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

FRANCIELI ZANON BOITO

Transversal I/O Scheduling for Parallel File Systems: from Applications to Devices

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

Advisor: Prof. Dr. Philippe Olivier Alexandre Navaux
Advisor: Prof. Dr. Yves Denneulin

Porto Alegre
March 2015

CIP – CATALOGING-IN-PUBLICATION

Zanon Boito, Francieli

Transversal I/O Scheduling for Parallel File Systems: from Applications to Devices / Francieli Zanon Boito. – Porto Alegre: PPGC da UFRGS, 2015.

171 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2015. Advisor: Philippe Olivier Alexandre Navaux; Advisor: Yves Denneulin.

1. I/O Scheduling. 2. Parallel File Systems. 3. High Performance Computing. I. Navaux, Philippe Olivier Alexandre. II. Denneulin, Yves. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Pós-Graduação: Prof. Vladimir Pinheiro do Nascimento
Diretor do Instituto de Informática: Prof. Luis da Cunha Lamb
Coordenador do PPGC: Prof. Luigi Carro
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

ABSTRACT

This thesis focuses on I/O scheduling as a tool to improve I/O performance on parallel file systems by alleviating interference effects. It is usual for High Performance Computing (HPC) systems to provide a shared storage infrastructure for applications. In this situation, when multiple applications are concurrently accessing the shared parallel file system, their accesses will affect each other, compromising I/O optimization techniques' efficacy.

We have conducted an extensive performance evaluation of five scheduling algorithms at a parallel file system's data servers. Experiments were executed on different platforms and under different access patterns. Results indicate that schedulers' results are affected by applications' access patterns, since it is important for the performance improvement obtained through a scheduling algorithm to surpass its overhead. At the same time, schedulers' results are affected by the underlying I/O system characteristics - especially by storage devices. Different devices present different levels of sensitivity to accesses' sequentiality and size, impacting on how much performance is improved through I/O scheduling.

For these reasons, this thesis main objective is *to provide I/O scheduling with double adaptivity*: to applications and devices. We obtain information about applications' access patterns through trace files, obtained from previous executions. We have applied machine learning to build a classifier capable of identifying access patterns' spatiality and requests size aspects from streams of previous requests. Furthermore, we proposed an approach to efficiently obtain the sequential to random throughput ratio metric for storage devices by running benchmarks for a subset of the parameters and estimating the remaining through linear regressions.

We use this information on applications' and storage devices' characteristics to decide the best fit in scheduling algorithm though a decision tree. Our approach improves performance by up to 75% over an approach that uses the same scheduling algorithm to all situations, without adaptability. Moreover, our approach improves performance for up to 64% more situations, and decreases performance for up to 89% less situations. Our results evidence that both aspects - applications and storage devices - are essential for making good scheduling choices. Moreover, despite the fact that there is no scheduling algorithm able to provide performance gains for all situations, we show that through double adaptivity it is possible to apply I/O scheduling techniques to improve performance, avoiding situations where it would lead to performance impairment.

Keywords: I/O Scheduling. Parallel File Systems. High Performance Computing.

Escalonamento de E/S Transversal para Sistemas de Arquivos Paralelos: das Aplicações aos Dispositivos

RESUMO

Esta tese se concentra no escalonamento de operações de entrada e saída (E/S) como uma solução para melhorar o desempenho de sistemas de arquivos paralelos, aleviando os efeitos da interferência. É usual que sistemas de computação de alto desempenho (HPC) ofereçam uma infraestrutura compartilhada de armazenamento para as aplicações. Nessa situação, em que múltiplas aplicações acessam o sistema de arquivos compartilhado de forma concorrente, os acessos das aplicações causarão interferência uns nos outros, comprometendo a eficácia de técnicas para otimização de E/S.

Uma avaliação extensiva de desempenho foi conduzida, abordando cinco algoritmos de escalonamento trabalhando nos servidores de dados de um sistema de arquivos paralelo. Foram executados experimentos em diferentes plataformas e sob diferentes padrões de acesso. Os resultados indicam que os resultados obtidos pelos escalonadores são afetados pelo padrão de acesso das aplicações, já que é importante que o ganho de desempenho provido por um algoritmo de escalonamento ultrapasse o seu sobrecusto. Ao mesmo tempo, os resultados do escalonamento são afetados pelas características do subsistema local de E/S - especialmente pelos dispositivos de armazenamento. Dispositivos diferentes apresentam variados níveis de sensibilidade à sequencialidade dos acessos e ao seu tamanho, afetando o quanto técnicas de escalonamento de E/S são capazes de aumentar o desempenho.

Por esses motivos, o principal objetivo desta tese é *prover escalonamento de E/S com dupla adaptabilidade*: às aplicações e aos dispositivos. Informações sobre o padrão de acesso das aplicações são obtidas através de arquivos de rastro, vindos de execuções anteriores. Aprendizado de máquina foi aplicado para construir um classificador capaz de identificar os aspectos espacialidade e tamanho de requisição dos padrões de acesso através de fluxos de requisições anteriores. Além disso, foi proposta uma técnica para obter eficientemente a razão entre acessos sequenciais e aleatórios para dispositivos de armazenamento, executando testes para apenas um subconjunto dos parâmetros e estimando os demais através de regressões lineares.

Essas informações sobre características de aplicações e dispositivos de armazenamento são usadas para decidir a melhor escolha em algoritmo de escalonamento através de uma árvore de decisão. A abordagem proposta aumenta o desempenho em até 75% sobre uma abordagem que usa o mesmo algoritmo para todas as situações, sem adaptabilidade. Além disso, essa técnica

melhora o desempenho para até 64% mais situações, e causa perdas de desempenho em até 89% menos situações. Os resultados obtidos evidenciam que ambos aspectos - aplicações e dispositivos de armazenamento - são essenciais para boas decisões de escalonamento. Adicionalmente, apesar do fato de não haver algoritmo de escalonamento capaz de prover ganhos de desempenho para todas as situações, esse trabalho mostra que através da dupla adaptabilidade é possível aplicar técnicas de escalonamento de E/S para melhorar o desempenho, evitando situações em que essas técnicas prejudicariam o desempenho.

Palavras-chave: Escalonamento de E/S, Sistemas de Arquivos Paralelos, Computação de Alto Desempenho.

LIST OF ABBREVIATIONS AND ACRONYMS

AGIOS	Application-Guided I/O Scheduler
ANSI	American National Standards Institute
API	Application Programming Interface
ASATD	Average Stripe Access Time Difference
DFS	Distributed File System
dNFSp	Distributed NFSp
FCFS	First Come First Served
FTL	Flash Translation Layer
FUSE	File System in User Space
GPFS	General Parallel File System
GPPD	Parallel and Distributed Processing Group
HDD	Hard-disk Drive
HDF5	Hierarchical Data Format
HDFS	Hadoop Distributed File System
HPC	High Performance Computing
I/O	Input/Output
IBM	International Business Machines
IOD	I/O Daemon
LBA	Logical Block Addressing
LBN	Logical Block Number
LDDF	Lowest Destination Degree First
LICIA	International Laboratory in High Performance and Ambient Informatics
LIG	Grenoble Informatics Laboratory
ML	Machine Learning

MLC	Multi-Level Cell
MLF	Multilevel Feedback
MPI	Message Passing Interface
MPP	Massively Parallel Processing
MSTII	Mathematics, Information Sciences and Technologies, and Computer Science
NFS	Network File System
NFSp	Parallel NFS
NRS	Network Request Scheduler
PFS	Parallel File System
PGAS	Parallel Gather-Arrange-Scatter file system
POSIX	Portable Operating System Interface
PVFS	Parallel Virtual File System
RAID	Redundant Array of Independent Disks
RPM	Revolutions Per Minute
SeRRa	Sequential to Random Ratio Profiler
SJF	Shortest Job First
SLC	Single-Level Cell
SSD	Solid-state Drive
SSTF	Shortest Seek Time First
SWTF	Shortest Wait Time First
UFRGS	Federal University of Rio Grande do Sul
VFS	Virtual File System

LIST OF FIGURES

Figure 1.1 Different levels of concurrency on the access to a parallel file system. 16

Figure 2.1 Logical components involved in performing I/O to parallel file systems.............. 21
Figure 2.2 Striping of a file of size $2 \times N$ among N servers starting by Server 1.................... 23
Figure 2.3 A client generates a write request of 320KB. Stripe size is 64KB and the maximum PFS transmission size is 32KB. 24
Figure 2.4 High-level overview of the local I/O stack. 25
Figure 2.5 Simplified view of SSD's flash packages. 27
Figure 2.6 One application with two processes accesses a file. Each process presents a 1-D strided local access pattern. 30
Figure 2.7 Interference on concurrent accesses to a PFS server. 38
Figure 2.8 Data path between applications and parallel file systems' servers. 39

Figure 3.1 Four examples of use for AGIOS. 44
Figure 3.2 A request's path to a PFS server with AGIOS.. 45
Figure 3.3 dNFSp configuration with 4 servers and 1 metadata server.................................. 53
Figure 3.4 Spatial locality aspect with a shared file.. 54
Figure 3.5 Results with TO over not using AGIOS in the Pastel cluster - tests with 16 processes. 57
Figure 3.6 Results with a single application and the shared file approach in the Graphene cluster (tests with 8 processes)... 58
Figure 3.7 Results with a single application and the shared file approach in the Edel cluster (tests with 16 processes)... 59
Figure 3.8 Results with a single application and the file per process approach in Pastel with 32 processes 62
Figure 3.9 Results with four applications and the file per process approach in Suno with 32 processes. 64

Figure 4.1 AGIOS' modules and trace generation.. 69
Figure 4.2 Scheduling with predicted future requests... 74
Figure 4.3 Results with the AGIOS's Prediction Module... 75
Figure 4.4 Increased average aggregation size with the Prediction Module for AGIOS. 76
Figure 4.5 Overhead caused by trace generation on AGIOS (MLF scheduling algorithm) with 8 clients in the Pastel cluster. 79
Figure 4.6 Variation between requests' arrival times through 8 traces in the Suno cluster...... 81
Figure 4.7 Requests that were not correctly paired when combining 8 traces from the Suno cluster with 10% of acceptable bounds for arrival times' variation...................... 82
Figure 4.8 Access patterns from the server point of view.. 85
Figure 4.9 Striping of a file among four data servers.. 86

Figure 5.1 Profiling data from a SSD - Access time per request for several request sizes. 95
Figure 5.2 Absolute error induced by using linear regressions to estimate the access time curves. 98
Figure 5.3 Absolute error induced by using SeRRa to obtain the sequential to random throughput ratio.. 100

Figure 6.1 Performance results for the tested scheduling algorithm selection trees............. 110

Figure 6.2 Performance results for the tested scheduling algorithm selection trees using a spatiality and request size detection tree. 113

Figure A.1 Results obtained for the single application scenarios with the shared file approach in the Pastel cluster. 140
Figure A.2 Results obtained for the single application scenarios with the file per process approach in the Pastel cluster. 141
Figure A.3 Results obtained for the multi-application scenarios with the shared file approach in the Pastel cluster. 142
Figure A.4 Results obtained for the multi-application scenarios with the file per process approach in the Pastel cluster. 143
Figure A.5 Results obtained for the single application scenarios with the shared file approach in the Graphene cluster. 144
Figure A.6 Results obtained for the single application scenarios with the file per process approach in the Graphene cluster. 145
Figure A.7 Results obtained for the multi-application scenarios with the shared file approach in the Graphene cluster. 146
Figure A.8 Results obtained for the multi-application scenarios with the file per process approach in the Graphene cluster. 147
Figure A.9 Results obtained for the single application scenarios with the shared file approach in the Suno cluster. 148
Figure A.10 Results obtained for the single application scenarios with the file per process approach in the Suno cluster. 149
Figure A.11 Results obtained for the multi-application scenarios with the shared file approach in the Suno cluster. 150
Figure A.12 Results obtained for the multi-application scenarios with the file per process approach in the Suno cluster. 151
Figure A.13 Results obtained for the single application scenarios with the shared file approach in the Edel cluster. 152
Figure A.14 Results obtained for the single application scenarios with the file per process approach in the Edel cluster. 153
Figure A.15 Results obtained for the multi-application scenarios with the shared file approach in the Edel cluster. 154
Figure A.16 Results obtained for the multi-application scenarios with the file per process approach in the Edel cluster. 155

Figure C.1 Interferência no acesso concorrente ao sistema de arquivos compartilhado. 160
Figure C.2 Ilustração da interface entre AGIOS e um servidor de dados de um sistema de arquivos paralelo. 162
Figure C.3 Arquitetura da ferramenta AGIOS. 165
Figure C.4 Erro absoluto induzido pelo uso da SeRRa para obter a razão entre acessar sequencialmente ou aleatoriamente. 167
Figure C.5 Resultados de desempenho para as árvores de seleção de algoritmo de escalonamento usando espacialidade e tamanho de requisições detectados. 169

LIST OF TABLES

Table 3.1 Summary of the presented scheduling algorithm's main characteristics - part 1..... 49
Table 3.2 Summary of the presented scheduling algorithm's main characteristics - part 2. M is the number of files, N is the number of requests and N_{queue} is the number of requests in the largest queue. 49
Table 3.3 Platforms used for AGIOS's performance evaluation. 52
Table 3.4 Sequential to Random Throughput Ratio with $8MB$ requests for all tested platforms described in Table 3.3. 52
Table 3.5 Nodes used for running our experiments. 53
Table 3.6 Amount of data accessed on this chapter's tests. 55
Table 3.7 Average absolute differences between TO and dNFSp' timeorder algorithm.......... 56
Table 3.8 Best choices in scheduling algorithms to all situations tested in this chapter......... 67

Table 4.1 Average performance improvements with AGIOS (base scheduler) 75
Table 4.2 Average improvements with the Prediction Module. 76
Table 4.3 Trace files' size and time in minutes required for the Prediction Module to make predictions from them in the Suno cluster. 78
Table 4.4 Overhead caused by trace creation. 79
Table 4.5 Confusion matrix obtained with the J48 algorithm using average distance and accesses' size to classify access patterns. 88
Table 4.6 Confusion matrix obtained with the J48 algorithm using average stripe access time difference to classify between small and large requests access patterns. 90
Table 4.7 Precision and recall observed with the SimpleCart algorithm using average distance between requests and average stripe access time difference to classify between the four considered access patterns. 90

Table 5.1 Original time in hours to profile the storage devices used in this study. 94
Table 5.2 Configuration of the evaluated storage devices 97
Table 5.3 Median from estimation errors. 99
Table 5.4 Median of the errors with SeRRa. 101
Table 5.5 SeRRa's time to measure (minutes). The fractions represent the ratio between these times and the originally required times (without SeRRa) 101
Table 5.6 Sequential do random ratio with 1200MB files - measured vs. estimated with SeRRa (4 repetitions). 102

Table 6.1 Summary of the five decision trees' characteristics - part 1 108
Table 6.2 Summary of the five decision trees' characteristics - part 2. 108
Table 6.3 Correct selection rate of all solutions compared with the oracle. 109
Table 6.4 Misclassification rates for each decision tree's two versions, using one to evaluate the other. 111
Table 6.5 Sizes of the files generated at the servers. 112
Table 6.6 Proportion of correctly detected access patterns (spatiality and request size) in all platforms using the decision tree from Chapter 4. 112
Table 6.7 Improvements provided by the scheduling algorithm decision trees over the aIOLi-only solution. 114
Table 6.8 Improvements provided by the scheduling algorithm decision trees over the SJF-only solution. 114

Table 7.1 Summary of related work on I/O scheduling - part 1 127

Table 7.2 Summary of related work on I/O scheduling - part 2. 127

CONTENTS

1 INTRODUCTION **15**
1.1 Problem Statement **17**
1.2 Objectives and thesis contributions **18**
1.3 Research context **19**
1.4 Document organization **19**
2 BACKGROUND **21**
2.1 Parallel File Systems **22**
2.1.1 Storage Devices 25
2.1.2 The Operating System's I/O Layers 28
2.2 Applications **29**
2.3 I/O Libraries **33**
2.4 I/O Optimizations for Parallel File Systems **33**
2.4.1 Requests Aggregation 34
2.4.2 Requests Reordering 34
2.4.3 Intra-node I/O Scheduling 35
2.4.4 I/O Forwarding 36
2.4.5 Optimizations for Small and Numerous Files 37
2.4.6 I/O Scheduling 38
2.5 Conclusion **39**
3 AGIOS: A TOOL FOR I/O SCHEDULING ON PARALLEL FILE SYSTEMS **43**
3.1 Scheduling algorithms **45**
3.1.1 aIOLi 46
3.1.2 MLF 47
3.1.3 SJF 48
3.1.4 TO and TO-agg 48
3.1.5 Summary of the scheduling algorithms 48
3.2 Performance evaluation **50**
3.2.1 Experimental platforms and configuration 52
3.2.2 Evaluated access patterns and experimental method 53
3.2.3 Performance results 56
3.2.3.1 Performance results with TO 56
3.2.3.2 Performance results for single application with shared file 57
3.2.3.3 Performance results with multiple applications and the shared file approach 61
3.2.3.4 Performance results for single application with the file per process approach 62
3.2.3.5 Performance results with multiple applications and the file per process approach 64
3.3 Conclusion **65**
4 APPLICATION-GUIDED I/O SCHEDULING **69**
4.1 Improving aggregations by using information from traces **70**
4.1.1 Predicting aggregations 71
4.1.2 Including Predictions on the Scheduling Algorithm 73
4.1.3 Performance Evaluation 74
4.2 Characteristics and limitations of the trace files approach **77**
4.2.1 Arrival times variability and combination of multiple trace files 80
4.2.2 Applications vs. Files 83
4.3 Classifying access patterns from traces **83**
4.3.1 Average distance between consecutive requests 86
4.3.2 Average stripe access time difference 89
4.4 Conclusion **91**

5 STORAGE DEVICES PROFILING **93**
5.1 SeRRa: A Storage Device Profiling Tool **94**
5.2 SeRRa's Evaluation **96**
5.2.1 Tests Environments and Method 96
5.2.2 Estimation error by linear approximation 97
5.2.3 SeRRa's Results Evaluation 100
5.3 Conclusion **102**
6 I/O SCHEDULING WITH DOUBLE ADAPTIVITY **105**
6.1 Scheduling algorithm selection trees **106**
6.2 Selection trees' evaluation **108**
6.2.1 Results with perfect access pattern detection 109
6.2.2 Results with detected access patterns 112
6.3 Characteristics and limitations of this approach **114**
6.4 Conclusion **115**
7 RELATED WORK **119**
7.1 Related work on I/O scheduling **119**
7.1.1 Dedicating all servers to an application 120
7.1.2 aIOLi 121
7.1.3 Reactive Scheduling 122
7.2 Related work on access patterns detection **123**
7.2.1 Hidden Markov Models 123
7.2.2 Post-mortem analysis of trace files 124
7.2.3 Grammar-based detection 125
7.3 Related work on storage devices profiling **125**
7.4 Conclusion **126**
8 CONCLUSION AND PERSPECTIVES **129**
8.1 Publications **131**
8.2 Research perspectives **132**
REFERENCES **133**
APPENDIX A I/O SCHEDULING ALGORITHMS PERFORMANCE EVALUATION **139**
APPENDIX B ABSTRACT IN FRENCH **157**
B.1 Résumé **157**
B.2 Résumé de thèse vulgarisé pour le grand public **158**
APPENDIX C EXTENDED ABSTRACT IN PORTUGUESE **159**
C.1 Resumo - "Escalonamento de E/S Transversal para Sistemas de Arquivos Paralelos: das Aplicações aos Dispositivos **159**
C.2 Introdução **160**
C.3 AGIOS: Uma Ferramenta para Escalonamento de E/S em Sistemas de Arquivos Paralelos **161**
C.3.1 Algoritmos de Escalonamento 162
C.3.2 Avaliação de Desempenho 162
C.4 Escalonamento de E/S Direcionado às Aplicações **164**
C.5 Perfilamento de Dispositivos de Armazenamento **166**
C.6 Escalonamento de E/S com Dupla Adaptatividade **167**
C.7 Conclusão **170**

1 INTRODUCTION

Computing systems inspired by the *Von Neumann architecture* have a memory where their programs' data is stored and from where instructions are fetched for processing in the CPU. Today's systems implement a memory hierarchy, with volatile RAM-based main memory and some levels of cache. Nonetheless, after the execution of computer programs, or *applications*, their data often must be stored in a non-volatile manner, so it will not be lost when the computer is no longer connected to a power source.

For years, *Hard Disk Drives* (HDDs) have been the main non-volatile storage devices available. Data is written to and read from magnetic rotating disks through a moving arm. Multiple hard disks can be combined into a virtual unit as a *RAID array* for performance and reliability purposes. RAID arrays stripe data among the disks to be retrieved in parallel, which improves performance. Another recent and alternative technology uses flash-based storage devices named *Solid State Drives* (SSDs). Their advantages over HDDs include: resistance to falls and vibrations, size, noise generation, heat dissipation, and energy consumption (CHEN; KOUFATY; ZHANG, 2009). Finally, hybrid solutions where devices with different capacities and speeds are used together have been gaining popularity. (WANG et al., 2006; QIU; REDDY, 2013)

Physical storage devices are abstracted to applications by file systems. Data is separated in files and associated with metadata, which are attributes about data such as name, size, and ownership. Through the file system, applications make *I/O requests* for portions of files. We call the way applications access data their *access pattern*: how many requests are issued, the requested portions' size, how these portions are spatially located in the file, etc.

As requests are generated by applications to a storage device through the file system, the *disk scheduler* is responsible for deciding the order in which these requests will be served. Because of storage devices' characteristics, the way they are accessed has a deep impact on performance. In order to achieve better performance, the disk scheduler may reorder requests to adapt applications' access patterns. For instance, most schedulers developed for hard disks try to reduce the required movement of the disk's arm, whose cost greatly affects performance. Techniques that try to improve the I/O subsystem's performance are of great importance, since disks and memory access speed have not increased in the same pace as processing power (PATTERSON; HENNESSY, 2013). Therefore, applications that need to access a large amount of data usually have their performance impaired by I/O.

Demanding applications - as weather forecast, seismic simulations, and hydrological mod-

els - execute on *supercomputers*. These *High Performance Computing* (HPC) systems usually present a *cluster* architecture, where multiple individual machines, called nodes, are interconnected through a high speed network. Applications divide their workload into smaller parts, or processes, and each of them is processed in a different node. These processes executing parts of the same application often need to access shared files. *Parallel File Systems* (PFS) allow that by providing an abstraction of a local file system. Through this abstraction, applications can access shared files transparently - i.e., without knowledge of where this file is actually stored.

Like local file systems, parallel file systems' performance is strongly affected by the manner accesses are performed. Several techniques as collective I/O (THAKUR; GROPP; LUSK, 1999) were developed to adjust applications' access patterns, improving characteristics as spatial locality and avoiding well-known situations detrimental to performance, like issuing a large number of small, non-contiguous requests (BOITO; KASSICK; NAVAUX, 2011).

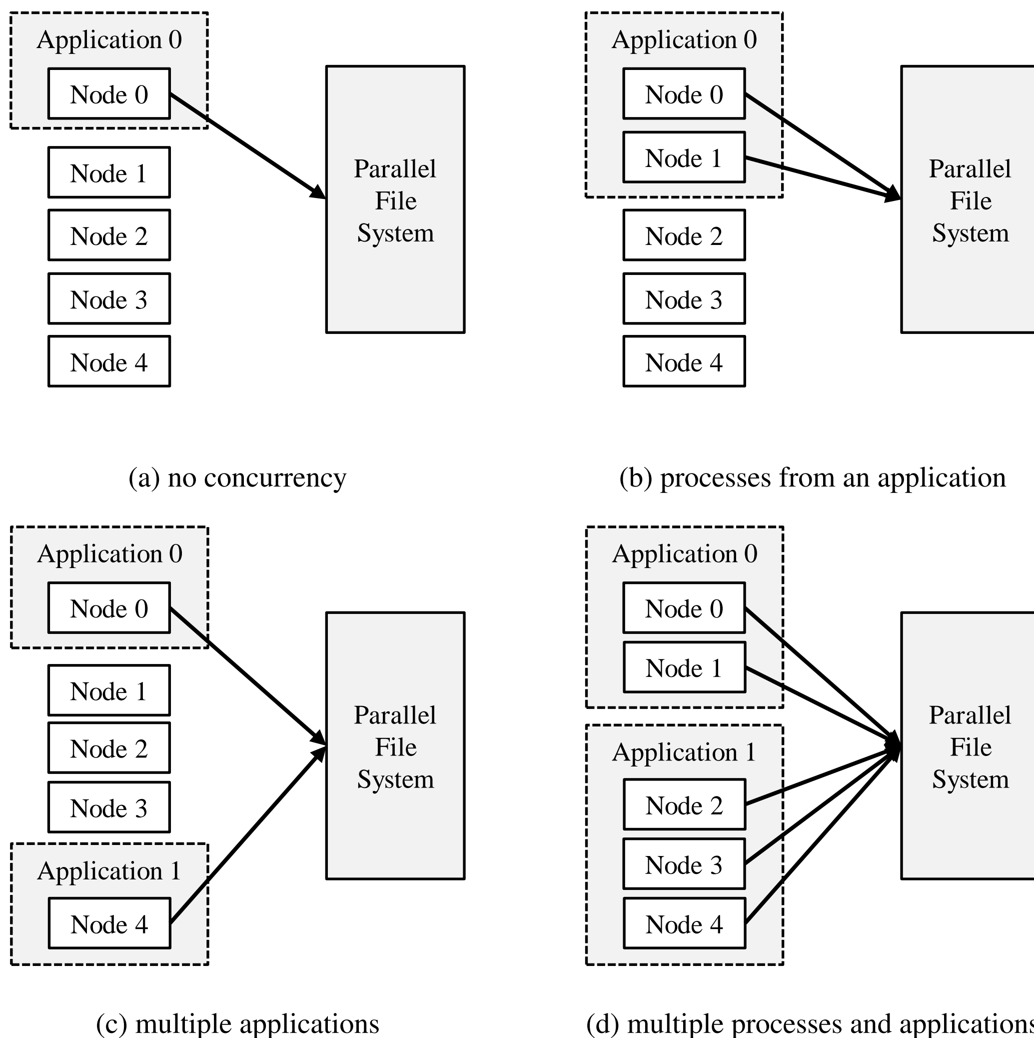


Figure 1.1: Different levels of concurrency on the access to a parallel file system.

It is usual for HPC systems to have multiple applications executing at the same time, each of them on a subset of the available processing nodes. These systems often offer a storage infrastructure, with a parallel file system deployment, that is shared by all applications. In this situation, when multiple applications generate requests to the same file system concurrently, these requests will interfere with each other. In this phenomenon, called *interference*, optimizations that work to adjust applications' access patterns will have their efficacy impaired and applications will observe poor I/O performance (LEBRE et al., 2006). Figure 1.1 illustrates the different levels of concurrency in the access to a parallel file system: between applications' processes and between multiple applications.

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

1.1 Problem Statement

I/O schedulers typically work on a stream of incoming requests. Since delaying these requests can have a significant impact on performance, scheduling decisions use only a small request window. On the other hand, a larger window would give more opportunities for optimizations and lead to a better scheduling. Knowledge about applications' access patterns could lead to better decisions for optimized performance. Nonetheless, this information is seldom available for schedulers since it is usually lost on the I/O stack. In the case of most parallel file systems' servers, all information they have comes from the requests they have at a given moment. These requests are to files' offsets with a size, and do not usually carry information about which application they come from, how many files this application accesses, what is the spatiality and the size of these accesses, etc.

One solution would be to optimize the scheduler for a given common access pattern, in order to achieve good performance when executing a certain class of applications. However, given the shared nature of a parallel file system deployment, it is preferable for schedulers to benefit all possible applications, and not only some of them. Therefore, I/O schedulers must *adapt* their behavior to applications' access patterns.

Scheduling algorithms work on performance assumptions, and these assumptions may not hold depending on the device where data is stored. For instance, HDDs are known for having

better performance when accesses are done sequentially rather than randomly. On the other hand, works that aim at characterizing SSDs' performance behavior achieve different conclusions. On some SSDs, there is no difference between sequential and random accesses, but on others this difference achieves orders of magnitude. The sequential to random throughput ratio on some SSDs surpasses what is observed on some HDDs (RAJIMWALE; PRABHAKARAN; DAVIS, 2009). The storage device's sensitivity to accesses sequentiality affects the effectiveness of the requests reordering done by schedulers. Moreover, the scheduler may employ optimizations that seek to change the access pattern, but these optimizations' influence on performance also depends on the storage device.

Additionally, new paradigms such as clouds, where the storage infrastructure is offered as a service, challenge the typical static parallel file system deployment. Since users' I/O needs may greatly vary over time, the storage infrastructure must be able to shrink or expand in order to attend these needs. In this scenario, new devices could be dynamically included for additional storage, and a static I/O scheduling configuration would not be suitable. For these reasons, I/O schedulers must *adapt* their behavior to storage devices characteristics.

1.2 Objectives and thesis contributions

Considering both issues, **the main objective of our research is to provide I/O scheduling for parallel file systems for improved performance**. We follow the hypothesis that, in order to provide good results, **I/O schedulers must have double adaptivity: to applications' access patterns, and to storage devices**. Considering these goals, our main contributions are the following:

- We show that I/O scheduling results depend on both applications' and storage devices' characteristics. We conducted an extensive evaluation of five scheduling algorithms over four different platforms and under different access patterns. Our results evidence that no scheduling algorithm is able to improve performance to all cases, and the best fit depends on the situation.
- We propose an approach to obtain information about applications' and to provide this information to algorithms. We show that such information can be used to improve I/O schedulers' decisions. Additionally, we also show that access patterns' aspects such as spatiality and requests size can be identified from applications' past accesses by applying machine learning techniques.

- We propose the use of the sequential to random throughput ratio metric to profile storage devices. This metric quantifies the difference between accessing a device sequentially or randomly. Since I/O profiling is a time consuming task, we developed a tool that uses models to provide accurate results as fast as possible.
- We applied machine learning to build decision trees that automatically select the best fit in scheduling algorithm using information about applications and platforms. We evaluate different approaches to build these trees, changing the used parameters. Our results evidence that, through double adaptivity, schedulers can provide good results by exploring the different scheduling algorithms' advantages.

1.3 Research context

This research is conducted in the context of a joint doctorate between the *Institute of Informatics* of the *Federal University of Rio Grande do Sul* (UFRGS) and the *Mathematics, Information Sciences and Technologies, and Computer Science* (MSTII) Doctoral School, part of the *University of Grenoble* (UdG). This collaboration is held within the *International Laboratory on High Performance and Ambient Informatics* (LICIA).

At UFRGS, the research is being developed in the *Parallel and Distributed Processing Group* (GAPED); and, at UdG, in the *Mescal* team, which is part of the *Grenoble Informatics Laboratory* (LIG). Both groups have a history of collaboration on I/O research. A previous joint doctorate resulted on the development of the dNFSp parallel file system, used in this work. Additionally, one of the used scheduling algorithms was proposed on a previous thesis from Mescal.

1.4 Document organization

The remaining chapters of this document are organized as follows:

- A review of parallel file systems' main characteristics and the typical I/O stack are presented in Chapter 2, together with a review on techniques to improve I/O performance.
- Chapter 3 presents our tool for I/O scheduling - AGIOS - and its five scheduling algorithms. An extensive performance evaluation is presented, using AGIOS to schedule requests to a parallel file system's server on four platforms under different access patterns.
- Our approach to obtain information from applications is discussed in Chapter 4. It details

the information provided by AGIOS, how it is obtained, and its use by a scheduling algorithm. An approach to detect access patterns' spatiality and requests size aspects is also proposed and evaluated in Chapter 4.

- Chapter 5 discusses storage devices profiling through the sequential to random throughput ratio metric. The SeRRa profiling tool is presented and its accuracy is evaluated on nine different platforms.
- The use of machine learning techniques to obtain decision trees that automatically select the best fit in scheduling algorithm is discussed in Chapter 6. Different decision trees are obtained by using different parameters, and their results are evaluated.
- Chapter 7 reviews related work on I/O scheduling, access patterns' aspects detection, and storage devices profiling.
- Concluding remarks and research perspectives are discussed in Chapter 8.

Additionally, Appendix A presents all results obtained in the performance evaluation discussed in Chapter 3.

2 BACKGROUND

Applications need non-volatile storage for their data, and this is done with storage devices as HDDs and SSDs. An abstraction of these devices is offered by file systems. They introduce the concept of files, which are data units associated with metadata. Files are accessed by applications through an interface that defines I/O operations like *open*, *write*, *read*, and *close*. These operations generate *requests* treated by the file system.

When applications execute on *clusters*, their computation is divided among multiple processes and distributed over a set of processing nodes. In this situation, applications' needs in I/O include reading from and writing to files shared by all processes. *Parallel File Systems* (PFS) allow that by providing a shared storage infrastructure so applications can access remote files as if they were stored on a local file system. We call processes that access a PFS its *clients*.

Figure 2.1 brings an overview of the main components that affect I/O performance when using parallel file systems. They are discussed on Sections 2.1 to 2.3, providing the base concepts needed for the rest of this document. Sections 2.4 and 2.4.6 present related work on techniques to improve I/O performance, the latter focusing on I/O scheduling. Finally, Section 2.5 summarizes this chapter's discussions by pointing desirable characteristics from I/O schedulers that we aim at providing.

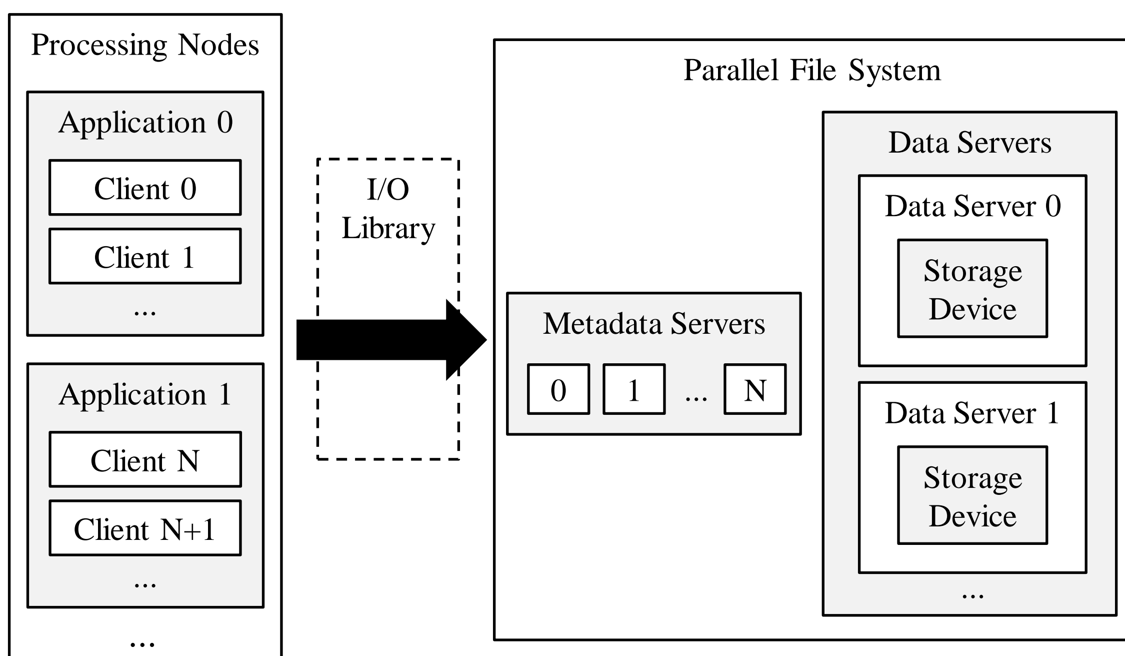


Figure 2.1: Logical components involved in performing I/O to parallel file systems.

2.1 Parallel File Systems

The first Distributed File Systems (DFS) were developed in the 80s aiming at allowing the sharing of storage devices, that were an expensive and rare resource (COULOURIS; DOLLIMORE; KINDBERG, 2005). These first systems, like Sun's *Network File System* (NFS) (SUN MICROSYSTEMS, INC, 1989), allow applications to store and access remote files as if they were local files. They are usually formed by a centralized server that is responsible for data storage and dealing with all clients' requests. However, as the number of clients and the volume of data grows, the performance of these centralized file systems becomes poor as the centralized server becomes a bottleneck (MARTIN; CULLER, 1999). The idea of distributing the server role among several nodes was introduced in the 90s by IBM's *Vesta File System* (CORBETT; FEITELSON, 1996).

Today's systems targeting HPC distribute files among several devices and allow data from these locations to be accessed in parallel, providing higher performance. For this reason, they are called *Parallel File Systems* (PFS) (THANH et al., 2008).

On systems that present *symmetric architectures*, as IBM's GPFS (SCHMUCK; HASKIN, 2002) and HDFS (TANTISIRIROJ et al., 2011), data is distributed among all cluster's processing nodes. This organization allows for the placing of processes close to data they must access, reducing data transmission over the network and improving performance. Nonetheless, this approach is more suitable for applications that fit the *MapReduce* programming model (AFRATI; ULLMAN, 2010), where sharing between instances is limited to data scattering and results gathering phases. Although our discussions and contributions could be expanded to symmetric systems, this thesis focuses on parallel file systems with *client-server architecture*.

These systems have two specialized servers: the *data server* and the *metadata server*. The former stores data, while the latter is responsible for metadata, which is information about data like size, permissions, and location among the data servers. On some systems the two roles are played by the same servers, and most systems allow the placement of both data servers and metadata servers on the same machines. In order to access data, clients must first obtain location information from metadata servers.

As all basic file systems' operations involve metadata operations, the scalability of metadata accesses impacts the whole system's scalability. Some systems cache metadata on clients in order to make this process faster. However, this technique brings the complexity of maintaining cache coherence, especially when a large number of clients is concurrently accessing the file system. Another way of improving metadata accesses' performance is to distribute the metadata

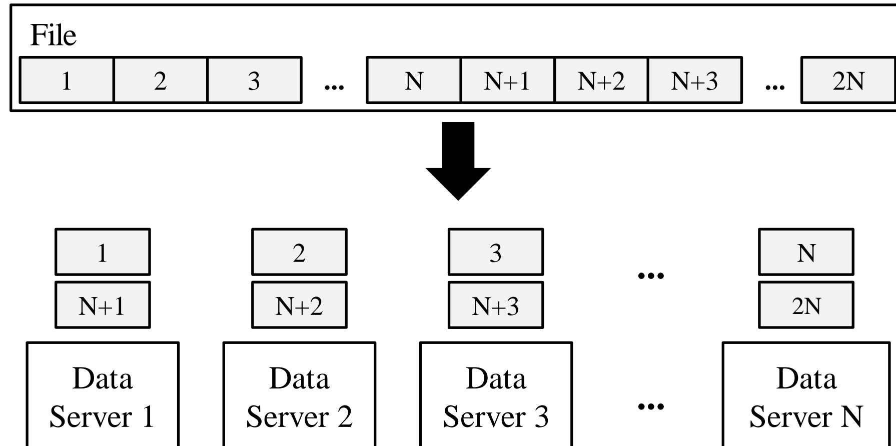


Figure 2.2: Striping of a file of size $2 \times N$ among N servers starting by Server 1.

storage among several nodes, in a similar way to what is done with data itself. This is done on systems like PVFS (LATHAM et al., 2004). Other systems, like Lustre (BRAAM; ZAHIR, 2002), decide not to distribute metadata in order to keep its management simple. This choice results in poor performance for metadata-intensive workloads, and is made considering the target applications' characteristics.

Files are distributed among data servers in an operation called *striping*. Each file is divided on portions of a fixed size, called *stripes*, and the stripes are given to the servers following a round-robin approach. Figure 2.2 illustrates this process through an example where a file of size $2 \times N$ is striped among N servers, starting at Server 1. Some file systems, like Lustre, allow striping configuration per file, while others, like dNFSp (AVILA et al., 2004), use a fixed approach. Always starting the striping on the same server has the disadvantage of potentially overloading this server, since all files, even the small ones, will be stored in it. PFS's main characteristic is the possibility of retrieving stripes from different servers in parallel, increasing throughput.

Some systems apply *locking* at the servers in order to keep consistency on the presence of concurrent accesses. This is done by Lustre using stripe granularity, i.e., multiple clients are not allowed to access the same stripe concurrently. Other systems, like PVFS, leave the consistency to be treated by applications and I/O libraries for simplicity and performance.

Data transmission unit between clients and servers is limited by stripe size and request size. Additionally, systems usually define a maximum transmission size according to their protocol's capacities. Figure 2.3 brings an example to help visualize these values' implications. In the pictured example, a file is striped among four data servers using a stripe size of 64KB. A client generates a contiguous 320KB write request, thus this request will be divided according to the stripe size in order to give the corresponding stripes to each server. Moreover, accesses to each

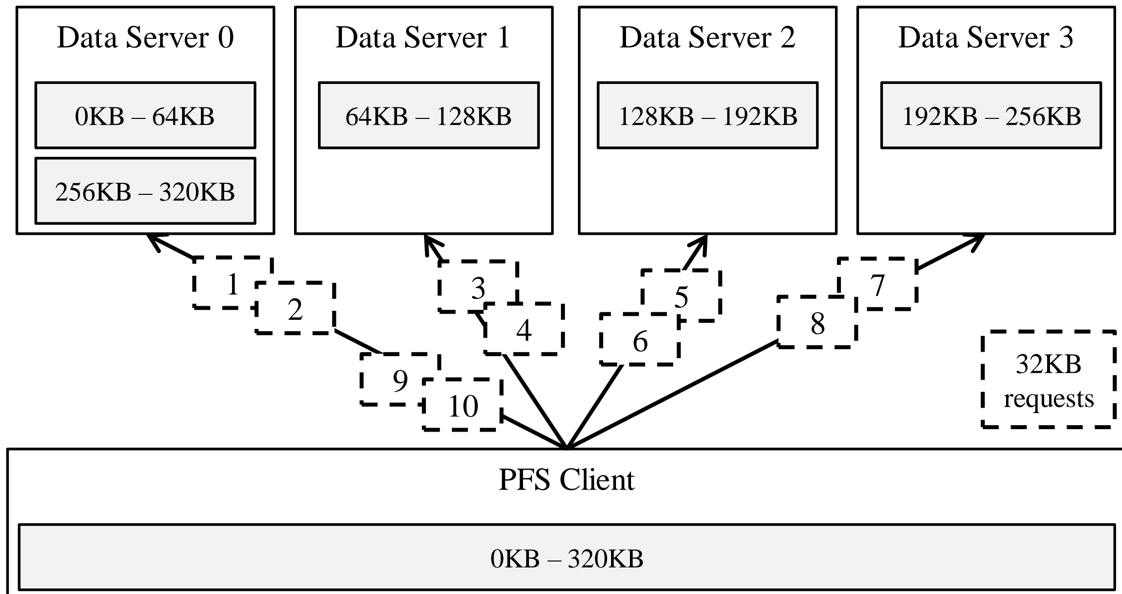


Figure 2.3: A client generates a write request of 320KB. Stripe size is 64KB and the maximum PFS transmission size is 32KB.

server will be done according to the file system's maximum transmission size, 32KB in this example. Therefore, the original 320KB request from the client generated ten 32KB requests to the parallel file system.

Considering this example, decreasing the stripe size to 16KB would require twice as many requests, increasing overhead. On the other hand, increasing stripe size to 1MB would affect performance by eliminating access parallelism. Moreover, if the file system used stripe locking and other client were to access the subsequent 320KB from the same file, a 1MB stripe size would cause these two accesses to be serialized despite the fact they do not overlap and should not have any effect on each other. Therefore, the striping configuration depends on system's target applications. The retired Google File System (GHEMAWAT; GOBIOFF; LEUNG, 2003) employed a 64MB stripe size because its target applications performed very large sequential accesses only. Most systems have a default between 32KB and 1MB, which is good for general use.

In order to include fault tolerance, several systems support replication of data and metadata. This is usually done by keeping mirrored servers, and has a performance impact since copies must be kept synchronized. On the other hand, having the same data on more than one server allows parallel access, improving performance. In addition, there are techniques that aim at placing replicas closer on the network hierarchy to applications that access them (YU et al., 2008).

File systems can be classified as *stateless* or *stateful*. Stateless systems do not keep information about connections with clients, contrary to stateful systems. Most parallel file systems

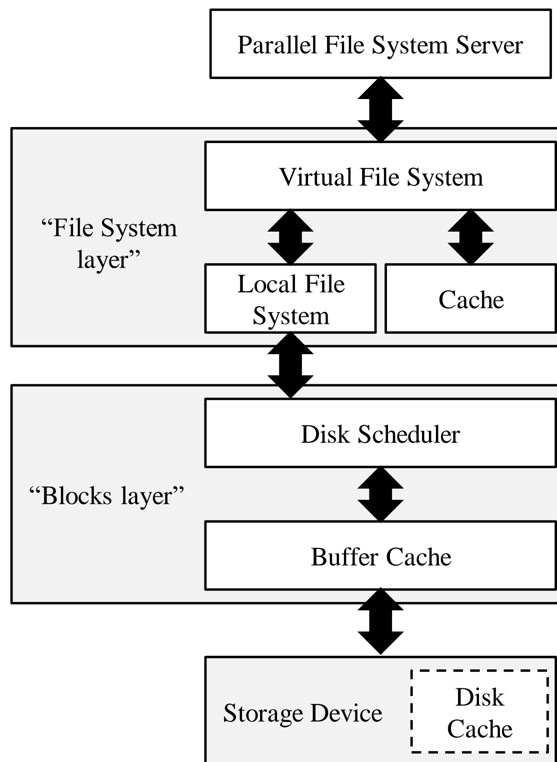


Figure 2.4: High-level overview of the local I/O stack.

are stateless, since this approach is simpler and better for performance. Moreover, on a stateless system a fault in a client will not affect the servers (THANH et al., 2008).

On most systems, servers run on dedicated nodes from the cluster, and store their data through files on the local file system of these machines. Therefore, their access performance is affected by local characteristics. Figure 2.4 presents an overview of the local I/O stack that PFS servers - applications in the local context's point of view - use to store data. Sections 2.1.1 and 2.1.2 discuss these levels following a bottom-up approach. Section 2.2 returns to the context of Figure 2.1 by discussing characteristics of applications that access parallel file systems.

Although this discussion can be expanded for Windows-based systems, we focus on UNIX systems' characteristics since they are used on most of HPC architectures. On the latest *TOP500* list, only two systems use Windows as operating system[1].

2.1.1 Storage Devices

Most of current systems are adapted to obtain good performance when accessing *hard disk drives (HDDs)*, since they have been the main storage device available for many years. These

[1]TOP500 lists the world's fastest supercomputers, evaluated with the LINPACK benchmark. The list was accessed in June 2014 and can be accessed at <www.top500.org>

devices are composed of magnetic surfaced *rotating platters* where data is recorded, and a moving *head* (or *arm*) that reads or writes data. Each disk surface is divided into concentric circles - the *tracks* - and each track is divided into *sectors*. The same track over the multiple rotating platters is called a *cylinder*. (PATTERSON; HENNESSY, 2013).

Accessing data from a hard disk requires moving the head to the proper track, in an operation known as *seek*. After the head is correctly placed, it has to wait as disk rotates for the correct sector to be positioned - this time is known as *rotational latency*. Access time also depends on *transfer time* and *disk controller*'s overhead. The controller is responsible for exporting a *Logical Block Addressing* of disk's contents and translating it to physical sectors (JACOB; NG; WANG, 2010).

Hard disks are known for having better performance when accesses are done sequentially instead of randomly, because the former minimizes seek times. Despite disk controllers being able to change logical blocks placement on the disk, it is generally accepted that nearby *Logical Block Numbers* (LBNs) translate well to physical proximity (RAJIMWALE; PRABHAKARAN; DAVIS, 2009).

A popular solution for storage on HPC systems is the use of *RAID arrays* that combine multiple hard disks onto a virtual unit for performance and reliability purposes. Data is striped among the disks and can be retrieved in parallel, which improves performance. RAID arrays' performance is affected by the combination of stripe size and accesses' size, similarly to what was previously discussed for striping among parallel file system's servers.

Solid State Drives (SSDs) are a recent alternative to hard disks. SSDs may use one of two types of NAND flash memory: *Single-Level Cell* (SLC) or *Multi-Level Cell* (MLC). The former stores a single bit on a flash cell, while the latter stores multiple bits. MLC SSDs have shorter lifetime and slower write operations than SLC ones (CHEN; KOUFATY; ZHANG, 2009).

SSDs are composed of a set of flash *packages* connected to a controller. Figure 2.5 brings a simplified vision of a flash package's architecture: flash cells are organized in *pages*, multiple pages form a *block*, and multiple blocks form a *plane*, that also contains a little RAM cache. Two or more planes are grouped in a *chip* (or *die*), and multiple chips form a package. The different levels offer parallelism - different channels, packages and chips can be accessed independently. Additionally, the same operation can be executed in parallel at different planes of the same chip, or at all blocks from the same *clustered block* inside a package. Therefore, issuing large requests to SSDs presents better performance than small ones, since it profits from available parallelism (KIM et al., 2012).

A flash page is the minimum amount of data that can be read or written, and its typical size

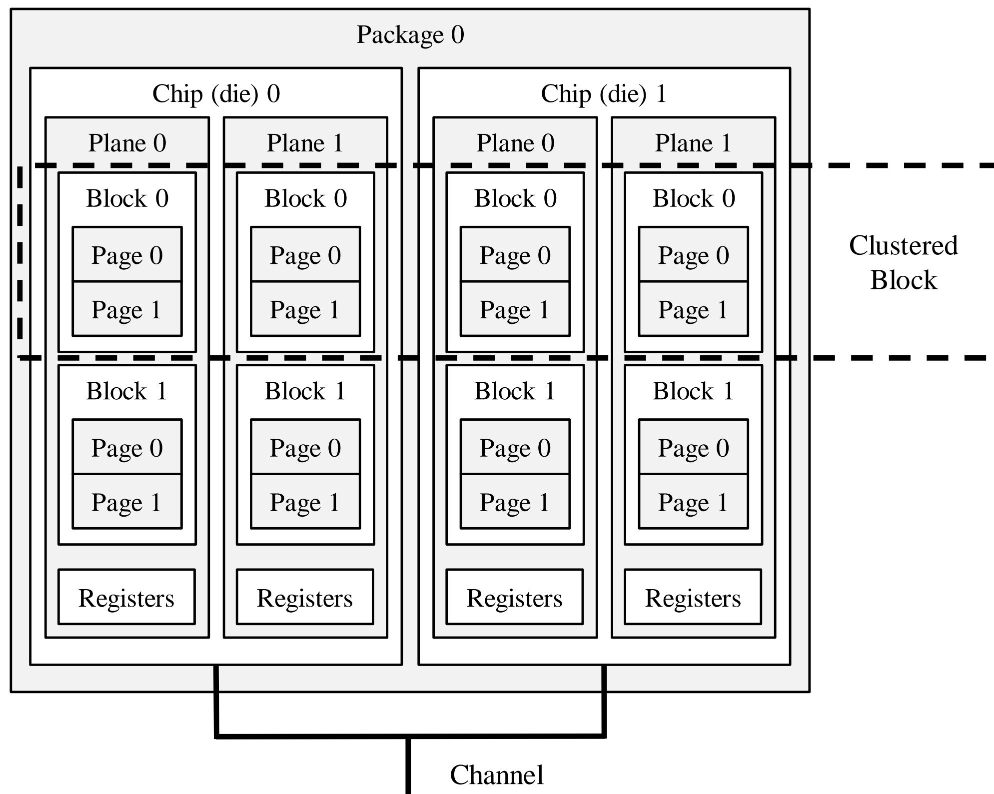


Figure 2.5: Simplified view of SSD's flash packages.

varies from 2KB to 16KB on today's drives. For this reason, it is better to access multiples of page size. It is not possible to update pages in place: to perform a write, the corresponding page must be read, modified and then re-written to a free location. The previous location of the page will be marked as stale and subject to garbage collection, that happens in background. Erases are done at block level, thus valid pages will be copied to a free block, and the whole original block will be erased. The read-modify-write process is one of the causes for *write amplification*, which impairs write performance on SSDs (RAJIMWALE; PRABHAKARAN; DAVIS, 2009).

The *Flash Translation Layer* (FTL) emulates a hard disk controller by translating LBNs to pages. Keeping a translating table at page scale would be harmful to performance and hard to manage, but translating at block scale (with a much smaller table) would incur in all operations having to be performed to whole blocks instead of pages. Modern SSDs employ a hybrid between page level translation and *log-structured design*: write operations are sequentially written to log blocks, translated at page level, and log blocks are merged to their corresponding data blocks, indexed at block level, when full. Since flash pages have a limitation of erases that can be performed (around 10K for MLC), another FTL attribution is to perform *wear-leveling*, re-distributing data over the device in order to avoid using some blocks more than others. Both background operations - merging of log blocks and wear-leveling - can be triggered by write operations, adding to write amplification's causes (CHUNG et al., 2009).

Because of these characteristics, there is no guarantee that sequential logical addresses will translate to sequential physical locations on SSDs. Nonetheless, it was reported that for some devices random writes perform worse than sequential ones, especially when writes are smaller than the clustered blocks' size (today's clustered blocks have around 32MB). This happens because random writes increase garbage collection overhead. Moreover, random accesses increase the associativity between log blocks and data blocks, causing more costly merge operations. These effects are alleviated when random accesses have at least the clustered block's size because a whole clustered block can be erased in parallel (MIN et al., 2012).

Despite the growing adoption of SSDs, their larger cost per byte still hampers their use on large-scale systems for HPC. Therefore, several parallel file system deployments on clusters still store data on hard disks.

2.1.2 The Operating System's I/O Layers

Despite storage devices' physical characteristics, performance behavior observed when accessing them also reflects characteristics from higher levels of the I/O stack. Most HDDs and SSDs contain a small cache in hardware for their accesses. Additionally, the operating system's kernel has a *buffer cache* to mask devices' access costs. Both these caches typically perform *read-ahead*, i.e. they read from devices more data than requested assuming that contiguous data will be accessed in a near future. Therefore, random read accesses may perform worse than sequential ones because they do not fully exploit these mechanisms.

Another technique usually applied to buffer caches is *prefetching*. This approach tries to predict data that will be accessed by applications on the future and make these requests earlier. Prefetching can also be applied on parallel file system clients' caches, both for data and metadata.

Requests for logical block addresses arrive to storage devices after possibly going through *disk schedulers*[2]. These schedulers decide the order in which to process requests to logical blocks issued by different tasks, trying to maintain *fairness* between them. They usually allow *prioritization* of tasks. Additionally, some employ *elevator algorithms* to reduce seek operations in HDDs (ZHANG; BHARGAVA, 2008).

Block requests are made to block layers by the *local file system*, that translates applications' requests - for files offsets - to logical blocks according to their internal organization. These

[2]These schedulers are sometimes called "I/O schedulers", but in this thesis we exclusively refer to them as "disk schedulers" to avoid confusion with "I/O schedulers" that work in the context of requests to files.

file systems usually try to store data from the same file in contiguous blocks. However, *data fragmentation* can occur over time, causing portions of files to be scattered over the logical blocks addressing space. For this reason, requests for contiguous portions of a file could be translated to requests for non-contiguous logical blocks.

Some local file systems perform *journaling*, keeping changes to files in a log that could be replayed to recover from faults. The log, or *journal*, typically occupies a dedicated contiguous portion of the file system addressing space. Depending on the applied journaling level, both data and metadata could be written on the journal and later committed to the actual structures. This approach would cause random write accesses to perform better, since they would be sequentially written on the journal during the operation, and random accesses would be made only when committing the operation.

The *Virtual File System (VFS)* defines an interface for applications to access the underlying I/O sub-system. Through the unified view provided by the VFS, applications access files that can be located at different file systems and storage devices transparently. The structures that describe files and directories - metadata - are cached on the VFS for fast access.

The next section moves on from the parallel file system's to applications' point of view. Access pattern aspects and their impact on performance will be discussed.

2.2 Applications

Scientific applications are used to simulate and understand complex phenomena, like weather forecasting and seismic simulations. These applications fuel the high performance computing field with performance requirements in order to simulate these events with more detail and achieve better results.

In general, these applications start their execution by reading data from previous executions, or even previous steps of the analysis. It is usual for simulations to have their execution organized as a series of *timesteps*. Each timestep evolves the simulated space on time, reaching its next state. The execution often finishes by writing the obtained results, but I/O operations may also be generated at every given number of timesteps. An example of such application is the *Ocean-Land-Atmosphere Model* (OSTHOFF et al., 2011; BOITO; KASSICK; NAVAUX, 2011).

Another common reason for applications to generate I/O operations is *checkpointing*. Some applications, like FLASH (FRYXELL et al., 2000; ROSS et al., 2001) periodically write their state on files so their execution can be easily resumed after interruptions. FLASH is an astro-

physics application which I/O kernel is widely used as an I/O benchmark. Its checkpoint file has a segment for each variable of the execution, where each process will write its value for this variable. Therefore, during a checkpointing phase, each process will generate small write requests to sparse positions of the file.

We call the description of the I/O operations performed by an application its *access pattern* or *I/O signature*. Applications' access patterns have a deep impact on performance.

Several works aim at identifying access patterns that are common among scientific applications. These works are important because they indicate situations that must be targets to optimizations. In general, these studies apply statistical analysis on execution traces from large clusters, focusing on I/O operations.

Purakayastha et al. (1995) showed that *small accesses are usual in scientific applications*. Pasquale and Polyzos (1994) observed that *most scientific applications have a regular access pattern*. This find justifies the effort put into classifying access patterns, since it says that it is possible to classify most applications. In the work from Roselli, Lorch and Anderson (2000), the study of traces from several file systems has shown that *metadata accesses may respond to more than 50% of applications' accesses*.

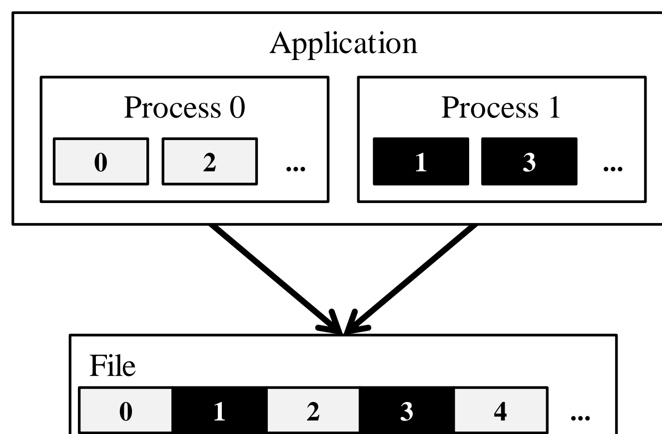


Figure 2.6: One application with two processes accesses a file. Each process presents a 1-D strided local access pattern.

A parallel application can present two distinct access patterns: the *global access pattern* describes the way the whole application does I/O, while the *local access pattern* refers to the access pattern of each process (or task) individually. Figure 2.6 illustrates this: an application composed of two processes accesses a file. The global access pattern says that this application accesses the file sequentially, while locally each process access portions with a fixed-size gap between them. This local pattern is known as *1-D strided access pattern*. Knowing the local access pattern is usually relevant for client-side optimizations: since they work in the context of

one (or sometimes a few) clients, the global pattern is not relevant for them. On the other hand, server-side optimizations usually work with the global information.

There is no globally accepted taxonomy for applications' access patterns. On the literature, works that provide some classification of patterns usually do it in the context of specific optimizations. In these works, the classification considers only the aspects that are relevant to the proposed optimizations, not intending to be a complete description for all purposes. This section discusses some of these works focusing on the relevant aspects when classifying access patterns.

The most usually considered access pattern aspect is *spatiality*. It tells the location of accesses into the file: if requests are contiguous, distant by a fixed value, randomly positioned, etc. Spatiality is an important aspect because of its direct impact on performance. This happens because, as discussed in Section 2.1, the storage infrastructure where file system servers store data has its performance affected by accesses' spatiality. Additionally, spatiality can affect the use of local cache on servers, the efficacy of prefetching and read-ahead mechanisms on clients, etc (WORRINGEN, 2006).

Kotz and Ellis (1991) proposed an access pattern classification in eight classes, four regarding local access patterns and four for global ones. The aspects used to define these classes are spatiality and *requests size*. According to this classification, it is possible for processes to access a shared file: a) from beginning to end; b) following a 1-D strided pattern; c) to randomly located random-sized portions; or d) sequentially into an exclusive portion of this file. There is also a class for representing patterns that are not represented by other classes.

Knowing if the requests size is constant or variable allows tools to know what to expect from the application and then decide, for instance, the granularity of the requests sent to the file system through aggregation mechanisms (THAKUR; GROPP; LUSK, 1999). Requests size affects performance mainly because, together with stripe size (as discussed in Section 2.1), it defines the transmission unit between clients and servers. Small requests can impair performance with overhead from a high number of transmissions. Moreover, there are fixed costs for treating requests at the parallel file system, so using small requests increases this overhead (CARNS et al., 2009). Requests size impacts are also due to storage devices' sensitivity to accesses' size.

We adapted the classification from Kotz and Ellis in a previous work for the creation of benchmarks, aiming at representing a large number of existing applications (BOITO; KASSICK; NAVAUX, 2011). Our classification for local access patterns considers spatiality, requests size, *number of files*, *number of processes per processing node*, and *time between consecutive requests*.

The number of files generated/accessed by applications affects the metadata operations load. Increasing the volume of metadata operations impacts overall performance, especially when handling small files (FRINGS; WOLF; PETKOV, 2009; CARNS et al., 2009).

The number of processes per processing node, or *intra-node concurrency*'s influence on performance is a consequence of concurrency on the node's network infrastructure that happens when multiple processes execute on the same machine. Moreover, caching mechanisms can be affected by this situation, and it can generate contention on the access to memory (OHTA; MATSUBA; ISHIKAWA, 2009).

The *temporal aspect* (time between requests) represents applications' I/O needs during their execution. For instance, if an application generates a large volume of requests in a short period of time, its requirement in throughput from the file system is larger than it would be if these same requests were generated along the whole execution. The points in time where the application reads or writes a large volume of data are called *bursts*, and the characteristic of having I/O bursts during execution is sometimes referred to as the application's *burstness*.

From the server point of view, it is usual to express applications' temporal aspect by a *request arrival rate* that gives the number of requests that arrive at the server in a certain amount of time (ZHANG; BHARGAVA, 2008). Another way of looking at this aspect is to define *affinity* between data structures (PATRICK et al., 2010), portions of files (SOUNDARARAJAN; MIHAILESCU; AMZA, 2008), or even whole files (LIN et al., 2008; KROEGER; LONG, 1999). Portions of data that have affinity are accessed in close instants of time. This information can be used to guide prefetching mechanisms.

Madhyastha and Reed (2002) proposed a classification that considers three aspects, later expanded by Byna et al. (2008) to five aspects: spatiality, requests size, temporal intervals, *operation type* (read or write), and *repetition*. They consider an application a sequence of access patterns that may or may not repeat themselves, thus the repetition aspect.

Their requests size classification states if requests are small, medium or big. A request is considered *small* if it is smaller than the stripe size, and *large* if it incurs in accesses to all file system servers (larger than number of server $\times$ stripe size), with the *medium* case laying in between. This definition for requests size is used in our work.

The information about the *operation type* is important because there are optimizations that are beneficial only for a certain type. For example, *prefetching* (BYNA et al., 2008) makes sense in the context of read operations, while attribution of exclusively dedicated servers to applications (KASSICK; BOITO; NAVAUX, 2011) improves performance of writes.

The way applications access remote parallel file systems is discussed in the next section.

2.3 I/O Libraries

As discussed in Section 2.1.2, the virtual file system layer allows transparent access from applications to files independently of where they are actually stored. Parallel file systems usually provide a client module to be installed on processing nodes so they can view the remote folders as local and access them through the *POSIX* API, that defines standard I/O operations.

Through the previous sections, the different levels involved in I/O and their sensitivity to access patterns were discussed. Depending on the complex interaction between the different levels and on the parallel file system's design choices, performance will be better for some access patterns than for others. Hence, achieving good performance when accessing a PFS depends on how well applications' access patterns suit the used system. Nevertheless, this tuning between applications and systems is not easily achieved because:

- parallel file systems do not have enough information about applications in order to adapt to them, as this information is lost through the I/O stack;
- tuning applications would require them to be developed considering the used file system, which would compromise their portability;
- for a good tuning, developers would need to know details about the target file system's performance behavior. Given the complexity of these systems, their behavior is not easily analyzed.

For these reasons, it is desirable to remove from applications the responsibility of adapting to file systems. One solution is the use of I/O libraries, the most popular being *MPI-IO* (CORBETT et al., 1996). These libraries take charge of applications' I/O operations and have the power to perform optimizations in order to adapt their access pattern. High level I/O libraries as *HDF5* (ROSS et al., 2001) also abstract I/O operations by allowing the definition of complex data types and file formats. These formats can be freely mapped to real files by these libraries, providing optimization opportunities. The next section details several techniques for I/O performance improvement.

2.4 I/O Optimizations for Parallel File Systems

Since performance depends on applications' access patterns, and some patterns are known to have better performance than others on some systems, ways of improving the performance observed by an application usually involve changing its access pattern to make it more suitable

to the used system. This can be done at the server side, by modifying the file system; or at the client side, by changing applications, I/O libraries, compilers, APIs, etc.

2.4.1 Requests Aggregation

Access patterns with small requests usually achieve poor performance from the parallel file system. However, this pattern is, as discussed before, common between scientific applications. In order to alleviate this problem, several optimization techniques focus on aggregating small requests into larger ones.

One of these techniques is the use of *data sieving* (THAKUR; GROPP; LUSK, 1999). With this optimization, instead of requesting small portions of data from the server, a large contiguous portion that contains all needed small portions is requested. In order to aggregate the small requests, some data may be unnecessarily requested in between. In order to increase performance, it is important for the amount of unnecessarily requested data not to be very large compared with useful data.

Another popular technique is the use of *collective operations*, whose idea consists on aggregating small requests issued by different clients, generating larger contiguous requests whenever possible. Collective operations can be implemented at server side (SEAMONS et al., 1995), or client side. The latter is more usual since it allows for portability, working with any parallel file system. MPI-IO uses a two-phased strategy: a phase for requesting data from the servers and another for exchanging it among the clients (THAKUR; GROPP; LUSK, 1999). The implementation of this method requires synchronizations and data movement between clients, often imposing an overhead that surpasses the performance improvements (LEBRE et al., 2006).

Collective operations' performance gains come from both aggregation and reordering of small requests. If made independently by clients, these requests would hardly arrive at the server in offset order. The next section discusses some techniques that aim at generating sequential access patterns.

2.4.2 Requests Reordering

Additionally to generating large requests instead of small ones, it is also important for performance to access sequential portions of files. Avoiding random accesses, as previously discussed, can improve performance of access to storage devices and promote a better cache us-

age, also helping the efficacy of techniques like prefetching and read-ahead. Clients' requests reordering is the focus of some optimization techniques.

The work from Zhang, Jiang and Davis (2009) shows that having requests arriving to servers in offset order improves observed performance. They call this phenomenon (when requests arrive by offset order) *resonance*. In order to cause resonance, they propose a technique at client side that reorders requests considering file system's striping. This information must be obtained from the metadata servers for this technique to work. Their results show performance improvements over the use of MPI-IO collective operations.

As previously discussed, during a checkpoint each application's process often generates small requests that are sparse on the shared file, resulting in an access pattern that achieves poor performance. Bent et al. (2009) proposed a virtual file system layer specific for checkpointing that maps each process' requests to an exclusive file, where these requests are contiguous. Despite the improvement by having sequential local access patterns, this approach could lead to the creation of a large number of small files, a pattern that often achieves poor performance on parallel file systems.

Although the distribution of data among servers allows for parallel accesses and is good for performance, in some cases the cost of establishing connections between a client and hundreds of servers can become a problem and impair performance. Dickens and Logan (2009) proposed a client-side I/O library for use with Lustre that reorders the accesses so each client will contact less servers.

On systems that apply stripe locking, like Lustre, it is better for performance when accesses are aligned with stripe size. This happens because the locking serializes the requests for the same stripe. Liao and Choudhary (2008) studied this phenomenon and proposed a modification in MPI-IO that considers the locking policy used by the systems.

2.4.3 Intra-node I/O Scheduling

As previously discussed, when multiple processes generate requests to the remote file system from the same node, there is contention in the access to memory and network resources. Ohta, Matsuba, and Ishikawa (2009) try to improve this scenario by proposing a new parallel file system named *PGAS* (Parallel Gather-Arrange-Scatter file system) where the requests from the same node are aggregated and reordered whenever possible, reducing concurrency. Their approach presented some performance improvements over MPI-IO collective operations.

With the same goal, the work from Dorier et al. (2012) proposes an approach named

Damaris that dedicates cores from SMP nodes for I/O. Processes assign their I/O operations to Damaris through a simple API. Processes' data is kept in main memory until the I/O thread uses routines provided by application itself to actually perform I/O to the parallel file system.

These techniques could be seen as *intra-node I/O scheduling*, since they introduce a central role of deciding the processing order of requests from multiple sources. They also perform optimizations (as requests aggregation) to adapt the resulting local access pattern.

2.4.4 I/O Forwarding

The IOFSL framework (ALI et al., 2009) proposes the creation of an I/O forwarding layer on large-scale architectures. The goal is to reduce the number of clients concurrently accessing the file system servers by having some special nodes (called *I/O nodes*) that receive the processing nodes' requests and forwards them to the file system. In this schema, the number of I/O nodes is larger than the number of file system servers, and smaller than the number of processing nodes. The processing nodes are powered with only a very simplified local I/O stack in order to avoid its interference on performance.

The IOFSL project[3] is focused on IBM Blue Gene architectures. The performance of I/O forwarding on an IBM Blue Gene/P from Argonne National Laboratory[4] was evaluated on the work from Vishwanath et al. (2010). They developed a simple *First Come First Served* (FCFS) scheduling mechanism to be applied by the I/O nodes, including the possibility of executing asynchronous operations. These modifications improved the I/O forwarding performance by up to 53%.

Another approach to I/O forwarding is presented in the work from Nisar, Liao and Choudhary (2008) that proposed an MPI-IO implementation where the user chooses to dedicate some of its processing nodes to I/O forwarding. Besides forwarding I/O, the I/O nodes also try to aggregate requests in order to improve performance.

The I/O forwarding technique was also applied to build the storage infrastructure from Tianhe-2, the current number one at the Top500 list. Tianhe-2's 16000 computing nodes do not have a local I/O stack, and transfer all I/O operations to the intermediate I/O nodes (the machine has 256 of these). Intermediate nodes have high speed SSDs, and the frequency with which data is transferred from these I/O nodes to the parallel file system can be configured to each file (XU et al., 2014).

[3]<http://www.iofsl.org/>
[4]<http://www.anl.gov/>

Besides performance, the idea of I/O forwarding has the advantage of providing a layer between application and file system. This layer can work to keep compatibility between both sides and apply optimizations like requests reordering and aggregation, etc.

2.4.5 Optimizations for Small and Numerous Files

As previously discussed, accessing files stored in parallel file systems involves accessing the metadata server. This access is usually made once for each accessed file, since the obtained information can be kept in cache by the clients. When accessing a large file, this metadata fetching overhead dilutes on the larger time taken to access data. However, when the file is small, this cost becomes important. Additionally, accessing multiple files overloads the metadata infrastructure, potentially turning this point into a bottleneck. Therefore, we can say that improving performance on situations where applications access a large number of small files comes from improving metadata access performance.

Carns, Settlemyer and Ligon's work (2008) proposes new algorithms for creation, removal and stat of files on PVFS. The modifications consist of applying collective communication between metadata servers to perform these operations. This communication was previously done following a binary tree topology. The authors argue that using collective communication allows for easy consistency treatment and improves performance, avoiding the bottleneck formed in the node commanding these operations (because it must access all other nodes). Four other optimizations are proposed in a later work from Carns et al. (2009): pre-creation of objects on data servers, stuffing of data, coalesced commits of changes to metadata servers and eager I/O protocol for communication between clients and servers when files are small.

At client side, aiming at avoiding the problem of having a large number of small files, the work from Frings, Wolf and Petkov (2009) proposes an extension of the I/O API from ANSI C called *SIONlib*. The library maps multiple applications' virtual files into a smaller number of real files in the file system. For that, open and close operations become collective calls where required synchronizations are performed. Their results indicate performance improvements in file creation.

2.4.6 I/O Scheduling

The last section presented several optimization techniques that aim at improving performance observed by applications when accessing a parallel file system. Most of these optimizations work at client side and therefore work in the context of an application, a processing node or even a process. However, large clusters usually offer a shared storage infrastructure, available to all applications running in the platform. In this situation, where multiple applications access the parallel file system at the same time, the discussed optimizations' efficacy can be compromised.

Figure 2.7 illustrates this situation with an example where two applications concurrently access a shared parallel file system's server. Despite these applications possibly applying optimizations to produce good access patterns, they will *interfere* with each other arriving at the server, where the resulting access pattern may present poor performance. We call this phenomenon *interference*.

This thesis focuses on *I/O scheduling* as a tool to improve I/O performance by alleviating interference's effects. In this context, the I/O scheduler's role is to reorder incoming requests to produce an access pattern with better performance. The overhead imposed by the scheduler cannot be large enough to surpass its performance improvements. In addition to the performance aspect, the scheduler should also try to keep *fairness* between applications and maintain an acceptable *response time* in the system.

Related work on I/O scheduling will be further discussed in Chapter 7.

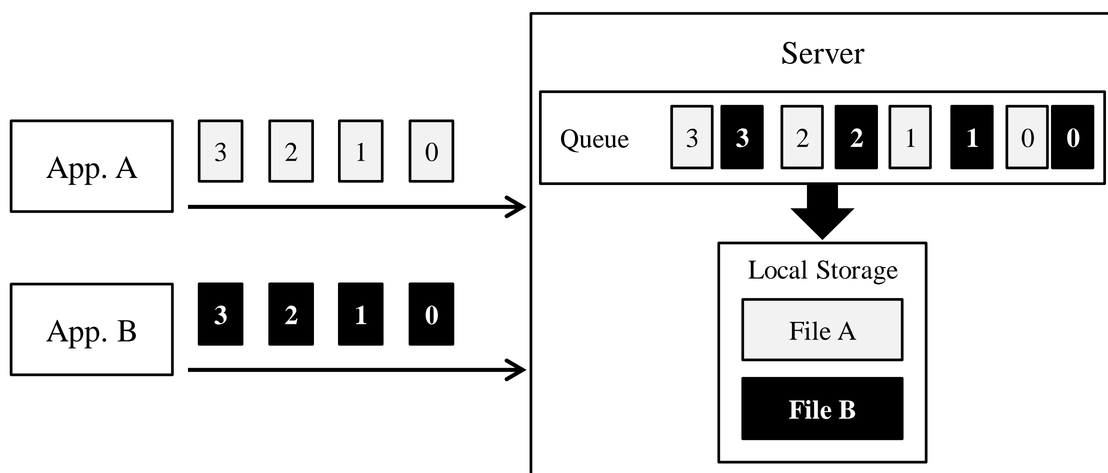


Figure 2.7: Interference on concurrent accesses to a PFS server.

2.5 Conclusion

It is accepted that, in order to achieve exascale, systems will need not only faster processing, but also a scalable high performance I/O infrastructure. In order to provide this infrastructure, a higher level of integration between different levels of the I/O stack is necessary to perform *smarter* optimizations that consider both applications' and systems' characteristics (DONGARRA et al., 2011).

This chapter presented basic concepts about most levels involved in I/O operations to parallel file systems. Figure 2.8 summarizes these levels presenting the typical data path between applications and parallel file systems' servers.

Applications may use I/O libraries to generate requests, or make them directly (through the POSIX API) to its lower levels. Files are accessed transparently, without difference between remote and local storage. The PFS client will be responsible to transfer requests to remote files to the appropriate parallel file system's servers. One application's request can generate several requests to the servers, depending on data striping and transmission size limit. Some PFS clients

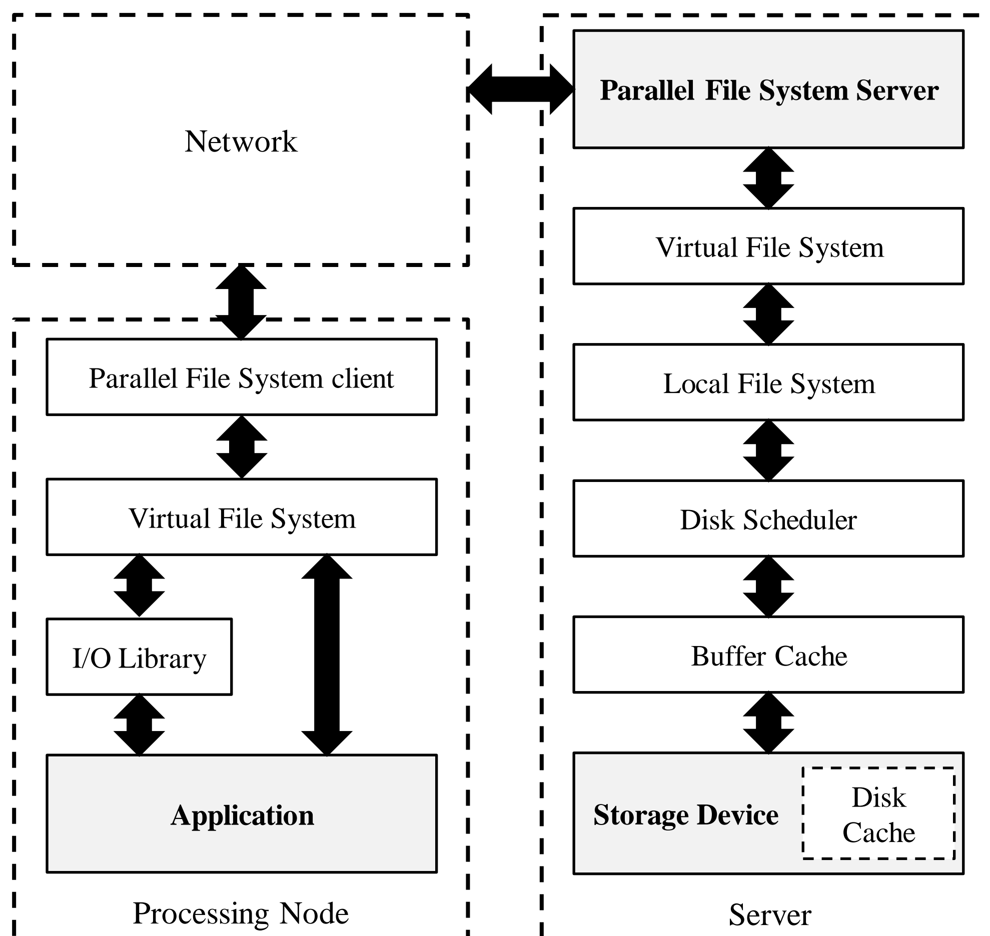


Figure 2.8: Data path between applications and parallel file systems' servers.

and I/O libraries can apply techniques to mask the cost of accessing the remote file system, such as data caching, read-ahead, and prefetching.

When the PFS server receives a request, it will process it by accessing a local file. This access is done through the local I/O stack, going through local file system, possibly by a disk scheduler, buffer cache, and finally to the storage device. The storage at the servers is typically based on either hard disk drives or solid state drives. Each of these devices has particular characteristics and performance behaviors.

The way applications perform I/O is called their access patterns. An application access pattern describes this applications' requests: how many requests, with which size, to how many files, with how much time between consecutive requests, to read or to write. Moreover, it considers how these requests are distributed inside the files: the file may be accessed sequentially, or with fixed-size gaps, or at random positions, etc.

There is no globally accepted notation for access patterns. For instance, if we described the FLASH application's access pattern, we would mainly say that:

- all its processes generate write requests to the same file;
- each process accesses sparse positions of this file;
- every two consecutive positions accessed by the same process have a fixed size gap between them;
- there is no intersection between different processes' accessed portions;
- globally, the whole file is accessed.

We could go into more or less detail on this access pattern, depending on the use we intend to give to this information.

One important conclusion to be drawn from what was presented in this chapter is that I/O performance is deeply affected by access patterns, i.e., to achieve good performance applications must perform their operations in the manner that is most suited to the used parallel file system. For this reason, several optimizations work to adjust applications' access pattern for improved I/O performance. These optimizations, which usually work on the I/O library level, apply techniques such as:

- requests reordering;
- requests aggregation;
- intra-node I/O scheduling;
- change the number of accessed files;
- align accesses with the PFS stripe size;

- work together with other application's processes; etc.

These optimizations work in the context of one application. Nonetheless, in cluster architectures, it is usual to have a storage infrastructure - with a parallel file system deployment - that is shared among the multiple applications that are using the cluster at a given moment. In this situation, applications' I/O performance will suffer from *interference*, as their requests arrive at the servers at the same time. I/O schedulers can improve performance by alleviating interference's effects. We focus on I/O scheduling to parallel file systems' servers, that work in the context of requests to files. In this scenario, the role of the I/O scheduling algorithm is to decide the order in which requests must be processed by the servers. While doing that, the scheduler can perform optimizations to improve performance, such as requests aggregation.

Access patterns' impact on performance is not the same for all devices, resulting on assumptions about performance no longer holding. The storage device's sensitivity to access sequentiality, for instance, defines the effectiveness of requests reordering performed by I/O schedulers. On the other hand, aggregating small requests into larger ones usually improves performance for all devices. In this context, I/O schedulers should adapt to the underlying I/O sub-system's characteristics.

Moreover, schedulers work on a small window of requests and have little information about applications, as this information is lost through the I/O stack. Better decisions could be made with more knowledge about applications' access patterns. Therefore, I/O schedulers should adapt to applications' access patterns.

The rest of this document presents our approach for providing I/O scheduling for parallel file systems with *double adaptivity*: to devices and applications.

3 AGIOS: A TOOL FOR I/O SCHEDULING ON PARALLEL FILE SYSTEMS

I/O schedulers found in the literature are usually specific to a given file system. Moreover, most of them impose specific file system configurations to work, such as a centralized metadata server (mode detail on related work will be presented in Chapter 7). These characteristics restrict their usability in new contexts and comparisons between them. Therefore, we needed a tool to study I/O scheduling adequately. Additionally to assisting this thesis' work, we wanted a tool that could be used by others to reproduce and expand our results.

For these reasons, we developed an I/O scheduling tool named *AGIOS*[1]. The main objectives for its development were to make it generic, non-invasive, easy to use, and to offer multiple choices on scheduling algorithms.

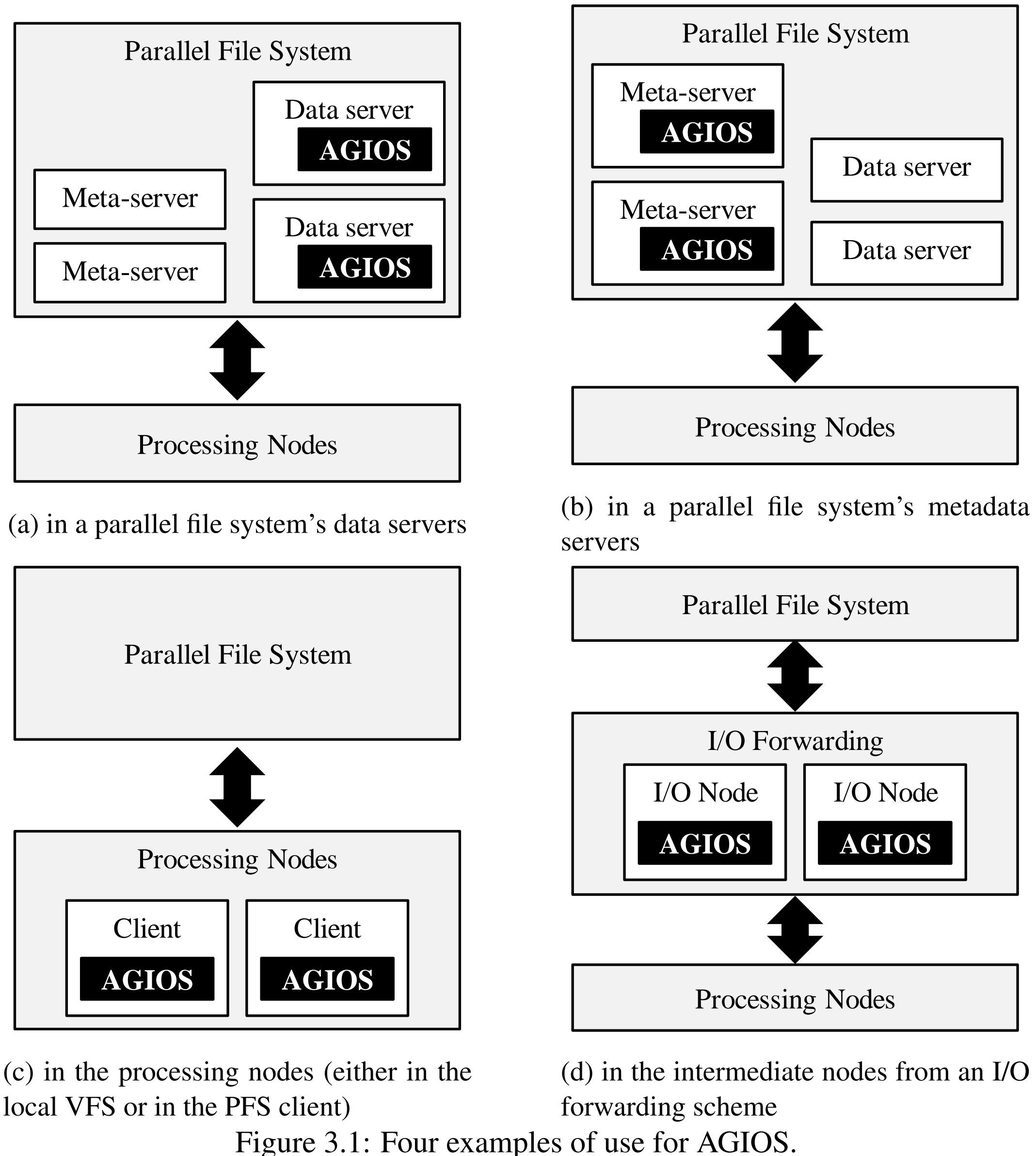
Although this thesis focuses on parallel file systems, we wanted to develop a tool that could be used by *any I/O service that treats requests at a file level*, such as a local file system or intermediate nodes in an I/O forwarding scheme. For this reason, AGIOS has a library implementation suitable for inclusion on most of today's parallel file systems' servers. On the other hand, AGIOS also offers a kernel module implementation that can be used, for instance, at the virtual file system layer. The I/O services that use AGIOS are called its "users".

Figure 3.1 presents four non exhaustive examples of use for AGIOS. All of the four presented placement options are I/O services that treat requests at a file level, and are places where it makes sense to use I/O scheduling. These placement options are not exclusive, in the sense that we could have all of them happening at the same time. In order to avoid creating a bottleneck, AGIOS instances (on different PFS servers, or at different levels of the I/O stack) are independent and do not make global decisions.

AGIOS exports a simple interface composed of four steps:

- **Initialization:** The user calls *agios_init* in order to initialize the AGIOS infrastructure. At this moment, the user must provide a callback function used to process a single request. If a function capable of processing a list of requests at once is available, the user should also provide it.
- **A new request arrives:** The user calls *agios_add_request* to transmit the request to the scheduler.
- **Requests are ready to be processed:** The scheduler passes the requests back to its user through the provided callback function. Therefore, the task of processing requests is left

[1] The tool's name - AGIOS - comes from "Application-Guided I/O Scheduler" and is Greek for "holy".



(a) in a parallel file system's data servers

(b) in a parallel file system's metadata servers

(c) in the processing nodes (either in the local VFS or in the PFS client)

(d) in the intermediate nodes from an I/O forwarding scheme

Figure 3.1: Four examples of use for AGIOS.

to the I/O services using the library. This keeps AGIOS's interface simple and generic.

- **Finalization:** The user calls *agios_exit* for cleaning up the tool's infrastructure.

Additional calls are provided to generate access statistics files and to reset statistics. An illustration of this interface is provided in Figure 3.2, that shows a situation where AGIOS is used by a parallel file system's data servers (scenario illustrated at Figure 3.1a). In step **1**, a server receives requests. It transmits the requests to the scheduler in step **2**. After applying a scheduling algorithm, AGIOS will give requests that are ready to be processed back to the server in step **3**. In step **4**, the server will process the requests. This whole process could also be happening in the other server, without affecting the first one.

Virtual requests obtained by aggregating single requests will be served together if the library user (the server) provided a callback for this functionality. Otherwise, the original requests

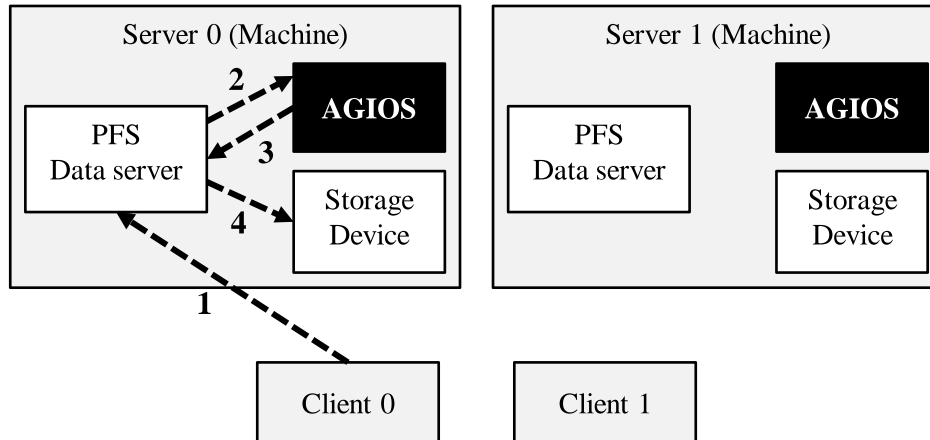


Figure 3.2: A request's path to a PFS server with AGIOS.

selected for aggregation will be served sequentially. However, even when it is not possible to effectively aggregate requests into larger ones, performance can still benefit from the execution of contiguous requests served in offset order instead of the original random order. Moreover, other levels of the I/O stack might aggregate these requests.

I/O scheduling is mainly meant for multi-application scenarios, aiming at avoiding the ill effects of interference in applications' performance. Additionally, different processing nodes can also interfere with each other even when executing the same application. Users such as parallel file systems' servers are often unable to determine from which application or process requests are coming, since this information is lost through the I/O stack. For this reason, the scheduler works in the context of files instead of applications. Moreover, the scheduler is able to provide performance improvements even on single-application scenarios.

3.1 Scheduling algorithms

As it will be later evidenced by a performance evaluation, different scheduling algorithms are good fits for different situations. In other words, no algorithm provides the best performance improvements in all cases. For this reason, we implemented five options in AGIOS: aIOLi, MLF, SJF, TO, and TO-agg. These algorithms were selected for their variety, in order to represent different situations and complement each other's characteristics. The next sections describe them, and Section 3.1.5 summarizes their main aspects.

3.1.1 aIOLi

We have adapted the *aIOLi* scheduling algorithm from Lebre et al. (2006). aIOLi is a quantum-based algorithm inspired in the classic Multilevel Feedback (MLF) from task scheduling (SILBERSCHATZ; GALVIN; GAGNE, 2013). It seeks to promote aggregation of requests into larger ones. A full explanation and discussion about this algorithm's characteristics can be found in the paper that describes it (LEBRE et al., 2006), but we can summarize it as follows:

- Whenever new requests arrive to the scheduler, they are inserted in the appropriate queue according to the file to be accessed. There are two queues for each file: one for reads, and another for writes.
- New requests receive an initial quantum of 0 bytes.
- Each queue is traversed in offset order and aggregations of contiguous requests are made. When an aggregation is performed, a *virtual request* is created, and this request will have a quantum that is the sum of its parts' quanta.
- All quanta (including the virtual requests' ones) are increased by a value that depends on its queue's past quanta usage.
- In order to choose a request to be served, the algorithm uses an offset order inside each queue and a FCFS criterion between different queues. Additionally, to be selected, the request's quantum must be large enough to allow its whole execution (it needs to match the request's size).
- The scheduler may decide to wait before processing some requests if a) a shift phenomenon is suspected or b) better aggregations were recently achieved for this queue (More information about these phenomena can be found in the paper). Whenever possible, this waiting time is overlapped with processing requests from other queues.
- After processing a request, if there is quantum left, other contiguous request from the same queue can be processed - given that they fit the remaining quantum. After stopping for a queue, its quanta usage ratio is updated. This scheduling algorithm works synchronously, in the sense that it waits until a virtual request is processed before selecting other ones.

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

queue (sorted by offset order). The former is expected to be:

$$O(M/S_{hash}) \tag{3.1}$$

where M is the number of files being concurrently accessed, and S_{hash} is the number of entries in the hash table. The time to find a request's place inside a queue is:

$$O(N_{queue}) \tag{3.2}$$

where N_{queue} is the number of requests in the largest queue. Selecting a request for processing, on the other hand, involves going through all queues:

$$O(2 \times M) \tag{3.3}$$

3.1.2 MLF

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

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

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

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

3.1.3 SJF

We have also developed for our study a variation of the *Shortest Job First (SJF)* scheduling algorithm (SILBERSCHATZ; GALVIN; GAGNE, 2013) that performs aggregation of requests. Its implementation also uses two queues per file and considers requests from each queue in offset order. The selection of the next request is done by going through the queues and selecting requests from the smallest one (considering each queue's total size, i.e. the sum of all its requests' sizes).

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

3.1.4 TO and TO-agg

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

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

$$O(N) \tag{3.4}$$

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

3.1.5 Summary of the scheduling algorithms

We have included five I/O scheduling algorithms in our tool: aIOLi, MLF, SJF, TO and TO-agg. Their main characteristics are summarized in Tables 3.1 and 3.2. Most of them work on at least one of two common assumptions about I/O performance:

1. Sequential accesses perform better than accesses that are randomly located inside the

files.

2. It is better to perform a smaller number of large accesses than a larger number of small ones.

Table 3.1: Summary of the presented scheduling algorithm's main characteristics - part 1.

	offset order	aggregation	waiting times	synchronization
aIOLi	✓	✓	✓	✓
MLF	✓	✓	✓	
SJF	✓	✓		
TO				
TO-agg		✓		

Table 3.2: Summary of the presented scheduling algorithm's main characteristics - part 2. M is the number of files, N is the number of requests and N_{queue} is the number of requests in the largest queue.

	global criterion	queues	cost for including	cost for selecting
aIOLi	FCFS	$2 \times M$	$O(N_{queue} + M)$	$O(2 \times M)$
MLF	None	$2 \times M$	$O(N_{queue} + M)$	$O(1)$
SJF	Size	$2 \times M$	$O(N_{queue} + M)$	$O(2 \times M)$
TO	FCFS	1	$O(1)$	$O(1)$
TO-agg	FCFS	1	$O(N)$	$O(1)$

aIOLi, MLF, and SJF try to process requests in offset order to generate access patterns that access each file sequentially. This is done by separating them in queues and selecting from these queues following offset order. However, they alternate between different queues, generating interleaved contiguous accesses to different files. In this situation, it is important to aggregate requests, so the interleaving will have less effect on the resulting access pattern. From the discussed algorithms, all but TO seek to aggregate requests. From these, aIOLi is the one expected to aggregate more, but at the cost of a high impact of scheduling overhead, because of its synchronous approach: aIOLi waits until previous requests were served before scheduling more, possibly giving time to new requests arrive and create better aggregations. However, this synchronous approach increases the impact of the scheduling algorithm's overhead on the time to serve requests.

Although the interleaving of requests from different queues done by aIOLi, MLF, and SJF affects these algorithm's ability to generate contiguous access patterns, it is important for main-

taining fairness between applications. From the discussed algorithms, aIOLi is expected to provide the best fairness while keeping response time, due to it using FCFS criterion between queues. On the other hand, MLF's lower overhead comes at the cost of decreased fairness. The SJF algorithm favors the smallest queue, possibly keeping it the smallest one and leading the other queues to starvation.

Nonetheless, this notion of fairness provided by alternating between queues for request selection is associated with files and not necessarily with applications. For instance, if an application accesses 500 files of $1MB$ each, while another is accessing one file of $500MB$, it is possible that the first one will be favored despite them both accessing the same amount of data. This is a limitation of the scheduling being at file level, since it typically do not have information of which applications are generating which requests. Next chapter will discuss strategies to include more information about applications on the scheduler, and it could be used to improve fairness. However, this thesis focuses on performance and thus we do not present this kind of fairness analysis. This is left as future work, as discussed in Chapter 8.

Keeping semantics between concurrent accesses is an usual concern of I/O services like local file systems. On the other hand, some parallel file systems such as PVFS favor performance by leaving the task of avoiding conflicting accesses to its users (LATHAM et al., 2004). AGIOS has a basic control of concurrent accesses that blocks read requests that are more recent than writes currently on queue (and vice versa). This functionality can be disabled for performance. All experiments presented in this document were obtained without semantics control.

3.2 Performance evaluation

We conducted a performance evaluation with our I/O scheduling tool aiming at generating understanding about I/O scheduling as a tool to improve performance, providing results that guide the work through the rest of this thesis. This evaluation's main objectives are:

- to identify the situations where I/O scheduling is able to improve performance and to understand what makes them suitable for this technique;
- to identify which of the implemented scheduling algorithms is the best fit for each scenario.

To meet these requirements, it is important for the performance evaluation to include multiple platforms with diverse characteristics. Because of the implemented algorithm's assumptions on performance, it is desirable to evaluate them on different storage devices, with different lev-

els of sensitivity to access sequentiality. Additionally, the used benchmarks' access patterns must also cover a big variety of possible scenarios. In this thesis, we represent access patterns through the following list of relevant aspects, previously discussed in Chapter 2:

- spatial locality (spatiality);
- number of files being concurrently accessed;
- number of concurrent applications;
- number of processes concurrently performing I/O;
- size of requests.

Moreover, the systems' configuration should evidence the cost of performing I/O to storage devices, minimizing cache effects. The scheduling algorithms' costs should also be as present as possible, in order to adequately compare them.

We have chosen to use *makespan* as the metric for our performance evaluation, since it represents the total time to process the whole workload from the file system's point of view. Therefore, our scheduling aims at reducing the global time, even if it means increasing some applications' individual execution times. Our metric does not reflect fairness or response time, as we are not focusing on these aspects.

The results obtained in this step will be incorporated in AGIOS for automatically selecting which I/O scheduling algorithm to use in different situations. This process will be described in Chapter 6.

Since this thesis focuses on I/O scheduling for parallel file systems, we included AGIOS in a parallel file system's data servers. For proof-of-concept purposes, we present our library's usage with dNFSp (AVILA et al., 2004), an NFS-based parallel file system composed of several metadata and data servers (also called "IODs"). The distributed servers are transparent to the file system client, which accesses the remote file system through a regular NFS client.

We integrated AGIOS within the IOD code, so each IOD contains an independent instance, as illustrated in Figure 3.1a. IODs use their machines' local file systems to store data. Each file stored in the PFS corresponds to one file in each data server, where its stripes are sequentially stored. This approach is similar to what is done by other parallel file systems, such as PVFS.

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

requests will be served in the order defined by the scheduler.

Sections 3.2.1 and 3.2.2 describe the method used for the library's evaluation using dNFSp. Section 3.2.3 presents the obtained results. Finally, Section 3.3 summarizes this chapter's main contributions.

3.2.1 Experimental platforms and configuration

We executed tests on four clusters from Grid'5000 (BOLZE et al., 2006), described in Table 3.3. These systems were selected by their variety on storage devices: we tested with SSDs, HDDs and RAID arrays. Moreover, these devices present diverse sequential to random throughput ratios, as shown in Table 3.4 for 8MB requests. This size is relevant because it is the transmission size between clients and servers in our dNFSp deployment and hence all requests arriving at the servers have size up to 8MB. A high ratio means that accessing files sequentially is several times faster than accessing them randomly. On the other hand, a ratio smaller than 1 means that accessing randomly is faster. More detail on calculating the sequential to random throughput ratio and of these storage devices will be provided in Chapter 5.

Table 3.3: Platforms used for AGIOS's performance evaluation.

Cluster	Nodes	Node Configuration		
		Processor	RAM	Storage Device
Pastel @ Toulouse	140	2× 2-core AMD Opteron	$8GB$	HDD
Graphene @ Nancy	144	4-core Intel Xeon	$16GB$	HDD
Suno @ Sophia	45	2× 4-core Intel Xeon	$32GB$	RAID-0
Edel @ Grenoble	72	2× 4-core Intel Xeon	$24GB$	SSD

Table 3.4: Sequential to Random Throughput Ratio with $8MB$ requests for all tested platforms described in Table 3.3.

	Pastel	Graphene	Suno	Edel
Write	21.29	15.12	8.17	0.66
Read	38.91	40.68	25.46	2.37

Despite the number of nodes shown in Table 3.3 for these clusters, it is unusual to have all of them available: some nodes may be unreachable due to technical problems, for instance. Therefore, we used as many nodes as possible for our analysis. Table 3.5 presents the number of client machines and servers used in our tests. One of dNFSp's data servers shares its machine with the metadata server. This configuration is illustrated in Figure 3.3. AGIOS is used by the data servers only, and not by the metadata server (despite the fact that it shares a machine with a data server).

Table 3.5: Nodes used for running our experiments.

	Pastel	Graphene	Suno	Edel
# IODs	4	4	4	4
# Client machines	16	32	16	32

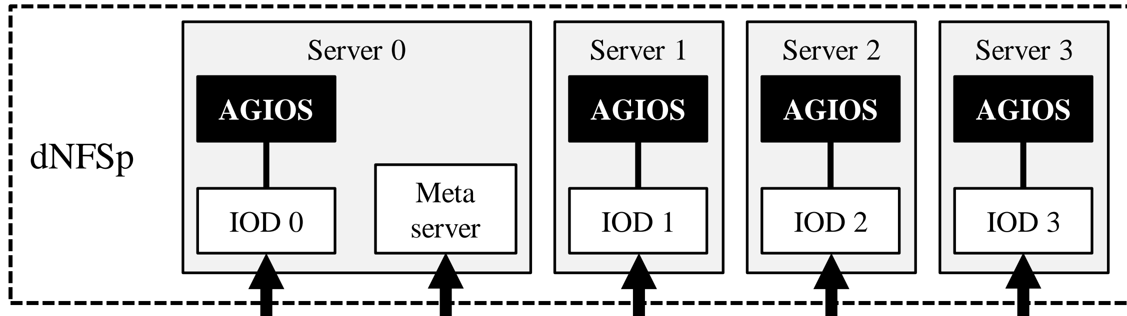


Figure 3.3: dNFSp configuration with 4 servers and 1 metadata server.

Although dNFSp allows for multiple metadata servers, we have chosen to use a single one in our experiments. In a large scale, this centralized server could become a bottleneck and impair performance. It is important to notice, however, that we are evaluating the scheduler on the data servers and with workloads that are not intensive on metadata operations, so this choice is not expected to influence our analysis.

The amounts of data accessed in our experiments were limited by the nodes' storage capacity and by the NFS protocol's limitations. In order to minimize cache effects on results and evidence the cost of accessing the storage devices, we generated direct requests (by using the *O_DIRECT* flag) and decreased the *dirty_background_ratio* and *dirty_ratio* Linux memory management parameters to 5% and 10% respectively. All nodes have the Debian 6 operating system with kernel 2.6.32. Both dNFSp and AGIOS were compiled with gcc 4.4.5. Virtual memory page size is 4MB.

3.2.2 Evaluated access patterns and experimental method

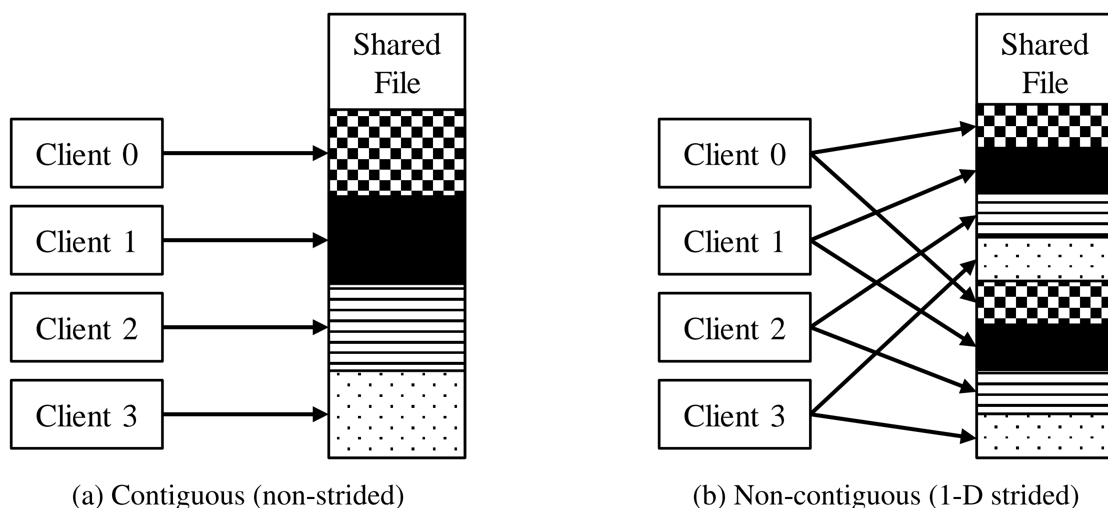
We developed a set of tests using the MPI-IO Test benchmarking tool[2]. They explore the previously discussed list of relevant access pattern aspects:

1. Spatial locality: if applications issue requests that are contiguous or non-contiguous. In our tests, the non-contiguous case is represented by a 1-D strided access pattern. Contiguous access patterns to a shared file mean that each process has an exclusive portion of this file, i.e. there is no overlapping between different processes' accessed data. Figure 3.4

[2] <http://institute.lanl.gov/data/software/mpi-io>

illustrates this situation with 4 clients.

2. Number of files: processes either share a file or have independent files (one per process). This configuration is done at application level, thus if four applications execute concurrently with a shared file access pattern, four files are currently being accessed on the file system.
3. Single or multi-application scenarios: we present results for single and multi-application scenarios. The multi-application case is represented by executing four instances of the same benchmark concurrently. In these cases, processing nodes are split evenly among the different instances.
4. Number of processes per application: how many processes perform I/O. We repeat our tests for different application sizes: 8, 16, and 32. Since the amount of data accessed by each process is constant, *the total amount of data grows with the number of processes*.
5. Size of requests: if requests are small (smaller than the file system's stripe size) or large (larger than the stripe size and large enough so all file system servers will have to be contacted in order to process it). These definitions of small and large requests follow what is presented by Byna et al. (2008), as discussed in Chapter 2. Table 3.6 summarizes the amounts of data accessed on different tests. The stripe size used by dNFSp is 32KB.



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

Figure 3.4: Spatial locality aspect with a shared file.

The benchmarking tool uses synchronous I/O operations, so requests from a process have to be served before the next batch can be sent to the server. However, the transmission size is limited to 8KB. Therefore, application's requests are divided in multiple actual requests: in small tests, each process will issue 2 requests at once; in large tests, 32. We believe that greater performance improvements could be achieved if our tests issued only asynchronous operations, since applications would not be so affected by the scheduling algorithm's overhead.

Table 3.6: Amount of data accessed on this chapter's tests.

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

Nonetheless, we have chosen the synchronous approach since it is more challenging to the scheduler, because its overhead has a higher impact, and little delays on processing a request can lead to longer execution times.

From each application execution, we take the completion time of the slowest process, since it defines this application's execution time. We take the maximum between concurrent applications' execution times because we are interested in reducing the *makespan*, as previously discussed.

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

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

The next section presents obtained results with AGIOS and dNFSp using all the presented scheduling algorithms. Since the volume of tests is considerably large, showing all of them would compromise this chapter's readability. Therefore only a subset of them is shown to

illustrate the discussions. Appendix A lists all obtained results.

3.2.3 Performance results

3.2.3.1 Performance results with TO

Since dNFSp already has a timeorder scheduling algorithm, results with AGIOS' TO (a simple timeorder) only evidence the library's overhead. Since TO has costs $O(1)$ for both including and selecting requests (see Table 3.2), this overhead is not expected to be important.

Table 3.7 presents the average absolute differences between results obtained without AGIOS - with dNFSp's default timeorder algorithm - and with the TO scheduling algorithm. The absolute difference in performance caused by the use of AGIOS with TO is obtained as follows:

$$|PerformanceDifference| = \frac{|Time_{noAGIOS} - Time_{AGIOS}|}{Time_{noAGIOS}} \tag{3.5}$$

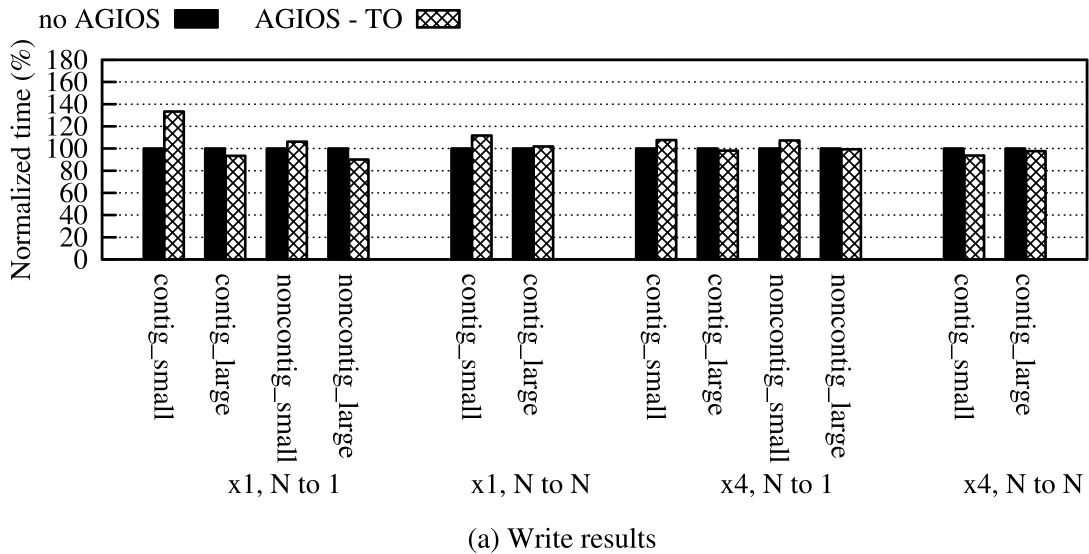
The average of the absolute differences from all tested access patterns was taken to generate the numbers in Table 3.7. For most access patterns, only a negligible difference (under 10%) was observed, but, for a few of them, the overhead had a larger impact.

Table 3.7: Average absolute differences between TO and dNFSp' timeorder algorithm.

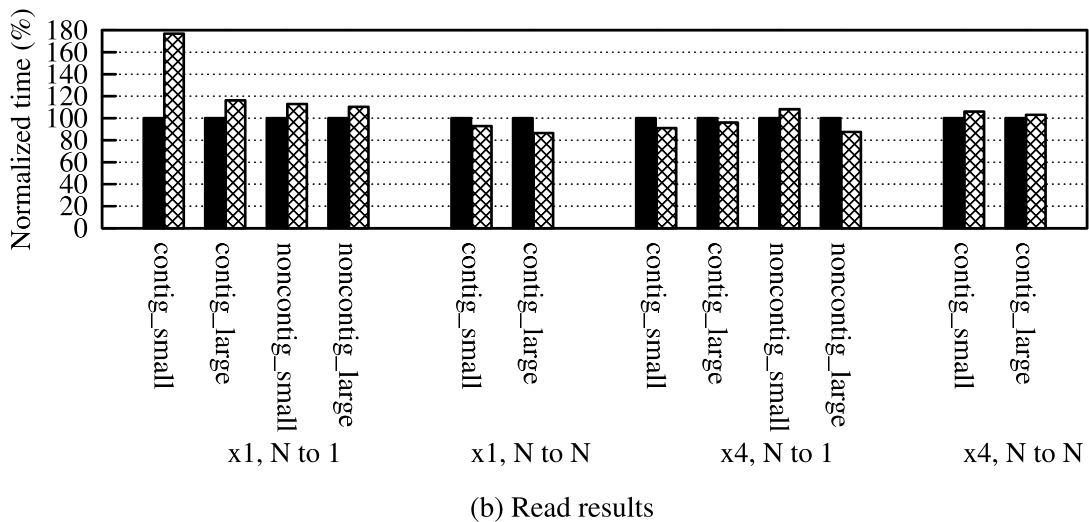
	Pastel	Graphene	Suno	Edel
Write	9.09%	8.8%	11.19%	9.37%
Read	17.66%	7.71%	26.94%	12.04%

Figure 3.5 illustrates this by presenting results obtained in the Pastel cluster for tests with 16 processes per application, normalized by the time without AGIOS. "N to 1" represents the shared file approach, and "N to N" the file per process one. "x1" represents the single application scenario, and the multi-application one is represented by "x4". In general, experiments where processes from a single application share a file are more sensitive to the library's overhead, since these tests have the smallest execution times (especially read tests).

Since the time obtained with TO was usually larger than what was obtained without AGIOS, we use the latter to represent the timeorder scheduling algorithm in the rest of this chapter. Using TO's results could mask the results by showing larger performance improvements by other scheduling algorithms.



(a) Write results



(b) Read results

Figure 3.5: Results with TO over not using AGIOS in the Pastel cluster - tests with 16 processes.

3.2.3.2 Performance results for single application with shared file

The experiments where a single application's processes share a file represent the smallest workloads (see Table 3.6). For these tests, a strong impact on performance is expected due to the scheduler's overhead.

Since during each test only one file is accessed, aIOLi, MLF, SJF, and TO-agg present practically the same costs for including and selecting requests (see Table 3.2). This situation maximizes the number of requests per queue for aIOLi, MLF, and SJF, as all requests belong in the same queue. Moreover, the scheduling algorithms' global criteria do not apply, since there is only one queue.

Experiments that issue large requests provide more aggregation opportunities, since large applications' requests will be split into several contiguous actual requests to the servers because of the file system's transmission size limit. Performing more aggregations, the scheduling algo-

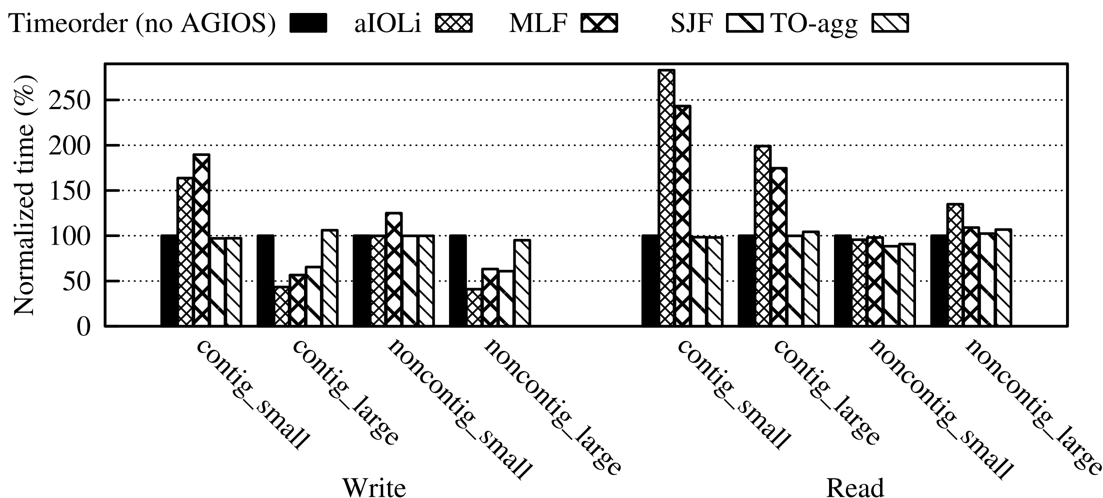


Figure 3.6: Results with a single application and the shared file approach in the Graphene cluster (tests with 8 processes).

rithms' effect on performance might be able to surpass their overheads. Additionally, because of the striping process, in contiguous access patterns, requests from different clients are not contiguous at the servers, hence strided tests provide more aggregation opportunities than contiguous ones. Furthermore, read tests have smaller execution times and hence are more affected by scheduling overhead.

Considering the clusters with HDDs (Pastel and Graphene), the scheduler was only able to improve performance of *write operations* in the *large requests* access patterns. Figure 3.6 presents the results obtained in the Graphene cluster with 8 processes, normalized by the time observed for the file system without AGIOS (timeorder algorithm). Pastel presented similar behaviors, as did results for Graphene with 16 and 32 processes. The cluster with RAID-0 (Suno) also presented the best results for large write requests. However, differently from the clusters with HDDs, all other situations presented performance improvements in the tests with 32 processes (tests with 8 and 16 behaved similarly to the clusters with HDDs).

The TO-agg scheduling algorithm is the simplest approach, since it does not seek at executing requests in offset order, but only performs aggregations. This resulted in performance *decreases* of up to 28% in Pastel, in small differences under 10% in Graphene, and in performance *improvements* of up to 32% in Suno (for 32 processes, small differences under 10% for 8 and 16 processes).

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

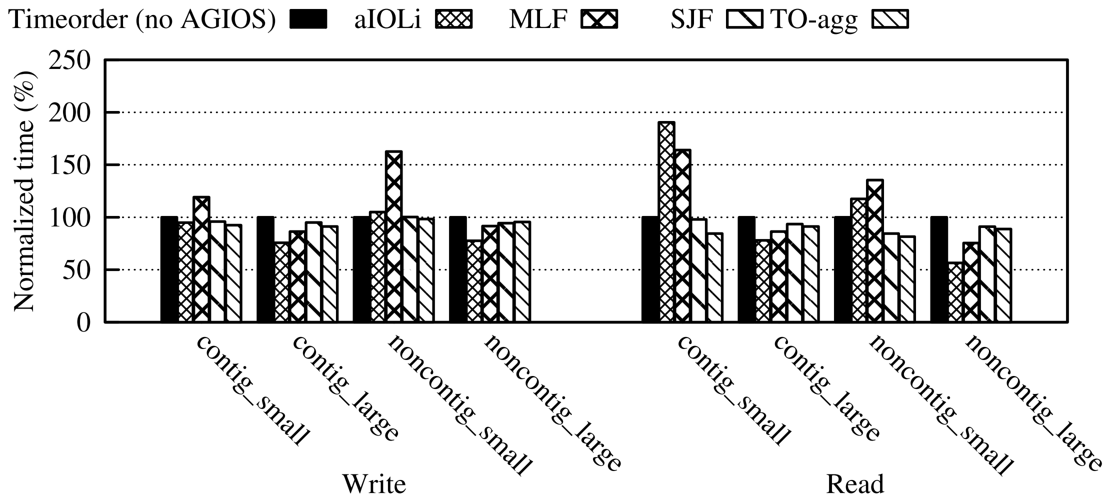


Figure 3.7: Results with a single application and the shared file approach in the Edel cluster (tests with 16 processes).

tions were provided by the aIOLi algorithm. However, for read tests - where the overhead is more significant - MLF performed better than aIOLi, as the former is expected to induce less overhead.

Performance improvements obtained for Pastel are the smallest despite this platform's storage devices having the highest sequential to random throughput ratio for writes (see Table 3.4). One possible reason is that the Pastel cluster is approximately three years older than the other two and its nodes have less memory and processing power. In this situation, scheduling algorithms' costs may become more important in the resulting performance. This also explains why large tests in Suno had better results, since this cluster has the largest amount of memory per node and thus is expected to be less affected by scheduling overhead.

In the experiments where there is only one queue being accessed, the main difference between MLF's and SJF's executions is that SJF does not have waiting times. aIOLi's and MLF's waiting times are expected to improve aggregations. SJF provided performance *decreases* of up to 23% in Pastel, performance *increases* of up to 39% in Graphene (only in the situations where aIOLi and MLF also improved performance) and performance *increases* of up to 32% in Suno (as TO-agg, for tests with 32 processes. It provided small differences under 10% for smaller tests).

In the cluster with SSDs (Edel), tests' execution times were 2 to 4 times *longer* than what was observed in the other three clusters. Therefore, in these tests scheduling overhead had less impact, and some performance improvements were obtained even for read operations. Figure 3.7 presents these results for the tests with 16 processes (tests with 8 and 32 processes presented similar behaviors).

Although Edel's storage devices have the lowest sequential to random throughput ratios, performance improvements can still be obtained by request aggregations. TO-agg was able to improve performance in up to 38% for tests with large requests (where there are more aggregation opportunities), and to *decrease* performance in up to 48% for small requests. SJF improved performance in up to only 19% (its improvements stayed under 10% for most cases) by performing aggregations 13% smaller than TO-agg on average. Differently from what happened in the other platforms for read operations, aIOLi performed better than MLF in most tests. aIOLi and MLF were able to outperform TO-agg for large requests access patterns. In these cases, aIOLi provided the *best results*, increasing performance in up to 44%. On the other hand, for tests that issue small requests, aIOLi's and MLF's effects on performance did not surpassed their overheads, and they *decreased* performance in up to 171%.

Summarizing the results for the *single application scenario with shared file approach*, the best choices in scheduling algorithm are:

- In the cluster with HDDs (*Pastel*) for *read* operations, the best choice would be to use the simple timeorder algorithm TO, since no tested algorithm provided performance improvements. On the other hand, as discussed in Section 3.2.3.1, AGIOS' TO decreased performance in these cases, so, if possible, the use of AGIOS should be avoided in Pastel for this access pattern. On the other hand, using SJF or TO-agg could be a good compromise as they provide only small performance decreases (around 13%).
- The same decision applies in the other cluster with HDDs (*Graphene*), where both SJF and TO-agg only provided negligible differences under 10% for read operations.
- In *Pastel* for *write* operations, the best algorithm for *large* requests would be aIOLi. For *small* requests access patterns, the only solution that does not decrease performance comes from avoiding AGIOS.
- Similarly, in *Graphene*, aIOLi is the best choice for applications that issue *large* requests, but using it for *small* requests would result in performance decreases. An alternative would be to use SJF, which provided only negligible performance differences in the situations where aIOLi impaired performance. On the other hand, for *large* requests, SJF's performance improvements are smaller than aIOLi's.
- In the cluster with RAID-0 (*Suno*), both SJF and TO-agg offer good options, since they provided performance improvements in most situations (with 32 processes, and negligible differences for 8 and 16 processes). However, aIOLi is the best solution for tests that issue *large write* requests (including for tests with 8 and 16 processes).

- For *Edel*, the cluster with SSDs, aIOLi is also the best solution for *large* requests access patterns, and SJF would be the right choice for *small* requests' ones.

3.2.3.3 Performance results with multiple applications and the shared file approach

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

In the clusters with HDDs (Pastel and Graphene), TO-agg and SJF provided negligible performance differences for most cases, and performance *decreases* of up to 21% and up to 16% (only in Pastel), respectively, mainly for small requests. aIOLi and MLF were also unable to surpass their overheads and improve performance. In Pastel, they provided performance improvements of up to 14%, but smaller than 10% for most cases. In Graphene, aIOLi *decreased* performance for tests that issue *contiguous small read* requests in up to 54%, and presented only negligible differences for all other tests. MLF also decreased performance for contiguous small read requests access patterns, but *increased* performance by up to 17% for tests that issue non-contiguous large read requests.

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

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

Summarizing the results obtained for the scenarios with multiple applications, where each application's processes share a file, the best choices in scheduling algorithm would be:

- In *Pastel* (with HDDs), where no scheduling algorithm was able to improve performance, the best choice would be to use aIOLi or MLF, since they did not decreased performance

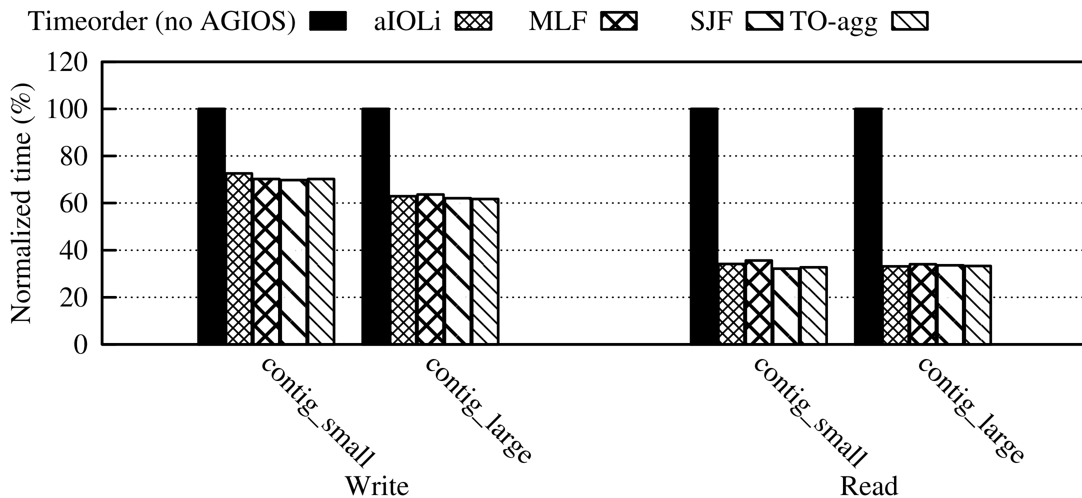


Figure 3.8: Results with a single application and the file per process approach in Pastel with 32 processes

significantly in any case. On the other hand, in *Graphene*, aIOLi and MLF decreased performance of *contiguous small read* operations. Therefore, MLF would be the right choice for all other situations, and for contiguous small requests the ideal would be to avoid AGIOS (the alternative is to use SJF or TO-agg, that provide slight decreases in performance).

- For the cluster with RAID-0 (*Suno*), any of the four tested algorithms would be a good choice (mainly SJF and TO-agg).
- In *Edel* (with SSDs), aIOLi is the best choice for *large read* requests access patterns. TO-agg would provide good results in some cases for *small read* operations, and SJF (or no AGIOS) is the right choice for tests that issue *write* requests.

3.2.3.4 Performance results for single application with the file per process approach

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

The best performance improvements for these experiments were observed in the Pastel cluster: up to 38% for write operations and up to 68% for reads with 32 processes. This situation is presented in Figure 3.8. In general, the larger the number of processes, the more performance was increased. In these tests, aIOLi performs slightly better than MLF, but there is no significant difference between the four scheduling algorithms.

Among the experiments with large workloads, in many cases better results were obtained for read tests than write ones. One reason for this is that all platforms' sequential to random throughput ratios are higher for reads than writes, as shown in Table 3.4. Therefore, reordering requests to generate sequential access patterns at the server has a larger impact for read operations.

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

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

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

Summarizing the results with a single application and file per process approach, the right choice in scheduling algorithm for each situation would be:

- In *Pastel*, all scheduling algorithms provided good performance results for all test cases, so any of them would be a good choice (especially aIOLi).
- On the other hand, in *Graphene* and in *Suno*, SJF is the best choice. It provides performance improvements for all situations in Suno, and in Graphene for large write requests. For other access patterns in Graphene, SJF results in negligible differences only.
- In the cluster with SSDs (*Edel*), TO-agg provides the best results.

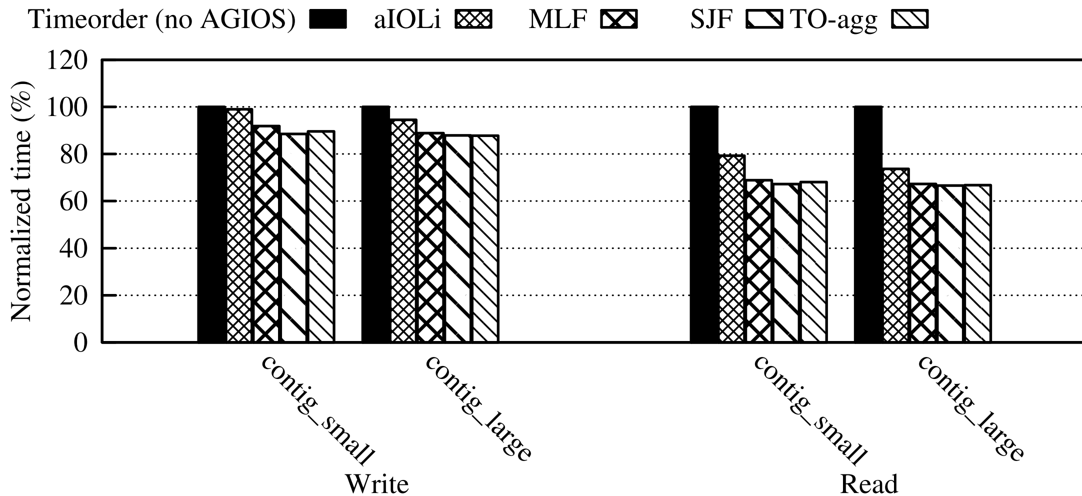


Figure 3.9: Results with four applications and the file per process approach in Suno with 32 processes.

3.2.3.5 Performance results with multiple applications and the file per process approach

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

In Pastel and Graphene, for most cases all scheduling algorithms resulted in small negligible differences. aIOLi improved performance in read operations with 8 processes per application by 16% in Pastel, and SJF improved by 16% with large read operations with 8 processes per application in Graphene. Performance was *decreased* slightly for write operations in some cases in Graphene (with aIOLi, MLF and TO-agg).

For the Suno cluster, the one expected to be less affected by scheduling overhead, results were similar to what was obtained for the previous set of tests (in Section 3.2.3.4). Nonetheless,the observed improvements were smaller: up to 14% for write operations and up to 33% for reads. Figure 3.9 presents the results obtained in Suno with 32 processes. aIOLi provided slightly worse results than the others, and SJF obtained the best results (with only a small difference over MLF and TO-agg).

In Edel, for write operations most cases resulted in only negligible differences. Nonetheless, aIOLi decrease the performance by up to 19%, MLF by 17%, and TO-agg by 15% (isolated cases only). In tests that generate read requests, aIOLi provided performance improvements of up to 24% for 8 and 16 processes per application, and TO-agg improved performance for

32 processes per application in up to 25%. MLF outperformed aIOLi for 32 processes per application providing improvements of up to 20%.

Therefore, summarizing the results with multiple applications and the file per process approach, the best choices in scheduling algorithm are the same than what was observed in the last section for Pastel (aIOLi), Graphene (SJF), and Suno (SJF). Nevertheless, in Edel, SJF is the best choice for write operations, since it did not decrease performance in any case (only providing negligible differences). For read operations, good algorithm choices for Edel would be aIOLi for 8 and 16 processes per application and TO-agg for 32.

The next section will conclude this Chapter by summarizing its contributions.

3.3 Conclusion

This chapter presented AGIOS, a scheduling tool for I/O services that handle requests at files level. Unlike other I/O schedulers found in the literature, AGIOS aims at being generic, non-invasive, and easy to use. Moreover, it provides five options in scheduling algorithms: aIOLi, MLF, SJF, TO, and TO-agg.

These algorithm's characteristics were discussed and their differences evidenced. All algorithms except TO seek to aggregate contiguous requests. aIOLi and MLF may decide to wait before processing a request in order to perform better aggregations. aIOLi, MLF, and SJF try to process requests to the same file in offset order. aIOLi, TO, and TO-agg try to keep a global FCFS criterion, while SJF prioritizes smaller queues. MLF does not have a global criterion in order to decrease its cost for making decisions.

In this thesis, we focus on AGIOS' usage to schedule requests to a parallel file system's data servers. This was illustrated by including our tool in an NFS-based file system (dNFSp). AGIOS was included in the data servers, and each server has an independent instance, hence there is no global coordination.

An extensive evaluation was conducted to understand the performance behavior of this approach with the implemented scheduling algorithms. This evaluation was done on four clusters, representing different alternatives in storage device: HDDs, RAID-0, and SSDs. Multiple access patterns were used, with aspects such as: spatiality (contiguous or 1-D strided accesses), request size (small or large), number of applications, number of processes per application, number of files accessed (shared file or file per process), etc.

The obtained results have evidenced that the scheduling algorithms' positive effect on performance have to surpass their overhead in order to achieve good results. In small workloads

(the workloads with shared files), tests that issue read requests are more sensitive to this overhead, since these tests' execution times are shorter. On the other hand, the best results with larger workloads (workloads with the file per process approach) were obtained for read tests. We believe this happens because, in general, the scheduling algorithms were able to aggregate more in read tests than on write ones for the large workloads, therefore generating a stronger positive impact in performance. Moreover, all used platforms' sequential to random throughput ratios are higher to read operations than to writes, indicating that reads are more affected by requests reordering.

Some access patterns (like the contiguous small requests one) are less prone to performance improvements, since they provide less aggregation opportunities. This difference between different spatialities and request sizes are especially important for the smaller workloads. Aggregations of contiguous requests play an important role in I/O scheduling results, as illustrated by results obtained for Edel (the cluster with SSDs). Despite this platform's storage devices presenting small sequential to random throughput ratios, performance improvements caused by request aggregations were observed.

The scheduling overhead increases with the number of files being concurrently accessed. For this reason, for good results to be achieved in situations where a large number of files is being concurrently accessed, the workload must be large enough to provide enough opportunities for performance improvements.

Among the scheduling algorithms, aIOLi usually provides the best results. However, it also provides the worst ones in some situations, since it has the highest overhead of the tested algorithms. TO-agg and SJF usually represent good alternatives, since they induce smaller overheads. From all tested access patterns and platforms, all five scheduling algorithm appear at least once as the best fit for a specific situation. This indicates that there is no algorithm that provides the best performance for all situations. The best fit depends on both access patterns and platforms. Table 3.8 visually summarizes the scheduling algorithm choices to all tested situations.

In addition to their storage devices' differences, the platform nodes' capabilities also play an important role. The Suno cluster, which does not have the highest sequential to random throughput ratios between the tested platforms, had performance improvements in situations where others did not. Having more memory and processing power in its nodes, this cluster is less affected by scheduling algorithms' overheads.

These results provide a base for the rest of this thesis. They evidenced that performance provided by I/O scheduling depends on both applications' and storage devices' characteristics.

Table 3.8: Best choices in scheduling algorithms to all situations tested in this chapter.

Access Pattern					Pastel	Graphene	Suno	Edel
x1	N to 1	contig	small	write	**no AGIOS**	**SJF**	**SJF**	**SJF**
x1	N to 1	contig	small	read	**no AGIOS**	**SJF**	**SJF**	**SJF**
x1	N to 1	contig	large	write	**aIOLi**	**aIOLi**	**aIOLi**	**aIOLi**
x1	N to 1	contig	large	read	**no AGIOS**	**SJF**	**SJF**	**aIOLi**
x1	N to 1	noncontig	small	write	**no AGIOS**	**SJF**	**SJF**	**SJF**
x1	N to 1	noncontig	small	read	**no AGIOS**	**SJF**	**SJF**	**SJF**
x1	N to 1	noncontig	large	write	**aIOLi**	**aIOLi**	**aIOLi**	**aIOLi**
x1	N to 1	noncontig	large	read	**no AGIOS**	**SJF**	**SJF**	**aIOLi**
x1	N to N	contig	small	write	**aIOLi**	**SJF**	**SJF**	**TO-agg**
x1	N to N	contig	small	read	**aIOLi**	**SJF**	**SJF**	**TO-agg**
x1	N to N	contig	large	write	**aIOLi**	**SJF**	**SJF**	**TO-agg**
x1	N to N	contig	large	read	**aIOLi**	**SJF**	**SJF**	**TO-agg**
x4	N to 1	contig	small	write	**aIOLi**	**MLF**	**SJF**	**SJF**
x4	N to 1	contig	small	read	**aIOLi**	**no AGIOS**	**SJF**	**TO-agg**
x4	N to 1	contig	large	write	**aIOLi**	**MLF**	**SJF**	**SJF**
x4	N to 1	contig	large	read	**aIOLi**	**MLF**	**SJF**	**aIOLi**
x4	N to 1	noncontig	small	write	**aIOLi**	**MLF**	**SJF**	**SJF**
x4	N to 1	noncontig	small	read	**aIOLi**	**MLF**	**SJF**	**TO-agg**
x4	N to 1	noncontig	large	write	**aIOLi**	**MLF**	**SJF**	**SJF**
x4	N to 1	noncontig	large	read	**aIOLi**	**MLF**	**SJF**	**aIOLi**
x4	N to N	contig	small	write	**aIOLi**	**SJF**	**SJF**	**SJF**
x4	N to N	contig	small	read	**aIOLi**	**SJF**	**SJF**	**aIOLi**
x4	N to N	contig	large	write	**aIOLi**	**SJF**	**SJF**	**SJF**
x4	N to N	contig	large	read	**aIOLi**	**SJF**	**SJF**	**aIOLi**

The next two chapters describe how we obtain and classify information about applications and devices, and Chapter 6 applies these techniques and the results from this chapter in AGIOS to select the best fit in scheduling algorithms automatically.

4 APPLICATION-GUIDED I/O SCHEDULING

Chapter 3 presented AGIOS, our tool for I/O scheduling, and demonstrated its use on parallel file systems' data servers. The obtained results indicate that it is important for I/O schedulers to adapt to applications' characteristics to improve performance. This chapter discusses our approach to obtain information about applications' access patterns through *trace files*.

We have decided to use trace files to obtain information from applications because we wanted to keep AGIOS generic and easy to use. Most methods to obtain such information include changes in I/O libraries, compilers or applications, which would compromise the portability of our tool. The trace file is generated by the scheduler itself and stored on its local storage device, without modifications to the application or to the file system. Chapter 7 will discuss related work on applications' access patterns extraction.

In the trace file, a "new request" entry stores the file identifier, offset, size, and timestamp of the request - the number of nanoseconds elapsed since the current trace's first request arrival, or 0 in the case of the first request. Different traces generated by executing the same application may present some variation between the arrival times of the same request. In order to obtain more realistic arrival time estimations, several trace files can be combined. This process is further discussed in Section 4.2.

Figure 4.1 presents AGIOS' modules. Trace generation is activated by a configuration parameter, and can be used with any of the provided scheduling algorithms. The *Prediction Module* is responsible for obtaining information from traces and providing them to scheduling algorithms.

The Prediction Module is initialized by the scheduler in the beginning of its execution if

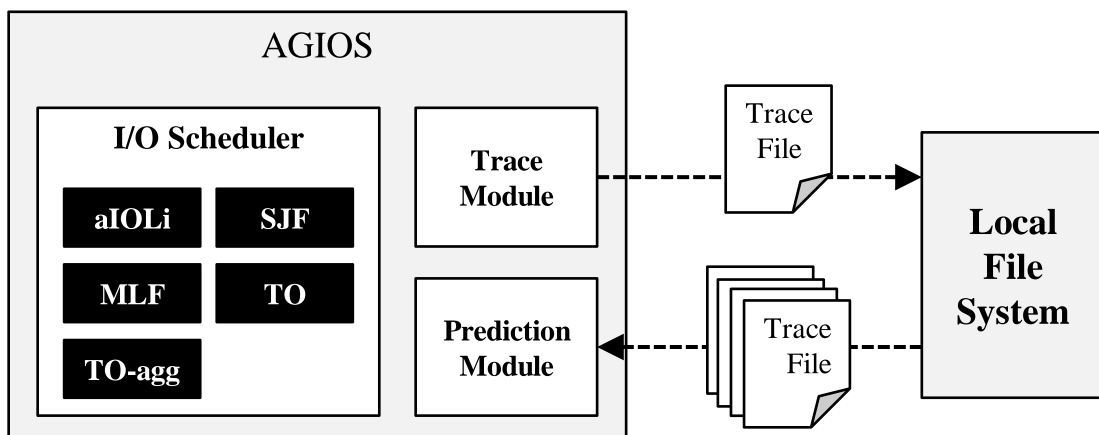


Figure 4.1: AGIOS' modules and trace generation.

trace files are present. A prediction thread then reads the traces and generates a set of queues identical to the ones used during execution, except that these contain "predicted" future requests. This initialization can also be triggered during execution, providing the ability to generate a trace file during some initial period of the execution and then start the Prediction Module if we expect the traced access pattern to happen again in the future.

The set of predicted requests obtained by the Prediction Module from trace files provides a larger window for scheduling algorithms to work. The obtained information can be used to make better decisions, leading to better results. We illustrate this potential by using predictions to improve aggregations on one of the previously presented scheduling algorithms. Section 4.1 details and evaluates this approach.

Section 4.2 presents a further analysis on the use of trace files to obtain information about applications. It discusses the time taken to obtain this information, trace generation overhead and variability between traces of the same application.

In addition to using predicted requests to make scheduling decisions, Chapter 6 will present our approach to use access patterns' aspects and storage device characteristics to decide which scheduling algorithm is the best fit to each scenario. These access patterns' aspects are extracted from trace files, in a process described in Section 4.3. Section 4.4 concludes this chapter.

4.1 Improving aggregations by using information from traces

This section describes our approach to improve aggregations' size and number by foreshadowing applications' requests. We believe that, if the scheduler can predict how future requests will be, it can make better decisions about its aggregations.

We have modified the aIOLi scheduling algorithm to use predictions of aggregations to decide if it should wait before processing each request. Although we present results with this scheduling algorithm, the same approach could be used to improve aggregations with MLF, SJF or TO-agg. The next section describes how aggregations are predicted. How this information is used during the scheduling algorithm's execution is discussed in Section 4.1.2, and results obtained with this approach are presented in Section 4.1.3.

4.1.1 Predicting aggregations

As previously discussed, the Prediction Module reads information from traces and generates a set of future requests - called "predicted requests". After obtaining a list of all future requests, it evaluates all possible aggregations. This analysis is done aiming at *predicting* aggregations that will be possible during execution. This is necessary because, during execution, requests are aggregated whenever possible: if they exist and are contiguous. On the other hand, having contiguous predicted requests does not necessarily mean it will be possible to aggregate them on execution time, because they can arrive at very distant times. Additionally, we want to predict aggregations that are beneficial to performance, since the goal is to guide decisions based on these predictions.

For every request R_i in the trace, we can obtain its *time of arrival* AT_i and its *size* S_i. We also estimate, by benchmarking, a *time to process function* $TTP(x)$ for x being a request size. If contiguous predicted requests R_1 and R_2 are observed, their aggregation is evaluated considering:

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

This means that we should aggregate two contiguous requests when the time to process them separately is big enough (when compared to the time to process them aggregated) to make it worth keeping the request in queue.

This is a conservative approach, since it does not consider that accepting to process the requests separately may have the extra cost of interrupting the spatial locality by having a non-contiguous request from another queue processed between them. Moreover, the time between the two requests is considered as an extra cost of the aggregation. However, this time does not have to be wasted, since it can be spent processing requests from other queues. To make this decision less conservative, we included a factor α that represents the ability to overlap the waiting time with processing other requests.

Therefore, R1 and R2 should be aggregated if

$$TTP(S_1) + TTP(S_2) > (TTP(S_1 + S_2) + (|AT_2 - AT_1|) \times (1 - \alpha)) \tag{4.1}$$

Since α is supposed to represent the ability to overlap waiting time, intuitively it should be given by:

$$\alpha = \frac{Ov}{T}, \;\; with \;\; Ov \leq T \tag{4.2}$$

where T is the total time the scheduler was supposed to wait before processing its requests, and Ov is the time the scheduler was supposed to wait but was able to process other requests instead. Since Ov is part of T and thus $Ov \leq T$, we always have $0 \leq \alpha \leq 1$.

However, at initialization, when the Prediction Module predicts aggregations, values for Ov and T are unknown and can only be estimated. The Prediction Module estimates α by going through all its predicted requests in expected time to arrival order. To each one, it finds its closest contiguous request and updates Ov and T depending on the size of all requests between the two contiguous ones.

Nonetheless, for this method to make sense in a multi-application scenario, it would be necessary to have traces for all applications that will execute concurrently (and only for them). To minimize the effect of an unrealistic estimation of α at initialization, it is possible to update the predicted aggregations periodically by adjusting α according to observed values for Ov and T (observed during the scheduler's execution).

If the scheduler has been able to overlap all of its waiting time with processing other requests - in a situation with a high level of concurrency - we would have $Ov = T$ and thus $\alpha = 1$. Applying this value on the equation 4.1, the Prediction Module would completely disregard the time between requests and decide on aggregations based on the time to process them. Similarly, no concurrency at all would result on no overlapping, $Ov = 0$, $\alpha = 0$, and the time between requests fully counting for aggregations decision.

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

4.1.2 Including Predictions on the Scheduling Algorithm

When an actual request arrives, the scheduler looks for predicted requests to the same file with the same offset, size and with a relative timestamp (relative to the first request to this file) within acceptable bounds. These "acceptable bounds" are defined by an acceptable error parameter, further discussed in Section 4.2. If the scheduler finds such a request, the two versions (predicted and actual) are linked and the predictions concerning this request will be considered during scheduling.

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

When a virtual request is selected to be served, the scheduler asks the Prediction Module if it should be done now, or if it should wait for some period. In order to make this decision, the Prediction Module analyzes the aggregation predicted for the requests' traced versions. If the aggregation is not as big as the predicted one, the scheduler must wait. In order to avoid starvation, the wait will happen only once for each virtual request.

If, in the instant T_K, the Prediction Module decided the scheduler should wait before processing a virtual request composed of requests $R_1, R_2, ..., R_N$, with predicted version composed of traced requests $P_1, P_2, ..., P_M$, the waiting time W will be obtained from:

$$W = Time_diff(P_1, P_2, ..., P_M) - (T_K - Minimum(AT_{R_1}, AT_{R_2}, ..., AT_{R_N})) \tag{4.3}$$

$$\begin{aligned} Time_diff(P_1, P_2, ..., P_M) = & Maximum(AT_{P_1}, AT_{P_2}, ..., AT_{P_M}) \\ & - Minimum(AT_{P_1}, AT_{P_2}, ..., AT_{P_M}) \end{aligned} \tag{4.4}$$

where AT_i is the arrival time of request i. This means that the waiting time will be obtained by subtracting the elapsed time since the first request of the performed aggregation arrived from the difference between the predicted aggregation's first and last requests' arrival times.

This process is illustrated in Fig. 4.2. In the instant t_1, the scheduler needs to decide if it dispatches the three highlighted incoming requests for execution or if it waits for longer. With information from the trace, the scheduler knows that a bigger aggregation is possible for these

requests.

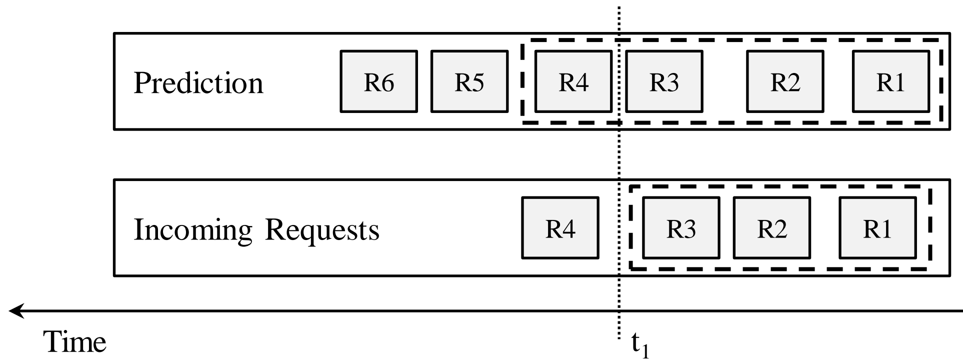


Figure 4.2: Scheduling with predicted future requests.

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

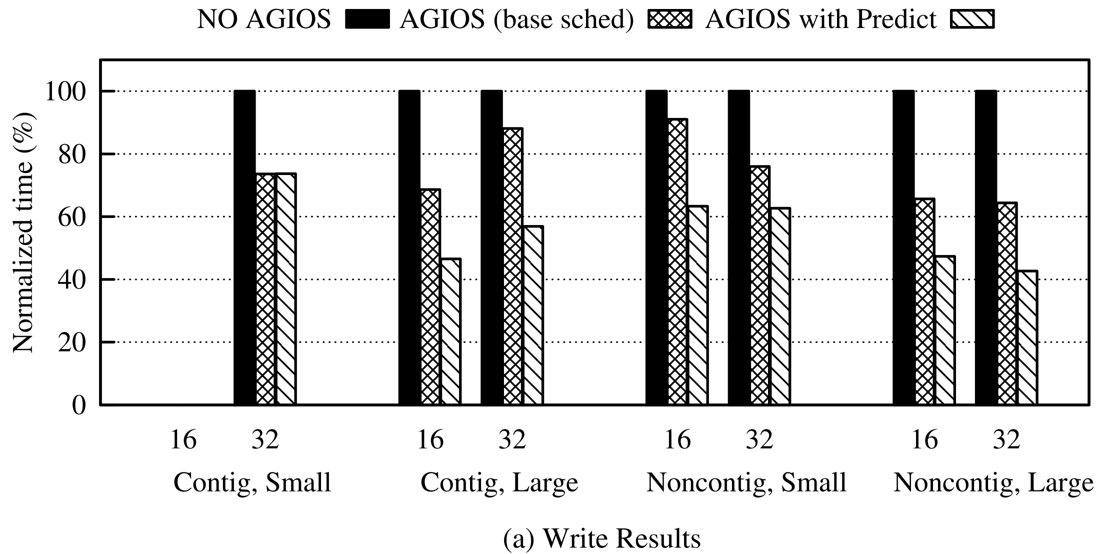
4.1.3 Performance Evaluation

To evaluate the proposed approach, we conducted tests following a similar method to what was presented in Section 3.2. We used $Edel_{old}$ from Grenoble, part of Grid'5000. Edel cluster had its storage devices replaced from HDDs to SSDs in October 2013. All tests in this section were executed before the replacement, with HDDs, and tests from other chapters were more recently obtained, with SSDs. Therefore, we call it "$Edel_{old}$" to explicit this difference.

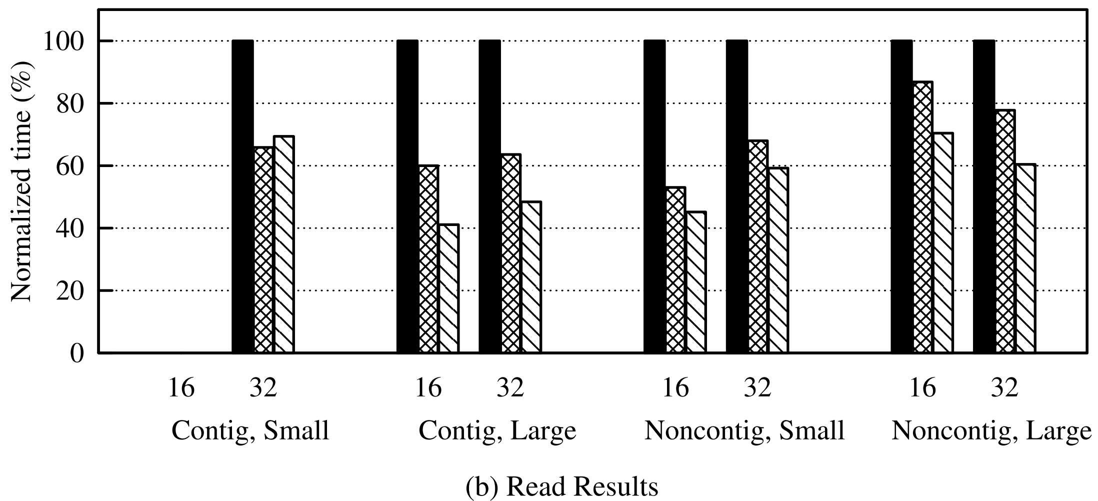
We present results with four dNFSp data servers, four concurrent applications (four instances of the same benchmark) running on 32 client machines (8 nodes per application) with 16 or 32 processes per application. On tests with small requests each process issues 128 requests of size 16KB, and on large requests tests, 8 requests of size 256KB. Each application's processes share a file.

Traces from 6 repetitions were provided to the Prediction Module. The times for generating traces, reading them and making predictions are not included in the results presented in this section. These values will be discussed in Section 4.2.

Figure 4.3 presents the results obtained for AGIOS with the Prediction Module, comparing with results for the base scheduling algorithm and for the file system without AGIOS. Write results are presented in Figure 4.3a, and read ones in Figure 4.3b. All values are normalized by the time without AGIOS.



(a) Write Results



(b) Read Results

Figure 4.3: Results with the AGIOS's Prediction Module.

Table 4.1: Average performance improvements with AGIOS (base scheduler)

	Contiguous		Non-contiguous	
	Small	Large	Small	Large
Write	26.6%	28.4%	24.7%	31.5%
Read	13.4%	40%	37.2%	17.6%

Table 4.1 summarizes the performance improvements obtained by using the base scheduling algorithm (aIOLi) only (without the Prediction Module) over not using AGIOS (dNFSp's original timeorder scheduler). The numbers represent the average performance improvements for each access pattern (the arithmetic average of the improvements obtained for both numbers of processes).

In this scenario, the use of AGIOS's base scheduling algorithm resulted in performance improvements of up to 42.05% - 27.81% on average - for write operations and of up to 46.95% - 27.06% on average - for read operations. No significant decrease in performance was observed

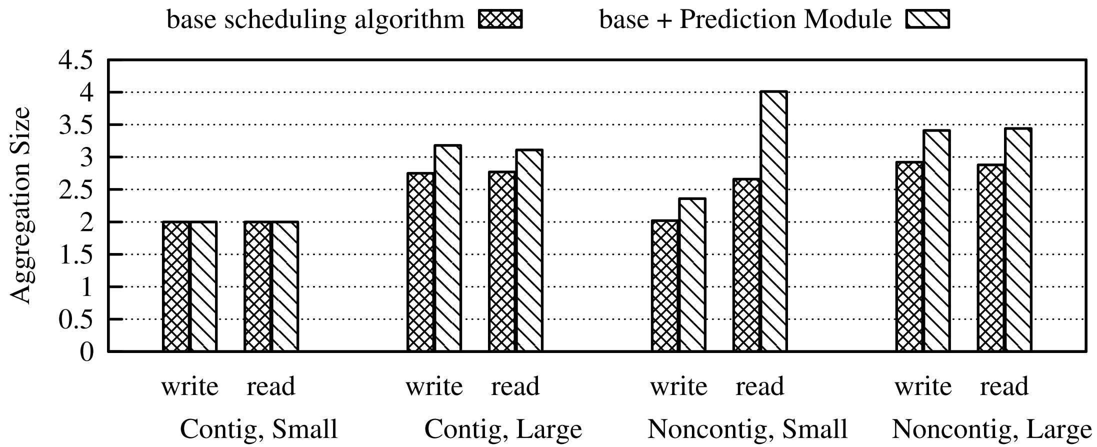


Figure 4.4: Increased average aggregation size with the Prediction Module for AGIOS.

by including the scheduling algorithm to the file system. We can observe that, on average, read and write operations benefited equally from scheduling.

Table 4.2: Average improvements with the Prediction Module.

	Contiguous	Non-contiguous	
	Large	Small	Large
Write	33.8%	24%	30.8%
Read	27.7%	13.9%	20.6%

Table 4.2 presents the average performance improvements obtained by using the Prediction Module over the base scheduling algorithm (AGIOS without Prediction Module). The observed average aggregation sizes are listed in Figure 4.4. Our approach led to aggregations 25.1% bigger on average.

In tests with contiguous requests, each process has a dedicated file portion, so requests coming from one process are not usually contiguous to requests coming from other processes. In these tests, all aggregation opportunities come from requests issued by the same process. As discussed in Section 3.2.2, our tests use synchronous requests, and application's requests are divided in multiple actual requests because of the NFS protocol. In the test with contiguous and small requests, the 16KB application's requests are divided into two 8KB requests to the file system. In this case, the maximum possible aggregation size is 2, since these two requests are contiguous, but not contiguous to the requests of other processes. Also, because the amount of data accessed by each process is multiple of the stripe size and of the number of servers, the two contiguous requests will always be to the same stripe.

As we can see in Fig. 4.4, the base scheduling algorithm is already able to aggregate as much as possible to the "contiguous, small requests" workload. Therefore, only the other 3 access patterns are interesting for the Prediction Module. We included a result with this access

pattern in Figure 4.3 in order to show that although the Prediction Module is not helpful in this case, it does not degrade performance significantly.

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

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

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

When comparing Figures 4.3 and 4.4, we can observe that the biggest increase in aggregation size (for read operations in non-contiguous, small requests - from 2.7 to 4) resulted in the lowest increase in performance by the Prediction Module ($\approx$ 13%). In this case, large aggregations were predicted, inducing more waiting times. As previously said, read operations are more affected by these waiting times, so the increase in performance was not as good as in other tests.

In the results presented in this section, the *total performance improvement* obtained by AGIOS with the Prediction Module (when compared to the scenario without AGIOS) is of up to 57.3% - 42.33% on average - for writes and up to 58.9% - 39.2% on average - for reads.

These results illustrate how information on access patterns can be used to improve scheduling algorithms' decisions. We apply predictions of future requests obtained from trace files to predict aggregations that will be possible during execution. Then these predicted aggregations guide the scheduling algorithm's decisions about processing requests or waiting. The next section details characteristics of this approach such as trace files' size and creation overhead.

4.2 Characteristics and limitations of the trace files approach

In the context of this thesis, where a parallel file system's data servers use AGIOS for I/O scheduling, each test results in the creation of one trace file per data server. The trace file has an entry per request received by its server, and each entry has a size of ≈ 100 bytes.

Table 4.3: Trace files' size and time in minutes required for the Prediction Module to make predictions from them in the Suno cluster.

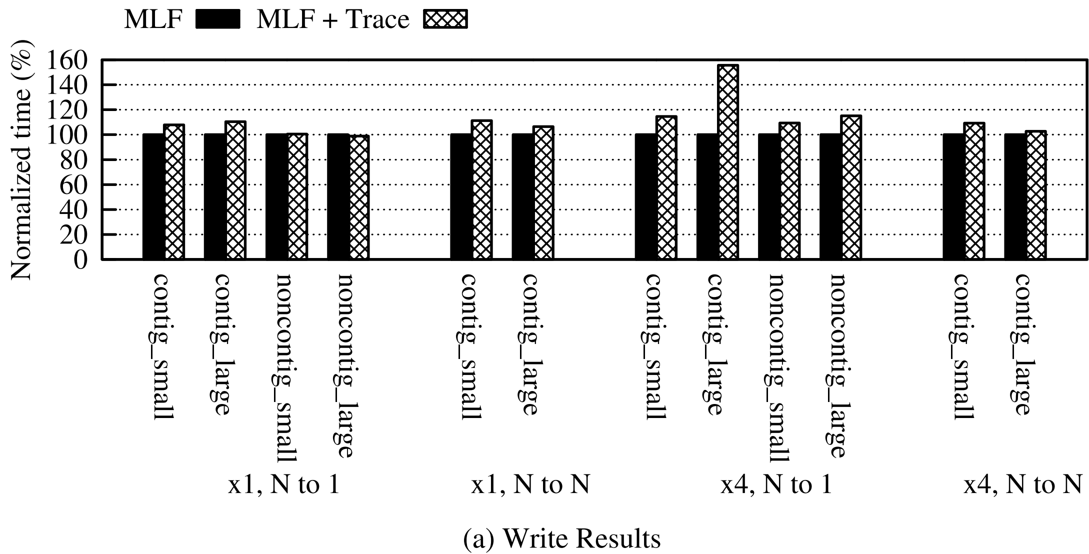
	1 application				4 applications			
	shared file		file per proc		shared file		file per proc	
clients	8	16	8	16	8	16	8	16
traces' size	1.2MB	2.5MB	20MB	40MB	4.8MB	9.8MB	40MB	79MB
time to read 8 traces	2.45	12.11	67.97	190.21	13.85	80.45	-	-

In the experiments presented in Section 4.1, each client process generated traces of around 6.5KB in each server, 26KB per client counting the 4 servers. 6 traces (from 6 repetitions) were provided for the tests with the Prediction Module. For these workloads, the observed overhead caused by trace creation was under 10%, and not statistically significant. The required time to read and combine the 6 trace files and make predictions was also negligible.

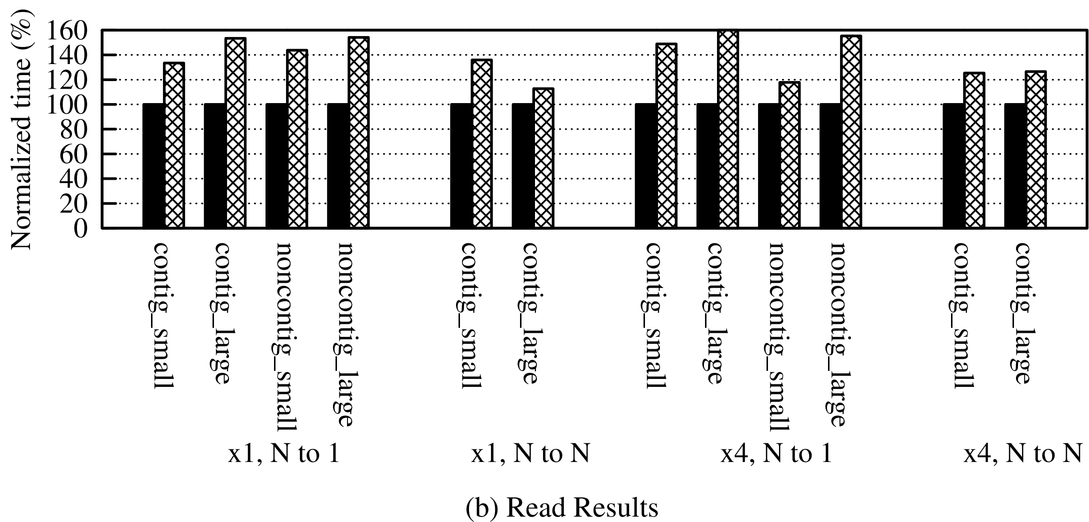
On the other hand, tests that generate a larger number of requests will result on larger traces. Table 4.3 lists the trace files' sizes from the experiments presented in Chapter 3. The values correspond to the size of each of the four servers' files. Doubling the number of clients generated twice the number of requests, hence trace files that have double the size. The file per process approach generates 16 times more requests for tests with a single application, and 8 times more requests for tests with 4 applications, therefore resulting in trace files that are 16 and 8 times larger, respectively. The traces obtained for workloads of 4 concurrent applications correspond to the sum of these applications' traces.

Table 4.3 also presents the time *in minutes* required to read these traces (from 8 repetitions) in the Prediction Module and generate predictions from them. Differently from what was observed with the workloads used for the experiments of Section 4.1, the Prediction Module's setup with larger trace files may take hours. Nonetheless, this step is required only once, while the obtained predictions can be used multiple times. Therefore, this approach makes sense for repeating access patterns.

This thesis focuses on applications that work on timesteps, such as weather forecasting and seismic simulations (BOITO et al., 2011; TESSER et al., 2014). These applications periodically write results to files using the same access pattern. It would be possible, therefore, to trace one or a few timesteps in the beginning of their execution and then use the information to optimize I/O performance for the next timesteps. Applications that perform checkpoints during execution usually have a similar behavior. Another possibility is to apply this method to an application that is executed frequently with the same access pattern. For instance, applications involved in weather forecasting are often executed daily with a similar workload.



(a) Write Results



(b) Read Results

Figure 4.5: Overhead caused by trace generation on AGIOS (MLF scheduling algorithm) with 8 clients in the Pastel cluster.

Larger traces also result in larger trace creation overheads. Constantly writing to the trace file, which is stored on local disk, could compromise the scheduler's results. It would also generate a pattern of several small requests, not ideal for performance. To avoid this situation, AGIOS keeps trace entries on a buffer until this buffer is full or until a function is called to cause its flush to disk. Through this method, we minimize AGIOS' accesses to its local I/O stack and hence its tracing mechanism's interference on performance.

Table 4.4 presents the observed overheads caused by trace generation for the workloads

Table 4.4: Overhead caused by trace creation.

	Pastel	Graphene	Suno	Edel
Write	12.5%	2.1%	19.5%	23.54%
Read	34.64%	81.8%	196.87%	334.29%

from Chapter 3. The overheads were measured comparing the execution times with trace generation enabled with the time previously observed with the same scheduling algorithm without tracing (in this case, we used the MLF algorithm). The values in the table represent the average overhead of all tested access patterns. Figure 4.5 illustrates these behaviors by presenting the overhead observed in the tests with 8 clients in the Pastel cluster. The graphs present normalized execution times separated by access pattern. Single applications are represented by “x1”, and multiple applications by “x4”. “N to 1” represent the shared file approach, and “N to N” the file per process one. Write results are presented in Figure 4.5a, and read ones in Figure 4.5b.

We can see that read workloads were more affected by tracing, presenting a larger overhead. This happens because these tests usually have a smaller execution time than write tests, but write the same amount of data to the trace files. Another possible reason is that each read test was executed right after a write one, without explicitly cleaning the buffer between them. Therefore, trace entries from the write test might have been left on buffer and flushed during the read test.

The larger the buffer, the smaller the overhead expected to be caused by writing to trace files. On the other hand, keeping a large buffer on memory could also affect the performance of the I/O service that is using AGIOS. The overheads listed in Table 4.4 were measured using 12.5% of the total available memory to trace buffering. Further analysis is required to determine the optimal buffer size.

Similar to the discussion about the time required to read traces and make predictions, trace files can be reused several times. Therefore, applications with repeating access patterns can pay the extra cost of trace generation a few times and benefit for a longer time.

4.2.1 Arrival times variability and combination of multiple trace files

Traces generated from repetitions of the same application always present some variation between each request’s arrival times. This happens because many factors affect the arrival time of requests on a server, such as network usage and task scheduling on clients’ machines. Moreover, I/O scheduling also affects requests’ arrival times, since a delay in processing one request may lead to delays in all subsequent synchronous ones.

For this reason, in order to link incoming requests at a server with their traced versions, the definition of acceptable bounds for difference in arrival times is necessary. If this margin is too large, then requests might be wrongly paired with predicted ones, resulting in mispredictions and possibly in performance decrease. On the other hand, if the margin is too low, then requests that are the same might not be paired and the scheduling algorithm would not benefit from

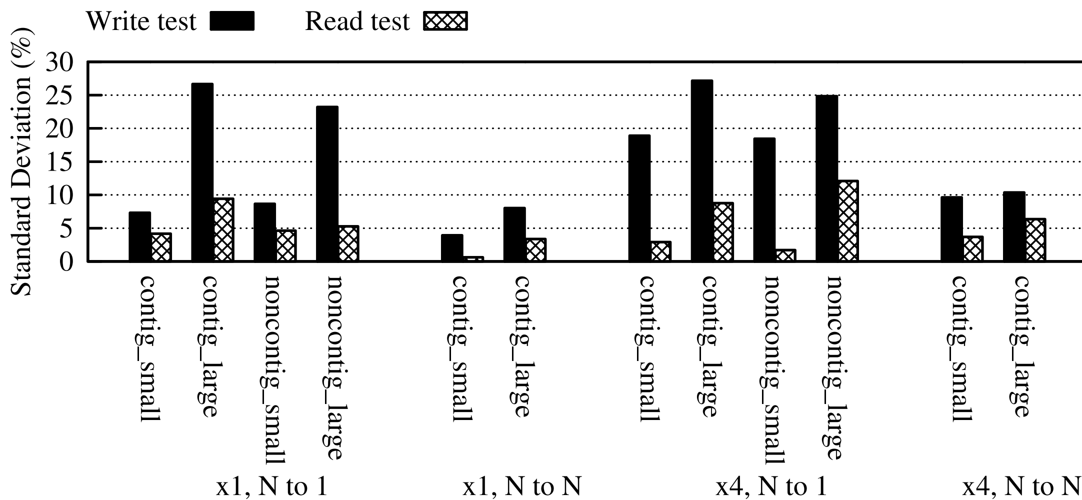


Figure 4.6: Variation between requests' arrival times through 8 traces in the Suno cluster.

predictions.

When combining multiple traces for the same workload, the arithmetic average of the arrival times to each request is taken. This helps providing predicted requests that have representative arrival times, i.e. predicted requests more likely to be correctly paired with real requests during execution. However, when reading and combining multiple traces, the Prediction Module does not know at first which requests are the same and must be combined. They are paired based on some acceptable bounds for arrival times, just like linking of real requests with their predicted versions during execution.

For the experiments presented in Section 4.1, we used a margin of acceptance of 10% for both combination of multiple traces and linking of real requests during execution. This value was chosen because it was observed that the standard deviation between requests' arrival times in the 6 used traces was never over 10%. We also observed that this margin was not too large, since it allowed for correct pairing between requests.

We have observed more variability in some of the traces obtained from the experiments described in Chapter 3. Figure 4.6 represents this variability in the Suno cluster. All obtained traces (for 8 and 16 clients) were analyzed by pairing the 8 repetitions separated by accessed file. The standard deviation of the set of 8 arrival times of each request was taken (in % of the average), and the average of all requests' standard deviations to each tested access pattern was used to obtain the numbers shown in the graph. Suno's results represent all used clusters, since their results are similar.

Read tests presented lower standard deviations for requests' arrival times than write ones. One possible reason for this is that read requests are smaller than write ones, since they do not carry data. Being smaller, they are less susceptible to network factors that cause variation

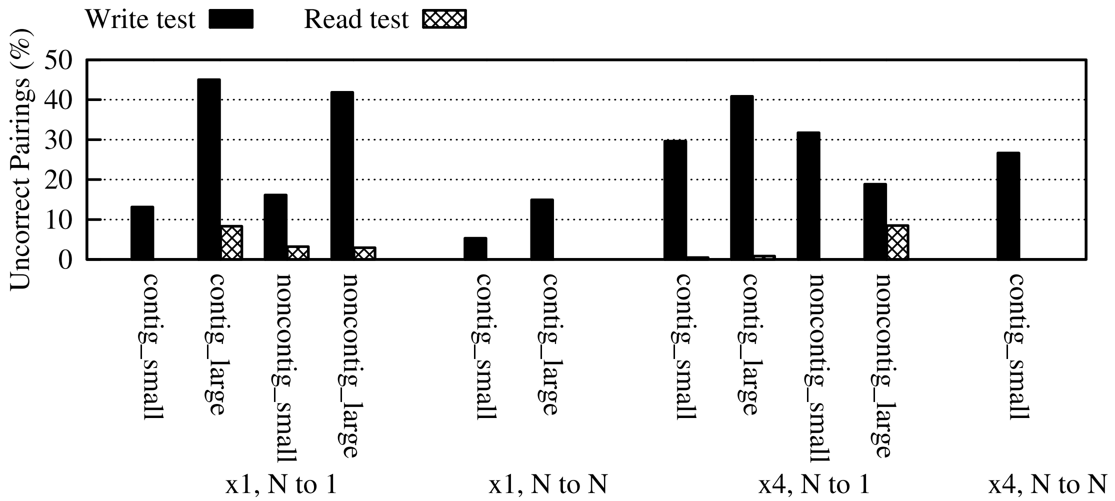


Figure 4.7: Requests that were not correctly paired when combining 8 traces from the Suno cluster with 10% of acceptable bounds for arrival times' variation.

between different executions.

The situations where applications' processes share a file presented highest deviations for requests' arrival times. In these tests, requests scheduling at the servers strongly depends on the order requests from different clients arrive, since they are placed in the same queue. Therefore slight variations in this order may lead to different scheduling decisions, inducing delays in the processing of requests that will affect all subsequent synchronous requests from the same client. This behavior is expected especially when using the MLF scheduling algorithm, since it makes faster decisions than aIOLi, as discussed in Section 3.1. The traces generated from the experiments in Section 4.1 were obtained with aIOLi and presented small variations even in the shared file approach.

High variability between requests' arrival time results in difficulty to correctly link the multiple versions of the same request. For instance, Figure 4.7 presents the number (in $\%$) of incorrectly paired requests from the 8 traces whose variability was illustrated in Figure 4.6. These values were observed when using 10% of margin for arrival times' variation. These incorrectly paired requests will result on a larger number of predicted requests (since the same request will appear multiple times on the queue), increasing the Prediction Module's memory footprint. Moreover, they will affect aggregations' predictions and the scheduling algorithm's use of this information. Further analysis would be required to quantify this impact on performance. An alternative to avoid this problem would be to combine traces before providing them to the Prediction Module. Knowing that they represent the same access patterns would facilitate the correct pairing of requests.

4.2.2 Applications vs. Files

Although we refer to traces *from applications*, they are generated by AGIOS, which works in the context of files. Therefore, the resulting trace files represent how specific files are accessed, and do not carry any kind of application identification. For our approach of obtaining information from traces to make sense, it is important to provide the right traces to the scheduler before executing a workload. In our context of a parallel file system, this means providing the right traces to each server's AGIOS instance.

On several PFS, such as dNFSp, the same file will have a different name on each server. Therefore, the workload will be correctly represented by the set of trace files generated during execution. Otherwise these files could be combined by taking the arrival times' average (as if they were different repetitions of the same test), since their represented access patterns is basically the same. If it becomes difficult to identify which server should receive which trace file, all traces could be provided to all of them. In this situation, each AGIOS instance would expect accesses to more files than it was supposed to. As previously discussed in Section 4.1, this would result in an inadequate α and affect aggregations' predictions. On the other hand, α may be quickly adjusted during execution.

Similarly, trace files obtained from multi-application scenarios could be provided to represent only a subset of the traced applications. This would result in a wrongly calculated α that could be adjusted later. Nonetheless, individual applications' access patterns can only be identified from a multi-application trace if their files are not concurrently accessed by other applications on the trace. If multiple applications access the same files concurrently, then the trace will be specific to their interaction, not representing any of them separately.

We can extrapolate this notion and conclude that the best option is to simply give all Prediction Module's instances all available traces. This would make them capable of improving performance to all traced applications. On the other hand, as previously discussed, making predictions from a large volume of traces might take a long time.

4.3 Classifying access patterns from traces

Chapter 3 presented AGIOS and evaluated its performance with five different scheduling algorithms. Among other aspects, access patterns were classified according to spatial locality - contiguous (non-strided) or non-contiguous (1-D strided) accesses - and size of requests - small (smaller than the stripe size) or large. This section details how the Prediction Module is able

to detect these access patterns' aspects through the information obtained from trace files. In other words, aside from predictions about future requests and possible aggregations, we wanted to make the Prediction Module able to detect spatiality and request size from the traced access pattern.

The difference between the two considered spatialities was previously discussed in Section 3.2.2 (Figure 3.4). Nonetheless, their classification is relative to each application's process *local* access pattern. From the point of view of the server, the behavior is different.

Figure 4.8 illustrates the applications' access patterns from the server point of view. The presented situations have two clients - A and B - concurrently accessing a shared file, striped among four servers with a stripe size of 32KB. Client A issues requests $A_1, A_2, ..., A_n$, and client B generates $B_1, B_2, ..., B_n$. Figure 4.9 presents the file striping among four servers.

Figures 4.8a and 4.8c represent each client's first 8 requests of 16KB each (small). In the contiguous test (Figure 4.8a), each client has a dedicated file portion and thus different clients' requests are not contiguous. Since the stripe size is twice the request size, each client will generate two requests to a server before proceeding to access the next server. The next requests A_9, A_{10}, B_9, and B_{10} are all to Server 0, while A_{11}, A_{12}, B_{11}, and B_{12} are to Server 1.

In the non-contiguous test (Figure 4.8c), requests from different clients are contiguous at the server. Each client will generate one request to each server following a round-robin scheme. Requests A_9 and B_9 will be to Server 0, while A_{10} and B_{10} will be to Server 1.

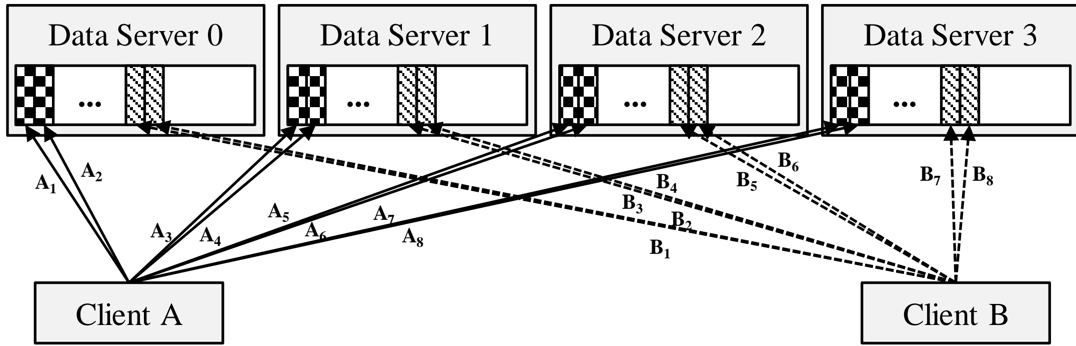
Despite being presented like 16KB units in Figures 4.8a and 4.8c, in our experiments each small request will actually generate 2 requests because of the 8KB file system's transmission size limit. Since our tests perform synchronous I/O operations, each client's requests 1 to n are expected to arrive in order (and not simultaneously), but there is no determined order between requests from A and B.

Figures 4.8b and 4.8d represent tests with large requests. Only each client's first 256KB request is represented. Since this request is larger than the stripe size, it is divided into 8 32KB accesses - $A_{1,1}, A_{1,2}, ..., A_{1,8}$ and similarly with B's requests. The accesses are done circularly among the servers. Moreover, in our tests each one of these 32KB requests will become four 8KB requests because of the transmission size limit. There is no guaranteed arrival order for these requests.

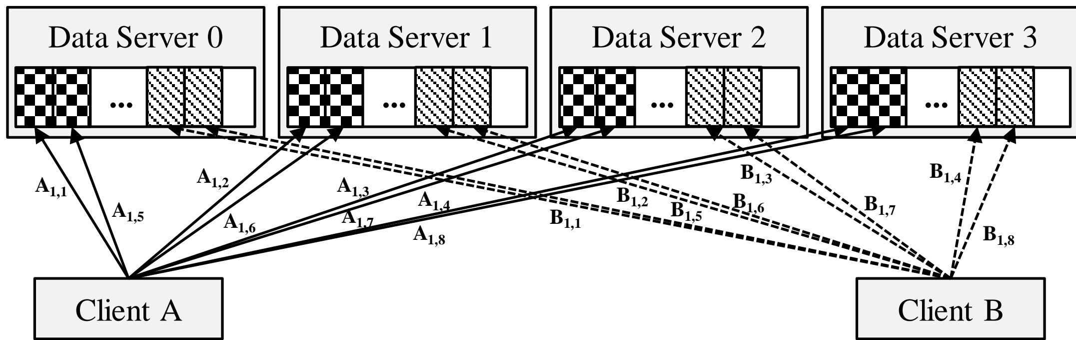
Similarly to what happened in the scenario with small contiguous requests, in the contiguous test with large requests (Figure 4.8b) different clients' requests to each server are not contiguous. For the non-contiguous scenario (Figure 4.8d), on the other hand, requests from different servers are contiguous. However, each client's portion is larger and thus aggregating different clients'

requests might be harder than on the situation with non-contiguous small requests, since it would be less likely to have them simultaneously at the server.

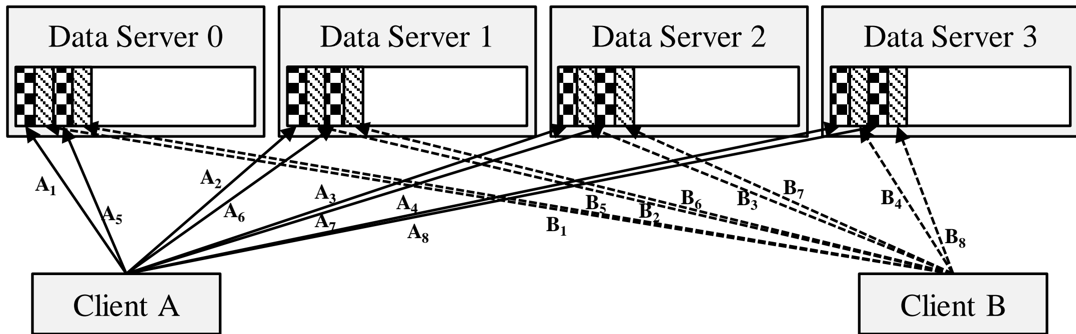
Therefore, contrary to what the access patterns' names suggest, the contiguous (non-strided)



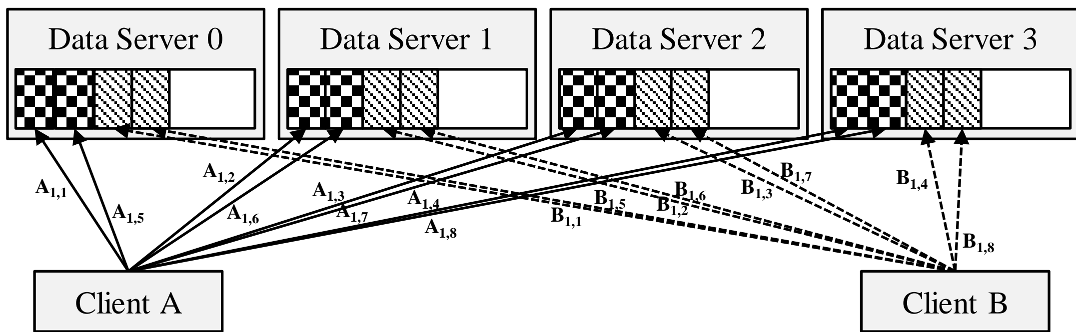
(a) Contiguous, small



(b) Contiguous, large



(c) Non-contiguous, small



(d) Non-contiguous, large

Figure 4.8: Access patterns from the server point of view.

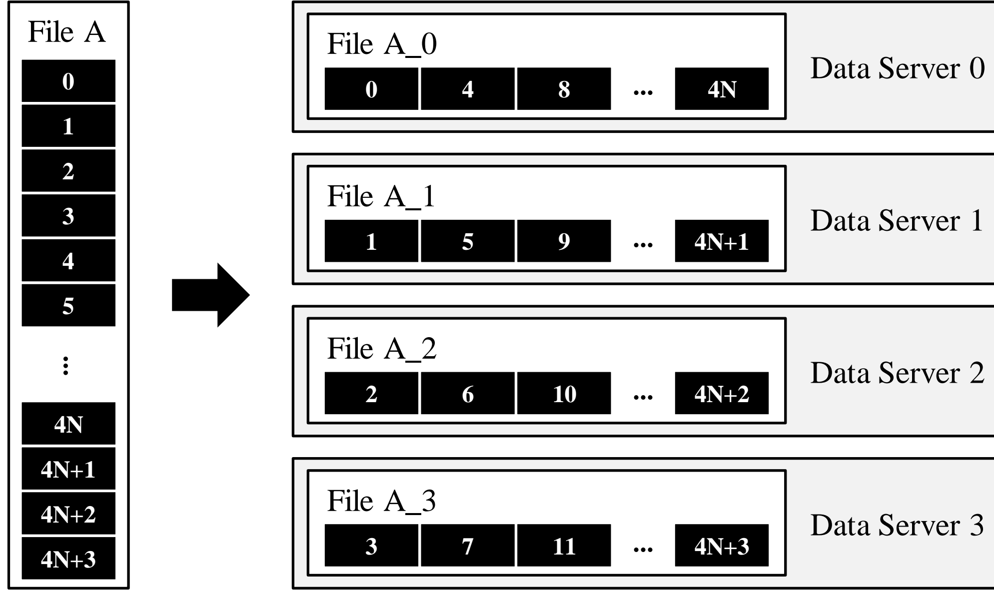


Figure 4.9: Striping of a file among four data servers.

tests generate *non-contiguous* access patterns at the servers, and the non-contiguous (1-D strided) tests generate *contiguous* accesses to the servers. In this thesis, we will continue referring to clients' local access patterns to avoid confusion. If our discussion were to be expanded to local file systems, for instance, then the access patterns' names would simply have to be inverted.

4.3.1 Average distance between consecutive requests

To represent access patterns' spatiality at the servers, we use the *average distance between consecutive requests* $\overline{D}$. This metric is obtained to each file by going through all traced requests to it in arrival order. For each consecutive pair of requests R_i and R_j, we measure the distance between them $D_{i,j}$ as:

$$D_{i,j} = \frac{|Of_j - (Of_i + S_i)|}{S_i} \tag{4.5}$$

where Of_n is the start offset of request R_n and S_n is R_n's size. We take the distance relative to request size to facilitate the comparison between patterns with different request sizes. If R_i and R_j are contiguous (and in offset order), then $Of_i + S_i = Of_j$ and hence $D_{i,j} = 0$.

After obtaining the distances between all pairs of requests $R_1, R_2, ..., R_N$, we calculate $\overline{D}$ to this file by taking the arithmetic average of all pairs' distances:

$$\overline{D} = \frac{D_{1,2} + D_{2,3} + D_{3,4} + ... + D_{(N-1),N}}{(N-1)} \tag{4.6}$$

where N is the number of traced requests.

Sequential access patterns at the server will result in a low average distance. On the other hand, the larger the average distance, the more non-contiguous the access pattern is. Therefore, in our experiments, *it is expected that applications with contiguous access patterns will result on higher average distances between requests than non-contiguous access patterns*. Moreover, this metric might also distinguish between access patterns with small or large requests in the following way:

- for contiguous access patterns (see Figures 4.8a and 4.8b), large application's requests will result on a larger number of actual requests to the server being generated at the same time. Although requests from other clients to their dedicated portions will still increase $\overline{D}$, the larger number of contiguous requests is expected to cause *the contiguous large requests access pattern to present lower average distance than the contiguous small requests one*.
- for non-contiguous access patterns (see Figures 4.8c and 4.8d), small application's requests will provide a more contiguous access pattern at the server. Although the difference is not expected to be very large, *the non-contiguous small requests access pattern is expected to present lower average distance than the non-contiguous large one*.

These relations between different situations' average distances hold for all trace files obtained from experiments described in Chapter 3. Nonetheless, simply knowing these relations between different access patterns' average distances is not enough to build a classifier, since it would need to have samples for all four patterns. In order to build an adequate classifier, we have used Machine Learning (ML) to build a decision tree.

The open-source Weka data mining tool (HALL et al., 2009) was used to obtain a decision tree from an input set composed of all obtained trace files (from all workloads described in Sections 3.2.1 and 3.2.2). All files involved in all traces from the different workloads and platforms are entries on the training set twice - once for the write test and once for the read one. We use results from all platforms in order to make our resulting classifier as platform-independent as possible. Each input set's entry contains the measured average distance and its correct classification: "contiguous, small", "contiguous, large", "non-contiguous, small", and "non-contiguous, large".

To evaluate the resulting decision tree, Weka allows for k-fold cross-validation. This means that the originally provided set is randomly partitioned into k parts of the same size. Then k different trees will be generated, each one of them using $k-1$ parts and leaving the remaining

one out. The left out part is used to evaluate the tree generated with the rest of the entries. The k results are combined to provide a single evaluation. Another possibility is to provide separated sets of entries for tree generation and evaluation.

Table 4.5: Confusion matrix obtained with the J48 algorithm using average distance and accesses' size to classify access patterns.

	contig_small	contig_large	noncontig_small	noncontig_large
contig_small	11493	9515	19	62
contig_large	7304	13789	15	96
noncontig_small	1242	343	443	566
noncontig_large	208	88	118	2223

The first results indicate that the average distance between consecutive requests is not enough to identify the four access patterns. Using 10-fold cross-validation, the decision tree provided by the *J48* algorithm misclassified over 43% of the entries. Including the accesses' sizes as a new attribute improve the results only slightly (the error decreases to $\approx 41\%$). Table 4.5 presents the confusion matrix obtained when using average distance and accesses' sizes to generate the decision tree. In a confusion matrix the cell (a, b) represents the number of entries which belong to class a and were classified as b by the decision tree. We can see that these metrics seem capable of detecting spatiality better than request sizes. Moreover, contiguous tests are easier to identify than non-contiguous ones.

Strided access patterns are harder to identify because the resulting contiguous access pattern at the servers depends on the order requests from different clients arrive. Since our experiments do not use barriers between accesses, it is possible that some processes execute before the others, increasing the observed average distance between requests. Another possible explanation is that we have more entries for the contiguous class, since we did not execute non-contiguous experiments with the file per process approach (and this one generates a large number of entries, since we have one entry per accessed file). 11% of our input set entries represent 1-D strided access patterns.

When using average distance between consecutive requests to classify between contiguous and non-contiguous only, the J48's resulting decision tree has a misclassification rate of $\approx 5.7\%$. Including the accesses' sizes as a new attribute decreases the error to $\approx 4.23\%$. Additionally, by including accesses' sizes as a new attribute the non-contiguous class' precision and recall are increased from 0.9 to 0.96 and from 0.55 to 0.64 respectively.

To a classifier, *precision* represents the ratio between the number of entries correctly assigned to a class and the total number of entries assigned to that class. In other words, precision answers the question "how many of the selected entries were correctly selected?". *Recall* gives

the ratio between the number of entries correctly assigned to a class and the total number of entries that should have been assigned to that class. Recall answers the question "how many of the whole set of entries that should have been selected were selected?".

Although including accesses' sizes increases the classifiers' ability to identify non-contiguous tests, it could also be *overfitting* results to our input set, i.e. providing a classifier that is too adapted to provide good results to our input set, but not necessarily to other situations. This happens because the file per process experiments' files are larger than the shared file's ones, so in our input set a large access size indicates contiguous access patterns. Another reason not to use this attribute is to allow detection on execution time. The scheduler could keep statistics of distance between requests for recent accesses and use it to detect the current access pattern, instead of obtaining this information from traces. Predicting the total access size, however, would depend on traces.

We evaluated trees generated with all algorithms provided by Weka (all the ones that apply to our input set), and the best results were obtained with J48. More details on these algorithms can be obtained in (WITTEN; FRANK, 2005). The provided decision tree is able to identify 0.99 of the contiguous access patterns and 0.55 of the non-contiguous ones (recall), therefore resulting in 5.7% of error (because it misclassifies half of the non-contiguous access patterns, which represent 11% of the input set).

4.3.2 Average stripe access time difference

In order to discern between small and large requests access patterns, we have included another metric: *average stripe access time difference* $\overline{\Delta T}_S$. To obtain it, we separate all requests to a file into its stripes $S_1, S_2, ..., S_M$. To each stripe S_i, which contains requests $R_{i,1}, R_{i,2}, ..., R_{i,K}$, we calculate its stripe access time difference ΔT_i as:

$$\Delta T_i = Maximum(AT_{i,1}, AT_{i,2}, ..., AT_{i,K}) - Minimum(AT_{i,1}, AT_{i,2}, ..., AT_{i,K}) \quad (4.7)$$

where $AT_{i,j}$ represents the arrival time of the request $R_{i,j}$. After obtaining this metric to all stripes inside a file, we take the average between them to obtain $\overline{\Delta T}_S$:

$$\overline{\Delta T}_S = \frac{\Delta T_1 + \Delta T_2 + ... + \Delta T_M}{M} \quad (4.8)$$

Table 4.6: Confusion matrix obtained with the J48 algorithm using average stripe access time difference to classify between small and large requests access patterns.

	small	large
small	16015	7668
large	6816	17025

with M being the number of stripes inside the file. The reasoning behind this metric is based on the fact that large requests are larger than the stripe size. Therefore *large requests access patterns are expected to present a smaller difference between arrival times of requests to each stripe*, since the client generated them at the same time. On the other hand, on small requests tests, requests to each stripe are either from different clients or from the same client, but generated synchronously one after the other.

Similarly to what was done with the average distance between requests, we have used Weka's J48 algorithm to generate a decision tree using $\overline{\Delta T_S}$ to classify between small and large requests access patterns. The resulting decision tree presented a misclassification rate of 30.48%, with recall 0.68 for the "small requests" class and 0.7 for the "large requests" one. The confusion matrix, presented in Table 4.6, illustrate that both classes present the same difficulty to classify.

Table 4.7: Precision and recall observed with the SimpleCart algorithm using average distance between requests and average stripe access time difference to classify between the four considered access patterns.

	Prediction	Recall
contiguous, small	0.77	0.82
contiguous, large	0.79	0.77
non-contiguous, small	0.91	0.63
non-contiguous, large	0.95	0.94

We have combined both metrics to generate a decision tree to classify access patterns into "contiguous, small", "contiguous, large", "non-contiguous, small", and "non-contiguous, large". From the available algorithms in Weka, *SimpleCart* provided the best decision tree, with a misclassification rate of 20.69%. Table 4.7 presents the precision and recall results for this decision tree using 10-fold cross-validation.

Therefore, the detection of spatiality and request size from a trace file is possible though the two simply calculated metrics presented in this section. Chapter 6 will discuss how this information is applied in AGIOS to automatically select the best fit in scheduling algorithm depending on applications' and storage devices' characteristics.

As discussed in Section 4.2, the process of reading, combining trace files and extracting information from them might take a long time. An alternative would be to obtain these metrics

from traces offline and provide only them to the Prediction Module. In this case, the scheduling algorithms would not benefit from predictions about aggregations, but AGIOS would still be able to use the metrics to decide on a scheduling algorithm. Another option is for AGIOS to calculate the metrics during execution (considering recent accesses) and dynamically use them to make decisions. This last option would not depend on trace files to work.

The access patterns' aspects are detected to each accessed file separately. However, the access patterns from multiple files must be combined somehow to represent the current access pattern at the server. For the use described in Chapter 6, we take the access pattern for the majority of the accessed files. This approach helps to mask the impact of incorrect classifications when we have traced accesses to multiple files. However, files are accessed with the same access pattern in all our experiments' scenarios. Further analysis is required to understand how concurrent different access patterns interact and thus how they should be treated.

4.4 Conclusion

The last chapter presented AGIOS, our I/O scheduling library. Results obtained with five scheduling algorithms indicated that the scheduler's provided performance improvements depend on applications' access patterns. Therefore, it would be interesting for the scheduler to know about applications in order to make better decisions and provide better performance. Nonetheless, this information is seldom available at server side since it is lost through the I/O stack.

This chapter discussed the obtainment of information about applications' access patterns from trace files. These traces are generated by the scheduler itself without changes to applications and file systems. AGIOS' *Prediction Module* obtains this information and uses it to make predictions about future requests. These predictions allow schedulers to make decisions based on a larger requests window than originally available.

We presented an approach to predict request aggregations and apply these predictions during execution. Based on them, the scheduler can decide if waiting would benefit performance by allowing larger aggregations. Results show performance improvements with this approach of $\approx 25\%$ on average over the AGIOS version that does not use this information (the base scheduling algorithm). These results demonstrate the potential of using information about applications' access patterns to improve I/O scheduling decisions.

This chapter also presented a detailed analysis of the trace files approach's characteristics and limitations. The overhead induced by trace generation depends on the used buffer size for

tracing requests, and further analysis is required to determine the optimal size for this buffer. Moreover, since trace files' sizes grow with the number of requests, larger workloads generate larger traces. Reading and combining multiple of these traces in the Prediction Module may take a long time. Nonetheless, applications would have to pay these extra costs once and profit from predictions several times. Therefore, our approach suits applications with repeating access patterns such as simulations that work on timesteps (weather forecasting, seismic simulations, etc) and applications that perform periodic checkpoints. Applications that are frequently executed with a similar workload could also benefit from our approach.

Finally, we have presented two metrics - average distance between consecutive requests and average stripe access times difference - that are good indicators of access patterns' spatiality and request sizes. Through a machine learning tool, we were able to build a decision tree that is able to correctly classify access patterns to files into the four classes "contiguous small", "contiguous, large", "non-contiguous small", and "non-contiguous large". Using 10-fold cross-validation, the detection tree presented a right answer rate of $\approx 80\%$. This access pattern detection can be done by measuring these metrics on trace files, or during execution by analyzing recent accesses.

Chapter 6 will discuss how this information is used to automatically decide the best fit in scheduling algorithm to applications and devices. In our goal of providing *double adaptivity*, the next chapter discusses how we classify and obtain information about storage devices.

5 STORAGE DEVICES PROFILING

Chapter 3 presented AGIOS, an I/O scheduling tool, and evaluated its performance with multiple scheduling algorithms on a parallel file system. We conducted experiments on four different platforms, aiming at representing the alternatives for storage devices: HDDs, SSDs and RAID arrays. The obtained results indicate that performance obtained with I/O scheduling is affected by the underlying storage system. Depending on devices' characteristics, the scheduler's overhead may not surpass its positive effects on performance. Therefore, it is important for I/O scheduler to adapt to storage devices' performance behavior.

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

In this scenario, this chapter presents a tool for storage devices profiling named SeRRa[1]. *Our tool reports, for a storage device, the sequential to random throughput ratio for read and write operations with different requests sizes.* Since I/O profiling of storage devices is a time consuming task, SeRRa aims at providing accurate results as fast as possible, facilitating its use. For that, benchmarks are executed only over a subset of all profiled requests sizes, and the remaining values are estimated through linear regressions.

Although several benchmarking tools report access time and throughput on the access to files over different access patterns, we could not find any tool that reports the sequential to random throughput ratio. Other tools also do not estimate results for a large set of parameters from a few measurements, as SeRRa does through linear regressions. These reasons motivated the development of the tool, described in the next section. Chapter 7 further discusses related work.

Information obtained with SeRRa are used by AGIOS to allow the adaptivity of I/O scheduling algorithms to devices' characteristics. Section 5.1 details our tool, and its evaluation is presented in Section 5.2. Finally, Section 5.3 concludes the chapter by summarizing its contributions.

[1]The tool's name comes from "SEquential to Random RAtio", and is Portuguese for either "saw" (the tool) or "chain of mountains".

5.1 SeRRa: A Storage Device Profiling Tool

This section describes SeRRa, a storage device profiling tool. Its development was motivated by the need for a fast way to obtain the sequential to random throughput ratio from devices as HDDs, SSDs and RAID arrays. The main goals of SeRRa's project are:

- Performance: the information must be provided as quickly as possible;
- Accuracy: the provided information must fairly reflect the real behavior of the profiled storage device;
- Generality: the tool must be easy to use and do not require user-provided information about the device.

Table 5.1: Original time in hours to profile the storage devices used in this study.

	One test execution	Total profiling time
Pastel (HDD)	15.22	329.49
Graphene (HDD)	12.26	120.38
Bali (HDD)	10.38	126.10
Chimint (RAID-0)	2.33	69.13
Suno (RAID-0)	4.21	61.32
Taurus (RAID-0)	2.43	57.95
Edel (SSD)	3.09	40.44
Graphite (SSD)	3.45	60.90
Bali (SSD)	2.67	82.49

Keeping both performance and accuracy goals at the same time is a challenging problem because profiling a storage device adequately can take several hours. Table 5.1 presents the time spent to profile the devices used in this study. It includes time required for one execution of the tests and for the complete profiling (that includes several executions in order to provide statistical guarantees). Details about these devices and the executed tests will be presented in Section 5.2. From the times reported in Table 5.1, it is clear that these tests are time consuming, as the fastest profile took over 1.5 days, while the slowest one took almost 14 days.

Ideally, such a complete profiling would be needed only once for each storage device and its stored results could be used for a long time. However, even considering this fact, to dedicate the environment for days can be prohibitive. Moreover, since aspects other than the storage device like local file system, disk scheduler and operating system also affect performance, the reported results are dependent on a system configuration. Changing the operating system version, for instance, could require a new profiling.

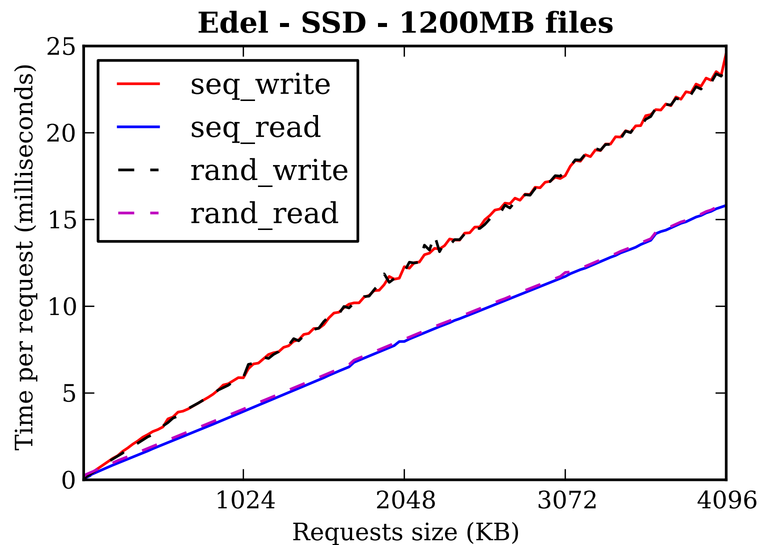


Figure 5.1: Profiling data from a SSD - Access time per request for several request sizes.

The slow profiling whose times are presented in Table 5.1 consists of executing an I/O benchmark several times varying file size, requests size and operation (sequential read, random read, sequential write, and random write). Figure 5.1 shows an example of access times graph - obtained for 1200MB files with different requests sizes on a SSD. Similarly to what can be observed in this example, most of the access times graphs present a linear function appearance. Because of this observation, we have decided to use linear regressions on the design of our profiling tool. Therefore, the following steps compose SeRRa's execution:

1. **Monte Carlo:** request sizes inside a given interval are randomly picked. This interval is provided as a parameter, as well as the gap between every two consecutive sizes. For instance: if asked to approximate the results for the interval [8KB, 40KB] using 8KB gaps, the tool would pick points from the set {8KB, 16KB, 24KB, 32KB, 40KB}. The number of points to be selected for each interval is also defined by the user. Multiple intervals, with different gaps, could be provided. The points at the extremes of the interval (in this example, 8 and 40KB) are always included on this step.
2. **Benchmark:** the test is executed for the requests sizes that were picked on the previous step. We decided to use *IOzone* (NORCOTT; CAPPS, 2006) because it is a widely adopted I/O benchmark, easy to install and use, and because it fits our needs. Other benchmarks that allow sequential and random accesses, like IOR (SHAN; ANTYPAS; SHALF, 2008), could be used with few modifications on the tool. The user also provides the file size to the tests.
3. **Linear Regression:** from the access times measured on the previous step, the complete set of access time results for each given interval is estimated through linear regression.

Each test (read or write, sequential or random) for each interval is estimated separately.

4. **Report:** The sequential to random throughput ratios for the read and write tests are reported. The tool provides such values for all requests sizes, as well as averages, maximum and minimum values. The estimated access times curves are also informed.

The tool, implemented on *Python*, is open source and available at <http://www.inf.ufrgs.br/~fzboito/serra.html>.

5.2 SeRRa's Evaluation

This section presents the SeRRa profiling tool's evaluation. First, we describe the test environments and methodology on Section 5.2.1. Then Section 5.2.2 discusses the error induced by the estimation of access times through linear regression. Lastly, Section 5.2.3 evaluates the tool's results and discusses its trade-off between performance and accuracy.

5.2.1 Tests Environments and Method

This section describes the nine systems used as our test environments, listed in Table 5.2. These systems were selected by their variety, aiming at representing most available devices. Aside from the system named "Bali", all environments are nodes from *Grid'5000* clusters.

The tests were executed on *GNU Linux* operating system, using *IOzone* 3.397. All tested devices were accessed through the *Ext4* local file system, and the default *cfq* disk scheduler was kept. Both virtual memory's page size and file system's block size are 4KB. Caching was explicitly disabled through the *O_DIRECT* POSIX flag (*"-I"* parameter for IOzone).

For the tests described in this paper, five different file sizes were used - 40MB, 200MB, 400MB, 800MB and 1200MB. On all tested devices, we observed that the access time curves usually stabilize before 1200MB, not showing significant differences as we increase the file size further.

The requests size was increased from 8KB to 4MB. This range was divided into two intervals: from 8KB to 64KB with gaps of 8KB; and from 64KB to 4MB with gaps of 32KB. Most parallel file systems employ stripe sizes of at least 32KB for their data servers, therefore the requests they receive (and are object to optimizations) are usually multiples of this value. This happens because each large request issued by an application becomes a set of requests, that are multiples of the stripe size, to servers. We defined a smaller gap on the first interval in order

Table 5.2: Configuration of the evaluated storage devices

Cluster	RAM	Storage Device			
		Type	Model	Capacity	Bus
Pastel	$8GB$	HDD	Hitachi HDS7225SC	$250GB$	SATA $1.5Gb/s$
Graphene	$16GB$	HDD	Hitachi HDS72103	$320GB$	SATA $3Gb/s$
Bali	$64GB$	HDD	Seagate ST91000640NS	$1TB$	SATA $6Gb/s$
Chimint	$16GB$	RAID-0	2×Seagate ST3300657SS-H	$2 \times 136GB$	SAS $3Gb/s$
Suno	$32GB$	RAID 0	2×Seagate ST9300653SS	$2 \times 300GB$	SAS $3Gb/s$
Taurus	$32GB$	RAID-0	2×Seagate ST3300657SS	$2 \times 300GB$	SAS $6Gb/s$
Edel	$24GB$	SSD	Micron C400	$64GB$	SATA $1.5Gb/s$
Graphite	$256GB$	SSD	Intel DC S3500 Series	$300GB$	SATA $6Gb/s$
Bali	$64GB$	SSD	SAMSUNG 840 Series MZ-7TD500BW	$500GB$	SATA $6Gb/s$

to represent small requests, and because some systems limit transmissions from clients to a few KB.

5.2.2 Estimation error by linear approximation

This section aims at quantifying the error induced by estimating the access time curves through linear regressions.

In order to do that, we executed the complete profiling - without estimations, executing the benchmarks for all the specified request sizes - on all tests environments with the parameters discussed on Section 5.2.1. All tests were repeated until a 90% confidence could be achieved with a *t-student* distribution - with at least six executions. The maximum accepted error was of 10%. The necessary time to obtain this complete profile per machine, up to 14 days, was previously presented in Table 5.1.

A modified version of SeRRa was then executed on all devices varying the number of measuring points per interval (number of requests sizes for which the benchmark is executed) from 2 to 8. In the end, instead of reporting the sequential to random throughput ratio, the modified tool compared its estimated curves with the measured ones, obtained through benchmarking

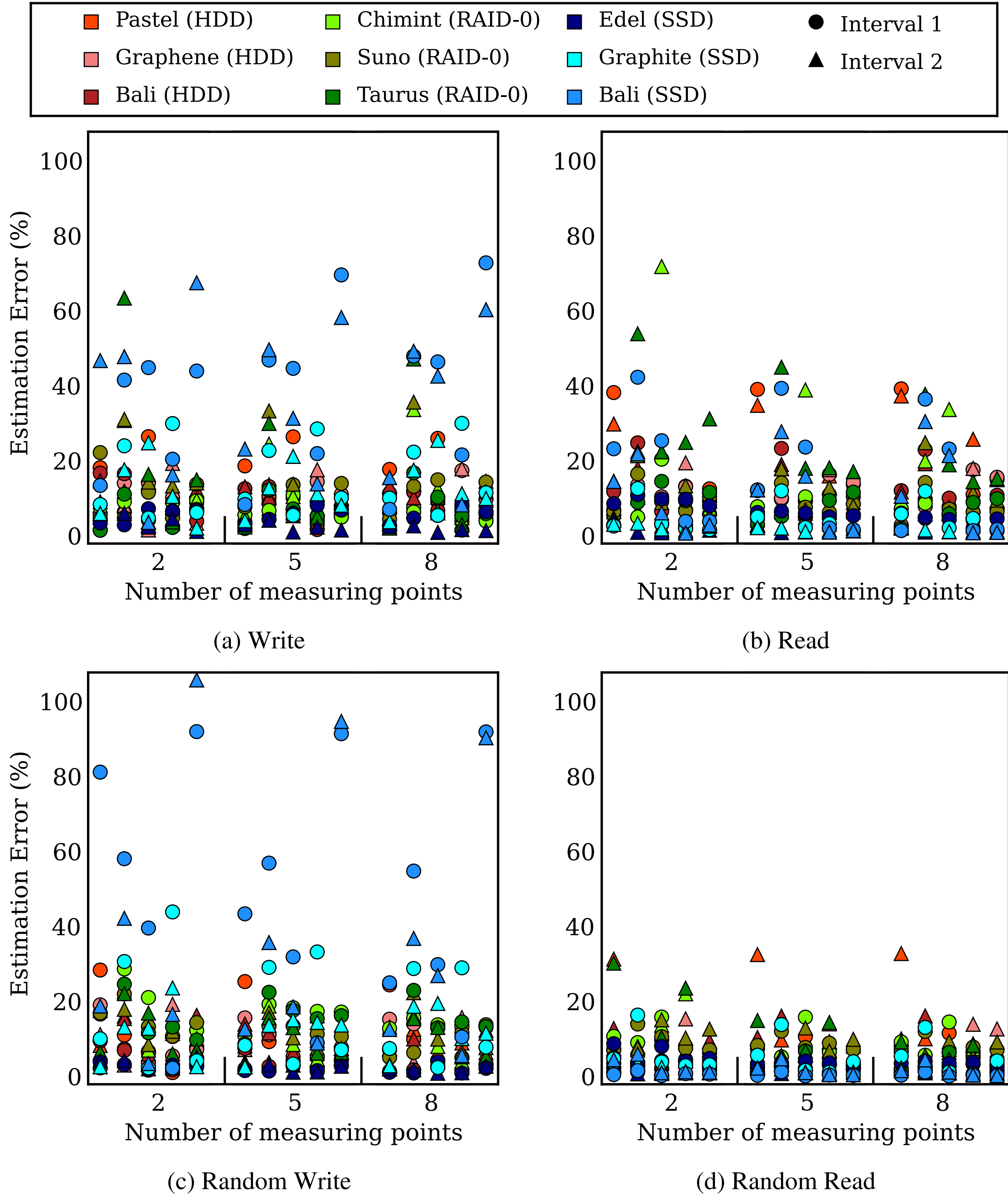


Figure 5.2: Absolute error induced by using linear regressions to estimate the access time curves.

only. The graphs from Figure 5.2 present the absolute values of estimation error for the four tests. For each tested environment (represented by a different color), we present results for both intervals (interval 1 - from 8KB to 64KB - represented by circles, and interval 2 - from 64KB to 4MB - by triangles). For each interval on each number of measuring points, the graphs show five points: one per file size, increasing from left to right. Each result is the median from the errors for all tested requests size. We chose to use the median when summarizing error values

because we believe it is a better central tendency estimator, since it is more resistant to outliers than the arithmetic mean.

We can see that most tests presented errors of up to approximately 20% (84% of the cases for sequential write, 86% for random write, 88% for sequential read, and 98% for random read). In general we did not observe improvements on the quality of the estimation by increasing the number of measuring points. In some cases, it is better to approximate the access time curve with only two points (the extremes of the interval) than including up to all points (Interval 1 has only 8 requests sizes). This happens because the real access time curve is not exactly a linear function, so slight variations from this linear format can impact the estimation, deviating it from the best approximation.

Since the values from the complete profiling are the average of several measurements (in order to provide statistical guarantees), the single benchmark executions made by our tool will hardly match them. We believe this inherent error is mostly responsible for the observed estimation error. This also helps to explain why including more points does not help estimation accuracy: more points potentially include more error.

From the graphs, we can observe that SSD and RAID devices were responsible for most of the cases where the error exceeded 40%. On the complete profiling, these devices took more repetitions in order to obtain statistical guarantees than the HDDs (17 repetitions on average were necessary for each test with HDDs, 43 with RAID arrays, and 26 with SSDs). Therefore, their results present more variability, and are harder to approximate. The machines with more variability were also the ones with incidence of high estimation errors (over 40%): Chimint (62 repetitions), Taurus (47), Bali-SSD (39), and Pastel (25).

Table 5.3: Median from estimation errors.

	Write	Random Write	Read	Random Read
HDD	10%	8%	10%	3%
RAID	10%	16%	9%	7%
SSD	17%	19%	5%	2%
All	9%	8%	7%	4%

Table 5.3 presents the medians of all values from the graphs of Figure 5.2 separated by type of storage device. Results show that RAID arrays' random access time curves are harder to approximate by linear functions than HDDs'. SSDs' write time curves are the hardest to estimate. On SSDs, maintenance operations may be executed by the controller during write operations, affecting performance. This phenomenon, known as "write amplification", causes higher variability between repetitions of tests in SSDs, making the approximation on these devices more difficult.

5.2.3 SeRRa's Results Evaluation

The previous section discussed the error induced by approximating access times curves with linear functions. The presented results show that increasing the number of measuring points (requests sizes to run the benchmarks with) does not necessarily increase the quality of the estimation. Because of that, all ratios obtained with our tool in this section used only 2 measuring points per interval. Instead of increasing execution time by running tests with more requests sizes, in this section we evaluate the number of repetitions of the benchmarks. The arithmetic average of multiple executions' results is taken for each measuring point (instead of the result of a single execution).

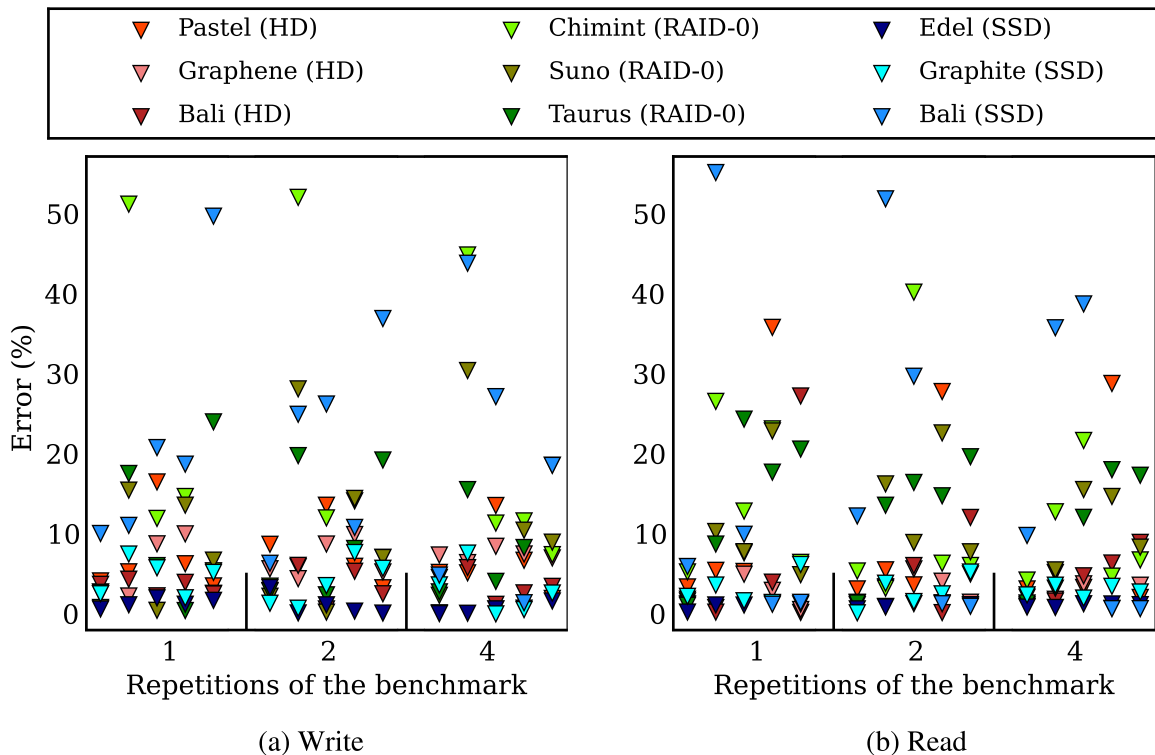


Figure 5.3: Absolute error induced by using SeRRa to obtain the sequential to random throughput ratio.

The graphs from Figure 5.3 present the absolute error obtained by using SeRRa to measure the sequential to random throughput ratio (compared to the ratios obtained from the complete profiling) with 2 points per interval and up to 4 repetitions of each measurement. For each number of repetitions, there are 5 results for each machine: one per file size increasing from left to right, similar to the graphs on Figure 5.2. We can see that most tests presented errors of up to $\approx 20\%$ (91% of the tests for write and 88% for read). The devices responsible for the cases with the highest errors were the same that presented the highest estimation errors on the previous section.

Table 5.4: Median of the errors with SeRRa.

	Write			Read		
	1 rep.	2 rep.	4 rep.	1 rep.	2 rep.	4 rep.
HDD	4.0%	5.7%	5.8%	3.4%	3.6%	3.3%
RAID	6.7%	8.2%	9.0%	10.3%	8.9%	8.4%
SSD	5.2%	3.6%	1.6%	1.5%	1.4%	2.0%
All	5.2%	5.9%	5.2%	4.9%	4.1%	3.6%

Table 5.5: SeRRa's time to measure (minutes). The fractions represent the ratio between these times and the originally required times (without SeRRa)

	1 repetition	4 repetitions
Pastel (HDD)	93.21 (1/212)	376.99 (1/52)
Graphene (HDD)	85.69 (1/84)	341.50 (1/21)
Bali (HDD)	74.03 (1/102)	282.79 (1/26)
Chimint (RAID-0)	22.83 (1/181)	72.73 (1/57)
Suno (RAID-0)	28.12 (1/130)	109.42 (1/33)
Taurus (RAID-0)	16.03 (1/216)	64.54 (1/53)
Edel (SSD)	5.92 (1/409)	23.58 (1/102)
Graphite (SSD)	6.49 (1/562)	24.86 (1/146)
Bali (SSD)	5.79 (1/855)	21.52 (1/229)
Sum	338.10 (1/168)	1317.96 (1/43)

Table 5.4 presents the medians of all values presented on Figure 5.3 separated by type of storage device. In general, increasing the number of repetitions did not increase the results' accuracy significantly. It happened only for read tests on RAID arrays, and for write tests on SSDs. Despite SSDs presenting the highest estimation errors for write tests on the previous section, these devices presented the lowest ratio measurement errors. The time needed by SeRRa to obtain all sequential to random throughput ratios is presented in Table 5.5, along with its comparison to the time needed to obtain these ratios through the complete profiling discussed previously. With SeRRa, the slowest profiling (Pastel) takes 1.6 hours (with 1 repetition) or 6.3 hours (4 repetitions) instead of 14 days in order to provide a value whose error's median is close to 5%.

We believe that more executions of the benchmark did not led to better results because 4 is still a small number of repetitions compared to what was necessary to provide the statistical guarantees defined on the complete profiling (from 13 to 62). In addition, increasing the number of executions can sometimes include outliers, perturbing the results. Since the decrease in time comes from both less repetitions of the tests and less requests sizes to be tested, running SeRRa to obtain this complete profiling with 62 repetitions, for instance, would be expected to take around $1/3$ of the original time and provide results that are more accurate. The choice depends on the needed accuracy. We believe that a median error of approximately 5% (and up to 55%)

Table 5.6: Sequential do random ratio with 1200MB files - measured vs. estimated with SeRRa (4 repetitions).

	Measured		Estimated with SeRRa	
	Write	Read	Write	Read
Pastel (HDD)	2.02	2.24	1.98	2.19
Graphene (HDD)	1.89	2.67	1.75	2.57
Bali (HDD)	1.74	2.61	1.68	2.84
Chimint (RAID-0)	1.95	3.48	1.80	3.25
Suno (RAID-0)	1.71	2.76	1.55	2.53
Taurus (RAID-0)	1.62	3.36	1.32	2.78
Edel (SSD)	0.99	1.08	0.97	1.07
Graphite (SSD)	1.01	1.04	1.04	1.06
Bali (SSD)	1.38	1.10	1.64	1.09

would be enough for comparison between devices and optimizations' evaluation.

Table 5.6 illustrates SeRRa's results by showing the sequential to random throughput ratios obtained with it for $1200MB$ files and 4 repetitions. The values are compared with results from the complete profiling.

5.3 Conclusion

This chapter presented a tool for storage devices profiling regarding access sequentiality named SeRRa. It quantifies the difference between accessing files sequentially and randomly for a given device. In order to obtain this information quickly, the benchmarks are executed on a small subset of the reported values and the remaining ones are estimated through a linear model. Although the tool's main objective is to provide the sequential to random throughput ratio, devices' performance behavior regarding requests size can also be analyzed from its results.

We presented results for a large space of parameters on 9 different storage devices, comprising HDDs, SSDs and RAID arrays. Our evaluation shows that SeRRa provides ratios with median errors of approximately 5% while taking $1/168$ of the originally needed time.

Storage devices present different performance behaviors, and this behavior cannot be simply stated for whole classes of devices. On the other hand, optimizations that work to adapt applications' access patterns impose overheads that may not be compensated by their performance improvements. In this context, the information provided by SeRRa can be used to decide which I/O optimizations will be beneficial for performance for a given storage device.

In the context of this thesis, information provided by SeRRa is used in AGIOS for selecting the best fit in I/O scheduling algorithm for different situations. The performance evaluation

discussed in Chapter 3 evidenced that I/O scheduling results depend on both applications and storage devices' characteristics. Through the tool proposed in this chapter and the contributions from Chapter 4, we are able to provide I/O scheduling with double adaptivity. Chapter 6 will detail this process.

6 I/O SCHEDULING WITH DOUBLE ADAPTIVITY

Chapter 3 presented our I/O scheduling tool named AGIOS. Our tool provides five scheduling algorithms: aIOLi, MLF, SJF, TO, and TO-agg. Through an evaluation that included different access patterns on four clusters, we have observed performance improvements of up to 68% by using I/O scheduling. On the other hand, the same algorithms can decrease performance by up to 278% for some situations.

Our results have evidenced that obtaining the best performance depends on selecting the right scheduling algorithm for each situation, and that the best fit depends on both applications' and storage devices' characteristics. This chapter discusses our approach to provide I/O scheduling with double adaptivity: to applications and devices. In order to make AGIOS capable of selecting the adequate scheduling algorithm automatically, we have decided to use machine learning to generate a decision tree.

The last chapter presented our approach to storage devices profiling with our tool named SeRRa. We use the information provided by it, i.e., the sequential to random throughput ratio for read and write operations with different request sizes, to represent platforms' characteristics.

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

Nonetheless, as evidenced by results presented in Chapter 3, the number of currently accessed files affects the scheduler's overhead. Moreover, the amount of accessed data per file indicates when the algorithms' positive effects on performance are able to surpass their overhead to provide improvements. Therefore, since we use trace files to obtain information about applications' access patterns, we include the number of accessed files and access sizes to represent the access pattern.

Furthermore, as discussed in Chapter 4, from the servers' stream of requests, we are able to classify access patterns regarding spatiality (contiguous or non-contiguous) and request size (small or large). This classification is done to each accessed file using information from traces.

It would be possible to obtain this information during run time using the scheduler's recent accesses, but our approach would still depend on traces if the amount of accessed data per file were required.

It is important to notice that the number of files and amount of data per file attributes were not included to detect spatiality and request sizes, as discussed in Section 4.3, since in that case they were unevenly distributed between the used classes and would result in overfitted classifiers. This does not apply to the discussion from this chapter, since we have tests with different files' sizes and numbers, and these values alone are not enough to decide on the best fit in scheduling algorithms.

In the discussions from Chapter 3, it was observed that the capabilities of platforms' nodes make a difference on I/O scheduling results, since nodes with more RAM memory seem to be less affected by scheduling overhead. For this reason, we also include available memory in our decision making.

The next section will discuss the decision trees obtained with these parameters using the best fits in scheduling algorithms summarized in Table 3.8. Different trees were generated using different subsets of the input parameters. Section 6.2 evaluates these trees. The characteristics and limitations of our approach are discussed in Section 6.3, and Section 6.4 closes this chapter by summarizing its main contributions.

6.1 Scheduling algorithm selection trees

We have used the Weka data mining tool (HALL et al., 2009), that provides multiple machine learning algorithms and an interface to analyze data, apply algorithms and evaluate results. We have provided to Weka an input set that consists of one entry per executed experiment from Chapter 3. Each entry contains:

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

Each pair (access pattern, cluster) generates two entries in the input set, one for each number of processes (8 and 16). The number of processes is not included, since it is not obtainable at server side. However, it affects the number of files (in the file per process approach) or their size (in the shared file approach). The scheduling algorithm decisions from Table 3.8 were made in the context of each access pattern separately, hence both entries appoint the same scheduling algorithm selection.

The sequential to random throughput ratio used for the decision corresponds to the operation and average request size at the server. This request size does not match the applications' one - which we classify in small or large - but the size of requests that arrive to each server, which are a result of striping and the transmission size limit. As previously discussed, in all our experiments, this size is 8KB.

Five different decision trees were generated using different subsets of the parameters listed above. The first tree was generated using all parameters. The complete input set was provided to Weka and all its available algorithms for decision trees generation were tested. Among them, the SimpleCart algorithm provided the best results. Using 10-fold cross-validation, its resulting decision tree has a misclassification rate of 7.81%. This tree, called T_1, has 39 nodes and 20 leaves, and does not use the spatiality and amount of data accessed per file attributes.

In order to see the difference the RAM memory attribute makes in the quality of the generated classifier, we generated a new decision tree using a similar input set, but removing the memory information. The best tree was computed by the J48 algorithm with a misclassification rate of 8.85%. The generated tree has 53 nodes, 27 leaves and all provided attributes appear in the decision making. This indicates that none of them was redundant or unnecessary. We call this tree T_2.

T_3 was obtained with the original input set without the available memory and the amount of data accessed from files. Not having the latter would make our approach less dependent on trace files. The resulting decision tree, computed with J48, uses all attributes and has a misclassification rate of 8.33%, 53 nodes, and 27 leaves.

Going further to also remove the number of files attribute from the input set generates a tree with a misclassification rate of 30.21%, 17 nodes, 9 leaves, and using all provided attributes.This tree, computed with algorithm J48, is called T_4.

In order to illustrate the importance of information about the platform, we generated the last tree - T_5 - without the sequential to random throughput ratio and available memory attributes. J48 generated a decision tree with a misclassification rate of 45.31%, 11 nodes, and 6 leaves. It uses only the attributes for number of files, request size, and operation.

Table 6.1 summarizes the five decision trees' characteristics by showing the attributes used by them. The decision trees' misclassification rates - obtained from a 10-fold cross-validation - are presented in Table 6.2. Since T_5's results are far from desirable, and it was included just to show the importance of including information about platforms, we will no longer discuss this tree in this chapter, focusing the evaluation on the other four.

Table 6.1: Summary of the five decision trees' characteristics - part 1.

	Operation	Number of files	Accessed data per file	Spatiality	Requests size	SeRRa	RAM memory
T_1	✓	✓	✓	✓	✓	✓	✓
T_2	✓	✓	✓	✓	✓	✓	
T_3	✓	✓		✓	✓	✓	
T_4	✓			✓	✓	✓	
T_5	✓	✓	✓	✓	✓		

Table 6.2: Summary of the five decision trees' characteristics - part 2.

	T_1	T_2	T_3	T_4	T_5
Misclassification rate	7.81%	8.85%	8.33%	30.21%	45.32%

6.2 Selection trees' evaluation

This section describes performance results of the four trees described in the last section: T_1, T_2, T_3, and T_4. We compare their results with an "oracle", which always gives the right answer according to Table 3.8. Additionally, we use two other solutions for comparison: one that always selects aIOLi, and another that always selects SJF. We use these two algorithms because they are the ones selected for the largest number of situations.

To evaluate each decision tree, we go through all experiments' situations - combinations of cluster, number of applications, number of files per application, spatiality, request size, number of processes per application, and operation, total of 192 situations - and apply the decision tree to each situation's parameters. Then we take the results previously obtained with the selected algorithm for this situation.

There are two main aspects to consider when evaluating scheduling algorithm selection trees: the situations where they are able to select an algorithm that improves performance, and the situations where the selected algorithm decreases performance. Ideally, a perfect decision tree would improve performance for all cases. However, as evidenced by the results presented in Chapter 3 for some scenarios no scheduling algorithm was able to improve performance. For these scenarios, it is not possible for one of our decision trees to improve performance.

First we evaluate the trees assuming that spatiality and request size are always correctly identified, i.e., without using the access pattern detection tree from the last chapter. These results are presented in the next section. Then, in Section 6.2.2, we evaluate the trees working together, i.e., the results of the access pattern decision tree as input to the scheduling algorithm selection trees.

6.2.1 Results with perfect access pattern detection

Table 6.3 presents the number of correct selections performed by the different solutions - the four decision trees, aIOLi-only, and SJF-only - compared with the oracle's selections. We can see that T_2 and T_3 are able to achieve the best result possible.

Table 6.3: Correct selection rate of all solutions compared with the oracle.

	aIOLi-only	SJF-only	T_1	T_2	T_3	T_4
Correct selections	60 (31%)	92 (48%)	184 (95%)	192 (100%)	192 (100%)	142 (73%)

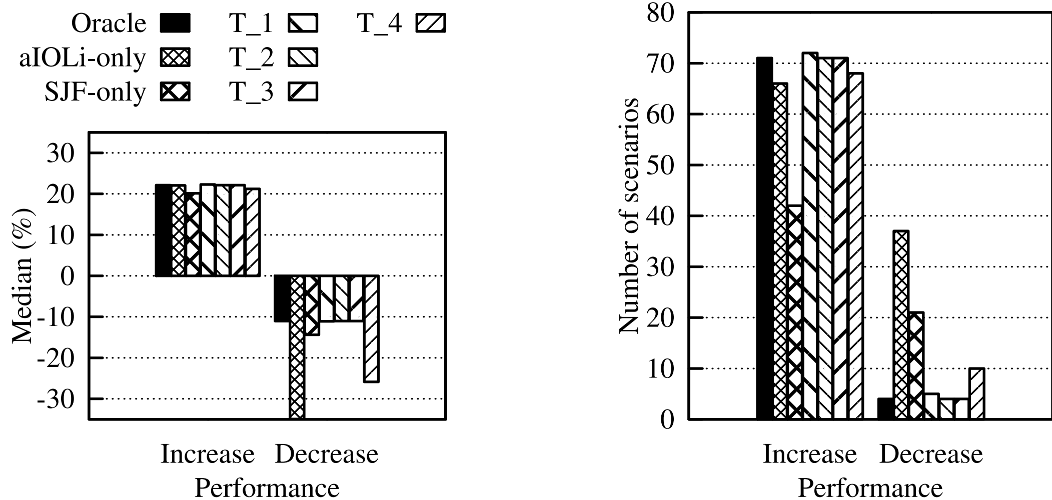
By always giving the right answer according to Table 3.8, the oracle only decreases performance - over not using AGIOS - significantly, i.e., over 10%, in 4 cases (out of 192). From these cases, the worst degradation is 12%. Performance is increased significantly (by over 10%) in 71 cases, ranging from 11% to 59%, 23% on average. The remaining 117 cases where performance was not increased nor decreased represent the situations where no scheduling algorithm was able to improve performance.

It is important to notice that even the oracle decreases performance in some situations because the decisions on the best fit in scheduling algorithm were made to each access pattern with no difference between tests with 8 and 16 processes per application. Therefore, we have selected algorithms that lead to these small performance decreases for some tests because they are good choices in others that were considered together.

The aIOLi-only solution results in significant performance decreases for 37 situations, by 72% on average (up to 278%). Performance is increased significantly for aIOLi-only in 66 cases, by 23% on average (up to 59%), and not affected in the remaining 89 cases.

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

Figure 6.1 summarizes these results and compares them with results obtained for the four scheduling algorithm selection trees. Figure 6.1a presents the median performance increases



(a) Median performance differences (b) Situations with performance differences

Figure 6.1: Performance results for the tested scheduling algorithm selection trees.

and decreases of the 7 solutions. The first group of bars (performance increase) considers only the results with performance *increases* over 10%, while the second group represents results with performance *decreases* over 10%. We can see that using aIOLi-only provided the worst performance decreases.

Moreover, from Figure 6.1b, which presents the number of situations where performance was significantly increased or decreased, we can see that using only one scheduling algorithm improves performance in *less* situations and *decreases* in more.

T_1 improves performance for 1 more situation over the oracle, but decreases performance for 1 more situation. Improving performance for more situations than the oracle is possible because, as previously discussed, the scheduling algorithm decisions were taken considering both numbers of processes per application together. T_2 and T_3 provide the same results than the oracle, as previously indicated. This happens despite the fact that T_3 uses less information than T_1 and T_2, and has a higher misclassification rate than T_2.

T_4 - the tree with the smallest number of input attributes - decreases performance more (by 25% over 11% of the other trees) and for more situations (10 against 4 and 5 of other trees). However, we can still say that our simplest tree - T_4 is better than using aIOLi-only or SJF-only, since T_4 increases performance for more situations.

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

with the input set containing entries with 16 processes only, and vice versa.

Table 6.4 presents the misclassification rates observed in this evaluation step. For T_3 and T_4 the exact same decision tree was generated with both input sets. This happens for T_4 because both input sets are the same, since they do not use attributes that depend on the number of processes per application (number of files and amount of accessed data per client). In this situation, where the same input set is used for generating and evaluating the decision tree, results indicate how well its rules represent the input set. Therefore, we can see that the attributes provided to T_4 are not enough to make a scheduling algorithm selection tree that gives the best answer in more than 75% of times.

Table 6.4: Misclassification rates for each decision tree's two versions, using one to evaluate the other.

	T_1	T_2	T_3	T_4
Tree generated with 8 clients	16.67%	18.75%	5.21%	26.04%
Tree generated with 16 clients	5.21%	5.21%	5.21%	26.04%

For T_3, which uses the number of accessed files, all comparisons with this attribute from both versions are with 1 (the number of accessed files in the shared file approach with a single application) or 4 (shared file with multiple applications), and hence not affected by the number of processes per application. These values are enough to identify the shared file approach in our experiments.

For the T_1 and T_2 versions generated in this evaluation step, we observed lower misclassification rates for the trees generated with the 16 processes per application input set. For T_2, the tree generated with this input set (16 processes) is the same as the T_3 versions. The J48 algorithm decided not to use the amount of accessed data per file attribute in this case, although it was available. Table 6.5 shows this attribute for all our experiments. In the input set with 16 processes, knowing the amount of accessed data per file is only useful to identify the situation with multiple applications where each process has an independent file. Looking again at Table 3.8, we can see that this information has a limited usefulness. On the other hand, the T_2 version generated with entries for 8 processes per application uses the amount of accessed data per file attribute to identify the situation with single application and file per process approach. Using this tree on the 16 processes per application entries, all shared file approach situations are wrongly interpreted, leading to this tree's poor results. A similar problem happens in the T_1 version generated with the 8 processes per application input set.

These results indicate that the attribute that gives the amount of accessed data per file may contribute to generate decision trees that are overfitted to the input set. Furthermore, not including this attribute (in T_3) did not affect the tree's results. On the other hand, not including the

Table 6.5: Sizes of the files generated at the servers.

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

number of files attribute (in T_4) significantly affect results, since spatiality and request size are not enough to represent all tested access patterns.

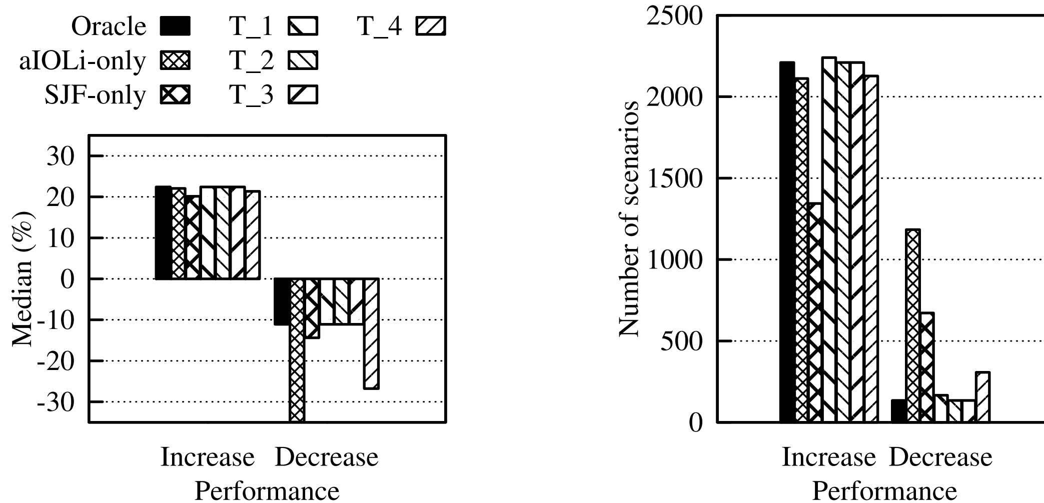
Moreover, providing memory as an attribute for T_1 eliminated the need for the spatiality information. Although spatiality affects aggregations opportunities for the scheduler, this attribute does not play an important role in all results from Chapter 3. For this reason, it was possible to generate a good decision tree to represent these results without this attribute. On the other hand, the available memory attribute serves to identify the machines in this case, since each one has a different value for this parameter. Therefore, using memory could be overfitting our results, providing a decision tree that is too adapted to our experiments' situations, but not capable of providing good results in other cases.

6.2.2 Results with detected access patterns

Spatiality and request size are detected by the decision tree described in Chapter 4. As previously discussed, this detection is done to each accessed file, and the decision for the majority of the files is taken to represent the whole access pattern. To determine the detection error that results from this approach, we applied the access pattern detection tree on all trace files. From each platform and access pattern, there are 32 traces from 8 repetitions over 4 servers. Each trace was evaluated by applying the decision tree to each of its accessed files and taking the decision of the majority. Table 6.6 presents the average and median correct detection rates observed in all platforms. We can see that, in most cases, the approach of taking the decision of the majority improves the access pattern detection - previously the decision tree's evaluation pointed to $\approx 20\%$ of misclassification rate.

Table 6.6: Proportion of correctly detected access patterns (spatiality and request size) in all platforms using the decision tree from Chapter 4.

		Pastel	Graphene	Suno	Edel
Write	Mean	94%	84%	93%	88%
	Median	100%	100%	98%	100%
Read	Mean	80%	87%	76%	86%
	Median	97%	100%	100%	97%



(a) Median performance differences (b) Situations with performance differences

Figure 6.2: Performance results for the tested scheduling algorithm selection trees using a spatiality and request size detection tree.

This section repeats the evaluations conducted in the last section, but using spatiality and request size provided by the access pattern detection tree. This detection' errors will affect the scheduling algorithm choices. To perform this evaluation, the 32 decisions about spatiality and request size obtained to each tested situation (from the 32 traces) were provided to the scheduling algorithm selection trees. Therefore, instead of 192 situations, in this evaluation step we have 6144, since each situation is evaluated 32 times.

In this evaluation step, the oracle was able to improve performance significantly for 2210 situations, decrease performance for 134, and result in negligible differences for 3800. aIOLi-only and SJF-only are not affected by the access pattern detection error, since they always give the same result. Figure 6.2 presents the median performance increases and decreases (in Figure 6.2a) and the number of situations with significant performance differences (in Figure 6.2b) for all evaluated trees.

Both T_2 and T_3 obtained the same results than the oracle solution. Using detected spatiality and request size, they *increased* performance significantly on 2.75% *less* experimented situations, and *decreased* performance on 4.7% *more* situations (compared with the previous evaluation, without the access pattern detection).

The impact was only slightly smaller on T_1, which increased performance on 2.7% less cases, and decreased on 3.75% more. Since T_1 does not consider spatiality for its decision-making, these results suggest that most of the impact comes from wrong request size detection. This happens because, as previously discussed, the spatiality is less important for identifying the best fit in scheduling algorithm for the experiments conducted in this thesis. Furthermore,

as discussed in Section 4.3, correctly detecting request size is harder than detecting spatiality, hence more misclassifications are expected to this parameter.

All median performance increases and decreases have less than 1% of difference with the values obtained in the last section, without using the detected parameters.

Although we evaluated the scheduling algorithm selection trees by comparing them with an oracle solution, the real alternative to them is using only one scheduling algorithm for all situations, without double adaptivity. In this sense, our approach provides performance improvements of up to 75% over aIOLi and of up to 38% over SJF. Moreover, in general, the decision trees are able to improve performance (over the base timeorder scheduler, without using AGIOS) for more situations, and decrease performance for less. Tables 6.7 and 6.8 summarize these results.

Table 6.7: Improvements provided by the scheduling algorithm decision trees over the aIOLi-only solution.

	T_1	T_2	T_3	T_4
Improvement over aIOLi (up to)	74.9%	74.9%	74.9%	74.9%
Situations with performance increase	6% more	5% more	5% more	1% more
Situations with performance decrease	86% less	89% less	89% less	74% less

Table 6.8: Improvements provided by the scheduling algorithm decision trees over the SJF-only solution.

	T_1	T_2	T_3	T_4
Improvement over SJF (up to)	38%	38%	38%	38%
Situations with performance increase	67% more	64% more	64% more	58% more
Situations with performance decrease	75% less	80% less	80% less	54% less

6.3 Characteristics and limitations of this approach

The last section evaluated the proposed scheduling algorithm selection trees. These trees are used by AGIOS' Prediction Module to choose one of the provided scheduling algorithms to obtain good performance. Their decisions are made considering input parameters such as number of accessed files, operation (read or write), spatiality and request size. This information on applications is obtained by the Prediction Module through trace files.

Previously, in Section 4.2, it was stated that a good way of alleviating the imposition of providing the right traces to the Prediction Module is to simply provide all available traces. This would cause wrong predictions at first that could be adjusted during execution. A similar

argument applies to including the number of files concurrently accessed as a parameter for the scheduling algorithm selection tree, since it requires providing *only* the right traces to AGIOS.

Having more files with traced accesses than actually accessed files may result in a wrong scheduling algorithm choice. Nonetheless, it would be possible to re-select the scheduling algorithm periodically according to the current load. The selection has a constant negligible cost, and the transition from one algorithm to other has no cost considering the ones that use the same data structure (MLF, SJF, and aIOLi use a hashtable, while TO and TO-agg use a simple queue). Changing the used data structure would incur in extra cost, depending on the cost of including a new request in the new data structure. Changing to TO-agg or TO requires ordering requests by arrival order, and changing to TO also requires disaggregating virtual requests (a step that could be overlooked since there is usually no harm in aggregating requests, only in spending time trying to do so). Therefore, changing the current scheduling algorithm would have cost:

$$Cost_{changing} = O(N^3) \tag{6.1}$$

where N is the number of requests currently on the scheduler.

This cost would need to be considered when deciding about changing the selected scheduling algorithm. Further analysis is also required to evaluate the periodicity with which this re-selection should be done. This topic will be subject of future work.

All tested scenarios provided homogeneous access patterns: all applications have the same access pattern, all files are accessed in the same way and all tests are either write or read only. In this situation, taking spatiality and request size from the accessed files' majority helps alleviating errors induced by these parameters' detection. However, this will not always be the case in real deployments. To make a decision from a heterogeneous access pattern through our current approach, the scheduler could simply decide based on the majority (for instance, to consider the whole access pattern as a write one, ignoring the reads). Further analysis is required to understand the impact of heterogeneous access patterns on performance.

6.4 Conclusion

This chapter presented our approach to automatically select the best fit in scheduling algorithm depending on applications' and platforms' characteristics. We have used machine learning to obtain decision trees to make this selection. These trees receive parameters such as spatiality

and request size, detected through the access pattern detection tree described in Chapter 4, and the sequential to random throughput ratio, obtained from SeRRa as described in Chapter 5.

We generated five trees providing different attributes. The first tree, T_1 was the only one to receive the machine's RAM memory size as a parameter. T_2 uses all other parameters, T_3 does not include the amount of accessed data per file, and T_4 does not use the amount of accessed data per file nor the number of accessed files. The decisions on scheduling algorithms to all tested situations were obtained from the performance evaluation described in Chapter 3.

The scheduling algorithm selection trees were compared with an oracle solution, which always gives the right answer according to Table 3.8, and with two solutions that always use the same scheduling algorithm, aIOLi or SJF. T_1, T_2, and T_3 obtained similar results, all close to the oracle's results. This indicates that both the amount of available memory and the amount of accessed data per file are not important attributes to identify the experiments' situations, since T_3 obtained good results without them. T_4 had the worst results, but using it would still be significantly better than a solution where the same scheduling algorithm is always applied.

The impact of spatiality and request size detection from trace files was evaluated. For all evaluated decision trees, this impact of this parameters detection error in their results was small: less than 5% of the situations started to have performance decreases. The differences on the solutions' median performance increases and decreases was always smaller than 1%.

Due to its good results, T_3 was included in the Prediction Module to scheduling algorithm selection. The main advantage of not using the amount of accessed data per file as a decision tree's parameter is that our approach is not dependent on the trace files approach. All parameters used to describe applications' characteristics - spatiality, request size, operation and number of files - are obtainable in execution time. Therefore, it would be possible to adapt to applications and devices even when no trace file was available. Moreover, the Prediction Module could periodically re-select the scheduling algorithm based on recent accesses. Exploring this possibility is subject of future work. Future work will also investigate the interaction of heterogeneous access patterns, seeking to expand our approach's good results.

The alternative from our approach would be to simply use the same scheduling algorithm for all situations, without double adaptivity. Our results have shown that our scheduling algorithm selection trees improve performance over the single algorithm solutions by up to 75% (over aIOLi) and 38% (over SJF), improving performance for 5% (aIOLi) and 64% (SJF) *more* situations, and decreasing performance for 89% (aIOLi) and 80% *less* situations. These results strengthen the importance of adapting to applications' and storage devices' characteristics in order to obtain good I/O scheduling results. We have shown that it is possible to automatically

select the best fit in scheduling algorithm for different situations by using these characteristics.

7 RELATED WORK

As discussed in Chapter 2, the performance observed when accessing data from a parallel file system depends on applications' access patterns. Several optimization techniques work to adjust applications' access patterns to achieve better I/O performance. However, as these techniques work in the context of one application, in multi-application scenarios their effects can be impaired by interference caused by concurrent accesses to the file system. I/O scheduling can be applied in this situation to coordinate requests processing.

The previous chapters from this thesis presented our approach to provide I/O scheduling to parallel file systems with double adaptivity. We observed that results obtained with schedulers depend on applications' and storage devices' characteristics in Chapter 3. Chapter 4 presented our approach to obtain information about applications' access patterns, and our approach to storage devices profiling was discussed in Chapter 5. Finally, Chapter 6 discussed how we use this information to select the best fit in scheduling algorithm.

This chapter presents research works that relate to ours. We separate this discussion in three parts: first, Section 7.1 lists related work on I/O scheduling; Section 7.2 discusses techniques to detect and classify applications' access patterns; related work on storage devices profiling are the subject of Section 7.3; finally, Section 7.4 summarizes this chapter.

7.1 Related work on I/O scheduling

I/O scheduling techniques are applied to alleviate interference effects by coordinating requests processing. This coordination can take place on client-side or server-side. However, client-side I/O coordination mechanisms (OHTA; MATSUBA; ISHIKAWA, 2009; DORIER et al., 2012) - discussed in Section 2.4.3 - are still prone to interference caused by concurrent accesses from other nodes to the shared file system. Moreover, techniques to perform I/O scheduling globally on client-side would require synchronization between all the processing nodes, impairing scalability. For these reasons, server-side I/O scheduling is more usual than the client-side approach.

Chen and Majumdar (2001) proposed an algorithm called *Lowest Destination Degree First* (LDDF) that represents processes and servers as nodes of a graph, with edges meaning that a server can treat I/O requests from a process. Servers are then given degrees depending on how many requests they can process, and requests are assigned by following the non-increasing

degree ordered servers list. This LDDF algorithm counts on data replication, so each request has multiple options of servers for processing. Moreover, it assumes a centralized control over all processes and all servers. In a large-scale environment, such a centralized control would impose a bottleneck and compromise scalability.

In their paper, they also evaluated algorithms to be used locally at the servers, after the first assignment done by the LDDF algorithm. The best performance was observed when using *Shortest Job First* (SJF) for local scheduling - over *First Come First Served* (FCFS). In their implementation, jobs' sizes are given by their total amount of requested data. Therefore, authors argue that better scheduling is achieved when *considering information about applications*. Their results motivated us to include SJF in our study. Additionally, another reason to include it was because a similar algorithm - *Shortest Wait Time First* (SWTF) - was reported to present good results as a disk scheduler for SSDs (RAJIMWALE; PRABHAKARAN; DAVIS, 2009). Nonetheless, our SJF implementation - discussed in Section 3.1.3 - uses current size of queues to represent "jobs' length", not including information about applications directly.

7.1.1 Dedicating all servers to an application

Zhang et al. (2010) proposed an approach named IOrchestrator to the PVFS parallel file system[1]. Their idea is to synchronize all data servers to serve only one application during a given period. This decision is made through a model considering the cost of this synchronization and the benefits of this dedicated service. In addition to modifications in the file system, their approach also requires modifications in MPI-IO in order to make it possible for the scheduler to know which files each application accesses.

The same authors adapted their approach to provide QoS support for end users (ZHANG; DAVIS; JIANG, 2011). Through a QoS performance interface, requirements can be defined in terms of execution time (deadline). Applications need a profiling execution, where the proposed mechanism obtains the application's access pattern. This access pattern considers the time portion used for I/O, average requests size, and average distance between requests inside each time slice (called "epoch"). A machine learning technique is used to translate the provided deadline to requirements in bandwidth from the file system, using the profiled access pattern. Their approach is similar to ours in the sense that it uses information from previous executions to detect applications' access patterns. Nonetheless, we use a more detailed access pattern classification,

[1] All related works that used PVFS worked with the file system's second version, "PVFS2". For simplicity sake, this thesis uses the term "PVFS" when talking about this version of the file system.

considering more aspects such as number of files and operation, and also considering storage devices' characteristics as a factor that affects performance. Furthermore, our approach is not deadline-oriented, as our work does not aim at providing QoS.

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

Song et al. (2011) proposed a scheme for I/O scheduling through server coordination that also aims at serving one application at a time. This comes from the observation of implicit and explicit synchronizations on the applications while doing I/O, which cause them often to wait until all the involved requests are finished. For this purpose, they implemented a *window-wide* coordination strategy by modifying PVFS and MPI-IO. Requests from clients carry a global application ID and a timestamp. At the server, they are separated in time windows, where the different windows must be processed in order to avoid starvation. Inside each window, requests are ordered by application ID. They do not use global synchronization, and argue that all servers decide for the same order since they use the same method. We cannot give the same guarantees about our approach, because we do not use global applications identifiers and timestamps. To obtain this information would require modifications in the file system and I/O libraries, compromising portability and making the approach less generic.

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

7.1.2 aIOLi

Lebre et al. (2006) proposed the aIOLi scheduling algorithm, used in this thesis - and discussed in Section 3.1.1. The algorithm was proposed in the context of an I/O scheduling framework (also called "aIOLi"). Their framework aims at being generic, non-invasive and easy to use. The development of our AGIOS tool was vastly inspired by their work. The main

differences between both tools is that aIOLi is a Linux kernel module, while AGIOS also offers an user-level library, since most parallel file systems' servers work at user-level. Moreover, aIOLi was used with a centralized file system (NFS), while our work with AGIOS focuses on parallel file systems. In scheduling algorithm choices, aIOLi offered its aIOLi algorithm and a simple timeorder. Our study included five scheduling algorithms.

Qian et al. (2009) used the same algorithm for the creation of a *Network Request Scheduler* (NRS) for the Lustre parallel file system. Instead of working in a centralized file system, like aIOLi, each instance of NRS works in the context of a Lustre's data server. There is no global coordination of accesses. Their successful use of the aIOLi scheduling algorithm in the context of a parallel file system's data servers motivated the inclusion of this algorithm in our study.

7.1.3 Reactive Scheduling

Ross et al. (2000) proposed a Reactive Scheduling strategy for PVFS's first version. They implemented four scheduling algorithms:

1. a simple timeorder;
2. an algorithm that process requests at offset order (they only consider single file accesses);
3. a window-based algorithm with a given window size (they suggest the size of the machine's physical memory) where all requests inside a window are selected for processing (concurrently);
4. an algorithm that always selects the request closest to the last processed one, similar to *Shortest Seek Time First* (SSTF).

They proposed a model that relates PVFS performance with these scheduling algorithms, and applies this model to select the right scheduling algorithm at run time. The model takes as parameters the number of tasks currently accessing the file, the number of requests currently at the server, the total size of these requests, the number of disjoint regions in the current requests, and the total range of file positions in the requests.

They worked with different scheduling algorithms, and used less details to describe access patterns than our work. Moreover, they did not consider multi-application scenarios, nor situations where applications access multiple files. They make the scheduling algorithm selection through a model, while we applied machine learning to build decision trees. The main disadvantage of using a model is the difficulty to match a model with actually observed results without making it specific to a given file system, platform, and situation. Our approach, on the

other hand, can be applied to multiple situations, and we could use new results to expand it, making it even more generic.

7.2 Related work on access patterns detection

Several techniques to improve performance need information about the applications' access patterns for the decision making process. Prefetching techniques and cache substitution policies (MADHYASTHA; REED, 2002; BYNA et al., 2008; CHEN et al., 2008; SOUNDARARAJAN; MIHAILESCU; AMZA, 2008; LIN et al., 2008; PATRICK et al., 2010) are common examples. This information is obtained in different ways: using machine learning techniques (MADHYASTHA; REED, 2002; ZHANG; BHARGAVA, 2008); by looking for groups of blocks or files that are commonly accessed together (SOUNDARARAJAN; MIHAILESCU; AMZA, 2008; LIN et al., 2008); by information explicitly inserted by in the application code (PATRICK et al., 2010; SEELAM et al., 2010); from a speculative pre-execution of the application (CHEN et al., 2008); or from traces (BYNA et al., 2008).

In this thesis, we used information about applications' access patterns to select the best fit in scheduling algorithm. This information is obtained from trace files, and the spatiality and request size aspects are detected through a decision tree generated with machine learning. This section describes related work on access patterns' detection.

7.2.1 Hidden Markov Models

The work from Madhyastha and Reed (2002) presents and compares two methods for extracting *local access patterns* from applications *at run time*. These two methods are based on learning algorithms and classify access patterns considering three aspects:

1. spatiality: sequential, 1-D strided, 2-D strided, nondecreasing, variably strided;
2. operation: read, write, or both;
3. request size: fixed or variable;

The first proposed method was the use of a *neural network with feedback*. This method was unable to detect different access patterns, so the authors investigated a second approach: the use of *hidden Markov models where each state is a file block*. In this context, an I/O operation takes the current state to a next one with a given probability. These probabilities are obtained

from a training module that runs together with the application. This module considers all issued requests and work to create probabilistic models of irregular patterns that appear repeatedly during the execution. Additionally, if available, data from previous executions can also be used for the training when the file is opened. Spatiality is obtained by multiplying the probabilities of the transitions between consecutive blocks and then comparing with some threshold.

This model needs multiple executions of the application in order to properly train the module. However, as previously discussed, it is usual for scientific applications to be executed several times, with similar behavior between executions. The results from this technique have shown good accuracy and performance improvements when using the obtained information to do prefetching and adapt cache replacement policies. On the other hand, it was also shown that obtaining the information at run time imposes a large overhead.

Their access patterns' classification is too detailed on spatiality (we only consider sequential and 1-D strided) and not detailed enough on requests size (we need to know if they are small or large) to be used in our approach. Moreover, a large overhead in execution time is not acceptable in our context, since it would increase the scheduling overhead.

7.2.2 Post-mortem analysis of trace files

Yin et al. (2012) propose IOSIG, a tool for generating "I/O signatures", which describe applications' access patterns. Their approach had been previously described in the work from Byna et al. (2008), which applies the obtained I/O signatures for a prefetching technique. Their classification considers five aspects:

1. spatiality: contiguous, fixed strided, 2-D strided, negative strided, random strided, k-D strided;
2. requests size: fixed or variable, small, medium or large;
3. temporal intervals: fixed or random;
4. operation: read, write or both;
5. repetition: single occurrence or repeating access pattern.

In order to detect fixed strided patterns, their algorithm uses hidden Markov models, as the work from the previous section. They allow tracing by a modification in MPI-IO. Additionally, the tool allows the analysis of trace files obtained through other methods.

Nonetheless, IOSIG is adequate for obtaining client-side information about processes' local access pattern. We tried to use the tool for detecting patterns for our server-side traces and

it was unable to provide useful results. The main difficulty is that the interaction between different applications and processes' generates a stream of requests - without a defined order between them - that is classified as a random access pattern. We believe it would be possible to use their tool if we provided traces separated by process. However, this information - of which process generated which requests - is not available to use by AGIOS.

7.2.3 Grammar-based detection

Dorier et al. (2014) proposed a grammar-based approach called Omnisc'IO. Their mechanism, integrated into POSIX and MPI-IO to observe I/O calls, is adequate for applications that work on timesteps or perform regular checkpoints. In a few I/O phases, Omnisc'IO is able to build a grammar that predicts future accesses with good accuracy. It does so by tracking requests' size, offsets and inter-arrival times.

Their approach is limited to scenarios where a single file is accessed, and works at the client-side context. Moreover, predictions of future requests do not provide enough information for our approach of selecting scheduling algorithms. It is possible that this client-side approach would present the same difficulties to detect server-side access patterns as the one described in the last section. On the other hand, we believe Omnisc'IO could be used to improve aggregations performed by scheduling algorithms, through the approach described in Section 4.1. Investigating this possibility will be the subject of future work.

7.3 Related work on storage devices profiling

In order to include information on storage devices' characteristics in our approach, we used a metric that quantifies the difference between accessing files at a storage device sequentially or randomly - the sequential to random throughput ratio. We proposed a tool named SeRRa that is able to obtain this metric to different requests sizes efficiently by using linear models.

With the growing adoption of solid state drives, several works focused at characterizing these devices by evaluating their performance over several access patterns (CHEN; KOUFATY; ZHANG, 2009; RAJIMWALE; PRABHAKARAN; DAVIS, 2009). These works point at SSDs' project options, their impact on performance, and illustrate common phenomena as *write amplification* and *stripe alignment*. Differently, we needed an approach to measure storage performance in a generic way, and not only modeling or explaining SSDs' performance.

El Maghraoui et al. (2010) propose a detailed model of SSDs' performance. Nonetheless, the model needs low-level information that must be profiled through micro-benchmarks. Moreover, the proposed model is a low-level model, focusing on the device only and not including higher levels of the I/O subsystem. Desnoyers (2012) used a similar approach. Such models are suitable for evaluating project options for SSDs, for instance, contrary to what our tool does. We aim at providing a high level profiling of the storage system in order to make decisions about optimizations, such as I/O scheduling.

For similar reasons, device simulators like *DiskSim* (BUCY et al., 2008) and others (AGRAWAL et al., 2008; KIM et al., 2009; YOO et al., 2013) have no use on our context. They allow the evaluation of devices parameters, but not the profiling of existing systems.

Although several benchmarking tools report access time and throughput on the access to files over different access patterns, we could not find any tool that reports the sequential to random throughput ratio. Other tools also do not estimate results for a large set of parameters from a few measurements, as SeRRa does through linear regressions. These reasons motivated the development of the tool.

7.4 Conclusion

This chapter discussed research works related to this thesis' three axis: I/O scheduling, detection of applications' access patterns, and storage devices profiling.

Information about applications' access patterns are obtained in our approach by the AGIOS' Prediction Module from trace files. Additionally, machine learning was used to generate a decision tree that detects spatiality and requests size from traces or recent accesses. Although other works use methods such as hidden Markov models and grammars to obtain more details on applications' access patterns, all of them work in the client-side context. At server-side, these techniques provide poor results because of the complex access pattern generated by multiple applications and processes' requests.

Table 7.1 summarizes some characteristics for related work on I/O scheduling, comparing with our approach (AGIOS): the used scheduling algorithm, if modifications are required in the file system and in the I/O library, if using the approach forces a file system configuration with a single meta-data server, and if centralized control is mandatory.

Among these works, our tool provides the larger number of options in scheduling algorithms. Although all listed works require modifications in the file system, aIOLi (the framework) and AGIOS require only minor modifications (passing requests to the scheduler and

Table 7.1: Summary of related work on I/O scheduling - part 1.

	Scheduling algorithm	Modifications: File system	Modifications: I/O library	Single meta-server	Centralized control
IOrchestrator	dedicate all servers to an application	✓	✓	✓	✓
IOrchestrator + QoS	dedicate all servers to an application	✓	✓	✓	✓
(SONG et al., 2011)	dedicate all servers to an application	✓	✓		
aIOLi	aIOLi + timeorder	✓			
Lustre NRS	aIOLi	✓			
Reactive Scheduling	timeorder + 3 algorithms	✓			
AGIOS	**timeorder + 4 algorithms**	✓			

providing a callback), while others include the scheduling algorithm implementation in the file system.

Table 7.2 presents more characteristics about these works: if they use information about applications or storage devices, if they are generic (instead of specific to a file system), and if they adapt to situations.

Table 7.2: Summary of related work on I/O scheduling - part 2.

	Uses information from: Applications	Uses information from: Storage Devices	Generic	Adaptivity
IOrchestrator	✓			Performance Model
IOrchestrator + QoS	✓			Machine learning
(SONG et al., 2011)	✓			
aIOLi			✓	
Lustre NRS				
Reactive Scheduling	✓			Performance Model
AGIOS	✓	✓	✓	**Machine learning**

Despite most related work using information about applications, AGIOS is the one that includes more details on access patterns. None of the other works used machine learning to detect access pattern aspects as we did with the Prediction Module. Moreover, most works are specific to a given file system, while AGIOS is generic and can be used by any I/O service that works at the files level. Our approach is the only one that uses information about storage devices to make scheduling decisions.

To represent storage devices' characteristics, we use the sequential to random throughput ratio metric, obtained by SeRRa. Our tool obtains this metric to different requests sizes and operations efficiently by executing benchmarks to a subset of the parameters, and estimating results for the remaining through linear regressions. To the best of our knowledge, no other profiling tool uses this approach.

The next chapter concludes this thesis by summarizing its main contributions and pointing research perspectives.

8 CONCLUSION AND PERSPECTIVES

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

In HPC architectures, the parallel file system infrastructure is commonly shared by all applications. When multiple applications concurrently access the same file system, their requests can arrive interleaved in the servers. In this situation, we say that applications' accesses *interfered* with each other, in a phenomenon called *interference*. This thesis focuses on I/O scheduling as a tool to improve I/O performance by alleviating interference effects. We focus on the performance aspect of I/O scheduling, not addressing fairness and individual applications' response time.

To improve performance, the I/O scheduler must perform optimizations on applications' access patterns. Nonetheless, the scheduler works on a small window of requests and does not have detailed information about applications, since this information is lost through the I/O stack. At the same time, optimizations' results depend on characteristics of the underlying I/O system, like storage devices' performance behavior. In this context, this thesis main objective is to provide I/O scheduling with *double adaptivity*: to applications' access patterns and to storage devices' characteristics.

We developed AGIOS, an I/O scheduling library that can be used by I/O services to manage incoming I/O requests at files level. AGIOS aims at being generic, non-invasive, and easy to use. Moreover, it offers five scheduling algorithm options, covering multiple advantages and drawbacks: aIOLi, MLF, SJF, TO, and TO-agg. We have demonstrated AGIOS' use with an NFS-based parallel file system's data servers and extensively evaluated performance of its scheduling algorithms on four clusters from Grid'5000, representing storage device alternatives: HDDs, SSDs and RAID arrays. Results indicate that both applications' access patterns and storage devices affect I/O scheduling efficacy.

It is important that applications generate enough data and optimizations opportunities to

surpass the scheduler's overhead. In this sense, aggregating requests plays a central role in improving performance by hiding scheduler's costs. Moreover, our results evidenced that there is no scheduling algorithm able to improve performance for all situations, and the best fit depends on applications' and storage devices' characteristics.

In order to obtain information about applications, AGIOS' Prediction Module applies trace files of previous executions. These traces are generated by AGIOS itself, without modifications to the file system or applications. We illustrate the usefulness of information obtained from traces by applying it to predict request aggregations. We have modified a scheduling algorithm to use such predictions to guide its decisions about waiting before processing requests. This approach led to performance improvements of 27% on average over the previous version of this algorithm (which does not use information on applications). This illustrates the potential of using information about access patterns to improve scheduling decisions.

Obtaining information from trace files incurs on extra costs for generating these traces, reading them, and making predictions. Our analysis has shown that these costs are non-negligible, especially for applications that access large amounts of data. Therefore, our approach is adequate for applications that present a repeating I/O behavior, such as simulations that work on timesteps (weather forecasting, seismic simulations, etc), or applications that perform periodic checkpoints. Applications that are frequently executed with a similar workload could also benefit from our approach. These applications can pay the extra costs once and benefit from the generated information multiple times.

To detect access pattern aspects from trace files, we have presented two metrics - average distance between consecutive requests and average stripe access times difference - that are good indicators of access patterns' spatiality and requests' sizes. Through a machine learning tool, we were able to build a decision tree that is able to correctly classify access patterns to files into the four classes "contiguous small", "contiguous, large", "non-contiguous small", and "non-contiguous large". Using 10-fold cross-validation, the detection tree presented a right answer rate of $\approx 80\%$. This access pattern detection can be done by measuring these metrics on trace files, or during execution by analyzing recent accesses.

Aiming at profiling storage devices, we have developed a tool named *SeRRa.* This tool reports, for a storage device, the sequential to random throughput ratio for read and write operations with different requests sizes. Since I/O profiling of storage devices is a time consuming task, SeRRa uses models to provide accurate results as fast as possible. We have shown that, through this approach, it is possible to profile storage devices in a fraction of the originally required time, and with errors as little as 5%.

We have used machine learning to build decision trees to select the best fit in scheduling algorithm for different situations. Multiple decision trees were generated using different input parameters, including information on applications' and storage devices' characteristics. All generated decision trees provided better overall results than approaches where the same scheduling algorithm is always used. Results close to the oracle solution were obtained by a decision tree that uses the following aspects to make a decision:

- operation: read or write;
- number of files concurrently accessed;
- spatiality: contiguous or non-contiguous;
- requests size: small or large;
- sequential to random throughput ratio of the storage device.

One advantage of this decision tree is that all included access patterns' aspects could be obtained during execution time, considering a window of recent accesses. This would allow our tool to make good decisions even when trace files are not available. Comparing with solutions where only aIOLi or SJF are used, our scheduling algorithm selection tree improves performance by up to 75% and 38%, respectively. Moreover, from all tested situations, our approach is able to improve performance for up to 64% *more* situations and to decrease performance for up to 89% *less* situations. Our results indicate that both applications' and platforms' characteristics are essential for correctly selecting the best I/O scheduling algorithm in a given situation, strengthening the importance of providing I/O schedulers with *double adaptivity: to applications and devices*.

8.1 Publications

The work described in this document resulted in three scientific papers. The first one presents the AGIOS library, its Prediction Module and results obtained by including predicted aggregations in the aIOLi scheduling algorithm. The second one presents SeRRa and the analysis from Chapter 5. The third article presents the scheduling algorithm selection trees approach.

- "AGIOS: Application-Guided I/O Scheduling for Parallel File Systems". In Parallel and Distributed Systems (ICPADS), 2013 International Conference on. IEEE.
- "Towards Fast Profiling of Storage Devices Regarding Access Sequentiality". Accepted for publication in ACM SAC 2015: Symposium on Applied Computing.

- “Automatic I/O Scheduling Algorithm Selection for Parallel File Systems”. Under review for a journal.

8.2 Research perspectives

The research presented in this thesis can progress in several directions, some of which are listed below:

- Considering aspects such as fairness and response time. This thesis focused on performance, but its discussion could be expanded to treat fairness and response time by using other metric instead of the makespan to evaluate algorithms.
- In this thesis, we focused our analysis on homogeneous access patterns. Interaction between different access patterns should also be investigated. Two possibles approaches would be to either look for a solution that benefits all concurrent access patterns, or prioritize some of them over the others.
- Another possibility would be to investigate the approach of having the scheduler adapting to applications during their execution, without prior information. A window of requests would need to be defined, from where information on the access pattern would be detected and used to decide the best fit in scheduling algorithm. It would be interesting for this decision to also consider the cost of changing the current scheduling algorithm. Moreover, further analysis would be required to define the right periodicity for this re-selection.

REFERENCES

AFRATI, F. N.; ULLMAN, J. D. Optimizing joins in a map-reduce environment. In: INTERNATIONAL CONFERENCE ON EXTENDING DATABASE TECHNOLOGY, 13th., 2010, New York, NY, USA. **Proceedings**... New York, NY, USA: ACM, 2010. p. 99–110.

AGRAWAL, N. et al. Design tradeoffs for ssd performance. In: USENIX ANNUAL TECHNICAL CONFERENCE, 2008, Berkeley, CA, USA. **Proceedings**... Berkeley, CA, USA: USENIX Association, 2008. p. 57–70.

ALI, N. et al. Scalable i/o forwarding framework for high-performance computing systems. In: IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING, 2009, New Orleans, LA, USA. **Proceedings**... New Orleans, LA, USA: IEEE, 2009. p. 1–10.

AVILA, R. B. et al. Performance evaluation of a prototype distributed nfs server. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 16th., 2004, Foz do Iguaçu, PR, Brazil. **Proceedings**... Foz do Iguaçu, PR, Brazil: [s.n.], 2004. p. 100–105.

BOITO, F. et al. I/o performance of a large atmospheric model using pvfs. In: RENCONTRES FRANCOPHONES DU PARALLÉLISME (RENPAR'20), 2011, Saint-Malo, France. **Proceedings**... Saint-Malo, France: [s.n.], 2011.

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, South Korea. **Proceedings**... Seoul, South Korea: IEEE, 2013. p. 43–50.

BOITO, F. Z. et al. Towards fast profiling of storage devices regarding access sequentiality. In: SYMPOSIUM ON APPLIED COMPUTING (SAC), 2015, Salamanca, Spain. **Proceedings**... Salamanca, Spain: ACM, 2015.

BOITO, F. Z.; KASSICK, R. V.; NAVAUX, P. O. A. The impact of applications' i/o strategies on the performance of the lustre parallel file system. **International Journal of High Performance Systems Architecture**, [S.l.], v. 3, n. 2, p. 122–136, 2011.

BOLZE, R. et al. Grid5000: A large scale and highly reconfigurable experimental grid testbed. **International Journal of High Performance Computing Applications**, [S.l.], v. 20, n. 4, p. 481–494, 2006.

BRAAM, P. J.; ZAHIR, R. **Lustre: A Scalable, High-Performance File System**, [S.l.]: Cluster File Systems Inc., 2002. White paper. Available http://www.lustre.org/docs/whitepaper.pdf. Accessed March 2010.

BUCY, J. S. et al. **The disksim simulation environment reference manual**, [S.l.]: Parallel Data Laboratory, 2008. Available at http://www.pdl.cmu.edu/PDL-FTP/DriveChar/CMU-PDL-08-101.pdf. Accessed in January 2015.

BYNA, S. et al. Parallel i/o prefetching using mpi file caching and i/o signatures. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2008, Austin, TX, USA. **Proceedings**... Austin, TX, USA: ACM/IEEE, 2008. p. 1–12.

CARNS, P. et al. Small-file access in parallel file systems. In: IEEE INTERNATIONAL SYMPOSIUM ON IN PARALLEL & DISTRIBUTED PROCESSING (IPDPS), 2009, Rome, Italy. **Proceedings**... Rome, Italy: IEEE, 2009. p. 1–11.

CHEN, F.; KOUFATY, D. A.; ZHANG, X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In: INTERNATIONAL JOINT CONFERENCE ON MEASUREMENT AND MODELING OF COMPUTER SYSTEMS, 11th., 2009, New York, NY, USA. **Proceedings**... New York, NY, USA: ACM, 2009. p. 181–192.

CHEN, Y. et al. Hiding i/o latency with pre-execution prefetching for parallel applications. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2008, Austin, TX, USA. **Proceedings**... Austin, TX, USA: ACM/IEEE, 2008. p. 40–51.

CHUNG, T.-S. et al. A survey of flash translation layer. **Journal of Systems Architecture**, [S.l.], v. 55, n. 5, p. 332–343, 2009.

CORBETT, P. et al. Overview of the mpi-io parallel i/o interface. In: JAIN, R.; WERTH, J.; BROWNE, J. C. (Ed.). **Input/Output in Parallel and Distributed Computer Systems**. 1 ed. [S.l.]: Springer, 1996.

CORBETT, P. F.; FEITELSON, D. G. The vesta parallel file system. **ACM Transactions on Computer Systems**, New York, NY, USA, v. 14, n. 3, p. 225–264, 1996.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. **Distributed systems: concepts and design**. 4. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., 2005.

DONGARRA, J. et al. The international exascale software project roadmap. **International Journal of High Performance Computing Applications**, [S.l.], v. 25, n. 1, p. 3, 2011.

DORIER, M. et al. Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free i/o. In: IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING, 2012, Beijing, China. **Proceedings**... Beijing, China: IEEE, 2012. p. 155–163.

FRINGS, W.; WOLF, F.; PETKOV, V. Scalable massively parallel i/o to task-local files. In: CONFERENCE ON HIGH PERFORMANCE COMPUTING NETWORKING, STORAGE AND ANALYSIS, 2009, New York, NY, USA. **Proceedings**... New York, NY, USA: ACM, 2009. p. 17–28.

FRYXELL, B. et al. Flash: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. **The Astrophysical Journal Supplement Series**, [S.l.], v. 131, p. 273, 2000.

GHEMAWAT, S.; GOBIOFF, H.; LEUNG, S. T. The google file system. **ACM SIGOPS Operating Systems Review**, [S.l.], v. 37, n. 5, p. 43, 2003.

HALL, M. et al. The weka data mining software: an update. **ACM SIGKDD explorations newsletter**, [S.l.], v. 11, n. 1, p. 10–18, 2009.

JACOB, B.; NG, S.; WANG, D. **Memory systems: cache, DRAM, disk**. 1. ed. [S.l.]: Morgan Kaufmann, 2010.

KASSICK, R.; BOITO, F. Z.; NAVAUX, P. O. A. Dynamic i/o reconfiguration for a nfs-based parallel file system. In: INTERNATIONAL EUROMICRO CONFERENCE ON PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING, 19th., 2011, Ayia Napa, Cyprus. **Proceedings**... Ayia Napa, Cyprus: IEEE, 2011. p. 11–18.

KIM, J. et al. Parameter-aware i/o management for solid state disks (ssds). **IEEE Transactions on Computers**, [S.l.], v. 61, n. 5, p. 636–649, 2012.

KIM, Y. et al. Flashsim: A simulator for nand flash-based solid-state drives. In: FIRST INTERNATIONAL CONFERENCE ON ADVANCES IN SYSTEM SIMULATION (SIMUL), 2009, Washington, DC, USA. **Proceedings**... Washington, DC, USA: IEEE Computer Society, 2009. p. 125–131.

KROEGER, T. M.; LONG, D. D. E. The case for efficient file access pattern modeling. In: WORKSHOP ON HOT TOPICS IN OPERATING SYSTEMS, 7th., 1999, Rio Rico, AZ. **Proceedings**... Rio Rico, AZ: IEEE, 1999. p. 14–19.

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

LEBRE, A. et al. I/o scheduling service for multi-application clusters. In: IEEE CONFERENCE ON CLUSTER COMPUTING, 2006, Barcelona, Spain. **Proceedings**... Barcelona, Spain: IEEE, 2006. p. 1–10.

LIN, L. et al. Amp: an affinity-based metadata prefetching scheme in large-scale distributed storage systems. In: IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID (CCGRID), 8th., 2008, Lyon, France. **Proceedings**... Lyon, France: IEEE, 2008. p. 459–466.

MADHYASTHA, T. M.; REED, D. A. Learning to classify parallel input/output access patterns. **IEEE Transactions on Parallel and Distributed Systems**, [S.l.], v. 13, n. 8, p. 802–813, 2002.

MARTIN, R. P.; CULLER, D. E. Nfs sensitivity to high performance networks. **SIGMETRICS Performance Evaluation Review**, New York, NY, USA, v. 27, n. 1, p. 71–82, 1999.

MIN, C. et al. Sfs: random write considered harmful in solid state drives. In: USENIX CONFERENCE ON FILE AND STORAGE TECHNOLOGIES, 10th., 2012, San Jose, CA, USA. **Proceedings**... San Jose, CA, USA: USENIX Association, 2012. p. 12–12.

NORCOTT, W. D.; CAPPS, D. **Iozone filesystem benchmark**, [S.l.: s.n.], 2006. Available at www.iozone.org. Accessed in March 2014.

OHTA, K.; MATSUBA, H.; ISHIKAWA, Y. Improving parallel write by node-level request scheduling. In: IEEE/ACM INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID, 9th., 2009, Washington, DC, USA. **Proceedings**... Washington, DC, USA: IEEE, 2009. p. 196–203.

OSTHOFF, C. et al. Improving performance on atmospheric models through a hybrid openmp/mpi implementation. In: IEEE INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING WITH APPLICATIONS (ISPA), 9th., 2011, Busan, South Korea. **Proceedings**... Busan, South Korea: IEEE, 2011. p. 69–74.

PATRICK, C. M. et al. Cashing in on hints for better prefetching and caching in pvfs and mpi-io. In: ACM INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING, 19th., 2010, New York, NY, USA. **Proceedings**... New York, NY, USA: ACM, 2010. p. 191–202.

PATTERSON, D. A.; HENNESSY, J. L. **Computer organization and design: the hardware/software interface**. 5. ed. [S.l.]: Morgan Kaufmann, 2013.

QIU, S.; REDDY, A. L. N. Nvmfs: A hybrid file system for improving random write in nand-flash ssd. In: IEEE SYMPOSIUM ON MASS STORAGE SYSTEMS AND TECHNOLOGIES (MSST), 29th., 2013, Long Beach, CA, USA. **Proceedings**... Long Beach, CA, USA: IEEE, 2013. p. 1–5.

RAJIMWALE, A.; PRABHAKARAN, V.; DAVIS, J. D. Block management in solid-state devices. In: USENIX ANNUAL TECHNICAL CONFERENCE, 2009, San Jose, CA, USA. **Proceedings**... San Jose, CA, USA: USENIX Association, 2009. p. 279–284.

ROSS, R. et al. A case study in application i/o on linux clusters. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2001, New York, NY, USA. **Proceedings**... New York, NY, USA: ACM, 2001. p. 11–22.

SCHMUCK, F.; HASKIN, R. Gpfs: A shared-disk file system for large computing clusters. In: USENIX CONFERENCE ON FILE AND STORAGE TECHNOLOGIES, 2002, Berkeley, CA, USA. **Proceedings**... Berkeley, CA, USA: USENIX Association, 2002. p. 19–31.

SEAMONS, K. E. et al. Server-directed collective i/o in panda. In: IEEE/ACM CONFERENCE ON SUPERCOMPUTING, 1995, San Diego, CA, USA. **Proceedings**... San Diego, CA, USA: IEEE, 1995. p. 57–69.

SEELAM, S. et al. Masking i/o latency using application level i/o caching and prefetching on blue gene systems. In: IEEE INTERNATIONAL SYMPOSIUM ON PARALLEL & DISTRIBUTED PROCESSING (IPDPS), 2010, Atlanta, GA, USA. **Proceedings**... Atlanta, GA, USA: IEEE, 2010. p. 1–12.

SHAN, H.; ANTYPAS, K.; SHALF, J. Characterizing and predicting the i/o performance of hpc applications using a parameterized synthetic benchmark. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2008, Piscataway, NJ, USA. **Proceedings**... Piscataway, NJ, USA: IEEE Press, 2008. p. 42–53.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Operating system concepts**. 9. ed. [S.l.]: Wiley, 2013.

SONG, H. et al. Server-side i/o coordination for parallel file systems. In: INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS, 2011, New York, NY, USA. **Proceedings**... New York, NY, USA: ACM, 2011. p. 1–11.

SOUNDARARAJAN, G.; MIHAILESCU, M.; AMZA, C. Context-aware prefetching at the storage server. In: USENIX ANNUAL TECHNICAL CONFERENCE, 2008, San Jose, CA, USA. **Proceedings**... San Jose, CA, USA: USENIX Association, 2008. p. 377–390.

SUN MICROSYSTEMS, INC. **NFS: Network File System Protocol specification**, USA: Sun Microsystems, Inc, 1989.

TANTISIRIROJ, W. et al. On the duality of data-intensive file system design: reconciling hdfs and pvfs. In: INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS, 2011, Seatle, WA, USA. **Proceedings**... Seatle, WA, USA: IEEE, 2011. p. 1–12.

TESSER, R. K. et al. Improving the performance of seismic wave simulations with dynamic load balancing. In: EUROMICRO INTERNATIONAL CONFERENCE ON PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING (PDP), 2014, Turin, Italy. **Proceedings**... Turin, Italy: IEEE, 2014. p. 196–203.

THAKUR, R.; GROPP, W.; LUSK, E. Data sieving and collective i/o in romio. In: THE SYMPOSIUM ON THE FRONTIERS OF MASSIVELY PARALLEL COMPUTATION , 7th., 1999, Annapolis, MD, USA. **Proceedings**... Annapolis, MD, USA: IEEE, 1999. p. 182–189.

THANH, T. D. et al. A taxonomy and survey on distributed file systems. In: IEEE. INTERNATIONAL CONFERENCE ON NETWORKED COMPUTING AND ADVANCED INFORMATION MANAGEMENT, 4th., 2008, Gyeongju, South Korea. **Proceedings**... Gyeongju, South Korea: IEEE, 2008. p. 144–149.

WANG, A.-I. A. et al. The conquest file system: Better performance through a disk/persistent-ram hybrid design. **ACM Transactions on Storage (TOS)**, New York, NY, USA, v. 2, n. 3, p. 309–348, aug 2006.

WITTEN, I. H.; FRANK, E. **Data Mining: Practical machine learning tools and techniques**. 2. ed. [S.l.]: Morgan Kaufmann, 2005.

WORRINGEN, J. Self-adaptive hints for collective i/o. In: EUROPEAN PVM/MPI USER'S GROUP CONFERENCE ON RECENT ADVANCES IN PARALLEL VIRTUAL MACHINE AND MESSAGE PASSING INTERFACE, 13th., 2006, Berlin, Heidelberg. **Proceedings**... Berlin, Heidelberg: Springer, 2006. p. 202–211.

XU, W. et al. Hybrid hierarchy storage system in milkyway-2 supercomputer. **Frontiers of Computer Science**, [S.l.], v. 8, n. 3, p. 367–377, 2014.

YOO, J. et al. Vssim: Virtual machine based ssd simulator. In: IEEE SYMPOSIUM ON MASS STORAGE SYSTEMS AND TECHNOLOGIES (MSST), 29th., 2013, Long Beach, CA, USA. **Proceedings**... Long Beach, CA, USA: IEEE, 2013. p. 1–14.

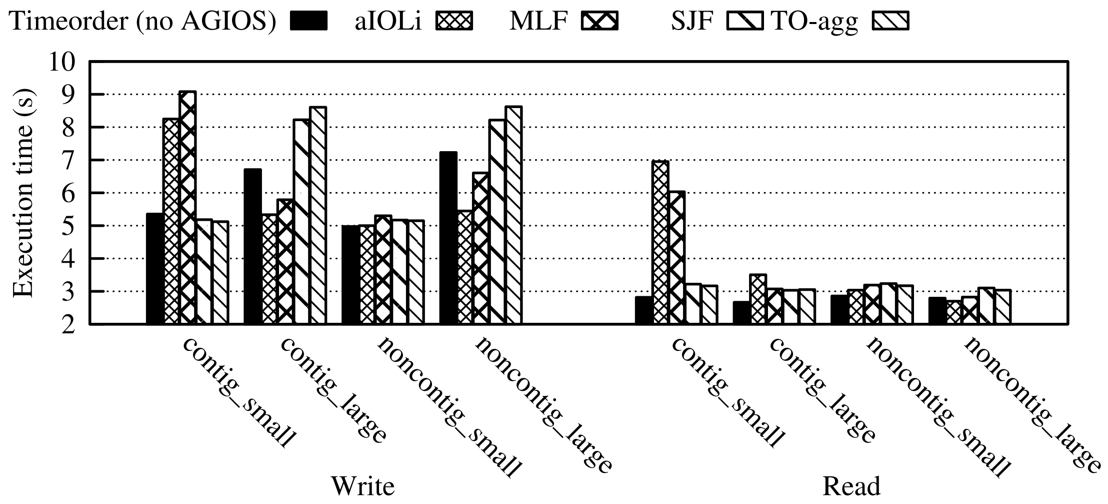
YU, W. et al. Empirical analysis of a large-scale hierarchical storage system. In: LUQUE, E.; MARGALEF, T.; BENÍTEZ, D. (Ed.). **Euro-Par 2008 – Parallel Processing**. [S.l.]: Springer, 2008.

ZHANG, X.; DAVIS, K.; JIANG, S. Qos support for end users of i/o-intensive applications using shared storage systems. In: INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS, 2011, New York, NY, USA. **Proceedings**... New York, NY, USA: ACM, 2011. p. 18–1.

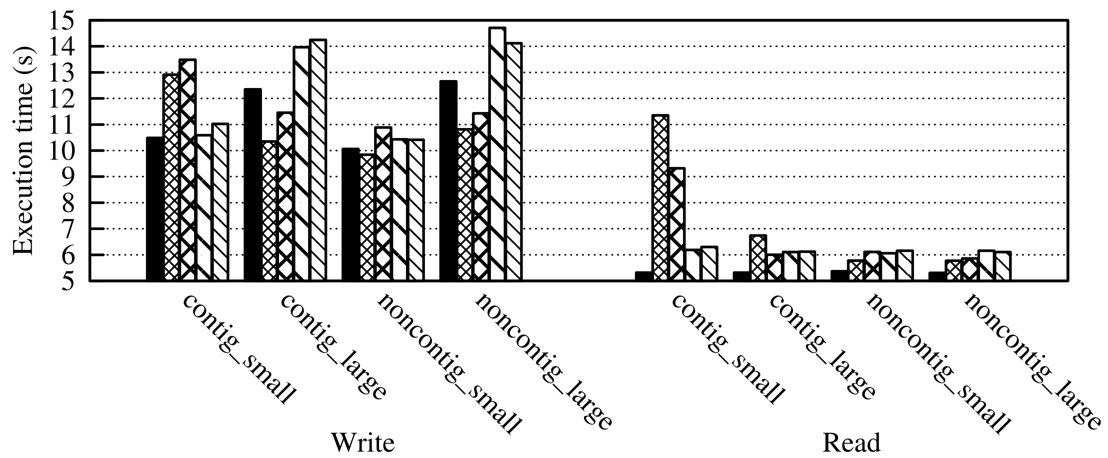
ZHANG, Y.; BHARGAVA, B. Self-learning disk scheduling. **IEEE Transactions on Knowledge and Data Engineering**, [S.l.], v. 21, n. 1, p. 50–65, 2008.

APPENDIX A I/O SCHEDULING ALGORITHMS PERFORMANCE EVALUATION

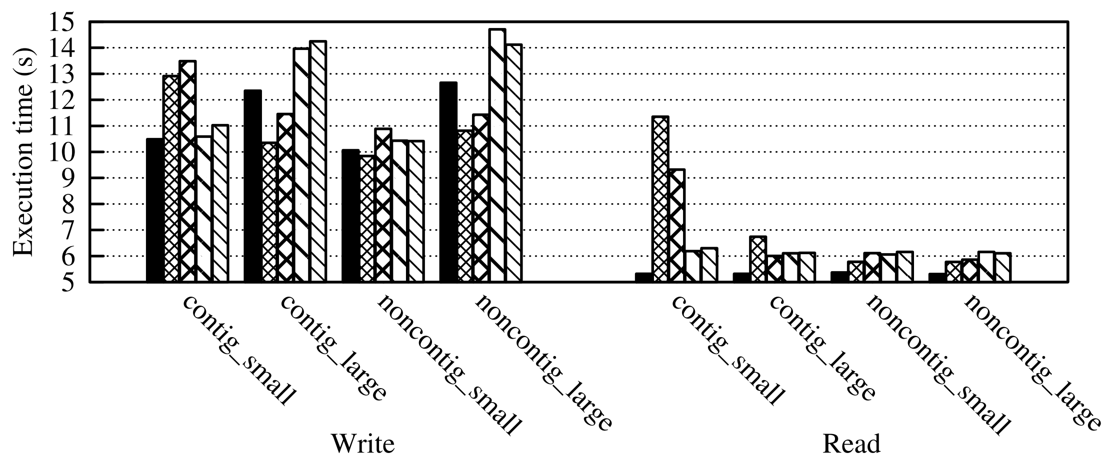
This chapter presents all obtained results from the performance evaluation discussed in Chapter 3. This evaluation was conducted with all our scheduling algorithms - described in Section 3.1 - on multiple platforms - described in Section 3.2.1 - with different access patterns - Section 3.2.2. Because of the large volume of results, only a part of them was shown with the discussion as to not compromise that chapter's readability. They are all presented here.



(a) 8 processes per application

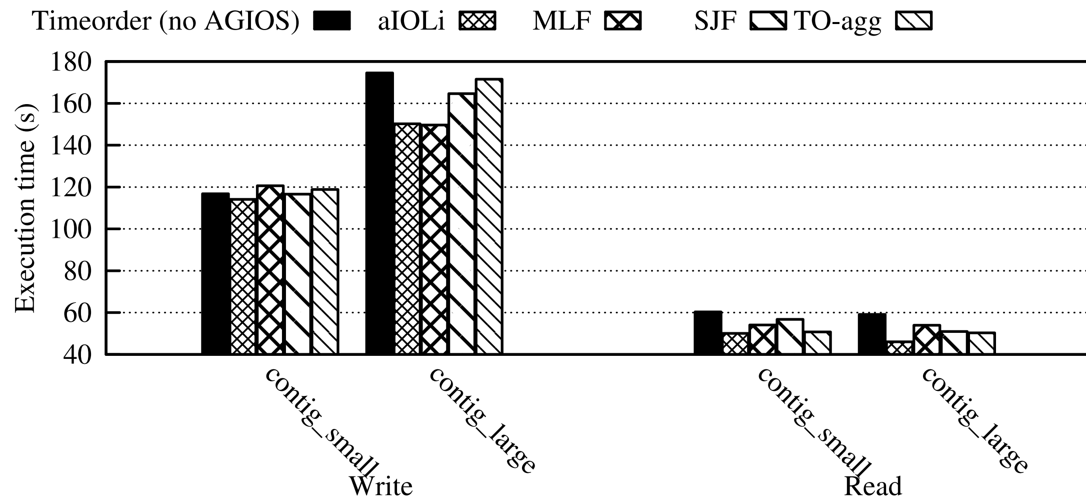


(b) 16 processes per application

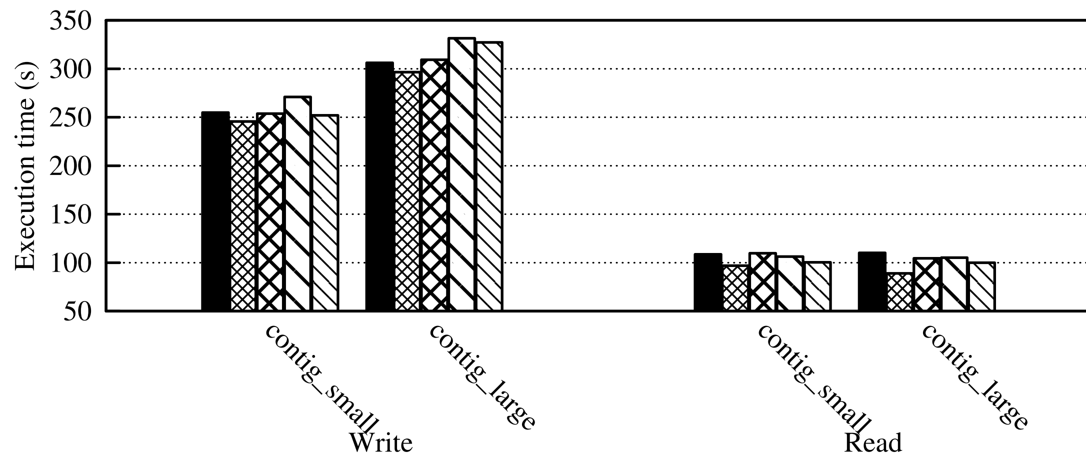


(c) 32 processes per application

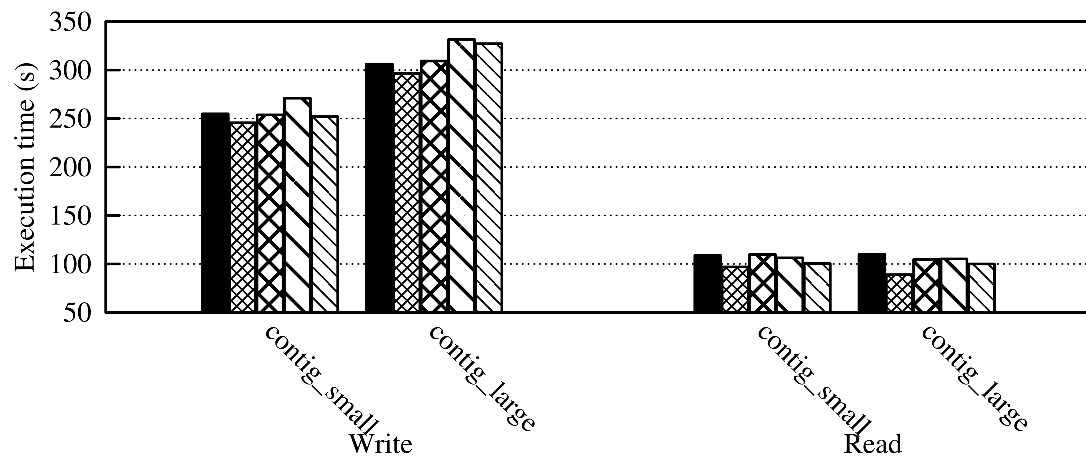
Figure A.1: Results obtained for the single application scenarios with the shared file approach in the Pastel cluster.



(a) 8 processes per application

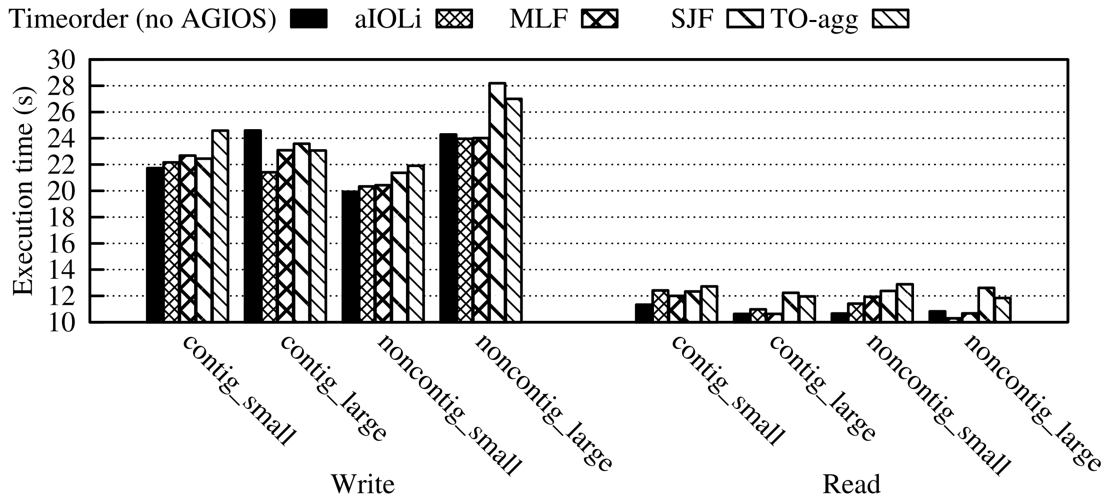


(b) 16 processes per application

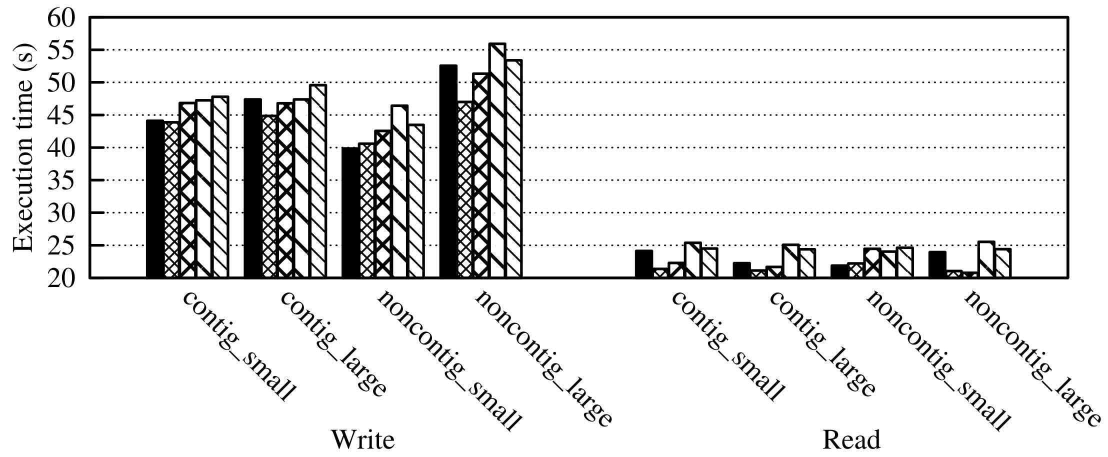


(c) 32 processes per application

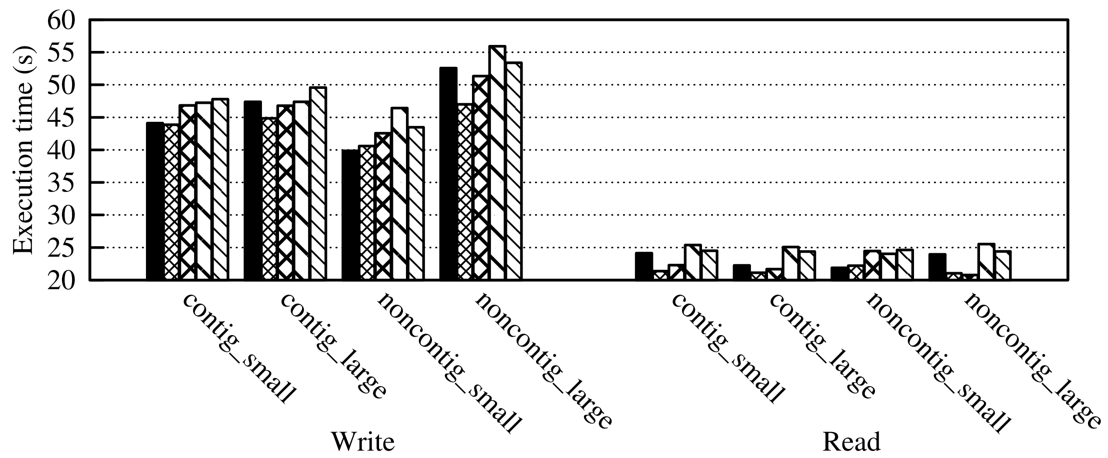
Figure A.2: Results obtained for the single application scenarios with the file per process approach in the Pastel cluster.



(a) 8 processes per application

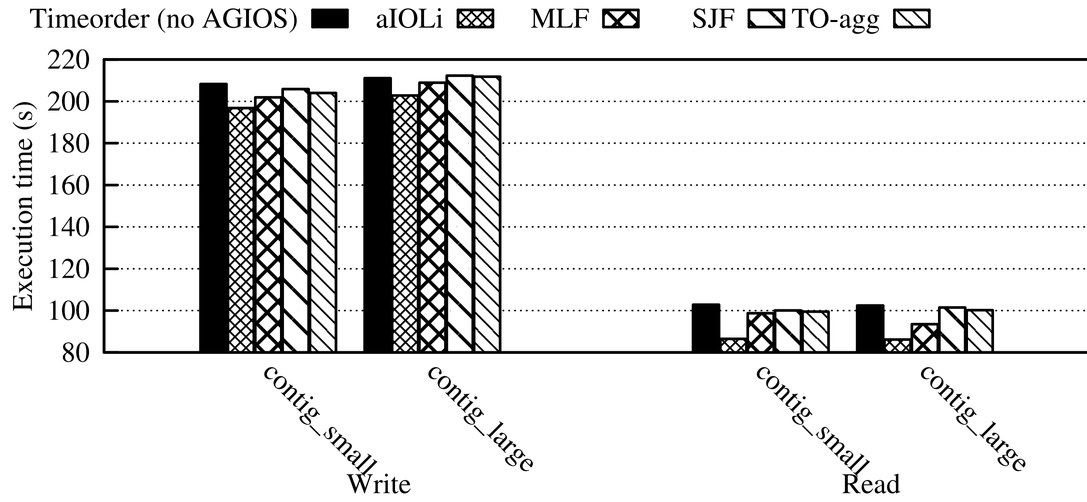


(b) 16 processes per application

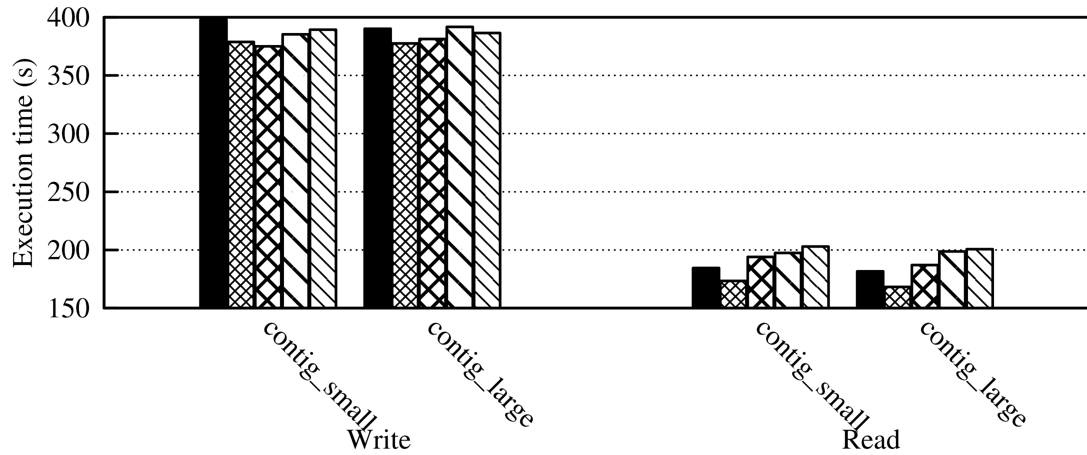


(c) 32 processes per application

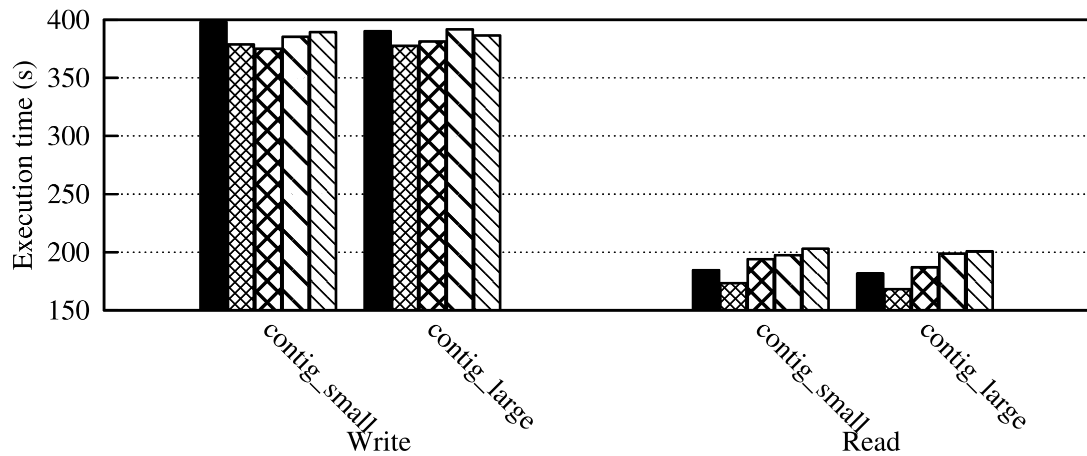
Figure A.3: Results obtained for the multi-application scenarios with the shared file approach in the Pastel cluster.



(a) 8 processes per application

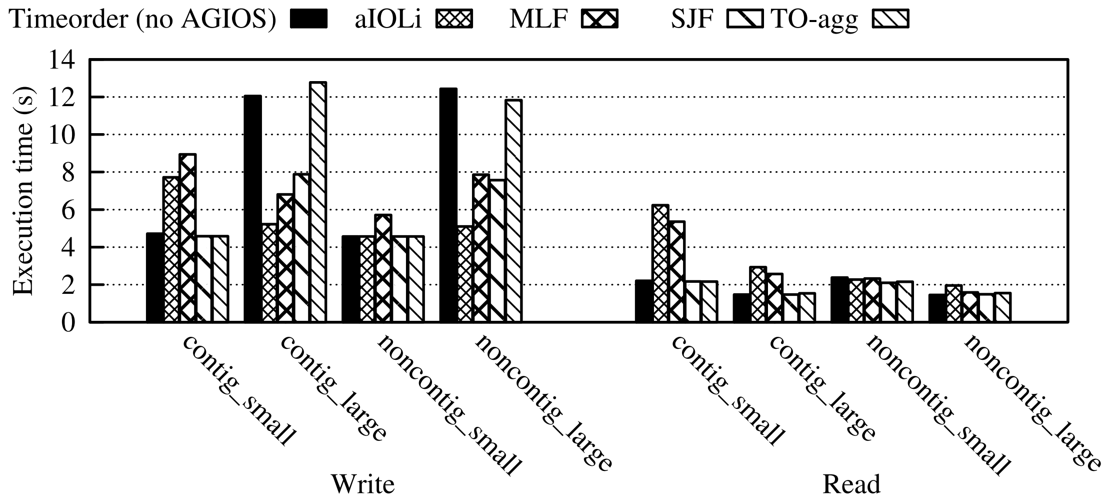


(b) 16 processes per application

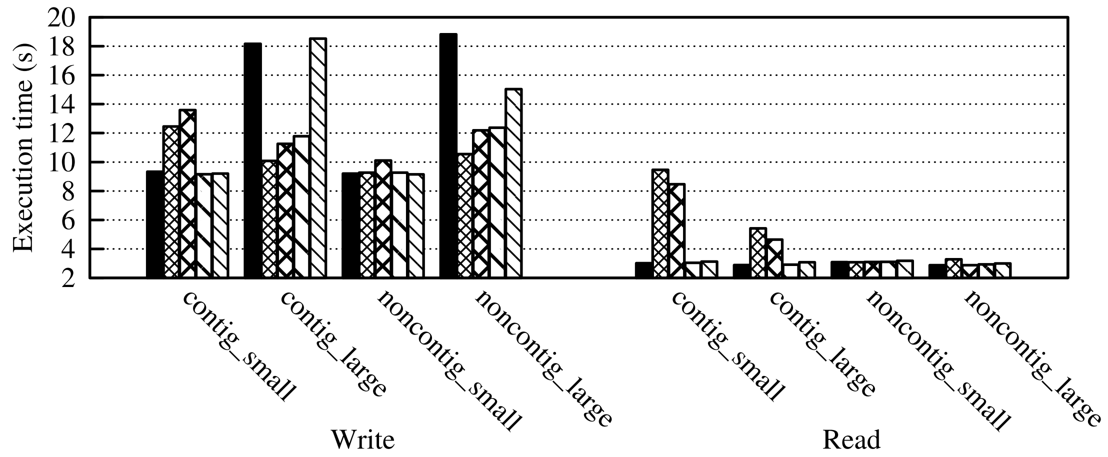


(c) 32 processes per application

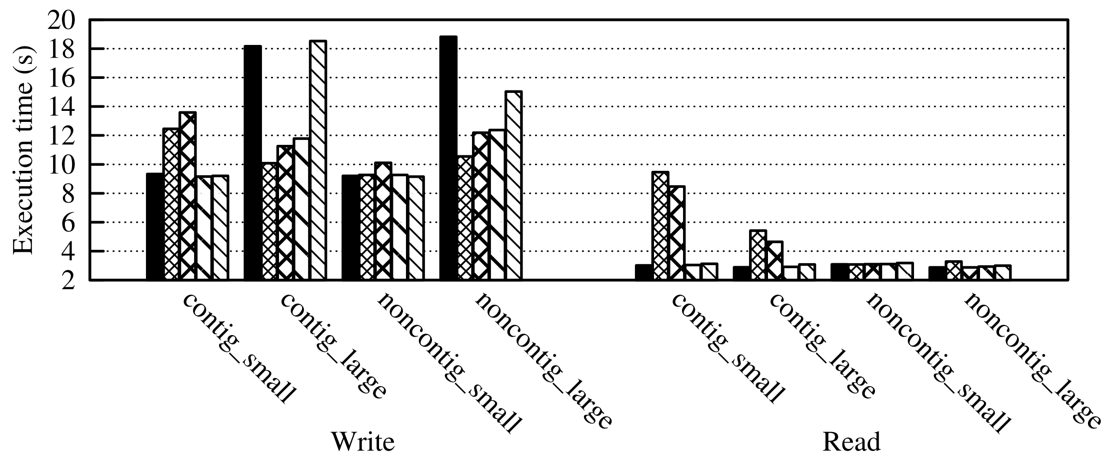
Figure A.4: Results obtained for the multi-application scenarios with the file per process approach in the Pastel cluster.



(a) 8 processes per application

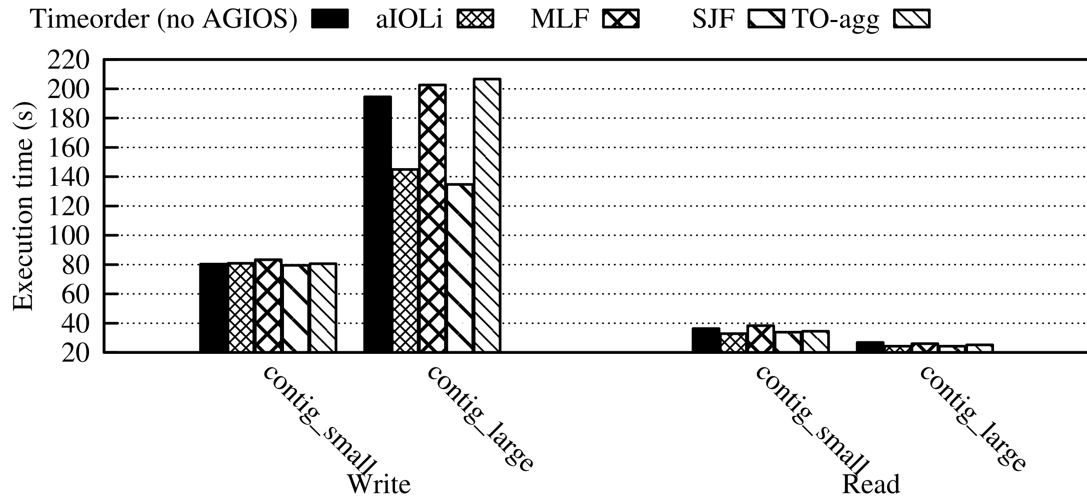


(b) 16 processes per application

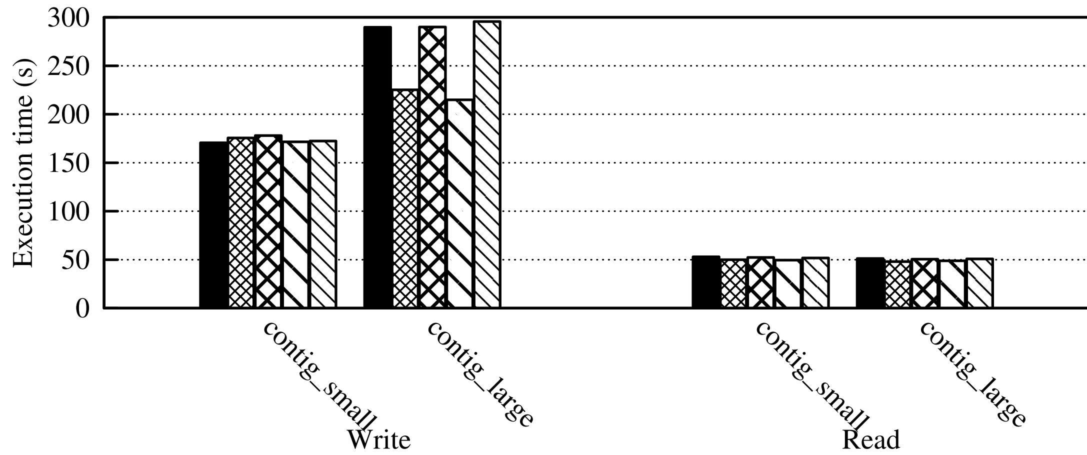


(c) 32 processes per application

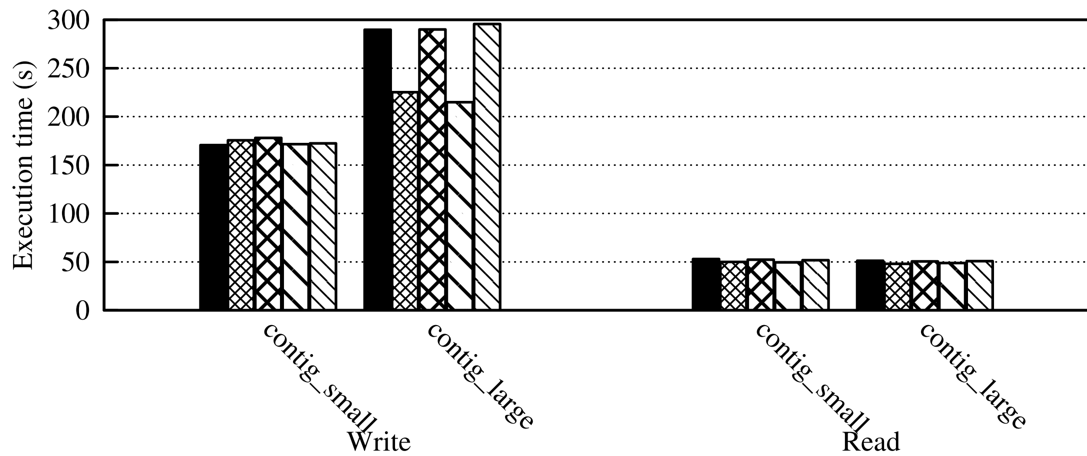
Figure A.5: Results obtained for the single application scenarios with the shared file approach in the Graphene cluster.



(a) 8 processes per application

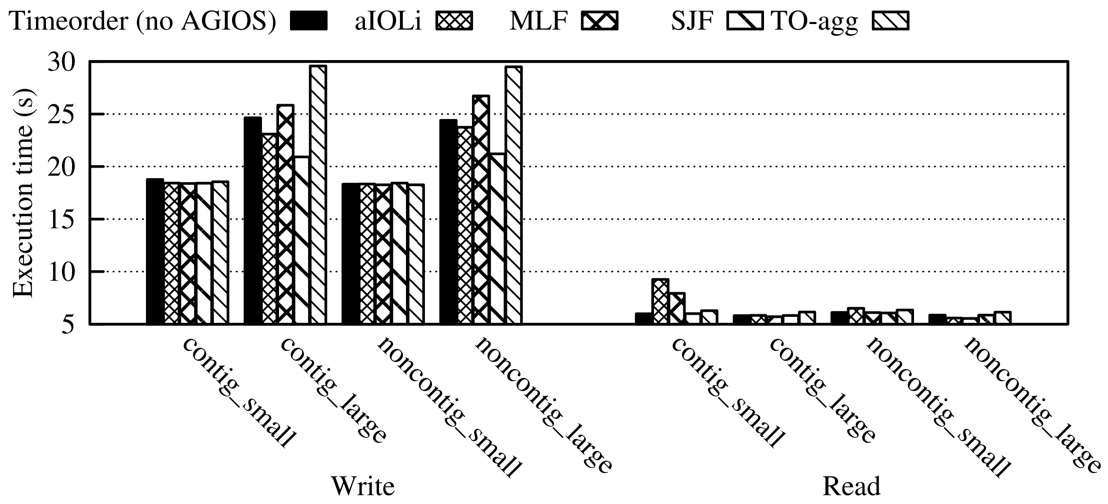


(b) 16 processes per application

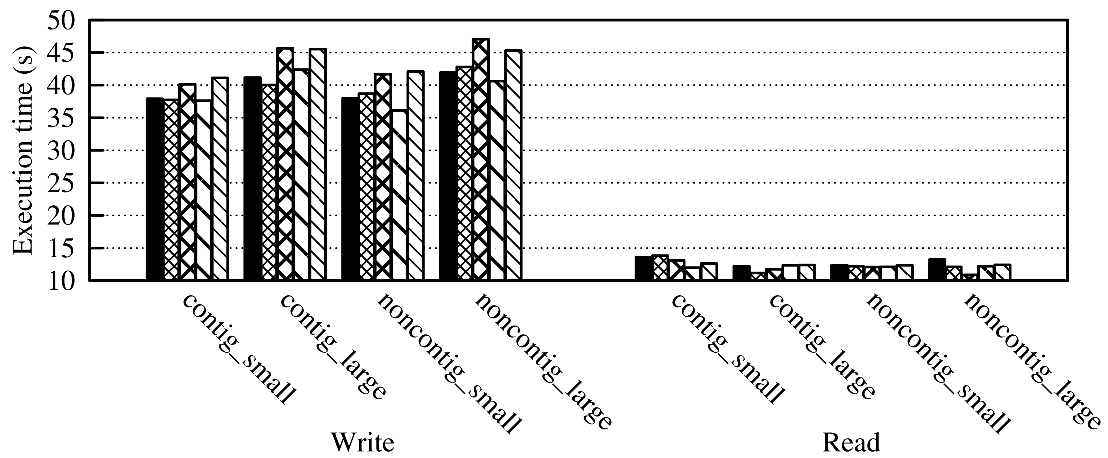


(c) 32 processes per application

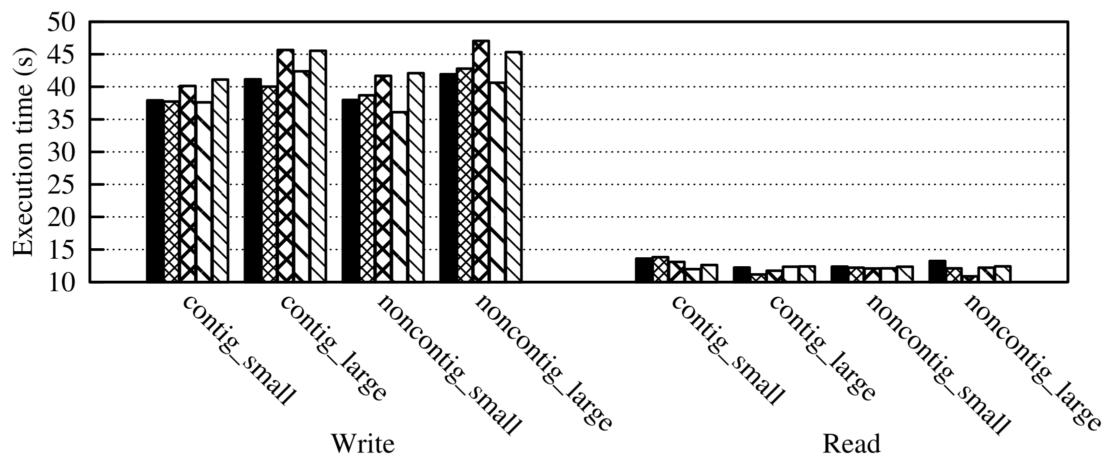
Figure A.6: Results obtained for the single application scenarios with the file per process approach in the Graphene cluster.



(a) 8 processes per application

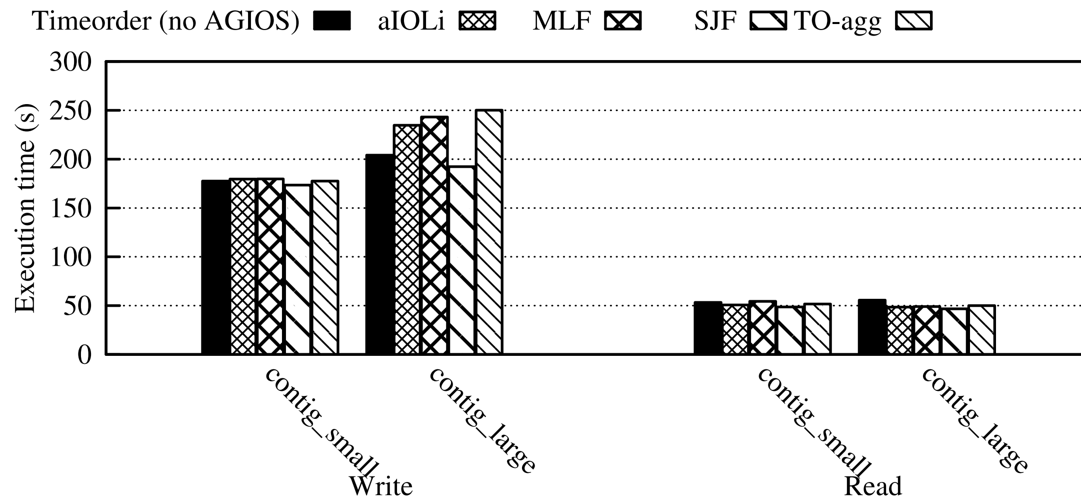


(b) 16 processes per application

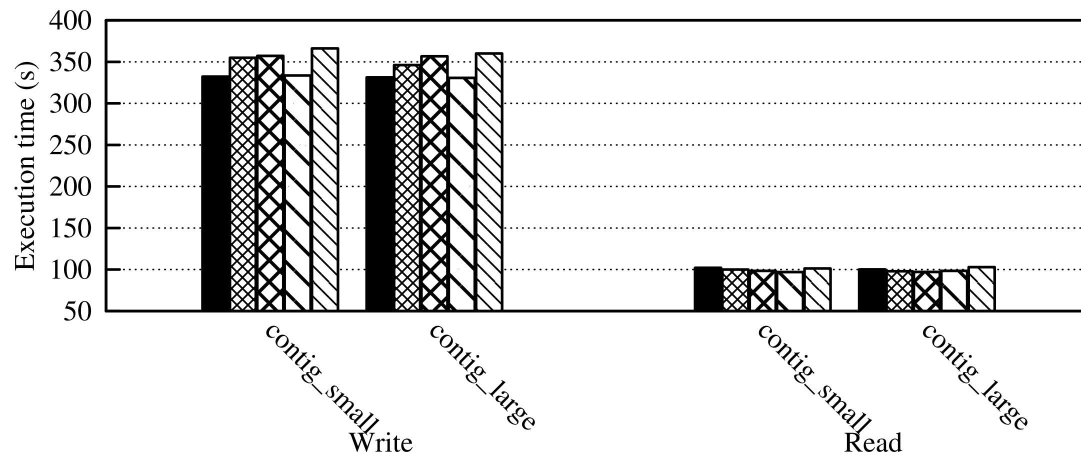


(c) 32 processes per application

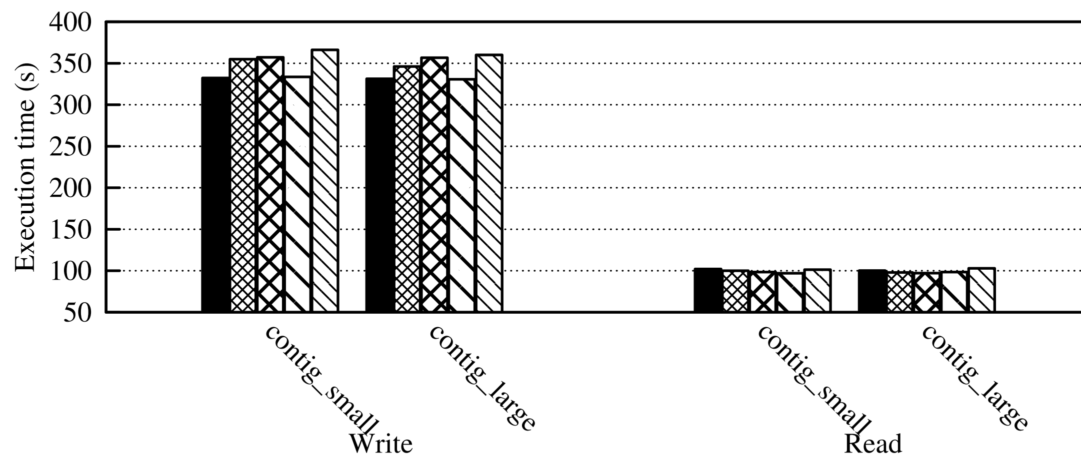
Figure A.7: Results obtained for the multi-application scenarios with the shared file approach in the Graphene cluster.



(a) 8 processes per application

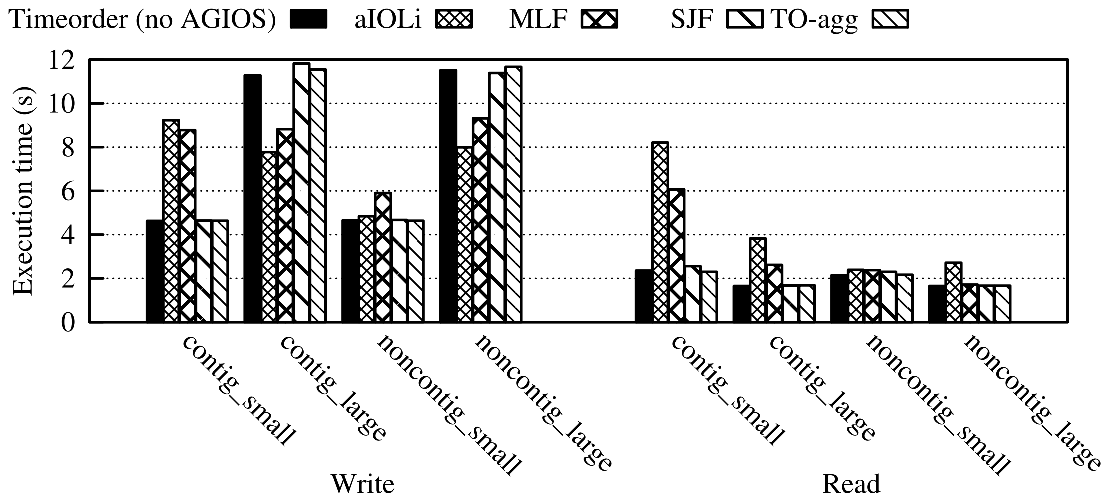


(b) 16 processes per application

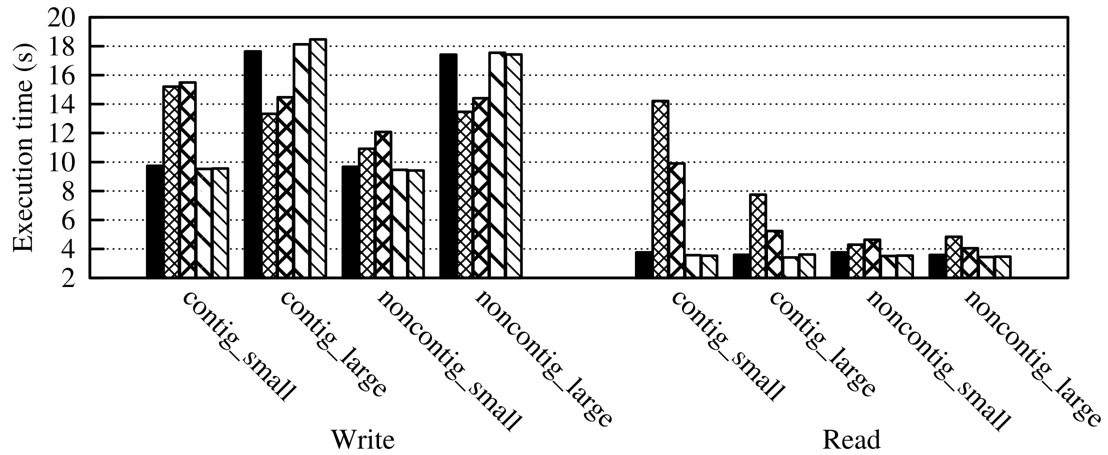


(c) 32 processes per application

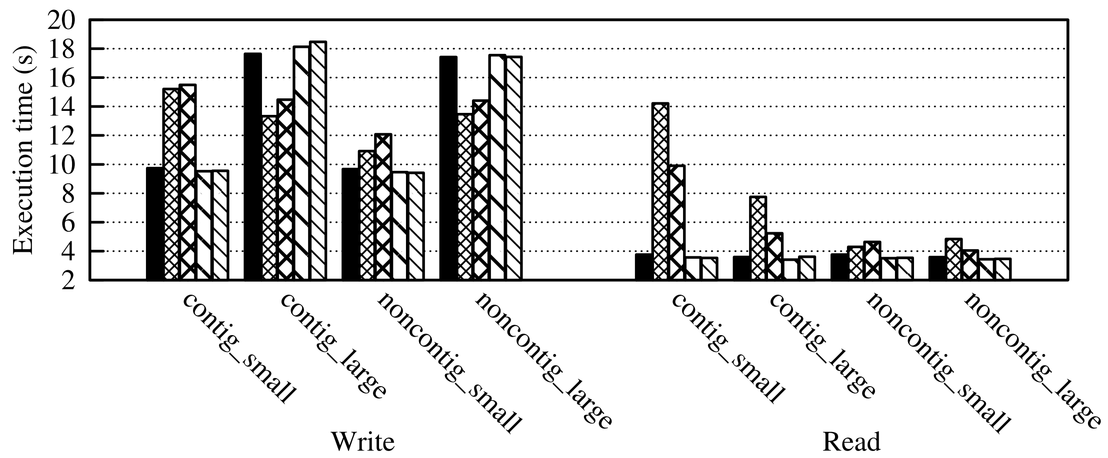
Figure A.8: Results obtained for the multi-application scenarios with the file per process approach in the Graphene cluster.



(a) 8 processes per application

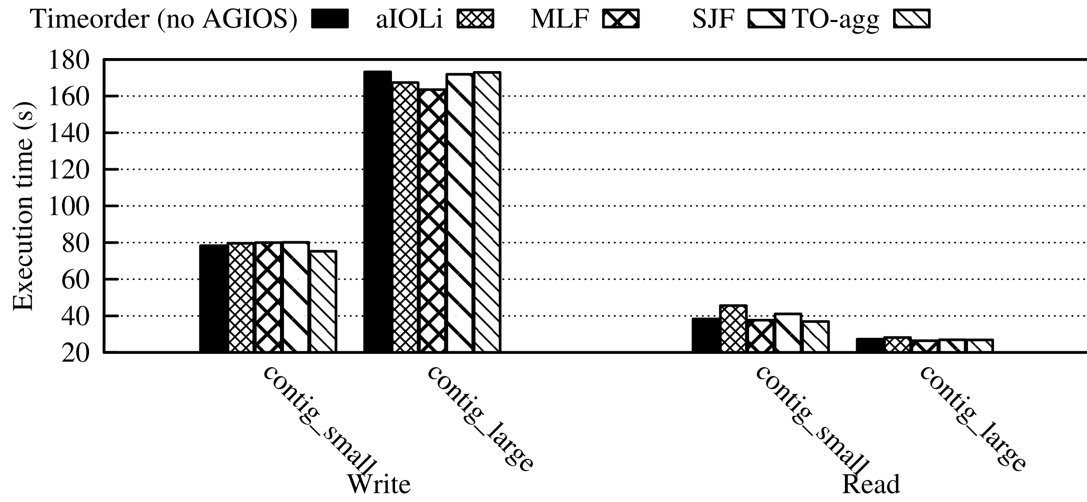


(b) 16 processes per application

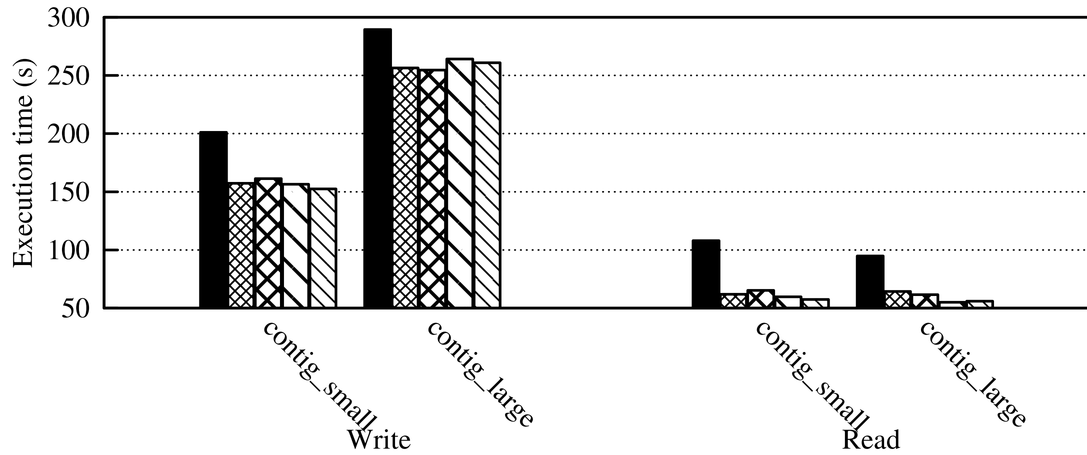


(c) 32 processes per application

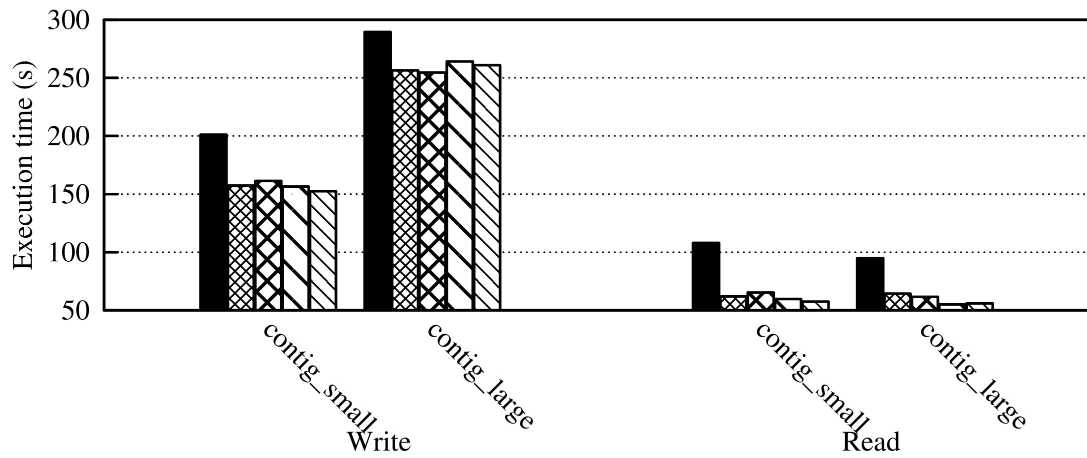
Figure A.9: Results obtained for the single application scenarios with the shared file approach in the Suno cluster.



(a) 8 processes per application

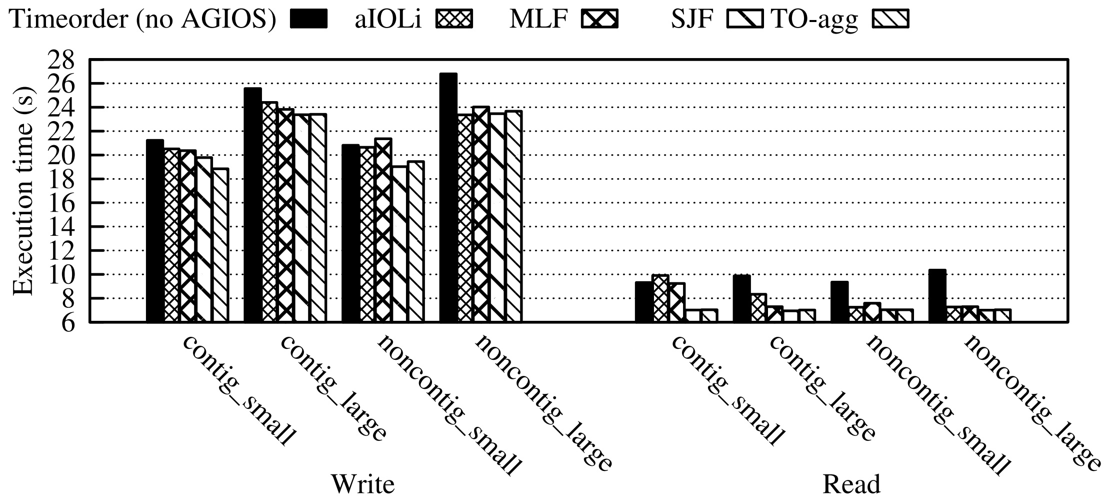


(b) 16 processes per application

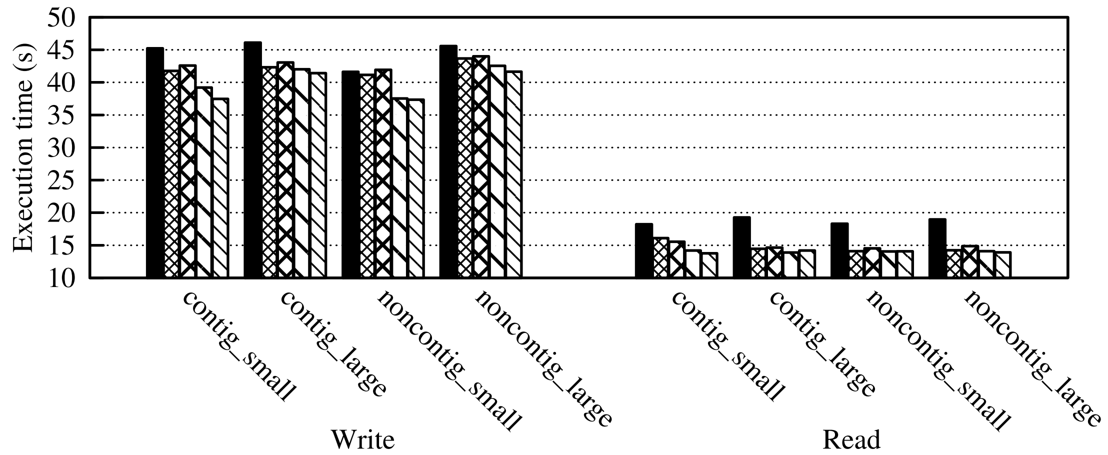


(c) 32 processes per application

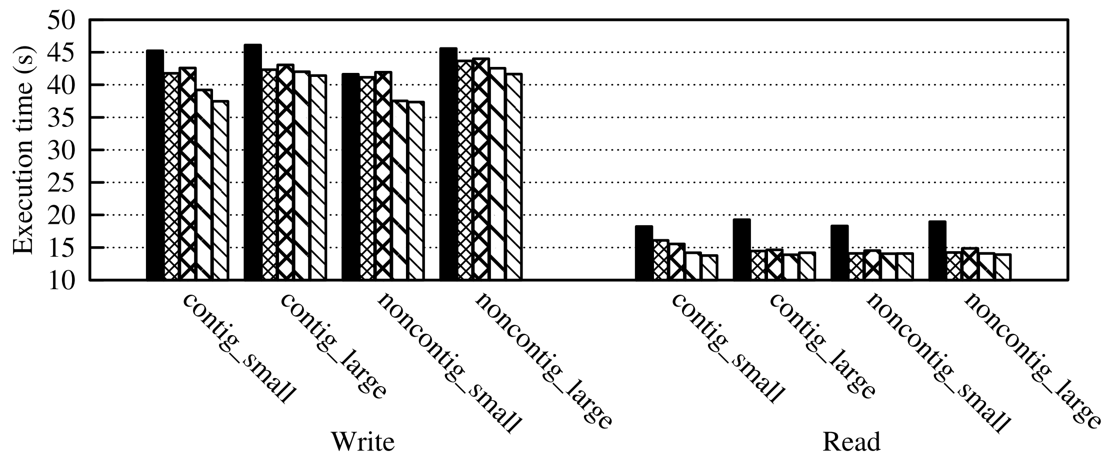
Figure A.10: Results obtained for the single application scenarios with the file per process approach in the Suno cluster.



(a) 8 processes per application

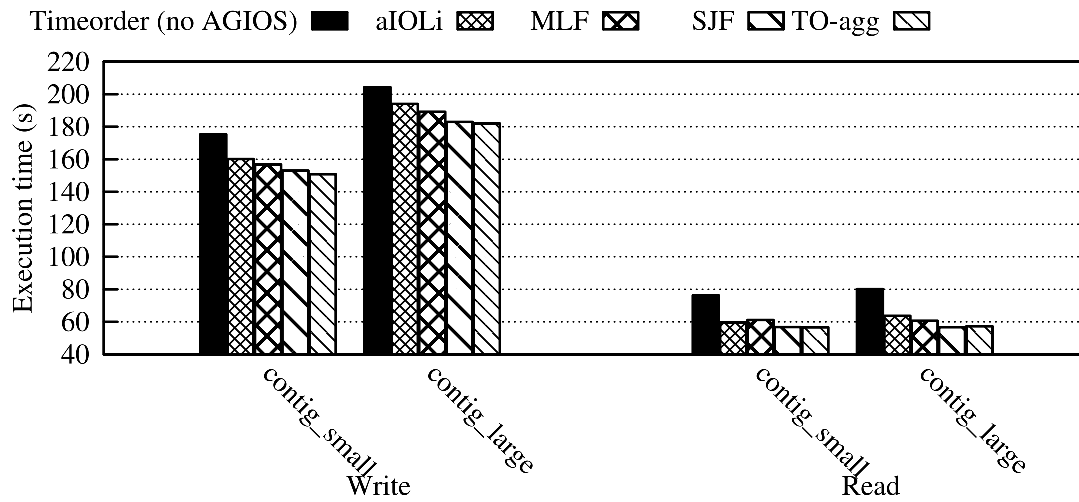


(b) 16 processes per application

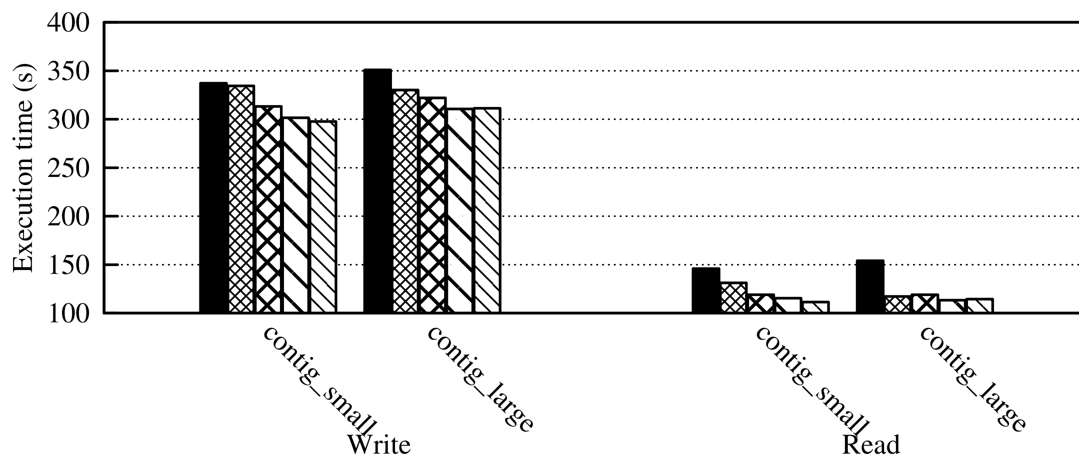


(c) 32 processes per application

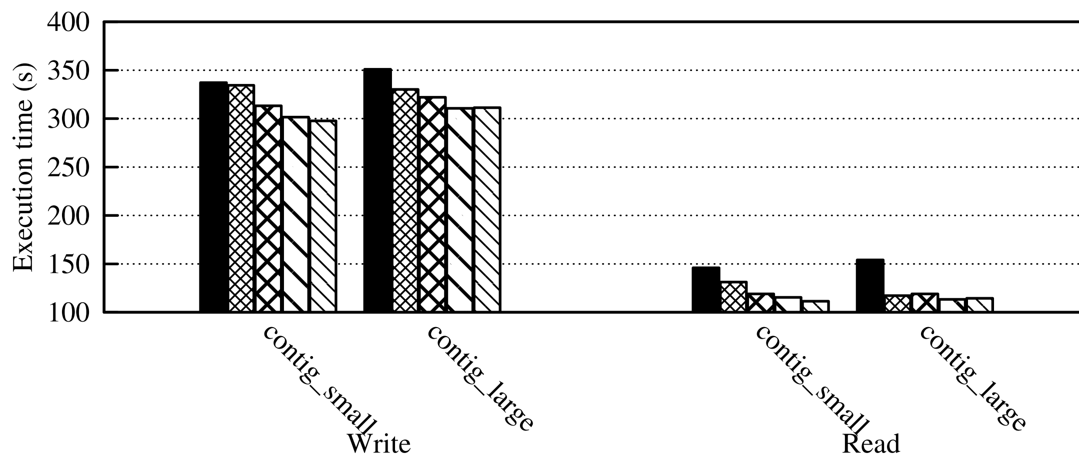
Figure A.11: Results obtained for the multi-application scenarios with the shared file approach in the Suno cluster.



(a) 8 processes per application

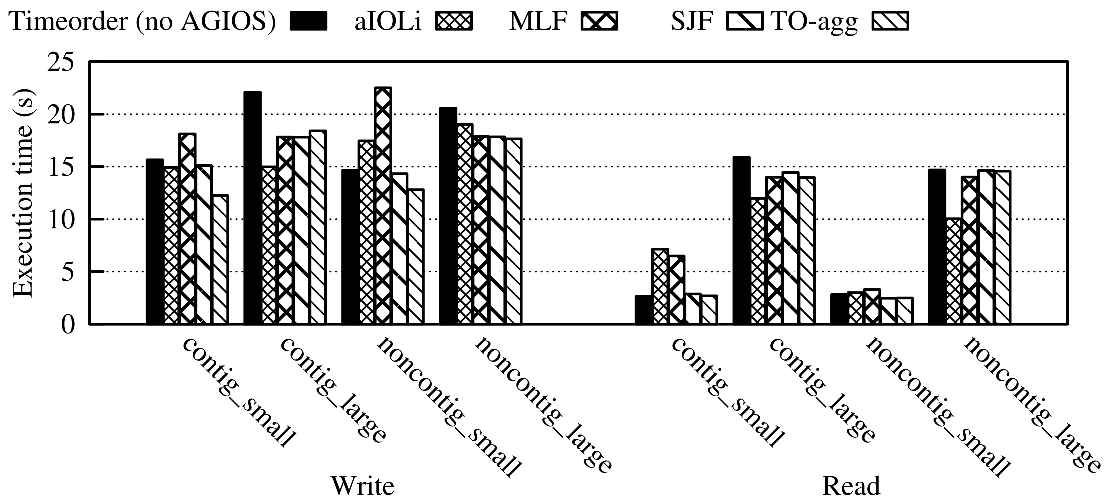


(b) 16 processes per application

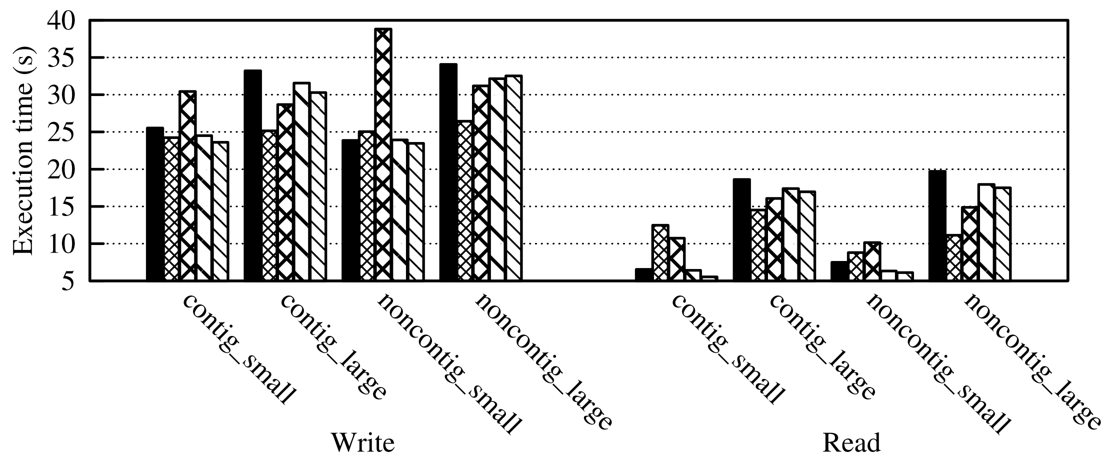


(c) 32 processes per application

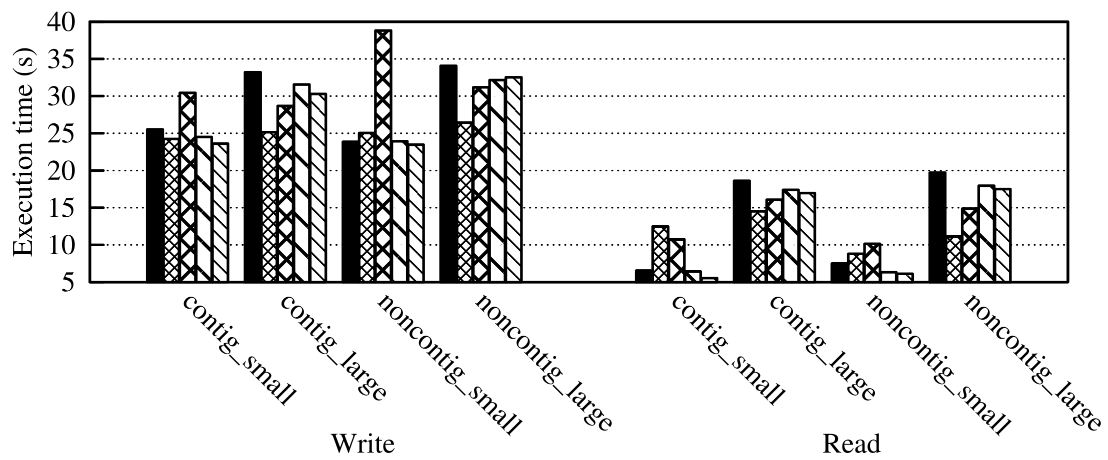
Figure A.12: Results obtained for the multi-application scenarios with the file per process approach in the Suno cluster.



(a) 8 processes per application

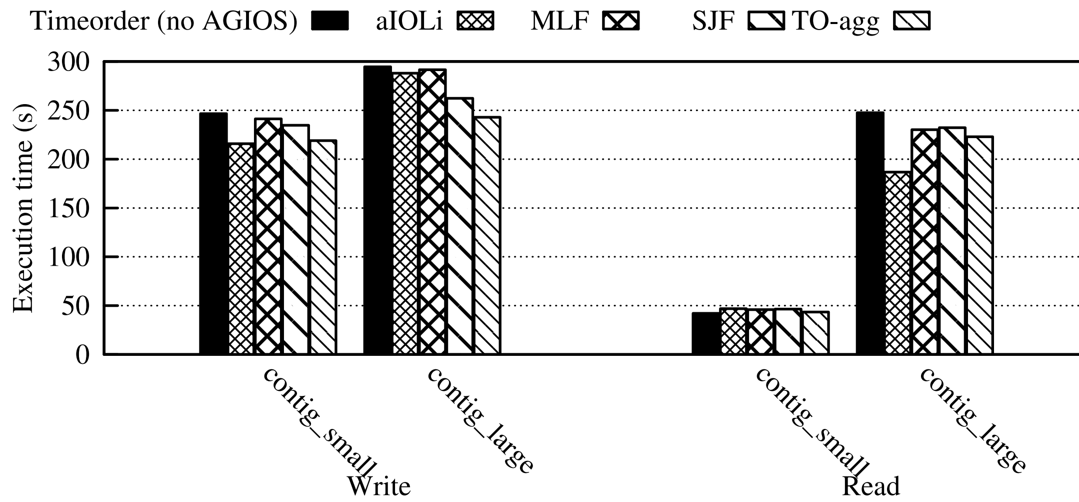


(b) 16 processes per application

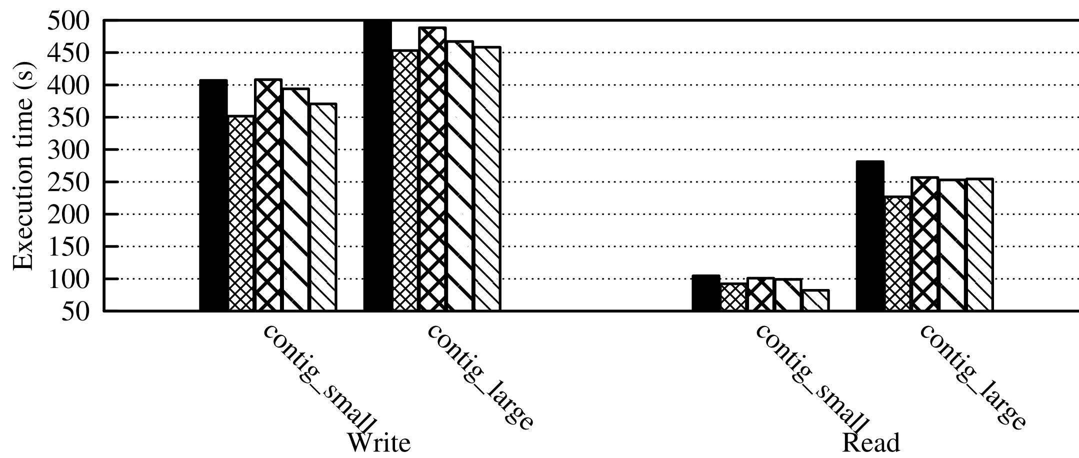


(c) 32 processes per application

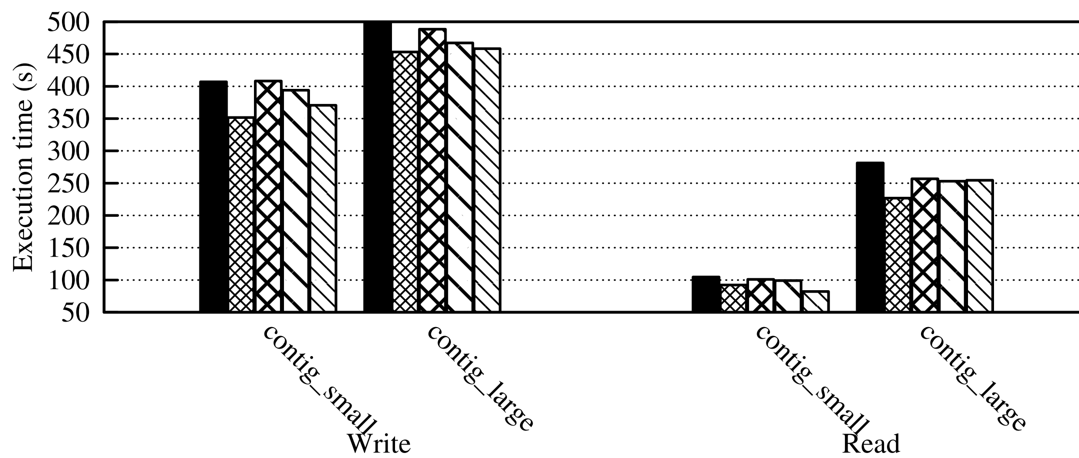
Figure A.13: Results obtained for the single application scenarios with the shared file approach in the Edel cluster.



(a) 8 processes per application

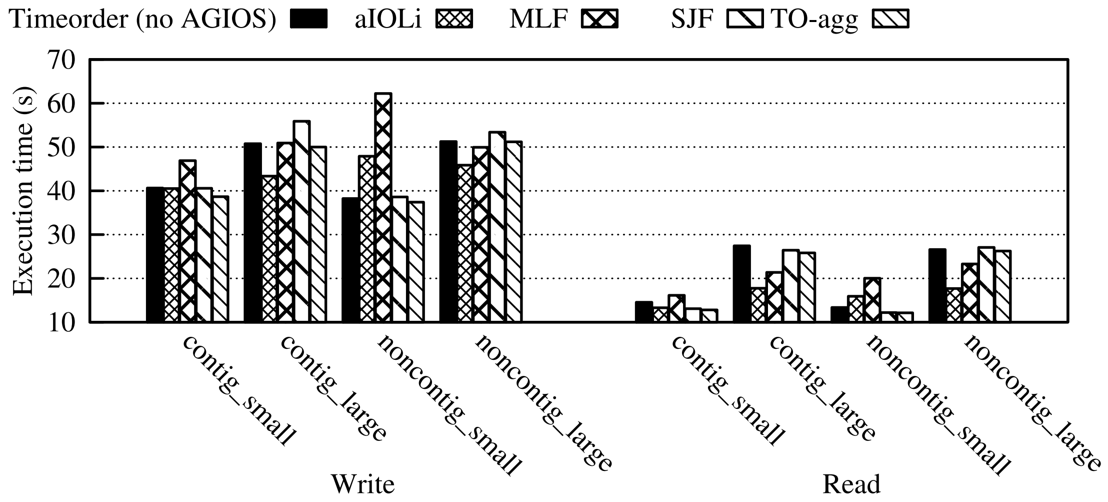


(b) 16 processes per application

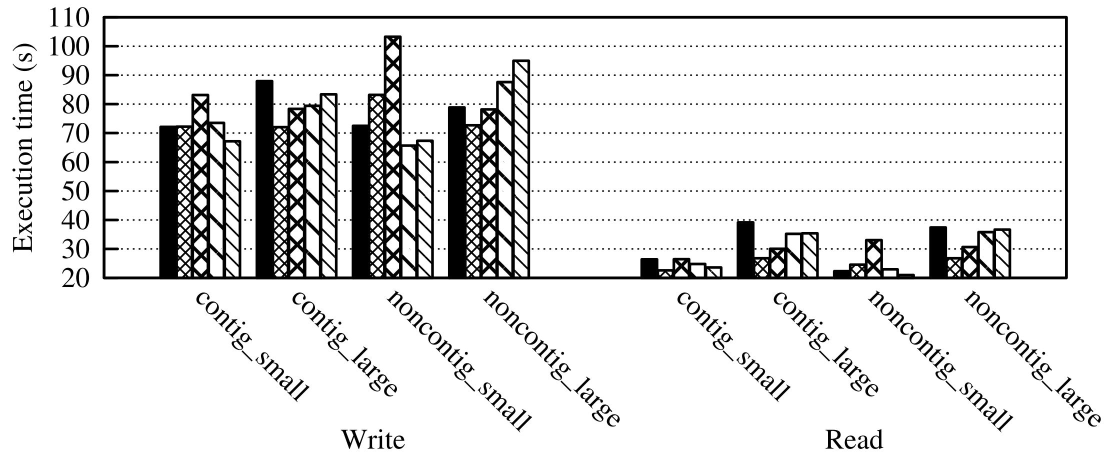


(c) 32 processes per application

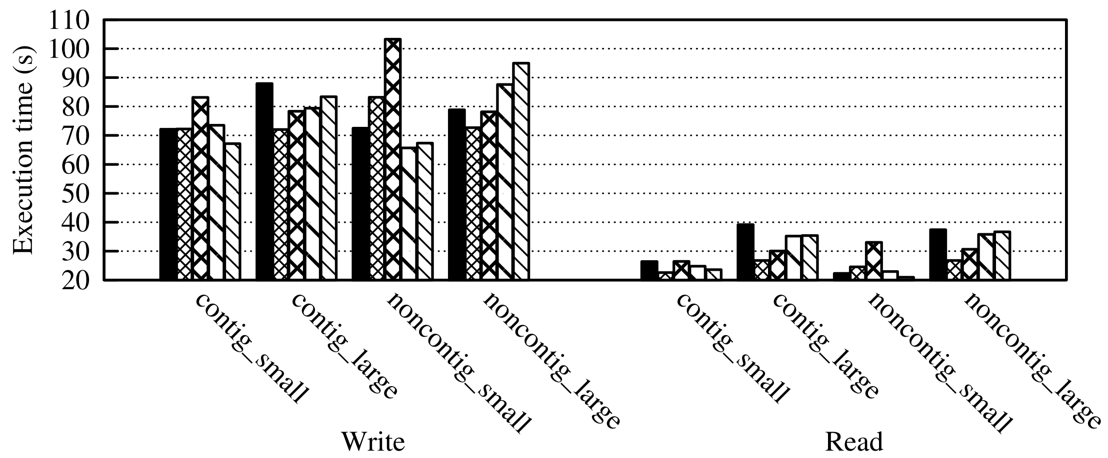
Figure A.14: Results obtained for the single application scenarios with the file per process approach in the Edel cluster.



(a) 8 processes per application

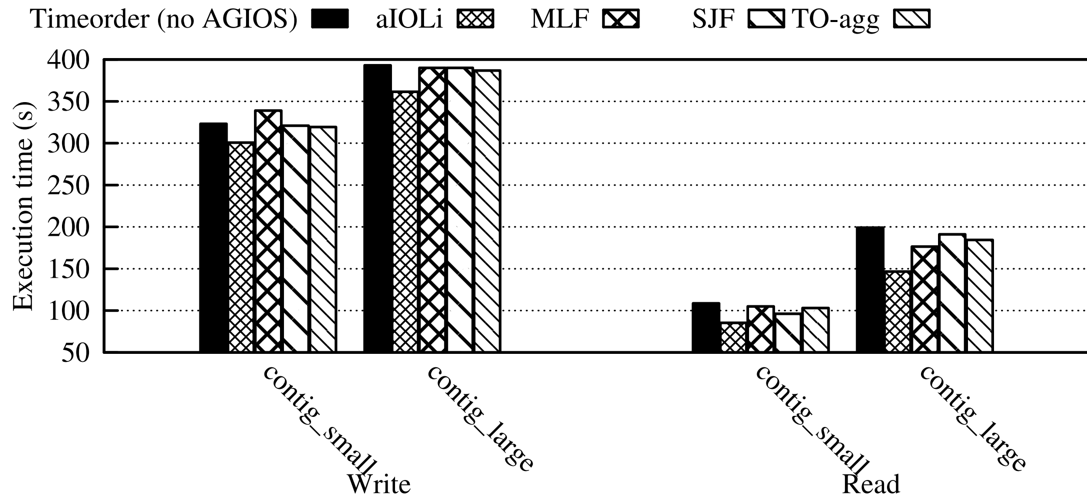


(b) 16 processes per application

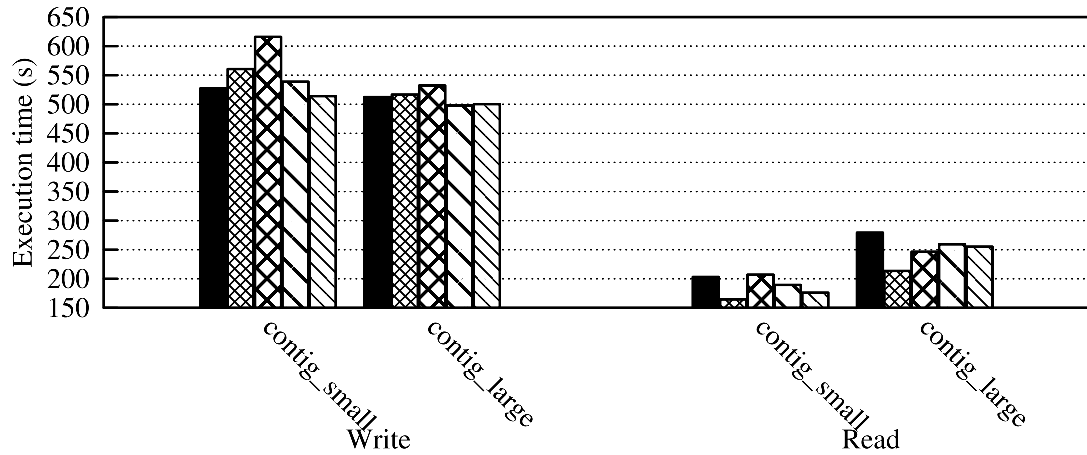


(c) 32 processes per application

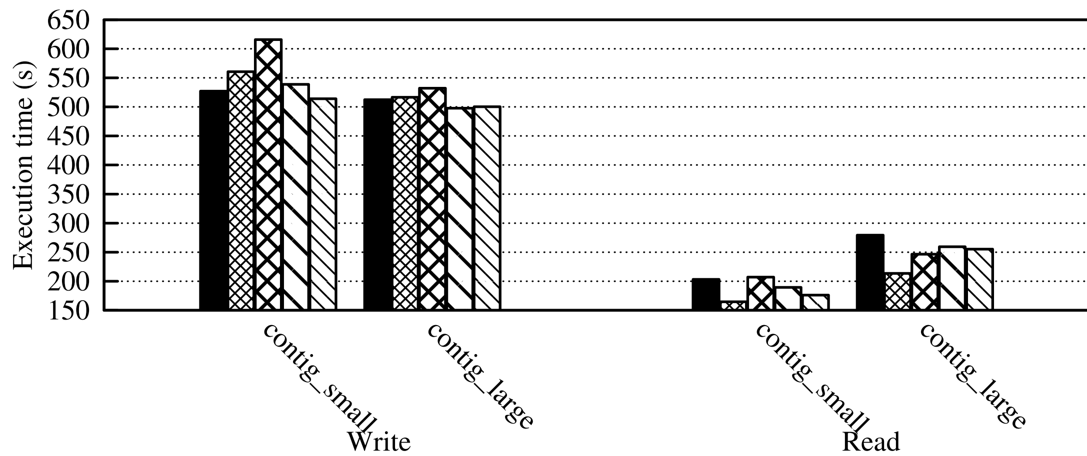
Figure A.15: Results obtained for the multi-application scenarios with the shared file approach in the Edel cluster.



(a) 8 processes per application



(b) 16 processes per application



(c) 32 processes per application

Figure A.16: Results obtained for the multi-application scenarios with the file per process approach in the Edel cluster.

APPENDIX B ABSTRACT IN FRENCH

B.1 Résumé

Cette thèse porte sur l'utilisation de l'ordonnancement d'Entrées/Sorties (E/S) pour atténuer les effets d'interférence et améliorer la performance des systèmes de fichiers parallèles. Il est commun pour les plates-formes de calcul haute performance (HPC) de fournir une infrastructure de stockage partagée pour les applications qui y sont hébergées. Dans cette situation, où plusieurs applications accèdent simultanément au système de fichiers parallèle partagé, leurs accès vont souffrir de l'interférence, ce qui compromet l'efficacité des stratégies d'optimisation d'E/S.

Nous avons évalué la performance de cinq algorithmes d'ordonnancement dans les serveurs de données d'un système de fichiers parallèle. Ces tests ont été exécutées sur différentes plates-formes et sous différents modèles d'accès. Les résultats indiquent que la performance des ordonnanceurs est affectée par les modèles d'accès des applications, car il est important pour améliorer la performance obtenue grâce à un algorithme d'ordonnancement de surpasser ses surcoûts. En même temps, les résultats des ordonnanceurs sont affectés par les caractéristiques du système d'E/S sous-jacent - en particulier par des dispositifs de stockage. Différents dispositifs présentent des niveaux de sensibilité à la séquentialité et la taille de accès distincts, ce qui peut influencer sur le niveau d'amélioration obtenue grâce à l'ordonnancement d'E/S.

Pour ces raisons, l'objectif principal de cette thèse est de proposer un modèle d'ordonnancement d'E/S avec une double adaptabilité: aux applications et aux dispositifs. Nous avons extrait des informations sur les modèles d'accès des applications en utilisant des fichiers de trace, obtenus à partir de leur exécutions précédentes. Ensuite, nous avons utilisé de l'apprentissage automatique pour construire un classificateur capable d'identifier la spatialité et la taille des accès à partir du flux de demandes antérieures. En outre, nous avons proposé une approche pour obtenir efficacement le ratio de débit séquentiel et aléatoire pour les dispositifs de stockage en exécutant des benchmarks pour un sous-ensemble des paramètres et en estimant les restants avec des régressions linéaires.

Nous avons utilisé les informations sur les caractéristiques des applications et des dispositifs de stockage pour décider automatiquement l'algorithme d'ordonnancement le plus approprié en utilisant des arbres de décision. Notre approche améliore les performances jusqu'à 75% par rapport à une approche qui utilise le même algorithme d'ordonnancement à toutes

les situations, sans capacité d'adaptation. De plus, notre approche améliore la performance dans 64% de scénarios en plus, et diminue les performances dans 89% moins de situations. Nos résultats montrent que les deux aspects - des applications et des dispositifs - sont essentiels pour faire de bons choix d'ordonnancement. En outre, malgré le fait qu'il n'y a pas d'algorithme d'ordonnancement qui fournit des gains de performance pour toutes les situations, nous montrons que avec la double adaptabilité il est possible d'appliquer des techniques d'ordonnancement d'E/S pour améliorer la performance, tout en évitant les situations où cela conduirait à une diminution de performance.

B.2 Résumé de thèse vulgarisé pour le grand public

Cette thèse porte sur l'ordonnancement d'E/S avec double adaptabilité: aux applications et aux dispositifs. Dans les environnements de cluster de calcul haute performance, des systèmes de fichiers parallèles fournissent une infrastructure de stockage partagé pour les applications. Dans le cas où plusieurs applications accèdent à cette infrastructure partagée simultanément, leur performance peut être altérée en raison d'interférence. Notre travail se concentre sur l'ordonnancement d'E/S comme un outil pour améliorer les performances en atténuant les effets de l'interférence. Le rôle de l'ordonnanceur d'E/S est de décider l'ordre dans lequel les requêtes des applications doivent être traitées par les serveurs du système de fichiers parallèle, en appliquant des optimisations pour ajuster le modèle d'accès résultant. Notre approche pour améliorer les résultats obtenues avec ordonnancement d'E/S est basée sur l'utilisation des informations des modèles d'accès et de la sensibilité à la séquentialité des accès des dispositifs de stockage. Nous avons obtenu des informations sur les applications à partir de traces de exécutions précédentes. Nous proposons aussi un outil pour le profilage des dispositifs de stockage qui utilise des régressions linéaires pour éviter l'exécution de benchmarks pour tous les paramètres. Nous avons appliqué l'apprentissage automatique pour pouvoir sélectionner automatiquement l'algorithme d'ordonnancement le plus approprié à chaque situation. Nos résultats montrent que les deux aspects - des applications et des dispositifs de stockage - sont essentiels pour prendre des bonnes décisions d'ordonnancement. En outre, malgré le fait qu'il n'y a pas d'algorithme capable d'améliorer les performances dans toutes les situations, nous avons montré que avec double adaptabilité il est possible d'appliquer des techniques d'ordonnancement d'E/S pour améliorer les performances, tout en évitant les situations où cela conduirait à sa diminution.

APPENDIX C EXTENDED ABSTRACT IN PORTUGUESE

C.1 Resumo - “Escalonamento de E/S Transversal para Sistemas de Arquivos Paralelos: das Aplicações aos Dispositivos

Esta tese se concentra no escalonamento de operações de entrada e saída (E/S) como uma solução para melhorar o desempenho de sistemas de arquivos paralelos, aleviando os efeitos da interferência. É usual que sistemas de computação de alto desempenho (HPC) ofereçam uma infraestrutura compartilhada de armazenamento para as aplicações. Nessa situação, em que múltiplas aplicações acessam o sistema de arquivos compartilhado de forma concorrente, os acessos das aplicações causarão interferência uns nos outros, comprometendo a eficácia de técnicas para otimização de E/S.

Uma avaliação extensiva de desempenho foi conduzida, abordando cinco algoritmos de escalonamento trabalhando nos servidores de dados de um sistema de arquivos paralelo. Foram executados experimentos em diferentes plataformas e sob diferentes padrões de acesso. Os resultados indicam que os resultados obtidos pelos escalonadores são afetados pelo padrão de acesso das aplicações, já que é importante que o ganho de desempenho provido por um algoritmo de escalonamento ultrapasse o seu sobrecusto. Ao mesmo tempo, os resultados do escalonamento são afetados pelas características do subsistema local de E/S - especialmente pelos dispositivos de armazenamento. Dispositivos diferentes apresentam variados níveis de sensibilidade à sequencialidade dos acessos e ao seu tamanho, afetando o quanto técnicas de escalonamento de E/S são capazes de aumentar o desempenho.

Por esses motivos, o principal objetivo desta tese é *prover escalonamento de E/S com dupla adaptabilidade*: às aplicações e aos dispositivos. Informações sobre o padrão de acesso das aplicações são obtidas através de arquivos de rastro, vindos de execuções anteriores. Aprendizado de máquina foi aplicado para construir um classificador capaz de identificar os aspectos espacialidade e tamanho de requisição dos padrões de acesso através de fluxos de requisições anteriores. Além disso, foi proposta uma técnica para obter eficientemente a razão entre acessos sequenciais e aleatórios para dispositivos de armazenamento, executando testes para apenas um subconjunto dos parâmetros e estimando os demais através de regressões lineares.

Essas informações sobre características de aplicações e dispositivos de armazenamento são usadas para decidir a melhor escolha em algoritmo de escalonamento através de uma árvore de decisão. A abordagem proposta aumenta o desempenho em até 75% sobre uma abordagem

que usa o mesmo algoritmo para todas as situações, sem adaptabilidade. Além disso, essa técnica melhora o desempenho para até 64% mais situações, e causa perdas de desempenho em até 89% menos situações. Os resultados obtidos evidenciam que ambos aspectos - aplicações e dispositivos de armazenamento - são essenciais para boas decisões de escalonamento. Adicionalmente, apesar do fato de não haver algoritmo de escalonamento capaz de prover ganhos de desempenho para todas as situações, esse trabalho mostra que através da dupla adaptabilidade é possível aplicar técnicas de escalonamento de E/S para melhorar o desempenho, evitando situações em que essas técnicas prejudicariam o desempenho.

C.2 Introdução

Aplicações científicas que executam em arquiteturas de alto desempenho (HPC) dependem de sistemas de arquivos paralelos (PFS) para obter bom desempenho mesmo quando lidando com grandes quantidades de dados. Uma vez que a velocidade de acesso a dados não cresceu no mesmo ritmo que o poder de processamento, diversas técnicas de otimização foram propostas para prover entrada e saída (E/S) escalável e de alto desempenho. Essas técnicas exploram o fato de o desempenho observado quando acessando um sistema de arquivos ser fortemente afetado pela maneira com a qual esse acesso é efetuado. Portanto, elas trabalham para ajustar o *padrão de acesso* das aplicações, priorizando características como localidade espacial e evitando situações conhecidas por resultarem em desempenho ruim, como a realização de um grande número de requisições pequenas e esparsas (BOITO; KASSICK; NAVAUX, 2011).

Essas otimizações que trabalham para melhorar o desempenho ajustando o padrão de acesso das aplicações geralmente o fazem no contexto de uma única aplicação. No entanto, as ar-

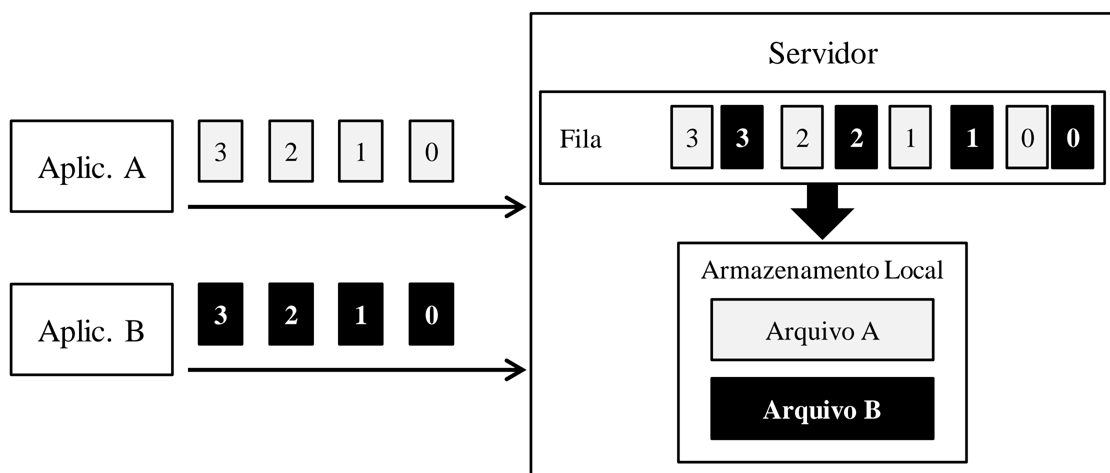


Figure C.1: Interferência no acesso concorrente ao sistema de arquivos compartilhado.

quiteturas atuais de HPC normalmente dedicam um conjunto de máquinas e dispositivos de armazenamento para a instalação de um sistema de arquivos paralelo. Essa infraestrutura de armazenamento é compartilhada por todas as aplicações que executam nessas arquiteturas. Quando múltiplas aplicações acessam concorrentemente o sistema de arquivos, o desempenho delas sofrerá com o efeito da *interferência*. Esse fenômeno é ilustrado na Figura C.1.

Esse trabalho foca no escalonamento de E/S como uma ferramenta para melhorar o desempenho, aleviando os efeitos da interferência. São considerados escalonadores que atuam no lado servidor, no contexto de requisições para servidores de sistemas de arquivos paralelos. A função desse escalonadores é decidir a ordem na qual as requisições devem ser processadas. Além disso, escalonadores podem aplicar otimizações, como agregação de requisições, para adaptar o padrão de acesso resultante, melhorando o desempenho.

O principal objetivo dessa tese é prover escalonamento de E/S para sistemas de arquivos paralelos. É seguida a hipótese de que, para se obter bons resultados, *escalonadores de E/S devem ser capazes de dupla adaptabilidade: aos padrões de acesso das aplicações e aos dispositivos de armazenamento*.

C.3 AGIOS: Uma Ferramenta para Escalonamento de E/S em Sistemas de Arquivos Paralelos

Escalonadores de E/S encontrados na literatura geralmente são específicos para um sistema de arquivos paralelo. Além disso, a maioria deles impõe configurações específicas, como um servidor de metadados centralizado. Essas características limitam a usabilidade de tais técnicas em novos contextos e a comparação entre elas.

Por esses motivos, foi desenvolvida uma ferramenta de escalonamento de E/S chamada AGIOS. Os principais objetivos que guiaram o seu desenvolvimento foram fazê-la genérica, não-invasiva, fácil de usar e oferecer múltiplas escolhas em algoritmos de escalonamento. A fim de evitar a formação de um gargalo, as instâncias AGIOS são independentes e não realizam decisões globais.

AGIOS oferece uma interface simples para os seus usuários que consiste em quatro passos: inicialização, inclusão de novas requisições, devolução das requisições ao usuário pelo escalonador e finalização. Essa interface é ilustrada na Figura C.2. O escalonador trabalha no contexto de arquivos ao invés de aplicações. Isso é feito dessa forma porque servidores de sistemas de arquivos paralelos geralmente não são capazes de determinar de que aplicação ou processo cada requisição veio, já que essa informação é perdida através da pilha de E/S.

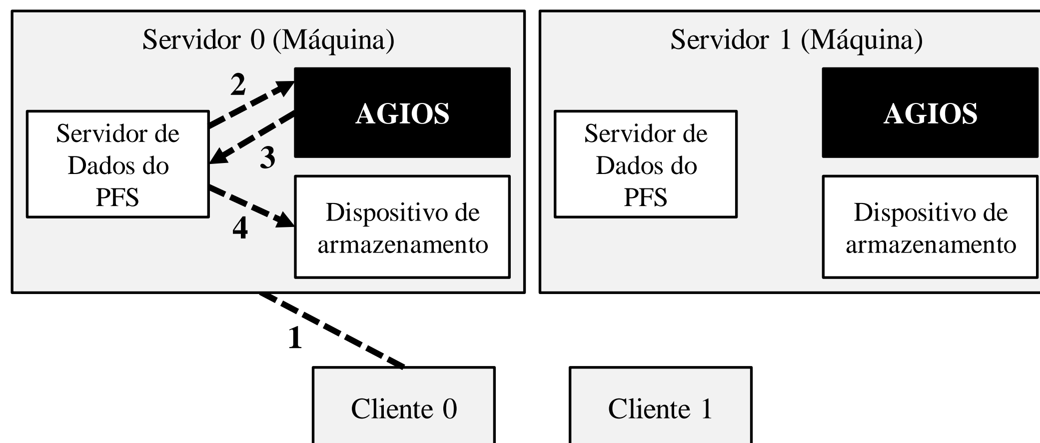


Figure C.2: Ilustração da interface entre AGIOS e um servidor de dados de um sistema de arquivos paralelo.

C.3.1 Algoritmos de Escalonamento

Foram implementadas cinco opções em algoritmos de escalonamento: aIOLi, MLF, SJF, TO e TO-agg. Esses algoritmos foram selecionados pela sua variedade, representando diferentes situações e complementando as características uns dos outros. A maioria deles trabalha em pelo menos uma das duas premissas comuns sobre desempenho de E/S: acessos sequenciais resultam em melhor desempenho que acessos aleatoriamente localizados nos arquivos, e é melhor realizar um menor número de grandes acessos do que um maior número de pequenos.

aIOLi, MLF e SJF trabalham para processar requisições em ordem de *offset* para gerar padrões de acesso sequenciais. Todos os algoritmos implementados, com exceção do TO, tentam agregar requisições. Desses, espera-se que o aIOLi realize melhores agregações, mas ao custo de um maior sobrecusto por causa da sua abordagem síncrona: esse algoritmo espera até que a requisição anterior tenha sido servida antes de selecionar uma nova, possivelmente dando tempo para novas requisições chegarem e formarem agregações. No entanto, essa abordagem síncrona aumenta o impacto do sobrecusto do algoritmo de escalonamento no tempo para processar requisições.

C.3.2 Avaliação de Desempenho

Foi conduzida uma avaliação de desempenho com a ferramenta AGIOS a fim de identificar situações onde essa técnica é capaz de prover melhorias de desempenho e de entender o que torna essas situações adequadas ao escalonamento. Além disso, intencionou-se identificar qual dos algoritmos implementados é o mais adequado para cada cenário.

Foi utilizado o *makespan* como métrica para essa avaliação, pois ele representa o tempo total para processar uma carga de trabalho do ponto de vista do sistema de arquivos. Portanto, no contexto desse trabalho, o objetivo do escalonamento é melhorar o desempenho global, mesmo que isso signifique piorar o desempenho de uma aplicação individualmente. A métrica não reflete justiça e tempo de resposta, pois esses aspectos não são tratados nessa tese.

A ferramenta foi usada nos servidores de dados de um sistema de arquivos paralelo chamado dNFSp (AVILA et al., 2004). Testes foram executados em quatro clusters do Grid'5000 (BOLZE et al., 2006), escolhidos pela sua variedade em dispositivos de armazenamento: SSDs, HDDs e *arrays RAID*. Além disso, esses dispositivos possuem diferentes razões entre acessos sequenciais e aleatórios.

Os testes foram desenvolvidos com a ferramenta *MPI-IO Test* a fim de cobrir variados padrões de acesso de aplicações. Foram variados os aspectos: espacialidade, número de arquivos acessados, número de aplicações concorrentes, número de processos por aplicação e tamanho das requisições.

Os resultados obtidos evidenciaram que o efeito positivo no desempenho provido pelos algoritmos de escalonamento precisa superar o seu sobrecusto para que os resultados sejam bons. Para testes pequenos (os padrões de acesso com arquivos compartilhados), testes de leituras são mais sensíveis a esse sobrecusto, pois o tempo de execução desses testes é menores. Por outro lado, os melhores resultados para os testes grandes (padrões de acesso com a abordagem de arquivos independentes por processo) foram obtidos para os testes de leitura. Acredita-se que isso acontece porque, em geral, os algoritmos de escalonamento foram capazes de agregar mais requisições em testes de leitura do que nos de escrita, gerando um maior efeito positivo no desempenho. Além disso, a razão entre acessar sequencialmente e aleatoriamente de todas as plataformas usadas nessa análise é maior para leituras do que para escritas, indicando que leituras são mais afetadas pela reorganização de requisições.

Alguns padrões de acesso (como o que gera requisições contíguas e pequenas) são menos propícios a ganhos de desempenho, pois eles proveem menos oportunidades para agregação. Essa diferença entre diferentes espacialidades e tamanhos de requisição é especialmente importante para os testes menores. Agregações de requisições contíguas realizam um papel importante nos resultados do escalonamento de E/S, como ilustrado pelos resultados obtidos para o cluster Edel (com SSDs). Apesar dos dispositivos de armazenamento dessa plataforma apresentarem as menores diferenças entre acessar sequencialmente e aleatoriamente, foram observados ganhos de desempenho causados por agregações.

O sobrecusto de escalonamento aumenta com o número de arquivos sendo acessado con-

correntemente. Por esse motivo, para que bons resultados sejam obtidos em situações onde um grande número de arquivos é acessado ao mesmo tempo, a carga de trabalho deve ser grande o bastante para prover oportunidades de melhorias.

Entre os algoritmos de escalonamento, aIOLi normalmente provê os melhores resultados. No entanto, esse algoritmo também causou os piores resultados em algumas situações, já que ele apresenta o maior sobrecusto entre os algoritmos testados. TO-agg e SJF geralmente representam boas alternativas, já que eles induzem menos sobrecusto. Entre todos os padrões de acesso e plataformas testados, todos os algoritmos estudados aparecem pelo menos uma vez como a melhor escolha para uma situação. Isso indica que não há algoritmo capaz de prover o melhor desempenho para todas as situações. A melhor escolha depende dos padrões de acesso e das plataformas.

Além das diferenças nos seus dispositivos de armazenamento, as capacidades dos nós também é um importante fator. O cluster Suno, que não possui a maior razão entre acessos sequenciais e aleatórios dentre as plataformas testadas, teve ganhos de desempenho em situações onde outros clusters não tiveram. Por ter mais memória RAM e poder de processamento nos seus nós, esse cluster foi menos afetado pelo sobrecusto dos algoritmos de escalonamento.

Os resultados obtidos formam uma base para o resto dessa tese. Eles evidenciaram que o desempenho obtido com algoritmos de escalonamento de E/S depende das características das aplicações e dos dispositivos de armazenamento.

C.4 Escalonamento de E/S Direcionado às Aplicações

No contexto dessa tese, decidiu-se pela obtenção de informações sobre as aplicações através de *arquivos de rastro*. Essa decisão foi feita com o objetivo de manter AGIOS genérica e fácil de usar. A maioria dos demais métodos para obter essa informação incluem mudanças nas bibliotecas de E/S, compiladores e aplicações, o que comprometeria a portabilidade da ferramenta. O arquivo de rastro é gerado pela própria ferramenta e mantida no seu dispositivo de armazenamento local, sem modificações nas aplicações ou nos sistemas de arquivos.

No arquivo de rastro, uma entrada de “nova requisição” guarda o identificador do arquivo, offset, tamanho e tempo de chegada da requisição. Rastros gerados durante a execução da mesma aplicação podem apresentar alguma variação entre os tempos de chegada para a mesma requisição. A fim de obter estimativas de tempo de chegada mais realistas, múltiplos arquivos de rastro podem ser combinados.

A Figura C.3 apresenta a arquitetura da ferramenta AGIOS. A geração de rastros é ati-

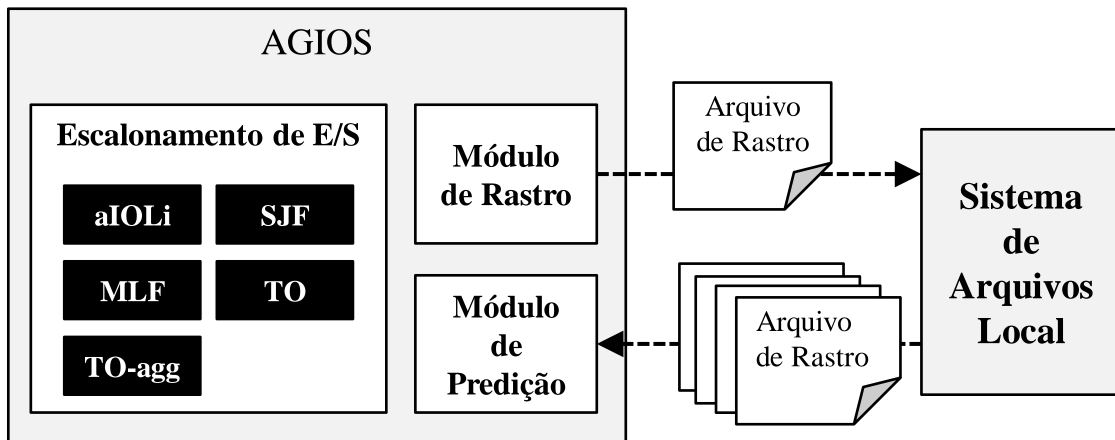


Figure C.3: Arquitetura da ferramenta AGIOS.

vada por um parâmetro de configuração, e pode ser usada com qualquer um dos algoritmos de escalonamento providos. O *Módulo de Predição* é responsável por obter informações de rastros e por prover essas informações aos algoritmos de escalonamento.

O Módulo de Predição é inicializado pelo escalonador no começo da sua execução se arquivos de rastro estiverem presentes. Uma *thread* de predição então lê os rastros e gera um conjunto de filas idênticas às usadas durante a execução, exceto que essas contêm futuras requisição "preditas". Essa inicialização pode também ser disparada durante a execução, provendo a habilidade de gerar arquivos de rastro durante um período inicial da execução e então começar a atividade do Módulo de Predição se o padrão de acesso rastreado for esperado novamente no futuro.

O conjunto de requisições preditas obtido pelo Módulo de Predição através dos arquivos de rastro provê uma janela maior para os algoritmos de escalonamento trabalharem. A informação obtida pode ser usada para realizar decisões melhores, levando a melhores resultados.

Para ilustrar esse potencial, foi proposta uma abordagem para predizer agregações de re–quisições e usar essas predições durante a execução. Baseado nessas predições, o escalonador pode decidir se esperar um pequeno intervalo de tempo poderia beneficiar o desempenho, permitindo melhores agregações. Os resultados obtidos mostram ganhos de desempenho com essa abordagem de $\approx 27\%$ em média, comparando com uma versão do escalonador que não usa essa informação. Esses resultados demonstram o potencial de usar informações sobre os padrões de acesso das aplicações para melhorar as decisões de escalonamento.

O sobrecusto induzido pela geração de arquivos de rastro depende do tamanho de *buffer* usado, e uma análise mais profunda é necessária para determinar o tamanho ótimo para esse *buffer*. Além disso, já que o tamanho dos arquivos de rastro cresce com o número de requisições, cargas de trabalho maiores geram rastros maiores. Ler e combinar múltiplos rastros no Módulo

de Predição pode levar bastante tempo. Apesar disso, as aplicações teriam que pagar esse custo extra apenas uma vez e se beneficiar das predições diversas vezes. Portanto, a abordagem proposta nessa tese é adequada para aplicações com padrões de acesso que se repetem, tais como simulações que trabalham em *timesteps* (previsão do tempo, simulações sísmicas, etc) e aplicações que efetuam *checkpoints* periódicos. Aplicações que são executadas frequentemente com uma carga de trabalho similar também poderiam se beneficiar desse trabalho.

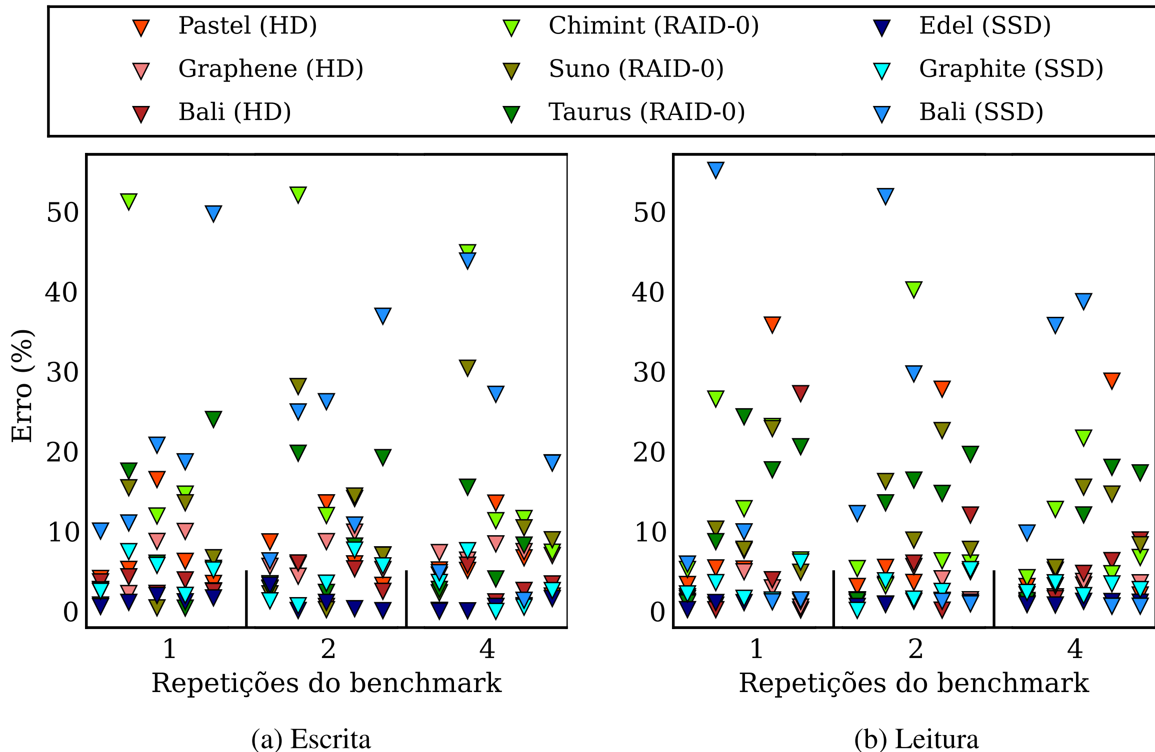
Foram propostas duas métricas - *distância média entre requisições consecutivas* e *diferença média de tempos de chegada nos stripes* - que são bons indicadores dos aspectos de padrão de acesso espacialidade e tamanho de requisições. Através de uma ferramenta de aprendizado de máquina, foi construída uma árvore de decisão capaz de classificar padrões de acesso em quatro classes: "contíguo pequeno", "contíguo grande", "não-contíguo pequeno" e "não-contíguo grande". Essa árvore de decisão apresentou uma taxa de acertos de $\approx 80\%$ e poderia ser usada mesmo durante a execução, usando informações dos acessos recentes.

C.5 Perfilamento de Dispositivos de Armazenamento

Os resultados obtidos com os algoritmos de escalonamento de E/S indicam que o desempenho obtido é afetado pelo subsistema de armazenamento. Dependendo das características dos dispositivos, o sobrecusto do escalonamento pode não ser compensado pelos seus efeitos positivos no desempenho. Portanto, é importante que escalonadores de E/S se adaptem ao comportamento de desempenho dos dispositivos de armazenamento.

Além disso, não é possível simplesmente classificar otimizações dizendo que elas são adequadas apenas para HDDs ou SSDs. Abordagens que intencionam a geração de acessos contíguos (originalmente pensadas para HDDs), por exemplo, podem melhorar o desempenho quando usadas em SSDs que são sensíveis à sequencialidade dos acessos. Adicionalmente, em qualquer dispositivo, a melhoria no desempenho causada pelo uso de uma otimização específica pode não compensar o seu sobrecusto. Portanto, essas otimizações poderiam ser classificadas de acordo com a razão entre a banda obtida para acessos sequenciais e aleatórios que os dispositivos devem apresentar a fim de se beneficiar delas.

No contexto dessa tese, foi proposta uma ferramenta para perfilamento de dispositivos de armazenamento em relação à sequencialidade dos acessos chamada SeRRa. Ela quantifica a diferença entre acessar arquivos sequencialmente e aleatoriamente para um dado dispositivo. Para obter essa informação de forma rápida, os *benchmarks* são executados em um pequeno subconjunto dos parâmetros, e os demais são estimados através de modelos lineares.



(a) Escrita (b) Leitura

Figure C.4: Erro absoluto induzido pelo uso da SeRRa para obter a razão entre acessar sequencialmente ou aleatoriamente.

Foram apresentados resultados cobrindo um grande espaço de parâmetros em nove diferentes dispositivos de armazenamento, incluindo HDDs, SSDs e *arrays* RAID. A avaliação mostrou que a SeRRa é capaz de prover perfilamentos com erros de aproximadamente 5% (mediana) enquanto levando apenas $1/168$ do tempo originalmente necessário.

C.6 Escalonamento de E/S com Dupla Adaptatividade

No contexto dessa tese, foi desenvolvida a ferramenta de escalonamento de E/S AGIOS. Essa ferramenta provê cinco algoritmos de escalonamento: aIOLi, MLF, SJF, TO e TO-agg. Através de uma avaliação que incluiu diferentes padrões de acesso em quatro clusters, foram observados ganhos de desempenho de até 68% obtidos com escalonamento de E/S. Por outro lado, os mesmos algoritmos podem diminuir o desempenho em até 278% para algumas situações.

Os resultados evidenciaram que obter o melhor desempenho depende da seleção do algoritmo de escalonamento mais adequado para cada situação, e que o melhor algoritmo depende das características das aplicações e dos dispositivos. Essa seção apresenta a abordagem proposta para prover escalonamento de E/S com dupla adaptabilidade: a aplicações e a dispositivos. Foi

empregada a técnica de aprendizado de máquina para gerar árvores de decisão.

A informação provida pela ferramenta SeRRa - a razão entre as bandas obtidas ao acessar sequencialmente e aleatoriamente para leitura e escrita com diferentes tamanhos de requisição - é usada para representar as características das plataformas.

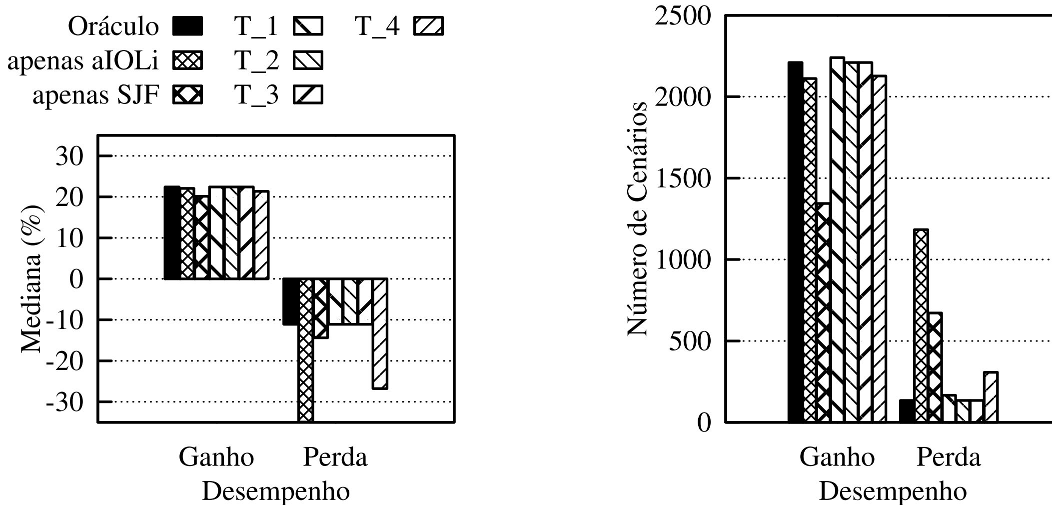
No contexto dessa tese, AGIOS é usada para escalonar requisições dos servidores de dados de sistemas de arquivos paralelos. Por esse motivo, apesar dos padrões de acesso usados na avaliação cobrirem diversos aspectos como número de processos e se processos compartilham ou não os seus arquivos, esses aspectos não podem ser usados para a construção da árvore de decisão. Isso acontece pois o servidor vê um fluxo de requisições a arquivos, e o resto da informação é perdida ao longo da pilha de E/S. Por exemplo, do ponto de vista dos servidores, não há diferença entre um padrão de acesso em que uma única aplicação acessa dois arquivos, e outra em que duas aplicações acessam um arquivo cada.

No entanto, como evidenciado pelos resultados obtidos, o número de arquivos acessados afeta o sobrecusto do escalonamento. Além disso, a quantidade de dados por arquivo indica quando o efeito positivo dos algoritmos no desempenho é capaz de compensar o sobrecusto deles. Portanto, como informação sobre os padrões de acesso é obtida de arquivos de rastro, o número de arquivos sendo acessados e a quantidade de dados acessados por arquivos foram incluídos na representação do padrão de acesso.

Adicionalmente, conforme discutido anteriormente, do fluxo de requisições aos servidores, é possível classificar padrões de acesso nos aspectos espacialidade (contíguo ou não-contíguo) e tamanho de requisições (pequeno ou grande). Essa classificação é feita para cada arquivo acessado usando informações dos rastros. Seria possível obter essa informação durante a execução usando os acessos recentes do escalonador, mas a abordagem proposta ainda dependeria dos rastros se a quantidade de dados acessada por arquivo fosse necessária.

Foi usada uma ferramenta de aprendizado de máquina para criar árvores de decisão para fazer a seleção de um algoritmo de escalonamento de acordo com parâmetros que caracterizam aplicações e dispositivos de armazenamento. Foram geradas cinco árvores usando diferentes atributos. A primeira árvore, T_1, é a única a usar a quantidade de memória RAM dos nós como um parâmetro. T_2 usa todos os demais parâmetros, T_3 não inclui a quantidade de dados acessados por arquivo, e T_4 não usa a quantidade de dados acessados por arquivo nem o número de arquivos sendo acessados. As decisões de algoritmo de escalonamento para todas as situações testadas foram obtidas da avaliação de desempenho previamente discutida.

As árvores de seleção de algoritmo de escalonamento foram comparadas com uma solução oráculo, que sempre dá a resposta correta de acordo com os resultados da avaliação de desem-



(a) Diferenças medianas de desempenho (b) Situações com diferenças de desempenho

Figure C.5: Resultados de desempenho para as árvores de seleção de algoritmo de escalonamento usando espacialidade e tamanho de requisições detectados.

penho, e com duas soluções que sempre usam o mesmo algoritmo de escalonamento, aIOLi ou SJF. T_1, T_2 e T_3 obtiveram resultados simulares, muito próximos ao oráculo. Isso indica que a quantidade de memória RAM dos nós e a quantidade de dados acessados por arquivo não são atributos importantes para identificar as situações dos experimentos, já que T_3 obteve bons resultados sem esses atributos. T_4 teve os piores resultados, mas usar essa árvore ainda seria significantemente melhor do que uma solução em que o mesmo algoritmo de escalonamento é sempre usado.

O impacto da detecção de espacialidade e tamanho de requisições dos arquivos de rastro foi avaliado. Para todas as árvores de decisão avaliadas, o impacto do erro na detecção desses parâmetros nos resultados foi pequeno: menos de 5% das situações passaram a ter quedas de desempenho. As diferenças nos ganhos e quedas de desempenho medianas foi sempre menor do que 1%.

Por causa dos seus bons resultados, T_3 foi incluída no Módulo de Predição da AGIOS para seleção de algoritmo de escalonamento. A maior vantagem de não usar a quantidade de dados acessados por arquivo como um parâmetro da árvore de decisão é que a abordagem proposta não é dependente do uso de arquivos de rastro. Todos os parâmetros usados para descrever as características das aplicações - espacialidade, tamanho de requisições, operação e número de arquivos - são possíveis de serem obtidos em tempo de execução. Portanto, seria possível adaptar a aplicações e dispositivos mesmo quando arquivos de rastro não estão disponíveis. Além disso, o Módulo de Predição poderia periodicamente re-selecionar o algoritmo de escalonamento baseado em acessos recentes. Explorar essa possibilidade é assunto de trabalhos futuros.

C.7 Conclusão

Para prover E/S de alto desempenho, o escalonador de E/S deve realizar otimizações nos padrões de acesso. No entanto, o escalonador trabalha numa pequena janela de requisições e não possui informações detalhadas sobre as aplicações, já que essas informações são perdidas ao longo da pilha de E/S. Ao mesmo tempo, os resultados dessas otimizações dependem das características do subsistema de E/S, como o comportamento do desempenho dos dispositivos de armazenamento. Nesse contexto, o principal objetivo dessa tese é prover escalonamento de E/S com *dupla adaptabilidade*: aos padrões de acesso das aplicações e às características dos dispositivos de armazenamento.

Foi desenvolvida a ferramenta AGIOS para escalonamento de E/S, podendo ser usada por qualquer serviço que trate requisições no nível de arquivos. AGIOS foi desenvolvida com a intenção de ser genérica, não-invasiva e fácil de usar. Além disso, a ferramenta oferece cinco opções de algoritmo de escalonamento, cobrindo diferentes vantagens e desvantagens: aIOLi, MLF, SJF, TO, e TO-agg. Demonstrou-se o uso de AGIOS nos servidores de dados de um sistema de arquivos paralelo, e foi apresentada uma extensiva avaliação de desempenho em quatro clusters do Grid'5000, representando diferentes alternativas em armazenamento: HDDs, SSDs e *arrays* RAID. Os resultados indicam que os padrões de acesso das aplicações e as características dos dispositivos de armazenamento afetam a eficácia do escalonamento de E/S, e não há algoritmo de escalonamento capaz de melhorar o desempenho para todas as situações.

Para obter informações sobre aplicações, o Módulo de Predição usa arquivos de rastro de execuções anteriores. Esses rastros são gerados pela AGIOS, sem modificações no sistema de arquivos ou nas aplicações. O potencial de uso dessas informações foi ilustrado pelo uso delas para predizer agregações. Um algoritmo de escalonamento foi modificado para usar essas predições para guiar as suas decisões. Essa abordagem levou a ganhos de desempenho de 27% em média comparado à versão desse algoritmo que não usa tais informações.

Para detectar o padrão de acesso de arquivos de rastro, foram apresentadas duas métricas - distância média entre requisições consecutivas e diferença média de tempos de chegada nos *stripes* - que são bons indicadores de espacialidade e tamanho de requisições. Através de uma ferramenta de aprendizado de máquina, foi criada uma árvore de decisão capaz de classificar padrões de acesso em relação a esses aspectos, dando a resposta correta em 80% dos casos. Essa detecção de padrão de acesso pode ser feita usando acessos de rastros ou durante a execução, olhando para acessos recentes.

A fim de obter informações sobre dispositivos de armazenamento, foi proposta uma fer-

ramenta chamada SeRRa. Essa ferramenta reporta, para um dispositivo de armazenamento, a razão entre acessá-lo sequencialmente e aleatoriamente para escritas e leituras de diferentes tamanhos. Como perfilamento de E/S é uma tarefa que pode demorar bastante tempo, SeRRa usa modelos lineares para prover resultados com uma boa acurácia o mais rápido possível. Com essa abordagem é possível obter um perfilamento de dispositivos de armazenamento em uma fração do tempo originalmente necessário, com erros de apenas 5%.

Aprendizado de máquina foi usado para construir árvores de decisão capazes de selecionar o algoritmo de escalonamento mais adequado para diferentes situações. Diversas árvores de decisão foram geradas usando diferentes parâmetros como entrada, incluindo informações sobre aplicações e dispositivos de armazenamento. Todas as árvores de decisão obtidas proporcionaram melhores resultados do que abordagens onde o mesmo algoritmo de escalonamento é sempre usado. Resultados próximos à solução oráculo foram obtidos por uma árvore de decisão que usa os seguintes aspectos para tomar uma decisão:

- operação: escrita ou leitura;
- número de arquivos sendo acessados concorrentemente;
- espacialidade: contíguo ou não-contíguo;
- tamanho de requisições: pequeno ou grande;
- razão entre a banda obtida por acessos sequenciais e aleatórios do dispositivo de armazenamento.

Uma vantagem dessa árvore de decisão é que todos os aspectos de padrão de acesso incluídos podem ser obtidos durante o tempo de execução, considerando-se uma janela de acessos recentes. Isso permitiria que a ferramenta tomasse boas decisões mesmo quando arquivos de rastro não estão disponíveis. Comparando com soluções onde apenas aIOLi ou SJF são usados, a árvore de seleção de algoritmo de escalonamento aumenta o desempenho em até 75% e 38%, respectivamente. Além disso, de todas as situações testadas, a abordagem proposta é capaz de aumentar o desempenho para até 64% *mais* situações e de prejudicar o desempenho de até 89% *menos* situações. Esses resultados indicam que informações sobre aplicações e dispositivos de armazenamento são essenciais para corretamente selecionar o melhor algoritmo de escalonamento de E/S para uma dada situação, reforçando a importância de prover escalonadores de E/S com *dupla adaptabilidade: a aplicações e dispositivos*.