I/O Scheduling Service for Multi-Application Clusters *

Adrien Lebre, Guillaume Huard, Yves Denneulin

Laboratoire Informatique et Distribution - IMAG

Montbonnot Saint-Martin, France

firstname.lastname@imag.fr

Przemyslaw Sowa

Institute of Computer and Information Sciences

Czestochowa University of Technology, Poland

sowa@icis.pcz.pl

May 5, 2006

Abstract

Distributed applications, especially the ones being I/O intensive, often access the storage subsystem in a non-sequential way (stride requests). Since such behaviors lower the overall system performance, many applications use parallel I/O libraries such as ROMIO to gather and reorder requests. In the meantime, as cluster usage grows, several applications are often executed concurrently, competing for access to storage subsystems and, thus, potentially canceling optimizations brought by Parallel I/O libraries.

The *aIOLi* project aims at optimizing the I/O accesses within the cluster and providing a simple POSIX API. This article presents an extension of *aIOLi* to address the issue of disjoint

*This work has been done within the ID laboratory jointly supported by CNRS, INPG, INRIA, and UJF and the project LIPS between INRIA and BULL Lab. Computer resources are provided by the grid5000 french experimental grid (`http://www.grid5000.fr/`).

accesses generated by different concurrent applications in a cluster. In such a context, good trade-off has to be assessed between performance, fairness and response time. To achieve this, an I/O scheduling algorithm together with a «*requests aggregator*» that take into account both application access patterns and global system load, have been designed and merged into *aIOLi*. This improvement led to the implementation of a new generic framework pluggable into any I/O file system layer. A test composed of two concurrent IOR benchmarks showed improvements on read accesses by a factor ranging from 3.5 to 35 with POSIX calls and from 3.3 to 5 with ROMIO, both reference benchmarks executed on a traditional NFS server without any additional optimizations.

1 Introduction

I/O bottlenecks have always been a major issue in Computer Science and it is likely to continue as performances increase slower for I/O hardware than for CPU and memory. This gap is widened by the increasing use of HPC platforms and the growing number of parallel I/O intensive scientific application. In this context, the I/O subsystem is stressed both by the overall throughput requirement and the peculiar access patterns of parallel applications. For instance, disjoint requests delivered at the same time may generate disk head movements, one of the most time consuming operations in modern computers (approximately 9ms). But, as shown by several studies [6, 17], parallel I/O accesses use recurrent determined patterns (based on stride parameters) that are good candidates for optimizations. This is different from “database accesses” which depend on the selection criteria and are usually more sparse.

In this article, we focus on multiple concurrent applications that perform parallel I/O accesses to the storage subsystem. This is a common situation on clusters since most batch schedulers [9], try to maximize the overall platform usage, leading to concurrent executions. In this case, each application generates its own recurrent parallel access patterns and parallel access patterns from distinct applications are interleaved due to concurrency. Thus, the I/O subsystem layer has to perform optimizations that take advantage of regularity of accesses from each application while balancing storage access between them. Our work focuses on non dedicated clusters where all applications are considered of equal importance. This translates into multiple optimization criterion:

throughput, fairness and response time.

I/O parallel accesses optimization issues have been addressed by several researchers. Proposed solutions can be classified in two main categories:

- *Parallel File systems* [4, 21, 22] manage to exploit as efficiently as possible hardware capabilities while taking into account distributed file system constraints (coherency, fault tolerance, remote accesses, ...). Most of these systems do not use I/O scheduling strategies as they are just built on schedulers located at block device layer (section 2.3). At this low level, due to kernel and file system implementation (see section 3.1) parallel applications information is not available and parallel I/O access patterns cannot be exploited for throughput optimization.

- *Parallel I/O Libraries* are focused on parallel I/O aspects and portability constraints inside a single application. They use a specific API to enable the developer to express I/O access patterns. The underlying optimization algorithm exploit these patterns to aggregate individual requests into larger ones (see section 2.1). Unfortunately, beside the complexity of the API of these systems, storage access balance between applications is not addressed by I/O libraries. Moreover, individual application optimizations are likely to be canceled by the interleaving of concurrent accesses made by distinct applications (see multi-applicative MPI/IO tests in section 4).

When dealing with concurrent parallel applications, it is important to handle requests in a global manner to provide a good trade-off between performance and balance between applications. Performance alone could be optimized by handling all the I/O requests from one application before serving another one. This provides good throughput but starves the waiting applications. Such a policy does not take into account response time and fairness criteria which are mandatory in a multiple applications environment.

We propose to design a high level infrastructure made of two main components: a transparent parallel I/O aggregation mechanism for throughput optimization and a scheduling algorithm that balance service provision between applications. Our main contribution is to integrate these two elements into a generic framework pluggable into any existing I/O system. This framework do not require any change in the application code (it only interact with the I/O subsystem). It only

requires a minimal change in the I/O layer code to subscribe to the *aIOLi* services (section 3.2). To take advantage of a global view of all the accesses without any negative impact on scalability, *aIOLi* should be plugged into any already existing I/O centralization point.

We implemented our proposal as a Linux module and evaluated it on a Network File System server. Even if NFS is not really suited to high performance I/O, it remains the standard configuration for small and medium sized clusters. The server has to deal with huge amounts of simultaneous requests, a good testbed to evaluate the interest of a high level scheduler like ours.

The rest of this paper is organized as follows: section 2 briefly presents the available I/O optimizations and their limits. Section 3 is focused on the architecture of our framework: the interest of a higher level scheduler, the architecture of the current version of *aIOLi* and the two integrated mechanisms are introduced. Section 4 gives some experimental results. Eventually, section 5 describes future extensions as well as possible improvements and section 6 concludes the paper.

2 Background

In this section we present main ideas used in parallel file-systems, in parallel I/O libraries or both. Most of them aim at optimizing I/O throughput for one parallel application (collective approaches, prefetch) while other do not take advantage of accesses regularity (scheduling). The goal of this section is to give an overview of I/O optimizations in general and to explain why they are not suited to the multi-applicative context.

2.1 Collective Approaches

Different processes of a parallel application usually send many small, non-contiguous requests simultaneously to the I/O server without checking for aggregation opportunities. Collective I/O methods solve this problem by merging different requests into a bigger one and issuing an aggregated request. This concept can be applied at different layers in a distributed architecture: at disk level [12], at server side [23] or also at client side [25].

The resulting performance of collective approaches depend on the underlying architecture (middleware, network interconnection and file system implementation) and the use of specific

"tuning" routines (*MPI I/O hints* functions) is often required. Furthermore, as pointed in [1], such approaches imply expensive synchronization mechanisms. In the case of writes, Ma et al. proposed to use active buffering with threads [15] to overlap I/O with computation efficiently. This lessen the impact of synchronization but require a modified ROMIO library.

Overall the collective approaches do not take into account the interleaving of non contiguous access resulting from concurrent execution of several applications. As shown in section 4 this lead to severe performance degradation.

2.2 Cache and Prefetching

Caching is a widespread technique used to reduce the number of accesses to hard drives and thus improve performance of the I/O system (see [3] for instance). In centralized/local systems, cache management is not very difficult, it significantly differs in distributed and parallel environments where strict coherency protocols are complex and impact performance. Some systems avoid this problem by sacrificing the client side file caching and keeping caching only on I/O nodes. Collective caching [27] is another method to improve I/O performance of parallel applications. It is based on the idea that all processes running the same application should be considered as a single client of a parallel file system. A modified MPI version provides user-level file caching by distributing the cached data equally between processes. Unfortunately, this mechanism requires a first step, where data are redistributed to different nodes to build the cache system. This step can be expensive for large workloads.

Usually, as mentioned in [26], several levels of cache could lead to inefficiency if, for each cache request, a cache miss occurs. Overall, caching can be combined with *aIOLi* but it do not solve the same problem: multi-applications balancing and initial fetch of data are not addressed.

Prefetching techniques are based on implicit or explicit anticipation of I/O requests. Some parallel intensive I/O programs reduce I/O congestion by retrieving explicitly data in a sequential way from one client (in a synchronous or asynchronous mode) and redistribute them to all participants. Such an approach leads to good performance in a non-concurrent environment thanks to the gain provided by the *Read Ahead* technique[7]. Daniel Ellard and Margo Seltzer [8] modified the FreeBSD 4.6 NFS server to improve the read-ahead heuristic strategy. Their new algorithm handle

stride access patterns in a better way.

Unfortunately, in multi applications environments, these prefetching methods generate parallel I/O problems. As in the case of collective approaches, they do not take into account the interleaving of non contiguous access resulting from concurrent execution of several applications.

2.3 I/O Sheduling Strategies

Several scheduling algorithms were proposed to minimize the completion time of a batch of I/O operations. These algorithms usually fall in two categories: disk scheduling [24] and parallel I/O scheduling [5, 11]. The first one tends to limit disk head movements while the other one distributes parallel I/O operations to different I/O servers to minimize the overall response time.

Disk scheduling algorithms, because they are all implemented at a low level, cannot have a good overview of distributed applications accessing the file system. Besides being limited by the size for their queues, they are strictly dependent on the implementation of the upper file system: for instance, if the file system is synchronous and mono-threaded, only one request can be handled at the same time which limits potential optimizations (section 3.1 for more details). As a direct consequence, such approaches are not suited to large workloads generated by HPC Intensive I/O applications.

Parallel I/O scheduling, in contrast to low level schedulers, try to exploit parallel I/O access patterns. In PVFS [20] a model for predicting performance of a system for a given workload is used. Based on this model, the system can choose dynamically the most appropriate scheduling algorithm. In the Clusterfile parallel file system [10], a scheduling heuristic tries to involve all the I/O servers in the system at the same time to maximize their utilization. However, to the best of our knowledge, these systems do not address the interleaving of multiple applications. Hence, they suffer from concurrent disjoint access resulting from the simultaneous execution of several applications.

3 System Overview

The first implementation of the *aIOLi* prototype [13] provided an efficient transparent management of parallel I/O for one application within a SMP node. All I/O requests, before being sent to a

remote server, were analyzed to find aggregation possibilities and reordered to favor sequentiality. The encouraging results of this first implementation have motivated the study of similar approaches but at cluster level [14]. In this intermediate work, we chose a client/server model comparable to the PANDA architecture [23]. Unfortunately, the lack of global memory and global clock made the management difficult and the expected improvements were not reached.

In this new study, we opted to move *aIOLi* from the applications to the storage system layer in order to collect more informations both about applications and global system load. Indeed, one of the main idea of this new proposition is to exploit the existing centralization point of file system architectures to plug our new framework. Therefore, *aIOLi* do not compromise the I/O system scalability by adding new bottlenecks.

3.1 Preliminary study

As mentioned in former sections, low-level scheduler do not have a global view of the I/O accesses made by a parallel application. This is due to limitations in the queues size and in the I/O server implemented on top of them.

We checked aggregation capabilities of these schedulers using the IOR benchmark presented in section 4.3. The experiments consists in evaluating IOR over 32 MPI instances decomposing a 4GB file on a Linux NFS server exporting an ext3 file-system stored on a single IDE disk (57MB/s peak). This is the same hardware configuration as in section 4.

Test have been performed on each Linux I/O low level scheduler, making the file access granularity vary from 8KB to 4096KB and by using POSIX API. To provide more aggregation opportunities to the scheduler, the number of active `nfsd` daemons ranged from 8 (default configuration) to 512 (more incoming simultaneous requests). The results are similar whatever the I/O scheduler in use, we only present the default one: anticipatory scheduler on the right of figure 1 (the results for all the other schedulers can be found on the *aIOLi* website `http://aioli.imag.fr`).

As we can notice, the greater the number of daemons, the better the performances are. Nevertheless, even with 512 daemons, the low-level scheduler performance is far from sequential performance (around 50MB/sec) in most cases.

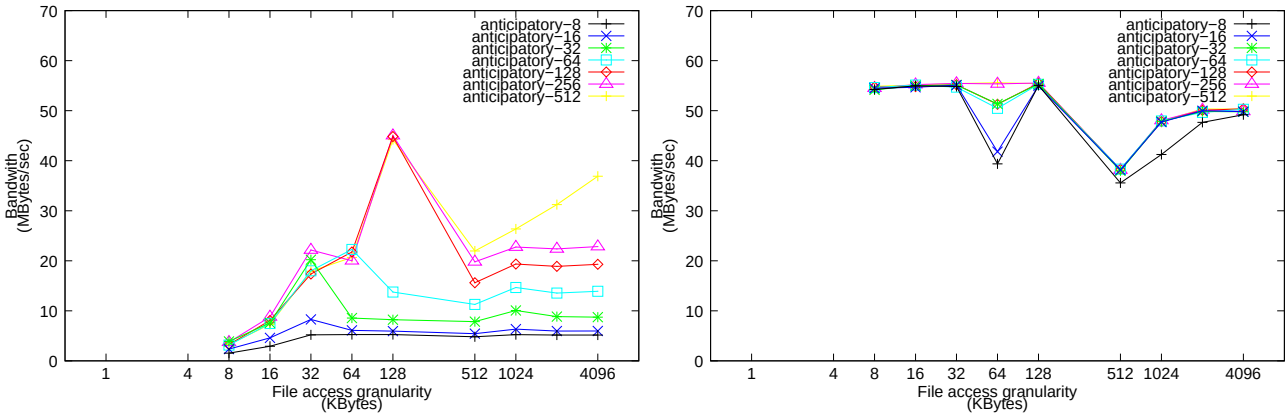


Figure 1: low scheduler impacts on a traditional NFS server and a NFS server plugged with aIOLi

4 GBytes file decomposition (IOR benchmark) on 32 MPI instances deployed on 32 nodes.

Usual NFS using 8 to 512 daemons on top of the anticipatory scheduler (left) and plugged to *aIOLi* (right).

3.2 A High Level I/O Scheduling Framework

The purpose of the *aIOLi* framework is to provide generalized I/O scheduling strategies independent of any storage medium or I/O subsystem. The latest version of *aIOLi* can extend almost any existing I/O management system using a simple pluggin mechanism.

This version uses a new architecture (figure 2) in which an *aIOLi client* can be the whole kernel I/O subsystem, a single remote file system or any other I/O intensive service. Each client is connected to an *aIOLi* I/O controller which implements the interface to the *aIOLi* framework. All the *aIOLi* client should be able to react to the two following events:

- a new request is delivered to the client. In this case, the client should posts this new request to the queue of the related *aIOLi* I/O controller.
- the *aIOLi* I/O controller notifies the client that one or more of its requests can be processed. The client then should process it.

We choose to let the clients process themselves the requests they posted. This way, *aIOLi* remains generic and independent, it just acts as an aggregator and scheduling service.

An *aIOLi* I/O controller can be in charge of scheduling requests from one or several clients at the same time. For instance, on a remote NFS server which exports an Ext3 partition, an *aIOLi* I/O

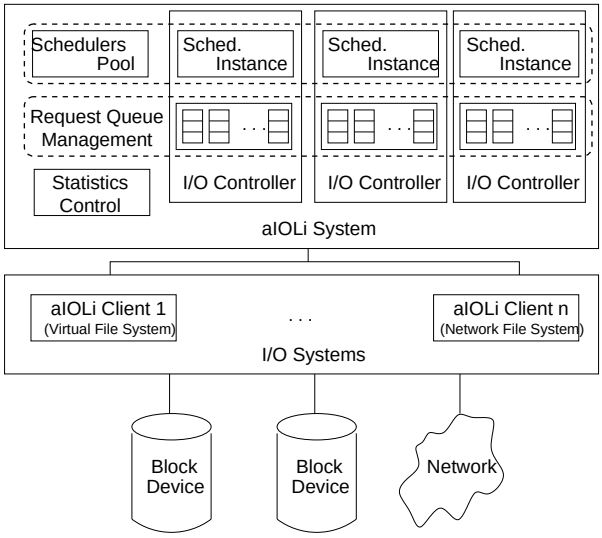


Figure 2: Architecture of the aIOLi system

Clients (I/O systems) put their incoming requests in the related *aIOLi* queues and are notified when any of them should be processed. To prevent coherency issues, clients remain in charge of this processing.

controller can be used to schedule both requests incoming to the NFS server and local requests to the file-system. In this case, the NFS server and the Ext3 file-system will be two clients associated to the same I/O controller. This allows *aIOLi* to be plugged where needed (on an independant file-system or on the I/O layer of the whole system) to perform fine optimizations at the appropriate level.

An *aIOLi* scheduler has to be chosen when initializing an I/O controller. These schedulers exploit file id, request size and start offset to chose the appropriate issue order. Thus, they deeply differ from low level schedulers mainly based on disk sector placement (section 2.3). Currently, *aIOLi* ships with two different schedulers [1]: a simple FIFO and a MLF variant discussed in section 3.4. Additional scheduling algorithms are easy to add either directly to *aIOLi* or as an external kernel module.

Regarding coherency, to prevent any issue, an unique timestamp is associated to each incoming request. Using these timestamps, the strict order between write and read accesses for the same resource can be enforced. Currently, all requests following a write to the same file are blocked by the system until the completion of this write. This method could be easily improved by using a

[1] A third one focusing on the interactivity criterion is under development: *Weighted SJF* [14].

finer locking mechanism such as the well-known *Byte-Range-Locking*.

aIOLi, which was formerly a library, is now a pluggable I/O scheduler. This change makes extensible and flexible: it is completely independant from applications and can be plugged into any existing I/O management system. The *aIOLi* is now a high-level I/O scheduler : it only requires generic informations for the requests it handles (offset and size). Our framework also includes a statistics collector which gather informations about I/O workers (number of requests proceeded, number of aggregation, average, etc.) either for *post mortem* or on-line analysis.

3.3 Aggregation and Virtual Aggregation

The first prototype of *aIOLi* [13], was a library dedicated to the optimization of I/O in a single node. Since we were within a single node (only one operating system), coherency mechanisms were provided by the underlying file-system stack (single buffer cache) and we implemented a physical aggregation mechanism: small contiguous requests were merged into a larger one which was sent by the kernel to the remote file system.

In a distributed environment, the implementation of such mechanisms becomes tedious (because of data replication, cache invalidation, etc.). Moreover, few file-system provide routines to access to a group of disjoint file parts (I/O vector). To remain generic *aIOLi* is able to perform its optimization using only simple access requests. The“virtual aggregation” mechanism consists in deciding on an execution order for several requests and let the actual execution of I/O calls to clients. For instance, the three following requests: *read(30,40)*, *read(20,30)*, *read(10,20)*[2], should be reordered and executed in the following order: *read(10,20)*, *read(20,30)*, *read(30,40)*. This way, all the accesses become contiguous.

When I/O systems provide specific routines to handle I/O vectors, *aIOLi* will use them and thus apply a “real aggregation”. But, our experiments show that the “virtual aggregation” mechanism is sufficient to reach near-optimal performance: the genericity does not imply a performance degradation.

[2] `read(x,y)`: read from offset x to offset y in the same file

3.4 Scheduling of I/O Requests on a Cluster

aIOLi provides scheduling strategies to efficiently share the I/O subsystem among all applications running on the cluster. As mentioned in section 1, in our case, we aim at providing scheduling strategies optimizing throughput first, but with a concern for fairness and response time.

3.4.1 Base Scheduling Algorithm

We want both to maximize the overall performance by using parallel I/O aggregation techniques and maintain a good balance among applications. Unfortunately, throughput, fairness and response time are incompatible: to be fair and responsive I/O resource access has to be switched regularly between applications thereby breaking the contiguity of accesses required to reach the maximal throughput. Thus, our scheduling strategies will consist in the best compromise between performing maximal aggregation and serving each application in turn. This I/O requests scheduling in our context is an on-line problem: jobs (requests) keep on arriving during the scheduling process and the total size of the access (total number of requests) is not known in advance [2].

The base of our algorithm is a variant of the Multilevel Feedback algorithm (MLF) [18]. The MLF algorithm is designed to optimize average response time while avoiding starvation by granting to individual waiting requests an adaptive time quantum for accessing resources (a quantum that grows with time). We modified MLF to integrate into its mechanics the virtual aggregation mechanisms. The resulting algorithm can be described as follows:

1. incoming requests are sorted by type (read or write) and inserted into two separate queues for each file accessed.

2. each request is assigned an initial quantum of zero upon its arrival in the system.

3. aggregation is performed on requests of both queues (read and write): the queues are traversed in offset order and contiguous requests are aggregated into larger virtual requests which quantum is the sum of individual requests quantum.

4. the quantum of each request is then increased by a fixed value QB (which is rather small to favor interactivity).

5. the first request, in offset order within a file and FIFO order between files, which quantum is large enough to enable its completion is selected for execution.

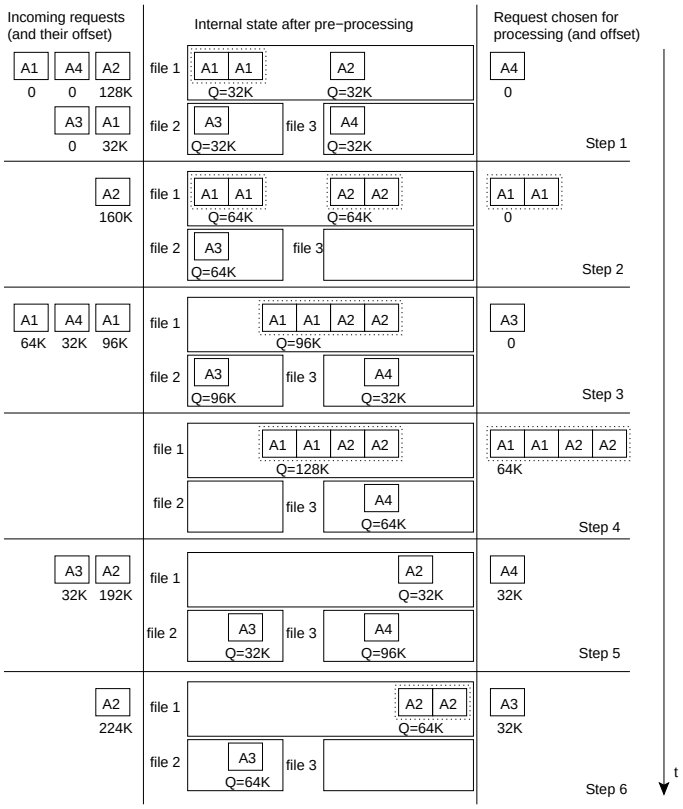


Figure 3: A variant of the Multiple Level Feedback algorithm

The figure 3 presents the behavior of our algorithm on a toy example made of two processes (A1 and A2) accessing a file (file 1) in a stride like manner (with strides of 128K long) and two processes (A3 and A4) accessing two files (file 2 and file 3) in a synchronous way (one request after the other). In this example all the accesses are 32K long and QB equals 32K. As with MLF, the use of small quantum that increase with time gives priority to small requests (not aggregated). Thus, big virtual requests will remain for some time in the system giving it opportunities for more aggregation (in other words, throughput optimization). Processing request in their size order is the optimal strategy to minimize the average response time making our algorithm good for interactive tasks. Nevertheless, because we bound aggregation and the quantum increases regularly, waiting aggregated requests will never starve and a decent fairness will be maintained. Notice that our algorithm is a compromise. Indeed, we could improve throughput by removing the quantum system

and stick to the offset order, but this would degrade response time and fairness as requests would possibly be delayed for a long time by other ones.

3.4.2 Performance Tuning

Although the preceding algorithm has a good overall behavior, optimizing for several antagonist criteria at once, it is not able to completely detect and exploit some widespread specific accesses behavior of common applications:

synchronous accesses applications such as `cat` make their access in a synchronous fashion, one byte at a time, waiting for the result before performing the next access. In that case, it should be advantageous for the server to wait for the next access at the end of the request processing. This could be done by giving the request a quantum larger than required for its completion: the extra time can then be used to wait and aggregate consecutive requests as they arrive.

very large accesses most applications generally make either very large or very small accesses to files. In the case of large accesses, the linear increase rate of the quantum in the base algorithm does not exploit sufficiently aggregation opportunities. Thus, it should be advantageous to give a larger quantum to consecutive requests to the same file, as long as this quantum is fully exploited.

very small accesses In the case of small accesses, the quantum given to requests might be too large (especially for requests of size lower than the base quantum size or requests that have stayed too long in the system). The issue is that if we use this extra quantum duration to wait for aggregation opportunities, this only ends up in useless delay.

To address these issues, we take advantage of file accesses history and we adapt dynamically the quantum size to applications specific behavior. For each file accessed, we store the utilization rate of the quantum given to the last access. If this rate is high, we are likely to perform a large or synchronous access. In that case we expand the size of the quantum given to requests to this file by a multiplicative factor. On the contrary, if this utilization rate is low, we are likely to perform small accesses and we reduce the quantum size accordingly. As a special case, when the utilization rate is very low the end of the file should have been reached. In that case, we simply reset the history information.

4 Experiments

Our testing system is a part of the machines from the "grid5000"[3] project located at the INRIA Sophia-Antipolis site. Each node, an IBM eServer 325, is composed of two AMD Opteron (2GHz) CPUs, 2GB RAM and a 80GB IDE hard-drive (bandwidth estimated to 57MB/s by the `hdparm` command). The cluster is interconnected by a gigabit ethernet network. All nodes were running a Debian GNU/Linux system with a 2.6.15 kernel. A dedicated NFS server (version 3, TCP, 32Kb read/write size, sync, cache disabled) on top of an ext3 file-system and several client SMP nodes have been used.

In a first part, we evaluated the overhead of aIOLi and its impact on scalability for both non HPC and HPC workloads. Then, we focused our experiments on the multi-application criteria. The IOR benchmark has been used to evaluate real I/O intensive HPC applications.

4.1 Implemented *aIOLi* clients

The *aIOLi* 's public interface is composed by three main functions (two optional complete the API). Any client *aIOLi* client has to call the initialization function with at least two callback functions as arguments: the `read` and `write` functions from the host I/O system. These functions will be called when *aIOLi* decide on the execution of one ore more requests. Additional optional callbacks can be given to *aIOLi* to handle "real aggregation".

Unfortunately, the client code has to be slightly modified (that is the I/O system layer): to redirect incoming request to *aIOLi*, it is necessary to add a `post` call. This is the only intrusive step when plugging *aIOLi* to a client. Up to now, we did not discover a completely transparent approach.

So far, two *aIOLi* clients have been developed. The first one is a NFS (network file system) server based on the source code from the 2.6.15 Linux kernel. The second one is an extension of the Linux Virtual File System which handle all I/O operations on a single node. Both of them are shipped with the *aIOLi* source code distribution[4]. Their evaluation led us to the same conclusion. Therefore, due to space considerations, only NFS experiments will be discussed in this article.

[3] `http://www.grid5000.fr/`.

[4] `http://aioli.imag.fr`

4.2 Overhead and Scalability Impact

4.2.1 Bonnie++

Bonnie++[5] is a popular and widely used benchmark to evaluate hard drive and file-system performances. It tests writes, reads and creation of files. The write test consists of three phases : a file is first written byte by byte, then it is overwritten in a block manner and finally it is read and overwritten block by block. The two-phase read test is similar to the first two phases of write test. Although Bonnie++ is often used as a benchmark for clusters, it is not really suited to this task because: no HPC application access data using a fine granularity (byte by byte) and stride accesses are not tested.

Nevertheless, we decided to benchmark *aIOLi* with Bonnie++ anyway to prove that our system does not have a negative impact on fine grained sequential performance. Our test consists of two parts: a first test with one client, then a second test with four clients. Both parts use one NFS server. We configured Bonnie++ to use 4 GB files (twice the RAM size) and to skip the file creation test. Results are presented in the table 1.

	Write			Read	
	char MB/s	block MB/s	rewrite MB/s	char MB/s	block MB/s
NFS (1 client)	21.69	28.40	2.00	34.84	43.54
aIOLi (1 client)	19.80	28.29	2.03	37.45	48.81
NFS (4 clients)	8.26	8.78	1.68	3.55	3.02
aIOLi (4 clients)	7.41	9.59	1.68	4.74	13.39

Table 1: Bonnie ++ evaluation

As we can notice, *aIOLi* does not have a significant impact on I/O accesses at a fine granularity. In this case, requests are mostly satisfied by the NFS cache because the server make access of 32KB anyway (our `rsize` parameter). Thus, *aIOLi* has no room for further optimizations. It does not benefit to sequential writes either as they are already handled asynchronously by the system. But, the performance is greatly improved when four concurrent clients make simultaneous read at a coarse granularity. In this case the adaptive quantum mechanism used in *aIOLi* (section 3.4.2) shows its strenght.

[5]Bonnie benchmark suite, `http://www.coker.com.au/bonnie++`

4.2.2 b_eff_IO benchmark

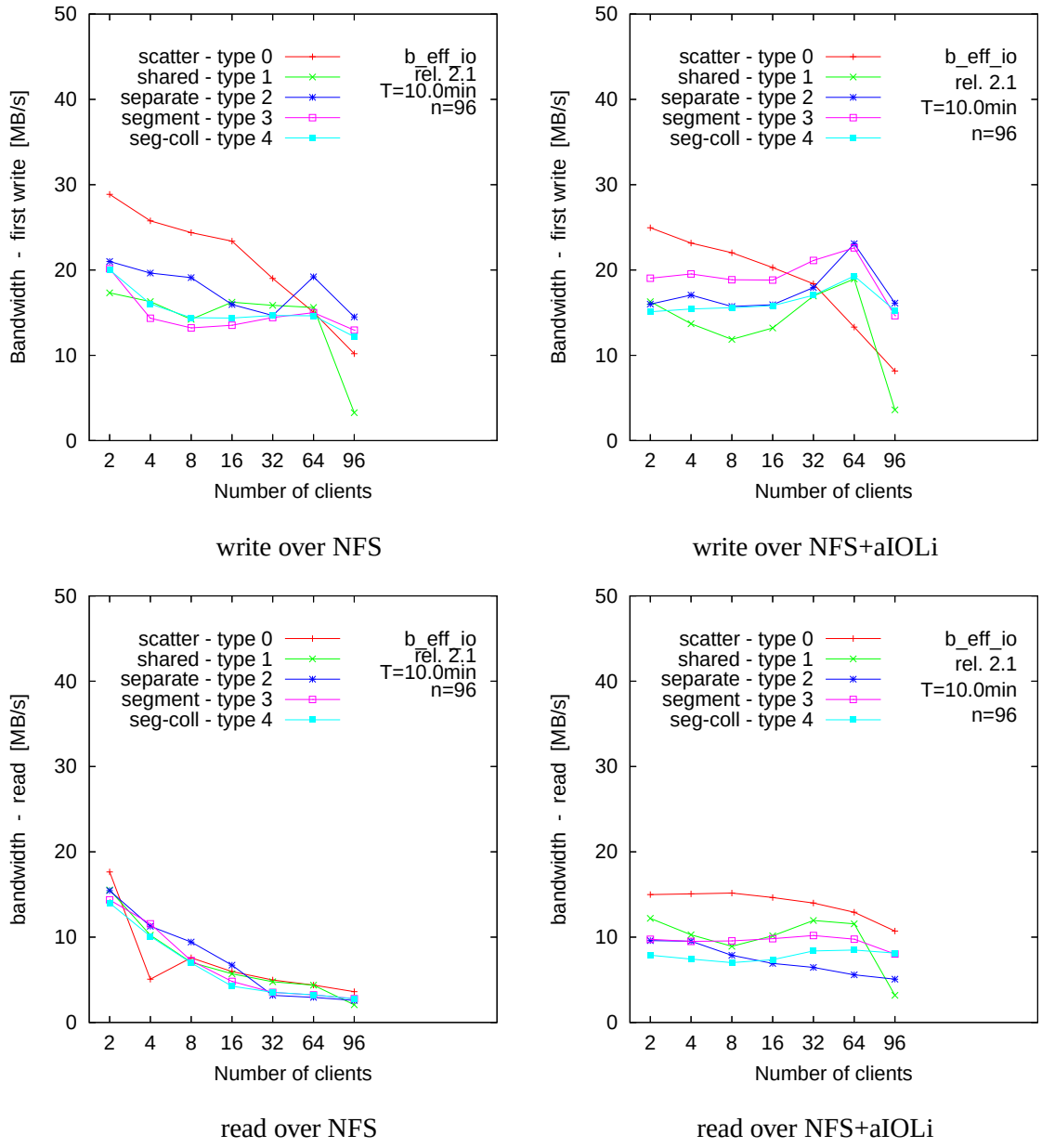


Figure 4: scalability impact: usual NFS server (left) and NFS server plugged with aIOLi (right)

To test the impact on scalability of our system, we have used the Effective I/O Bandwidth Benchmark[19] (b_eff_io), which provide two tests: the first evaluates the average I/O bandwidth achievable when using the MPI-I/O library and the second gather detailed informations depending on the access patterns and buffers length.

The benchmark tests “first write”, “rewrite” and “read” routines using several combinations for its parameters such as: various parallel access patterns (different kinds of stride like patterns), collective / non collective accesses, aligned / unaligned accesses. In this test, similar to a single parallel application, the I/O operations are already optimized by the MPI-I/O library and *aIOLi* is not likely to produce further improvements. The goal is only to see the impact of *aIOLi* on a heavily loaded system. The number of clients varied from 2 to 96 (a high load for usual NFS servers).

The results are presented in figure 4. Regarding the write performance, because the system handle write requests asynchronously, *aIOLi* has few room for improvements. As a consequence, the optimizations it provides are compensated by the overhead of the scheduler. The resulting performance is roughly the same and scalability is not degraded.

Regarding read performance, thanks to its scheduling strategy, *aIOLi* performs clearly better than the NFS server alone. The slight performance degradation when the number of clients increases is due to the CPU cost of processing a RPC request as shown in [16]. We do not present results for the rewrite test which are similar to the read performance results. Overall, this test shows that *aIOLi* does not have any negative impact on I/O system scalability and can even bring some improvements when aggregation opportunities still exist.

4.3 Multi-application criteria

In this part, experimentations focus on parallel application and multi-application aspects. The I/O Stress Benchmark Codes[6] has been exploited to emulate the behavior of a parallel I/O intensive application. It consists in a parallel file system code developed by the Scalable I/O project at Lawrence Livermore National Laboratory. This parallel program performs parallel writes and reads to/from a file using the POSIX or MPI-IO API and reports the throughput rates.

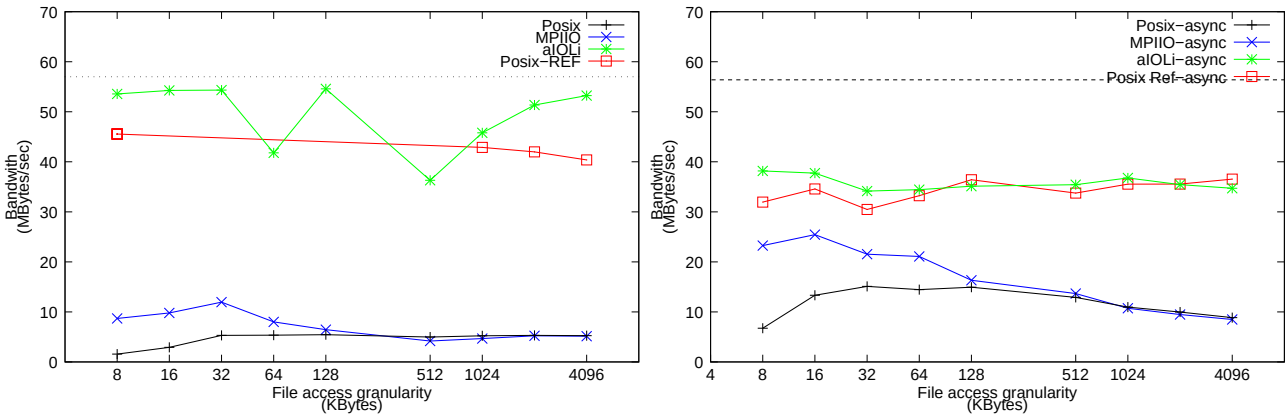
In a preliminary part, we check that parallel access are efficiently managed by our framework (aggregation for one application). Indeed, this is necessary as our main objective is to efficiently manage I/O in presence of multiple and concurrent parallel applications. Then, we evaluate the impact of one application on another one: 1./ we evaluate the degradation implied by one parallel

[6]IOR benchmark, `http://www.llnl.gov/asci/purple/benchmarks/limited/ior/`

I/O intensive application on a less I/O dependant program 2./ we analyse the behavior of two concurrent I/O intensive applications. Finally, the performances of a traditional NFS Server and a server exploiting *aIOLi* are compared using a workload composed by ten distinct applications.

4.3.1 Multi-node coordination

In this experiment, one IOR instance has been deployed on 32 single processor nodes. The file size has been set to 4GBytes and the file access granularity ranges from 8KBytes to 4MBytes.



Read decomposition

Write decomposition

Figure 5: 1 IOR - Validation of aggregation mechanisms inside aIOLi

4 GBytes file decomposition (IOR benchmark) on 32 MPI instances deployed on 32 nodes.

The figure 5 presents the performance provided by the usual NFS server accessed with POSIX (“Posix” curve), the usual NFS server accessed with collective MPI I/O API (“MPI-I/O” curve) and the POSIX API on top of an NFS server plugged to the *aIOLi* framework (“aIOLi” curve).

On this test, MPI-I/O has been ran without file view but the results with file view are similar. The “Posix-Ref” curve has been obtained by prefetching the data from one node. From the point of view of the NFS server this match a large sequential synchronous access (like a `cat`). The dashed line is the hard drive sustained bandwidth (obtained with the `hdparm` command), it is an upper bound for the performance of our NFS server. NFS Version 3 provides two modes for write access: synchronous and asynchronous, both were evaluated. Nevertheless, we only present results for the asynchronous write mode as the performance in synchronous mode is poor for all the compared systems and is not suited to HPC computing.

Regarding the read performance, thanks to its adaptive algorithm, *aIOLi* provides clearly better performance than Posix and MPI-I/O. It even surpasses Posix-ref when a sufficient number of aggregation opportunities exists. The only situations when Posix-ref performs better is when the granularity is between 64KB and 512KB : greater than the NFS access granularity (32KB in our case). In this case, each client access is resolved by very few NFS requests and because of concurrency these requests are disjoints. Because of the low number of requests, the scheduling window on which *aIOLi* works is too small and optimizations are limited. The problem does not appear neither with fine granularities (thanks to *aIOLi* offset order policy) nor with coarse granularities (because the scheduling windows is sufficiently large to find aggregation opportunities).

Regarding the write performance, once again *aIOLi* performs better than Posix and MPI-IO. In this case its performance is very close to the performance of Posix-ref. This is because Posix-ref write in asynchronous mode and benefit from write-behind policy.

Overall, *aIOLi* is still better than any other system in most cases. In general, it is comparable to Posix-ref and clearly better than both Posix and MPI-I/O.

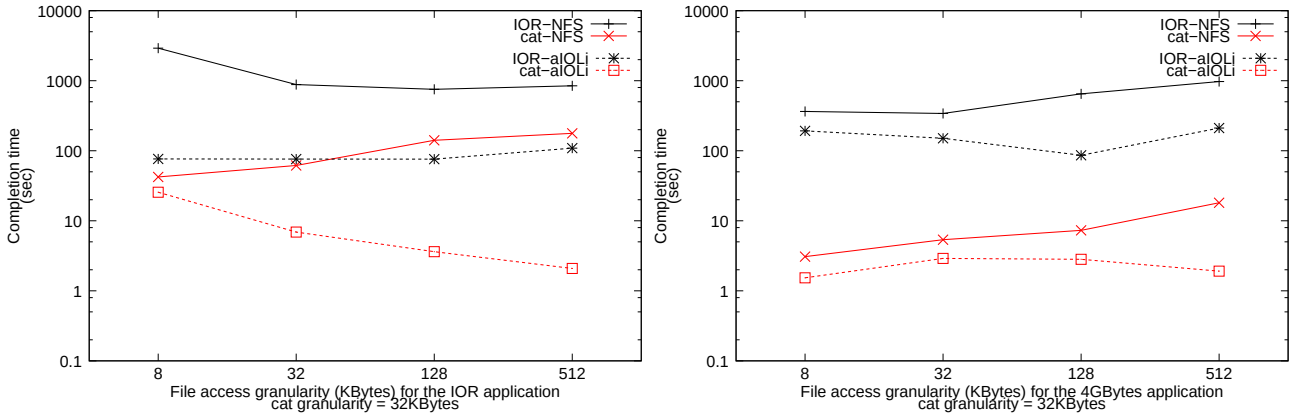
4.3.2 Several applications

In this part, experiments focus on the mutual hindering generated by several applications executing concurrently. These experiments are composed of two parts: firstly a parallel I/O intensive application and a non I/O intensive program started during its execution and secondly, two similar I/O intensive applications striving for resources access. This test aim at demonstrating the balancing capabilities of *aIOLi*: no application is sacrificed to another one and all benefit from *aIOLi* optimizations. Time completion is measured as it is more relevant when applications are different.

Impact of an I/O intensive application on a less intensive one

The parallel I/O intensive applications consist in one IOR instance deployed on 32 single processor nodes. The file size has been set to 4GBytes and the file access granularity ranges from 8KBytes to 4MBytes. For this application, there are two cases: in case one, Posix calls are used for accesses and in the second case, MPI-I/O is used for accesses.

The non I/O intensive program is a `cat`-like program retrieving sequentially a 16MB file on a single node using granularities ranging from 8KB to 512KB. This latter application is small (run



Read - POSIX for IOR

Read - MPI I/O for IOR

Figure 6: Impact of an I/O intensive application on a lesser one

alone, it completes in roughly 2s) and is started during the execution of the first one (15s after its beginning). For this application, Posix is always used for accesses. As all the results lead to the same conclusion, we only present tests using a cat granularity of 32KB[7]. Due to the large difference between the completion times, a log scale for the Y axis has been used. The figure 6 presents the results of this test.

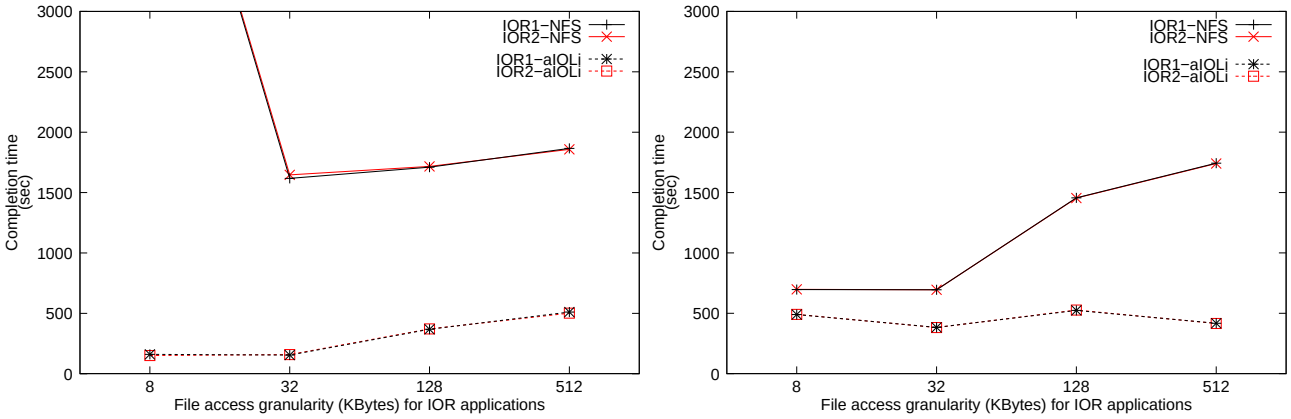
On the left, the parallel I/O intensive application use the POSIX routines. As we can see, *aIOLi* improves the efficiency for both applications. For smaller granularities and due to the adaptive window, the IOR application benefits more than the `cat` program. Indeed, as we mentioned in the former section, *aIOLi* optimizes sequential accesses when large granularities are exploited.

On the right, the parallel I/O intensive application has been launched with the MPI I/O optimisations (the `cat` program was still based on the POSIX API). First, we can note that *aIOLi* under POSIX (curve IOR-aIOLi on the left) provides better performance than the MPI I/O under usual NFS server (curve IOR-NFS on the right) for the IOR benchmark. Moreover, as we expected, the MPI I/O routines reduce congestion issues on the NFS server side which lead to better performance for the `cat` program (curve cat-NFS on the left vs cat-NFS on the right). However, when we compare the IOR-NFS and IOR-aIOLi on the right, we can see that *aIOLi* does not benefit from the MPI I/O optimization. Indeed the synchronisation overhead implied by MPI I/O is added to *aIOLi* performance.

[7] other results can be found on the *aIOLi* website: http://aioli.imag.fr.

Impact of two I/O intensive applications

In this second part, we analyse the behavior of two concurrent IOR instances. Each instance is made of 32 process deployed on a distinct groups of 16 single processor nodes.



Read - POSIX Read - MPI IO

Figure 7: Impact of an I/O intensive application on a second one

As expected in figure 7, the execution of two I/O intensive applications degrades performances for all instances, but *aIOLi* minimizes this phenomenon while balancing the I/O accesses between them. Even, if the collective MPI I/O performances show a slight improvement for small granularities, they quickly reach the POSIX ones for granularities greater than 128Kb. Moreover, even if the MPI I/O collective approach is improved by *aIOLi* (aIOLi curves on the right), the best result is provided by the standard API POSIX under *aIOLi* (aIOLi curves on the left). *aIOLi* takes advantage of all the freedom given by Posix without synchronization overhead, this is why it performs better than MPI-IO. For the smallest granularity, Posix requires more than 1 hour and half, MPI I/O needs 11 minutes and 30 seconds whereas *aIOLi* only takes 2 minutes and 35 seconds.

Regarding the fairness, the 2 IOR benchmarks have been launched at the same time: at worst, the gap between the two completion time is 8.5 seconds for 8Kb but with a 35 times improvement for POSIX and near 5 for MPI I/O. By choosing more specific quantum value for each accessed file, it is possible to set up a desired quality of service for each parallel application running on the cluster. In our case, the quantum variations are similar for the two IOR benchmark.

High concurrency

In this last part, the completion time of 10 distinct applications launched under a NFS server and a NFS server plugged to aIOLi are discussed. 96 nodes were dedicated for this experiment. The table 2 summarizes all values. The description of each application is given : 4 applications worked on the file in a parallel way and 6 others in sequential. The whole size of data represents 6GBytes.

Application description	Completion time			
	NFS		NFS+aIOLi	
	POSIX	MPI IO	POSIX	MPI IO
Read decomposition: 2GB over 32 nodes (granularity=128K)	490	840	134	500
Write Decomposition: 2GB over 32 nodes (granularity=128KB)	409	815	107	604
Read decomposition: 256MB over 16 nodes (granularity=8KB)	595.5	728	104	415
Write decomposition: 128MB over 8nodes (granularity=64KB)	51	257	14.5	247
Sequential read: 32MB on 1 node (granularity=4KB)	531	9	48.5	3
Sequential write: 32MB from 1 node (granularity=4KB)	208	9	47	6
Sequential read: 4MB on 1 node (granularity=32KB)	57	1.5	6	1
Sequential write: 4MB from 1 node (granularity=32KB)	39	2	19	2
Sequential read: 1GB on 1 node (granularity=2MB)	558	59	143.5	54
Sequential write: 512MB from 1 node (granularity=2MB)	192	71	84	61.5

Table 2: Cluster workloads - completion time for NFS and NFS plugged to aIOLi

Values are given in second

On a traditional NFS server, the MPI I/O mechanisms exploited inside the parallel I/O intensive applications favor the sequential program: the optimizations are done on the client side which tends to reduce congestion issue on the NFS and thus enables to proceed the smaller tasks. Unfortunately, these optimizations are not well suited for a multi-applicative environment and the overhead generated by internal MPI I/O mechanisms becomes important. Regarding the NFS server plugged to aIOLi, both POSIX and MPI I/O performances are significantly improved for all the applications. The executions of all applications requires less than 2 minutes and 23 seconds for aIOLi whereas POSIX needs closed to 10 minutes and MPI I/O takes 14 minutes. Finally, once again,

the optimizations made by MPI I/O for parallel applications favor on the one hand the sequential programs but on the other hand add an useless overhead on parallel I/O intensive programs.

5 Future Work

During the performance evaluation of our extended NFS server, we found an unusual behavior due to the file system protocol granularity and file system implementation. When several processes are deployed on the same node, they they strive to access the NFS client layer which results in some starvation problems between them. To solve this problem, we need to force the file system client part to proceed the request in the order we choose. This can be done by issuing only one request at a time. In that case, even if requests are divided into smaller ones, they still are issued in the aggregation order. *aIOLi* already has the required internal structure to support this mechanism. We just need to insert it into the Linux Virtual File System on the client side and to launch another instance of *aIOLi* on the server side to make it work. Moreover, we observed that congestion issue on the server side is reduced by the MPI I/O optimizations made on the client side. An approach made of two aIOLi modules may also moderate the load of the file server.

This kind of approach will end up in a multi-level scheduler. On the local node, *aIOLi* chooses the best requests order according to the knowledge of all pending I/O operations generated by local processes. On the server side, *aIOLi* schedules requests taking into account the global traffic coming from all the nodes. Finally, an I/O scheduler in the Linux operating system chooses the most suitable requests order according to the layout of data on the physical storage medium. At the end of the day, each I/O operation is handled by many consecutive schedulers working at different levels and optimizing at different access granularity. We call this scheduling method *cascading scheduling*. A similar idea consists of exploiting the *meta node concept* used by several modern file systems (GPFS, NFSV4, Lustre, etc.) to provide consistency on files. Each time a process accesses a file for the first time, it becomes the meta node for this file and will be in charge of the coherency for this file. Our proposal is to add the *aIOLi* scheduler at the same level. Thus, the meta node will schedule I/O requests for this file. The *cascading scheduling* is thus deployed and well balanced on several nodes on the cluster which reduces congestion issues.

6 Conclusion

In this article, we presented *aIOLi* , a framework for high performance file access in distributed multi-applications environments. We emphasized all the original characteristics that make the strength of *aIOLi* : it is transparent to the applications because it is set between them and the underlying I/O subsystems, it maximizes the file system throughput by taking advantage of aggregation opportunities despite the distributed context, it maintains fairness between applications by using a quantum-based scheduling algorithm and it does not degrade interactive tasks behavior by deriving the scheduling algorithm from the MLF algorithm.

We validated *aIOLi* by conducting experiments: first on the widespread Bonnie++ and b_eff_io to observe overhead and scalability and second on IOR benchmarks and `cat`-like programs to evaluate efficiency. Results show that our approach dramatically improves performances when several distributed applications make simultaneous access to distinct files. Furthermore, as expected, our solution maintains fairness between applications and does not significantly degrade interactivity.

Future work will include *cascading scheduling*, the integration of specialized schedulers at different levels in the I/O resolving process.

We plan to build a hierarchical scheduling infrastructure with *aIOLi* master nodes using a remote server, typically a NFS one, integrating *aIOLi* too. Together with the *aIOLi* part on each client, this would lead to a cascading scheduler that could balance congestion point in the architecture to avoid bottlenecks no matter the granularity value.

We will also extend *aIOLi* to enable it efficient use with parallel file systems, by taking into account the distributed data layout. We currently study how *aIOLi* could be connected to a parallel version of NFS and to Lustre.

References

[1] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. K. Hollingsworth, J. Saltz, and A. Sussman. Tuning the performance of I/O intensive parallel applications. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 15–27, Philadelphia, May 1996. ACM Press.

[2] N. Bansal. *Algorithms for Flow Time Scheduling*. PhD thesis, Dept. of Computer Science, Carnegie Mellon University, 2003.

[3] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, 1996.

[4] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000. USENIX Association.

[5] F. Chen and S. Majumdar. Performance of parallel i/o scheduling strategies on a network of workstations. *Eighth International Conference on Parallel and Distributed Systems*, 2001.

[6] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[7] K. A. S. Elizabeth Shriver, Christopher Small. Why does file system prefetching work? In *Proceedings of the USENIX Annual Technical Conference - Monterey, California, USA*.

[8] D. Ellard and M. Seltzer. Nfs tricks and benchmarking traps.

[9] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling — a status report. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–16. Springer Verlag, 2004. Lect. Notes Comput. Sci. vol. 3277.

[10] F. Isaila, G. Malpohl, V. Olaru, G. Szeder, and W. Tichy. Integrating collective I/O and cooperative caching into the "clusterfile" parallel file system. In *Proceedings of the 18th Annual International Conference on Supercomputing*, pages 58–67, Sain-Malo, France, July 2004. ACM Press.

[11] R. Jain, K. Somalwar, J. Werth, and J. C. Browne. Heuristics for scheduling I/O operations. *IEEE Transactions on Parallel and Distributed Systems*, 8(3):310–320, March 1997.

[12] D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74. USENIX Association, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

[13] A. Lebre and Y. Denneulin. aioli: An input/output library for cluster of smp. In *Proceeding of the 5th International Symposium on Cluster Computing and Grid,Cardiff, UK*, May 2005.

[14] A. Lebre, Y. Denneulin, and T.-T. Van. Controlling and scheduling parallel i/o in a multi-applications environment. Technical Report Research Report RR-5689, INRIA, 38330 Montbonnot FR, September 2005. http://aioli.imag.fr/DOWNLOADS/aiolimaster-report-rr5689.pdf.

[15] X. Ma, M. Winslett, J. Lee, and S. Yu. Improving MPI IO output performance with active buffering plus threads. In *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, April 2003.

[16] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, A. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber. Structure and performance of the direct access file system, 2001.

[17] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. Best. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, October 1996.

[18] K. Pruhs, J. Sgall, and E. Torng. Online scheduling. In *Hanbook of Scheduling*, chapter 15. CRC Press, 2004.

[19] R. Rabenseifner, A. E. Koniges, J.-P. Prost, and R. Hedges. The parallel effective i/o bandwidth benchmark: b_eff_io.

[20] R. Ross. Reactive scheduling for parallel i/o systems, 2000.

[21] R. L. F. B. Schmuck. Gpfs: A shared-disk file system for large computing clusters. *In Proceedings of the 5th Conference on File and Storage Technologies*, January 2002.

[22] P. Schwan. Lustre : Building a file system for 1,000-node clusters. In *Proceedings of the Linux Symposium, Ottawa*, July 2003.

[23] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[24] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *Proceedings of USENIX*, pages 313–323, 1990.

[25] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, February 1999.

[26] M. Vilayannur, A. Sivasubramaniam, M. Kandemir, R. Thakur, and R. Ross. Discretionary caching for i/o on clusters. May 2003.

[27] A. C. L. W. E. R. W.K. Liao, K. Coloma and S. Tideman. Collective caching: Application-aware client-side file caching. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing*, July 2005.