CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2015; **00**:1–18
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Automatic I/O scheduling algorithm selection for parallel file systems

Francieli Zanon Boito[1,2*], Rodrigo Virote Kassick[1,2], Philippe O. A. Navaux[1], Yves Denneulin[2]

[1]*Institute of Informatics – Federal University of Rio Grande do Sul (UFRGS) – Porto Alegre, Brazil*
[2]*LIG Laboratory – INRIA – University of Grenoble – Grenoble, France*

SUMMARY

This article presents our approach to provide I/O scheduling with double adaptivity: to applications and devices. In High Performance Computing (HPC) cluster environments, Parallel File Systems (PFS) provide a shared storage infrastructure to applications. In the situation where multiple applications access this shared infrastructure concurrently, their performance can be impaired because of interference. Our work focuses on I/O scheduling as tool to improve performance by alleviating interference effects. The role of the I/O scheduler is to decide the order in which applications' requests must be processed by the parallel file system's servers, applying optimizations to adjust the resulting access pattern for improved performance. Our approach to improve I/O scheduling results is based on using information from applications' access patterns and storage devices' sensitivity to access sequentiality. We have applied machine learning to provide the ability of automatically select the best scheduling algorithm for each situation. Our approach improves performance by up to 75% over an approach that uses the same scheduling algorithm to all situations, without adaptability. Moreover, our approach improves performance for up to 64% more of the tested scenarios, and decreases performance for up to 89% less scenarios. Our results evidence that both aspects - applications and storage devices - are essential to make good scheduling decisions. Furthermore, despite the fact that there is no scheduling algorithm able to improve performance for all situations, we have shown that through double adaptivity it is possible to apply I/O scheduling to improve performance, avoiding situations where it would lead to performance impairment. Copyright © 2015 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: I/O Scheduling; Parallel File Systems; High Performance Computing

1. INTRODUCTION

High Performance Computing (HPC) applications that execute on cluster or MPP architectures rely on Parallel File Systems (PFS) to achieve performance even when having to input and output large amounts of data. Since data access speed has not increased in the same pace as processing power, several approaches like collective I/O [1] were defined to provide scalable, high performance I/O. These techniques usually explore the fact that performance observed when accessing a file system is strongly affected by the manner accesses are performed. Therefore, they work to adjust applications' *access patterns*, improving characteristics such as spatial locality and avoiding well-known situations detrimental to performance, such as issuing a large number of small, non-contiguous requests [2].

Most assumptions about performance behavior that guide optimizations' development come from the use of *Hard Disk Drives* (HDDs). For years, magnetic disks have been the main non-volatile

*Correspondence to: Instituto de Informática, Av. Bento Gonçalves, 9500 - Campus do Vale - Bloco IV, Bairro Agronomia, cep 91509-900, Porto Alegre-RS, Brazil.

storage devices available. Multiple hard disks can be combined into a virtual unit as a *RAID array* for performance and reliability purposes. Another recent and alternative technology uses flash-based storage devices named *Solid State Drives* (SSDs). Their advantages over HDDs include: resistance to falls and vibrations, size, noise generation, heat dissipation, and energy consumption [3].

Since both SSDs and RAID solutions are inherently different from HDDs, they should not be simply treated as "faster disks". Several assumptions about performance from HDDs do not hold when using SSDs or RAID arrays, and different requirements arise. Regarding spatial locality, HDDs are known for having better performance when accesses are done sequentially. On the other hand, works that aim at characterizing SSDs' performance behavior achieve different conclusions. On some SSDs, there is no difference between sequential and random accesses, but on others this difference achieves orders of magnitude [3]. The *sequential to random throughput ratio* on some SSDs surpasses what is observed on some HDDs [4].

Therefore, we cannot simply classify optimizations by saying they are only suitable for HDDs or SSDs. Approaches that aim at generating contiguous accesses (originally designed for HDDs) can greatly improve performance when used on SSDs that are also sensitive to access sequentiality. Furthermore, on any device, the performance improvement caused by the use of a specific optimization may not compensate its overhead. Hence, these optimizations could be classified according to the sequential to random throughput ratio that devices must present in order to benefit from them. [5]

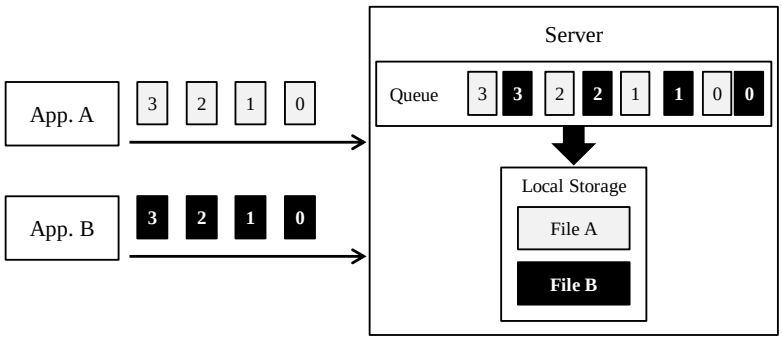


Figure 1. Interference on concurrent accesses to a PFS server.

These optimizations that work to improve performance by adjusting applications' access patterns usually do so in the context of one application. However, today's HPC architectures usually deploy a parallel file system on a set of dedicated machines and storage devices. This storage infrastructure is shared by all applications. When multiple applications access the file system concurrently, their performance will suffer from *interference*. This phenomenon is illustrated in Figure 1, where two applications ("App. A" and "App. B") concurrently access a server for the files named A and B. Despite I/O optimizations that may have been employed by these applications, they will observe poor performance because the access pattern at the server is not ideal. In this case, applications' accesses interfered with each other. Nonetheless, it is also possible to observe interference between processes of the same application.

Our work focuses on I/O scheduling as a tool to improve performance by alleviating interference effects. We consider server-side schedulers, that work in the context of requests to parallel file systems' servers. Their functionality consists on deciding the order in which these requests must be processed. Moreover, schedulers may apply optimizations, as requests aggregation, to adapt the resulting access pattern for improved performance.

I/O scheduling algorithm's success in improving performance depends on both applications' and platforms' characteristics. As previously discussed, the storage devices' sensitivity to access sequentiality limits the efficacy of optimizations that adjust access patterns. Moreover, access patterns' aspects such as request size and number of accessed files define the requests window the scheduler will have to work on, affecting how much "room for improvement" there will be during

DOI: 10.1002/cpe

applications' execution. Therefore, it is important for I/O schedulers to have double adaptivity: to applications and to storage devices.

This article presents our approach to provide I/O scheduling for parallel file systems with double adaptivity. We use our I/O scheduling tool named AGIOS with an NFS-based parallel file system to evaluate five I/O scheduling algorithms. Our results evidence that no scheduling algorithm is able to improve performance for all situations, and the best fit depends on both applications and devices. We have used machine learning to make AGIOS capable of making this selection automatically. Our main contributions are threefold:

- We present AGIOS, an I/O scheduling tool generic, non-invasive and easy to use. It provides five options of scheduling algorithm: aIOLi, MLF, SJF, TO, and TO-agg.
- We extensively evaluate the five scheduling algorithms under different access patterns and over four clusters, representing different storage devices' sensitivity to access sequentiality.
- We have used the results obtained from our evaluation to make AGIOS capable of automatically select the best fit in scheduling algorithm for each situation. We discuss and evaluate our approach.

The remaining of this article is organized as follows: Section 2 presents AGIOS and the five studied scheduling algorithms. Section 3 evaluates these algorithms with different benchmarks and platforms. Section 4 details how the performance evaluation's results were used to build a decision tree that is able to automatically select the best fit in scheduling algorithm for each situation. Results obtained with this approach are presented in Section 5. Section 6 discusses related work, and Section 7 brings final remarks and future work.

2. AGIOS: AN I/O SCHEDULING TOOL

I/O schedulers found in the literature are usually specific to a given file system. Moreover, most of them impose specific file system configurations to work, such as a centralized metadata server. These characteristics restrict their usability in new contexts and comparisons between them. For these reasons, we developed an I/O scheduling tool named *AGIOS*. The main objectives for its development were to make it generic, non-invasive, and easy to use.

Although this work focuses on parallel file systems, our tool could be used by *any I/O service that treats requests at a file level*, such as a local file system or intermediate nodes in an I/O forwarding scheme. These placement options are not exclusive, in the sense that we could have all of them happening at the same time. In order to avoid creating a bottleneck, AGIOS instances (on different PFS servers, or at different levels of the I/O stack) are independent and do not make global decisions.

Our previous work [6] described AGIOS' interface with its users and one of its scheduling algorithms - aIOLi [7]. Comparing to our previous paper, our tool now provides five options of I/O scheduling algorithms: aIOLi, MLF, SJF, TO, and TO-agg. These algorithms were selected for their variety, in order to represent different situations and complement each other's characteristics. The next sections describe them, and their performance will be evaluated in Section 3.

2.1. aIOLi

We have adapted the *aIOLi* scheduling algorithm from Lebre et al. A full explanation and discussion about this algorithm's characteristics can be found in the paper that describes it [7], but we can summarize it as follows:

- Whenever new requests arrive to the scheduler, they are inserted in the appropriate queue according to the file to be accessed. There are two queues for each file: one for reads, and another for writes.
- New requests receive an initial quantum of 0 bytes.
- Each queue is traversed in offset order and aggregations of contiguous requests are made. When an aggregation is performed, a *virtual request* is created, and this request will have a quantum that is the sum of its parts' quanta.

DOI: 10.1002/cpe

- All quanta (including the virtual requests' ones) are increased by a value that depends on its queue's past quanta usage.
- In order to choose a request to be served, the algorithm uses an offset order inside each queue and a FCFS criterion between different queues. Additionally, to be selected, the request's quantum must be large enough to allow its whole execution (it needs to match the request's size).
- The scheduler may decide to wait before processing some requests if a) a shift phenomenon is suspected or b) better aggregations were recently achieved for this queue.
- After processing a request, if there is quantum left, other contiguous request from the same queue can be processed - given that they fit the remaining quantum. After stopping for a queue, its quanta usage ratio is updated. This scheduling algorithm works synchronously, in the sense that it waits until a virtual request is processed before selecting other ones.

The implementation uses a hash table indexed by file identifier for accessing the requests queues. At a given moment, the cost of including a new request to a queue can be represented as the sum of the required time to find the right queue plus the time to find its place inside the queue (sorted by offset order). The former is expected to be:

$$O((M/S_{hash}) + N_{queue}) \tag{1}$$

where M is the number of files being concurrently accessed, S_{hash} is the number of entries in the hash table, and N_{queue} is the number of requests in the largest queue. Selecting a request for processing, on the other hand, involves going through all queues:

$$O(2 \times M) \tag{2}$$

2.2. MLF

Under a workload where several files are being accessed at the same time, the cost of aIOLi's selection may become a significant part of requests' lifetime in the server. This happens due to the synchronous approach where the algorithm waits until the previous request was served before selecting a new one. In order to have a scheduler capable of providing more throughput, we developed a variation of aIOLi that we called *MLF*. We have chosen this name because our version is closer to the traditional MLF task scheduling algorithm than aIOLi.

To reduce the algorithm's overhead, we removed the synchronization between user and library after processing requests. Therefore, the new algorithm works repeatedly, possibly overflowing its user with requests.

Despite its possibly high scheduling overhead, one advantage of the synchronous approach is that having some time before the next algorithm's step gives chance for new requests to arrive and more aggregations to be performed. Therefore, it is possible that this new algorithm will not be able to perform as many aggregations as aIOLi.

Other difference between MLF and aIOLi is that MLF does not respect a FCFS order between the different queues. Therefore, not all queues need to be considered before selecting a request, improving the algorithm's throughput. MLF's cost for including new requests is the same as aIOLi's, but its cost for selection is $O(1)$.

2.3. SJF

We have also developed for our study a variation of the *Shortest Job First (SJF)* scheduling algorithm [8] that performs requests aggregation. Its implementation also uses two queues per file and considers requests from each queue in offset order. The selection of the next request is done by going through the queues and selecting requests from the smallest one (considering each queue's total size, i.e. the sum of all its requests' sizes).

Therefore, the cost for including a new request and for selection are the same as aIOLi's. However, our SJF variation does not work synchronously.

DOI: 10.1002/cpe

2.4. TO and TO-agg

TO is a timeorder algorithm. It has a single queue, from where requests are extracted for processing in arrival time order (FCFS). Both the costs of including and selecting requests are, therefore, constant. We have included TO in our analysis to cover situations where no scheduling algorithm is able to improve performance.

We have also included a timeorder variation that performs aggregations: *TO-agg*. Since there is a single queue for requests, aggregating a request possibly requires going through the whole queue looking for a contiguous one. Therefore, this algorithm's cost for including requests is:

$$O(N) \tag{3}$$

where N is the number of requests currently on the scheduler. The time for selection is still constant. TO-agg is included in this study mainly to show the impact of aggregations alone on performance, without the impact of requests reordering.

3. I/O SCHEDULING ALGORITHMS EVALUATION

This section presents a performance evaluation of the five scheduling algorithms discussed in the last section. This evaluation's main objectives are to identify the situations where I/O scheduling is able to improve performance and to understand what makes them suitable for this technique; and to identify which of the implemented scheduling algorithms is the best fit for each scenario.

For proof-of-concept purposes, we present our library's usage with dNFSp [9], an NFS-based parallel file system composed of several metadata and data servers (also called "IODs"). The distributed servers are transparent to the file system client, which accesses the remote file system through a regular NFS client. We integrated AGIOS within the IOD code, so each IOD contains an independent instance.

We executed tests on four clusters from Grid'5000 [10], described in Table I. These systems were selected by their variety on storage devices: we tested with SSDs, HDDs and RAID arrays. Moreover, these devices present diverse sequential to random throughput ratios, as shown in the table's last two columns for 8MB requests. This size is relevant because it is the transmission size between clients and servers in our dNFSp deployment and hence all requests arriving at the servers have size up to 8MB. A high ratio means that accessing files sequentially is several times faster than accessing them randomly. On the other hand, a ratio smaller than 1 means that accessing randomly is faster.

Table I. Platforms used in this work.

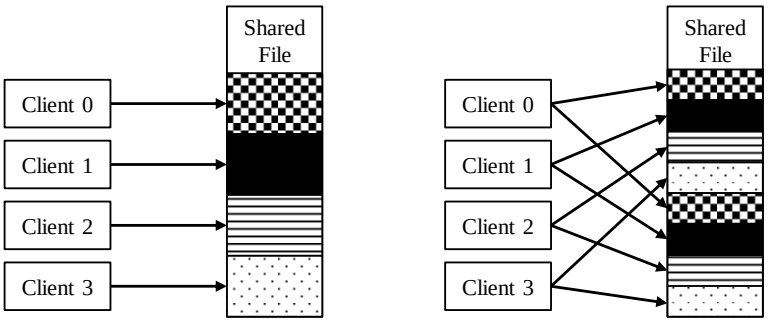
Cluster	Node Configuration				
	Processor	RAM	Storage Device	Sequential to Random Ratio	
			Type	Write	Read
Pastel @ Toulouse	2× 2-core AMD Opteron	$8GB$	HDD	21.29	38.91
Graphene @ Nancy	4-core Intel Xeon	$16GB$	HDD	15.12	40.68
Suno @ Sophia	2× 4-core Intel Xeon	$32GB$	RAID-0	8.17	25.46
Edel @ Grenoble	2× 4-core Intel Xeon	$24GB$	SSD	0.66	2.37

In all platforms, we used four nodes as dNFSp's data servers, one of them sharing its machine with a metadata server. Up to 32 processing nodes were used as clients. All nodes have the Debian 6 operating system with kernel 2.6.32. Both dNFSp and AGIOS were compiled with gcc 4.4.5. Virtual memory page size is 4MB.

DOI: 10.1002/cpe

We developed a set of tests using the MPI-IO Test benchmarking tool[†]. They explore the following list of relevant access pattern aspects:

1. Spatial locality: if applications issue requests that are contiguous or non-contiguous. In our tests, the non-contiguous case is represented by a 1-D strided access pattern. Contiguous access patterns to a shared file mean that each process has an exclusive portion of this file, i.e. there is no overlapping between different processes' accessed data. Figure 2 illustrates this situation with 4 clients.
2. Number of files: processes either share a file or have independent files (one per process). This configuration is done at application level, thus if four applications execute concurrently with a shared file access pattern, four files are currently being accessed in the file system.
3. Single or multi-application scenarios: we present results for single and multi-application scenarios. The multi-application case is represented by executing four instances of the same benchmark concurrently. Processing nodes are split evenly among the different instances.
4. Number of processes per application: how many processes perform I/O. We repeat our tests for different application sizes: 8, 16, and 32. Since the amount of data accessed by each process is constant, *the total amount of data grows with the number of processes*.
5. Size of requests: if requests are small (smaller than the file system's stripe size) or large (larger than the stripe size and large enough so all file system servers will have to be contacted in order to process it). These definitions of small and large requests follow what is presented by Byna et al. [11]. Table II summarizes the amounts of data accessed on different tests. The stripe size used by dNFSp is 32KB.



(a) Contiguous (non-strided) (b) Non-contiguous (1-D strided)

Figure 2. Spatial locality aspect with a shared file.

The benchmarking tool uses synchronous I/O operations, so requests from a process have to be served before the next batch can be sent to the server. However, the transmission size is limited to 8KB. Therefore, application's requests are divided in multiple actual requests: in small tests, each process will issue 2 requests at once; in large tests, 32. We have chosen the synchronous approach since it is more challenging to the scheduler, because its overhead has a higher impact, and little delays on processing a request can lead to longer execution times.

From each application execution, we take the completion time of the slowest process, since it defines this application's execution time. We take the maximum between concurrent applications' execution times because we are interested in reducing the *makespan*. We have chosen to use *makespan* as the metric for our performance evaluation, since it represents the total time to process the whole workload from the file system's point of view. Our metric does not reflect fairness or response time, as we are not focusing on these aspects, but on performance.

[†]http://institute.lanl.gov/data/software/mpi-io

DOI: 10.1002/cpe

Table II. Amount of data accessed in our experiments.

	Single Application				Multi-application			
	Shared File		N to N		Shared File		N to N	
	Small	Large	Small	Large	Small	Large	Small	Large
Processes	8-32				32-128			
Total data per process	64MB		1GB		64MB		512MB	
Accessed Files	1		8-32		4		32-128	
Files' size	512MB-2GB		1GB		512MB-2GB		512MB	
Total amount	512MB-2GB		8GB-32GB		2GB-8GB		16GB-64GB	
Requests per process	4096	256	64K	4096	4096	256	32K	2048
Requests' size	16KB	256KB	16KB	256KB	16KB	256KB	16KB	256KB

We use the POSIX API to generate applications' requests because we want to evaluate performance under the described access patterns. Using a higher level library such as MPI-IO [12] would potentially affect these patterns, compromising our analysis.

Each set of tests (with a different scheduling algorithm) was executed in a random order to minimize the chance of having some effect caused by a specific experiment order. Clients' portions and independent files were "shifted" after the write test so they would not read the same data they wrote (avoiding clients' caching effects). All results are the arithmetic mean of *at least* eight executions, with 90% confidence and 10% maximum relative error.

The next sections discuss obtained results with AGIOS and dNFSp using all the presented scheduling algorithms. Since the volume of tests is considerably large, showing all of them would compromise this article's readability. Therefore only a subset of them is shown to illustrate the discussions. The whole set of results can be obtained from `http://www.inf.ufrgs.br/~fzboito/all_agios_results.tgz`.

Since dNFSp already has a timeorder scheduling algorithm, results with AGIOS' TO (a simple timeorder) only evidence the library's overhead. Since TO has costs $O(1)$ for both including and selecting requests, this overhead is not expected to be important. In general, experiments where processes from a single application share a file are more sensitive to the library's overhead, since these tests have the smallest execution times (especially read tests). In the following sections, we focus on the remaining four scheduling algorithms, comparing them with times obtained with dNFSp without AGIOS.

3.1. Performance results for single application with shared file

The experiments where a single application's processes share a file represent the smallest workloads (see Table II). For these tests, a strong impact on performance is expected due to the scheduler's overhead. Since during each test only one file is accessed, aIOLi, MLF, SJF, and TO-agg present practically the same costs for including and selecting requests. This situation maximizes the number of requests per queue for aIOLi, MLF, and SJF, as all requests belong in the same queue. Moreover, the scheduling algorithms' global criteria do not apply, since there is only one queue.

Experiments that issue large requests provide more aggregation opportunities, since large applications' requests will be split into several contiguous actual requests to the servers because of the file system's transmission size limit. Performing more aggregations, the scheduling algorithms' effect on performance might be able to surpass their overheads. Additionally, because of the striping process, in contiguous access patterns, requests from different clients are not contiguous

DOI: 10.1002/cpe

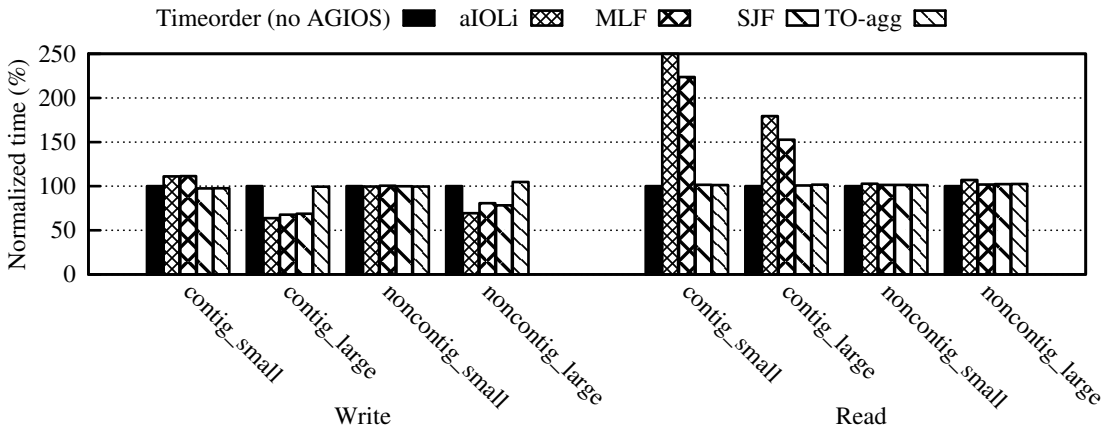


Figure 3. Results with a single application and the shared file approach in the Graphene cluster (tests with 8 processes).

at the servers, hence strided tests provide more aggregation opportunities than contiguous ones. Furthermore, read tests have smaller execution times and hence are more affected by scheduling overhead.

Considering the clusters with HDDs (Pastel and Graphene), the scheduler was only able to improve performance of *write operations* in the *large requests* access patterns. Figure 3 presents the results obtained in the Graphene cluster with 8 processes, normalized by the time observed for the file system without AGIOS (timeorder algorithm). Pastel presented similar behaviors, as did results for Graphene with 16 and 32 processes. The cluster with RAID-0 (Suno) also presented the best results for large write requests. However, differently from the clusters with HDDs, all other situations presented performance improvements in the tests with 32 processes (tests with 8 and 16 behaved similarly to the clusters with HDDs).

In the three platforms, the *worst results* were obtained with aIOLi and MLF for the tests with contiguous small requests (the situation that provides less aggregation opportunities). These algorithms provided *performance improvements* for large write requests tests of up to 25% in Pastel, up to 59% in Graphene , and up to 31% in Suno. The *best results* for large write operations were provided by the aIOLi algorithm.

Performance improvements obtained for Pastel are the smallest despite this platform's storage devices having the highest sequential to random throughput ratio for writes (see Table I). One possible reason is that the Pastel cluster is approximately three years older than the other two and its nodes have less memory and processing power. In this situation, scheduling algorithms' costs may become more important in the resulting performance. This also explains why large tests in Suno had better results, since this cluster has the largest amount of memory per node and thus is expected to be less affected by scheduling overhead.

In the experiments where there is only one queue being accessed, the main difference between MLF's and SJF's executions is that SJF does not have waiting times. aIOLi's and MLF's waiting times are expected to improve aggregations. SJF provided performance *decreases* of up to 23% in Pastel, performance *increases* of up to 39% in Graphene (only in the situations where aIOLi and MLF also improved performance) and performance *increases* of up to 32% in Suno. For access patterns of small write requests and for read ones, SJF is the best choice in Graphene because it provides only small performance differences (under 10%), and in Suno because it improves performance for most cases.

In the cluster with SSDs (Edel), tests' execution times were 2 to 4 times *longer* than what was observed in the other three clusters. Therefore, in these tests scheduling overhead had less impact, and some performance improvements were obtained even for read operations. aIOLi provided the *best results* for large requests access patterns, increasing performance in up to 44%. On the

DOI: 10.1002/cpe

other hand, for tests that issue small requests, aIOLi's effects on performance did not surpassed its overhead, and it *decreased* performance in up to 171%. For small requests, SJF is the best choice, improving performance by up to 19%.

3.2. Performance results with multiple applications and the shared file approach

In this section, we will discuss results for the experiments where multiple (four) applications access the file system concurrently, but each application's processes share a file. In this situation, the workload is larger, but so is the scheduling algorithms' overhead, since more queues are used.

In Pastel, no scheduling algorithm was able to improve performance significantly. For this platform, aIOLi provided only small improvements (under 10%). In Graphene, aIOLi *decreased* performance for tests that issue *contiguous small read* requests in up to 54%, and presented only negligible differences for all other tests. MLF also decreased performance for contiguous small read requests access patterns, but *increased* performance by up to 17% for tests that issue non-contiguous large read requests.

In Suno, the cluster with RAID-0, where results are expected to suffer less effects of scheduling overhead, performance improvements were observed for most cases. Moreover, no algorithm provided significant performance decreases. Improvements were of up to 17% for write operations and up to 36% for reads. SJF outperformed the other algorithms for most cases, but there are no significant difference between the four scheduling algorithm's results. This indicates that gains in performance in Suno are mainly the result of requests aggregation.

For the cluster with SSDs (Edel) TO-agg is able to improve performance of *read* operations in some cases (mainly for 32 processes) by up to 22%. Nonetheless, it *decreases* performance for some of the *write* tests by up to 40%. SJF resulted in negligible performance differences only. aIOLi and MLF *decreased* performance for tests that issue non-contiguous small requests by up to 33% and 63%, respectively. However, they improved performance of *large read* requests by up to 35% (aIOLi) and up to 23% (MLF). aIOLi outperformed MLF in most cases.

3.3. Performance results for single application with the file per process approach

In the experiments where each process has an independent file, the scheduling algorithms' global criteria act to choose between the multiple queues being accessed concurrently. Moreover, aIOLi and SJF have the largest costs for selecting requests, and TO-agg has the largest cost for including requests. Although these tests are expected to suffer with larger scheduling overhead, they provide workloads that are significantly larger than what was provided by tests with the shared file approach. Since a large number of different files are accessed concurrently, results are expected to be more affected by how sequential the scheduler can make the resulting access pattern at the server, especially in clusters where the sequential to random throughput ratio is high.

The best performance improvements for these experiments were observed in the Pastel cluster: up to 38% for write operations and up to 68% for reads with 32 processes. This situation is presented in Figure 4. In general, the larger the number of processes, the more performance was increased. In these tests, aIOLi performs slightly better than MLF, but there is no significant difference between the four scheduling algorithms.

Among the experiments with large workloads, in many cases better results were obtained for read tests than write ones. One reason for this is that all platforms' sequential to random throughput ratios are higher for reads than writes, as shown in Table I. Therefore, reordering requests to generate sequential access patterns at the server has a larger impact for read operations.

In the Graphene cluster, performance improvements were obtained mainly for the access pattern with *large write* requests. In this case, SJF outperformed aIOLi decreasing tests' execution times by up to 31%. All remaining results presented only small differences under 10%. One possible reason for Pastel's results being so much better than the results obtained for Graphene (although they have close sequential to random throughput ratios) is that the tests' execution times in Pastel were 1.5 to 6 times longer than in Graphene, especially for the read tests. Therefore, the scheduling algorithms had more room to improve performance.

DOI: 10.1002/cpe

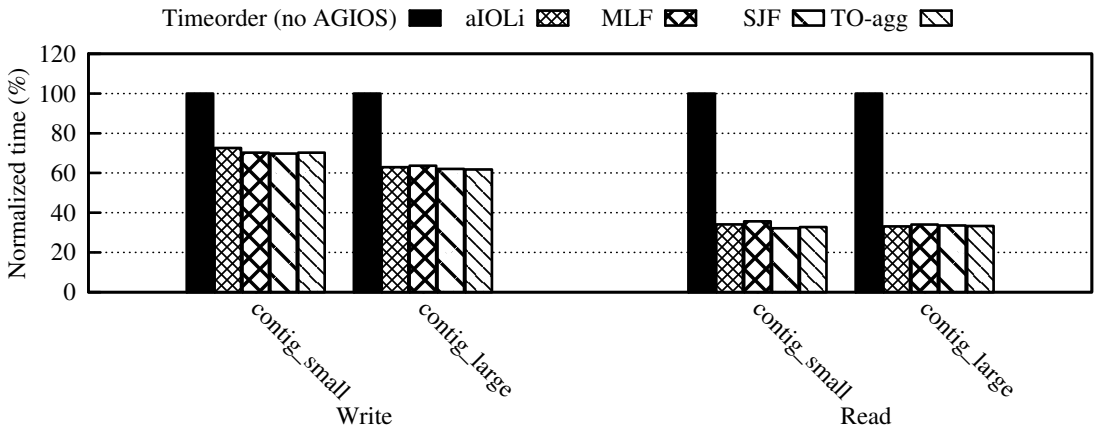


Figure 4. Results with a single application and the file per process approach in Pastel with 32 processes

Results obtained in the Suno cluster are similar from what was observed in the Pastel cluster, with no significant differences between AGIOS' four tested scheduling algorithms, and all of them providing performance improvements. SJF slightly outperformed the other algorithms for most cases. The performance improvements were smaller in Suno than in Pastel: up to 24% for write operations and up to 47% for reads.

The larger overhead caused by scheduling algorithms working to generate more sequential access patterns resulted in only small performance improvements in the Edel cluster of up to 26%, mainly for read operations. No significant performance decreases caused by the scheduler were observed either. For several cases, TO-agg outperforms aIOLi, MLF, and SJF.

3.4. Performance results with multiple applications and the file per process approach

In the tests where multiple applications concurrently access the file system and each process accesses an independent file, the workload has double the size of the one described in the last section (single application with the file per process approach). On the other hand, it generates four times more files, each file having half the size than before. Therefore, these tests are expected to suffer with more scheduling overhead.

In Pastel and Graphene, for most cases all scheduling algorithms resulted in small negligible differences. aIOLi improved performance in read operations with 8 processes per application by 16% in Pastel, and SJF improved by 16% with large read operations with 8 processes per application in Graphene.

For the Suno cluster, the one expected to be less affected by scheduling overhead, results were similar to what was obtained for the previous set of tests (in Section 3.3). Nonetheless,the observed improvements were smaller: up to 14% for write operations and up to 33% for reads. aIOLi provided slightly worse results than the others, and SJF obtained the best results (with only a small difference over MLF and TO-agg).

In Edel, for write operations most cases resulted in only negligible differences. Nonetheless, aIOLi decrease the performance by up to 19% (isolated cases only). In tests that generate read requests, aIOLi provided performance improvements of up to 24%.

4. AUTOMATIC I/O SCHEDULING SELECTION

Table III summarizes the results discussed in the last section by presenting the best choice in scheduling algorithm for all tested situations. We can see that all algorithms appear at least once, indicating that the best fit in scheduling algorithm depends on both applications' and storage devices' characteristics. In order to make AGIOS capable of selecting the adequate scheduling

DOI: 10.1002/cpe

Table III. Best choices in scheduling algorithms to all experiments.

<table>
<tr><th colspan="5">Access Pattern</th><th>Pastel</th><th>Graphene</th><th>Suno</th><th>Edel</th></tr>
<tr><td rowspan="16">x1</td><td rowspan="8">N to 1</td><td rowspan="4">contig</td><td rowspan="2">small</td><td>write</td><td rowspan="2">no AGIOS</td><td colspan="3" rowspan="2">SJF</td></tr>
<tr><td>read</td></tr>
<tr><td rowspan="2">large</td><td>write</td><td colspan="4">aIOLi</td></tr>
<tr><td>read</td><td rowspan="2">no AGIOS</td><td colspan="3" rowspan="2">SJF</td></tr>
<tr><td rowspan="4">noncontig</td><td rowspan="2">small</td><td>write</td></tr>
<tr><td>read</td></tr>
<tr><td rowspan="2">large</td><td>write</td><td colspan="4">aIOLi</td></tr>
<tr><td>read</td><td>no AGIOS</td><td colspan="3" rowspan="2">SJF</td></tr>
<tr><td rowspan="8">N to N</td><td rowspan="4">contig</td><td rowspan="2">small</td><td>write</td><td rowspan="8">aIOLi</td></tr>
<tr><td>read</td></tr>
<tr><td rowspan="2">large</td><td>write</td><td rowspan="2">TO-agg</td></tr>
<tr><td>read</td></tr>
<tr><td rowspan="16">x4</td><td rowspan="8">N to 1</td><td rowspan="4">contig</td><td rowspan="2">small</td><td>write</td><td>MLF</td></tr>
<tr><td>read</td><td>no AGIOS</td><td>TO-agg</td></tr>
<tr><td rowspan="2">large</td><td>write</td><td rowspan="4">MLF</td><td>aIOLi</td></tr>
<tr><td>read</td><td>TO-agg</td></tr>
<tr><td rowspan="4">noncontig</td><td rowspan="2">small</td><td>write</td></tr>
<tr><td>read</td><td>aIOLi</td></tr>
<tr><td rowspan="2">large</td><td>write</td><td rowspan="4">SJF</td><td rowspan="2">aIOLi</td></tr>
<tr><td>read</td></tr>
<tr><td rowspan="8">N to N</td><td rowspan="4">contig</td><td rowspan="2">small</td><td>write</td></tr>
<tr><td>read</td><td rowspan="2">aIOLi</td></tr>
<tr><td rowspan="2">large</td><td>write</td></tr>
<tr><td>read</td><td>aIOLi</td></tr>
</table>

algorithm automatically, we have decided to use machine learning to generate a decision tree based on the obtained results.

We have used the Weka data mining tool [13], that provides multiple machine learning algorithms and an interface to analyze data, apply algorithms and evaluate results. We have provided to Weka an input set that consists of one entry per executed experiment. Each entry contains:

1. Operation (read or write);
2. Number of accessed files;
3. Amount of accessed data per file;
4. Spatiality of the access pattern ("contiguous" or "non-contiguous");
5. Applications' request size ("small" or "large");
6. Sequential to random throughput ratio for the platform's storage devices;
7. Scheduling algorithm that should be used in this situation.

Each pair (access pattern, cluster) generates two entries in the input set, one for each number of processes (8 and 16). The number of processes is not included, since it is not obtainable at server side. However, it affects the number of files (in the file per process approach) or their size (in the shared file approach). The scheduling algorithm decisions from Table III were made in the context of each access pattern separately, hence both entries appoint the same scheduling algorithm selection.

To represent platforms' characteristics, we use information provided by our tool SeRRa[‡], proposed in a previous work of ours [5]. It provides the sequential to random throughput ratio for read and write operations with different request sizes. The ratio used for the decision corresponds to the operation and average request size at the server. This request size does not match the

[‡] http://www.inf.ufrgs.br/~fzboito/serra.html

DOI: 10.1002/cpe

applications' one - which we classify in small or large - but the size of requests that arrive to each server, which are a result of striping and the transmission size limit. In all our experiments, this size is 8KB.

Although the access patterns used in our evaluation were defined by a list of aspects that include number of processes and if applications' processes share a file or not, we cannot use all these aspects to build our decision tree. This happens because the server sees a stream of requests to files, and the rest of the information is lost through the I/O stack. For instance, from the servers' point of view, there is no difference between an access pattern where a single application accesses two files, and another where two applications access one file each.

All information about applications' access patterns - attributes 1-5 - is obtained from trace files of previous executions. We have applied machine learning to build a classifier capable of detecting, from a stream of requests, applications' spatiality and requests size. This detection is out of this article's scope. The spatiality and requests size detection is made to each accessed file, and the majority between all accessed files is taken to represent the access pattern. Aside from the amount of accessed data per file, all other parameters could be obtained from the scheduler's recent accesses.

Three different decision trees were generated using different subsets of the input attributes. The first tree was generated using all listed parameters. The complete input set was provided to Weka and all its available algorithms for decision trees generation were tested. Among them, the J48 algorithm provided the best results. Using 10-fold cross-validation, its resulting decision tree has a misclassification rate of 8.85%. This tree, called T_1, has 53 nodes, 27 leaves, and all provided attributes appear in the decision making. This indicates that none of them was redundant or unnecessary.

The second tree, T_2 was obtained with the original input set without the amount of data accessed from files - attribute 3. Not having this attribute would make our approach less dependent on trace files, since all other information on applications could be obtained from the scheduler's recent accesses. The resulting decision tree, computed with J48, has a misclassification rate of 8.33%, 53 nodes, 27 leaves, and also uses all provided parameters.

Finally, a third decision three, T_3 was computed without the amount of data accessed from files and the number of accessed files - attributes 2 and 3. Computed with J48, T_3 has a misclassification rate of 30.21%, 17 nodes, and 9 leaves. The next section presents an evaluation of these three decision trees.

5. SCHEDULING ALGORITHM SELECTION TREES' EVALUATION

This section describes performance results of the trees described in the last section: T_1, T_2, and T_3. To evaluate each decision tree, we go through all experiments' situations - combinations of cluster, number of applications, number of files per application, spatiality, requests' size, number of processes per application, and operation, total of 192 situations - and apply the decision tree to each situation's parameters. Then we take the results previously obtained with the selected algorithm for this situation. Since the access pattern detection is out of this article's scope, for this evaluation we consider the access pattern is always correctly detected.

There are two main aspects to consider when evaluating scheduling algorithm selection trees: the situations where they are able to select an algorithm that improves performance, and the situations where the selected algorithm decreases performance. Ideally, a perfect decision tree would improve performance for all cases. However, as evidenced by our results, for some scenarios no scheduling algorithm was able to improve performance. For these scenarios, it is not possible for one of our decision trees to improve performance.

We compare our selection trees with an "oracle" solution, which always gives the right answer according to Table III. By doing that, the oracle only decreases performance - over not using AGIOS - significantly, i.e., over 10%, in 4 cases (out of 192). From these cases, the worst degradation is 12%. Performance is increased significantly (by over 10%) in 71 cases, ranging from 11% to 59%, 23% on average. The remaining 117 cases where performance was not increased nor decreased represent the situations where no scheduling algorithm was able to improve performance.

DOI: 10.1002/cpe

It is important to notice that even the oracle decreases performance in some situations because the decisions on the best fit in scheduling algorithm were made to each access pattern with no difference between tests with different numbers of processes per application. Therefore, we have selected algorithms that lead to these small performance decreases for some tests because they are good choices in others that were considered together.

Additionally, we use two other solutions for comparison: one that always selects aIOLi, and another that always selects SJF. We use these two algorithms because they are the ones selected for the largest number of situations. The first, "aIOLi-only", results in significant performance decreases for 37 situations, by 72% on average (up to 278%). Performance is increased significantly for aIOLi-only in 66 cases, by 23% on average (up to 59%), and not affected in the remaining 89 cases.

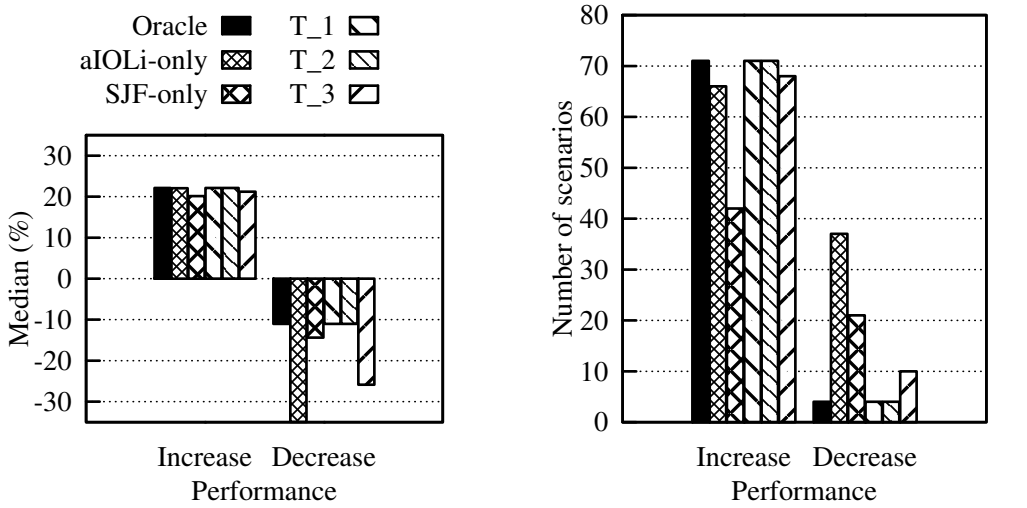
The SJF-only solution results in significant performance decreases for 21 situations, by 14% on average (up to 23%). This solution provided performance increases for 42 scenarios by 21% on average (up to 45%) and did not affect performance of 129 cases.

Table IV presents the number of correct selections performed by the different solutions - the three decision trees, aIOLi-only, and SJF-only - compared with the oracle's selections. We can see that T_1 and T_2 are able to achieve the best result possible.

Table IV. Correct selection rate of all solutions compared with the oracle.

	aIOLi-only	SJF-only	T_1	T_2	T_3
Correct selections	60 (31%)	92 (48%)	192 (100%)	192 (100%)	142 (73%)

Figure 5 compares the results obtained for the different scheduling algorithm solutions. Figure 5a presents the median performance increases and decreases. The first group of bars (performance increase) considers only the results with performance *increases* over 10%, while the second group represents results with performance *decreases* over 10%. We can see that using aIOLi-only provided the worst performance decreases.



(a) Median performance differences

(b) Situations with performance differences

Figure 5. Performance results for the tested scheduling algorithm selection trees.

Moreover, from Figure 5b, which presents the number of situations where performance was significantly increased or decreased, we can see that using only one scheduling algorithm improves performance in *less* situations and *decreases* in more.

T_1 and T_2 provide the same results as the oracle, as previously indicated. This happens despite the fact that T_2 uses less information than T_1. T_3 - the tree with the smallest number of input attributes

DOI: 10.1002/cpe

- decreases performance more (by 25% over 11% of the other trees) and for more situations (10 against 4 of other trees). However, we can still say that our simplest tree - T_3 is better than using aIOLi-only or SJF-only, since T_3 increases performance for more situations.

In order to see how these trees perform when evaluated with entries that were not included in their input sets, we generated two new versions of each tree: one using only the entries obtained with 8 processes per application, and another with the entries for 16 processes per application. We used each of these input sets to evaluate the tree generated with the other, i.e., the input set containing only entries with 8 processes per application was used to evaluate the tree generated with the input set containing entries with 16 processes only, and vice versa.

Table V presents the misclassification rates observed in this evaluation step. For T_2 and T_3 the exact same decision tree was generated with both input sets. This happens for T_3 because both input sets are the same, since they do not use attributes that depend on the number of processes per application (number of files and amount of accessed data per client). In this situation, where the same input set is used for generating and evaluating the decision tree, results indicates how well its rules represent the input set. Therefore, we can see that the attributes provided to T_3 are not enough to make a scheduling algorithm selection tree that gives the best answer in more than 75% of the times.

Table V. Misclassification rates for each decision tree's two versions, using one to evaluate the other.

	T_1	T_2	T_3
Tree generated with 8 clients	18.75%	5.21%	26.04%
Tree generated with 16 clients	5.21%	5.21%	26.04%

For T_2, which uses the number of accessed files, all comparisons with this attribute from both versions are with 1 (the number of accessed files in the shared file approach with a single application) or 4 (shared file with multiple applications), and hence not affected by the number of processes per application.

For the T_1 versions generated in this evaluation step, we observed lower misclassification rate for the tree generated with the 16 processes per application input set, which is the same tree as both T_2 versions. The J48 algorithm decided not to use the amount of accessed data per file attribute in this case, although it was available. Table VI shows this attribute for all our experiments. In the input set with 16 processes, knowing the amount of accessed data per file is only useful to identify the situation with multiple applications where each process has an independent file. Looking again at Table III, we can see that this information has a limited usefulness. On the other hand, the T_1 version generated with entries for 8 processes per application uses the amount of accessed data per file attribute to identify the situation with single application and file per process approach. Using this tree on the 16 processes per application entries, all shared file approach situations are wrongly interpreted, leading to this tree's poor results.

Table VI. Sizes of the files generated at the servers.

	Single application		Multiple applications	
	Shared file	File per process	Shared file	File per process
8 processes	128MB	256MB	128MB	128MB
16 processes	256MB	256MB	256MB	128MB

These results indicate that the attribute that gives the amount of accessed data per file may contribute to generate decision trees that are overfitted to the input set. Furthermore, not including this attribute (in T_2) did not affect the trees' results. On the other hand, not including the number of files attribute significantly affect results, since sequentiality and requests size are not enough to represent all tested access patterns.

Although we evaluated the scheduling algorithm selection trees by comparing them with an oracle solution, the real alternative to them is using only one scheduling algorithm for all situations, without double adaptivity. In this sense, our approach provides performance improvements of up to 75%

DOI: 10.1002/cpe

over aIOLi and of up to 38% over SJF. Moreover, in general, the decision trees are able to improve performance (over the base timeorder scheduler, without using AGIOS) for more situations, and decrease performance for less. Table VII summarizes these results.

Table VII. Improvements provided by the scheduling algorithm decision trees over other solutions.

	T_1	T_2	T_3
Improvement over **aIOLi** (up to)	74.9%	74.9%	74.9%
Situations with performance increase	5% more	5% more	1% more
Situations with performance decrease	89% less	89% less	74% less
Improvement over **SJF** (up to)	38%	38%	38%
Situations with performance increase	64% more	64% more	58% more
Situations with performance decrease	80% less	80% less	54% less

6. RELATED WORK

This section discusses related work on I/O scheduling. I/O scheduling techniques are applied to alleviate interference effects by coordinating requests processing. This coordination can take place on client-side or server-side. However, client-side I/O coordination mechanisms [14, 15] are still prone to interference caused by concurrent accesses from other nodes to the shared file system. Therefore, server-side I/O scheduling is more usual than the client-side approach.

Chen and Majumdar [16] proposed an algorithm called *Lowest Destination Degree First* (LDDF) that represents processes and servers as nodes of a graph, with edges meaning that I/O requests from a process can be treated by a server. Servers are then given degrees depending on how many requests they can process, and requests are assigned by following the non-increasing degree ordered servers list. This LDDF algorithm counts on data replication, so each request have multiple options of servers for processing. Moreover, it assumes a centralized control over all processes and all servers. In a large-scale environment, such a centralized control would impose a bottleneck and compromise scalability.

In their paper, they also evaluated algorithms to be used locally at the servers, after the first assignment done by the LDDF algorithm. The best performance was observed when using *Shortest Job First* (SJF) for local scheduling - over *First Come First Served* (FCFS). In their implementation, jobs' size is given by their total amount of requested data. Therefore, authors argue that better scheduling is achieved when *considering information about applications*. Their results motivated us to include SJF in our study. Additionally, another reason to include it was because a similar algorithm - *Shortest Wait Time First* (SWTF) - was reported to present good results as a disk scheduler for SSDs [4].

An approach named IOrchestrator was proposed by Zhang et al. [17] to the PVFS2 parallel file system. Their idea is to synchronize all data servers to serve only one application during a given period of time. This decision is made through a model considering the cost of this synchronization and the benefits of this dedicated service. In addition to modifications in the file system, their approach also requires modifications in MPI-IO in order to make it possible for the scheduler to know which files each application accesses.

The same approach was adapted to provide QoS support for end users by the same authors [18]. Through a QoS performance interface, requirements can be defined in terms of execution time (deadline). Applications need a profiling execution, where their mechanism obtains its access pattern. This access pattern considers time portion used for I/O, average requests size, and average distance between requests inside each time slice (called "epoch"). A machine learning technique is used to translate the provided deadline to requirements in bandwidth from the file system, using the profiled access pattern. Their approach is similar to ours in the sense that it uses information from previous executions to detect applications' access patterns. Nonetheless, we use a more detailed access pattern classification, considering more aspects such as number of files and operation, and

DOI: 10.1002/cpe

also considering storage devices' characteristics as a factor that affects the resulting performance. Furthermore, our approach is not deadline-oriented, as our work does not aim at providing QoS.

Both approaches - IOrchestrator and its QoS support version - are limited to situations with a centralized meta-data server which, in this case, is responsible for the synchronization and global decision making. This centralized architecture can present scalability issues at large scale. Our approach sacrifices the ability of making global decisions in order to avoid this centralization point.

Song et al. [19] proposed a scheme for I/O scheduling through server coordination that also aims at serving one application at a time. For this purpose, they implemented a *window-wide* coordination strategy by modifying PVFS2 and MPI-IO. Requests from clients carry a global application ID and a timestamp. At the server, they are separated in time windows, where the different windows must be processed in arrival time order to avoid starvation. Inside each window, requests are ordered by application ID. They do not use global synchronization, and argue that all servers decide for the same order since they use the same method. We cannot give the same guarantees about our approach, because we do not use global applications identifiers and timestamps. To obtain this information would require modifications in the file system and I/O libraries, compromising portability and making the approach less generic

Another difference between their approach and ours is that theirs do not seek at generating contiguous access patterns. They decided not to focus on hard-disks, aiming at a more generic solution. Although SSDs usage has been increasing, HDDs are still the solution available in most HPC architectures. This holds especially at the file system infrastructure, where storage capacity is a limiting factor. Additionally, as evidenced by our results, access sequentiality is not a desirable characteristic only for HDDs, and performance can also be improved by requests aggregation.

Lebre et al. [7] proposed the aIOLi scheduling algorithm, used in this work. The algorithm was proposed in the context of an I/O scheduling framework (also called "aIOLi"). Their framework aims at being generic, non-invasive and easy to use. The development of our AGIOS tool was vastly inspired by their work. The main differences between both tools is that aIOLi is a Linux kernel module, while AGIOS also offers an user-level library, since today's most parallel file systems' servers work at user-level. Moreover, aIOLi was only used with a centralized file system (NFS), while our work with AGIOS focused on parallel file systems. In scheduling algorithm choices, aIOLi offered its aIOLi algorithm and a simple timeorder. Our study included five scheduling algorithms.

Qian et al. [20] used the aIOLi algorithm for the creation of a *Network Request Scheduler* (NRS) for the Lustre parallel file system. Instead of working in a centralized file system, like aIOLi, each instance of NRS works in the context of a Lustre's data server. There is no global coordination of accesses. Their successful use of the aIOLi scheduling algorithm in the context of a parallel file system's data servers motivated the inclusion of this algorithm in our study.

7. CONCLUSIONS AND FUTURE WORK

This article focused on I/O scheduling as a tool to improve performance by alleviating interference effects. Our work aimed at providing I/O scheduling for parallel file systems with double adaptivity: to applications and storage devices.

We have presented AGIOS, our I/O scheduling tool, and its five scheduling algorithms: aIOLi, MLF, SJF, TO, and TO-agg. Through an extensive performance evaluation over four clusters under different access patterns, we have shown that both applications' access patterns and storage devices' sensitivity to access sequentiality affect I/O scheduling efficacy. We have used machine learning to build decision trees able to select the best fit in scheduling algorithm for different situations.

Multiple decision trees were generated using different input parameters. All of them provided better overall results than approaches where the same scheduling algorithm is always used. Comparing with situations where aIOLi or SJF are always used, our approach represents a performance improvement of up to 75%, *increasing* performance for up to 64% *more* situations and *decreasing* performance for up to 89% *less* situations. We have shown that it is possible, through machine learning, to select the best fit in scheduling algorithm to each situation automatically.

DOI: 10.1002/cpe

Moreover, our results indicate that both applications' and platforms' characteristics are essential for correctly selecting the best I/O scheduling algorithm in a given situation.

Our tool, with its I/O scheduling algorithms and the ability to automatically select between them through the discussed T_2 decision tree, is freely available at http://www.inf.ufrgs.br/~fzboito/agios.html.

All tested scenarios provided homogeneous access patterns: all applications have the same access pattern, and all files are accessed in the same way. As future work, we intend to expand this study by investigating the impact of heterogeneous access patterns on I/O scheduling results.

ACKNOWLEDGEMENTS

This research has been partially supported by CNPq and CAPES-BRAZIL under the grants 5847/11-7 and Stic-Amsud 6132-13-8. The experiments presented in this article were carried out on the Grid'5000 experimental test bed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr). This research was accomplished in the context of the International Joint Laboratory LICIA and of the HPC-GA project.

REFERENCES

1. Thakur R, Gropp W, Lusk E. Data sieving and collective i/o in romio. *frontiers* 1999; :182.
2. Boito FZ, Kassick RV, Navaux POA. The impact of applications' i/o strategies on the performance of the lustre parallel file system. *International Journal of High Performance Systems Architecture* 2011; **3**(2):122–136.
3. Chen F, Koufaty DA, Zhang X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, 2009; 181–192.
4. Rajimwale A, Prabhakaran V, Davis JD. Block management in solid-state devices. *Proceedings of the USENIX Annual Technical Conference*, 2009; 279–284.
5. Boito FZ, Kassick RV, Navaux PO, Denneulin Y. Towards fast profiling of storage devices regarding access sequentiality. *Applied Computing (SAC), 2015 ACM Symposium on*, 2015 (to appear).
6. Boito FZ, Kassick RV, Navaux PO, Denneulin Y. Agios: Application-guided i/o scheduling for parallel file systems. *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*, 2013; 43–50.
7. Lebre A, Denneulin Y, Huard G, Sowa P. I/o scheduling service for multi-application clusters. *Proceedings of IEEE Cluster 2006, conference on cluster computing*, 2006.
8. Silberschatz A, Galvin PB, Gagne G. *Operating system concepts*, vol. 8. Wiley, 2013.
9. Avila RB, Navaux POA, Lombard P, Lebre A, Denneulin Y. Performance evaluation of a prototype distributed nfs server. *16th Symposium on Computer Architecture and High Performance Computing*, 2004; 100–105, doi: 10.1109/SBAC-PAD.2004.33.
10. Bolze R, Cappello F, Caron E, Dayde M, Desprez F, Jeannot E, Jegou Y, Lanteri S, Leduc J, Melab N, *et al.*. Grid5000: A large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications* 2006; **20**(4):481–494, doi:10.1177/1094342006070078. URL http://hpc.sagepub.com/cgi/content/abstract/20/4/481.
11. Byna S, Chen Y, Sun XH, Thakur R, Gropp W. Parallel i/o prefetching using mpi file caching and i/o signatures. *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008; 44.
12. Corbett P, Feitelson D, Fineberg S, Hsu Y, Nitzberg B, Prost JP, Snir M, Traversat B, Wong P. Overview of the mpi-io parallel i/o interface. *KLUWER INTERNATIONAL SERIES IN ENGINEERING AND COMPUTER SCIENCE* 1996; :127–146.
13. Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH. The weka data mining software: an update. *ACM SIGKDD explorations newsletter* 2009; **11**(1):10–18.
14. Ohta K, Matsuba H, Ishikawa Y. Improving parallel write by node-level request scheduling. *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid-Volume 00*, IEEE Computer Society, 2009; 196–203.
15. Dorier M, Antoniu G, Cappello F, Snir M, Orf L. Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free i/o. *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, 2012; 155–163, doi:10.1109/CLUSTER.2012.26.
16. Chen F, Majumdar S. Performance of parallel i/o scheduling strategies on a network of workstations. *Parallel and Distributed Systems, International Conference on* 2001; **0**:0157, doi:http://doi.ieeecomputersociety.org/10.1109/ICPADS.2001.934814.
17. Zhang X, Davis K, Jiang S. Iorchestrator: Improving the performance of multi-node i/o systems via inter-server coordination. *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, IEEE Computer Society: Washington, DC, USA, 2010; 1–11, doi: http://dx.doi.org/10.1109/SC.2010.30. URL http://dx.doi.org/10.1109/SC.2010.30.
18. Zhang X, Davis K, Jiang S. Qos support for end users of i/o-intensive applications using shared storage systems. *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage*

and Analysis, SC '11, ACM: New York, NY, USA, 2011; 18–1, doi:10.1145/2063384.2063408. URL http://doi.acm.org/10.1145/2063384.2063408.
19. Song H, Yin Y, Sun XH, Thakur R, Lang S. Server-side i/o coordination for parallel file systems. *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, ACM: New York, NY, USA, 2011; 17–1, doi:10.1145/2063384.2063407. URL http://doi.acm.org/10.1145/2063384.2063407.
20. Qian Y, Barton E, Wang T, Puntambekar N, Dilger A. A novel network request scheduler for a large scale storage system. *Computer Science-Research and Development* 2009; **23**(3):143–148.

DOI: 10.1002/cpe