Controlling and Scheduling Parallel I/O in Multi-application Environments

Adrien Lebre, Yves Denneulin, Thanh Trung Van

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Controlling and Scheduling Parallel I/O in Multi-application Environments

Adrien Lebre — Yves Denneulin — Thanh Trung Van

N° 5689

Septembre 2005

Thème NUM

ISSN 0249-6399 ISRN INRIA/RR--5689--FR+ENG

INRIA
RHÔNE-ALPES

Controlling and Scheduling Parallel I/O in Multi-application Environments*

Adrien Lebre , Yves Denneulin , Thanh Trung Van

Thème NUM — Systèmes numériques
Projet MESCAL

Rapport de recherche n° 5689 — Septembre 2005 — 19 pages

Abstract: As clusters usage grows, a lot of scientific applications (biology, climatology, nuclear physics ...) have undergone rewrites to harness the extra CPU and extra storage provided. These demanding software, besides handling huge amounts of data with peculiar parallel I/O access patterns, are run on clusters, environments where concurrency between those applications occurs.

Several propositions have been made to manage both the intensive parallel I/O applications and the cluster constraints. Nevertheless, available *Parallel File Systems* or *Parallel I/O Libraries* are based on specific API's, which limit portability and require good knowledge of their internal mechanisms to get good performances. Moreover, *Parallel I/O Libraries* are usually focused on running only one application without taking into account the load that the other ones generate on the cluster. This paper presents a new strategy to handle parallel I/O in a multi-application and distributed environment. Our framework detects parallel I/O accesses without inter-processes synchronization mechanisms and uses a simple interface based on the classic UNIX system calls (`creat/open/read/write/close`). In addition, we analyze two scheduling strategies to improve global performances and provide fairness between applications. Early experiments have given promising results and have shown that using such approaches may lead to better performances as well as improvements of quality of service.

Key-words: aIOLi, Parallel I/O, cluster, SMP, HPC, File Systems

* This work is done in the context of the joint research project MESCAL supported by CNRS, INPG, INRIA, and UJF and the project LIPS between INRIA and BULL Lab. Computer resources are provided by the g5k cluster (further information at `http://www.grid5000.fr/`).

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 —Télécopie +33 4 76 61 52 52

Régulation et Ordonnancement des Entrées/Sorties Disques dans les Environnements Multi-Applicatifs †

Résumé : Un grand nombre d'applications scientifiques (biologie, climatologie, ...) utilise et génère des quantités de données qui ne cessent de croître en plus de modes d'accès parallèles qui leur est particulier. Aux problématiques "out-of-core" ou "parallel I/O" longuement abordées dans un contexte d'accès local et mono-applicatif, viennent s'ajouter les considérations et les contraintes imposées par un environnement distribué et multi-applicatif, comme l'est une grappe. Plusieurs approches ont été proposées par la communauté scientifique dans le but d'améliorer les performances lors d'accès parallèles à des fichiers distants (systèmes de fichiers ou encore bibliothèques d'entrées/sorties spécialisées). Toutefois, ces solutions intègrent des API plus ou moins lourdes mais surtout spécifiques qui nécessitent une connaissance exacte de chaque subtilité interne au modèle. De plus, ces solutions proposent des mécanismes permettant d'améliorer les performances au sein d'une même et unique application sans tenir compte de la charge induite par les applications concurrentes.
Ce rapport présente une solution globale pour l'accès aux données au sein d'une grappe. Notre approche permet de découvrir les accès parallèles émanant des applications afin de les réguler et limiter ainsi les phénomènes de type "goulet d'étrangelement" dans un premier temps. À partie de nos précédents travaux, cette solution ne nécessite pas d'API spécifique et est totalement transparente pour les utilisateurs. Deux stratégies d'ordonanncement ont été incluses au sein de notre prototype afin d'optimiser les performances tout en tenant compte du critère d'équité entre les applications. Les performances obtenues avec ce prototype sont prometteuses et ont révélé qu'une telle approche peut améliorer nettement les performances globales tout en proposant une qualité de service paramétrable.

Mots-clés : aIOLi, Entrées/Sorties Parallèles, HPC, grappe, SMP, système de fichiers

Contents

1 Introduction

I/O bottlenecks have always been a major issue in Computer Science [2], and it is likely to continue as I/O hardware performances increase slower than the ones of CPU and memory, [7]. Furthermore, this gap is amplified by the increasing use of clusters of workstations or SMPs as well as the number of scientific application developments (molecular biology, climatology, nuclear physics ...) with ever more demanding I/O requirements and different patterns of access (for instance, in a parallel matrix product, each process has to access specific parts according to the array-distributions used). As a consequence, lots of researchers have attempted to develop new I/O sub-systems that take into account both hardware aspects and parallel computing accesses. The proposed systems often have too many features, which complicates developments and the maintainability part of scientific applications. Moreover, they require deep knowledge of their specific API and model subtleties in order to achieve performances.

These aspects have been emphasized in our previous work[1] [11], we reported several inefficient points in the classical way to treat remote I/O accesses in a SMP context:

Sending of I/O requests in parallel :
Several requests of various applications are sent in parallel to the same I/O server. This kind of I/O behavior drastically impacts performances and generates congestion because of a bad scheduling policy on the remote storage server.

Lack of transparent aggregation mechanisms :
Common POSIX I/O requests such as `read()` or `write()` are usually sent to I/O server without checking if they can be aggregated. Thus potential benefits for larger accesses are lost. The time to fetch data can be strongly decreased if the client I/O stack provides such a mechanism.

[1] aIOLi library for parallel I/O

All those facts led to the development of the first version of the aIOLi prototype: an efficient and transparent I/O library for parallel access to remote storage from one SMP node (intra-node, section 2).

In this second work, an approach to distribute the former concepts to cluster level (inter-node) is introduced. It consists in designing a system of coordination (or regulation) between I/O requests of various processes (dependent or not) in order to synchronize their requests and to aggregate them if possible. The remaining parts of the paper falls into the following parts: section 2 reviews succinctly the concepts of the aIOLi solution within one SMP node and its limitations. Section 3 gives an overview of our system and its main objectives. Then, section 4 deals with some technical issues. Section 5 focuses on two scheduling policies : a variant of WSJF[2] and an MLF[3] based approach which have been included in our framework to improve efficiency without decreasing fairness between applications. Section 6 gives some early results. Some related works are discussed in section 7. Eventually, section 8 concludes and describes further extensions and improvements.

2 Former aIOLi prototype

The aIOLi system was born from the need to have powerful mechanisms for data access within a cluster without having to use a complex and tedious API. Relying on the ubiquitous interface of the standard C library, the first implementation of the aIOLi prototype [11] provides an efficient management of parallel I/O within a SMP node : I/O requests sending gets some synchronization so that they may be sent in a sequential manner to avoid the parallel sending to a same I/O node which can be suboptimal. The temporary storage of requests within a centralized point let the accesses be analyzed in order to find aggregation possibilities and to apply various optimization techniques. It is composed of two components: a client module, linked to the application, overloads the standard calls (`open/close/read/write/lseek`) in order to redirect them towards a server module (the aIOLi daemon) in charge of handling these accesses. Each time a process generates a request, the client module will send this request to the daemon where it will be kept in a queue before being transmitted to the data server. The daemon is a multi-threaded process: a first thread receives the requests from the clients and stores them in the corresponding queues. Many I/O threads analyze these queues, aggregate the contiguous requests before executing real system calls.

One of the strong points of this library is its ease of usage; it does not require the users to learn the API of a new library, which could be complex. It only overloads the standard C interface, which lets the approach being used for all the POSIX architectures that is nearly all modern architectures.

Experiments carried out with this prototype gave promising results compared to the performances provided by POSIX interfaces or even ROMIO, the most deployed MPI I/O implementation. Currently, we are developing a Linux Kernel Module to provide aIOLi mechanisms in a full transparent way. A more detailed analysis of the first prototype as well as the evaluations are available on the site `http://aioli.imag.fr/`. As written in the introduction, this second paper deals with the distribution of these concepts to cluster level and will be described in the following sections.

3 aIOLi at cluster level

From the interesting results of the first version of aIOLi, we decided to study the implementation of a similar approach but at cluster level. The principles of this second version are closed to the first one but now we have to manage the I/O requests coming from many nodes. The lack of a global memory and a global clock makes the management more difficult. Moreover, we want to provide a framework handling the I/O requests at 3 distinct levels:

1. Intra-node coordination: Management (synchronization and aggregation) of I/O requests within one node to treat the I/O accesses of one application. For example, in a matrix product, each element of the matrix is computed by one SPMD instance[4]. This module is already provided by the first version of aIOLi.

2. Multi-node coordination: Management (synchronization, aggregation and scheduling) of I/O requests belonging to the same application deployed on many SMPs nodes on a cluster. For example, in a matrix product, each node is in charge of a part of the computation. This part is distributed between different processors of the concerned node. In this case, it is important to synchronize the requests coming from

[2] Weighted Shortest Job First
[3] MultiLevel Feedback
[4] Single Program Multiple Data

different nodes in order to use in an efficient manner the storage system. The basic idea consists in providing some mechanisms such as collective I/O but still in a transparent way.

3. Multi-application coordination: combination of (1) and (2) when taking care of concurrent execution of many applications. In this mode, several applications can send their requests at the same time. For example, two applications run in concurrence in one cluster, the first one doing matrix computation while the second one is processing FFT[5]. We want to include some scheduling policies to provide efficiency as well as fairness.

To provide such features, we have to solve several constraints :

Synchronize the I/O requests coming from several nodes to avoid inefficient parallel accesses to the same I/O node. This first difficulty could be treated as a distributed mutual exclusion problem (section 4.1).

Aggregate contiguous requests to bring efficiency and to favor large access. As a physical aggregation requires complex and expensive mechanisms to be implemented in a distributed environment, we choose to exploit a derived approach that we called "virtual aggregation" (section 4.2).

Schedule I/O requests to obtain a good ratio of fairness/performance between different applications in the cluster and between different processes of one application (in case of SPMD application). The interactivity parameter should also take into account even if we are in HPC[6] context. Users monitoring tools should not starve due to concurrent intensive I/O applications (section 5).

The two following sections will describe more clearly these problems.

4 Technical issues

This section presents technical aspects related to the prototype and the proposed solutions. The synchronization of I/O requests and aggregation mechanisms are discussed. Due to the importance of the suggested strategies, we choose to deal with scheduling policies in a subsequent section.

4.1 Synchronization of I/O requests

As we mentioned during the previous section, sending many requests to the same I/O server in parallel can lower the overall performance of the system. One of the major purpose of this version of the aIOLi solution is to synchronize the I/O requests coming from several nodes in cluster. Ideally, there should be only one request at server side at each moment. In our context, the problem of I/O requests synchronization is similar to the distributed mutual exclusion problem. There were many researches done about this problem [24, 17]. In our case, the most important criteria are:

Number of message used in each synchronization round,

The synchronization delay, which is the time between the execution of two requests.

The simple solution for this synchronization problem consist in using a "master server" in charge of controlling the I/O requests. At each time a client, to access the storage server, will send a demand message to the master server [7] (1). The master server keeps this message in a waiting queue and informs the client when the resource is freed (2), then the client can begin to access to storage server. When the client completes its request, it will inform the master server (3) then the master server can grant this resource to another client.

There are several drawbacks to this approach: first, it is not fault tolerant[8] ; second, the master server can be overloaded by several requests from several clients. The last but not the least concerns the synchronization delay which is always $2T$ (T is the required time to send one message on the network). This parameter is preponderant in our case : on the one hand, $2T$ could be non-negligible according to the required time to process a small request and on the other hand, the number of request could be quickly very large (which could generate an important overcost : $n * 2T$).

[5] Fast Fourier Transform
[6] High Performance Computing
[7] In our case, the aIOLi server.
[8] Single Point Of Failure

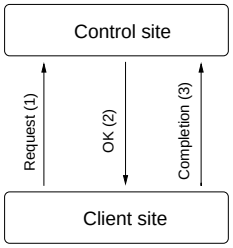


Figure 1: Simple synchronization mechanism
A client sends a request to access to the resource (1) ; the server informs the client when the ressource is available (2) ; after completion the client notifies the server (3).

Based on the prediction of the execution time, we implemented an enhanced approach of this algorithm to reduce the synchronization delay: each time the master server receives a demand message, it will compute the related approximated execution time and will use this value to notify just in time the next demand. The execution time of one request is estimated by the following formula:

$$execution\_time = \frac{request\_size}{disk\_bandwidth} \quad (1)$$

The "completion request" is then exploit to fix the potential degradation in the prediction (adjust the current bandwith, ...). Unfortunately, the success of this method is dependent on the network characteristics and the accuracy of the prediction of execution time according to the server load. In the optimum case, the real time is equal to predicted time, figure 2 (a). However, if the real time is lesser than predicted time, the disk is not used, efficiently, figure 2 (b) ; finally in the last case if the real time is greater than the predicted time, the second client will begin its transfer whereas the first site is carrying out its request, figure 2 (b). To minimize case (c) which could quickly lead to bad performance, we add a constant k to the former formula. This value should be set between 0 and $2T$ according to the hardware architecture.

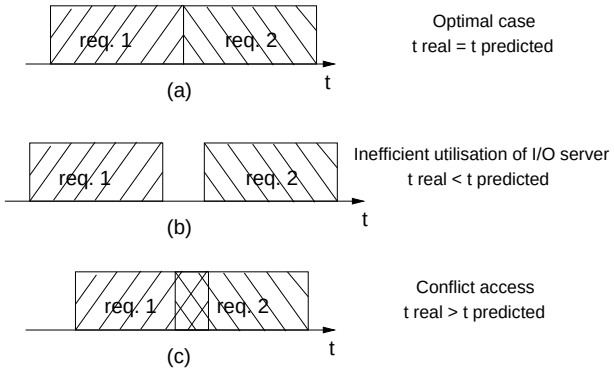


Figure 2: Prediction approach cases
From a theoretical point of view, the purpose of a mechanism of synchronization based on the prediction is to reduce the synchronous delay (optimal case). Unfortunately, the accuracy of the prediction depends on hardware characteristics.

The first experiments disclosed the difficulty to make accurate prediction as well as find the right value for the constant k (section 6.3.3).

4.2 Physical aggregation vs. virtual aggregation

In the first version of aIOLi, a physical aggregation mechanism has been implemented: small contiguous requests are merged in one larger request that is sent to the remote file system. For example, the data return by a `read()` call were retrieved in an internal buffer before to be redistribute to final user buffers. Since we were within a node (one operating system), coherence mechanisms was provide by the under file system stack (single buffer cache). In a distributed environment, the implementation of such mechanism becomes more difficult. While simple shared-memory segments used to be enough to aggregate and redistribute data, dedicated complex

protocols are now mandatory to offer the same coherence warranties (data replication, cache invalidation, ...). So we decided to implement a variant that we call "virtual aggregation". The idea of this method is to find the contiguous requests and set an execution order as if all requests would form one larger : For example, the three following requests *read(30,40)*, *read(20,30)*, *read(10,20)*[9], requests will be reordered and executed in the order: *read(10,20)*, *read(20,30)*, *read(30,40)*. In this way, disjoint accesses become contiguous. This "virtual aggregation" can exploit cache mechanism like read ahead (from client and server sides) and improve the global performance. We could take into account the stripe criterion in case of parallel file systems (section 5.3).

5 Scheduling of I/O requests on a cluster

Scheduling I/O is somewhat similar to the problem of process scheduling in an operating system. In our context, if several processes (or applications) want to access disk resource which is normally limited in a cluster, we have to propose a policy to share this resource between them in a reasonable manner. The following example illustrates the necessity of such a policy : Suppose that we have 3 I/O requests (numbered 1, 2, 3 respectively) being sent from 3 different applications to the same I/O server. The amounts of accessed data are 10, 1, 5 KB, respectively. Suppose that these requests are executed according to the FIFO[10] order and the execution time of a request matches its data size. The execution time of the first request is 10 time units. The execution time of the second request is only 1 unit but it has to wait 10 units from the first request : so the response time of the second request is $10 + 1 = 11$ units. Similarly, the response time of the third request is $10 + 1 + 5 = 16$ units. The total response time of the three requests is $10 + 11 + 16 = 37$ units.

Now we consider another scenario, the requests are executed in the "Shortest Job First" (SJF) order. That means the smallest request will be executed first. In this case, the execution order of the requests is 2, 3, 1. The response time of the first request is 1 unit, the response time of the second request is $1 + 5 = 6$ units and the response time of the third request is $1 + 5 + 10 = 16$ units. Thus, the total response time of the three requests is only $1 + 6 + 16 = 23$ units!

In fact, each scheduling strategy is optimal for a particular purpose. The FIFO strategy in our example is good to minimize the maximal response time while the SJF strategy can minimize the total response time. The criteria in our model are complex. We try to maximize the overall performance by aggregating as many requests as possible but we have to assure a minimal fairness between applications and avoid the starvation problem[11]. The I/O requests scheduling in our context is online non-clairvoyant problem : the jobs (requests) arrive in time and their size are unknown in advance due to the aggregation process [3]. So we propose two scheduling algorithms which are described below.

5.1 Algorithm Weighted Shortest Job First

From the above example, we can see that the algorithm SJF is efficient to minimize the total response time. However, this is a clairvoyant algorithm and it can cause the starvation problem with big requests. So we propose to apply a variant of this algorithm that we call "Weighted Shortest Job First". This algorithm can exploit the advantages of the SJF algorithm and avoid the starvation problem. The idea came from the paper [22]. In this algorithm, a virtual measurement of request size is used instead of its real size. Suppose that the execution time of one request is Tr, the waiting time of this request since its arrival is E and M is a constant number; the virtual execution time of this request can be calculated by:

$$Tv = Tr * \frac{M - E}{M} \tag{2}$$

Each time the scheduler have to choose a request, it will select the one with the smallest virtual size. Because the virtual size of a request will lower with time, the starvation problem is avoided. However, in our model, the size of one request is not fixed in advance: it can merge with other requests and becomes bigger by virtual aggregation. In addition, the number and the arrival time of these requests are not known in advance, too. So we have modified the above formula like this:

$$Tv = \sum_{i=1}^{i=n} Tvi \tag{3}$$

[9] `read(x,y)`: read from offset x to offset y in the same file
[10] First In First Out, also known under First Come First Served algorithm
[11] An application has to wait indefinitely for the resource

In this formula, n is the number of requests in the "aggregated request", Tvi is the virtual execution time of the i^{th} request that is calculated by the formula 2.

There is one more problem with this algorithm. In fact, if the size of an "aggregated request" is not limited, it can grow with time and the starvation problem still remains if other applications generate many small requests in parallel. To tackle this problem, we choose to limit the real size of an "aggregated request" by a threshold. If the aggregation process finds a request that has a size greater than this threshold, it will "break" this request into two small requests: the size of the first one is M and size of the second one is $Tr - M$.

The algorithm WSJF is illustrated in the following example (figure 3)

At the first step, there are 4 requests from application A1, 3 requests from application A2. Suppose that the execution time of each request is 5 and M is 30. After the first aggregation process, the formula 3 gives us the virtual sizes of "aggregated requests" of A1 (=20) and A2 (=15). So the "aggregated request" of A2 is executed first.

At the second step, when the "aggregated request" of A2 has just finished, 1 new request of A1 and 4 requests of A3 come to the aIOLi server. These requests are reordered by the aggregation process. The waiting time of the 4 first A1 is 15 units (this is the execution time of the "aggregated request" A2 selected in the first step) and the waiting time of the new request A1 is 0; so the virtual size of "aggregated request" A1 is $4 * 5 * (30 - 15)/30 + 5 * (30 - 0)/30 = 15$ units while the virtual size of "aggregated request" A3 is $4 * 5 = 20$. So the "aggregated request" A1 is selected because its virtual size is smaller even if its real size is bigger than the real size of the the "aggregated request" A3.

At the third step, while the "aggregated request" A1 is being executed, other requests A3 come to the aIOLi server. In this case, the aggregation process returns two "aggregated requests" because if there is only one request then the size of this request will be greater than M.

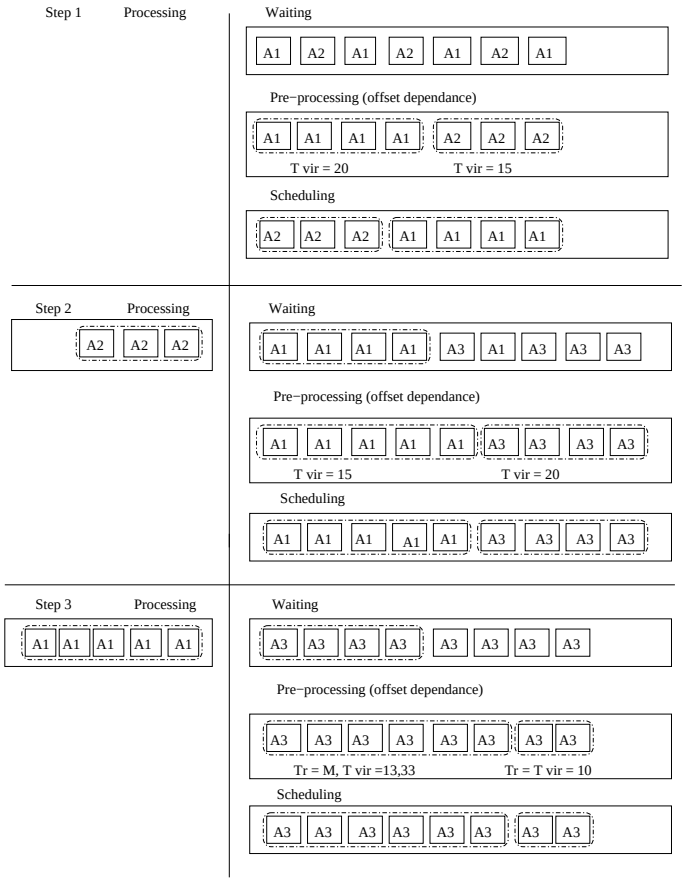


Figure 3: Algorithm WSJF

5.2 A variant of algorithm Multilevel Feedback

The second algorithm that we want to try to apply in our model is a variant of algorithm Multilevel Feedback (MLF), [16], one of the algorithms for process scheduling in the Unix operating system. This variant is described as follows:

Each time the scheduler want to select a request to execute, it will propose a time quantum for each request in the waiting list. If the execution time of a request is lesser or equal to this time quantum, this request can be selected.

If there are several requests satisfying the selection condition (execution time is greater or equal to the proposed time quantum), the FIFO criterion will be applied between these requests.

The proposed time quantum is not identical for every request. The "new" requests will receive smaller quanta than "old" requests. If a request is not selected after a selection time, it will be proposed a greater quantum in the next time (k time greater than the last time). This approach can avoid the starvation problem because the proposed quantum for one request can grow by the multiplication factor with the waiting time.

The figure 4 illustrates this algorithm:

At the first step, there are 4 requests of application A1, 3 requests of application A2 and 1 request of application A3. Suppose that the execution time of each request is 5 and the original quantum proposed is 10. At the first time, the requests are aggregated in the pre-processing phase, then a time quantum is proposed for each request. Because this is the first time these requests are in the waiting queue, the proposed time quanta are 10 for every request. The execution times of "aggregated requests" A1 and A2 are 20 and 15, greater than the proposed quanta so they are not selected. The execution time of request A3 is 5, this request satisfies the selection condition so it is selected in this round.

At the second step, when the request A3 is being executed, others requests A2 and A3 come to aIOLi server. The requests A2 are aggregated in the pre-processing phase. Because this is the second time the "aggregated requests" A1 and A2 are in the waiting queue, they are proposed a greater quantum. In this example we choose the multiplication factor $k = 2$, so the proposed quanta for these two requests are $10 * 2 = 20$. The request A3 is proposed an original quantum (10) because this is the first time this request is in the queue. The size of requests A1, A2, A3 are 20, 25, 5 respectively while their quanta are 20, 20, 10. The requests A1 and A3 satisfy the selection condition, so the criterion FIFO is applied to choose a request to process. In this case the "aggregated request" A1 is selected.

Similarly, while the request A1 is being executed, others requests A2 and A3 come to aIOLi server. After the pre-processing phase, the size of "aggregated requests" A2 and A3 are 30 and 20. Because this is the third time the request A2 is in the queue the proposed quantum for this request is $20 * 2 = 40$ and the proposed quantum for the request A3 is $10 * 2 = 20$. The two requests A2 and A3 satisfy the selection condition, so the "aggregated request" A2 is selected according to the FIFO order.

5.3 Parallel file system constraint

In this section, we discuss the constraint related to the parallel file systems. In these systems, the data are often distributed on several storage nodes[12] (*data striping*) . Since data of a request can be distributed on several nodes, we must consider the relation between these sub-requests of the same application. For example, assume that we have two requests R1 (of application A1) and R2 (of application A2) of size of 15 and 10 KB and two I/O nodes having the "stripe size" of 5 KB. Then R1 is devised in 3 sub-requests and R2 is devised in 2 sub-requests (the size of each sub-request is 5 KB). Assume that we use strategy WSJF without checking the constraints between sub-requests of the same application and the execution time of each sub-request corresponds to its size, i.e. 5 time units for example. Then, as shown in the figure 5, the execution time of R1 is 15 units and R2 is 10 units; thus the total response time of the two requests is 25 units. By taking into account of data distribution, if sub-requests of A2 on both I/O is carried out before sub-requests of A1, the execution time of R2 is of 5 units and R1 of 15; in this case the total response time is 20 units.

[12] This distribution reduces congestion problems since accesses are balanced over I/O nodes.

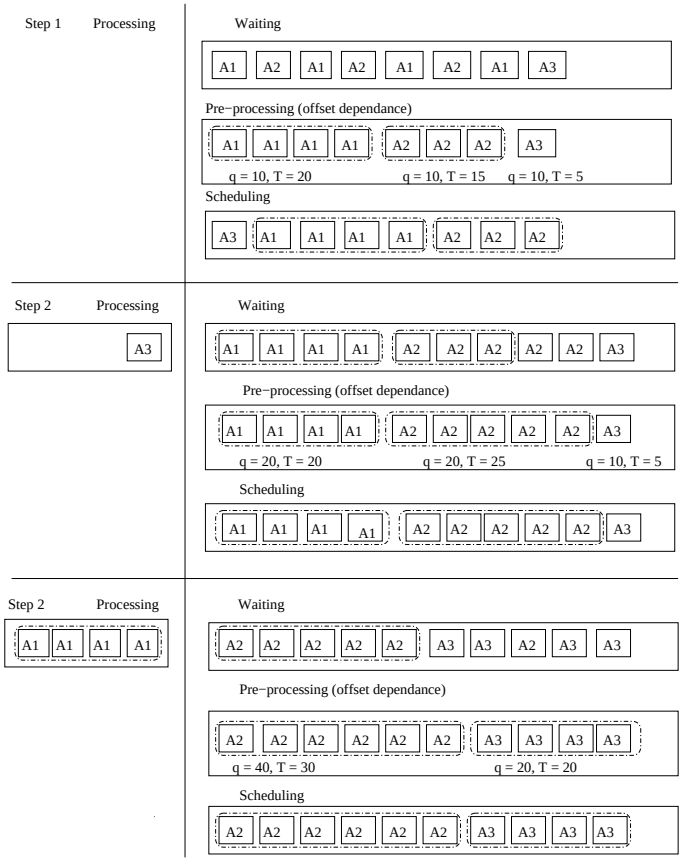


Figure 4: Variant MLF

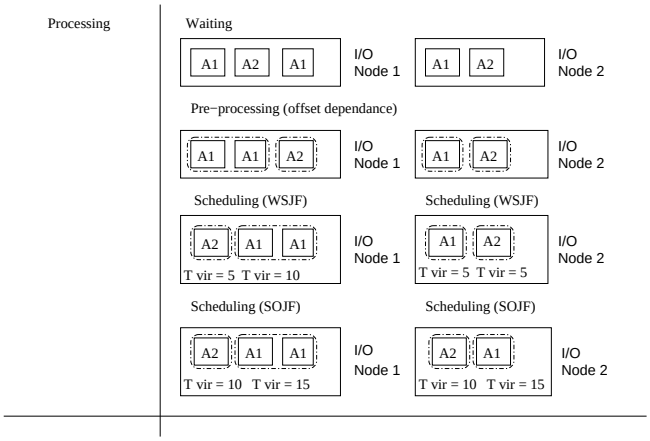


Figure 5: Constraint between sub-requests

The above example shows the importance of constraint between sub-requests in the parallel file systems. A simple idea is to synchronize sub-requests of the same application so that they are carried out at the same time. A possible solution is to apply a common selection criterion for all sub-requests belonging to the same application. For example, we can apply strategy WSJF in a independent way to each I/O queue while taking into account the total size of the request for the selection criterion (instead of the size of sub-requests). This method is similar to the approach "Shortest Outstanding I/O Demand Job First" or SOJF, [4]. In this example the request R2 has the smallest size between the two requests; approach SOJF selects sub-requests of A2 (total size is 10) and then sub-requests of A1 (total size is 15) which could be better. However this method, like method SJF, can involve starvation problems with the requests having big total size. So we can use the weighting method exploited in algorithm WSJF to treat this problem. Actually, we have not integrated this last aspect into our prototype, this work is an additional prospect for this work and will require a more thorough study.

6 Experiments

Based on the principles discussed in the previous sections, we built a second prototype of the aIOLi solution. Developed in C, this second implementation has to be linked to the application and could be used in coordination of the former version.

6.1 Prototype implementation

The system is based on a traditional client-server model, figure 6: a client module that overloads the POSIX calls is linked to each application and a server daemon is deployed at a aIOLi server to centralize and regulate the I/O requests of client. With regards to the size of the cluster, the aIOLi server could be deployed on the file server node as well as a distinct node. The client requests are transmitted by a `tcp` channel to the aIOLi server. Each request is kept in a corresponding queue according to the concerned file, access offset, request size and arrival time on the server side. A "timer[13]" based approach enables to notify clients in time when the prediction mode is enabled. The scheduling policy as well as the prediction mode are set at the launch time.

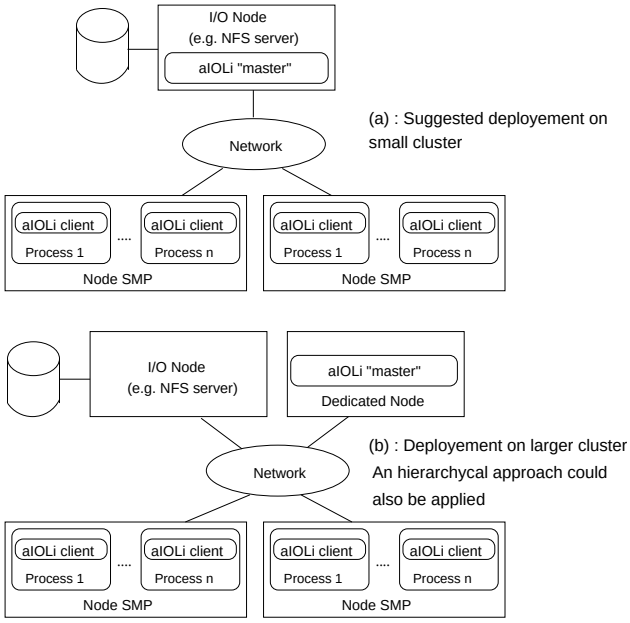


Figure 6: Conceivable system architectures

[13] POSIX `SIGALRM` signal.

6.2 Experimentations

The testing system is a sub part of the grid "grid5000"[14] located at INRIA south site (Sophia-Antipolis - FRANCE). Each node (a IBM eServer 325) is composed by two AMD opteron (2GHz), 2GB RAM and a 80GB IDE hard-drive (bandwidth estimated to 57MB/s by `hdparm command`). The cluster is interconnected by a giga-ethernet network. The operating system Debian Linux was used. A dedicated NFS server (version 3, TCP, 32Kb read size) and several SMP nodes have been exploited.

The benchmarks correspond to the cases mentioned in section 3 : multi-node coordination (one application distributes on several nodes) and multi-application coordination (concurrency between many applications distribute on several nodes). In the first experiment, section 6.3, one MPI application (8 instances) decomposes a 2GB remote file stored on the NFS server. It enables to evaluate the performance of our solution in presence of parallel accesses (automatic discovering of parallel patterns). The second benchmark, section 6.4 concerns two MPI applications deployed on two distinct nodes (each composed of 4 instances) : they decompose concurrently two different files (2*2GB) stored on the NFS server. Thanks to this test, we analyzed how our solution handles the concurrency between two parallel I/O applications and how its impacts on the fairness criterion (i.e. difference of completion times). The last experiment, section 6.5 consists in observing the behavior of such a approach in a "real" case : 5 distinct MPI applications decompose 5 files in a concurrent manner. To avoid the cache influence, each experiment uses a different file at each execution. Every decomposition have been executed with different file access granularity to analyze the influence related to the number of messages transmitted to the aIOLi server. In addition, we launched each benchmark on both architectures (figure 6) : NFS and aIOLi solution on the same and then on distinct nodes.

6.3 Multi-node coordination

6.3.1 Detection of parallel access schema

In the preliminary tests, we quickly observed that the use of the algorithms WSJF or MLF were not suited to maximize "virtual aggregation" within one application which led to bad performance. We slight modified these algorithms in order to obtain more virtual aggregations. The following example illustrates this idea: suppose that we have four requests of the same size that need to be sent to the same data server (`read(10,20)`, `read(20,30)`, `read(30,40)`, `read(40,50)` on the same file [15]). Suppose that requests `read(10,20)` and `read(30,40)` come to the aIOLi server at the same time while the requests `read(20,30)` and `read(40,50)` come to aIOLi server later. These requests are disjoint, so they can not be aggregated. At the next step, the selection criterion of the two algorithms are applied to select a request to be executed. Because the requests `read(10,20)` and `read(30,40)` have the same size and the same arrival time they have the same priority. However, if the request `read(30,40)` is selected and executed before the request `read(10,20)`; in the next step when the requests `read(20,30)` and `read(40,50)` come to the aIOLi server, the waiting queue will contain disjoint requests: `read(10,20)`, `read(20,30)` and `read(40,50)`. On the other hand, if the request having the smallest offset, `read(10,20)` is selected, the waiting queue will contain the requests `read(20,30)`, `read(30,40)` and `read(40,50)` that can be aggregated in one request. From these observations, the first idea is to select the request having the smallest offset among the available requests while keeping the principal goal of each algorithm. In the algorithm WSJF, we have introduced a junction coefficient: between two requests, a request will be selected if its offset is smaller than the second request's offset and its virtual size is not greater than a determined ratio of the second request. We set this ratio to 10%. In the algorithm MLF, the FIFO criteria between two requests are replaced by the offset criterion if they concern the same file (the smallest offset is preeminent).

6.3.2 Results

The figure 7 (a) and 7 (b) illustrate the required time to make the decomposition over 8 MPI instances. The "Posix" curve corresponds to the classical `read()` call, the "Two Phases" to the collective I/O mechanism provide by ROMIO [25]. The 4 last curves show the results according to the related scheduling policy ; prediction mode is disabled and enabled (`XXXX-TIME`).

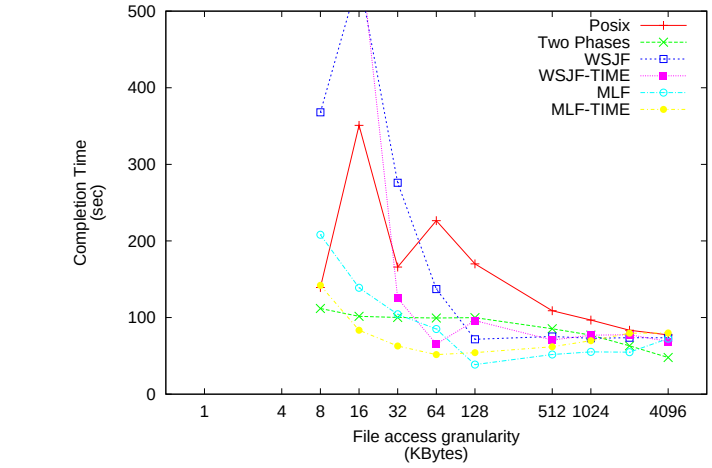
From a global point of view, we could see that the aIOLi server doesn't really impact on NFS performance in this first case. The curves WSJF, WSJF-TIME, MLF and MLF-TIME have the same form on both graphs. However we could note a slightly overhead for smallest granularity due to the importance of network messages.

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

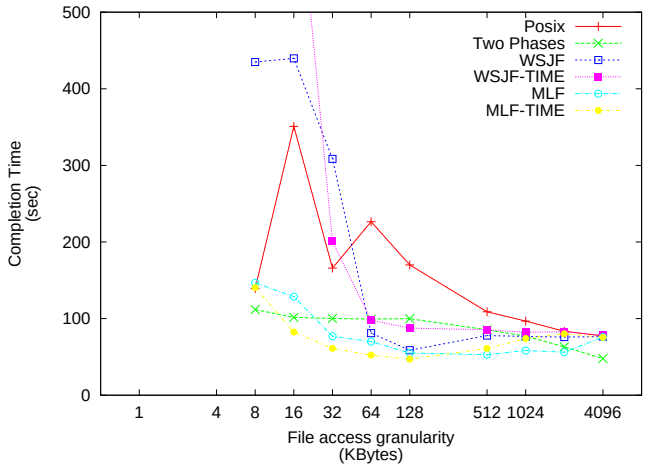
[15] `read(x,y)` : read from offset x to offset y in the file.

So let's focus on the graph 7 (a) : for the mode without prediction, no profit is brought by the two algorithms for the accesses less than 32 Kb. Indeed for small accesses (less than 32 Kb), each request between 1Kb and 32Kb generates a NFS read request of 32 Kb. So many requests can be directly satisfied by the client cache. In the aIOLi approach, every request (regardless of granularity access) are sent to the aIOLi server where they are synchronized and maybe aggregated. Thus, for each request an overhead of T is added (section 4.1). For example, with a 2GB file decomposition at 8 Kb granularity, the number of necessary requests are 256000. In our network, the synchronization delay is about 100μs, so 27 seconds are necessary to realize the synchronization process.

In addition, we can observe that the junction coefficient used in WSJF is not initialized with a sufficient value. The requests are poorly reordered (bad detection of parallel access schema), the client does not benefit from the cache and on the contrary generates a multitude of disjoined accesses. The MLF approach profits quite well from the cache effect so it generates a small overhead compared to the standard approach.



(a) aIOLi server on NFS



(b) aIOLi server on a dedicated node

Figure 7: 2GB file decomposition by one application
8 MPI instances decompose a file stored on a dedicated NFS server. The "Two Phases" corresponds to the collective I/O operations supplied by ROMIO.

Throughout the experiment, the algorithm MLF behaves better because it always chooses the request having smallest offset which maximizes "virtual aggregation" latter. From 32 Kb, the aIOLi approach becomes powerful for the two algorithms ; the POSIX requests can not be satisfied by the client cache and as consequence generate conflicts at the file server side which have to manage them in parallel. For larger granularity, the required requests number for the decomposition becomes less (for 4MB only 512), and the synchronization time becomes transparent. Finally, without exploiting tedious routines like "Two Phase approach"required (`MPI_File_Set_View()`, `MPI_File_Read_All()`, ...), we reach similar performance (even better for file access between 16KB to 1MB) with the MLF approach.

6.3.3 Observed problems

Regarding the performance reached in our former implementation [11], we traced our application to analyze and understand why we could not provide better gains. We discovered two inefficient behaviors that we describe in the following part.

Shift phenomenon

In spite of the changes described in section 6.3.1, we continued to observe a problem that we call "shift phenomenon": the order and the periodicity of the arrival of requests coming from the various processes are dependent on a greater number of factors than in the preceding version, [11]. Indeed, there is a strong dependence between the performances and the scheduling policy of the processes established by the Linux system within a same node. A reflection window used in the anticipatory scheduling approach had been integrated to partly solve the problem. In our case, this dependence is amplified by the number of nodes taking part in the decomposition since the scheduling strategy of each node is independent. Moreover, we have to take into account the network factors. Thus a process can deliver a new request as soon as its last request was treated whereas another has a much longer re-emission period. The figure 8 illustrates this phenomenon: four processes take part in the decomposition of the same file. At step 1, the request of process P0 arrives late compared to the other requests (P1, P2, P3). The requests of P1, P2, P3 are contiguous and are aggregated in the same "virtual" request. When the P0 request arrives to the aIOLi server, the acceptance message for the request P1 was already delivered, so it cannot be aggregated in this "virtual" request and is inserted in the queue. It is executed at step 2 after the treatment of the whole previous aggregated request. When the request of P0 is being executed, the new requests of P1, P2 and P3 arrive and are aggregated in step 3. The scenario is repeated in the next steps. Requests coming from P0 can never be aggregated with requests P1, P2, P3. This problem can degrade 20% of overall performance of the system.

Prediction problem

The launching of new tests enabled us to detect an additional problem related to the prediction method: when a prediction is erroneous, it will influence the following predictions. If the duration of a request is given in a false way, the acceptance message of the following request will be erroneous and the new request will begin too early (conflict of accesses, figure 2 (c)). There will be a considerable performance reduction since the two accesses will be treated in parallel. In a similar way, the second prediction will be also false, followed by the performance reduction and will generate a new derive on the following one and so on. The difference between predicted and real time will grow with time. We have slightly improved our system in order to take into account this problem with each new prediction. However, with the accesses of small size, we have not yet been able to reach an exact prediction. A similar problem was presented in [23].

6.4 Multi-application coordination

This experiment is done to evaluate our approach with the execution of many parallel applications in a cluster. The application in the previous test is re-executed on two distinct nodes (4*2 instances MPI). Each application realizes a decomposition on a 2GB file stored on a remote NFS server. The results are presented in the figures 9 (a) and 9 (b). Like the previous experiment, the aIOLi server doesn't impact on the NFS server.

The approach without prediction generates an important overhead for low granularity due to the number of synchronization messages (2*2GB to decompose by block of 8K) and the synchronization delay implied between two demands.

In addition, we can notice that even in the MLF with time prediction, the performances are worse than the ones given by the Posix approach. The MLF policy implies lot of switches between the requests from the first

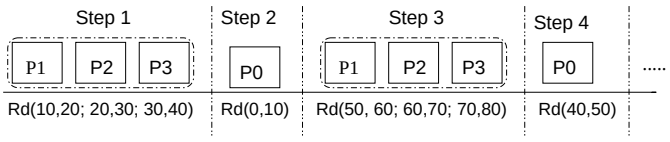
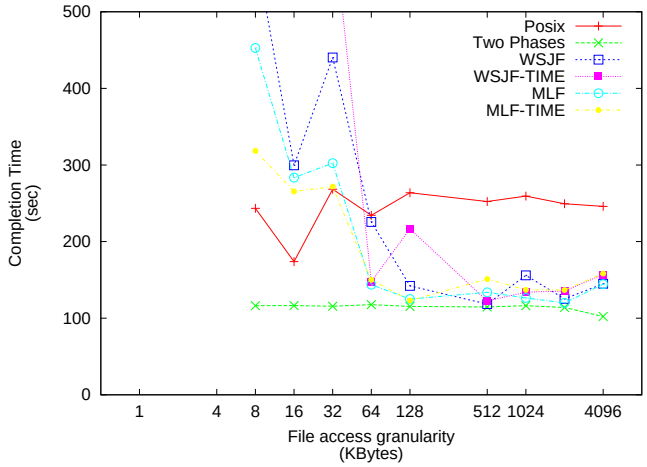
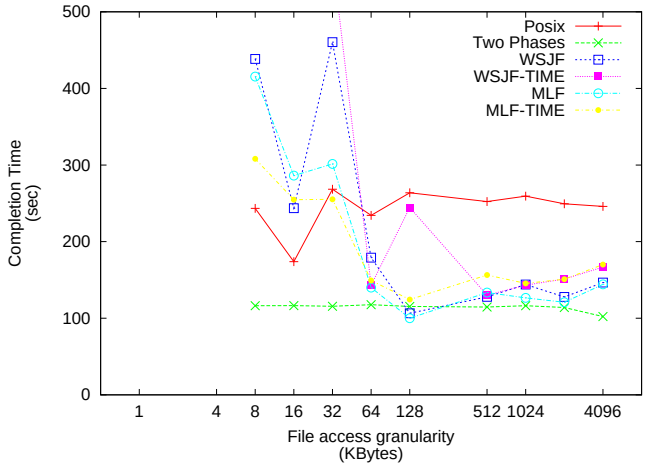


Figure 8: Shift request
Arrival order of messages has an influence on the aggregation process. In this example, requests of P0 can never be aggregated.

application and the second one. These "I/O switches" do not benefit of cache mechanism. For larger accesses, we can observe a slight fall of the performances that will be confirmed in the next experiment (section 6.5). It would be interesting to look further into this behavior and to analyze if the degradation of the predictions is stronger for large accesses and/or in multi-application mode. It is in particular the same for the WSJF approach. From a global point of view, the collective I/O approach provided by ROMIO gives the best performance in this case. Actually, The "Two Phase" approach sends only for 4MB requests (ROMIO internal size buffer). In addition, since two applications are executed, the NFS server has to process only two requests at each time.



(a) aIOLi server on NFS



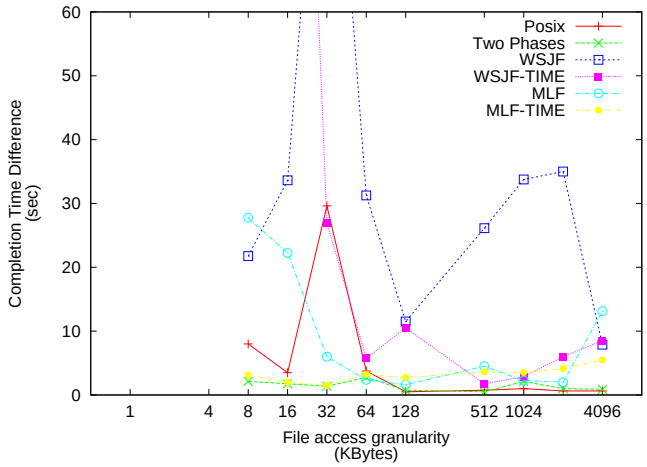
(b) aIOLi server on a dedicated node

Figure 9: 2*2GB file decompositions
Two application (4*2 MPI instances) decompose two files stored on a dedicated NFS server. The two applications are independent and work on various files.

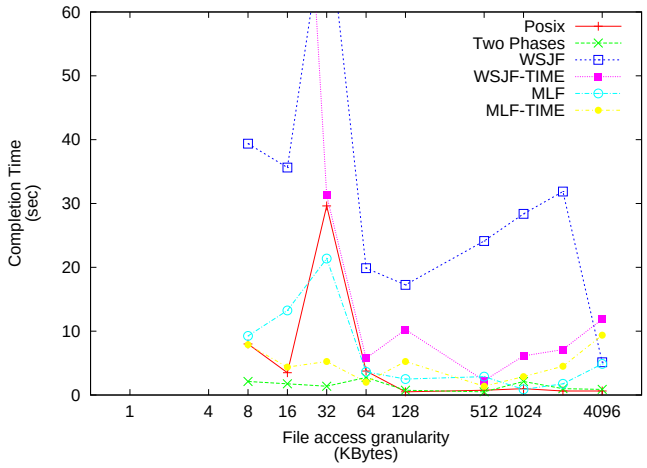
One of the goals of this experiment is to evaluate the performance brought by our approach[16] but also to study the fairness criterion between the applications. The importance of finding an acceptable tradeoff between efficiency (maximization of the bandwidth) and fairness criterion. The figure 10 illustrates this parameter : the difference between completion times of each application is measured. The interest of such a criterion is to observe if between two applications requiring the same amount of data, one is not starved by the other one.

The standard POSIX approach provide significant difference between the completion time of both applications for small accesses : each request is treated in a completely independent manner and is distributed on the 2*2GB data (certain requests are favored in comparison with others). The WSJF approach seems to be affected by the shift phenomenon and gives results which require furthermore analysis. Finally, the MLF algorithm with predicted time provides a very good ratio fairness/performance for access larger than 32KB. For instance, at access granularity of 4 MB, the difference of completion time between the two applications is only about 10 seconds for a profit close to 50% (in comparison with the Posix approach).

[16] Parallel access disovery in multi-application mode.



(a) aIOLi server on NFS



(b) aIOLi server on a dedicated node

Figure 10: 2*2GB file decompositions - Fairness
Fairness criterion, variation between completion times is acceptable for the algorithm MLF compared to its profit. (cf. figure9).

6.5 To the Multi-application environments

This experiment is done to evaluate our approach in a "real" case : various independent applications are executed according to the cluster batch scheduler without regarding if they could be concurrent on the file server. More precisely, 5 MPI applications are executed : the first one is composed of 6 MPI instances distributed on 3 nodes which decompose 1.5GB file, the second one is relied on 4 MPI instances distributed on two nodes to decompose 1GB, the third one, the fourth and the last one are based on 2 MPI instances all distributed on one node and make respectively 0.8GB, 500MB and 200MB file decomposition. The results are presented in figure 11. The benchmark has been executed for Posix, "Two Phases", MLF and MLF with predicted time mode. We can see that the results are quite promising. The MLF approach gives the best performance (even better than the MPI I/O approach). The fairness quality is currently being evaluated and should be available in the final version of the paper. We want to analyze the provided time by our scheduling strategies to the applications according to their required file size.

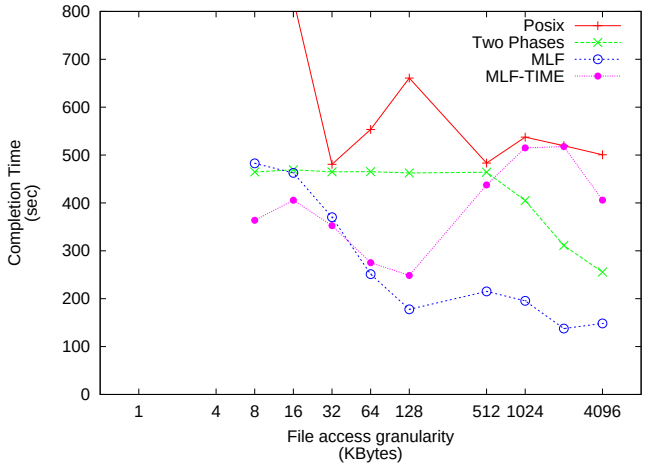


Figure 11: 4GB - 5 file decompositions
Many independent MPI instances decompose several files stored on a dedicated NFS server. The total size of data retrieved from the NFS server is 4GB. aIOLi server is deployed on a dedicated node.

7 Related works

A lot of works have been carried out on the parallel I/O. The presented architecture is similar to the library PANDA, [21], (using of master servers in charge of I/O request management) but it is only specific for input and output of multidimentional arrays. [5] suggests a new approach to handle non contiguous access from individual client. This technique, called "list I/O" uses ideas close to the stream-based I/O developed in PVFS or to the lio_listio call (defined in the POSIX standard): a list of tuples offset/length represents several non contiguous I/O requests. NFS V4, [1], introduces a similar technique called "COMPOUND procedures" to reduce the RPC operations. Besides, many parallel file systems [12, 13, 19, 6, 14, 15, 20], were developed in order to support the concurrent accesses from many clients. These file systems distribute data on many disks and are usually integrated different aggregation techniques to ameliorate the transfer time. They are efficient but usually require specific APIs which, as we mentioned above, imply deep knowledge of their internal mechanisms.

There were many researches about I/O scheduling but none fitted our need. For example, [22, 9] presents many disk scheduling algorithms. In [18], the author presents an approach called "reactive scheduling" allowing to combine different scheduling algorithms within a single system for a correct optimization for a given system workload. [10] proposes many heuristic algorithms for parallel I/O scheduling but they use a centralized batch-oriented scheduling model requiring a large amount of control information which is not always available for many systems. In [8], the authors propose a non-centralized scheduling policy applied for their Clusterfile system, but this policy is based on some specific assumptions so it is not portable.

8 Conclusion

This paper has presented the aIOLi system to optimize I/O requests within a cluster. Principles, constraints as well as selected solutions (with their drawbacks) have been shown. Yet, our approach has the asset to be transparent for the users as it uses the ubiquitous C API (`open/read/write/close`). The experimental results have confirmed the benefits of such an approach (mainly the MLF strategy, which in the multi-application environments,is better than ROMIO and POSIX at all granularities).

Main difficulties consist in the synchronization of I/O requests on the file server and in establishing the multi-criterion scheduling algorithms (performance and fairness). The predictive approach, which, in theory, should reduce the synchronization delays, becomes quite complex in practice. The difficulty of establishing a prediction reliable enough with the accesses of small size is the main limitation. Likewise, further investigations have to be carried out to understand the slight degradation for larger requests. However, we hope to reduce partly this problem by using an improved prediction model which is similar to the model used in the *Network Weather Services* [26] tool to increase the reliability of the predictions.

The detection of access patterns implemented in the first version of aIOLi system becomes more and more complex. Lots of factors, such as process scheduling policy, the network latency, the network load may alter the reception order of I/O requests. Using simple junction coefficient in the WSJF algorithm seems not to be enough and requires a more detailed analysis. At the contrary, the MLF approach has shown that with the accesses bigger than 32K, the ratio fairness/performance is quite promising. We plan to study the constraints implied by a real aggregation model to assess the tradeoff between potential benefits and the costs of the mechanisms (distributed cache).

The evaluation against the fairness criterion for the latest experiments is currently in progress. Meanwhile, work on a finer-grain scheduler is carried out to provide an extra-efficiency when traditional I/O programs (such as unix `cat` command) are launched. In such a case, increasing the wait delay may indeed lead to getting the maximum from the read-ahead mechanisms before switching to another I/O queue.

Last but not least, the point consisting in including to the scheduling algorithms the parallel file system constraints (the current algorithms are applied for centralized data servers such as NFS) has to be tackled. This strategy might also be improved by giving specific hints from the batch scheduler to the aIOLi system in order to provide for instance, different levels of quality of service (best effort, minimal bandwidth...).

References

[1] Network file system (nfs) version 4 protocol, 2003.

[2] G.M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. pages 483–485, 1967.

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

[4] 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.

[5] A. Ching, A. Choudhary, K. Coloma, Wei keng Liao (Northwestern University), R. Ross, and W. Gropp (Argonne National Laboratory). Noncontiguous i/o accesses through mpi-io. May 2003.

[6] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 20, pages 285–308. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[7] J. L. Hennessy and D. A. Patterson. Computer architecture: A quantitative approach, 1996.

[8] Florin Isaila, Guido Malpohl, Vlad Olaru, Gabor Szeder, and Walter Tichy. Integrating collective i/o and cooperative caching into the "clusterfile" parallel file system. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 58–67, New York, NY, USA, 2004. ACM Press.

[9] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. *Appear in the 18th ACM Symposium on Operating Systems Principles*, 2001.

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

[11] A. Lebre and Y. Denneulin. aioli: An input/output library for cluster of smp. Decembre 2004.

[12] W. B. Ligon and R. B. Ross. Implementation and performance of a parallel file system for high performance distributed applications. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 471–480. IEEE Computer Society Press, August 1996.

[13] Pierre Lombard and Yves Denneulin. nfsp : A distributed nfs server for cluster of workstations. In *Proceeding of the 16th international Parallel and Distributed Processing Symposium*, April 2002.

[14] Steven A. Moyer and V. S. Sunderam. PIOUS: a scalable parallel I/O system for distributed computing environments. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.

[15] Nils Nieuwejaar and David Kotz. The Galley parallel file system. *Parallel Computing*, 23(4):447–476, June 1997.

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

[17] M. Raynal. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. on Computer Systems (TOCS)*, 1989.

[18] Robert B. Ross. *Reactive Scheduling For Parallel I/O Systems.* PhD thesis, Clemson University, 2000.

[19] Roger L.Haskin Frank 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.

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

[21] 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.

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

[23] Manish Sharma and John W. Byers. How well does file size predict wide-area transfer time? In *Proceedings of the 2002 Globecom Global Internet Symposium*, Taipei, Taiwan, October 2002.

[24] M. Singhal and N. G. Shivaratri. *Advanced concepts in operating systems.* McGraw-Hill, 1994.

[25] Rajeev Thakur, William Gropp, and Ewing Lusk. Optimizing noncontiguous accesses in MPI-IO. *Parallel Computing*, 28(1):83–105, January 2002.

[26] R. Wolski. Dynamically forecasting network performance using the network weather service. *appeared in Cluster Computing: Networks, Software Tools, and Applications*, Jan. 1998.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399