AGIOS: Application-guided I/O Scheduling 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 – Porto Alegre, Brazil
{fzboito, rvkassick, navaux}@inf.ufrgs.br
[2]LIG Laboratory – INRIA – University of Grenoble – Grenoble, France
{yves.denneulin}@imag.fr

***Abstract*—In this paper, we improve the performance of server-side I/O scheduling on parallel file systems by transparently including information about the applications' access patterns. Server-side I/O scheduling is a valuable tool on multi-application scenarios, where the applications' spatial locality suffers from interference caused by concurrent accesses to the file system. We present AGIOS, an I/O scheduling library for parallel file systems. We guide scheduler's decisions by including information about the applications' future requests. This information is obtained from traces generated by the scheduler itself, without changes in application or file system. Our approach shows performance improvements under different workloads of 46.3% on average when compared to a scenario without an I/O scheduler, and of 25.1% when compared to a scheduler which does not use information about future accesses.**

I. Introduction

High Performance Computing (HPC) applications 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 the 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], [3], [4].

In HPC architectures the parallel file system infrastructure is usually shared by all applications. When multiple applications concurrently access the same file system, their requests can arrive interleaved in the servers. We call this phenomenon *interference*, as illustrated in Fig. 1. In this example, two applications ("App. A" and "App. B") concurrently access a server for the files named A and B. Although each application's individual access pattern is ideal for optimization by the file system, they will achieve poor performance because the server will not observe the same contiguous access pattern as the applications. In this situation, server-side I/O scheduling is a valuable technique to improve application performance.

During the execution of an application, a scheduler must make several decisions about incoming requests. However, scheduler working on server-side in a stateless parallel file system usually lacks information about applications and their access patterns. As a result, a scheduler may not be able to help

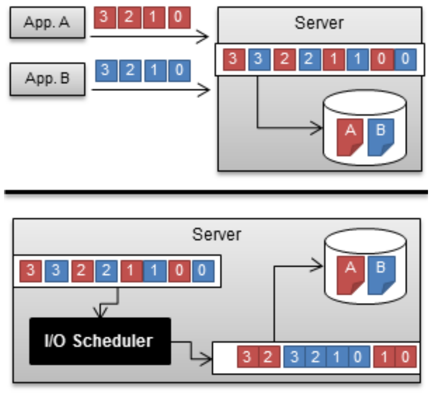


Fig. 1. Interference from concurrent accesses to the PFS.

applications as much as it would otherwise. In this paper, we propose the use of application information on the server-side I/O scheduler in order to improve its decisions and achieve better results.

We developed an I/O scheduling library named *AGIOS* . Our library can be easily integrated into a parallel file system's server code to manage incoming I/O requests using a variation of the Multi-Level Feedback algorithm [5]. This algorithm uses a quantum-based approach to schedule requests coming from different applications, improving performance while trying to keep fairness in the resource sharing and applications' response time. We illustrate the use of the library in a NFS-based parallel file system.

Our scheduler obtains information on future accesses to its server via trace files from previous executions of the applications. This information is then used to predict possible requests aggregations and to guide the scheduler's decisions during execution. When compared to a version of the library which does not use such information, our approach led to performance improvements of up to 35.4% - 29.5% on average - for write operations and of up to 31.4% - 20.7% on average - for read operations.

We show that using information collected automatically about applications' access patterns can improve the service offered by the scheduler and, consequently, by the file system. To the best of our knowledge, there is no other I/O scheduling technique which uses such information to improve its performance.

The remaining of this paper is organized as follows: related works are discussed in Section II. Section III describes AGIOS's base scheduling algorithm and its usage with a parallel file system. Section IV presents our new approach which involves the use of information from traces to improve the scheduler's decisions. Performance results are discussed in Section V. Final remarks and future work are presented in Section VI.

II. Related Work

There are several client-side optimizations that adjust an application's access pattern to achieve better performance from the parallel file system [1], [6], [7]. As they work on the context of one application, in multi-application scenarios their effects can be impaired by interference caused by concurrent accesses to the file system [5]. I/O scheduling techniques are applied in this situation to coordinate the execution of the requests. This coordination can take place on client-side or server-side.

However, client-side I/O coordination mechanisms [8], [9] are still prone to interference caused by concurrent accesses from other nodes to the shared file system. Moreover, techniques to globally perform I/O scheduling on client-side would require synchronization between all the processing nodes, impairing scalability. For these reasons, server-side I/O scheduling is more usual than the client-side approach.

An approach named IOrchestrator is proposed by Zhang et al. [10]. Their idea is to synchronize all data servers to serve only one application during a given period of time. This decision is made considering the cost of this synchronization and the benefits of this dedicated service. The same approach was adapted by the same authors to provide QoS support for end users [11]. Through a QoS performance interface, requirements can be defined in terms of execution time (deadline). A machine learning technique is used to translate this deadline to requirements from the file system. Both approaches 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. proposed a scheme for I/O scheduling through server coordination that also aims at serving one application at a time [12]. This comes from the observation of implicit and explicit synchronizations on the applications while doing I/O, which cause them often to wait until all the involved requests are finished. For this purpose, they implemented a *window-wide* coordination strategy by modifying PVFS2 and MPI-IO, and obtained performance improvements of up to 40%. Similarly to AGIOS, there is no global coordination, since each scheduler instance works in the context of one server. As all servers use the same method to select the next served requests, one can expect them to process the same application's requests almost at the same time. Our base scheduling algorithm's performance is not greatly impaired by applications' synchronizations because of its iterative nature: requests from different queues are selected following a FIFO criterion, avoiding that they spend too much time in the server before being executed.

The approach from Song et al. does not explore spatial locality; they do not focus on hard-disks, aiming at a more generic solution. Although SSDs usage has been increasing, HDs are still the solution available in most HPC architectures. This holds specially at the file system infrastructure, where storage capacity is a limiting factor. Additionally, their solution requires modifications in the file system and in the I/O library.

The algorithm proposed with aIOLi [5] was used with the Lustre file system by Qian et al. [13]. They describe the incorporation of the scheduling algorithm in the data servers' code with no global coordination, resulting in performance improvements of up to 40%. This is similar to our work in the sense that we use the same scheduling algorithm as a base. However, our library aims at being generic and easily plugged into any I/O subsystem code, while their approach is specific to one file system. Moreover, we improve the scheduling algorithm by including information about applications' accesses.

Several techniques to improve performance need information about the applications' access patterns for the decision making process. Prefetching techniques and cache substitution policies [14], [15], [16], [17], [18], [19] are common examples. This information is obtained in different ways: using machine learning techniques [14], [20], by looking for groups of blocks or files that are commonly accessed together [17], [18], by information explicitly inserted by in the application code [19], [21], from a speculative pre-execution of the application [16] or from traces [15].

The presence of interference caused by multiple applications concurrently accessing the shared file system makes the detection of stable access patterns very challenging. Moreover, in stateless file systems, little information is available to the data server about where the requests are coming from. Because of that, we decided to obtain information on the scheduler from traces, generated by the scheduler itself. This approach minimizes the modifications needed in the file system code, keeping our library generic.

This section described some server-side I/O coordination techniques [10], [11], [12], [13]. However, although information about access patterns is used for different optimization techniques, to the best of authors' knowledge, this is the first work to propose and demonstrate its use to improve I/O scheduling.

III. I/O Scheduling for Parallel File Systems

This section presents an I/O scheduling library for parallel file systems named *AGIOS*. The library, intended to be easily pluggable with today's parallel file systems, must be included in the file system's server source code. Fig. 2 illustrates an example where a PFS server included the AGIOS library and uses it to schedule requests – the red line represents the requests flow from the client machine to the server and then back to the client. The interaction between the scheduler and the server is done as follows:

1) **Initialization:** The server calls *agios_init* in order to initialize the AGIOS infrastructure. In this moment, the server must provide a callback function used to process a single request. If available, a function capable of processing a list of requests at once can also be provided.

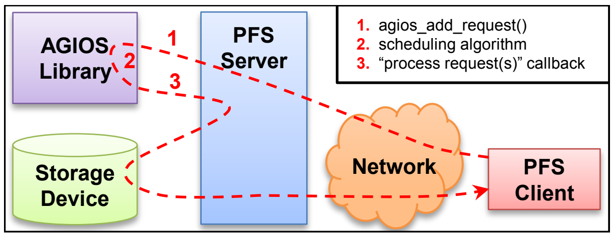


Fig. 2. AGIOS library placement and use.

2) **A new request arrives:** The server uses the *agios_add_request* call to transmit it to the scheduler.
3) **Requests are ready to be processed:** The scheduler passes them back to the server through the provided callback function (see item 1). Therefore, the task of processing the requests is left to the clients. This keeps the tool's interface simple and generic.

AGIOS uses the scheduling algorithm proposed with the aIOLi framework [5]. This framework was presented in the context of a centralized NFS file system and implemented as a kernel module. Its scheduling algorithm is a variation of the Multilevel Feedback (MLF) which seeks to promote aggregation of requests into larger ones. The resulting algorithm works by iterating through the following steps:

- 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 the other for writes.
- New requests receive an initial quantum of 0 seconds.
- 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.
- All quanta (including the virtual requests' ones) are increased by a fixed value, the estimated amount of time needed to process a request of a given size.
- In order to choose a request to be served, the algorithm uses an offset order inside each queue and a FIFO criterion between the different queues. Additionally, in order to be selected, the request's quantum must be large enough to allow its whole execution. The scheduler decides if it is large enough by estimating the time the request will take to be served (given its size), and this time must be smaller or equal to the current quantum attributed to this request. In AGIOS this processing time is estimated from microbenchmarking prior to its use.

The virtual request will be served as a single request if a callback for this functionality was provided by the library user (the server). Otherwise the original requests that were selected for aggregation will be served sequentially. However, even when it is not possible to effectively aggregate the requests into larger ones, performance will still benefit from the execution of contiguous requests served in offset order, instead of having random requests.

This scheduling algorithm was meant for multi-application scenarios, aiming at avoiding the ill effects of interference in application performance. Additionally, different processing nodes can also cause interference on each other, even when executing the same application. However, when working with stateless file systems, the server is often unable to determine from which application or process requests are coming. For this reason, the scheduler works in the context of files instead of applications. Since it is not usual for different applications to access the same files at the same time, the scheduler is able to present the expected behavior and avoid interference while keeping some degree of fairness and maintaining the response time of the applications. Moreover, the scheduler is able to provide performance improvements even on single-application scenarios.

A. AGIOS with a Parallel File System

For proof-of-concept purposes, we present our library's usage with dNFSp [22], a NFS-based parallel file system composed by several meta-data and data servers (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. Since the tool works in a centralized fashion, each IOD contains an independent instance. Therefore, the use of the scheduling library does not provide a global request coordination, but tries to improve access performance for each server.

An IOD receives requests from meta-servers, and includes them in a queue to be processed in a FIFO order. To use AGIOS, some minor changes had to be performed: first, when I/O requests are received, they are included in the scheduler queues through the *agios_add_request* function. The IOD's callback function, provided on initialization, is called by AGIOS when a request is scheduled. This function simply includes the request in the IOD queue. Since it is a FIFO queue, we guarantee that the requests will be served in the order defined by the scheduler.

B. Request Reordering vs. Aggregation

The described scheduling algorithm improves performance in two ways:

- **Reordering of requests:** assuming that the performance of executing I/O operations contiguously is better than to execute them randomly, the scheduler tries to serve requests in offset order.
- **Aggregations:** this approach assumes that an application can achieve better performance when accessing a parallel file system if its requests are fewer and large rather than many small ones, thus the scheduler aggregates contiguous requests.

In order to understand the impact of these approaches on the scheduler's performance, we executed some experiments with a version of the algorithm which does not perform aggregations. This test was executed with a dNFSp configuration with 6 data servers. A single application was executed with 40 processes on 40 nodes (processes do not share a node). Results

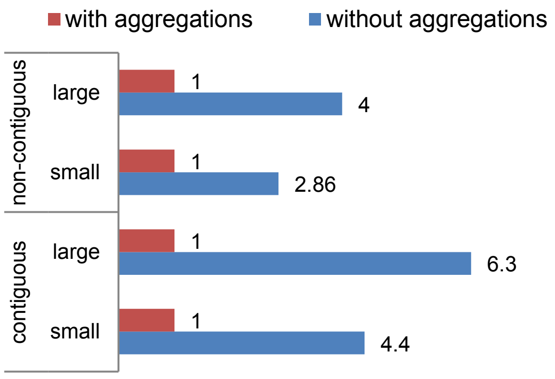


Fig. 3. Impact of aggregations (normalized application execution time).

for different access patterns are presented in Fig. 3, normalized by the time obtained with the regular scheduler.

These results show that scheduling without performing aggregations can increase the execution time of an application (and, thus, worsen performance) by up to 6.3 times when compared to the regular scheduling algorithm. Although reordering requests is important to restore the spatial locality of the access pattern, without the aggregations the overhead of the scheduling mechanism becomes dominant. This result illustrates the impact requests aggregations have on the scheduler results. The next section describes our approach to improve aggregations size and number by foreshadowing applications' requests.

IV. Application-guided Scheduling with AGIOS

In order to improve the aggregations performed by AGIOS's scheduling algorithm, we have decided to include information about applications' access patterns. We believe that, if the scheduler can know how future requests will be, it can make better decisions about its aggregations. We developed a new module for the library named *Prediction Module*. This module obtains information about the application through trace files.

A. Trace Files Generation and Prediction of Future Requests

The trace file is generated by the scheduler itself and stored on local disk, with no modifications to the application or to the file system. In this trace file, a "new request" entry stores the file identifier, offset, size, and timestamp of the request (the number of nanoseconds elapsed since the current trace's first request arrival, or 0 in the case of the first request). In the experiments presented in this paper, each client process generated **traces** of around 6.5KB in each server, 26KB per client counting the 4 servers, or 25.57 bytes of trace per KB accessed (2.5%). The generation of traces is activated by a library configuration parameter. We observed an overhead on execution time of 10.98% on average induced by tracing.

Different traces generated by executing the same application may present some variation between the arrival times of the same request. In order to obtain more realistic arrival time estimations, several trace files can be combined. In our simulations, we have combined 6 traces by taking the arithmetic means of the different values for each request's arrival time, and all standard deviations remained under 10% of the mean. We have observed that this variation on arrival times is not large enough to lose the access pattern's characteristics.

The prediction module is initialized by the scheduler in the beginning of its execution if trace files are present. A prediction thread then reads the traces and generates a set of queues identical to the ones used during execution, except that these contain predicted future requests. This initialization can also be triggered during execution through a function called by the library's user (the server). This provides the ability to generate a trace file during some initial period of the execution and then start the prediction module if we expect the traced access pattern to happen again in the future. However, for the results presented in this paper, we used the trace files provided at initialization approach only.

B. Predicting Requests' Aggregations

After obtaining a list of all future requests, the prediction thread then evaluates all possible aggregations. For every request R_i in the trace, we can obtain its **time of arrival** $AT_\mathbf{i}$ and its **size** $S_\mathbf{i}$. We also estimate, by micro-benchmarking, a **time to process function** $TTP(x)$ for x being a request size.

If contiguous predicted requests R_1 and R_2 are observed, the scheduler evaluates their aggregation considering:

- The estimated times to process R_1 and R_2 independently ($TTP(S_1) + TTP(S_2)$), and as an aggregated virtual request ($TTP(S_1 + S_2)$). If processing them together takes less time than separately, performance could benefit from this aggregation.
- The difference between their predicted arrival times. If R_1 will have to stay in the scheduler queue waiting for a long time before R_2 arrives to enable the aggregation, then maybe there would be no benefit from this aggregation.

This means that we should aggregate two contiguous requests when the time to process them separately is big enough (when comparing with the time to process them aggregated) to make it worth keeping the request in queue. This is a very conservative approach, since it does not consider that accepting to process the requests separately may have the extra cost of interrupting the spatial locality by having a non-contiguous request from another queue processed between them. Moreover, the time between the two requests is considered as an extra cost of the aggregation. However, this time does not have to be wasted, since it can be spent processing requests from other queues. In order to make this decision less conservative, we included a factor α that represents the ability to overlap the waiting time with processing other requests.

Therefore, R1 and R2 should be aggregated if

$$TTP(S_1) + TTP(S_2) >$$
$$TTP(S_1 + S_2) + (|AT_2 - AT_1|) * (1 - \alpha)$$

This factor α can be obtained by looking at the requests predicted to arrive to the server between R_1 and R_2 and considering their sizes and the different queues they are added to. On the other hand, for this to make sense in a multi-application scenario, it is necessary to have traces for all applications that will execute concurrently (and only for

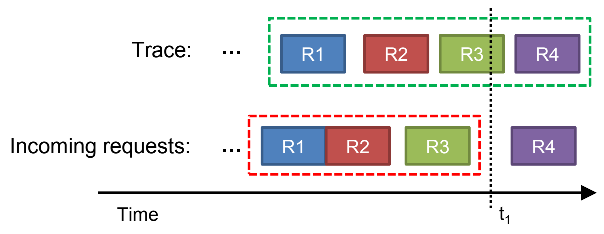


Fig. 4. Scheduling with future requests: in the instant t_1, the scheduler needs to decide if it dispatches the three highlighted incoming requests for execution (in red) or if it waits longer. With information from the trace, the scheduler knows that a bigger aggregation is possible for these requests.

them). An alternative is to periodically update the predicted aggregations by adjusting the α according to the concurrency level observed by the scheduler.

This aggregation analysis is made to two requests at a time. If their aggregation is predicted, then they will be considered as one virtual request, and this virtual request will be analyzed for aggregation with the next contiguous request considering the same criteria and so on. Therefore, it is possible to predict aggregations of any size (and not only two requests). On the other hand, if the aggregation of these two requests is not considered advantageous, the second one (by offset order) will be tested for aggregation with the next contiguous request (if existent).

C. *Including Predictions on the Scheduling Algorithm*

When an actual request arrives, the scheduler looks for predicted requests to the same file with the same offset, size and with a relative timestamp (relative to the first request to this file) within acceptable bounds. If the scheduler finds such a request, the two versions (predicted and actual) are linked and the predictions concerning this request will be considered during scheduling. The limitation of this approach is that, when using traces that are not from the same execution, the scheduler needs to know the name of the file or an unique identifier that will not change between executions.

It is important to notice that, when the current access pattern is not the same as what was traced, most of the requests will not find a correspondent prediction. In this case, the scheduler will work normally as it would do without the prediction module. Therefore, mispredictions are not expected to result in a significant decrease in performance.

When a virtual request is selected to be served, the scheduler asks the prediction module if it should be done now, or if it should wait for some period of time. In order to make this decision, the prediction module analyzes the aggregation predicted for the traced versions of the requests. If the aggregation is not as big as the predicted one, the scheduler must wait. In order to avoid starvation, the wait will happen only once for each virtual request. The waiting time will be obtained by subtracting the elapsed time since the first request of the performed aggregation arrived from the difference between the predicted aggregation's first and last requests' arrival times. This process is illustrated in Fig. 4.

After the processing of actual requests, predicted versions will not be discarded. Therefore, repeating access patterns (or repeated executions of the same application) can benefit several times from the same predictions. Traces can be reused for applications' executions with different parameters as long as they do not change what file portions are accessed.

V. Performance Evaluation

This section presents results obtained for the AGIOS Library with the dNFSp file system. We start by presenting our experimental methodology. Subsection V-B shows the performance improvements obtained by using the base scheduling algorithm (discussed in Section III) over not using a scheduler. In subsection V-C, results obtained by our new approach (the library with the Prediction Module, as presented in Section IV) are discussed.

A. *Experiments' Methodology*

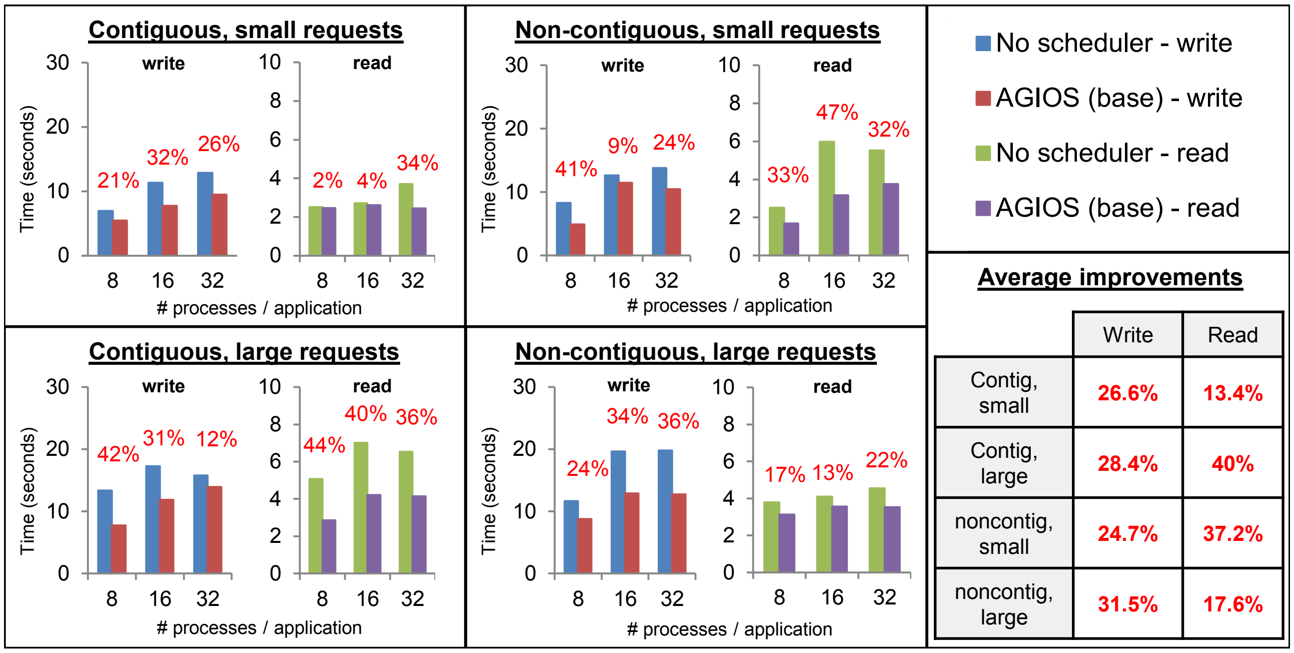
Our tests were executed in the Edel cluster from the Grid5000 testbed, which has 72 bi-processor nodes with 4-core Intel Xeon processors, and 24GB of memory per node. We used 36 of these nodes, reserving 4 of them to the dNFSp file system: our testing environment has 4 data servers, one of them being simultaneously a meta-data server (an usual configuration for dNFSp).

Although dNFSp allows for multiple meta-data servers, we have chosen to use a single one in our experiments. In a large scale, this centralized server could become a bottleneck and impair performance, but this did not happen for these experiments. It is important to notice, however, that we are evaluating the scheduler on the data servers, so this choice is not expected to impact our analysis.

We developed a set of tests using the MPI-IO Test benchmarking tool [23] in order to simulate the way that scientific applications perform their I/O operations. We explore the following list of aspects:

1) **Spatial locality:** if applications issue requests that are contiguous or non-contiguous. In our tests, the non-contiguous case is represented by an one-dimension strided access pattern.
2) **Size of requests:** if the requests are small (smaller than the file system stripe size) or large (larger than the stripe size and large enough so all the 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. [15]. In our simulations, **each process** issues 128 requests of size 16KB on the small test, and 8 requests of size 256KB in the large one. The stripe size used by dNFSp is 32KB.
3) **Number of processes:** how many processes perform I/O. We repeat our tests for different application sizes, using up to all available cores in the nodes. Since the amount of data accessed by each process is constant, **the total amount of data grows with the number of processes**.

The benchmarking tool uses synchronous I/O operations, so all requests from an application will have to be served before the next batch can be sent to the server. However, the NFS client has a maximum request size of 8KB. Therefore,



	Write	Read
Contig, small	26.6%	13.4%
Contig, large	28.4%	40%
noncontig, small	24.7%	37.2%
noncontig, large	31.5%	17.6%

Fig. 5. Results for AGIOS with the base scheduling algorithm. The numbers in red represent performance improvement over not using a scheduler.

application requests are divided in multiple actual requests: in the small tests, each process will issue 2 requests at once; in the large tests, 32. Greater performance improvement could be achieved if our tests issued only asynchronous operations, since the application would not be affected by AGIOS scheduling algorithm's waiting times. We chose the synchronous approach since it is more challenging to the scheduler, and highlights the importance of maintaining applications' response time and avoiding requests starvation.

We simulate multi-application scenarios by running 4 instances of the same benchmark at the same time. In these cases, processing nodes are split among the different instances. From each application execution, we take the completion time of the slowest process, since it defines the execution time. We take the maximum between the concurrent applications' execution times because we are interested in reducing the makespan, as it represents the time to process the whole workload from the point of view of the file system.

All results in this paper are the arithmetic mean of at least 4 executions, repeated until presenting an acceptable variation. Different executions are separated by a restart in the file system to avoid effects from caching.

B. I/O Scheduler's Impact on Performance

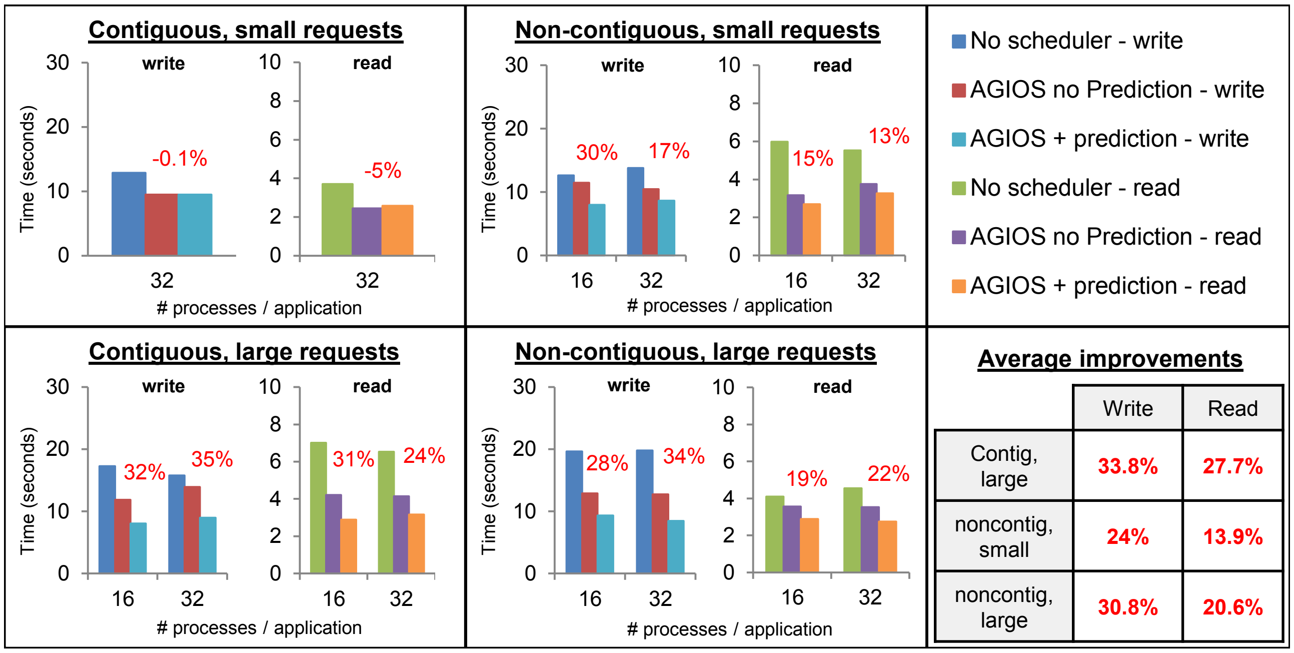
Fig. 5 shows the results obtained with our library using the base scheduling algorithm only (without the prediction module) over not using a scheduler. Each rectangle contains the results for one of the four tested access patterns. The write test is presented in the left (blue and red bars) and the read test in the right (green and purple bars). The x-axes represent the number of processes per application and the y-axes represent time in seconds. The numbers in red above each bar give the performance improvement, obtained dividing the difference between the values of the two bars by the first one (time of the test without the scheduler). In the inferior right corner of the figure, a table presents the average performance improvements for each access pattern (the arithmetic average of the improvements obtained for the 3 numbers of processes).

The use of AGIOS's base scheduling algorithm resulted in performance improvements of up to 42.05% - 27.81% on average - for write operations and of up to 46.95% - 27.06% on average - for read operations. No significant decrease in performance was observed by including the scheduling algorithm to the file system. We can observe that, on average, read and write operations benefited equally from the scheduling.

C. Results with the Prediction Module

Fig. 6 presents the results for the AGIOS library with the prediction module and compares it with the previously presented results (the library without the prediction module and the file system without a scheduler). The numbers in red above the bars show the difference between the two versions of the scheduler, and so the table on the lower right corner. The average aggregation sizes from the presented results are listed in Fig. 7. The numbers in red above the bars give the improvement in aggregation size of the prediction module over the base scheduling algorithm. We can see that our approach led to aggregations 25.1% bigger on average.

In tests with contiguous requests, each process has a dedicated file portion, so requests coming from one process are not usually contiguous to requests coming from other processes. In these tests all aggregation opportunities come from requests issued by the same process. As discussed in Section V-A, our tests use synchronous requests, and application's requests are divided in multiple actual requests because of the NFS protocol. In the test with contiguous and small requests, the 16KB application's requests are divided in two 8KB requests to the file system. In this case, the maximum possible aggregation size is 2, since these two requests are contiguous, but not contiguous to the requests of other processes. Also, because



	Write	Read
Contig, large	33.8%	27.7%
noncontig, small	24%	13.9%
noncontig, large	30.8%	20.6%

Fig. 6. Results with the AGIOS's prediction module for 4 concurrent applications. Numbers in red show the difference between the two scheduler versions.

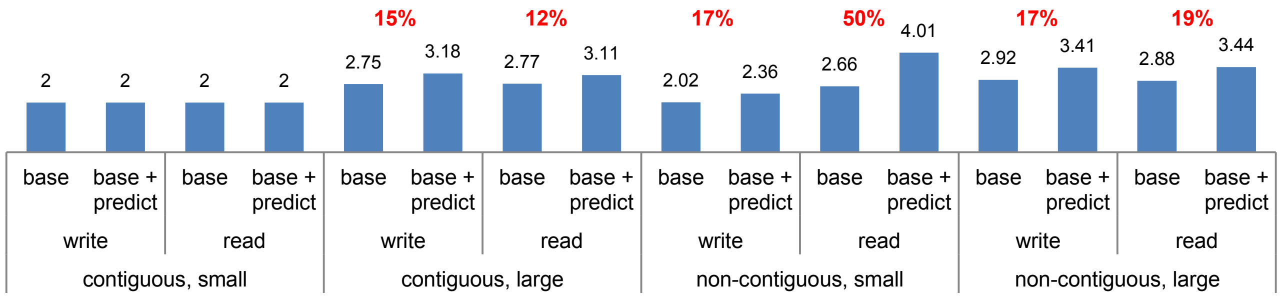


Fig. 7. Increased average aggregation size with the prediction module for AGIOS.

the amount of data accessed by each process is multiple of the stripe size and of the number of servers, the two contiguous requests will always be to the same stripe.

As we can see in Fig. 7, the base scheduling algorithm is already able to aggregate as much as possible to the "contiguous, small requests" workload. Therefore, only the other 3 access patterns are interesting for the prediction module. We included a result with the first one (top left of the figure) in order to show that although the prediction module is not helpful in this case, it does not degrade performance significantly.

Considering the other three workloads, the prediction module was able to improve over the performance previously obtained with our library in up to 35.4% - 29.5% on average - for write operations and in up to 31.4% - 20.7% on average - for read operations.

The improvements for tests with large requests are more expressive than for small requests. This happens because application's large requests result in a larger number of actual requests to the file system generated at once. These requests are contiguous (even if the application's access pattern is not) and tend to arrive at the server almost at the same time, therefore offering more aggregation potential than tests with small requests.

We can also observe that read operations did not benefit from the new approach as much as the write ones. We believe this difference is due to the time the file system takes to process reads being significantly (in our tests, an average of 83.8%) smaller than to process writes. Being faster, read operations are more affected by the algorithm's waiting times.

When comparing Fig. 6 and Fig. 7, we can observe that the biggest increase in aggregation size (for read operations in non-contiguous, small requests) resulted in the lowest increase in performance by the prediction module. In this case, large aggregations were predicted, inducing more waiting times. As previously said, read operations are more affected by these waiting times, so the increase in performance was not as good as in other tests (but still existent).

The **total performance improvement** obtained by our library with the prediction module (when compared to the scenario without a scheduler) is of up to 57.3% - 42.33% on average - for writes and up to 58.9% - 39.2% on average - for reads.

VI. Conclusion

This paper presented AGIOS, an I/O scheduling library for parallel file systems that can be easily plugged into the servers' code. We demonstrated its use with a NFS-based file system, and tested it with 4 different access patterns. Our results with the base scheduling algorithm have shown performance improvements of up to 46.95%, and of 27.44% on average.

We then proposed an approach to improve aggregations by using information about applications' accesses. This information is obtained from traces, generated by the scheduler with no modifications to the file system or to the application, and creates a list of predicted future requests. This look into the workload's future is used by the scheduler in its decision-making. Our approach improved performance in up to 35.4% and in 25.1% on average over the base scheduling algorithm. The total performance improvement of the library with the new approach (comparing with not using the library) was of up to 58.9% and of 40.76% on average. We did not observe any case where the use of AGIOS resulted in performance degradation.

We are already in the process of integrating and evaluating AGIOS with the PVFS2 parallel file system. As future work, we plan to refine our scheduling model in order to use more information about the application, as this approach has been shown to be successful. We also plan to study the use of the library in other levels of the I/O stack.

VII. Acknowledgments

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 paper 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] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective i/o in romio," in *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, ser. FRONTIERS '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 182–. [Online]. Available: http://dl.acm.org/citation.cfm?id=795668.796733

[2] E. Smirni and D. A. Reed, "Workload characterization of input/output intensive parallel applications," in *Proceedings of the 9th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools*. London, UK, UK: Springer-Verlag, 1997, pp. 169–180. [Online]. Available: http://dl.acm.org/citation.cfm?id=647807.737633

[3] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, D. D. E. Long, and T. T. McLarty, "File system workload analysis for large scale scientific computing applications," pp. 139–152, 2004.

[4] F. Z. Boito, R. V. Kassick, and P. O. A. Navaux, "The impact of applications' i/o strategies on the performance of the lustre parallel file system," *International Journal of High Performance Systems Architecture*, vol. 3, no. 2, pp. 122–136, 2011.

[5] A. Lebre, Y. Denneulin, G. Huard, and P. Sowa, "I/o scheduling service for multi-application clusters," in *Proceedings of IEEE Cluster 2006, conference on cluster computing*, sep 2006.

[6] W.-K. Liao and A. Choudhary, "Dynamically adapting file domain partitioning methods for collective i/o based on underlying parallel file system locking protocols," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, 2008, pp. 1–12.

[7] P. M. Dickens and J. Logan, "Y-lib: a user level library to increase the performance of mpi-io in a lustre file system environment," in *Proceedings of the 18th ACM international symposium on High performance distributed computing*, ser. HPDC '09. New York, NY, USA: ACM, 2009, pp. 31–38. [Online]. Available: http://doi.acm.org/10.1145/1551609.1551617

[8] K. Ohta, H. Matsuba, and Y. Ishikawa, "Improving parallel write by node-level request scheduling," in *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid-Volume 00*. IEEE Computer Society, 2009, pp. 196–203.

[9] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, "Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free i/o," in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, 2012, pp. 155–163.

[10] X. Zhang, K. Davis, and S. Jiang, "Iorchestrator: Improving the performance of multi-node i/o systems via inter-server coordination," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.30

[11] ——, "Qos support for end users of i/o-intensive applications using shared storage systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 18–1. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063408

[12] H. Song, Y. Yin, X.-H. Sun, R. Thakur, and S. Lang, "Server-side i/o coordination for parallel file systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 17–1. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063407

[13] Y. Qian, E. Barton, T. Wang, N. Puntambekar, and A. Dilger, "A novel network request scheduler for a large scale storage system," *Computer Science-Research and Development*, vol. 23, no. 3, pp. 143–148, 2009.

[14] T. M. Madhyastha and D. A. Reed, "Learning to classify parallel input/output access patterns," *IEEE Transactions on Parallel and Distributed Systems*, pp. 802–813, 2002.

[15] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp, "Parallel i/o prefetching using mpi file caching and i/o signatures," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 44:1–44:12. [Online]. Available: http://portal.acm.org/citation.cfm?id=1413370.1413415

[16] Y. Chen, S. Byna, X.-H. Sun, R. Thakur, and W. Gropp, "Hiding i/o latency with pre-execution prefetching for parallel applications," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 40:1–40:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413411

[17] G. Soundararajan, M. Mihailescu, and C. Amza, "Context-aware prefetching at the storage server," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, 2008, pp. 377–390.

[18] L. Lin, X. Li, H. Jiang, Y. Zhu, and L. Tian, "Amp: an affinity-based metadata prefetching scheme in large-scale distributed storage systems," in *Cluster Computing and the Grid, 2008. CCGRID'08. 8th IEEE International Symposium on*, 2008, pp. 459–466.

[19] C. M. Patrick, M. Kandemir, M. Karaköy, S. W. Son, and A. Choudhary, "Cashing in on hints for better prefetching and caching in pvfs and mpi-io," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010, pp. 191–202.

[20] Y. Zhang and B. Bhargava, "Self-learning disk scheduling," *IEEE Transactions on Knowledge and Data Engineering*, pp. 50–65, 2008.

[21] S. Seelam, I. H. Chung, J. Bauer, and H. F. Wen, "Masking i/o latency using application level i/o caching and prefetching on blue gene systems," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 2010, pp. 1–12.

[22] R. B. Avila, P. O. A. Navaux, P. Lombard, A. Lebre, and Y. Denneulin, "Performance evaluation of a prototype distributed nfs server," in *16th Symposium on Computer Architecture and High Performance Computing, 2004. SBAC-PAD 2004*, 2004, pp. 100–105.

[23] "Mpi-io test user's guide," 2008. [Online]. Available: http://institutes.lanl.gov/data/software/src/mpi-io/README_21.pdf