp. 2:1–2:17.

[4] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham *et al.*, "Scalable I/O Forwarding Framework for High-Performance Computing Systems," in *Proceedings...*, IEEE International Conference on Cluster Computing and Workshops. IEEE, Aug. 2009, pp. 1–10.

[5] V. Vishwanath, M. Hereld, K. Iskra, D. Kimpe, V. Morozov, M. E. Papka *et al.*, "Accelerating I/O forwarding in IBM Blue Gene/P systems," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, Nov. 2010, pp. 1–10.

[6] G. Almási, R. Bellofatto, J. Brunheroto, C. Caşcaval, J. G. Castanos, L. Ceze *et al.*, "An overview of the Blue Gene/L system software organization," in *Proceedings...*, Euro-Par 2003 Conference, Lecture Notes in Computer Science. Springer-Verlag, 2003, pp. 543–555.

[7] K. Ohta, D. Kimpe, J. Cope, K. Iskra, R. Ross, and Y. Ishikawa, "Optimization Techniques at the I/O Forwarding Layer," in *Proceedings...* International Conference on Cluster Computing, 2010, pp. 312–321.

[8] R. McLay, D. James, S. Liu, J. Cazes, and W. Barth, "A user-friendly approach for tuning parallel file operations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. IEEE Press, 2014, pp. 229–236. [Online]. Available: https://doi.org/10.1109/SC.2014.24

[9] J. L. Bez, F. Z. Boito, L. M. Schnorr, P. O. A. Navaux, and J.-F. Méhaut, "TWINS: Server Access Coordination in the I/O Forwarding Layer," in *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, March 2017, pp. 116–123.

[10] F. Z. Boito, R. V. Kassick, P. O. A. Navaux, and Y. Denneulin, "Automatic I/O scheduling algorithm selection for parallel file systems," *Concurrency and Computation: Practice and Experience*, 2015.

[11] R. S. Sutton and A. G. Barto, "Reinforcement Learning: An Introduction," http://incompleteideas.net/book/the-book-2nd.html, 2017, accessed: August 2018.

[12] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and Improving Computational Science Storage Access Through Continuous Characterization," *Trans. Storage*, vol. 7, no. 3, pp. 8:1–8:26, Oct. 2011.

[13] R. L. Walko and R. Avissar, "The Ocean–Land–Atmosphere Model (OLAM). Part I: Shallow-Water Tests," *Monthly Weather Review*, vol. 136, no. 11, pp. 4033–4044, 2008.

[14] P. H. Carns, "ALCF I/O Data Repository," http://ftp.mcs.anl.gov/pub/darshan/data/, 2013.

[15] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai, "Server-side log data analytics for I/O workload characterization and coordination on large shared storage systems," in *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 2016, pp. 819–829.

[16] D. A. Berry and B. Fristedt, "Bandit problems: sequential allocation of experiments (monographs on statistics and applied probability)," *London: Chapman and Hall*, vol. 5, pp. 71–87, 1985.

[17] J. L. Bez, F. Z. Boito, R. Nou, A. Miranda, T. Cortes, and P. O. Alexandre, "Detecting I/O Access Patterns of HPC Workloads at Runtime," in *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Brazil, Oct. 2019.

[18] F. Z. Boito, R. Nou, L. L. Pilla, J. L. Bez, J.-F. Méhaut, T. Cortes, and P. Navaux, "On server-side file access pattern matching," in *HPCS 2019 - 17th International Conference on High Performance Computing & Simulation*. Dublin, Ireland: IEEE, Jul. 2019, pp. 1–8.

[19] R. Bolze *et al.*, "Grid5000: A large scale and highly reconfigurable experimental grid testbed," *International Journal of High Performance Computing Applications*, vol. 20, no. 4, pp. 481–494, 2006.

[20] Los Alamos National Laboratory, "MPI-IO Test Bechmark," http://freshmeat.sourceforge.net/projects/mpiiotest, 2008.

[21] J. Borrill, L. Oliker, J. Shalf, and H. Shan, "Investigation of leading HPC I/O performance using a scientific-application derived benchmark," in *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, Nov 2007, pp. 1–12.

[22] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka, "Topology-aware data movement and staging for I/O acceleration on blue gene/p supercomputing systems," in *Proceedings...*, ser. SC '11, 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, 2011, pp. 19:1–19:11.

[23] F. Isaila, J. Garcia Blas, J. Carretero, R. Latham, and R. Ross, "Design and evaluation of multiple-level data staging for blue gene systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 6, pp. 946–959, 2011.

[24] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, R. Aydt, Q. Koziol *et al.*, "Taming parallel I/O complexity with auto-tuning," in *SC'13: Proceedings of the International Conf. on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, pp. 1–12.

[25] F. Isaila, J. Carretero, and R. Ross, "CLARISSE: A middleware for data-staging coordination and control on large-scale HPC platforms," in *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing)*, May 2016, pp. 346–355.

[26] R. McLay, D. James, S. Liu, J. Cazes, and W. Barth, "A user-friendly approach for tuning parallel file operations," in *Proceedings...*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 229–236.

[27] R. Nou, J. Giralt, and T. Cortes, "Automatic I/O scheduler selection through online workload analysis," in *9th International Conference on Autonomic and Trusted Computing*. IEEE, 2012, pp. 431–438.

[28] Y. Li, O. Bel, K. Chang, E. L. Miller, and D. D. E. Long, "Capes: Unsupervised storage performance tuning using neural network-based deep reinforcement learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2017.