UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JEAN LUCA BEZ

Evaluating I/O Scheduling Techniques at the Forwarding Layer and Coordinating Data Server Accesses

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Philippe O. A. Navaux

Porto Alegre
December 2016

CIP — CATALOGING-IN-PUBLICATION

Bez, Jean Luca

Evaluating I/O Scheduling Techniques at the Forwarding Layer and Coordinating Data Server Accesses / Jean Luca Bez. – Porto Alegre: PPGC da UFRGS, 2016.

73 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2016. Advisor: Philippe O. A. Navaux.

1. High Performance I/O. 2. Parallel File Systems. 3. Parallel I/O. 4. I/O Forwarding. 5. I/O Scheduling. 6. Access Coordination. I. O. A. Navaux, Philippe. II. Título.

"We keep moving forward, opening new doors, and doing new things, because we're curious and curiosity keeps leading us down new paths."

— Walt Disney

ACKNOWLEDGMENTS

I want to thank my advisor Philippe Navaux for the opportunity to work with him and his research group, for all the guidance and advice during the past two years. I would also like to thank professor Lucas Schnorr who taught me a lot about some important concepts of scientific research and for helping me organize, document and structure all my research projects.

I want to express my sincere gratitude to my friends of the GPPD research group, in special to my collegues of the parallel I/O field, Francieli Boito and Rodrigo Kassick, for introducing and guiding me on my first steps in this amazing research field. I also would like to acknowledge Francieli Boito for all the suggestions, insights, and discussions throughout this period.

A special thank to my family who supported me during this exiting new phase of my life and who encouraged me to persue my dreams. To my friends and all those not mentioned here, that helped me in this journey, a sincere and thankful hug.

I would also like to extend my gratitude to the Federal University of Rio Grande do Sul (UFRGS) and the Institute of Informatics (INF) for the opportunity of conducting my research in this amazing scientific community; and the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) and the HPC for Energy (HPC4e) project for the scholarship.

Experiments presented in this document were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (https://www.grid5000.fr).

ABSTRACT

In High Performance Computing (HPC) environments, scientific applications rely on Parallel File Systems (PFS) to obtain Input/Output (I/O) performance especially when handling large amounts of data. However, I/O is still a bottleneck for an increasing number of applications, due to the historical gap between processing and data access speed. To alleviate the concurrency caused by thousands of nodes accessing a significantly smaller number of PFS servers, intermediate I/O nodes are typically employed between processing nodes and the file system. Each intermediate node forwards requests from multiple clients to the parallel file system, a setup which gives this component the opportunity to perform optimizations like I/O scheduling. The objective of this dissertation is to evaluate different scheduling algorithms, at the I/O forwarding layer, that work to improve concurrent access patterns by aggregating and reordering requests to avoid patterns known to harm performance. We demonstrate that the FIFO (*First In, First Out*), HBRR (*Handle-Based Round-Robin*), TO (*Time Order*), SJF (*Shortest Job First*) and MLF (*Multilevel Feedback*) schedulers are only partially effective because the access pattern is not the main factor that affects performance in the I/O forwarding layer, especially for read requests. A new scheduling algorithm, TWINS, is proposed to coordinate the access of intermediate I/O nodes to the parallel file system data servers. Our approach decreases concurrency at the data servers, a factor previously proven to negatively affect performance. The proposed algorithm is able to improve read performance from shared files by up to 28% over other scheduling algorithms and by up to 50% over not forwarding I/O requests.

Keywords: High Performance I/O. Parallel File Systems. Parallel I/O. I/O Forwarding. I/O Scheduling. Access Coordination.

Avaliação de Técnicas de Escalonamento de E/S na Camada de Encaminhamento e Coordenação de Acessos ao Servidores de Dados

RESUMO

Em ambientes de Computação de Alto Desempenho, as aplicações científicas dependem dos Sistemas de Arquivos Paralelos (SAP) para obter desempenho de Entrada/Saída (E/S), especialmente ao lidar com grandes quantidades de dados. No entanto, E/S ainda é um gargalo para um número crescente de aplicações, devido à diferença histórica entre a velocidade de processamento e de acesso aos dados. Para aliviar a concorrência causada por milhares de nós que acessam um número significativamente menor de servidores SAP, normalmente nós intermediários de E/S são adicionados entre os nós de processamento e o sistema de arquivos. Cada nó intermediário encaminha solicitações de vários clientes para o sistema, uma configuração que dá a este componente a oportunidade de executar otimizações como o escalonamento de requisições de E/S. O objetivo desta dissertação é avaliar diferentes algoritmos de escalonamento, na camada de encaminhamento de E/S, cuja finalidade é melhorar o padrão de acesso das aplicações, agregando e reordenando requisições para evitar padrões que são conhecidos por prejudicar o desempenho. Demonstramos que os escalonadores FIFO (*First In, First Out*), HBRR (*Handle-Based Round-Robin*), TO (*Time Order*), SJF (*Shortest Job First*) e MLF (*Multilevel Feedback*) são apenas parcialmente eficazes porque o padrão de acesso não é o principal fator que afeta o desempenho na camada de encaminhamento de E/S, especialmente para requisições de leitura. Um novo algoritmo de escalonamento chamado TWINS é proposto para coordenar o acesso de nós intermediários de E/S aos servidores de dados do sistema de arquivos paralelo. Nossa abordagem reduz a concorrência nos servidores de dados, um fator previamente demonstrado como reponsável por afetar negativamente o desempenho. O algoritmo proposto é capaz de melhorar o tempo de leitura de arquivos compartilhados em até 28% se comparado a outros algoritmos de escalonamento e em até 50% se comparado a não fazer o encaminhamento de requisições de E/S.

Palavras-chave: E/S de Alto Desempenho, Sistemas de Arquivos Paralelos, E/S Paralela, Encaminhamento de E/S, Escalonamento de E/S, Coordenação de Acessos.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AGIOS	Application-Guided I/O Scheduler
ADIOS	Adaptable I/O System
API	Application Programming Interface
CIOD	Console I/O Daemon
DOE	United States Department of Energy
DVS	Data Virtualization Service
FIFO	First In, First Out
FUSE	Filesystem in Userspace
GPFS	General Parallel File System
HBRR	Handle-Based Round-Robin
HDD	Hard-disk Drive
HDF5	Hierarchical Data Format, Version 5
HPC	High Performance Computing
I/O	Input/Output
IBM	International Business Machines
IOD	I/O Daemon
IOFSL	I/O Forwarding Scalability Layer
ION	I/O Node
MLF	Multilevel Feedback
MPI	Message Passing Interface
NetCDF	Network Common Data Form
PFS	Parallel File System
POSIX	Portable Operating System Interface
PVFS	Parallel Virtual File System

RAID	Redundant Array of Independent Disks
SJF	Shortest Job First
SSD	Solid State Drive
SW	Server Window Scheduler
TO	Time Order
TO-agg	Time Order with Aggregation
TWINS	Time WINdows Scheduler
UFRGS	Federal University of Rio Grande do Sul
UFS	Unix File System
ZOID	ZeptoOS I/O Daemon

CONTENTS

1 INTRODUCTION

Scientific applications such as climate, flow, and seismic simulations fill the High-Performance Computing (HPC) field with rising performance requirements in order to provide knowledge and help understand complex phenomena. These requirements justify the appearance of ever increasing large scale parallel platforms. For instance, the Sunway TaihuLight supercomputer (NSCCWX, 2016) has $40,960$ nodes to achieve 93 petaflops and the the Aurora supercomputer (ARGONNE, 2016), expected for the next few years, will have over $50,000$ processing nodes to achieve 180 petaflops.

"The Opportunities and Challenges of Exascale Computing" report presented by the U.S. Department of Energy (DOE, 2010) stated that the Exascale problem is more than just a matter of scale. Applications' behavior and performance will be determined by a complex interplay of the program code, processor, memory, interconnection network, and I/O operation. Therefore, achieving good performance on scale requires an optimized orchestration of those components and a whole system view in order to understand root causes of inefficiencies. Since I/O is a bottleneck for an increasing number of applications, due to the historical gap between processing and data accesses speeds, it has the potential of critically impacting applications' performance on the next generation of supercomputers.

To alleviate this existing imbalance and to reduce concurrency, a technique known as I/O forwarding can be applied. This technique introduces a new layer in the I/O software stack to decrease contention in the access to a file system. Thus, instead of having thousands of machines accessing the servers at the same time, only a few hundreds of intermediate nodes will be directly interacting with the storage servers. This additional layer presents great potential to apply optimization on I/O requests. These optimizations include, but are not limited to, requests reordering, aggregation and compression.

The I/O scheduling optimization technique has already been successfully applied to the forwarding layer (VISHWANATH et al., 2010; OHTA et al., 2010) to adjust the applications' access patterns. Nonetheless, in this work, we evaluate a distinct set of schedulers including algorithms that were proven to bring performance improvements when applied directly on the data servers, now employed at the forwarding nodes. Furthermore, we demonstrate that both sets are only partially effective because the access pattern is not the main factor that influences the performance of requests through I/O nodes. Additional details, such as concurrency and contention when accessing the data servers, must be taken into account in this scenario.

Therefore, we propose two new scheduling algorithms for the I/O nodes which work to decrease contention in the access to the parallel file system data servers. Our algorithm uses time windows and coordinates accesses from intermediate nodes so that at each time window they focus on one of the servers. As far as we know, this is the first work to propose a scheduling technique such as this to the forwarding layer.

Based on an extensive set of experiments, we detect I/O performance improvements with our algorithm over state-of-the-art algorithms. Moreover, our solution provides gains for 1D strided access pattern, comparable to the use of collective I/O operations, while being completely transparent to applications and I/O library-independent.

1.1 Contributions

The main objective of our research is to **evaluate I/O scheduling in the forwarding layer**, detecting algorithms that help improve performance and those whose overhead prevent them from being used in this context.

Considering these goals, our main contributions are the following:

- We **evaluate** a total of five **scheduling algorithms** in the **I/O forwarding layer**. We demonstrate that existing schedulers, that provide improvements when used in the file system servers, do not provide similar results when applied to the forwarding layer. Furthermore, in most of the evaluated scenarios, their usage does not harm performance.
- We propose two **new scheduling algorithms** to coordinate accesses to the parallel file system servers, aiming at **reducing contention** and increasing performance. We conducted an extensive evaluation of these new schedulers considering different access patterns and forwarding scenarios.

1.2 Document Organization

The document is organized as follows. Chapter 2 presents a background on the topics of this dissertation and discusses related work in parallel I/O optimizations. Chapter 3 details the forwarding framework and scheduling library employed in our experiments, alongside some insights on their integration. Additionally, this chapter describes and presents the first results of our evaluation, considering the available scheduling algorithms. Our new I/O schedulers are detailed in Chapter 4. In Chapter 5, we evaluate our scheduler on the same scenarios as the previous experiments, comparing the results with our previous investigation. Finally, Chapter 6 draws conclusions based on our findings and presents some insights on future work.

2 BACKGROUND AND RELATED WORK

Some important concepts that serve as a base for this dissertation are explained in the following sections. A brief overview of the I/O stack for High Performance Computing is presented, alongside known I/O optimizations such as request reordering and aggregation. Furthermore, this chapter also details related work in optimizing parallel I/O, comparing those efforts to our own approach.

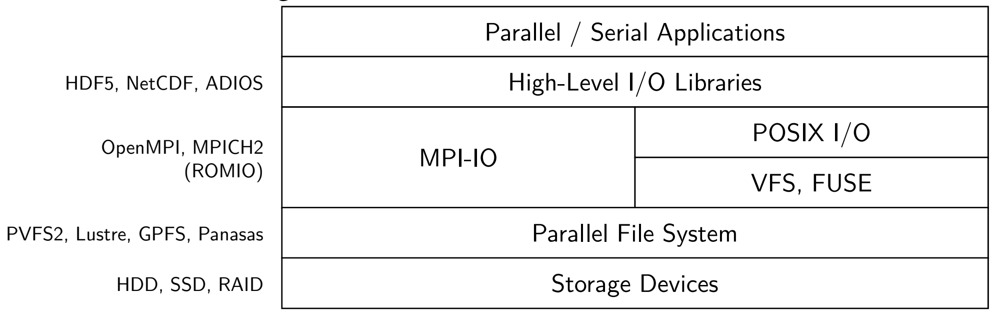
2.1 Parallel I/O for High Performance Computing

Scientific applications such as climate, flow, and seismic simulations fill the High-Performance Computing (HPC) field with several performance requirements in order to provide understanding of complex phenomena. These performance requirements also include the ability to perform faster I/O operations at an increasing rate to collect as much information as possible.

When such applications execute in a cluster environment or on a supercomputer, their computation is divided among processes and those are distributed over selected nodes to perform the computation. However, those applications generally have to read or write data from shared files. In the case of supercomputers, the nodes responsible for executing the application often have a simplified kernel to avoid possible interferences, and they do not have local storage devices. For such scenarios, parallel file systems are the alternative because they provide a shared storage infrastructure so applications can access remote files as if they were stored on a local file system.

Because of the historical gap between processing and data access speeds, parallel I/O is a limiting factor for many applications. Moreover, if all processing nodes were to concurrently access the shared file system servers, contention would impair performance.

Figure 2.1: Parallel I/O software stack.



Source: Author, inspired by Ohta et al. (2010)

In order to support I/O from serial or parallel scientific applications, current supercomputers provide a multilayer software environment, as depicted by Figure 2.1. High-level I/O libraries such as HDF5 (The HDF Group, 1997-2016), NetCDF (LEE; YANG; AYDT, 2008) and ADIOS (LIU et al., 2014), provide storage abstraction and data portability for the applications. Those libraries execute on compute nodes, mapping application abstractions into files, and encoding data in portable formats. Interfaces such as MPI-IO (CORBETT et al., 1995) and POSIX (FUSE) are employed to interact with the parallel file systems servers. These, in their turn, provide a logical file system abstraction over many storage devices such as HDDs, SSDs or RAID.

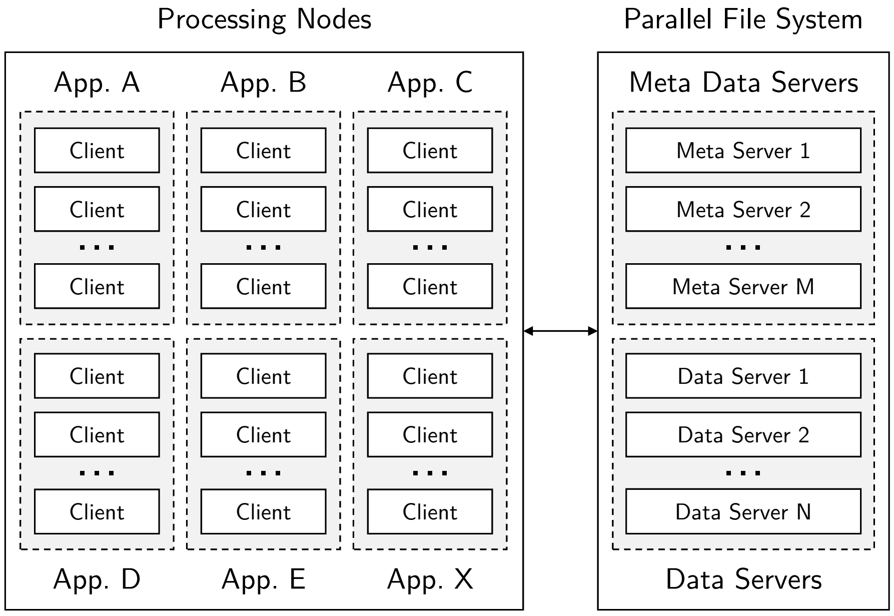
2.1.1 Parallel File Systems

Large-scale systems, such as cluster and supercomputers, rely on parallel file systems (PFS) to provide a persistent shared storage infrastructure. These systems provide a shared namespace, so applications can access remote files as if they were stored on their local file system. Furthermore, to achieve high performance they harness parallelism by distributing data across multiple storage nodes.

The parallel file system's servers are divided into two groups: the *data servers* and the *metadata servers*. The former are responsible for storing data, while the latter are responsible for the metadata. Metadata is information about the stored data such as its size, permissions, and its distribution among the data servers. Figure 2.2 depicts a common parallel file system deployment, with separated metadata servers. However, in some systems, the data and metadata servers roles can be played by the same node. Additionally, parallel applications may span over several compute nodes or clients which will generate concurrency when accessing the PFS's servers.

All basic file system operations involve metadata access. For instance, when a client wants to read or write a file it must first obtain the layout information and permissions from the metadata server. Therefore the scalability of those accesses has a direct impact on the overall system (REN et al., 2014). An alternative to improve performance is to allow clients to cache metadata information, however in this scenario a cache coherence policy must be in place. Some parallel file systems such as PVFS2 (LATHAM et al., 2004) distribute metadata among multiple servers, while others such as Lustre (SUN, 2007) maintain a single centralized metadata storage. Centralizing metadata operations may became a bottleneck for applications that work with a large number of small files.

Figure 2.2: Major components of a parallel file system.



Source: Author

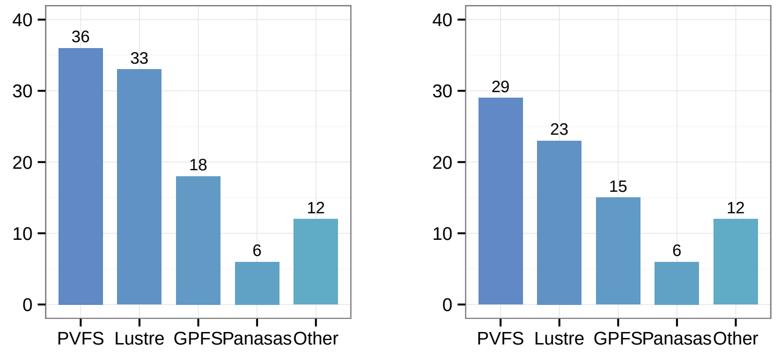
Each file is broken down into small sized chunks and distributed among the data servers following a distribution algorithm (STENDER et al., 2008). This technique, called *striping*, allows PFS to harness parallelism when reading or writing data. For instance, PVFS2 and Lustre file systems use a default round-robin policy between data servers. Additionally, the stripe size used in each PFS generally depends on the target applications.

The major parallel file systems in use are Lustre, IBM's General Parallel File System (GPFS) (SCHMUCK; HASKIN, 2002), Panasas (WELCH et al., 2008), and the Parallel Virtual File System (PVFS) or its new branch, OrangeFS (DELL, 2012). From the ten most powerful supercomputers in the world according to the November 2016 edition of the Top500 list[1], five use Lustre, two use their own solutions, two are based on Lustre, and one uses GPFS.

As an overview of the most used parallel file systems in scientific research, we analyzed several papers in a pre-defined five-year window for a survey on parallel I/O. We have made a selection of widely known, leading quality conferences and journals. This window covers publications between 2010 and 2014. We went through all proceedings and issues inside the time window ($5,159$ publications) to identify relevant work by looking at title and abstract. During this process, 120 papers were pre-selected for further analysis (2.3%). After reading the articles and answering a set of questions some papers were excluded because they were not relevant for the survey. In the end, 86 articles remained (1.7%).

[1]https://www.top500.org/lists/2016/11/

Figure 2.3: Most used parallel file systems in parallel I/O research.



(a) Parallel I/O research (b) I/O optimizations

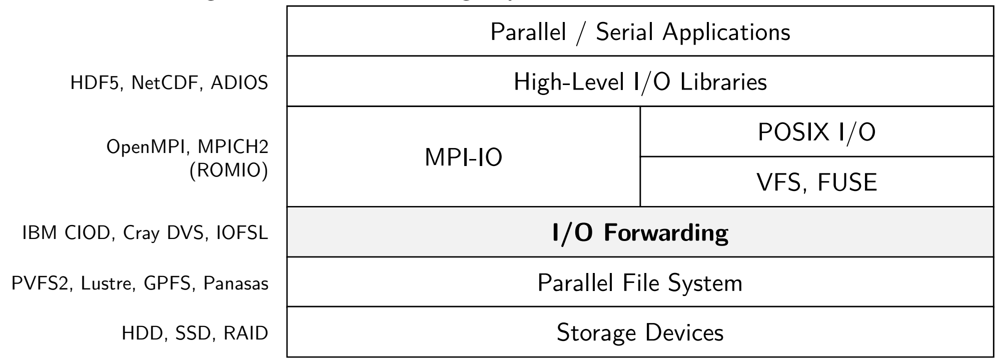
Source: Author

We present those results in Figure 2.3 grouped by PFS and research purpose. We can see that PVFS is one of the most used systems for parallel I/O research (Figure 2.3(a)). Furthermore, studies that focused on proposing I/O optimization techniques (Figure 2.3(b)) were also carried out in PVFS. Therefore, for the purposes of this research, we also employ PVFS. Since we focus on the I/O nodes, the choice of PFS does not make our solution less generic.

2.1.2 The Forwarding Layer

The layered construction presented by Figure 2.1 can potentially accelerate parallel application I/O for smaller-scale systems (ALI et al., 2009). However, as the number of processing servers starts to grow, so does the existing bottleneck on the parallel file system servers. A new layer was included in the I/O stack to alleviate the contention by grouping accesses, and thus reducing the number of clients that directly interact with the servers, as depicted by Figure 2.4.

Figure 2.4: Forwarding layer in the HPC I/O stack.

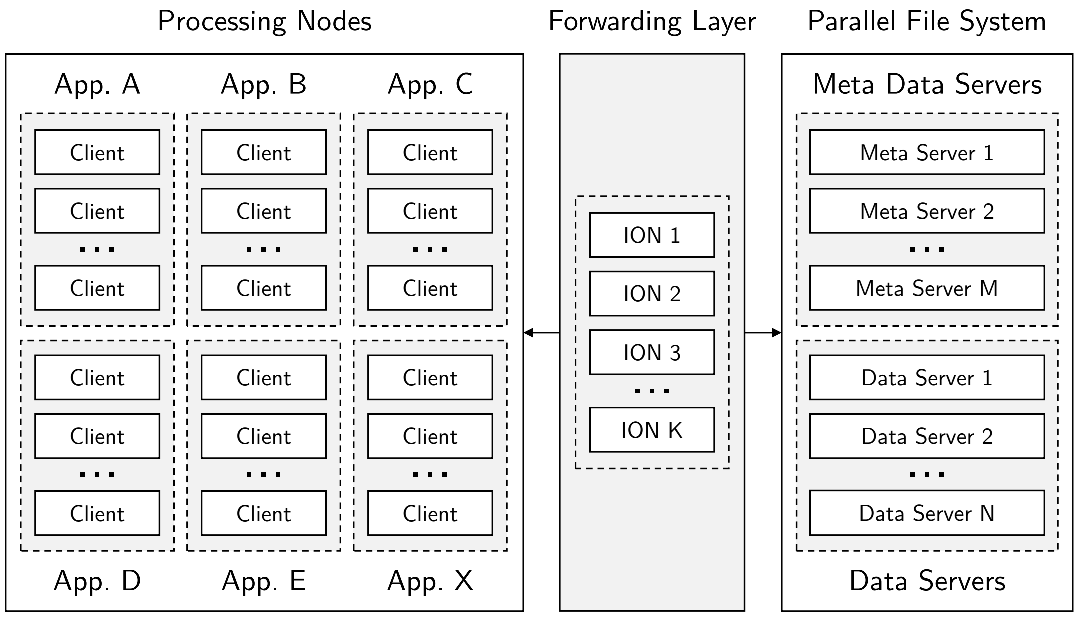


Source: Author, inspired by Ohta et al. (2010)

With the I/O forwarding layer, all requests are forwarded to dedicated processing elements, known as I/O nodes. Typically, the number of I/O nodes is larger than the number of file system servers, and smaller than the number of processing nodes. In this scenario, the processing nodes may be powered with only a very simplified local I/O stack in order to avoid its interference on performance, also known as operating system "noise" (VISHWANATH et al., 2010).

When an I/O node (ION) receives requests, it redirects them to the back-end parallel file system, as depicted by Figure 2.5. This strategy reduces the number of clients concurrently accessing the file system and can potentially reduce the file system traffic by aggregating and reordering I/O requests (OHTA et al., 2010).

Figure 2.5: I/O forwarding scheme on a large-scale cluster or supercomputer.



Source: Author

By interposing this layer above the file system but below the rest of the I/O software stack, as depicted by Figure 2.4, the I/O forwarding framework provides a compelling point for optimizations (ALI et al., 2009). The main reason for this is that this layer is transparent to applications and high-level I/O libraries and all optimizations performed at the forwarding level are generally not file system dependent. Existing forwarding alternatives include IBM CIOD (ALMÁSI et al., 2003), Cray DVS (SUGIYAMA; WALLACE, 2008) and the open-source IOFSL (ALI et al., 2009).

Considerable research (VISHWANATH et al., 2010; OHTA et al., 2010; VISHWANATH et al., 2011; ISAILA et al., 2011) has been focused on improving the I/O forwarding layer performance. Some of them (VISHWANATH et al., 2010; VISHWANATH et al., 2011; ISAILA et al., 2011) studied the I/O subsystem of an IBM Blue Gene/P supercomputer. In this architecture, the data staging mechanism initially applied multiple threads per I/O node (one per processing node), without any coordination among them. Vishwanath et al. (2010) identified some contention-related bottlenecks associated with this design. They improved performance by allowing asynchronous operations in the I/O nodes and by including a simple FIFO scheduler to coordinate accesses from multiple threads. This scheduler alone provides improvements of up to 38%. They also optimized data movement between layers through a topology-aware approach. Isaila et al. (2011) proposed a two-level pre-fetching scheme for this architecture.

Similarly Ohta et al. (2010) improve performance of the IOFSL framework by using I/O scheduling. They implement two algorithms: a simple FIFO and a *quantum* based algorithm called *Handle-Based Round-Robin* (HBRR). The latter is based on an algorithm successfully applied to parallel file systems' data servers (LEBRE et al., 2006; QIAN et al., 2009; BOITO et al., 2013; BOITO et al., 2015), that aims at reordering and aggregating requests to improve the performance of the applications by modifying their access pattern.

The I/O forwarding layer is present in the HPC I/O stack of several supercomputers. For instance, the I/O forwarding technique was applied to build the storage infrastructure of Tianhe-2 supercomputer (XU et al., 2014), currently the second in the Top 500 list for November 2016 [2]. In this scenario, as the computing nodes do not have a local I/O stack, all I/O operations are transferred to the intermediate I/O nodes. Other supercomputers, such as the Titan (#3) (ZIMMER; GUPTA; LARREA, 2016), the Sequoia (#4) (PRABHAT; KOZIOL, 2014), the Cori (#6) (DECLERCK et al., 2016), the K Computer (#7) (MIYAZAKI et al., 2012) and the Mira (#9) (PRABHAT; KOZIOL, 2014) also have in their infrastructure some nodes dedicated to forward I/O requests and reduce contention. This highlights the fact that the I/O forwarding technique is widely adopted. Furthermore, several optimization strategies are applied to this layer for being transparent to applications and file system-independent.

2.2 I/O Optimizations

It is known that numerous factors may interfere with applications' I/O performance, especially at large scale. Performance degradation can occur because of network problems, software bugs, slow disk or contention when accessing the shared storage sys-

[2]TOP 10 Sites for November 2016 - https://www.top500.org/lists/2016/11/

tem (LARREA et al., 2015). A lot of research effort is put into optimizing, at different layers of the I/O stack, how the applications perform their I/O. Modifying the file system servers or clients, the applications, or using libraries and APIs are distinct ways of achieving the same goal: adjust the way applications issue their I/O requests, avoiding situations that are known to degrade performance.

The following sections detail concepts related to I/O optimizations. The application's access pattern is described in Section 2.2.1 and some optimizations techniques applied on these patterns are described in Sections 2.2.2 and 2.2.3.

2.2.1 Access Patterns

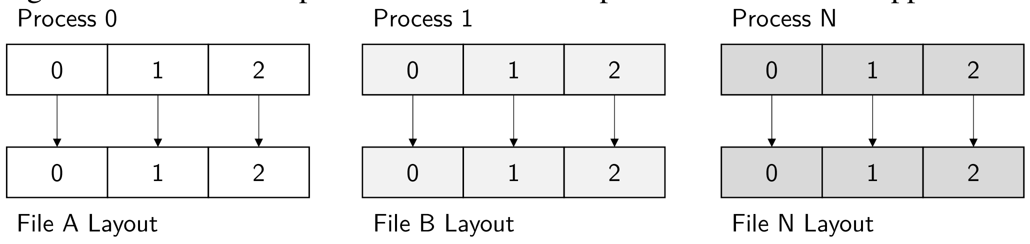
Applications issue their I/O requests (read/write) to the parallel file system servers in different ways, depending on how they were designed and coded. Several characteristics such as how many requests are issued, the requests' sizes and their spatial location in the file compose what we call the application's *access pattern*. This pattern has a direct impact on performance, hence a lot of research effort is put into optimizing data access (LOFSTEAD et al., 2011; HE et al., 2013; YIN et al., 2013; KUO et al., 2014).

We can classify the access pattern in *global* or *local*. The global pattern describes the behavior of the entire application, whereas the local pattern does it in the context of a process or task (YIN et al., 2013). The local access pattern information is usually employed to identify and apply optimizations on the client side, while the global access pattern is more suitable in the context of the forwarding layer or file system servers since it has an overview of the application's data accesses.

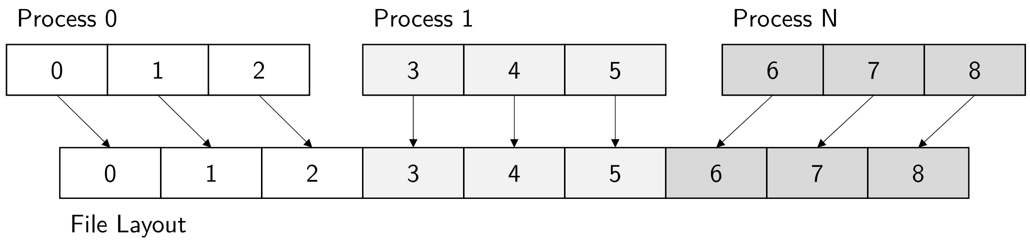
Despite the fact that there is not a globally accepted convention to describe these patterns, some factors or parameters are examined by several researchers of the parallel I/O field. In this study, we consider the following key aspects to describe the application's data access pattern: the number of files, the spatial locality within the file, the size of accesses and the I/O operation.

Regarding the number of files, we consider two common scenarios that portray how most of the HPC scientific applications perform I/O. In the first one, each process of an application that reads or writes data executes its operations in its individual file (*file-per-process*) as depicted by Figure 2.6(a). On the second scenario, all the process share a common file (*shared file*). Furthermore, the spatial locality parameter describes if the access to a shared file is sequential, i.e. each process accesses contiguous chunks of the

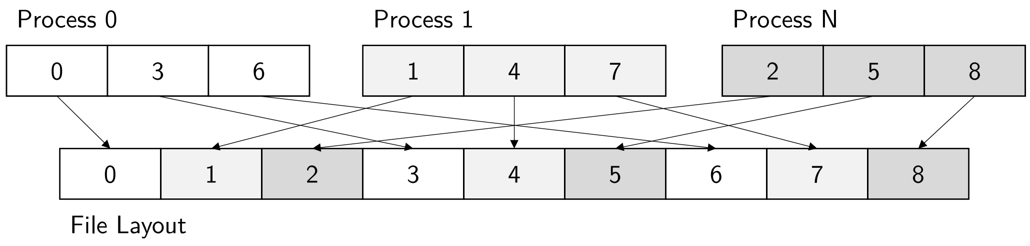
Figure 2.6: Different representative I/O access patterns for scientific applications.



(a) File-per-process



(b) Shared file with contiguous access



(c) Shared file with 1D-strided access

Source: Author

file (Figure 2.6(b)) or 1D-strided, i.e. each process accesses portions with a fixed-size gap between them (Figure 2.6(c)).

When each application accesses its own file, the existence of a shared file system is not always required. However, even when this pattern is present, often the files are accessed by other processes or another application for post-processing or visualization, requiring the existence of a common shared storage. Additionally, when the processing nodes do not have storage devices attached to them, the PFS is a commonly used alternative, independently of the type of access.

The request size also has a profound impact on the I/O performance because of the storage devices' sensitivity to access sizes and network cost transmissions (BOITO et al., 2015). For instance, small requests suffer more due to the overhead imposed by the network latency, which dominates the cost of processing the request.

It is valuable and feasible to make use of application's characteristic information such as its access pattern to apply I/O optimizations (YIN et al., 2013). The next section describes why aggregating and reordering requests, thus modifying the access pattern, is essential to improve the I/O performance.

2.2.2 Request Aggregation and Reordering

The performance of contiguous data access is normally higher than that of non-contiguous ones (YIN et al., 2013). This holds true for both hard disk drives (HDD) and solid state disks (SSD). Furthermore, (ZIMMER; GUPTA; LARREA, 2016) points out that small and random I/O request patterns negatively impact the file system performance. Thus, applications benefit from continguously accessing a file and issuing fewer requests to the file system, reducing the high I/O latency.

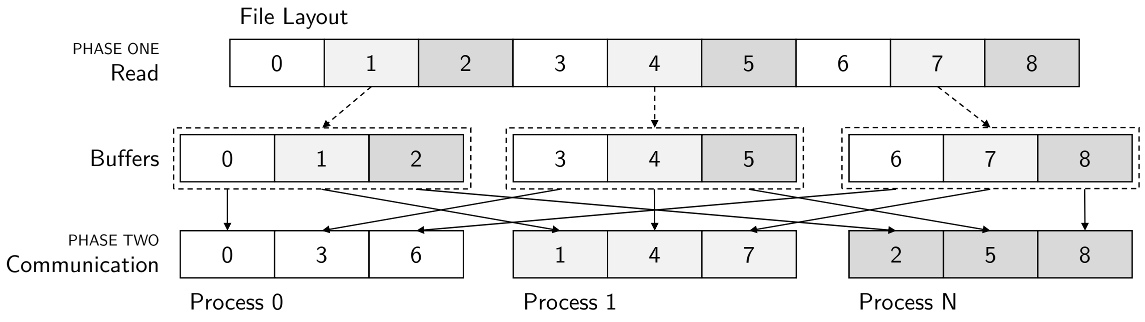
A technique called *data sieving* (THAKUR; GROPP; LUSK, 2002) attempts to optimize read requests by issuing larger requests than the ones described by the user. So, instead of making several non-contiguous access, a single call could be made that enclosed all the offsets required by the application. This technique, however, is not advantageous when the gaps between requests outweigh the cost of reading and transferring the extra data.

Collective I/O is another optimization strategy to improve read and write requests. This technique can be employed at the disk level, server level or client level (THAKUR; GROPP; LUSK, 2002). The MPI-IO interface allows users to collectively specify the I/O requests of a group of processes, thereby providing additional access information and a greater scope for optimization. Collective calls force all processes in an MPI communicator to issue their I/O operation simultaneously and to wait for each other upon completion. Therefore, requests from each process are combined and merged whenever possible to optimize data access. This allows the application to perform large, contiguous accesses, even though the application's requests may represent a non-contiguous one.

To implement collective operations, MPI-IO uses a technique called *two-phase I/O* (ROSARIO; BORDAWEKAR; CHOUDHARY, 1993). In the first phase, processes access data by making a single, large contiguous access. In the second phase, processes distribute the data among themselves according to the desired offsets, as depicted by Figure 2.7. This technique usually provides performance improvements because the I/O cost is significantly reduced by issuing fewer, larger and more contiguous requests, even though an additional communication is required.

It is important to notice that the optimizations cited so far, i.e. data sieving and collective I/O, typically require the applications to modify its source code. The next section describes additional efforts in providing better access patterns to the parallel file systems without further alterations to the applications.

Figure 2.7: Illustration of the two phases of a collective I/O operation in MPI.



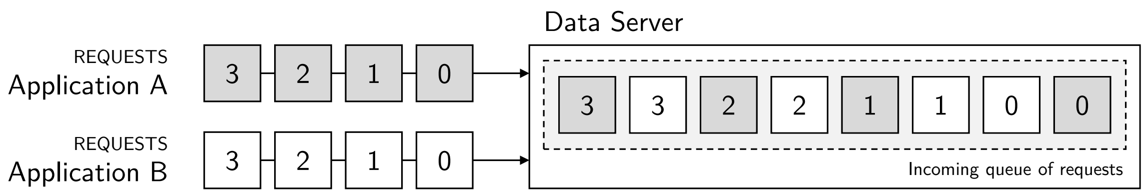
Source: Author

2.2.3 Request Scheduling

Applications concurrently running on large scale clusters or supercomputers have to perform their I/O operations to a shared file system, as stated in Section 2.1.1. However, as one might expect, this concurrency is most likely to impair performance. Applications may use high-level libraries as an attempt to improve their local access pattern. Nevertheless, interferences generated by multiple applications accessing the shared storage infrastructure might break or compromise the efficiency of the optimizations performed on the client side.

In these scenarios, the I/O scheduling technique is applied to improve access to the file systems data servers by organizing and reordering requests, taking into account multiple competing applications. For instance, consider two applications that were locally optimized by a library to issue their I/O requests contiguously. When those requests reach the forwarding layer or the data servers, they may be interleaved, affecting each other and possibly reducing the performance when compared to processing all the requests of the application if it were to execute by itself. Furthermore, this may also happen in the context of a single application. For example, if distinct processes were to contiguously access a shared file, the file system's data servers will observe a non-contiguous access pattern. This phenomenon, illustrated in Figure 2.8, is called *interference* and it is the cause of many performance losses in these shared environments.

Figure 2.8: Interference on the access pattern of concurrently executing applications.



Source: Author

The role of the I/O scheduler is then to reorder, aggregate and determine the best time to process each request. These scheduling techniques, applied at some layer of the I/O stack (clients, I/O nodes or servers), decide *where* and *when* requests must be served. Different schedulers can be found in the literature that range from a simple *First-Come, First-Served* to more complex ones that involve coordination or access pattern adaptation.

Parallel file systems stripe data across data servers to explore parallelism. This action, although effective in serving asynchronous requests, can break individual program's spatial locality. This holds true especially for synchronous requests of multiple concurrent applications. Based on the principle that applications usually rely on strong spatial locality to ensure high I/O performance, Zhang, Davis and Jiang (2010) propose a scheme named IOrchestrator. Their proposal coordinates request scheduling across data servers by using time slices, based on the access patterns. Thus it can exploit spatial locality by dedicating the service to one program at a time. Towards a similar goal Song et al. (2011) proposed a scheduling algorithm for PFS servers. A window-wide coordination concept was employed to make all data servers focus on serving requests from only one application at a time.

The performance of collective I/O operations could be degraded in today's HPC systems due to the increasing shuffle cost caused by highly concurrent data accesses. To address this issue Liu, Chen and Zhuang (2013) propose a hierarchical I/O scheduling algorithm. They argue that the non-contiguous access pattern of many scientific applications results in a large number of I/O requests, which can seriously impair the performance. The usage of two-phase collective I/O operations is a commonly employed alternative but it also implies in increasing shuffle cost (both inter and intra-node) as the scale and concurrency increases. Hence, they implement a scheduler that considers an acceptable delay time to minimize the shuffle cost.

Different I/O scheduling algorithms were analyzed by Boito et al. (2015) at the parallel file system's data servers layer. These algorithms, selected and adapted from state of the art, decide the order in which requests to each data server must be processed. They do not focus on cross-application interference *per se*, but on adjusting access patterns to obtain the best performance of the underlying I/O system. In many cases, this means generating offset ordered requests or aggregating a large number of small requests into a smaller number of larger requests.

Although many schedulers were created for the data servers, only a few were designed or tested in the forwarding layer. Applying such technique in this layer of the

I/O stack has the benefits of being able to work with the global access pattern of the applications and even coordinate access between concurrent ones. Based on this principle, this work aims at evaluating and analyzing the behavior of selected schedulers that were proven to show performance improvements in the parallel file system data servers but now applied to the forwarding nodes. We hope that by working on a layer above the file system, we could improve the overall I/O performance of the applications. Moreover, applying schedulers in this layer of the I/O stack is complementary to using them in other levels of the stack, i.e. this technique could be simultaneously applied in the clients, in the I/O nodes, and in the parallel file system data servers.

2.3 Summary

Parallel I/O is a limiting factor for many applications because of the historical gap between processing and data access speeds. Additionally, at scale, I/O performance degradation can occur because of network errors, software bugs, slow disk or contention when accessing the shared storage system. As an attempt to improve this unbalanced environment several techniques were proposed during the years. The I/O stack of large scale clusters and supercomputers was expanded, including a forwarding layer to reduce contention. Furthermore, optimizations on the clients, I/O nodes and servers were proposed to adjust the way applications issue I/O requests, avoiding situations that are known to degrade performance.

By studying related work, we observed that previous research efforts aimed at improving the application's access pattern. This was done by using collective I/O operations or high-level libraries at the client side, or requests scheduling and aggregation at the parallel file system servers. Only a few work focused on the forwarding layer, present in most of the today's supercomputers. The research focused on the I/O nodes, still aims at transparently improving the I/O performance of parallel applications. In this research, we also tackle this problem by evaluating different scheduling algorithms in the context of the forwarding nodes. We selected known schedulers, proven to improve performance when employed by the parallel file system servers, to understand and measure their benefits and drawbacks in the forwarding layer. Therefore, we focus our work on investigating this and on proposing new schedulers for this layer.

3 EVALUATING SCHEDULING IN THE I/O NODES

In this chapter, we provide an investigation of five existing schedulers, covering read and write requests for the three access patterns detailed in Section 2.2.1. Furthermore, we introduce the tools we have selected to help in our analysis of the benefits and drawbacks of distinct scheduling algorithms in the forwarding layer.

3.1 I/O Forwarding Software Layer

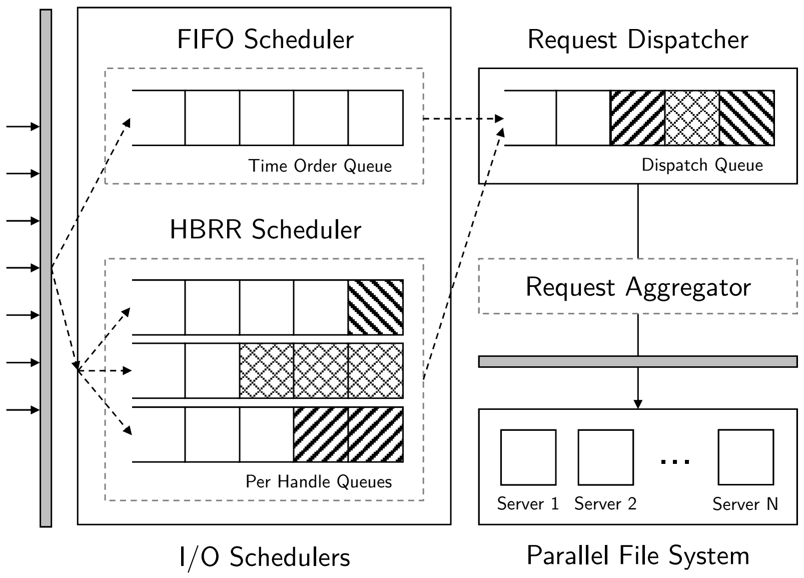
The IOFSL framework (ALI et al., 2009) implements the I/O forwarding technique as an attempt to bridge the increasing performance scalability gap between computing and I/O components (LIU et al., 2013). IOFSL ships I/O calls from the applications, running on computing nodes, to dedicated I/O nodes. The latter will then transparently perform operations on behalf of the computing nodes.

This framework uses the stateless ZOIDFS I/O protocol and API from the ZOID forwarding infrastructure (ISKRA et al., 2008), and the Buffered Message Interface (BMI) network abstraction layer for high-performance parallel I/O. BMI provides request forwarding over multiple parallel file systems (PVFS, Lustre, UFS, and PanFS) and interconnection networks (TCP/IP, InfiniBand, and Myrinet). The framework's software stack consists of two main components: a ZOIDFS client library running on the computing nodes and I/O forwarding daemon (IOD) running on I/O nodes. The client library forwards I/O requests from the compute node kernel to the IOD which performs I/O on behalf of the compute nodes.

In the I/O nodes, multiple threads are created to process the client's requests. The request scheduler component coordinates these threads' accesses. It offers two options of scheduling algorithms – FIFO and HBRR – to fill a dispatch queue and thus decide the order requests must be processed. FIFO is a simple time order algorithm, and HBRR stands for *Handle-Based Round-Robin*. HBRR employs multiple queues, one per handle, where contiguous requests are aggregated whenever possible. From each queue, a maximum number of requests (defined by the *quantum* parameter) may be served before moving to the next queue (OHTA et al., 2010).

The flow of I/O requests through the IOFSL I/O node daemon is illustrated by Figure 3.1. After going through one of the scheduling algorithms, requests are stored in the dispatch queue. From the dispatch queue, requests to the same file and of the same

Figure 3.1: Flow of requests through the IOFSL I/O node daemon.



Source: Author

type (read or write) are aggregated before being forwarded to the file system. Although all data and metadata operations go through the IOFSL nodes, only read and write operations go through the request scheduler component and are affected by scheduling algorithms.

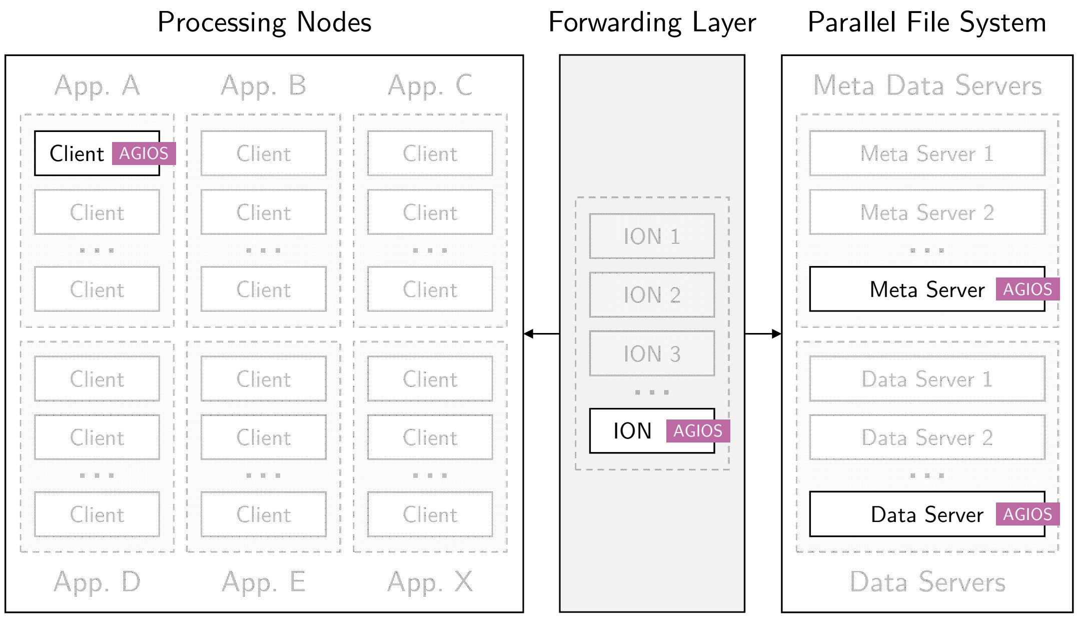
For the experimental purposes of this work, we have selected the IOFSL framework because besides being the only open-source alternative, it was already tested on small and medium scale clusters and in production in a supercomputer. Additionally, we could build on previous contributions and effectively compare our new scheduler, described in Chapter 4, with the state of the art. Section 3.4.3 will discuss the performance obtained by IOFSL with the FIFO and HBRR algorithms.

3.2 AGIOS Scheduling Library

Application-Guided I/O Scheduler (AGIOS) is a scheduling library (BOITO et al., 2015) developed to be used by any I/O service that treats requests at file level, such as parallel file systems or intermediates I/O nodes. The library itself is generic and can be employed at four distinct and independent locations, as depicted by Figure 3.2. These locations include the clients (left), the intermediary nodes from an I/O forwarding scheme (middle), the parallel file system servers for metadata (top right) or data (bottom right).

Since AGIOS does not make global scheduling decisions, no overhead is expected to come from that. Furthermore, the efficiency of the scheduling library is also dependent on its placement in the I/O stack. For instance, each I/O node or data server could have

Figure 3.2: Four possible locations to use the AGIOS scheduling library.



Source: Author

its local scheduler managed by AGIOS and the decisions would be local concerning the requests that arrive at that server. Therefore, the information about the access pattern the scheduler has to work with is bounded by the environment deployment. For instance, if AGIOS is placed on the clients, it would work only on scheduling requests from the processes of that host. If placed on the forwarding nodes, it could work with a global access pattern of a single application or with multiple applications (depending on the forwarding layer deployment). Finally, if placed on the data servers, AGIOS would be able to schedule the request to that specific server, without knowledge of other requests or servers.

The AGIOS scheduling library implements five distinct scheduling algorithms:

- aIOLi (LEBRE et al., 2006);
- *Multilevel Feedback* (MLF) (BOITO et al., 2015);
- *Shortest Job First* (SJF);
- *Time-Order* (TO);
- *Time-Order with Aggregation* (TO-agg).

These algorithms, alongside their advantages, drawbacks and costs are further detailed in the following section.

3.2.1 Schedulers

The aIOLi scheduler (LEBRE et al., 2006) is a quantum-based scheduler that works to aggregate requests into larger ones. Once requests arrive at the scheduler, they are inserted into the proper queue, based on the file. There are two queues per file to separately store read and write requests. The cost for including requests is $O(M + N_{queue})$, where M is the number of files and N_{queue} is the number of requests in the largest queue. To process the requests, each queue is iterated in offset order, aggregating contiguous requests. A *First-Come*, *First-Served* criteria is used to select between distinct queues. Additional waiting time may be introduced. However, while waiting, the scheduler is able to aggregate requests in other queues. To select a request, the scheduler must go through all the queues, thus the cost for selecting is $O(M \times N)$. From the algorithms implemented in AGIOS, aIOLi is the only one that behaves synchronously, i.e. the next request will only be selected to be served after the current one is processed.

The MLF was developed based on aIOLi. Thus, its insertion cost is the same as aIOLi's. However, MLF is capable of providing more throughput because there is no synchronization between the user and the library after processing requests. Consequently, it may not have the same aggregation opportunities as aIOLi. Additionally, not all queues need to be considered before selecting a request, thus the incoming request order may not be respected, which makes the selection $O(M + N)$.

The SJF scheduling algorithm consists of two separate queues per file: one for read and one for write requests. In each queue, requests are considered in offset order, examining possible aggregations of contiguous requests. The first request of the smallest queue (based on the sum of all its request' sizes) is selected first. Therefore, the cost for including requests is the same as aIOLi's and for selecting requests is $O(M)$.

The TO algorithm works with a unique single queue for both reads and writes. Requests are processed in the *First-Come*, *First-Served* order. Furthermore, TO does not perform aggregations by default. Therefore no scheduling overhead is expected, so the cost for inserting new requests is constant. TO-agg, on the other hand, works just like TO but it performs aggregations of contiguous requests. Because of that, inserting a request implies in going through the entire queue. Thus, the cost is $O(N)$, where N is the number of requests in the queue. The cost for selecting a requests is $O(1)$.

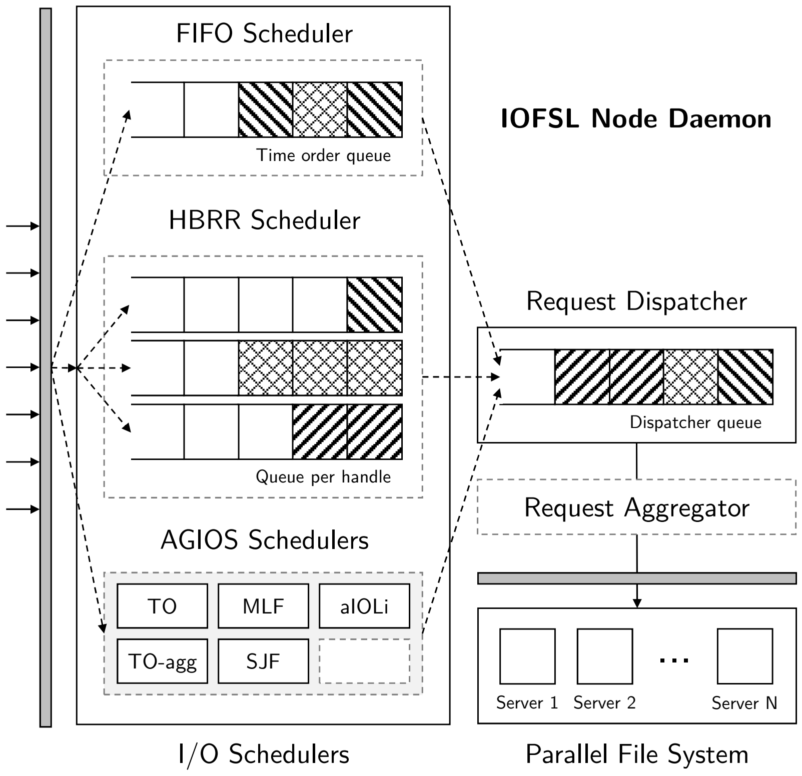
It is important to notice that no scheduling algorithm is able to improve performance for all situations, and the best fit depends on applications' and storage devices' characteristics (BOITO, 2015). The scheduling algorithms in AGIOS were evaluated only in the context of the parallel file system servers. Hence, it is still unknown if those that presented performance improvements are also able to improve read and write times when decisions are made at the I/O forwarding layer. For this reason, we focus this study on evaluating these different scheduling algorithms in this layer.

For the purposes of this research, from the available alternatives, we select three schedulers: TO, SJF, and MLF. Although based on the same principle as the FIFO scheduler, we included the TO to illustrate the overhead of redirecting the flow of requests through AGIOS. Moreover, the aIOLi scheduler is not considered because its synchronous approach is not suited for the forwarding layer, and neither is the TO-agg because IOFSL already has an aggregator before dispatching the requests to the file system.

3.3 Integrating AGIOS to the IOFSL Framework

Since AGIOS can be used by I/O services to manage incoming I/O requests at file level (file offsets), we have integrated the scheduling library into the IOFSL framework as a new scheduling option, just like FIFO or HBRR. This integration allows us to evaluate

Figure 3.3: Flow of requests through an IOFSL node daemon with the new schedulers.



Source: Author

existing schedulers for the parallel file system servers in the forwarding layer.

The new organization inside the I/O node is illustrated in Figure 3.3. With the AGIOS scheduling option, incoming requests are added to the library's queues when they arrive at the forwarding layer. When the algorithm applied by AGIOS decides that it is time to process a request, the callback function written inside IOFSL adds it to the dispatch queue. This ensures that requests will be processed in the order dictated by the scheduling algorithm in use by AGIOS.

Scheduling decisions for the existing algorithms implemented in IOFSL are local to an I/O node, i.e. no global decisions or explicit coordination is done. Furthermore, since AGIOS' algorithms also share this characteristic, this behavior is maintained after the integration. The library also exposes an API to prototype new schedulers.

3.4 Performance Evaluation

In this section, we evaluate and analyze the existing scheduling algorithms in IOFSL and in AGIOS. Details about the platform used in our experiments are given in Section 3.4.1 and our evaluation methodology is described in Section 3.4.2. Sections 3.4.4 and 3.4.4 discuss the results obtained with IOFSL and AGIOS, respectively. Finally, Section 3.5 summarizes the results and brings finals remarks to this chapter.

3.4.1 Experimental Platform

All experiments conducted in this dissertation were carried out in two clusters from the Nancy site of Grid'5000 (BALOUEK et al., 2013). Four machines from the *Grimoire* cluster were used as PVFS2 servers (acting as both data and metadata servers) and 32 machines from the *Grisou* cluster were employed as clients. Up to 8 additional machines from Grisou were selected to act as forwarding servers. Hence, each I/O node runs on a separate machine not shared with clients or PFS servers. Furthermore, the forwarding nodes are placed in the same cluster, close to the clients, to represent a real life-like scenario.

Each node of the Grimoire cluster has two 8 core Intel Xeon E5-2630 v3 and 128GB of RAM. Grisou nodes are identical to Grimoire ones. A 558GB HDD is used for storage at each server. Nodes are interconnected through a 10Gbps Ethernet network, and there is also a 10Gbps link between the clusters. During the experiments, we exclusively reserved all nodes of both clusters to minimize interference of concurrent jobs.

We used version 2.8.2 of PVFS with all its default parameters, including a simple stripe distribution and a 64KB stripe size. Data servers were configured to perform I/O operations directly to their storage devices, bypassing buffer caches (`-trove-method directio`). This was done to avoid a situation where the scale of the tests would hide the access pattern impact on performance.

The IOFSL framework was deployed in each I/O node, and its dispatcher uses the PVFS client library to communicate with the file system, allowing a direct access instead of accessing it through the PVFS kernel module. The use of IOFSL is transparent to applications, as accesses are forwarded through the ZOID API. Therefore, no additional modifications in the application's source code were necessary. An environment variable (`ZOIDFS_ION_NAME`) is set at the processing nodes to determine where requests must be redirected. In tests where we work with more than one IOFSL node, clients are equally distributed among I/O nodes. Additionally, the IOFSL daemon, which runs on each I/O node was executed with all its default parameters, keeping the maximum number of requests that can be aggregated from the dispatch queue (batch size) as 16. For the event handler, IOFSL uses state machines.

3.4.2 Experimental Methodology

Los Alamos National Lab's MPI-IO Test was written for parallel I/O and scale testing (LANL, 2006). The MPI-IO test is built on top of MPI's I/O calls and is used to gather timing and bandwidth information for different access patterns such as N processes writing to N files, N processes writing to one file, N processes sending data to M processes writing to M files, or N processes sending data to M processes to one file. We selected the MPI-IO Test benchmark because it implements different access patterns, including the scenario where a single file is shared between processes and the access to this file is not contiguous (1D strided). Other benchmarks, such as the IOR do not support this access pattern common to applications and know to harm performance (LIU; CHEN; ZHUANG, 2013; WANG et al., 2014).

The MPI-IO Test benchmark was executed by 128 processes to generate requests through the MPI-IO interface. Tests were executed for the file-per-process approach, where each process accesses contiguously its own independent file, and for the shared file one. With the shared file approach, processes either access their own contiguous portions or follow a 1D strided access pattern. These experiments represent access patterns that are usual among scientific applications (LARREA et al., 2015).

Processes issue small (32KB) or large (256KB) requests to read or write files. Small requests sizes are denoted to be less than the PFS stripe size, i.e. 64KB. On the other hand, large request sizes represents a scenario where more than one data server is required to complete the request. For the purposes of our experiments, we selected a value so that all four data servers were involved in the operation. In each experiment the application reads or writes a total of 4GB, i.e. 32MB per process.

From each execution, we take the *makespan*, i.e. the completion time of the slowest process, as the test's execution time. We use this value as a performance metric because it represents the total time to process a workload from the file system point of view.

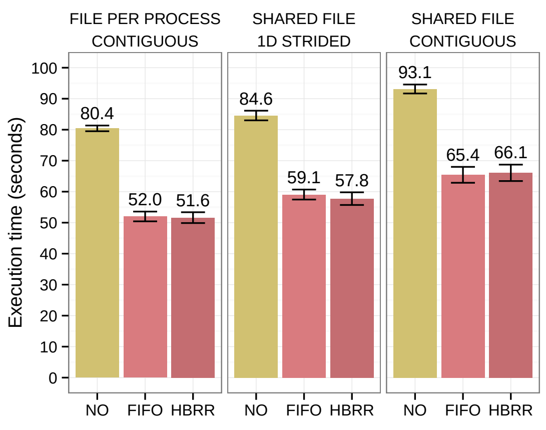
Experiments were repeated at least 8 times, and error bars were calculated using a $99,7\%$ confidence interval, i.e. there is a 99.7% probability that the true mean lies between the lower and upper bounds of the interval. These bounds are equivalent to three times the standard deviation divided by the square root of the number of measurements. Different experiments were executed in a random order to avoid bias imposed by some uncontrolled parameter, or some unexpected effect caused by a specific experimentation order. Additionally, the PFS servers and the forwarding services were stopped and restarted before each experiment to avoid possible interference of previous executions.

3.4.3 Performance of the IOFSL Scheduling Algorithms

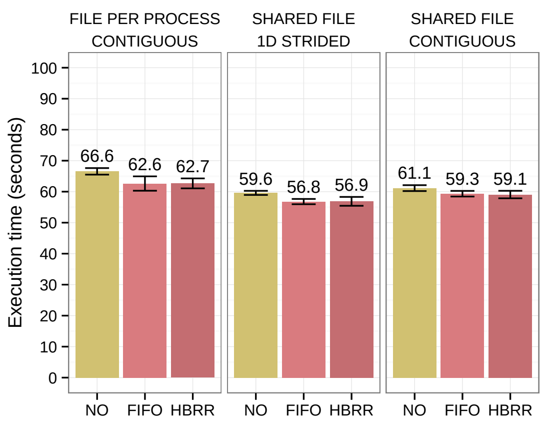
In order to evaluate and analyze the trade-offs of different scheduling algorithms at the forwarding layer, we must determine the impact of this layer on the studied scale. For that, we define a scenario where each node directly accesses the file system servers.

We consider two schedulers already implemented in the IOFSL framework: FIFO and HBRR. The FIFO scheduler highlights the impact of forwarding the requests because the intermediate I/O nodes are an extra hop between processing nodes and the file system. In this approach, I/O nodes do not act as burst buffers but instead clients' expectation of persistent storage in the file system is met. Therefore, as this scheduler does not include additional waiting times or any complex decision-making process, we can evaluate the cost of the requests going though the I/O nodes plus the extra network hop before reaching the data servers.

Figure 3.4: Execution time of **read** requests directly accessing PVFS and with the IOFSL default schedulers: FIFO and HBRR.



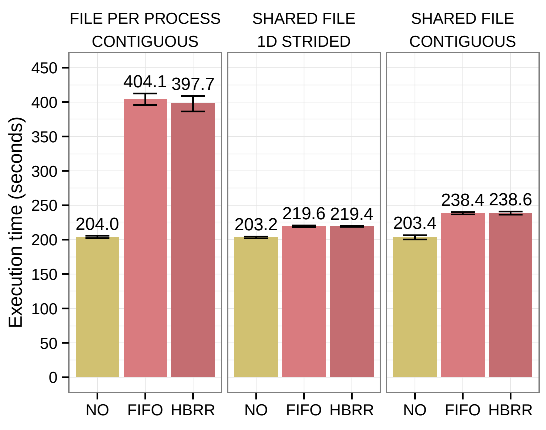
(a) Small requests (32KB)



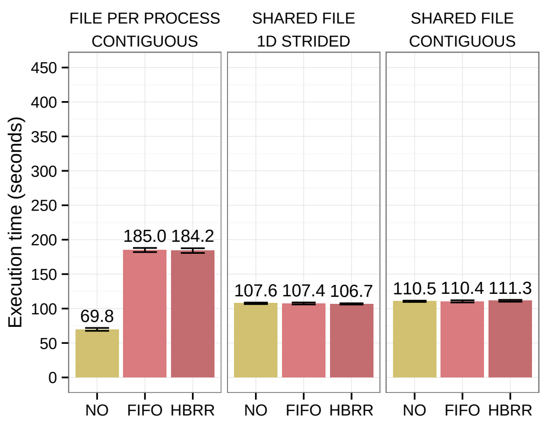
(b) Large requests (256KB)

Source: Author

Figure 3.5: Execution time of **write** requests directly accessing PVFS and with the IOFSL default schedulers: FIFO and HBRR.



(a) Small requests (32KB)



(b) Large requests (256KB)

Source: Author

Figures 3.4 and 3.5 present the results obtained in the read and write tests, respectively, with the three selected access patterns, described in Section 2.2.1. In all plots, the first column (in yellow) denotes the time obtained without using the forwarding layer, labeled as NO (no forwarding); and the second and third columns (in shades of red) show the time obtained using IOFSL with its base scheduling algorithms. On the latter, the first bar illustrates the FIFO scheduling algorithm, while the second bar represents HBRR.

We can see that the read performance for small requests (32KB) is improved up to 35.80% just by using IOFSL (Figure 3.4), despite the extra transmission cost between clients and the file system. Performance benefits from using the I/O forwarding layer to all tested read access patterns: 35.80% when each process accesses its own file, 31.70%

and 23.73% when a common shared file is used with 1D strided and contiguous accesses, respectively. Better results were observed for small requests than for large ones (256KB). This can be explained by the fact that small requests benefit more from being aggregated before arriving at the servers than the larger ones.

On the other hand, for write requests (Figure 3.5), performance significantly decreased in all scenarios, except for large requests to a shared file where no degradation was observed. However, for small accesses to a shared file, the increase in execution time is significant: 8.05% for 1D strided and 17.33% contiguous accesses. Large requests in the file-per-process scenario presented the worse results. We observed performance degradation of up to 165.23% if compared to writing data directly to the PVFS data servers. Additionally, if we take into consideration only the FIFO and HBRR schedulers they do present, on average, small differences in time, but error bars do not allow us to say that they are statistically different.

One could believe the explanation for the good results observed for read tests is that the gains obtained by aggregating requests before forwarding them to the PFS compensates the overhead imposed by the extra hop. Nonetheless, this is not the case. Table 3.1 details the median request size at different layers of the I/O stack during at least 10 repetitions of each one of the experiments with the IOFSL schedulers and the shared file scenario. We did not include values for file-per-process because each file is accessed by a single process, one request at a time, so there are no aggregation opportunities.

Table 3.1: Average request size (in KB) at different levels of the I/O stack for the shared file scenario with small (32KB) 1D strided and contiguous accesses.

		1D Strided (KB)		Contiguous (KB)	
		READ	**WRITE**	**READ**	**WRITE**
FIFO	Leaving the clients	32	32	32	32
	Arriving at the I/O nodes	32	32	32	32
	Leaving I/O nodes	57.75	57.72	57.72	57.79
	Arriving at the data servers	49.87	49.23	43.52	43.66
HBRR	Leaving the clients	32	32	32	32
	Arriving at the I/O nodes	32	32	32	32
	Leaving I/O nodes	57.77	57.68	57.77	57.69
	Arriving at the data servers	49.80	49.36	42.75	43.65

Source: Author

We can see that write tests present similar aggregated request sizes by IOFSL, but they still do not achieve the same gains as the read tests. Moreover, the average request size aggregated by the FIFO and HBRR schedulers are also similar.

Therefore, despite aggregating requests usually being helpful for performance, this is not the main factor in the observed improvements of read operations. Another evidence in this direction is that the best gains were seen with small requests in the file-per-process scenario, where there are no aggregations opportunities. Furthermore, despite making more effort into generating a better access pattern, HBRR does not outperforms FIFO. Results for the two algorithms were not statistically different in any of the tests.

Table 3.2 presents the average offset distance of requests leaving the I/O nodes during the shared file tests. The offset distance is a spatiality metric calculated by taking the offset difference between every two consecutive requests. The higher the distance, the less contiguous an access pattern is. The contiguous local access pattern presents the highest average offset distances, i.e., it is actually the least contiguous global access pattern. This happens because each process contiguously accesses its own portion of the shared file, but different processes are accessing requests that are sparse in the file. During the 1D strided test, requests from different processes are contiguous to each other. The average offset distances during tests with FIFO and HBRR are very similar. Hence, we can notice both algorithms result in very similar access patterns. This explains why they perform similarly.

Table 3.2: Average offset distance (in MB) leaving the I/O nodes for the shared file tests.

	1D Strided (MB)		Contiguous (MB)	
	READ	**WRITE**	**READ**	**WRITE**
FIFO	42.94	47.83	1358.18	1353.51
HBRR	42.05	47.57	1357.22	1353.79

Source: Author

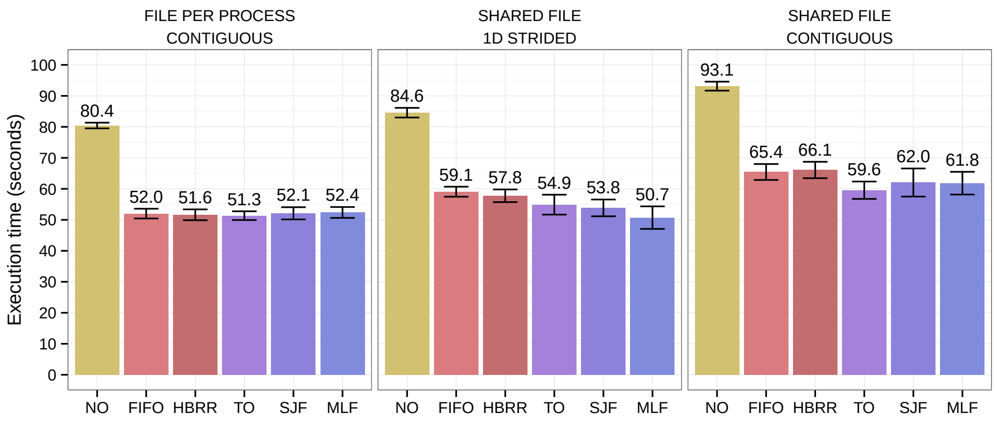
We have also measured the time difference between consecutive requests. These values were obtained from four new executions of the 1D strided shared file test through a single I/O node. In the intermediate I/O node, we have traced all requests' arrival time. We have considered only the first 128, i.e. the first request from each of the 128 processes. Since tests are synchronous, all requests after the first 128 depend on the time it took to process the previous ones, so they are not independent. We have used the median because it is less sensitive to outliers, and timing values inside each test tend to have high variability. For read requests, the median time difference is 26.09μs, whereas for write requests it is 50.92μs. Since read requests are smaller than writes (they do not carry data when issued), they arrive at a faster pace to the I/O nodes or to the server, if I/O nodes are not present. Hence, the extra hop between clients and PFS works to "funnel" requests and decrease concurrency at the servers.

Therefore, despite the fact that read and write requests present similar aggregated sizes in the shared file scenario, when leaving the I/O nodes, they do not provide the same performance gains. Furthermore, as demonstrated by the average offset distance of requests leaving the I/O nodes, both algorithms result in very similar access patterns, although HBRR puts more effort into optimizing the requests.

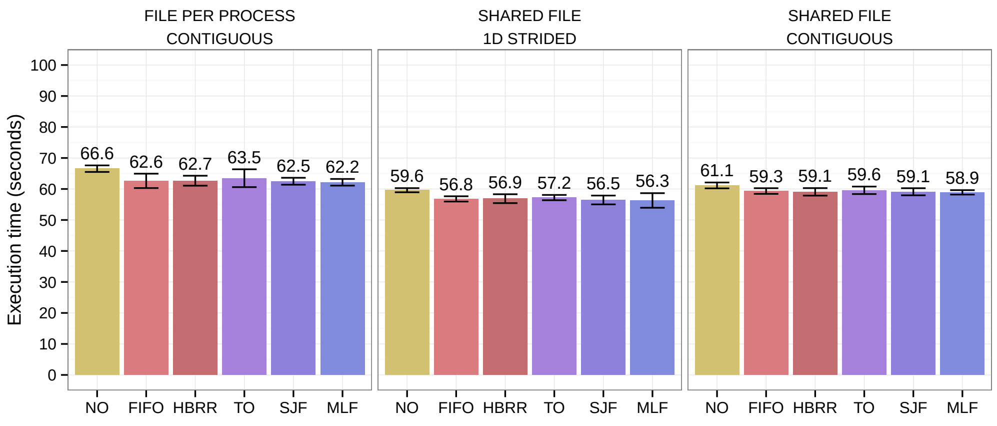
3.4.4 Performance of the AGIOS Scheduling Algorithms

We applied the same methodology to evaluate three additional schedulers selected from AGIOS. Figure 3.6 expands the previous plots, introducing the results of read operations with the three selected schedulers: TO, SJF, and MLF (in shades of purple).

Figure 3.6: Execution time of **read** requests directly accessing PVFS with the IOFSL default schedulers (FIFO and HBRR) and AGIOS schedulers (TO, SJF, and MLF).



(a) Small requests (32KB)



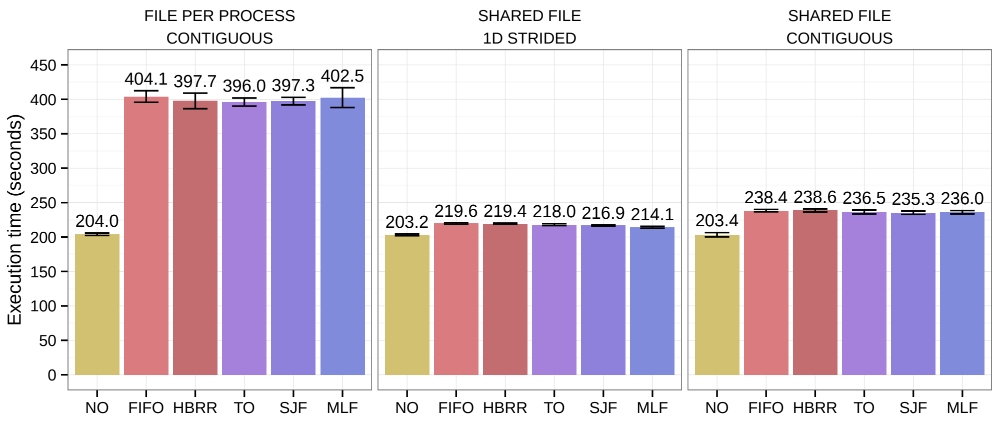
(b) Large requests (256KB)

Source: Author

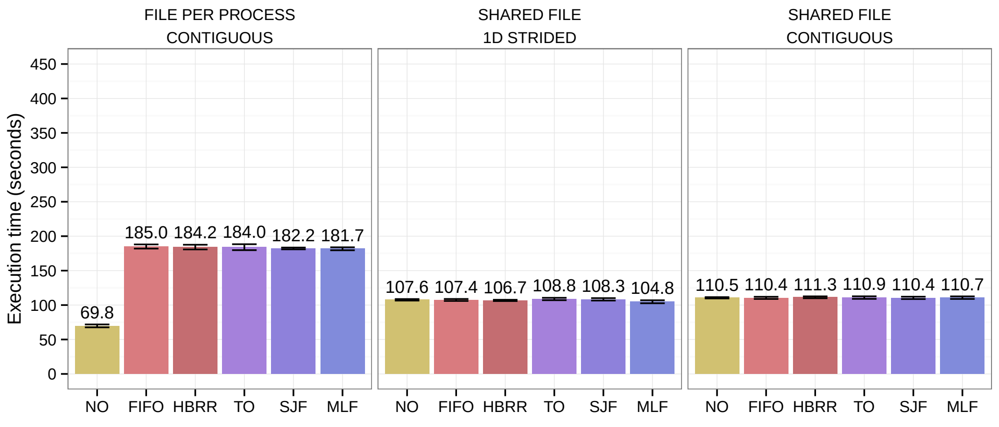
Considering first the file-per-process scenario we still do not see any statistically significant difference between the alternatives. The included schedulers behave just like FIFO and HBRR for both small (32KB) and large (256KB) requests. Moreover, for the shared file scenario, with large requests the same conclusion applies. With small strided requests it is possible to see that SJF presented a small reduction in the execution time. MLF significantly improved performance by 14.17% if compared to the default FIFO scheduler. For contiguous small accesses, the TO scheduler from AGIOS also improved the performance up to 9.87% over HBRR.

On the other hand, for write requests, we observed the same behavior for all the tested schedulers, as depicted by Figure 3.7. There are small variations between the mean time of each alternative, but they are not statistically significant and in some situations, this is not expressive enough to justify usage of one scheduler rather than the other.

Figure 3.7: Execution time of **write** requests directly accessing PVFS with the IOFSL default schedulers (FIFO and HBRR) and AGIOS schedulers (TO, SJF, and MLF).



(a) Small requests (32KB)



(b) Large requests (256KB)

Source: Author

Table 3.3 details the average request size at different levels of the I/O stack for these schedulers. We have also included the previous results with FIFO and HBRR to facilitate the correlation of results. The first noticeable behavior is that for both read and write requests TO, SJF, and MLF schedulers from AGIOS manage to aggregate more than the previously tested alternatives.

Table 3.3: Average request size (in KB) at different levels of the I/O stack for the shared file scenario with small (32KB) 1D strided and contiguous accesses.

		1D Strided (KB)		Contiguous (KB)	
		READ	**WRITE**	**READ**	**WRITE**
FIFO	Leaving the clients	32	32	32	32
	Arriving at the I/O nodes	32	32	32	32
	Leaving I/O nodes	57.75	57.72	57.72	57.79
	Arriving at the data servers	49.87	49.23	43.52	43.66
HBRR	Leaving the clients	32	32	32	32
	Arriving at the I/O nodes	32	32	32	32
	Leaving I/O nodes	57.77	57.68	57.77	57.69
	Arriving at the data servers	49.80	49.36	42.75	43.65
TO	Leaving the clients	32	32	32	32
	Arriving at the I/O nodes	32	32	32	32
	Leaving I/O nodes	75.09	65.70	77.07	66.34
	Arriving at the data servers	53.69	51.41	53.37	50.98
SJF	Leaving the clients	32	32	32	32
	Arriving at the I/O nodes	32	32	32	32
	Leaving I/O nodes	74.79	65.71	76.81	65.91
	Arriving at the data servers	53.63	51.33	53.31	50.94
MLF	Leaving the clients	32	32	32	32
	Arriving at the I/O nodes	32	32	32	32
	Leaving I/O nodes	79.59	66.39	76.53	66.82
	Arriving at the data servers	51.17	48.31	50.44	49.22

Source: Author

We believe that just for including the library in the flow of requests through the I/O nodes, requests are been slightly delayed, providing more aggregation opportunities. This small delay may be imposed by the additional coordination between the library's threads when concurrently accessing the shared queues. Although AGIOS is aggregating more, the results are still similar to the other schedulers. This indicates that the aggregations are not what define the performance in this scenario. Thus, there may be something else that is influencing the results.

3.5 Conclusions

By including the forwarding layer in the I/O stack of our experimental testbed, small read requests have demonstrated large performance improvements. Even though this represents an extra hop between clients and data servers, we could see a reduction in the time it took to serve those requests for the file-per-process and shared file scenarios.

Although it was expected that not all algorithms were to improve performance on the forwarding layer, results differ from our initial assumption by demonstrating few cases where significant performance was gained. The existing FIFO and HBRR schedulers implemented in the IOFSL framework do not appear to be distinct at the scale and workload tested. Moreover, performance was also not improved by the solutions implemented in AGIOS: TO, SJF, and MLF. It was possible to see, by measuring the average size of requests at different points of the I/O stack, that those schedulers were able to aggregate more, thus issuing larger requests to the PFS. Despite that, performance did not benefit from their usage, which suggests that they are only partially effective.

We believe results to be related to the congestion still present when accessing the parallel file system data servers. Therefore, the lack of a coordination mechanism between the I/O nodes may still be directly affecting performance. To test our hypothesis, we propose a new scheduler that focuses on coordinating this access to further mitigate contention, as it is detailed in Chapter 4.

4 TWINS: AN I/O SCHEDULER TO COORDINATE SERVER ACCESS

Based on the results collected from our initial experiments, presented in Chapter 3, with the schedulers available in IOFSL (FIFO and HBRR) and in AGIOS (TO, SJF, and MLF), we propose two new schedulers whose goal is to further decrease concurrency when accessing the file system servers. In this chapter, we argue about the impact of I/O contention and the importance of coordinated data accesses. Building on that, we describe the concepts and mechanisms of our new I/O scheduling algorithms for the I/O forwarding layer: *Server Window* (SW) and *Server Time WINdows* (TWINS).

4.1 I/O Contention and Coordination

Research on optimizations to reduce the effects of I/O contention can be classified into two categories: client side and server side. As we focus on the forwarding nodes, which act as PFS's clients – just as the computing nodes do in a setup where no I/O forwarding exists – we must examine this problem from the client's perspective. On the client side, processes usually collaborate by coordinating accesses to the PFS.

For instance, Nisar, Liao and Choudhary (2008) propose a mechanism to delegate certain tasks, such as file caching, consistency control, and collective I/O optimization to a small set of compute nodes, thus mitigating resource contention. Their experimental evaluation indicates considerable performance improvement with a small percentage of computing resources reserved to act as I/O delegators. Additionally, Abbasi et al. (2009) argue that the use of asynchronous methods for data transfer can reduce or eliminate the blocking time experienced by HPC codes using synchronous I/O. However, an issue with such approach is the need for a coordinated use of machine resources, to avoid the contention caused by the aggressive data transfers performed for I/O purposes.

In the previous experiments with the different scheduling algorithms, we noticed only small differences in performance between the evaluated solutions. Furthermore, differently from what was observed when AGIOS schedulers were employed in the parallel file system data servers (BOITO, 2015), at first glance, the tested algorithms did not seem fit for the forwarding layer. When we observed the request aggregation sizes, we could notice that the algorithms are performing their work, i.e. they are merging and combining contiguous requests. Therefore, there must be another factor that is affecting the performance.

Our initial hypothesis is that the uncoordinated access to the data servers could still be harming performance. Therefore, we believe that by creating a new scheduler that is able to focus access to one server at a time, we could reduce contention and improve performance. The next section provides details on how we plan on achieving this goal.

4.2 Server Access Coordination

The main idea behind our proposed scheduler is to coordinate intermediate I/O nodes' accesses to the file system so that, at any given moment, the following two conditions hold true:

I. an I/O node is focusing its accesses to only one data server;

II. different I/O nodes are focusing on different servers.

We created a new scheduler scheduler to use time windows as a coordination mechanism, similar to what was proposed by (SONG et al., 2011) for the PFS servers. There, a window-wide coordination concept was employed to make all data servers focus on serving requests from one application at a time. Our proposal to make intermediate I/O nodes dedicate time windows to different data servers was inspired by their work. Nonetheless, there are at least two differences between their proposal and ours. First, they target the PFS servers while we change the behaviour of the intermediate I/O nodes. Second, their algorithm coordinates access from different applications, while we coordinate server accesses.

We named this scheduler as *Server Window* (SW). The pseudo-code is presented by Algorithm 1. As soon as a request arrives at the I/O node, it is sent to AGIOS. The scheduler then calculates its priority and inserts it in a single request queue, in ascending order of "priority", thus requests with smaller priority would be scheduled earlier.

Algorithm 1 *Server Window* (SW)

```
Require: $Q$ is the list of requests to be serve
Require: $R$ is the new incoming request
1: $priority \leftarrow ((R.timestamp/windowSize) * max) + R.serverID$
2: for each $request$ in $Q$ do
3:     if $request.priority > priority$ then
4:         $Q.insert\_after(request.previous, R)$
5:         break
6:     end if
7: end for
```

Since we have integrated the AGIOS scheduling library (BOITO et al., 2015) in IOFSL as a scheduling option (just like FIFO and HBRR), as detailed in Section 3.3, we harness its API to prototype and evaluate our solution. It would have been possible to implement it inside the IOFSL source code instead. However, doing so with AGIOS makes our solution more generic, as it can be used by other I/O services which use this library, or even by other I/O forwarding frameworks.

As a prerequisite for this scheduler to work, we need to know the destination server of each incoming request, i.e. since a file is broken down into stripes that are distributed amongs the servers, we need to know where the stripe that contains the request's data is located among the data servers. Normally that information is not available at this layer of the I/O stack. The next section details how it is obtained and the additional overhead.

4.2.1 Required Information to Determine the Data Servers

In addition to typically available information about requests such as file handle, offset, type of operation, and size, our algorithm requires a server identifier to disclose the location of the very first stripe of a file. Based on that and on the distribution employed by the PFS we can determine the location of every other stripe for that file. We have modified IOFSL to collect the file distribution information from the PFS metadata servers when opening or creating a file. Since this information is easily available to clients in most file systems such as PVFS and Lustre, our solution can still be considered file system generic.

The distribution information is requested only once per file. To determine the overhead of fetching such information, we conducted a small experiment, by measuring the time it took to obtain this data. The introduced overhead is of 54.3ms on our experimental environment (average of 124 observations). Since this information is requested only once per file, it is possible to say that it is relatively small if compared to the read and write times to the PFS. Furthermore, as more operations are issued to the same file, this overhead is expected to be diluted throughout the execution. Therefore, the total overhead introduced is the sum of all calls (one per file), and it depends on the network speed.

Using the file distribution information, the starting server for a request is obtained as a function of its starting offset and stripe size, also part of the collected distribution information, as illustrated by Equation 4.1. Servers are mapped to identifiers between 0

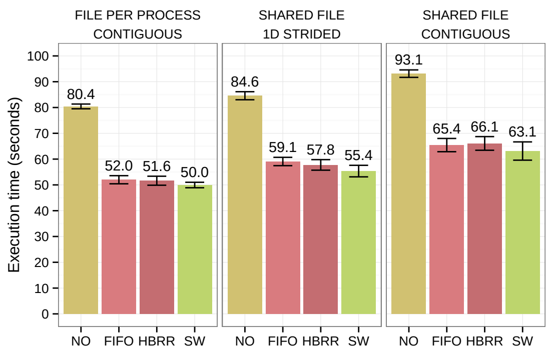
and $numberofservers - 1$, according to their machines names.

$$destionationServer = \left(firstStripeLocation + \frac{offset}{stripeSize}\right)\%\ totalServers \quad (4.1)$$

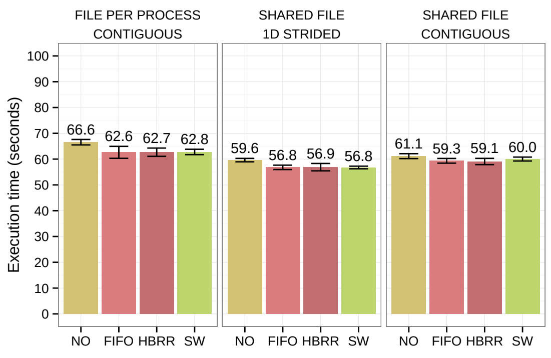
4.2.2 Performance Evaluation

In this section, we employ the same experimental setup and methodology described in Sections 3.4.1 and 3.4.2 to evaluate the *Server Window* (SW) scheduler. For this first set of experiments we define the window size as one second. Additionally, an investigation of this parameter is presented in Section 4.2.3.

Figure 4.1: Execution time of **read** requests using the SW scheduler when compared to alternatives.



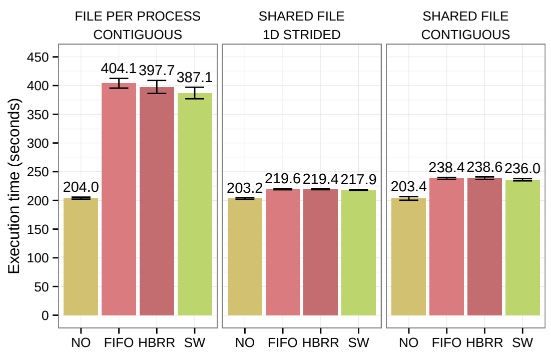
(a) Small requests (32KB)



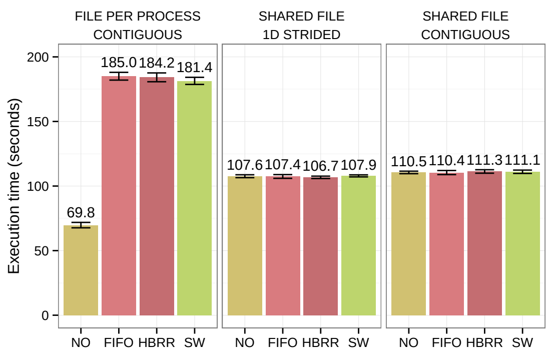
(b) Large requests (256KB)

Source: Author

Figure 4.2: Execution time of **write** requests using the SW scheduler when compared to alternatives.



(a) Small requests (32KB)



(b) Large requests (256KB)

Source: Author

Table 4.1: Average request size (in KB) at different levels of the I/O stack for the shared file scenario with small (32KB) 1D strided and contiguous accesses.

		1D Strided (KB)		Contiguous (KB)	
		READ	**WRITE**	**READ**	**WRITE**
FIFO	Leaving I/O nodes	57.75	57.72	57.72	57.79
	Arriving at the data servers	49.87	49.23	43.52	43.66
HBRR	Leaving I/O nodes	57.77	57.68	57.77	57.69
	Arriving at the data servers	49.80	49.36	42.75	43.65
TO	Leaving I/O nodes	75.09	65.70	77.07	66.34
	Arriving at the data servers	53.69	51.41	53.37	50.98
SJF	Leaving I/O nodes	74.79	65.71	76.81	65.91
	Arriving at the data servers	53.63	51.33	53.31	50.94
MLF	Leaving I/O nodes	79.59	66.39	76.53	66.82
	Arriving at the data servers	51.17	48.31	50.44	49.22
SW (1s)	Leaving I/O nodes	73.08	64.93	72.64	64.11
	Arriving at the data servers	50.08	48.38	50.16	48.17

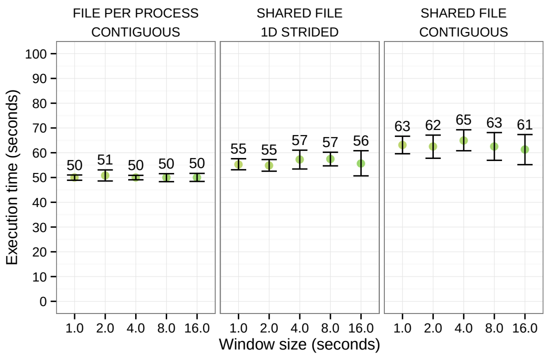
Source: Author

SW does not seem to bring performance improvements when compared to the previously tested alternatives. Figures 4.1 and 4.2 illustrate the average execution time for read and write requests, respectively, considering the distinct access patterns taken into account in this study. It appears that, just by ordering requests by destination server and using a "limiting" window so requests do not starve, it is not enough to coordinate access and reduce contention. For the sake of completeness, we also show the aggregate sizes observed with SW in Table 4.1. Although SW is aggregating more than FIFO and HBRR, it is not able to aggregate as much as the schedulers from AGIOS. Despite that, no overhead or loss in performance is visible.

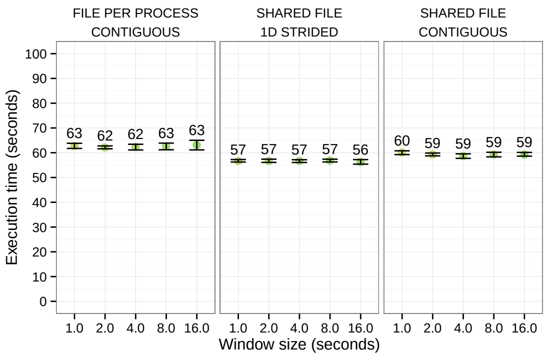
4.2.3 Investigating the Window Size

Since we are working with a time-window based scheduler, we must also take into consideration the impact of distinct values for window size. In our previous experiments we considered initialy a one second window, as proposed by Song et al. (2011) in his approach for the data servers with HDD devices. However, since we are no longer in the context of a single PFS data server, but on an upper layer of the I/O stack, we should also investigate larger window sizes to account for the costs of remote accessing the data. Accordingly, we evaluate five other window sizes, from 1s to 16s.

Figure 4.3: Execution time of **read** requests using different window sizes for the SW.



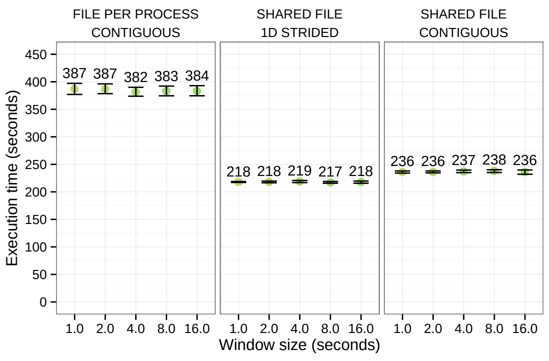
(a) Small requests (32KB)



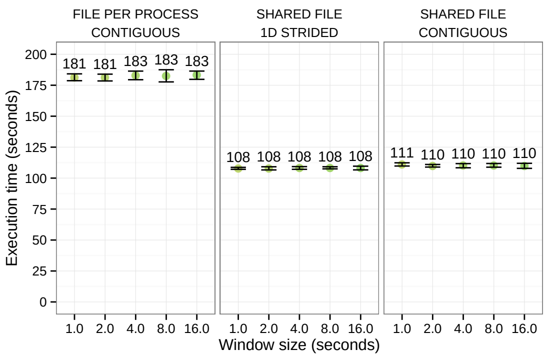
(b) Large requests (256KB)

Source: Author

Figure 4.4: Execution time of **write** requests using different window sizes for the SW.



(a) Small requests (32KB)



(b) Large requests (256KB)

Source: Author

Figure 4.3 depicts the different window sizes (seconds) in the x-axis and the execution time of the slowest process to complete (makespan) in the y-axis, for all the tested access patterns. It is important to notice that the x-axis is non-linear. We can see that despite increasing the window size there are no improvements nor degradation on performance. The same holds true for write requests, as illustrated in Figure 4.4.

4.2.4 Discussion

We proposed a new scheduler named *Server Window* (SW) as an effort to reduce concurrency when accessing the PFS data servers, yet SW seemed ineffective. Two possible explanations for the observed results are:

I. concurrency was actually not affecting performance;

II. SW was not, in fact, able to reduce concurrency.

We traced the execution of the algorithm, in the forwarding layer, by recording additional information such as when requests arrived at the I/O node; when they were included in the algorithm's queue; when they were selected to be served; and their aggregated sizes and offsets. We were able to determine that although SW indeed sorted the requests grouping them by servers, they ended up merged on the dispatch queue. Consequently, they were sent together to the PFS data servers. This indicates that the requests were not staying long enough in SW's queues to enforce the desired coordination effect.

Based on that, the next section proposes a new scheduler to target that issue by considering a fixed time window. This may imply in additional waiting times. Yet, we do this as an effort to improve performance by coordinating accesses and by further aggregating incoming I/O requests.

4.3 Time Window Based Scheduler

As demonstrated in the previous section, the SW scheduler is not able to coordinate accesses. We believe this to be correlated to the way SW works, with no fixed time window but rather an abstract window so requests do not starve. Thus, requests are not staying long enough in the algorithm's queue to ensure a coordination. In order to improve that, we propose a new scheduler with fixed time windows.

We present a new scheduling algorithm for the I/O forwarding layer named *Server Time WINdows* (TWINS). TWINS pseudo-code is presented in Algorithm 2. Differently from TO and SW, it keeps multiple request queues, one per data server. During the exe-

Algorithm 2 *Server Time WINdows* (TWINS)

```
Require: $Q[i]$ is the updated list of requests to server $i$
 1: $i \leftarrow 0$
 2: while $true$ do
 3:     $resetTimer()$
 4:     while $elapsedTime() < windowSize$ do
 5:         if $length(Q[i]) > 0$ then
 6:             $processRequest(Q[i])$
 7:         else
 8:             $timeout \leftarrow windowSize - elapsedTime()$
 9:             $timedWaitForRequests(Q[i], timeout)$
10:         end if
11:     end while
12:     $i \leftarrow nextServer(i)$
13: end while
```

cution, TWINS iterates between the different queues in a round-robin fashion, respecting a time window that must be dedicated to each server. This means that, if server i is the current server being accessed but there are no requests to this server, the scheduler will wait until requests to server i arrive or the time window ends, even if there are incoming or queued requests to other servers.

Using TWINS, requests are added to the per-server queues upon arrival at IOFSL. When the algorithm decides to process a request, a callback function written inside IOFSL simply adds it to the dispatch queue. This ensures requests will be processed in the order dictated by the scheduler.

Differently from FIFO, that uses a single queue, and HBRR, that uses two queues per file handle, TWINS uses one queue per data server. Considering that the number of files is typically far superior to the number of servers, the overhead induced by TWINS regarding the management of multiple queues is expected to be lower than what is caused by HBRR.

Examining the described TWINS algorithm, we can notice that simply following this approach would cause all intermediate I/O nodes to focus on the same servers at the same time. To cause the desired distribution effect, we add an extra server identifier translation step before adding requests to the corresponding queues. This translation is done according to the I/O node identifier. The N_{th} I/O node will use the N_{th} permutation of the servers list as a translation rule. Therefore, if the number of intermediate nodes is larger than the number of servers, more than one node may access the same server at the same time, but these concurrent accesses are minimized. For instance, if there are 4 I/O nodes (N_1 to N_4) and four data servers (S_1 to S_4) each I/O node will use the following order:

- $N_1 : S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4$
- $N_2 : S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_1$
- $N_3 : S_3 \rightarrow S_4 \rightarrow S_1 \rightarrow S_2$
- $N_4 : S_4 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3$

Therefore the translation function in each I/O node maps the initial server to a different data server. This ensures that each I/O node will focus on a different server at each time window.

Table 4.2 summarizes the complexity costs for inserting and selecting requests from the queue(s) of the proposed schedulers, compared to TO. It is important to notice

Table 4.2: Summary of the time complexity for inserting and selecting requests in each scheduler. M is the number of servers and N is the number of requests in the queue.

Scheduler	Queues	Insert	Select
TO	1	$O(1)$	$O(1)$
SW	1	$O(N)$	$O(1)$
TWINS	M	$O(1)$	$O(1)$

Source: Author

that despite being more elaborated than SW, TWINS has a lower complexity cost for inserting requests due to the additional information the algorithm has. For instance, TWINS does not have to iterate over M queues, one for each server, because the destination server is previously known. Furthermore, as requests are appended to the end of each server's queue, TWINS do not need to iterate over it.

4.4 Conclusions

This chapter presented two new schedulers crafted for the I/O forwarding layer: *Server Window* (SW) and *Server Time WINdows* (TWINS). Unlike other schedulers, they aim at coordinating accesses to the parallel file system data servers, to further reduce the contention.

We presented SW and an evaluation of its performance and aggregation capabilities considering distinct window sizes. However, since SW does not work with a fixed time window, requests were not staying in the algorithm's queues long enough to be grouped by the destination server. To solve this, we created TWINS, that imposes additional waiting times by using a fixed window size. Furthermore, we discussed how TWINS proposes to coordinate the accesses without additional communications between the I/O nodes.

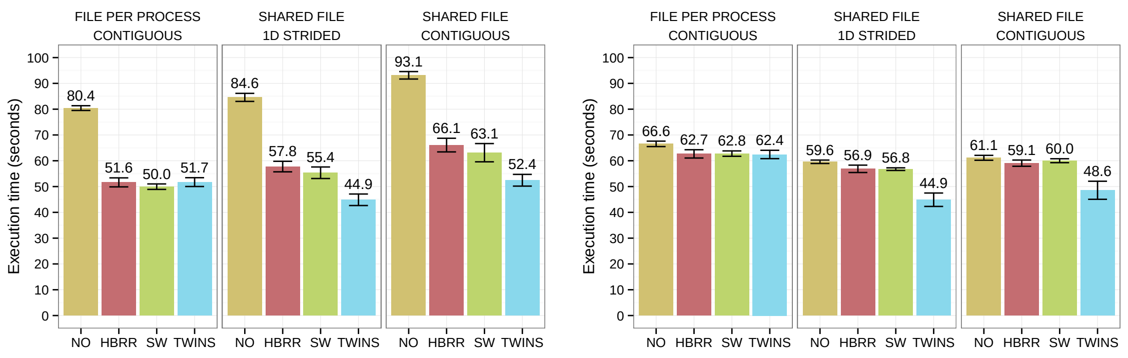
The next chapter is dedicated to a thorough evaluation of TWINS, including an investigation of the impact of distinct window sizes and its usage on a multi-application scenario. Moreover, we compare our new scheduler to all the previously tested algorithms, considering aggregation sizes and execution time.

5 EXPERIMENTAL RESULTS

In this chapter, we evaluate the performance of the new TWINS algorithm, which works to decrease contention in the access to the parallel file system data servers. As described in Section 4.3, in order to do that, TWINS divides the execution in time windows and focuses each IOFSL node's accesses to a different data server. The experimental environment and setup remain the same for this evaluation. Further details can be found in Sections 3.4.1 and 3.4.2.

Since we will be dedicating an entire window to a server, we selected a smaller value (1ms) for the time window in our first set of experiments. Figure 5.1 summarizes the results of read operations with TWINS for the three access patterns. We can see that the shared file scenario with small accesses (Figure 5.1(a)) benefits from TWINS. For 1D strided access we observe improvements of 46.93% over not using the forwarding layer, 24.03% over using FIFO, 22.30% over using HBRR and 18.93% over using SW. For the contiguous accesses these values are 43.68%, 19.85%, 20.63% and 16.92% respectively.

Figure 5.1: Execution time of **read** requests using the TWINS scheduler compared to HBRR and SW.



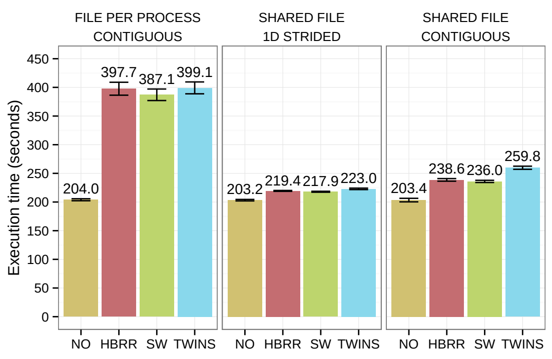
(a) Small requests (32KB) (b) Large requests (256KB)

Source: Author

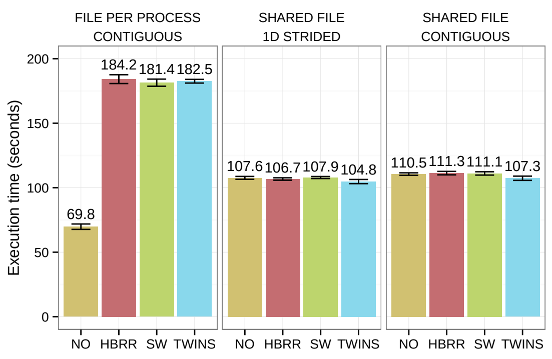
For large read requests (Figure 5.1(b)), TWINS' performance on the file-per-process scenario is similar to the other tested schedulers. On the other hand, when processes shared a file, with 1D strided acceses, our approach yields 24.68% improvements over not using the forwarding layer and 21.06% if compared to HBRR. Additionally, for contiguous access, improvements are of 20.58% and 17.79% respectively.

The lower improvements obtained for contiguous access patterns are justified by the requests distribution among the data servers, caused by the access pattern. In the 1D strided test, processes start their accesses at different servers and this behavior is kept

Figure 5.2: Execution time of **write** requests using the TWINS scheduler compared to HBRR and SW.



(a) Small requests (32KB)



(b) Large requests (256KB)

Source: Author

throughout the whole execution. Therefore, with this access pattern the scheduler always has requests for all servers and thus has the opportunity to perform meaningful coordination. In the contiguous test, since each process segment has 32MB, which is a multiple of the stripe size $\times$ the number of servers, all processes start their accesses by the same server. In this case, the situations where there are queued requests for multiple servers come during the execution as the delays induced by the TWINS scheduling algorithm causes the execution of some processes to advance faster than others. This phenomenon is not guaranteed to happen.

Performance does not benefit from using TWINS when each process issues read requests to its own file, or with write requests (Figure 5.2). Moreover, in all of the tested scenarios, with exception of small 32KB contiguous requests to a shared file, no performance degradation was observed. In the aforementioned case, we believe a different window size may help reduce the scheduler's overhead, as will be presented in Section 5.2. Therefore, in addition to improving the performance in some situations, our proposal does not necessarily harm performance in other scenarios.

5.1 Performance of Write Requests

Lofstead et al. (2011) points to some approaches that aim at addressing the mismatch between the output organization and the read pattern needs. These include the use of a staging area to host data reorganization and pre-analysis routines. Both synchronous data staging and the I/O Forwarding Software Layer (IOFSL) effectively manage the writing time spent by an application through aggregating such requests and thereby partially managing the resulting impact on the storage system. However, the before-mentioned work has not taken advantage of staging areas to accelerate subsequent data use for analysis or other reading tasks.

The previous observations concur with our findings. Based on the extensive experimentation so far, we believe there are not further opportunities to improve write requests, that is why no difference was spotted when applying different scheduling algorithms. Because write requests carry the data when they are issued, they arrive at the I/O nodes at a slower pace, if compared to the rate of read requests. This was demonstrated, in Section 3.4.3, where we observed that the time between consecutive write requests is almost twice of the time observed between read requests. Thus, at this scale, writes do not generate such an intense flow of requests which explains why funneling them through the I/O nodes does not yield performance improvements.

Additionally, as IOFSL was created to improve throughput (OHTA et al., 2010), the aggregation mechanisms in place in its dispatch queue are quite “aggressive”, which also justifies why no further improvements were observed even when we changed the scheduler.

5.2 Investigating the Window Size

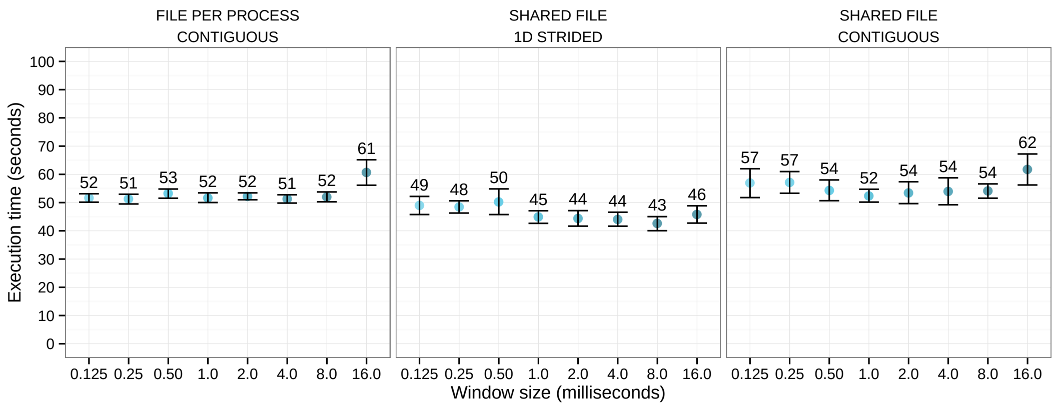
TWINS’ behavior is affected by its time window duration. A window that is too small does not allow for an effective coordination of accesses among the data servers because it is not long enough to allow the execution of multiple requests. Moreover, a fast time window does not hold requests to other servers for long enough so requests to the currently accessed data server are out of the dispatch queue to the file system. If requests for different servers are in the dispatch queue at the same time, they could be aggregated before being forwarded to the PFS and thus the scheduling algorithm work would be undone. On the other hand, a window that is too large imposes overhead as there are not enough requests to each data server to fill a whole window, so the scheduler spends too much time waiting. Another source of overhead, in this case, would be the delay imposed to requests, which could not be compensated by the gains of decreasing concurrency at the data servers.

Figure 5.3 summarizes all the experiments with different window sizes. It is importante to notice that the x-axis is non-linear. The best window duration is not the same for all situations where TWINS improves performance. The best results for small (32KB)

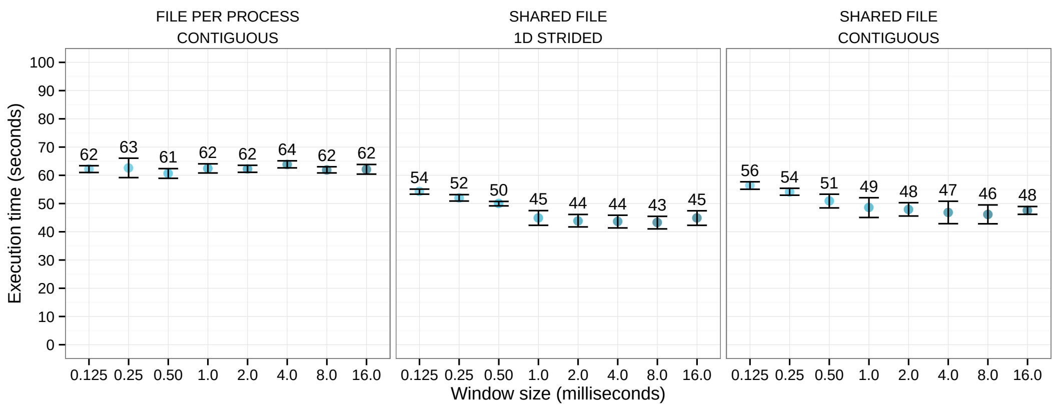
1D strided requests to a single shared file, with read operations, were observed using a 8ms window, as depicted by Figure 5.3(a). However, for the shared-file tests, a 1ms time window appears to be more suited. On the other hand, for write requests, different window sizes do not bring performance benefits. Furthermore, degradation is only observed in the shared file scenario if the window is too large, as expected, which imposes additional waiting times and thus more overhead, as illustrated by Figure 5.4.

TWINS provides read performance improvements of up to 28% over the baseline scheduling algorithms and of up to 50% over not using IOFSL. The best results were obtained for the shared file 1D strided access pattern, and gains for the shared-file contiguous pattern were also observed – up to 20% over the baseline. There is a trade-off to be observed between the induced overhead and how distributed among the servers requests are. Further analysis is required to determine how the scheduler could automatically find the best window duration. This will be the focus of future work.

Figure 5.3: Execution time of **read** requests using different window sizes for TWINS.



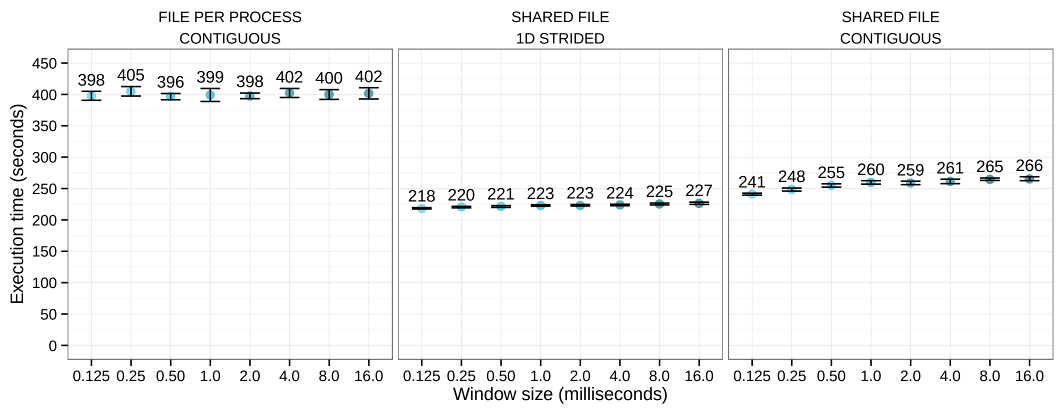
(a) Small requests (32KB)



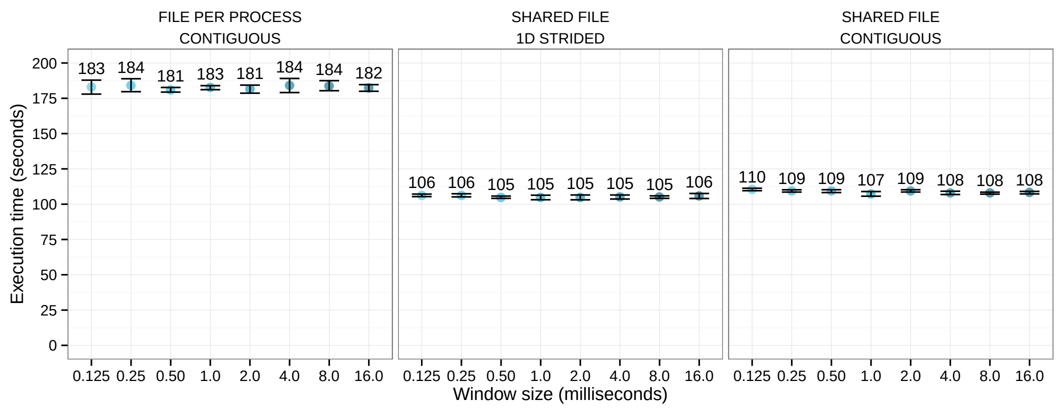
(b) Large requests (256KB)

Source: Author

Figure 5.4: Execution time of **write** requests using different window sizes for TWINS.



(a) Small requests (32KB)



(b) Large requests (256KB)

Source: Author

5.3 Aggregation Sizes and Contention

An evidence towards explaining why delaying requests at the forwarding nodes, with the TWINS scheduler, can benefit performance is presented by the aggregation metrics. By including additional delays, using fixed time windows, requests might stay longer in the algorithm's queues, thus there are more aggregations opportunities. Table 5.1 summarizes this information considering all schedulers evaluated in this work plus the best two window sizes observed for the tests with a small (32KB) request size. Nonetheless, as we have previously pointed, just combining requests is not enough to provide performance in the tested scenarios. The gains obtained with TWINS also come from coordinating accesses to the data servers, thus reducing contention.

To determine if the coordination mechanism is indeed working and helping in reducing contention when accessing the data servers, we designed an additional experiment. All the parameters were the same as the previous ones, but now we monitored the TCP

Table 5.1: Average request size (in KB) at different levels of the I/O stack for the shared file scenario with small (32KB) 1D strided and contiguous accesses.

		1D Strided (KB)		Contiguous (KB)	
		READ	**WRITE**	**READ**	**WRITE**
FIFO	Leaving I/O nodes	57.75	57.72	57.72	57.79
	Arriving at the data servers	49.87	49.23	43.52	43.66
HBRR	Leaving I/O nodes	57.77	57.68	57.77	57.69
	Arriving at the data servers	49.80	49.36	42.75	43.65
TO	Leaving I/O nodes	75.09	65.70	77.07	66.34
	Arriving at the data servers	53.69	51.41	53.37	50.98
SJF	Leaving I/O nodes	74.79	65.71	76.81	65.91
	Arriving at the data servers	53.63	51.33	53.31	50.94
MLF	Leaving I/O nodes	79.59	66.39	76.53	66.82
	Arriving at the data servers	51.17	48.31	50.44	49.22
SW (1s)	Leaving I/O nodes	73.08	64.93	72.64	64.11
	Arriving at the data servers	50.08	48.38	50.16	48.17
TWINS (1ms)	Leaving I/O nodes	119.29	87.83	106.17	78.08
	Arriving at the data servers	57.79	51.41	53.15	48.07
TWINS (8ms)	Leaving I/O nodes	149.99	95.75	162.60	85.66
	Arriving at the data servers	68.08	49.25	61.38	46.71

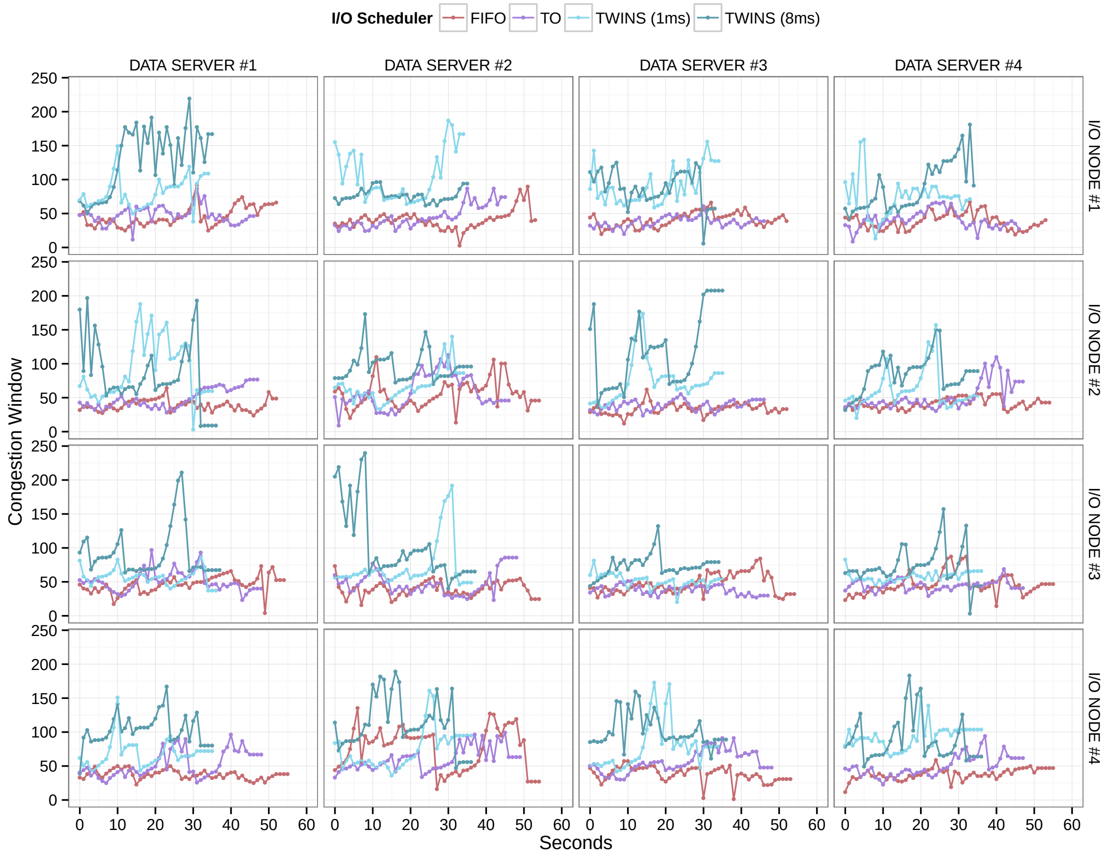
Source: Author

congestion window size. This window is maintained by the sender as an upper bound for the data communication between a sender and a receiver. The congestion window (*cwnd*) and slow start threshold (*ssthresh*) are two internal variables of the TCP congestion control mechanism, in addition to the advertised window by the receiver (*awnd*). By monitoring the congestion window parameter it is possible to determine if the connection is undergoing stress due to congestion (ALENEZI; REED, 2013).

To collect this information, we opted to use the *ss* tool available in the Linux operating system, commonly used to dump socket statistics. This tool is similar to *netstat*, however, it can display more TCP and state information than other tools. We collected metrics, in intervals of one second, during the entire execution, on each one of the four PVFS data servers. Therefore they represent the flow of requests sent (replied) by each PVFS server to the connected IOFSL server.

For this experiment, we considered only small (32KB) 1D strided read operations to a shared filed as this represent the scenario where we observed more improvements. Figure 5.5 summarizes the results, comparing the FIFO, TO, TWINS (1ms) and TWINS (8ms) schedulers. We plotted all communications between each of the four data servers

Figure 5.5: Congestion window size for small **read** 1D strided accesses to a shared file.



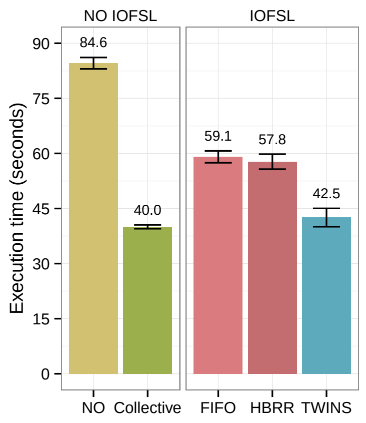
Source: Author

and the four I/O nodes. It is possible to see that by using TWINS (blue lines) we are indeed reducing contention, thus the congestion window is larger, allowing more data to flow if compared to the FIFO and TO schedulers. Furthermore, the 8ms window, that was the most suited for this scenario, has on average a higher value for the window than the 1ms alternative.

5.4 TWINS vs. Collective Operations

The traditional way of improving performance of small non-contiguous requests is to use collective I/O operations (THAKUR; GROPP; LUSK, 2002; LIU; CHEN; ZHUANG, 2013; WANG et al., 2014). Figure 5.6 compares the performance obtained by TWINS for the 1D strided access pattern with what is achieved by making the single application perform collective calls. As a reference, times obtained using IOFSL with the baseline algorithms are also presented. We can see TWINS is able to provide as much performance as the use of collective operations for the small (32KB) 1D strided read access pattern.

Figure 5.6: TWINS vs. collective I/O operations



Source: Author

It is important to notice that TWINS represents a more transparent and generic solution than MPI-IO collective operations. Because it is applied in the I/O nodes, TWINS is completely transparent to applications and I/O library generic. Therefore applications using any method to perform I/O operations, such as POSIX, can benefit from this optimization, without modifications to the source code.

To the best of our knowledge, this is the first work to propose a scheduler to the I/O forwarding layer which transparently coordinates accesses to alleviate concurrency at the PFS data servers.

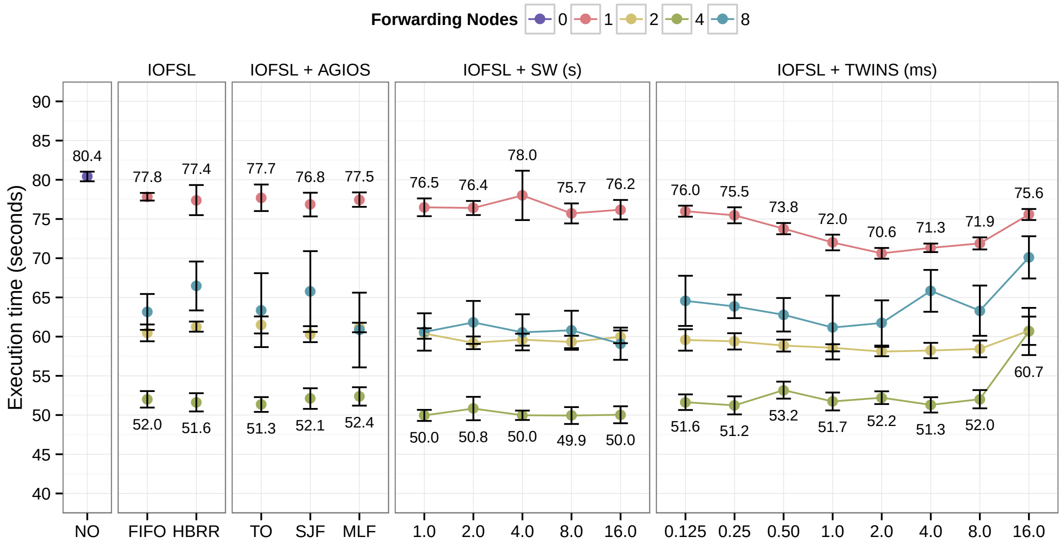
5.5 Mapping I/O Nodes

The forwarding layer deployment and configuration may also interfere with the effectiveness of the schedulers employed in this layer. Because of the huge parameter scope, so far, we have selected a few representative scenarios to conduct a more thorough investigation. However, for the sake of completeness, we also present a summary evaluation of all the previous tested algorithms, considering different ratios of clients per I/O nodes.

For instance, Figure 5.7 illustrates the impact of different numbers of forwarding nodes (0, 1, 2, 4 or 8 I/O nodes), combined with distinct requests schedulers, and considering only one common access pattern (file-per-process, 32KB contiguous requests). Parameters concerning the parallel file system and the forwarding layer configuration were left untouched.

Results are grouped into five blocks. The first one represents a scenario where the

Figure 5.7: Overview of the impact of the number of I/O nodes using distinct schedulers for the small file-per-process access pattern.



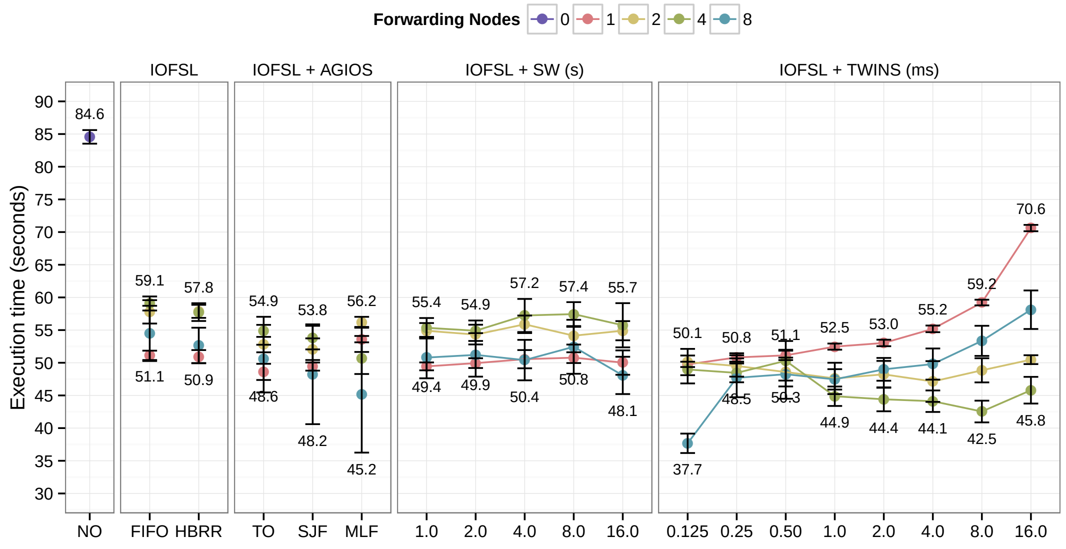
Source: Author

forwarding layer is not present, i.e. clients directly access the file system data servers. The second group illustrates two schedulers from the IOFSL forwarding framework: FIFO and HBRR. On the third group, we integrated AGIOS scheduling library into IOFSL to test other schedulers that were demonstrated to provide performance improvements in the parallel file system data servers: TO, SJF, and MLF. The fourth group represents the SW scheduler with distinct window sizes (in seconds). Finally, the last block depicts different window sizes for TWINS (in milliseconds), our best solution. It is important to notice that the y-axis of these plots does not start at zero to better visualize the differences. We have also included lines in the window based schedulers to help visualize the trend.

We can see that just by changing the number of I/O nodes and the scheduler, the I/O performance is significantly improved or worsened, without any additional modification to the executing application. Furthermore, if we consider an extra parameter, such as the window size of time-window based schedulers (represented by IOFSL + SW and IOFSL + TWINS) the results are impacted by that configuration as well.

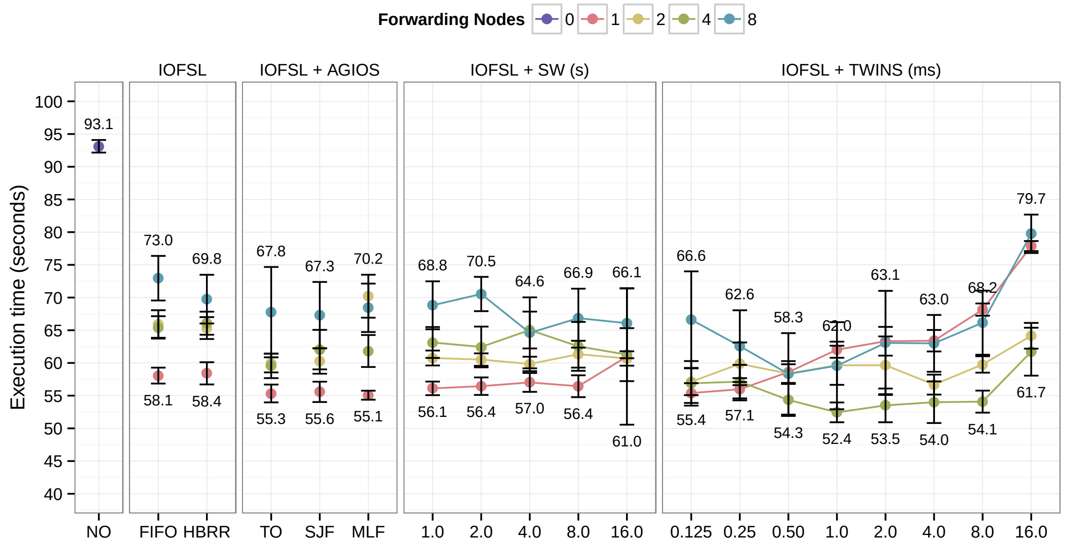
Figure 5.8 summarizes the results for small 1D strided requests to a common shared file. The difference between using a distinct number of I/O nodes with each scheduler is not as prominent as the file-per-process scenario, but it is still relevant. For instance, observe that FIFO and HBRR demonstrate better results (lower is better) using only one I/O node (ratio of $1:32$). However, if four I/O nodes are used (ratio of $1:8$), TWINS with a window size greater than one millisecond is more suited. Furthermore, for eight

Figure 5.8: Overview of the impact of the number of I/O nodes using distinct schedulers for the small 1D strided access to a shared file.



Source: Author

Figure 5.9: Overview of the impact of the number of I/O nodes using distinct schedulers for the small contiguous access to a shared file.



Source: Author

I/O nodes (ratio of $1:4$) TWINS with a really small window size (0.125ms) presents the best result.

To complete our analysis, Figure 5.9 brings the results for small contiguous accesses to a shared file. A similar conclusion can be drawn here. FIFO and HBRR are the best alternatives when only one I/O node is connected to the 32 clients, and TWINS is better suited when 4 I/O nodes (ratio of $1:8$) are in place.

These results demonstrate the complexity of proper configuring the forwarding layer, without considering other levels of the I/O stack. Furthermore, manually tuning and finding the near-optimal configuration for each application is a time consuming and somewhat an elusive job. Moreover, no single scheduler or configuration is able to improve performance for all scenarios and access patterns. That alone justifies the increasing research effort in seeking new schedulers and alternatives to improve I/O performance. Techniques for automatic tuning the HPC IO stack could assist in this task. However, this is out of the scope of this dissertation and it is suggested as future work.

5.6 Multiple Applications Scenario

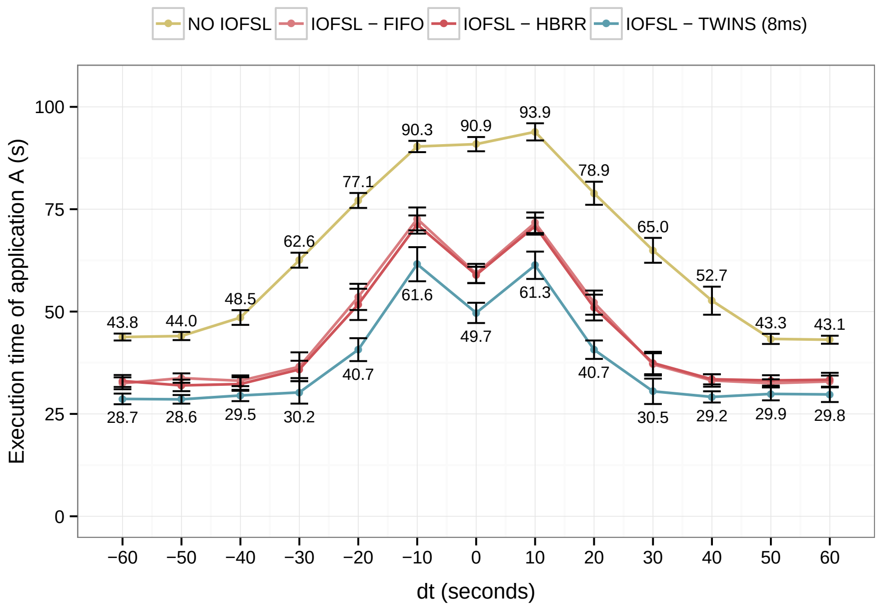
Albeit applications run on dedicated nodes on large scale clusters or supercomputers, they do share the access to the file system servers and the forwarding nodes, if the latter are present. Since our scheduling algorithm was designed for this particular layer, we have to test it on a multi-application scenarios. To confirm our algorithm's performance, we have conducted additional experiments using the *Ifer* micro benchmark[1]. *Ifer* is a benchmark similar to IOR that starts by splitting its set of processes into groups running on two different sets of nodes. Each group of processes executes a series of MPI-IO operations, simulating two applications accessing the shared file system in contention (YILDIZ, 2016). *Ifer* measures the time taken by each group of processes to complete its set of I/O operations.

Ifer was created to issue only write requests, however we have modified it to perform read operations to previously created files. For these experiments, each application has 64 processes and presents the shared file 1D strided access pattern. Figure 5.10 presents execution times of the first application (A) in the multi-application experiments. The lines represent different options: not using the forwarding layer, using it with the baseline algorithms, and using it with TWINS. The x-axis represents the time difference between start time for applications A and B: when dt is 0 both start at the same time, positive dt means A starts before B and negative dt means B starts first.

We can see the I/O forwarding layer also improves read performance for the multiple applications scenario – up to 35% with the baseline algorithms. The interference experienced by the application is decreased by FIFO and HBRR up to 25%, except when applications start with a 10 seconds difference, as depicted by Figure 5.11. The inter-

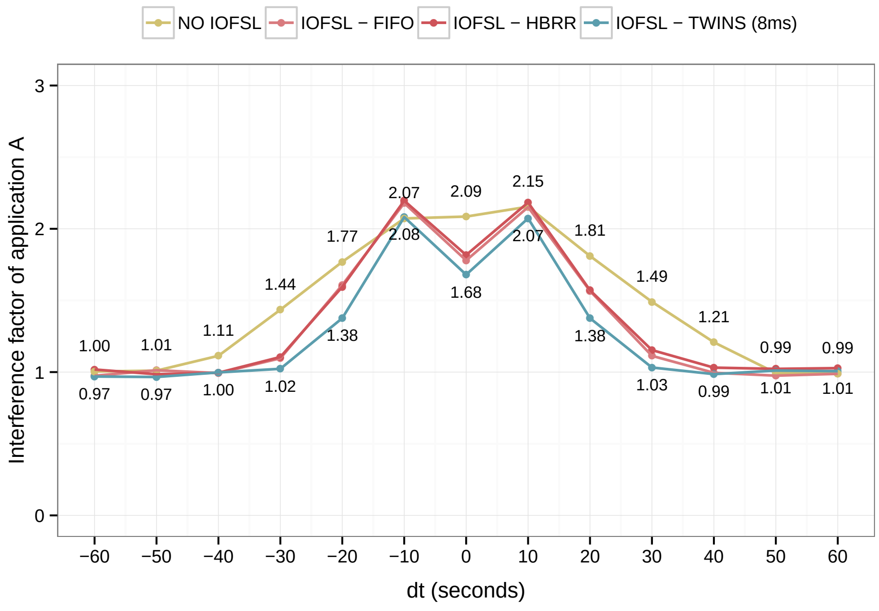
[1]https://team.inria.fr/kerdata/ifer-microbenchmark-for-studying-the-cross-application-io-interference/

Figure 5.10: Execution time for an application under contention caused by another concurrent running application.



Source: Author

Figure 5.11: Interference factor between concurrent running applications using distinct scheduling algorithms.



Source: Author

ference factor is calculated as the ratio between the execution time of the application under contention and the time of the application executing by itself. TWINS improves performance up to 16% over FIFO and HBRR, and up to 45% over not using IOFSL. Interference is further decreased by using TWINS – up to 12% over the baseline algorithms and up to 31% over not using intermediate I/O nodes.

5.7 Conclusions

This chapter presented a thorough evaluation of our proposed scheduler for the I/O forwarding layer. Better results were observed for the shared file scenario. For 1D strided access improvements were of 22.30% over using HBRR and 18.93% over using SW. For the contiguous accesses, improvements were of 20.63% over HBRR and 16.92% over SW. For write requests, TWINS was not able to provide improvements in this scale, similarly to the other tested alternatives.

We have also investigated different values for window size parameter. Best results for small (32KB) 1D strided requests to a single shared file, with read operations, were observed in a 8ms window. However, for the shared-file tests 1ms time window appears to be more appropriate. Furthermore, degradation is only observed in the shared file scenario if the window is too large, which imposes additional waiting times and thus more overhead.

By collecting additional metrics such as request aggregation size and congestion window, on the PFS data servers, we were able to correlate the results and explain the performance improvements observed with our solution. Additionally, TWINS also improved the performance of read requests up to 16% over FIFO and HBRR for the multiple applications scenario. Interference among concurrent applications was also further decreased by up to 12%.

6 CONCLUSIONS

This work presented a study of the I/O scheduling technique at the I/O forwarding layer. We evaluated five algorithms, two of which were previously applied to this layer – FIFO and HBRR – and three that have demonstrated performance improvements when applied to the parallel file system data servers. This evaluation has shown that no statistically significant difference was observed between the simple FIFO and the more complex HBRR algorithm. Furthermore, the solutions implemented in the AGIOS scheduling library also did not significantly improve performance. However, it was possible to see, by measuring the average size of requests at different points of the I/O stack, that those schedulers – TO, SJF, and MLF – indeed were able to improve aggregations. This demonstrates that they are only partially effective and other factors were responsible for harming performance.

We believed the lack of a coordination mechanism between the I/O nodes, when accessing the data servers, might still be directly affecting performance. To confirm our hypothesis, we proposed a new scheduling algorithm for the I/O forwarding layer named *Server Window* (SW). Although SW grouped the requests by the destination server, they ended up being merged on the dispatch queue. Hence, they were sent together to the PFS data servers, without focusing accesses to one data server at a time.

As an effort to enforce the desired coordination effect, we proposed *Server Time WINdows* (TWINS). Our algorithm uses multiple requests queues – one per PFS data server – and fixed time windows to coordinate the I/O nodes' accesses and decrease contention. To the best of our knowledge, ours is the first work to apply such a technique to the forwarding layer of the HPC I/O stack. Since our proposed schedulers were implemented in the AGIOS library, they remain generic and can be employed by other forwarding frameworks or even by the file system servers.

TWINS results have shown performance improvements for shared-file read access patterns of up to 28% over the state-of-the-art algorithms. Compared to not using I/O forwarding nodes, the gains were of up to 50%. Improvements were also shown for a multi-application scenario, accompanied by a decrease in interference. Moreover, even for situations where TWINS is not able tpo improve performance, it does not necessarily harm it.

By analyzing the congestion window size at each data server we succeeded in correlating the improvements observed due to congestion reduction. Additionally, the performance obtained by TWINS for the 1D strided read access pattern was comparable to what can be achieved by making the application use collective operations. We compared our results with collective I/O because this is a popular alternative applied to improve the performance of the applications with 1D strided access patterns. Nevertheless, opposed to collective I/O, our proposal is completely transparent to applications and library independent.

6.1 Future Work

Future work will focus on proposing an automatic mechanism to tune the TWINS' time window duration parameter based on the observed access pattern. Moreover, we plan on expanding our evaluation of TWINS and other schedulers considering additional parameters and factors of other levels of the I/O stack that may directly or indirectly affect the schedulers' performance. We also plan on increasing the number of servers and forwarding nodes to test scenarios where the algorithm's overhead may impact performance. In those scenarios, an idea would be to focus access to a group of servers, instead of treating them separately.

Furthermore, we plan on proposing automatic tuning mechanisms to adjust the stack's configurations, including selecting the best scheduler and tune it's parameters, to achieve good performance based on each application's access pattern and characteristics.

6.2 Publications

The following papers were produced during this dissertation. We first list the ones related to the this work and to the parallel I/O research field, including those submitted and under review:

- **BEZ, J. L**; BOITO, F. Z.; SCHNORR, L. M.; NAVAUX, P. O. A.; MEHAUT, J. TWINS: Server Access Coordination in the I/O Forwarding Layer. In: Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2017 (*Accepted*).

- MACHADO, V. R.; RAMPON, N. G.; BRAGA, A. B.; **BEZ, J. L.**; BOITO, F. Z.; KASSICK, R. V.; PADOIN, E. L.; DIAZ, J.; MAHUT, J.; NAVAUX, P. O. A.;

Towards Energy-Efficient Storage Servers. In: The 32nd ACM Symposium On Applied Computing, 2017 (*Accepted*).

- BOITO, F. Z.; **BEZ, J. L**; DUPROS, F.; DANTAS, M.; NAVAUX, P. O. A.; AOCHI, HIDEO. High Performance I/O for Seismic Wave Propagation Simulations. In: Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2017 (*Accepted*).
- BOITO, F. Z.; INACIO, E. C.; **BEZ, J. L.**; NAVAUX, P. O. A.; DANTAS, M. A. R; DENNEULIN, YVES. A Checkpoint of Parallel I/O Research. In: ACM Computing Surveys, 2016 (*Submitted*).
- PAVAN, P. J.; LORENZONI, R. K.; **BEZ, J. L.**; BOITO, F. Z.; PADOIN, E. L.; NAVAUX, P. O. A.; MEHAUT, J. Eficiência Energética e Desempenho de E/S com Arquiteturas de Baixa Potência. In: WSCAD 2016 - XVII Simpósio em Sistemas Computacionais de Alto Desempenho, 2016, Aracaju.
- **BEZ, J. L.**; BOITO, F. Z.; SCHNORR, L. M.; NAVAUX, P. O. A. Escalonamento de I/O em Servidores de Encaminhamento. In: XVI Escola Regional de Alto Desempenho (ERAD/RS), 2016, São Leopoldo. Anais da XVI Escola Regional de Alto Desempenho (ERAD/RS). São Leopoldo, 2016. v. 1. p. 173-174.

  BEZ, J. L.; BOITO, F. Z.; SCHNORR, L. M.; NAVAUX, P. O. A. Coordinating Data Access at I/O Forwarding Nodes. In: WSPPD 2016 - XIV Workshop de Processamento Paralelo e Distribuído, 2016, Porto Alegre. Proceedings WSPPD 2016 - XIV Workshop de Processamento Paralelo e Distribuído. Porto Alegre, 2016. v. 1. p. 9-11.
- PAVAN, P. J.; LORENZONI, R. K.; **BEZ, J. L.**; BOITO, F. Z.; PADOIN, E. L.; NAVAUX, P. O. A. Análise de Consumo Energético e Desempenho de Operações E/S em Arquiteturas de Baixa Potência. In: WSPPD 2016 - XIV Workshop de Processamento Paralelo e Distribuído, 2016, Porto Alegre. Proceedings WSPPD 2016 - XIV Workshop de Processamento Paralelo e Distribuído. Porto Alegre, 2016. v. 1. p. 5-9.
- BRAGA, A. B.; RAMPON, N. G.; MACHADO, V. R.; **BEZ, J. L.**; BOITO, F. Z.; KASSICK, R. V.; PADOIN, E. L.; NAVAUX, P. O. A. Viability of Low-Power Architectures as Parallel File Systems. In: WSPPD 2016 - XIV Workshop de Processamento Paralelo e Distribuído, 2016, Porto Alegre. Proceedings WSPPD 2016

- XIV Workshop de Processamento Paralelo e Distribuído. Porto Alegre, 2016. v. 1. p. 27-30.

- MACHADO, V. R.; BOITO, F. Z.; KASSICK, R. V.; **BEZ, J. L.**; NAVAUX, P. O. A.; DENNEULIN, Y. Parallel Storage Devices Profiling with SeRRa. In: 14º WPerformance - XIV Workshop em Desempenho de Sistemas Computacionais e de Comunicação, 2015, Recife. 14º WPerformance - XIV Workshop em Desempenho de Sistemas Computacionais e de Comunicação, 2015.

- **BEZ, J. L.**; BOITO, F. Z. ; KASSICK, R. V. ; MACHADO, V. R. ; NAVAUX, P. O. A. . Faster Storage Devices Profiling with Parallel SeRRa. In: WSPPD 2015 - XIII Workshop de Processamento Paralelo e Distribuído, 2015, Porto Alegre. Proceedings WSPPD 2015 - XIII Workshop de Processamento Paralelo e Distribuído, 2015. v. 1. p. 33-36.

The following papers were also published but are not directly related to the parallel I/O field, though they are still relevant to the HPC research field:

- **BEZ, J. L**; BERNART, E. E.; SANTOS, F. F.; SCHNORR, L. M.; NAVAUX, P. O. A. Performance and Energy Efficiency Analysis of HPC Physics Simulation Applications in a Cluster of ARM Processors. In: Concurrency and Computation: Practice and Experience, 2016.

- **BEZ, J. L.**; BERNART, E. E.; SANTOS, F. F.; SCHNORR, L. M.; NAVAUX, P. O. A. Análise da Eficiência Energética de uma Aplicação HPC de Geofísica em um Cluster de Baixo Consumo. In: WSCAD 2015 - XVI Simpósio em Sistemas Computacionais de Alto Desempenho, 2015, Florianópolis. Anais da 16a Edição do Simpósio em Sistemas Computacionais de Alto Desempenho, 2015. v. 1. p. 228-239.

- **BEZ, J. L.**; SCHNORR, L. M.; NAVAUX, P. O. A. Characterizing Anomalies of a Multicore ARMv7 Cluster with Parallel N-Body Simulations. In: 10th Workshop on Applications for Multi-Core Architectures, 2015, Florianópolis. Proceedings 10th Workshop on Applications for Multi-Core Architectures, 2015. v. 1. p. 27-32.

- DOS ANJOS, JULIO C.S.; ASSUNCAO, MARCOS D.; **BEZ, JEAN**; GEYER, CLAUDIO; DE FREITAS, EDISON PIGNATON; CARISSIMI, ALEXANDRE; COSTA, JOAO PAULO C. L.; FEDAK, GILLES; FREITAG, FELIX; MARKL,

VOLKER; FERGUS, PAUL; PEREIRA, RUBEM. SMART: An Application Framework for Real Time Big Data Analysis on Heterogeneous Cloud Environments. In: 2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing, 2015, LIVERPOOL. CIT/IUCC/DASC/PICOM. p. 199.

REFERENCES

ABBASI, H. et al. DataStager: Scalable data staging services for petascale applications. In: **Proceedings...** New York, NY, USA: ACM, 2009. (HPDC '09), p. 39–48. ISBN 978-1-60558-587-1. Available from Internet: <http://doi.acm.org/10.1145/1551609.1551618>.

ALENEZI, M.; REED, M. J. Denial of service detection through TCP congestion window analysis. In: **Proceedings...** [S.l.: s.n.], 2013. (WorldCIS-2013), p. 145–150.

ALI, N. et al. Scalable I/O forwarding framework for high-performance computing systems. In: IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING AND WORKSHOPS, 2009, Berkeley, USA. **Proceedings...** [S.l.]: IEEE, 2009. p. 1–10.

ALMÁSI, G. et al. An overview of the Blue Gene/L system software organization. In: EURO-PAR 2003 CONFERENCE, LECTURE NOTES IN COMPUTER SCIENCE, 2003. **Proceedings...** [S.l.]: Springer-Verlag, 2003. p. 543–555.

ARGONNE, L. C. F. **Aurora Supercomputer**. 2016. <http://aurora.alcf.anl.gov/>. Accessed: October 2016.

BALOUEK, D. et al. Adding virtualization capabilities to the Grid'5000 testbed, In: IVANOV, I. ET AL. **Cloud Computing and Services Science**. [S.l.]: Springer International Publishing, 2013. (Communications in Computer and Information Science), p. 3–20. ISBN 978-3-319-04518-4.

BOITO, F. Z. **Transversal I/O Scheduling for Parallel File Systems: from Applications to Devices**. Thesis (PhD) — PPGC - Federal University of Rio Grande do Sul, 2015.

BOITO, F. Z. et al. Towards fast profiling of storage devices regarding access sequentiality. In: 30TH ANNUAL ACM SYMPOSIUM ON APPLIED COMPUTING, 2015, Salamanca, Spain. **Proceedings...** ACM, 2015. (SAC '15), p. 2015–2020. ISBN 978-1-4503-3196-8. Available from Internet: <http://doi.acm.org/10.1145/2695664.2695701>.

BOITO, F. Z. et al. AGIOS: Application-guided I/O scheduling for parallel file systems. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED SYSTEMS (ICPADS), 2013, Seoul. **Proceedings...** [S.l.]: IEEE, 2013. p. 43–50. ISSN 1521-9097.

BOITO, F. Z. et al. Automatic I/O scheduling algorithm selection for parallel file systems. **Concurrency and Computation: Practice and Experience**, 2015. ISSN 1532-0634. Available from Internet: <http://dx.doi.org/10.1002/cpe.3606>.

CORBETT, P. et al. **Overview Of The MPI-IO Parallel I/O Interface**. 1995.

DECLERCK, T. et al. Cori - a system to support data-intensive computing. In: **Proceedings...** [s.n.], 2016. p. 8. Available from Internet: <http://www.fujitsu.com/global/documents/about/resources/publications/fstj/archives/vol48-3/paper02.pdf>.

DELL. **OrangeFS Reference Architecture**. [S.l.], 2012. Available from Internet: <http://i.dell.com/sites/doccontent/business/solutions/engineering-docs/en/Documents/orange-fs-reference-architecture.pdf>.

DOE. **The Opportunities and Challenges of Exascale Computing**. [S.l.], 2010.

HE, J. et al. I/O acceleration with pattern detection. In: 22ND INTERNATIONAL SYMPOSIUM ON HIGH-PERFORMANCE PARALLEL AND DISTRIBUTED COMPUTING, 2013, New York, New York, USA. **Proceedings...** ACM, 2013. (HPDC '13), p. 25–36. ISBN 978-1-4503-1910-2. Available from Internet: <http://doi.acm.org/10.1145/2462902.2462909>.

ISAILA, F. et al. Design and evaluation of multiple-level data staging for blue gene systems. **Parallel and Distributed Systems, IEEE Transactions on**, IEEE Computer Society, v. 22, n. 6, p. 946–959, 2011.

ISKRA, K. et al. ZOID: I/O forwarding infrastructure for petascale architectures. In: 13TH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING. **Proceedings...** [S.l.], 2008. p. 153–162.

KUO, C.-S. et al. How file access patterns influence interference among cluster applications. In: 2014 IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING (CLUSTER), 2014, Madrid, ES. **Proceedings...** [S.l.]: IEEE, 2014. p. 185–193. ISSN 1552-5244.

LANL. **Los Alamos National Lab MPI-IO Test, User's Guide**. 2006.

LARREA, V. G. V. et al. A more realistic way of stressing the end-to-end I/O system. In: CRAY USER GROUP MEETING, 2015, Chicago, IL. **Proceedings...** [S.l.], 2015.

LATHAM, R. et al. A next-generation parallel file system for linux clusters. **LinuxWorld Magazine**, v. 2, n. 1, January 2004.

LEBRE, A. et al. I/O scheduling service for multi-application clusters. In: IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING, 2006, Barcelona. **Proceedings...** [S.l.]: IEEE, 2006. p. 1–10. ISSN 1552-5244.

LEE, C.; YANG, M.; AYDT, R. Netcdf-4 performance report. 2008. Available from Internet: <https://www.hdfgroup.org/pubs/papers/2008-06_netcdf4_perf_report.pdf>.

LIU, J.; CHEN, Y.; ZHUANG, Y. Hierarchical I/O scheduling for collective I/O. In: **Proceedings...** [S.l.: s.n.], 2013. (CCGRID'13), p. 211–218.

LIU, Q. et al. Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks. **Concurrency and Computation: Practice and Experience**, v. 26, n. 7, p. 1453–1473, 2014. Available from Internet: <http://dx.doi.org/10.1002/cpe.3125>.

LIU, Z. et al. Profiling and improving I/O performance of a large-scale climate scientific application. In: 22ND INTERNATIONAL CONFERENCE ON COMPUTER COMMUNICATIONS AND NETWORKS (ICCCN), 2013, Nasssau. **Proceedings...** [S.l.]: IEEE, 2013. p. 1–7.

LOFSTEAD, J. et al. Six degrees of scientific data: Reading patterns for extreme scale science IO. In: 20TH INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING, 2011, San Jose, California, USA. **Proceedings...** ACM, 2011. (HPDC '11), p. 49–60. ISBN 978-1-4503-0552-5. Available from Internet: <http://doi.acm.org/10.1145/1996130.1996139>.

MIYAZAKI, H. et al. **Overview of the K computer system**. [S.l.], 2012. 48 p. Available from Internet: <http://www.fujitsu.com/global/documents/about/resources/publications/fstj/archives/vol48-3/paper02.pdf>.

NISAR, A.; LIAO, W.-k.; CHOUDHARY, A. Scaling parallel I/O performance through I/O delegate and caching system. In: **Proceedings...** [S.l.: s.n.], 2008. (SC'08), p. 1–12. ISSN 2167-4329.

NSCCWX, N. S. C. **Sunway TaihuLight Supercomputer**. 2016. <http://www.nsccwx.cn/wxcyw/>. Accessed: December 2016.

OHTA, K. et al. Optimization techniques at the I/O forwarding layer. In: IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING, 2009, Heraklion, Crete. **Proceedings...** [S.l.]: IEEE, 2010. p. 312–321.

PRABHAT; KOZIOL, Q. **High Performance Parallel I/O**. 1st. ed. [S.l.]: Chapman & Hall/CRC, 2014. ISBN 1466582340, 9781466582347.

QIAN, Y. et al. A novel network request scheduler for a large scale storage system. **Computer Science - Research and Development**, Springer-Verlag, v. 23, n. 3–4, p. 143–148, 2009. ISSN 1865-2034. Available from Internet: <http://dx.doi.org/10.1007/s00450-009-0073-9>.

REN, K. et al. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In: **Proceedings...** Piscataway, NJ, USA: IEEE Press, 2014. (SC'14), p. 237–248. ISBN 978-1-4799-5500-8. Available from Internet: <http://dx.doi.org/10.1109/SC.2014.25>.

ROSARIO, J. M. del; BORDAWEKAR, R.; CHOUDHARY, A. Improved parallel i/o via a two-phase run-time access strategy. **ACM SIGARCH Computer Architecture News**, ACM, New York, NY, USA, v. 21, n. 5, p. 31–38, dec. 1993. ISSN 0163-5964. Available from Internet: <http://doi.acm.org/10.1145/165660.165667>.

SCHMUCK, F.; HASKIN, R. GPFS: A shared-disk file system for large computing clusters. In: 1ST USENIX CONFERENCE ON FILE AND STORAGE TECHNOLOGIES, 2002, Monterey, CA. **Proceedings...** USENIX Association, 2002. (FAST '02). Available from Internet: <http://dl.acm.org/citation.cfm?id=1083323.1083349>.

SONG, H. et al. Server-side I/O coordination for parallel file systems. In: **Proceedings...** [S.l.: s.n.], 2011. (SC '11), p. 1–11. ISSN 2167-4329.

STENDER, J. et al. Striping without sacrifices: Maintaining posix semantics in a parallel file system. In: **Proceedings...** Berkeley, CA, USA: USENIX Association, 2008. (LASCO'08), p. 6:1–6:8. Available from Internet: <http://dl.acm.org/citation.cfm?id=1411725.1411731>.

SUGIYAMA, S.; WALLACE, D. Cray dvs: Data virtualization service. 2008.

SUN. **High-Performance Storage Architecture and Scalable Cluster File System**. [S.l.], 2007. Available from Internet: <http://www.csee.ogi.edu/~zak/cs506-pslc/lustrefilesystem.pdf>.

THAKUR, R.; GROPP, W.; LUSK, E. Optimizing noncontiguous accesses in mpi-io. **Parallel Computing**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 28, n. 1, p. 83–105, jan. 2002. ISSN 0167-8191. Available from Internet: <http://dx.doi.org/10.1016/S0167-8191(01)00129-6>.

The HDF Group. **Hierarchical Data Format, version 5**. 1997–2016. /HDF5/.

VISHWANATH, V. et al. Accelerating I/O forwarding in IBM blue gene/p systems. In: 2010 ACM/IEEE INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS, 2010, New Orleans, USA. **Proceedings...** [S.l.]: IEEE Computer Society, 2010. (SC'10), p. 1–10. ISBN 978-1-4244-7559-9.

VISHWANATH, V. et al. Topology-aware data movement and staging for I/O acceleration on blue gene/p supercomputing systems. In: 2011 INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS, 2011, Seattle, Washington. **Proceedings...** ACM, 2011. (SC '11), p. 19:1–19:11. ISBN 978-1-4503-0771-0. Available from Internet: <http://doi.acm.org/10.1145/2063384.2063409>.

WANG, Z. et al. Iteration based collective I/O strategy for parallel I/O systems. In: **Proceedings...** [S.l.: s.n.], 2014. (CCGrid'14), p. 287–294.

WELCH, B. et al. Scalable performance of the panasas parallel file system. In: 6TH USENIX CONFERENCE ON FILE AND STORAGE TECHNOLOGIES, 2008, San Jose, California. **Proceedings...** USENIX Association, 2008. (FAST'08), p. 2:1–2:17. Available from Internet: <http://dl.acm.org/citation.cfm?id=1364813.1364815>.

XU, W. et al. Hybrid hierarchy storage system in MilkyWay-2 supercomputer. **Frontiers of Computer Science**, Higher Education Press, v. 8, n. 3, p. 367–377, 2014. ISSN 2095-2228. Available from Internet: <http://dx.doi.org/10.1007/s11704-014-3499-6>.

YILDIZ, O. **IFER: MicroBenchmark for Studying the Cross-Application I/O Interference**. 2016. <https://team.inria.fr/kerdata/ifer-microbenchmark-for-studying-the-cross-application-io-interference/>. Accessed: September 2016.

YIN, Y. et al. Pattern-direct and layout-aware replication scheme for parallel i/o systems. In: 2013 IEEE 27TH INTERNATIONAL SYMPOSIUM ON PARALLEL DISTRIBUTED PROCESSING (IPDPS), 2013, Boston, MA. **Proceedings...** [S.l.]: IEEE, 2013. p. 345–356. ISSN 1530-2075.

ZHANG, X.; DAVIS, K.; JIANG, S. IOrchestrator: Improving the performance of multi-node I/O systems via inter-server coordination. In: **Proceedings...** [S.l.: s.n.], 2010. p. 1–11. ISSN 2167-4329.

ZIMMER, C.; GUPTA, S.; LARREA, V. G. V. Finally, a way to measure frontend i/o performance. In: CRAY USER GROUP MEETING, 2016, London, UK. **Proceedings...** [S.l.], 2016.