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

JEAN LUCA BEZ

Dynamic Tuning and Reconfiguration of the I/O Forwarding Layer in HPC Platforms

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

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

External Coadvisor: Dr. Antonio Cortés Rosselló
Universitat Politècnica de Catalunya (UPC)
Barcelona Supercomputing Center (BSC)

Porto Alegre
April 2020

CIP — CATALOGING-IN-PUBLICATION

Bez, Jean Luca

Dynamic Tuning and Reconfiguration of the I/O Forwarding Layer in HPC Platforms / Jean Luca Bez. – Porto Alegre: PPGC da UFRGS, 2020.

136 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2020. Advisor: Philippe O. A. Navaux; External Coadvisor: Antonio Cortés Rosseló.

I. O. A. Navaux, Philippe. II. Cortés Rosseló, Antonio. III. Título.

“Agradeço todas as dificuldades que enfrentei.
Não fosse por elas, eu não teria saído do lugar.
As facilidades nos impedem de caminhar.
Mesmo as críticas nos auxiliam muito.”

— CHICO XAVIER

ACKNOWLEDGMENTS

I want to thank my advisor, Prof. Philippe Navaux, for the opportunities and invaluable guidance during the past four years. Likewise, I want to thank my co-advisor, Prof. Toni Cortes, for the valuable discussions regarding this thesis's ideas, for receiving me during my stay in Barcelona, and for the encouragement. I also want to express my gratitude towards Dr. Francieli Boito for guiding my initial steps in this research topic, for the reviews, and the brainstorming sessions that helped shaping this work. I also want to thank Dr. Ramon Nou and Dr. Alberto Miranda from Barcelona Supercomputing Center with whom I had the amazing oportunity to interact, learn, and grow personally and professionally during my six month stay in their laboratory. Finally, I want to thank Jay Lofstead, from Sandia Laboratory, for presenting new oportunities, welcoming to the Supercomputing community, and motivating me in my next steps.

I am incredibly grateful for my parents Osvaldo and Cleusa, and my brother Renan, who always encouraged, supported, and inspired me during this exciting new phase of my life. Thank you for motivating and believing in me! My sincere thanks to all my friends (near and far) and all those who played an important role in my life throughout this journey. Thank you for being there for me!

I would also like to express my gratitude to the Federal University of Rio Grande do Sul (UFRGS) and the Institute of Informatics (INF) for the opportunity of conducting my research within its walls and interacting with this fantastic scientific community.

This research was financed by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001. It has also received funding from the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), Brazil.

Experiments presented in this thesis were carried out using the Grid'5000 testbed[1], supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations. We thankfully acknowledge the computer resources, technical expertise and assistance provided by the Barcelona Supercomputing Center – Centro Nacional de Supercomputación. We thankfully acknowledge the National Laboratory for Scientific Computing (LNCC/MCTI, Brazil) for providing HPC resources of the SDumont supercomputer[2], which have contributed to the research results reported within this document.

[1] <https://www.grid5000.fr>
[2] <http://sdumont.lncc.br>

ABSTRACT

Input and output (I/O) operations are a bottleneck for an increasing number of applications in High-Performance Computing (HPC) platforms. Furthermore, it has the potential of critically impacting performance on the next generation of supercomputers. I/O optimization techniques can provide improvements for specific system configurations and application access patterns, but not for all of them. We call the access pattern the way an application performs its I/O operations. These techniques frequently rely on the precise tune of parameters, which commonly falls back to the users. In such large scale systems, we have an ever-changing application set running with distinct characteristics and demands. Hence, to improve performance successfully, it is essential to adapt the system to a changing workload dynamically. In this work, we seek to guide optimization and tuning strategies by identifying the application's I/O access pattern. We evaluate three machine learning techniques to detect such patterns at runtime automatically: decision trees, random forests, and neural networks. Using the detected pattern, we propose a tuning strategy that uses a reinforcement learning technique (contextual bandits) to make the system capable of learning the best parameter value to each observed access pattern during its execution. That eliminates the need for a complicated and time-consuming previous training phase. Finally, we argue in favor of a dynamic on-demand allocation of I/O nodes considering the application's I/O characteristics. We show that the forwarding layer's global deployment combined with the existing static allocation policy based solely on application size should instead be dynamic and consider the applications' access patterns to improve global performance. We presented a user-level I/O forwarding solution named GekkoFWD that does not require application modifications and allows a dynamic remapping of forwarding resources to compute nodes. We proposed a novel I/O forwarding allocation policy based on the Multiple-Choice Knapsack Problem. We demonstrate our dynamic MCKP policy's applicability to arbitrate I/O nodes through extensive evaluation and experimentation. We show it could transparently improve global I/O bandwidth by up to $23\times$ compared to the existing static policy.

Keywords: High Performance I/O. Parallel I/O. I/O Forwarding. I/O Scheduling. Dynamic Tuning. Dynamic Reconfiguration.

Adaptação e Reconfiguração Dinâmicas da
Camada de Encaminhamento de E/S em Plataformas HPC

RESUMO

As operações de entrada e saída (E/S) são um gargalo para um número crescente de aplicativos em plataformas de Processamento de Alto Desempenho (PAD). Além disso, tem o potencial de impactar criticamente o desempenho da próxima geração de supercomputadores. As técnicas de otimização de E/S podem melhorar o desempenho para configurações específicas do sistema e para alguns padrões de acesso das aplicações, mas não para todos eles. Chamamos o padrão de acesso a maneira como uma aplicação executa suas operações de E/S. Essas técnicas freqüentemente dependem do ajuste preciso dos parâmetros, que normalmente recai sobre os usuários. Em tais sistemas de grande escala, temos um conjunto de aplicações em execução com características e demandas distintas. Portanto, para melhorar o desempenho global, é essencial adaptar o sistema a uma carga de trabalho que está sempre em constante mudança de forma dinâmica. Neste trabalho, buscamos guiar estratégias de otimização e reconfiguração identificando o padrão de acesso de E/S da aplicação. Avaliamos três técnicas de aprendizado de máquina para automaticamente detectar esses padrões em tempo de execução: árvores de decisão, florestas aleatórias e redes neurais. Utilizando o padrão detectado, propomos uma estratégia de reconfiguração que utiliza uma técnica de aprendizado por reforço (bandidos contextuais) para tornar o sistema capaz de aprender o melhor valor de parâmetro para cada padrão de acesso observado durante sua execução. Isso elimina a necessidade de uma fase anterior de treinamento complicada e demorada. Finalmente, argumentamos a favor de uma alocação dinâmica e sob demanda de nós de E/S considerando as características de E/S da aplicação. Mostramos que a aplicação global da camada de encaminhamento combinada com a política de alocação estática existente baseada exclusivamente no tamanho do aplicativo deve ser dinâmica e considerar os padrões de acesso dos aplicativos para melhorar o desempenho global. Apresentamos uma solução de encaminhamento de E/S em nível de usuário chamada GekkoFWD que não requer modificações nas aplicações e permite um remapeamento dinâmico de recursos de encaminhamento para nós de computação. Propusemos uma nova política de alocação de encaminhamento baseada no problema da mochila de múltipla escolha. Demonstramos a aplicabilidade de nossa política dinâmica MCKP para arbitrar nós de E/S por meio de extensa avaliação e experimentação. Mos-

tramos que tal solução pode melhorar, de forma transparente, a largura de banda de E/S global em até $23\times$ em comparação com a política estática existente.

Palavras-chave: E/S de Alto Desempenho, E/S Paralela, Encaminhamento de E/S, Adaptação Dinâmica, Reconfiguração Dinâmcia.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AGIOS	Application-Guided I/O Scheduler
ADIOS	Adaptable I/O System
API	Application Programming Interface
BSC	Barcelona Supercomputing Center
CIOD	Console I/O Daemon
CMB	Cosmic Microwave Background
CPU	Central Processing Unit
DNN	Deep Neural Network
DOE	United States Department of Energy
DT	Decision Tree
DVS	Data Virtualization Service
FIFO	First In, First Out
FORGE	I/O Forwarding Emulator
FPP	File-per-process access pattern
FUSE	Filesystem in Userspace
G5K	Grid'5000 Test Platform
GPFS	General Parallel File System
HBRR	Handle-Based Round-Robin
HDD	Hard-disk Drive
HDF5	Hierarchical Data Format, Version 5
HPC	High Performance Computing
I/O	Input/Output
IBM	International Business Machines
IOD	I/O Daemon

IOFSL	I/O Forwarding Scalability Layer
ION	I/O Node
IOR	Interleaved Or Random Benchmark
IP	Internet Protocol
JSON	JavaScript Object Notation
LDN	Lustre Network Driver
LNCC	Laboratório Nacional de Computação Científica
LNet	Lustre Networking
MCKP	Multiple-Choice Knapsack Problem
MIT	Massachusetts Institute of Technology
ML	Machine Learning
MN4	Mare Nostrum IV Supercomputer
MPI	Message Passing Interface
MPI-IO	Message Passing Interface – Input/Output
NASA	National Aeronautics and Space Administration
NetCDF	Network Common Data Form
NFS	Network File System
NN	Neural Network
NPB	NASA Advanced Supercomputing Division Parallel Benchmarks
NRS	Network Request Scheduler
OBRR	Object-Based Round Robin
OLAM	Ocean-Land-Atmosphere Model
PFS	Parallel File System
POSIX	Portable Operating System Interface
PVFS	Parallel Virtual File System
RAID	Redundant Array of Independent Disks

RAM	Random Access Memory
RDMA	Remote Direct Memory Access
RL	Reinforcement Learning
RPC	Remote Procedure Call
SC	Shared-file contiguous access pattern
SCSI	Small Computer System Interface
SS	Shared-file 1D-strided access pattern
SSD	Solid State Drive
TCP	Transmission Control Protocol
TWINS	Time WINdows Scheduler
UCB	Upper Confidence Bound Algorithm
UFRGS	Federal University of Rio Grande do Sul
UFS	Unix File System
VFS	Virtual File System
ZOID	ZeptoOS I/O Daemon

CONTENTS

1 INTRODUCTION

Scientific applications impose ever-growing performance requirements on the High-Performance Computing (HPC) field. Furthermore, increasingly heterogeneous workloads are entering HPC installations, from Big Data to Machine Learning applications, making the systems more complex than ever. These requirements justify the continuous upgrades and deployment of new large-scale platforms. As these systems' complexity tends to grow, so does the number of parameters and factors that may directly or indirectly affect performance. "The Opportunities and Challenges of Exascale Computing" report presented by the U.S Department of Energy (DOE) stated that the Exascale problem is more than just a matter of scale. Application behavior and performance will be determined by a complex interplay of the program code, processor, memory, interconnection network, and input/output (I/O) operation (DOE, 2010). Therefore, achieving performance at scale requires an optimized orchestration of those components and a complete system view to understand the root causes of inefficiencies.

Regarding storage systems management, the DEO "Storage Systems and I/O: Organizing, Storing, and Accessing Data for Scientific Discovery" report (ROSS et al., 2018) acknowledges the increasing popularity of machine learning as a tool within the systems community, which introduces new possibilities in the application of statistical models to storage systems management. They mention efforts that could contribute to more effective use of current and emerging storage hardware. Among those, we highlight storage systems capable of responding to cyclic workload demands, policy-driven toolsets capable of managing resource sharing, and new capabilities for enabling rapid data reorganization. The research presented here aligns with these emerging trends.

As I/O is a bottleneck for an increasing number of applications, it has the potential of critically impacting application performance on the next generation of supercomputers. While HPC clusters typically rely on a shared storage infrastructure powered by a Parallel File System (PFS) such as Lustre (SUN, 2007), GPFS (SCHMUCK; HASKIN, 2002), or Panasas (WELCH et al., 2008), the increasing I/O demands of applications from fundamentally distinct domains stress this shared infrastructure. As systems grow in the number of compute nodes to accommodate larger applications and more concurrent jobs, the PFS is not able to keep providing performance due to increasing contention and interference (XU et al., 2014; Kougkas et al., 2016; Yildiz et al., 2016; YU et al., 2018; YANG et al., 2019a).

To mitigate this issue, the I/O forwarding technique (ALMÁSI et al., 2003) seeks to reduce the number of nodes concurrently accessing the PFS servers by creating an additional layer between the compute nodes and the data servers. Thus, rather than applications accessing the PFS directly, the I/O forwarding technique defines a set of *I/O nodes* that are responsible for receiving I/O requests from applications and forwarding them to the PFS in a controlled manner, allowing the application of optimization techniques such as request scheduling, aggregation, and compression. Moreover, its presence on an HPC system is transparent to applications and file system agnostic. Due to these benefits, the forwarding technique is applied by Top 500 machines[1] as detailed by Table 1.1.

Table 1.1: Some of the TOP 500 machines that implement I/O forwarding.

Rank	**Supercomputer**	**Compute Nodes**	**I/O Nodes**
3	Sunway TaihuLight (YANG et al., 2019b)	40,960	240
4	Tianhe-2A (XU et al., 2014)	16,000	256
6	Piz Daint (GORINI; CHESI; PONTI, 2017)	6,751	54
7	Trinity (VIGIL, 2015)	19,420	576
13	Sequoia (PRABHAT; KOZIOL, 2014)	98,304	768

Source: TOP 500 November 2019, and Ji et al. (2019).

I/O optimization techniques (including but not limited to the I/O forwarding layer) typically provide improvements for specific system configurations and application access patterns, but not for all of them. We call the *access pattern* the way the application perform I/O operations: number of accessed files, spatiality (contiguous, 1D-strided, etc), and request size. Moreover, such optimizations often rely on the correct choice of parameters (for example, the size of the buffer for MPI-IO collective operations). However, the responsibility of making this choice often lies with the user. The techniques fail to provide improvements for all patterns because they are designed to explore specific characteristics of systems and workloads (MCLAY et al., 2014). Boito et al. (2016) and Bez et al. (2017) demonstrate that for request scheduling at different levels of the I/O stack. Therefore, considering that in such large scale systems we have an ever-changing application set running, with distinct characteristics and demands, to improve performance successfully, **it is essential to dynamically adapt the system to a changing workload**. Thus, we remove the tuning responsibility from the users by making the system capable of adapting itself to deliver the best possible performance.

We propose a novel approach to adapt the I/O forwarding layer to the current I/O

[1]November 2019 TOP500:<https://www.top500.org/lists/2019/06/>.

workload. In our proposal, we periodically observe access pattern metrics collected by the I/O nodes. A reinforcement learning technique, contextual bandits (SUTTON; BARTO, 2017), is used so that the system can learn the best choice to each access pattern at runtime. After observing a pattern enough times, the acquired knowledge will be used to improve its performance during the whole life of the system, i.e., for years. Furthermore, as the learning mechanism continues to update its knowledge, it can adapt to changes in the system. The novelty of our proposal consists in using k-armed bandits and access pattern detection to automatically and transparently tune parameters that impact I/O performance at the forwarding layer during runtime.

By making the system capable of learning at runtime (using Reinforcement Learning), we eliminate the need for a previous training step (though that is required for the access pattern detection step, it could be done asynchronously) without adding a high overhead. That is essential as designing and executing a training set to represent the diverse set of applications that will run in a supercomputer, and the interactions between concurrent applications (over the shared I/O infrastructure), is difficult, error-prone (as we might not accurately cover all patterns and their interactions), and time-consuming.

Furthermore, on such machines, the forwarding layer is traditionally physically deployed on special nodes, and the mapping between clients and I/O nodes is static. Consequently, a subset of compute nodes will only forward requests to a single fixed I/O node, which ends up forcing applications to use I/O forwarding with a statically predefined number of I/O nodes, even if that decision might not be in the best interest for a given workload. Though this setup seeks to distribute I/O nodes between compute nodes evenly, it lacks the flexibility to adjust to applications' I/O demands, and it can even cause the misallocation of forwarding resources and an I/O load imbalance, as demonstrated by Yu et al. (2017c) on the Sunway TaihuLight[2] supercomputer and Bez et al. (2020) on the MareNostrum 4[3] supercomputer.

To showcase this issue, Figure 1.1 depicts the achieved bandwidth computed from the execution time (makespan), measured at client-side, when multiple clients issue their requests following an access pattern (represented by the scenarios A to H) and taking into account the number of available I/O nodes (0, 1, 2, 4, and 8) on the MareNostrum 4 supercomputer at Barcelona Supercomputing Center (BSC)[4]. Each experiment was repeated at least 5 times, in random order, and spanning different days and periods of the

[2]<https://www.top500.org/system/178764>
[3]<https://www.top500.org/system/179067>
[4]<https://www.bsc.es>

Table 1.2: Details of the access patterns shown in Figure 1.1.

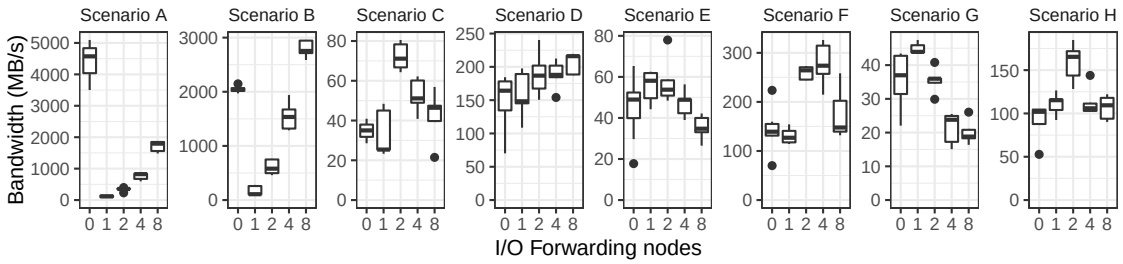
Scenario	Nodes	Processes	File Layout	Request Spatiality	Request (KB)
A	32	1536	File-per-process	Contiguous	1024
B	32	1536	File-per-process	Contiguous	128
C	32	1536	Shared	Contiguous	1024
D	16	192	Shared	1D-strided	128
E	8	192	Shared	1D-strided	1024
F	16	384	Shared	Contiguous	128
G	32	384	Shared	1D-strided	512
H	8	384	Shared	Contiguous	4096

Source: Author

day. Table 5.1 describes each depicted pattern. It is possible to notice there does not seem to be a simple rule regarding the number of I/O nodes, which is to be expected given the complexity of factors that can influence I/O performance. Furthermore, some patterns seem to benefit the most from having access to more I/O nodes than others. For instance, in scenario A, not using forwarding is the best choice. On the other hand, for scenario B, eight I/O nodes are more suited. For the scenario C, only two would suffice. Consequently, a static mapping of I/O nodes to compute nodes without considering an application's workload does not always result in the best performance (BEZ et al., 2020). Hence, the need for appropriate allocation policies that take into account these issues to maximize *globally-perceived* I/O performance.

We argue in favor of a dynamic, on-demand allocation of I/O nodes that considers an application's workload characteristics. Accordingly, given a set of applications ready to run and a fixed number of forwarding resources, solving their allocation problem would consist in determining how many I/O nodes each of them should receive to maximize the aggregated global bandwidth. The allocation policy should be invoked before new applications start to run, and when the set of running jobs has changed.

Figure 1.1: I/O bandwidth of distinct write access patterns with a varying number of I/O nodes in the MareNostrum 4 supercomputer. The y-axis is not the same in the plots.



Source: Author

However, due to the static nature of traditional I/O forwarding infrastructures and the inherent limitations involved in running a production supercomputer, it is not always possible for system administrators to explore different I/O allocation strategies without negatively impacting user jobs. Thus, a research/exploration solution is required that allows both I/O researchers and system administrators to obtain an overview of the benefits or drawbacks of the different access patterns under different I/O forwarding configurations. For such a solution to be useful, it should be portable, allow existing applications to run without modifications to their source-code and, if possible, run as a user-level service to simplify deployment. As we advocate for dynamic allocation, such a solution should also allow changing the number of I/O nodes assigned to an application during its execution without disrupting it.

1.1 Objectives and Contributions

The main objective of our research is to **dynamically tune the I/O forwarding layer in HPC platforms to improve global performance**. We explore two fronts that use the application's access patterns as guideline to make decisions: tuning scheduling parameters at the forwarding layer and arbitrating I/O nodes between the set of running applications. Considering these goals, our contributions are the following:

- We investigate and demonstrate how Machine Learning (ML) techniques (Decision Tree, Random Forests, and Neural Network) can aid in automatically detecting the most common I/O access patterns of HPC applications;

- We demonstrate the applicability of detection strategies at the I/O forwarding layer when tuning an I/O scheduler's parameter in which the accurate detection of the access pattern is paramount to achieve performance;

- We propose a novel approach to adapt the I/O forwarding layer to the current I/O workload by combining Reinforcement Learning (RL) and access pattern detection to automatically and transparently tune I/O-related parameters at runtime;

- We propose a lightweight I/O forwarding explorer tool named FORGE to gather performance metrics and aid in understanding the impact of I/O forwarding in a system;

- We propose an I/O forwarding allocation policy based on the Multiple-Choice Knapsack Problem (MCKP) to optimally arbitrate I/O nodes between applications;
- We evaluate different I/O forwarding allocation policies and demonstrate that a dynamic allocation approach can improve overall global bandwidth and system usage, while efficiently using the available I/O nodes;
- We present an I/O forwarding service called GekkoFWD that acts as an on-demand forwarding layer and implements the MCKP allocation policy. GekkoFWD builds on top of a user-level ad-hoc file system, enriching it to allow exploring different forwarding deployments. It does not require application modifications and it is simple to run in production.

1.2 Document Organization

The remaining chapters of this document are organized as follows:

- **Chapter 2** presents the background on the topics of this thesis covering parallel file systems, the I/O forwarding layer, access patterns detection, request scheduling and resource allocation;
- **Chapter 3** explores Machine Learning (ML) techniques to detect the most common I/O access patterns of HPC applications and demonstrates the applicability of such strategies at the I/O forwarding layer;
- **Chapter 4** details our Reinforcement Learning (RL) approach to automatically tune I/O parameters at runtime based on the application's access pattern, and presents an evaluation when applying the technique to optimize TWINS I/O scheduling algorithm window size parameter;
- **Chapter 5** focuses on the arbitration of available I/O forwarding resources between running applications by considering their characteristics and exploring distinct allocation policies (both static and dynamic);
- **Chapter 6** discusses related work;
- **Chapter 7** presents concluding remarks and discusses future research perspectives.

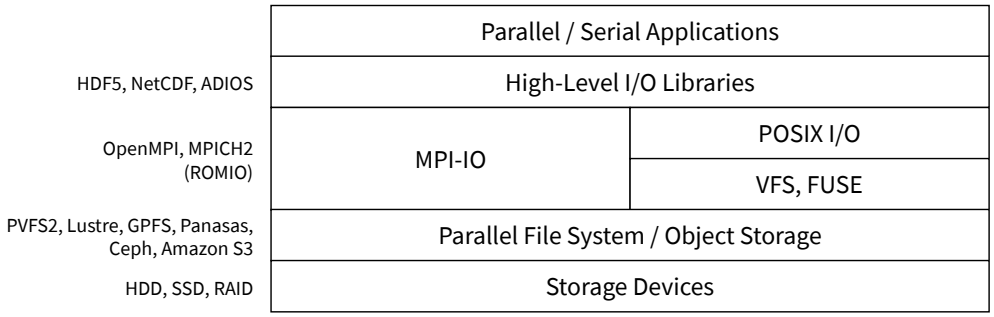
2 BACKGROUND

In this chapter, we introduce some concepts that serve as a foundation for this work. An overview of the I/O stack for High-Performance Computing (HPC) is presented alongside some I/O optimizations techniques. Furthermore, this chapter discusses the concept of access patterns and resource allocation in HPC platforms.

2.1 Parallel I/O for High Performance Computing

HPC applications often span multiple compute nodes that have a simplified kernel to avoid possible interference and usually do not have local storage devices. The latest trend changes this paradigm, where some supercomputers have local SSD devices used as a Burst Buffers (LIU et al., 2012). Nonetheless, even in these scenarios, applications rely on a Parallel File System (PFS) to provide a globally shared storage infrastructure where access to these remote files is transparent for the applications. Due to the increasing number of compute nodes required to access the shared storage, contention and interference becomes a limiting factor for achieving performance.

Figure 2.1: Standard parallel I/O stack of HPC platforms.



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

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

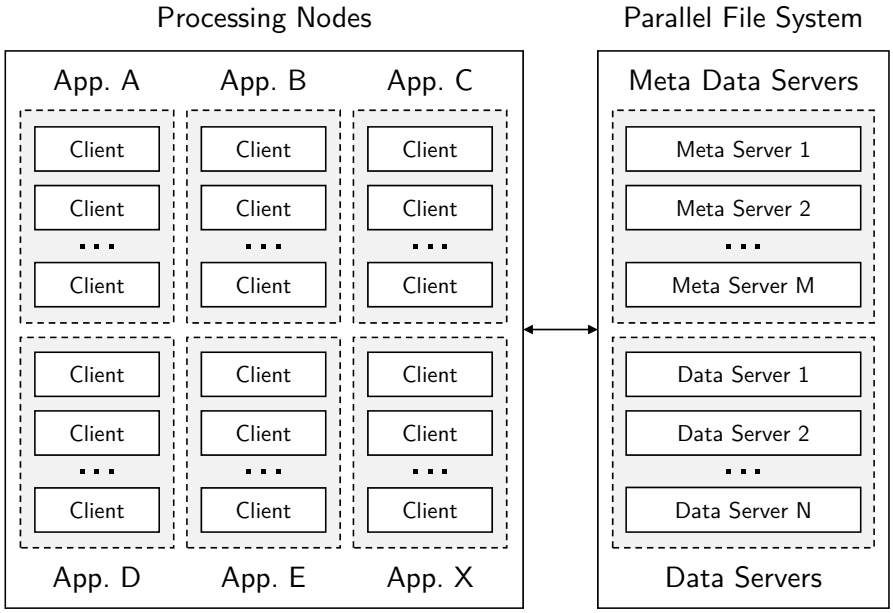
age devices such as Hard Disk Drives (HDDs), Solid State Drives (SSDs), or Redundant Array of Independent Drives (RAID).

2.1.1 Parallel File Systems

Large-scale systems, such as supercomputers, rely on Parallel File Systems (PFS) to provide a persistent shared storage infrastructure. These systems are deployed over a set of dedicated machines and offer a shared namespace, so applications can access remote files as if they were stored on their local file system. Furthermore, to achieve high-performance, they harness parallelism by breaking files and distributing data into fixed-size chunks across multiple storage nodes through an operation called *data striping* (STENDER et al., 2008).

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

Figure 2.2: Major components of a parallel file system.



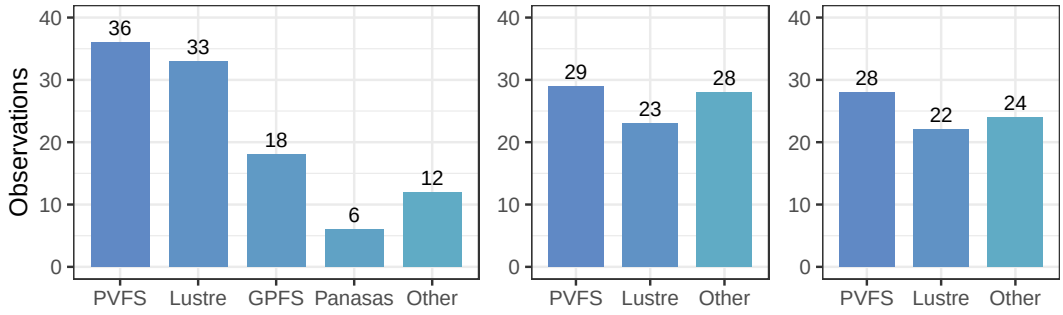
Source: Author

All basic file system operations involve metadata access. For instance, when a client wants to read or write a file, it must first obtain the layout information and permissions from the metadata server. Hence the scalability of those accesses has a direct impact on the overall system performance (REN et al., 2014). An alternative to improve performance is to allow clients to cache metadata information. However, in this situation, a cache coherence policy must be in place. Some parallel file systems such as PVFS2 (LATHAM et al., 2004) and OrangeFS (DELL, 2012) distribute metadata among multiple servers, while others such as Lustre (SUN, 2007) maintain a single centralized metadata storage. Centralizing metadata operations may become a bottleneck for applications that work with a large number of small files.

The major parallel file systems in use are Lustre, IBM's Spectrum Scale (former General Parallel File System — GPFS) (SCHMUCK; HASKIN, 2002), Panasas (WELCH et al., 2008), and the Parallel Virtual File System (PVFS) or its new branch, OrangeFS. From the ten most powerful supercomputers in the world according to the November 2019 edition of the Top500 list[1], four use Lustre, two are based on Lustre, and four uses GPFS/IBM Spectrum Scale.

As an overview of the most used parallel file systems in scientific research, we analyzed several papers in a pre-defined five-year window for a survey on parallel I/O (BOITO et al., 2018). We have made a selection of widely known, leading quality conferences and journals. This window covered publications between 2010 and 2014. We went through all proceedings and issues inside the time window ($5,159$ publications) to identify relevant work by looking at title and abstract. During this process, 120 papers were pre-selected for further analysis (2.3%). After reading the articles and answering a set of questions to determine its relevance for the service, 86 articles remained (1.7%).

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



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

Source: Author

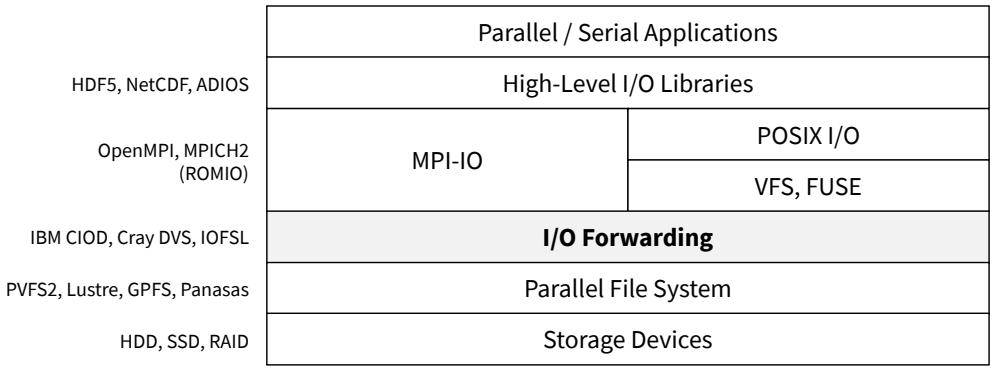
[1] <https://www.top500.org/lists/2019/11/>

We summarize the results in Figure 2.3 grouped by PFS and research purposes. We can see that PVFS is one of the most used systems for parallel I/O research (Figure 2.3(a)). Furthermore, studies that focused on proposing I/O optimization techniques (Figure 2.3(b)) were also carried out in PVFS. Accordingly, for this research, we have used PVFS' new branch named OrangeFS, Lustre, and GPFS depending on the machine. Nonetheless, as we focus on the I/O nodes, the PFS choice does not restrict our solution for other PFS.

2.1.2 The Forwarding Layer

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

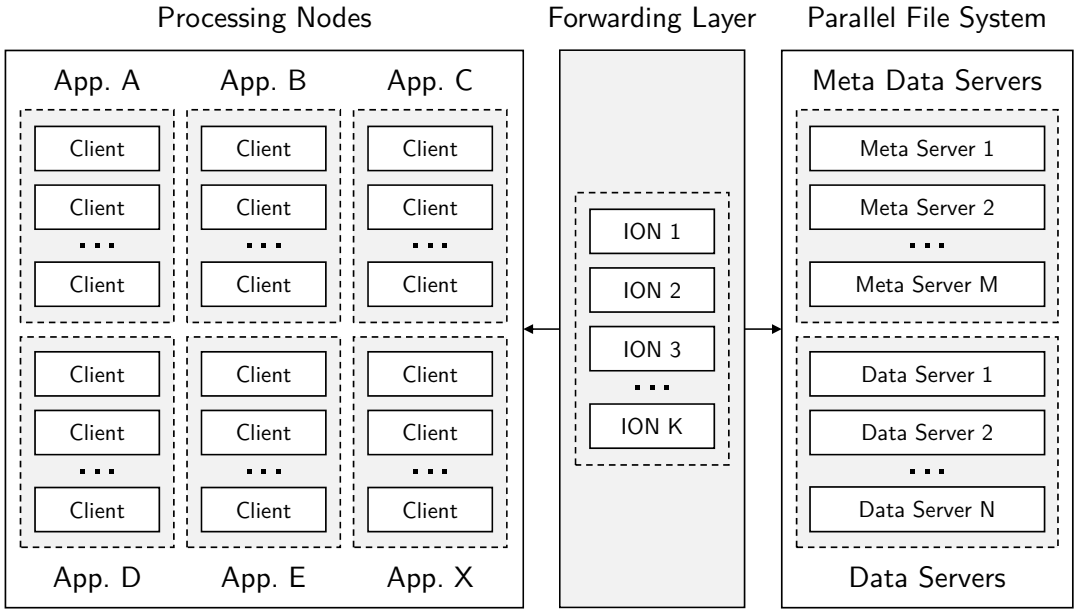
Figure 2.4: Forwarding layer in the HPC I/O stack.



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

With the I/O forwarding layer, all requests are forwarded to dedicated nodes, known as I/O nodes. Typically, the number of I/O nodes is larger than the number of file system servers and smaller than the number of compute nodes. In this scenario, the latter may be powered with only a very simplified local I/O stack to avoid its interference on performance, also known as operating system noise (VISHWANATH et al., 2010). When an I/O node (ION) receives requests, it redirects them to the back-end PFS, as depicted by Figure 2.5. This strategy reduces the number of clients concurrently accessing the file system and can potentially reduce the file system traffic by aggregating and reordering I/O requests (OHTA et al., 2010).

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



Source: Author

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

- **Cray Data Virtualization Service** (DVS) is a distributed network service that projects local file systems resident on I/O nodes or remote file servers to compute and service nodes within the Cray system. Projecting is making a file system available on nodes where it does not physically reside. DVS-specific configuration settings enable clients (compute nodes) to access a file system projected by DVS servers. Thus, Cray DVS, while not a file system, represents a software layer that provides scalable transport for file system services. Cray DVS uses the Linux-supplied virtual file system (VFS) interface to process file system access operations. It allows DVS to project any POSIX-compliant file system. Cray has extensively tested DVS with NFS and General Parallel File System (now Spectrum Scale). DVS has two primary modes of use: serial and parallel. In serial mode, one DVS node projects a file system to multiple compute node clients. In parallel mode, multiple DVS nodes – in configurations that vary in purpose, layout, and performance –

project a file system to multiple compute node clients.

- **Lustre Networking (LNet)** is a custom networking API that provides the communication infrastructure to handle metadata and file I/O data for the Lustre file system servers and clients. LNet has support for many commonly-used network types, such as InfiniBand and IP networks, and allows simultaneous availability across multiple network types with routing between them. Remote direct memory access (RDMA) is allowed when supported by underlying networks (LND). It is possible to configure LNet to act as a router, forwarding communications and I/O requests between all local networks.

- The **IOFSL framework** (ALI et al., 2009) implements the I/O forwarding technique as an attempt to bridge the increasing performance scalability gap between computing and I/O components (LIU et al., 2013). IOFSL ships I/O calls from the applications, running on computing nodes, to dedicated I/O nodes. The latter will then transparently perform operations on behalf of the computing nodes. This framework uses the stateless ZOIDFS I/O protocol and API from the ZOID forwarding infrastructure (ISKRA et al., 2008), and the Buffered Message Interface (BMI) network abstraction layer for high-performance parallel I/O. BMI provides request forwarding over multiple parallel file systems (PVFS, Lustre, UFS, and PanFS) and interconnection networks (TCP/IP, InfiniBand, and Myrinet). The framework's software stack consists of two main components: a ZOIDFS client library running on the computing nodes and I/O forwarding daemon (IOD) running on I/O nodes. The client library forwards I/O requests from the compute node kernel to the IOD which performs I/O on behalf of the compute nodes.

Considerable research (VISHWANATH et al., 2010; OHTA et al., 2010; VISHWANATH et al., 2011; ISAILA et al., 2011; Yu et al., 2017a) has been focused on improving the I/O forwarding layer performance. Some of them (VISHWANATH et al., 2010; VISHWANATH et al., 2011; ISAILA et al., 2011) studied the I/O subsystem of an IBM Blue Gene/P supercomputer. In this architecture, the data staging mechanism initially applied multiple threads per I/O node (one per processing node), without any coordination among them. Vishwanath et al. (2010) identified some contention-related bottlenecks associated with this design. They improved performance by allowing asynchronous operations in the I/O nodes and by including a simple FIFO scheduler to coordinate access from multiple threads. This scheduler alone provides improvements of up to 38%. They

also optimized data movement between layers through a topology-aware approach. Isaila et al. (2011) proposed a two-level pre-fetching scheme for this architecture.

Similarly, Ohta et al. (2010) improve IOFSL's performance by using I/O scheduling. They implement two algorithms: a simple FIFO and a *quantum* based algorithm called *Handle-Based Round-Robin* (HBRR). The latter based on an algorithm successfully applied to parallel file systems' data servers (LEBRE et al., 2006; QIAN et al., 2009; BOITO et al., 2013; BOITO et al., 2016), that aims at reordering and aggregating requests to improve the performance of the applications by modifying their access pattern.

In recent efforts, Yu et al. (2017b) proposes to employ file striping on the I/O forwarding layer to handle heavy workloads. Their approach seeks to balance the workload at this layer by recruiting multiple idle I/O nodes. Furthermore, they coordinate file striping on the forwarding layer and storage system layers to minimize contention (when I/O nodes compete for file lock or data stripes) and exploit better aggregate bandwidth.

2.2 I/O Optimizations

Numerous factors may interfere with applications' I/O performance, especially at large scale. Performance degradation can occur because of network problems, software bugs, slow disk, or contention when accessing the shared storage system (LARREA et al., 2015). Hence, significant research effort is put into optimizing, at different layers of the I/O stack, how the applications perform their I/O. Modifying the file system servers or clients, the applications, or using libraries and APIs are distinct ways of achieving the same goal: adjusting the way applications issue their I/O requests and avoiding situations that are known to degrade performance.

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

2.2.1 Application's Access Patterns

Applications issue their I/O requests to the PFS servers in diverse ways, depending on how they were designed and coded. Several characteristics, such as the number of issued requests, the requests' sizes, and their spatial location in the file, compose what

we call the application's *access pattern*. The pattern has a direct impact on performance, hence a lot of research effort is put into optimizing data access (LOFSTEAD et al., 2011; HE et al., 2013; YIN et al., 2013; KUO et al., 2014).

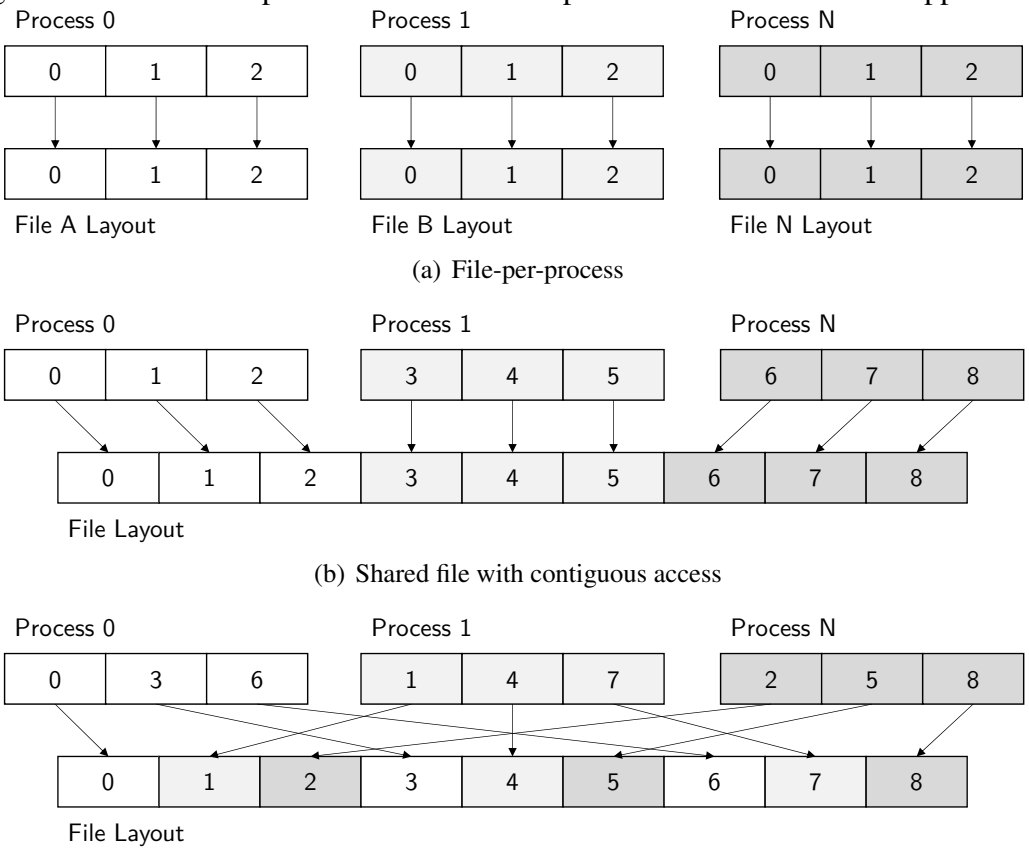
The access pattern can be classified in *local*, *global*, or *system-wide*. The local pattern describes an application's behavior in the context of a process or task, whereas the global pattern does it at the application level, considering all processes and tasks (YIN et al., 2013). On the other hand, the system-wide one describes the patterns of the mixed concurrent applications when using the shared storage infrastructure or the I/O nodes. The local access pattern information is usually employed to identify and apply optimizations on the client-side, while the global access pattern is more suitable in the context of the forwarding layer or file system servers since it has an overview of the application's data accesses. The system-wide one can also be used in the data servers and forwarding layer to coordinate accesses and optimize I/O performance system-wide.

Although there is no globally accepted convention to describe what elements define an access pattern, several researchers of the parallel I/O field examined some common factors or parameters. In this research, we consider the following key aspects to describe the application's data access pattern: the number of files, the spatial locality within the file, the size of accesses, and the type I/O operation.

Regarding the number of files, we consider two well-known scenarios that depict how most of the HPC scientific applications perform I/O. In the first one, each process of an application issues its operations in its individual file (*file-per-process*), as pictured by Figure 2.6(a). In the second scenario, all the processes share a common file (*shared file*). Furthermore, the spatial locality parameter specifies if the access to a shared file is sequential, i.e. each process accesses contiguous chunks of the file (Figure 2.6(b)) or 1D-strided, i.e. each process accesses portions with a fixed-size gap between them (Figure 2.6(c)). It is valid to notice that when each application accesses its own file, the presence of a shared file system is not always required. Yet, even when this pattern is present, the data is often accessed by other processes or different applications for post-processing or visualization, requiring the existence of shared storage space. Additionally, when the compute nodes do not have storage devices attached to them, the PFS is a commonly used alternative, independently of the type of access.

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

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



(a) File-per-process

(b) Shared file with contiguous access

(c) Shared file with 1D-strided access

Source: Author

the network latency, which dominates the cost of processing the request. Moreover, in the case of PFS, they fail to perform well for small requests, especially random requests (He; Wang; Sun, 2016). Besides leading to poor I/O parallelism among multiple servers, the usage of HDD as storage media on data servers is notoriously slow for random data access due to the mechanical nature of disk head movements.

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

2.2.2 Request Aggregation and Reordering

The performance of contiguous data access usually is higher than that of non-contiguous ones (YIN et al., 2013) for both Hard Disk Drives (HDD) and Solid-State Disks (SSD). The work by Zimmer, Gupta and Larrea (2016) points out that small and

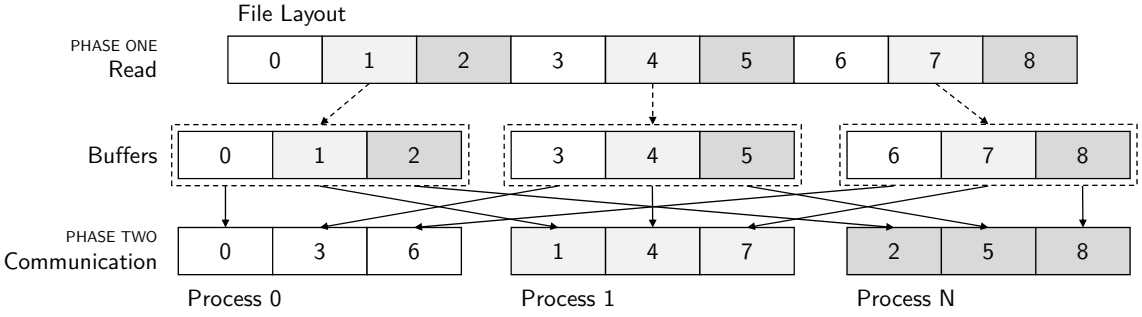
random I/O request patterns negatively impact the file system performance. Consequently, applications benefit from continguously accessing a file and issuing fewer requests to the file system, reducing the high I/O latency.

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

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

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

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



Source: Author

The optimizations cited so far require the applications to modify its source code.

The next section describes additional efforts in transparently producing better access patterns to the PFS without further changes to the applications.

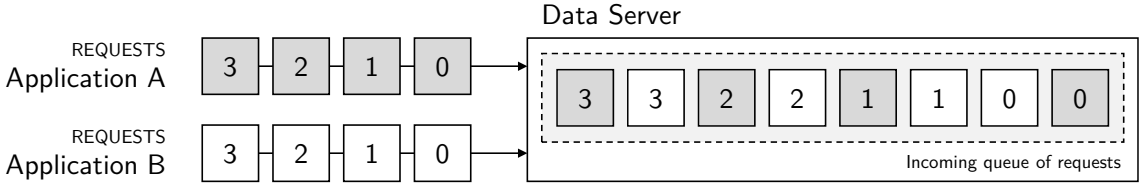
2.2.3 Request Scheduling

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

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

The I/O scheduler will then reorder, aggregate, and determine when to process each request. These scheduling techniques, applied at some layer of the I/O stack (clients, I/O nodes, or data servers), decide *where* and *when* requests must be served. Literature covers different schedulers that range from a simple First-Come, First-Served (FIFO) (OHTA et al., 2010) to more complex ones such as aIOLi (LEBRE et al., 2006), Object-

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



Source: Author

Based Round Robin (OBRR) Network Request Scheduler (NRS) (QIAN et al., 2009), or the ones available in the Application-Guided I/O Scheduling (AGIOS) (BOITO et al., 2016) library. There have also been approaches that target coordination, such as the IOrchestrator (ZHANG; DAVIS; JIANG, 2010). Others achieve coordination by using time windows, for instance, the one proposed by Song et al. (2011b) or the Server Time Windows (TWINS) (BEZ et al., 2017) scheduler. Finally, some seek to adapt to the access pattern using pattern matching solutions such as the Fast Dynamic Time Warping (FastDTW) (BOITO et al., 2019).

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

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

Boito et al. (2016) analyzed different I/O scheduling algorithms at the PFS data servers. These algorithms decide the order in which requests to each data server must be processed. They do not focus on cross-application interference, but on adjusting access patterns to obtain the best performance of the underlying I/O system. In many cases, this involves generating offset ordered requests or aggregating a large number of small requests into a smaller number of larger requests.

Although many schedulers were created for the data servers, only a few were

designed or tested in the forwarding layer. Applying such a technique in this layer of the I/O stack allows working with the global access pattern of the applications and even coordinate access between concurrent ones. Moreover, it is complementary to using them in other levels, i.e., this technique could be simultaneously applied in the clients, in the I/O nodes, and in the parallel file system data servers.

2.3 I/O Tuning

In recent years, some researchers (BEHZAD et al., 2013; MCLAY et al., 2014; ISAILA; CARRETERO; ROSS, 2016; LI et al., 2017; Agarwal et al., 2019) are focusing on automatically configuring the HPC I/O stack so that the applications could obtain the desired performance with minimal setup effort as if they and the system were manually tuned based on their characteristics. However, several factors and parameters may affect the I/O performance: the application's workload, access pattern, interference, and even system deployment. Moreover, no single optimization and configuration is suited for every possible scenario. Therefore, considerable research is being conducted to understand these factors and their interplay to propose techniques and mechanisms seeking to improve I/O performance while keeping the simplicity for scientists who run those applications on a daily basis.

Isaila et al. (2014) argue that the current uncoordinated development model of independently applying optimizations at each level of the I/O stack will not scale to the new levels of concurrency, storage hierarchy, and capacity expected of from the exascale systems. Furthermore, they also point out that only a limited number of work has focused on topology awareness for improving the I/O stack's performance and scalability. Building on those ideas, they propose CLARISSE, a middleware for data-staging coordination and control on large-scale HPC platforms (ISAILA; CARRETERO; ROSS, 2016). This framework decouples the policy, control, and data layers of the I/O stack to simplify the global coordination of data staging. Despite bringing significant performance benefits for collective operations, other common scenarios were not yet considered. For instance, the forwarding layer or the usage of burst buffers on other layers were left out of the equation. Furthermore, the impact of different I/O request schedulers on performance and the interference in the multi-application scenario is still an open issue.

Behzad et al. (2013) propose an auto-tuning system for optimizing the I/O performance of HDF5 applications. By intercepting HDF5 calls at runtime, they evaluate

and tune a set of parameters located at different levels of the I/O stack using genetic algorithms to evolve the parameters to the ones that present a better combination. Only a subset of all the possible parameters for all the layers of the I/O stack was selected. The stripe count and size (for Lustre), locking and large blocks (for GPFS), the number of collective buffering nodes and collective buffer size (for MPI-IO), and alignment and chunk size (for HDF5) were part of the equation.

McLay et al. (2014) propose a model for understanding collective parallel MPI write operations on the Lustre file system and a library to optimize parallel write performance. The authors point out that the default stripe count is rarely the right choice for parallel I/O, and its performance depends on a balance between the number of stripes and the actual number of collective writes. Authors argue that performance tuning is all about the stripe count because parallel write performance depends heavily on an appropriate choice of this parameter. Furthermore, if informed to the MPI stack, it can assign (collective) writers to achieve near-optimal performance. They propose the T3PIO library to manage this tuning automatically. However, additional scenarios covering collective read operations, independent writes, and more complex access patterns were not studied. Furthermore, resource contention and interference on the multi-application scenario are not taken into account when adjusting the parameters.

Agarwal et al. (2019) propose two auto-tuning models, based on active learning, to recommend a set of parameter values for MPI-IO hints and Lustre configuration for an application on a given system. They employ Bayesian optimization to find the parameter values. Though their approach still requires training, due to their separation of real-application execution and I/O prediction model, training time is reduced compared to other methods. A model-less deep reinforcement learning-based unsupervised parameter tuning system driven by a deep neural network (DNN) called CAPED was proposed by Li et al. (2017). They seek to address the slow and costly current practice of numerous tweak-benchmark cycles used to tune parameters. Their approach takes periodic measurements of a target computer system's state and trains a DNN that uses Q-learning to suggest changes to its current parameter values. They demonstrate that CAPES can find optimal values for the congestion window size and I/O rate limit of a distributed storage system in a noisy environment.

2.4 Summary

Researchers are tackling the automatic tuning problem of the HPC I/O stack on distinct fronts, some considering parameters from the client and library side, others on the file system servers. However, not all representative scenarios and access patterns are considered in the existing evaluations. Furthermore, the impact of the deployment and configuration of the forwarding layer is not yet adequately addressed. We believe this investigation is valid and necessary due to the widespread adoption of this layer and its potential in affecting the I/O performance of HPC applications. Based on these observations, the next chapters will detail our proposal.

3 ACCESS PATTERN DETECTION AT RUNTIME

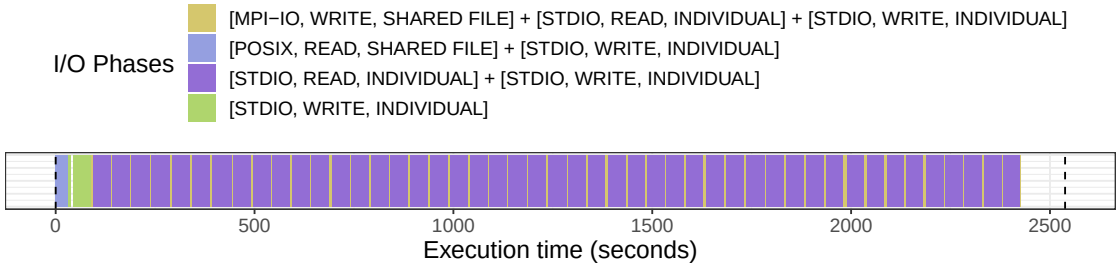
Optimization techniques that seek to improve I/O performance by modifying the application I/O access patterns often cover aggregations, request reordering, collective operations, and I/O request scheduling (KUMAR et al., 2013; WANG et al., 2014; CONGIU et al., 2016; TESSIER; VISHWANATH; JEANNOT, 2017; BEZ et al., 2017; Bağbaba, 2020; Liu; Wu; Xu, 2020). In this work, an *access pattern* refers to how an application performs its I/O operations, covering aspects such as the number of accessed files, spatiality (contiguous, 1D-strided, etc), and request size. These optimization techniques can be applied at different layers of the I/O stack, e.g., at the compute nodes, at the forwarding level, or the PFS. In general, optimization techniques typically provide improvements for particular system configurations and access patterns, but not for all of them. Furthermore, they often rely on the right selection of parameters, as demonstrated for request scheduling at different levels (BOITO et al., 2016; BEZ et al., 2017). Therefore, achieving the best results proposed by an optimization technique often relies on correctly applying them to the proper workload, and configuring it accordingly. The main issue in practice is that the workload keeps changing as new applications start and finish their I/O phases. Therefore, it becomes of paramount importance for systems that seek to auto-tune their parameters to correctly detect the access patterns, at runtime, to make decisions.

To motivate the need and benefits for adaptation, Figure 3.1 illustrates the executions of two applications as reported by Darshan (CARNS et al., 2011) traces. The x-axis represents the execution time, different colors represent different access patterns, and the boxes identify I/O phases. We use the concept of I/O phases to identify intervals where I/O operations are made using an access pattern. We infer this information from coarse-grained aggregated logs. That means that I/O operations are not necessarily happening throughout those entire periods, but we can be sure that those patterns characterize the I/O operations that are occurring in those phases. The duration of each phase is defined by the interval between the first and the last I/O operations from a sequence of operations with the same access pattern.

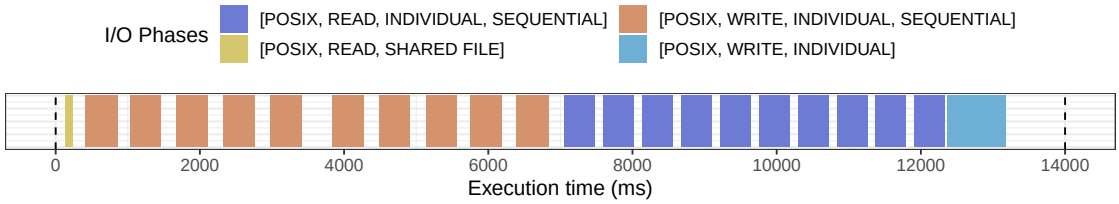
Figure 3.1(a) illustrates the input and output behavior the Ocean-Land-Atmosphere Model (OLAM) (WALKO; AVISSAR, 2008), executed in the Santos Dumont[1] supercomputer, at the National Laboratory for Scientific Computation (LNCC), in Brazil. We selected a job that used 240 processes and ran for 2433 seconds, of which 221 seconds

[1] <https://www.top500.org/system/178568>

Figure 3.1: I/O phases of two real-world executions of applications in HPC systems, as inferred from Darshan logs. For each execution, we depict only the top four access patterns (in accumulated duration).



(a) The I/O phases of one execution of the OLAM application at the Santos Dumont Supercomputer (LNCC).



(b) The anonymously identified application `2201660091` (job `15335183665324813784`) running in the Intrepid supercomputer at Argonne National Laboratory (ANL).

Source: Author

were spent on I/O operations. OLAM processes read time-dependent input files, write per-process logs (purple), and periodically write to a shared-file with MPI-IO (yellow). Though the intervals cover nearly the entire execution time of the application, it does not means that the application is issuing I/O requests non-stop. It only demonstrates that for a given (stable) period of time, we can be sure that if any I/O is happening, it will be characterized as described.

The second application, depicted by Figure 3.1(b), is anonymously identified as `2201660091` (JOB `15335183665324813784`) from the Argonne Leadership Computing Facility I/O Data Repository (CARNS, 2013). This repository contains anonymized traces from the Intrepid supercomputer. That particular execution was randomly chosen from those who spent at least 30% of their time on I/O. We can see the application performs sequential writes to individual files in roughly the first half of the execution, and then it moves on to perform sequential reads from individual files.

The examples illustrate that HPC applications tend to present a consistent I/O behavior, with a few access patterns being repeated multiple times over an extensive period (CARNS et al., 2009; DORIER et al., 2014; GAINARU et al., 2015; LIU et al., 2016; HU et al., 2016). Therefore, one way of adapting the stateless I/O system would be to observe the current access pattern over some time and combine it with a tuning strategy.

Nonetheless, runtime detection techniques should pose minimum overhead. They should also be capable of performing its detection as fast as possible to allow tuning mechanisms and optimizations methods to act on the information. That would enable the system to benefit from good choices quickly and promptly adapt once the observed I/O behavior changes. In this scenario, we could consider applying machine learning techniques. Although training phases could be expensive, once the model has learned its parameters, inferences on previously unseen data, at runtime, are fast.

In this chapter, we demonstrate how machine learning techniques can aid in automatically detecting the I/O access pattern of HPC applications at runtime. We have selected three methods with distinct characteristics: (I) a decision support tool that uses a tree-like model of decisions; (II) an ensemble learning method; and (III) a wide-spread "black-box" approach. Thus, we investigate decision trees, random forests, and neural networks to classify metrics collected at runtime into common access patterns often used by the HPC I/O community to evaluate new I/O optimizations. Furthermore, to demonstrate its applicability, we assessed these detection strategies in a case study in which the accurate detection of the access pattern is paramount to tune an I/O scheduler's parameter at the I/O forwarding layer.

3.1 Workload and Metrics

Modifying the I/O forwarding layer or the file system configuration in production machines to evaluate new mechanisms or optimization techniques is often not allowed. Such actions could disrupt services or even harm an application's performance at scale. Furthermore, to detect the I/O access patterns of HPC workloads, we require an initial dataset of metrics. Such datasets with all the necessary parameters to do so are not often made available from production machines sites.

Therefore, we deployed an I/O stack with forwarding infrastructure in clusters from the Grid'5000 (BOLZE et al., 2006) experimental testbed. That gives us the flexibility to evaluate our proposal and demonstrate its feasibility to be later applied to large-scale machines. We collected metrics on each I/O node every second throughout the execution of multiple benchmarks and configurations (details in Section 3.1.1). Since the patterns have a fixed duration, the number of observations is *not* the same for each benchmark, as it depends on the execution time. These metrics comprise a data set of over one million observations.

3.1.1 Experimental Methodology

We used two clusters from the Nancy site: four PFS servers in Grimoire; 32 clients and multiple (1, 2, 4, and 8) forwarding nodes in separated Grisou nodes. Nodes from both clusters have similar characteristics. Each one has two 8-core Intel Xeon E5-2630 v3, 128GB of RAM, and a 558GB hard disks. A 10Gbps Ethernet network interconnect the nodes and the two clusters. For our evaluation, both clusters were exclusively reserved during the experiments to minimize interference.

PVFS version 2.8.2 was used with default 64KB stripe size and striping through all four servers. Data servers perform writes directly to their disks, bypassing caches, to ensure the scale of tests would be enough to see access pattern impact on performance. Clients are equally distributed among the I/O nodes, that communicate directly with the file system through the IOFSL dispatcher. The IOFSL daemon was executed with all its default parameters.

To cover the most common I/O access pattern of HPC applications, we used the MPI-IO Test benchmark tool (Los Alamos National Laboratory, 2008), to issue requests using the MPI-IO library. We varied the number of processes (128, 256, or 512), operation (read or write), file layout (shared-file or file-per-process), spatiality (contiguous or 1D-strided), and request sizes (32 or 256KB — smaller than the stripe size or large enough so that all servers are accessed). For each experiment, a total of 4GB of data was accessed. We also deployed a different number of intermediate I/O nodes (1, 2, 4, or 8). In total, 144 different scenarios (we do not test the file-per-process 1D-strided combinations as this access pattern is not usual) were considered. The full set was executed in random order to minimize unforeseen impacts. The number of observations for each pattern representing different benchmark parameters is detailed in Table 3.1.

Table 3.1: Representativity of the access patterns in the dataset.

Observations			
Read	29,929	*vs.* 100,406	**Write**
Shared-file	72,802	*vs.* 57,533	**File-per-process**
Contiguous	94,997	*vs.* 35,338	**1D-strided**

Source: Author

3.2 Access Pattern Detection

In this section, we detail three approaches we consider to detect the I/O access pattern of HPC applications at runtime. We explore different machine learning techniques to identify the access pattern based on the system's metrics collected at runtime. We classify the access patterns by file layout and spatiality into three classes. File layout relates to the number of files, i.e., if all processes use a single shared file or if each process writes and reads to its file. Spatiality expresses if requests issued by the application are contiguous or follow a strided pattern.

The three distinct classes cover I/O patterns common among scientific applications and used by several benchmarks such as MPI-IO Test and IOR[2] to test I/O optimizations. Furthermore, they group situations that, in our experience, exhibit similar behavior. The three classes are:

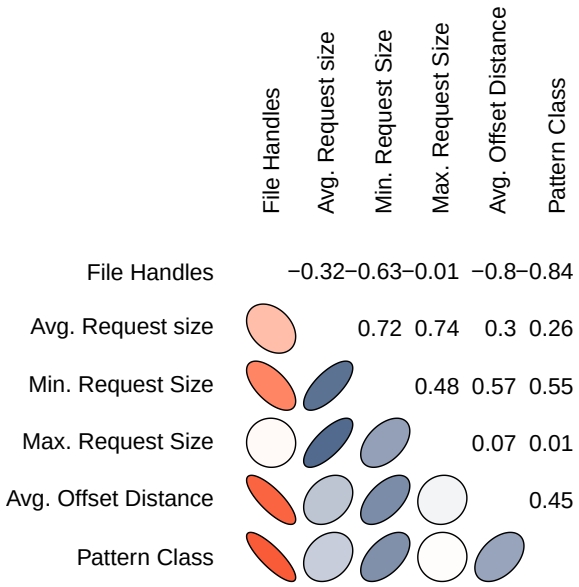
- file-per-process with contiguous accesses (FPP);
- shared-file with 1D-strided accesses (SS);
- shared-file with contiguous accesses (SC).

The access pattern detection mechanism receives input information collected at each I/O node. This information consists of the number of file handles, the request size (maximum, minimum, average), and the average offset distance between consecutive requests to the same file handle. These parameters were selected from a set of 35 metrics we have initially considered, by calculating the Spearman's nonparametric correlation (SPEARMAN, 1904). It aids in identifying the ones most related to the access pattern class, as that correlation determines the strength and direction of the monotonic relationship between two variables. Figure 3.2 shows the coefficients for the selected metrics. It is possible to see a strong negative relationship between the number of file handles and the pattern class. This metric should allow us to detect the file layout. The minimum and average request size, and the average offset distance exhibit a direct correlation to the pattern we want to classify. Intuitively these last metrics should allow us to detect if requests are contiguous or 1D-strided.

To build our access pattern detection mechanism and eliminate potential noise from start-up and tear-down phases, we extracted the observations from the center of each test. We also made sure to take the same number of observations for each access pattern

[2] <https://github.com/hpc/ior>

Figure 3.2: Spearman's nonparametric correlation coefficient for the metrics. Positive correlations are displayed in blue and negative correlations in red. The color intensity and the size of the ellipse are proportional to the coefficients.



Source: Author

and configuration, to avoid bias toward one of the classes. The final data set contains 40 observations, from each of the $1,008$ experiments (144 scenarios $\times$ 7 window sizes), yielding a total of $\approx$ 40 thousand observations. We explore decision trees in Section 3.2.1, and random forests in Section 3.2.2. We investigate neural networks in Section 3.2.3. For all the approaches, we have split the $40,240$ observations into two: 70% for training ($28,168$ observations) and 30% for testing ($12,072$ observations).

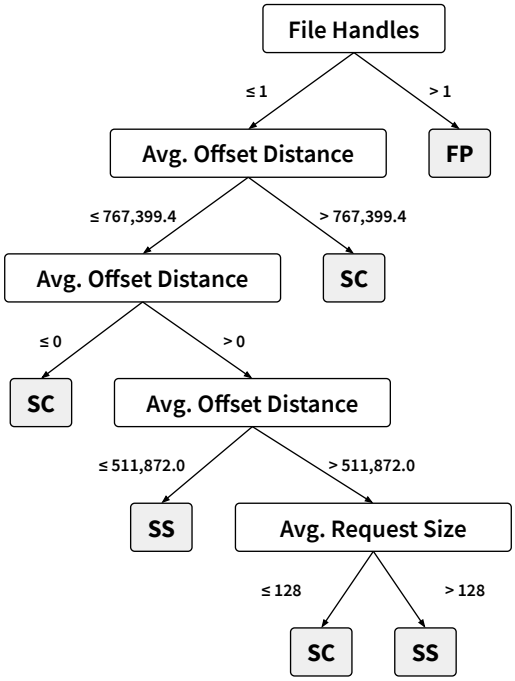
It is paramount for any detection mechanism to generalize when faced with previously unseen metrics. That is one of the main reasons we compare distinct learning approaches to this problem. To evaluate their behavior in such a scenario, we collected additional $54,210$ metrics with requests sizes of 64KB and 128KB covering read and write operations, a different number of I/O nodes, processes, file layout, and spatiality. Different request sizes not seen during training could potentially make it harder to identify the spatiality of requests.

3.2.1 Decision Trees Approach

Decision trees are often an efficient approach to classification problems. Therefore they might prove suitable for detecting the I/O access pattern at runtime. To build our tree, we applied the C5.0 algorithm (KUHN; JOHNSON, 2013), a data mining tool for discovering patterns that delineate categories, assembling them into classifiers, and using

them to make predictions. The classifiers are expressed as decision trees or sets of if-then rules, that are generally easy to understand and implement. The C5.0 algorithm has become the industry standard for producing decision trees because it does well for most problems directly out of the box (BALI; SARKAR; LANTZ, 2017). Compared to more advanced and sophisticated machine learning models, this solution generally performs nearly as well but are much easier to understand and deploy.

Figure 3.3: Decision Tree to classify access patterns into the tree classes: file per process (FP); shared file, contiguous (SC); and shared file, 1D strided (SS).

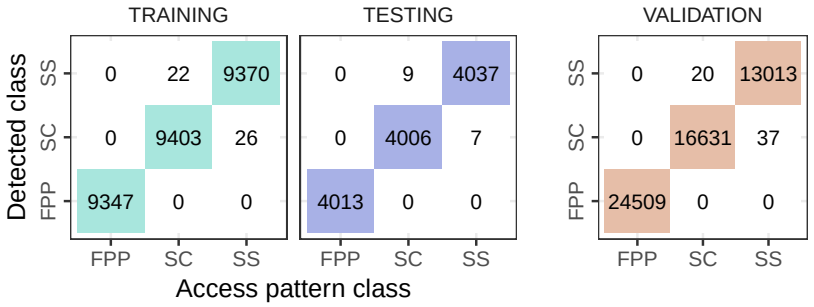


Source: Author

Figure 3.3 depicts the generated tree with its decisions. It uses three attributes and five predictors (features): the number of file handles, the average offset distance, and the average received size. Figure 3.4 illustrates the confusion matrices for the training and testing data sets. The overall accuracy was of 0.9976. Table 3.2 details the sensitivity and specificity considering a *one-vs-all* scenario. As we have three classes, these results are calculated by comparing each level to the remaining levels. Sensitivity measures the proportion of actual positives that are correctly identified as such. On the other hand, specificity measures the proportion of actual negatives that are correctly identified.

It is paramount for any detection mechanism to generalize when faced with previously unseen metrics. Therefore, we also evaluated the decision tree behavior with the validation dataset composed of $54,210$ metrics. A validation dataset is different from the test dataset. Both comprise data that was held back while training. However, it is instead

Figure 3.4: Confusion matrices for the training, testing, and validation datasets. The x-axis shows the real class, and the y-axis shows what was detected by the DT. The classes are: file per process (FPP); shared file, contiguous (SC); and shared file, 1D strided (SS).



(a) $40,240$ patterns for training and testing (b) $54,210$ patterns

Source: Author

Table 3.2: C5.0 algorithm statistics for each access pattern.

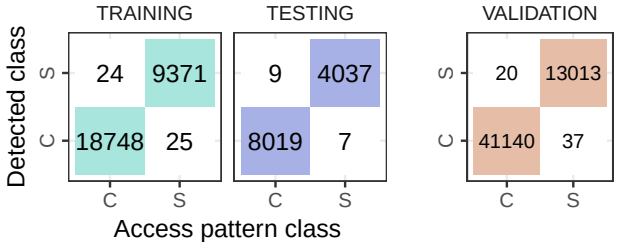
	FPP	**SC**	**SS**
Sensitivity	1.0000	0.9972	0.9977
Specificity	1.0000	0.9988	0.9986

Source: Author

used to give an unbiased estimate of the skill of the final tuned model when comparing or selecting between final models. As depicted by Figure 3.4(b), only 57 observations were incorrectly classified, which represents approximately 0.1% of the total. The accuracy was 0.9989 with Kappa of 0.9984. Kappa (COHEN, 1960) is a metric that compares an observed accuracy with an expected accuracy. It can handle both multi-class and imbalanced class problems. It describes how much better a classifier performs over another that makes random guesses based on the frequency of each class. It is always less than or equal to one. Zero or negative values indicate that the classifier is useless. Landis and Koch (1977) provide a way to characterize the output where a value < 0 indicates no agreement, $[0 - 0.20]$ slight, $[0.21 - 0.40]$ fair, $[0.41 - 0.60]$ moderate, $[0.61 - 0.80]$ substantial, and $[0.81 - 1]$ represents almost perfect agreement.

One could argue that identifying file layout, i.e., file-per-process or shared-file could be simplistic or even yield better results if such parameter was not considered when building the three, as on the shared-file scenario we expect a single file to be accessed. To evaluate such a hypothesis, we have removed that parameter and repeated our analysis. Differently from our previous approach, the decision tree uses only four predictor variables, and a tree of size five. Furthermore, only two attributes were used to build the tree: the average offset distance, and the average received request size.

Figure 3.5: Confusion matrices for the training, testing, and validation datasets. The x-axis shows the real class, and the y-axis shows what was detected by the DT. The classes are: contiguous (C) and 1D strided (S) accesses.



(a) $40,240$ patterns (b) $54,210$ patterns

Source: Author

Figure 3.5 depicts the confusion matrix with the training and testing data sets. During training, our model correctly classified $28,119$ of the $28,168$ inputs, an accuracy of 0.9983. When compared to the decision tree using the three classes, the accuracy is very similar. Therefore, we could select the simpler model using it only to identify the spatiality of accesses. However, in practice, as a decision tree is implemented in a series of if-else statements, we do not expect any changes in performance using the simple method.

3.2.2 Random Forests Approach

Random forests are an ensemble method that makes predictions by averaging over the predictions of several independent decision trees (BREIMAN, 2001). Such an approach often demonstrates improvements in classification accuracy due to the ensemble of trees and voting for the most popular class.

To train the random forest, we employed a re-sampling using cross-validation (25 fold). Resampling methods can generate different versions of our training set that can be used to simulate how well models would perform on new data. The k-fold cross-validation creates k different versions of the original training set with the same approximate size. Each set contains $1/k$ of the training set, and each excludes different data points. The analysis sets have the remainder (the "folds") (KUHN; JOHNSON, 2019).

Cutler et al. (2007) reported that the classification rates and performance metrics of their random forest model were stable with different values for *mtry*, which represents the number of variables available for splitting at each tree node. Conversely, Strobl et al. (2008) noticed a strong influence on predictor variable importance. Thus, we explored different values for the $mtry$ parameter. Kappa (COHEN, 1960) was used to select the

optimal model using the largest value, considering different values for the *mtry* parameter. Table 3.3 summarizes the results.

Table 3.3: Random forests to detect the pattern class.

	$mtry = 2$	$mtry = 3$	$mtry = 5$
Accuracy	0.99801	0.99818	0.99733
Kappa	0.99701	0.99728	0.99600

Source: Author

Using the best option, i.e., $mtry = 3$, with a forest of 500 trees, the accuracy for training was of 0.9983 and 0.9986 for testing. Once more, the file-per-process (FP) class is the one with perfect detection, as it only depends on the number of file handles detected at the forwarding layer. For the other two categories, each one has only misclassified 47 and 17 metrics during training and testing.

Due to the perfect detection of the file-per-process class, we also investigated if there are performance improvements by simplifying the model to detect only the spatiality of the requests. We employ the same methodology, data set, and configuration of the random forest as before. Table 3.4 details the results. It is possible to notice that accuracy is also not impacted by simplifying the model. Nonetheless, when applied in practice, an additional verification would be required to detect the file layout, i.e., if each process is issuing an operation to its own file or a single shared-file is used. We do not depict the confusion matrices for the random forest approach as they only presented minor differences from the decision tree.

Table 3.4: Random forests to detect the spatiality of the accesses.

	$mtry = 2$	$mtry = 3$	$mtry = 5$
Accuracy	0.99827	0.99823	0.99765
Kappa	0.99614	0.99606	0.99478

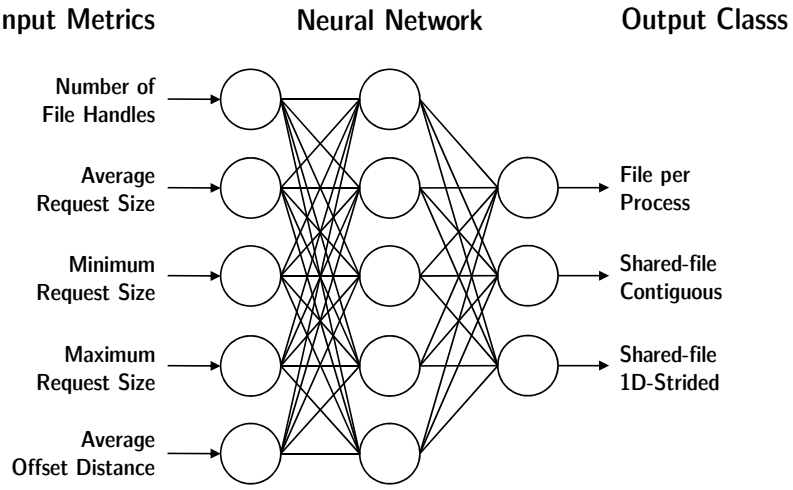
Source: Author

3.2.3 Neural Network Approach

Our Neural Network (NN) classifier was built with Keras (CHOLLET et al., 2015), a high-level Neural Network API, using TensorFlow (ABADI et al., 2015) as back-end. The dataset was split into two: 70% for training and 30% for testing. Before feeding our metrics to the NN, we applied Yeo-Johnson (YEO; JOHNSON, 2000), scale, and cen-

ter data transformations. The Yeo-Johnson transformation is similar to Box-Cox (BOX; COX, 1964), but it does not requires the input to be strictly positive. Both can be used to improve data normality. The scale transformation computes the standard deviation for a feature and divides each value by it. Finally, the center calculates the mean of each feature and subtracts it from each value. These transformations are applied to the input dataset so that the data is better suited for the NN and to speed up training.

Figure 3.6: Neural Network architecture employed to classify the metrics into the three classes, regarding the file layout and the spatiality of access.

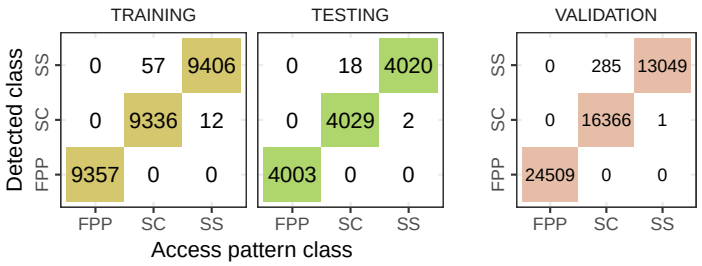


Source: Author

We initially explored different network architectures (varying the number of hidden layers and neurons on each layer) and activation functions. The model that achieved the highest precision is described in detail in this section. Our final model consists of three layers, as illustrated by Figure 3.6: an input layer with the five features, a hidden layer with the same number of neurons, and an output layer with three units, one to represent each class. The first two layers use a Rectified Linear Unit (ReLU) (HAHNLOSER et al., 2000) activation function with a normal kernel initialization function. The output layer uses *softmax* to squash the outputs of each unit in the range $[0, 1]$ and to ensure that the total sum of the outputs is equal to one.

We used the *RMSProp* optimizer with learning rate of 0.001 and a momentum of 0.9. The loss function was the categorical cross-entropy. The output is an n-dimensional vector that is all-zeros except for a 1 at the index corresponding to the class of the sample. In our case, we have a 3-dimensional vector for each sample. We trained our model on $22,534$ samples and tested it on $5,634$ samples with a batch size of 32 and 50 epochs. An epoch refers to one cycle through the full training dataset. The training accuracy was 99.76%, and the testing accuracy was 99.73%.

Figure 3.7: Confusion matrices for training, testing, and validation. The x-axis shows the real class, and the y-axis the class detected by the NN: file per process (FPP); shared file, contiguous (SC); and shared file, 1D strided (SS).



(a) $40,240$ patterns for training and testing (b) $54,210$ patterns

Source: Author

Figure 3.7 shows the confusion matrix of the generated NN with the training and testing data sets. During training, our model correctly classified $28,099$ of the $28,168$ inputs, an accuracy of 99.76%. We also checked our model's performance with our testing data set (30% of the original data). During testing, our model incorrectly classified only 20 samples out of the $12,072$. It is also important to notice all three classes are correctly identified with a reasonable probability. We applied the neural network with the validation dataset. As depicted by Figure 3.7(b), 286 observations were incorrectly classified which represents approximately 0.53% of the total.

3.3 Discussion

Our results have shown that all detection approaches covered in this work can correctly detect the access pattern. The most straightforward approach, represented by the decision tree, presented 0.99 accuracy for the training, testing, and validation datasets. The random forest approach also yielded similar results. Despite the similar accuracy, the neural network represents a black-box model if compared to the alternatives.

Table 3.5: Runtime to train and make predictions.

	Train (s)	**Predict (ms)**
Decision Tree	0.369	1.371
Random Forest	364.505	1.363
Neural Network	54.177	9.825

Source: Author

We evaluated the time taken by each approach to train and to make predictions.

The training phase is often done offline and can take longer to complete. Moreover, it will only be required to update the model, which should not happen as frequently as the predictions in runtime. Table 3.5 summarizes the median of 10 repetitions to train each model and the median prediction time of all the metrics available in the dataset used to train and validate. All the approaches take in the order of milliseconds to predict once the model is trained. However, the neural network takes $\approx 7.2\times$ more than the others.

3.4 Case Study: Tuning an I/O Scheduler Parameter

Server Time WINdowS (TWINS) (BEZ et al., 2017) is an I/O scheduler designed for the I/O forwarding layer. It seeks to coordinate the I/O nodes' accesses to the shared PFS servers to mitigate contention and interference. To achieve coordination TWINS assumes that, at any given moment two conditions hold true:

I. an I/O node is focusing its accesses to only one of the PFS data servers;

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

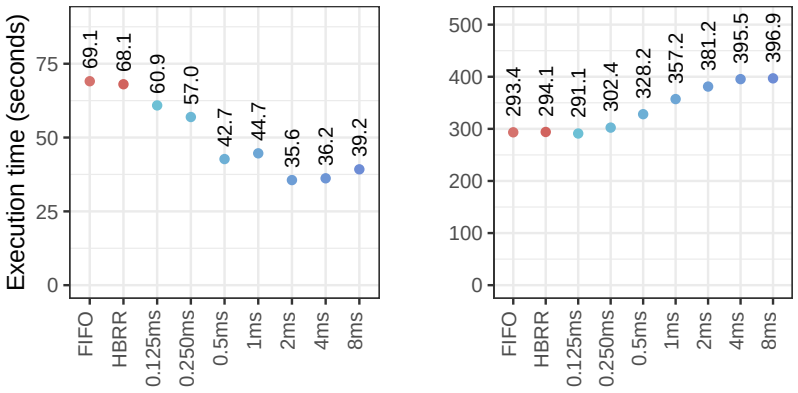
Algorithm 1 describes TWINS pseudo-code. Each I/O node keeps multiple request queues, one per data server. During a configurable *time window*, requests are taken (in arrival order) from only one of the queues to access only one of the servers. Different I/O nodes target different servers in a given time window. When the time window ends, the scheduler moves to the next queue following a round-robin scheme. To cause the desired distribution effect, we add an extra server identifier translation step before adding requests to the corresponding queues. This translation is done according to the I/O node identifier. Details about TWINS' implementation and performance evaluation in multiple scenarios are available in our previous work (BEZ et al., 2017).

Algorithm 1 TWINS I/O request scheduler

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

TWINS can increase (Figure 3.8(a)) but also decrease (Figure 3.8(b)) performance over other schedulers available for the I/O forwarding layer. We depict two scenarios that highlight such differences. We compared it to FIFO and HBRR provided by the IOFSL framework (ALI et al., 2009). In both situations, **the correct selection of the time window parameter, which depends on the access pattern, is of paramount importance**.

Figure 3.8: Impact of the window size on performance based on the execution time as perceived by the user (makespan). A total of 128 processes access a 4GB shared file in 32KB 1D-strided requests. Baseline algorithms are colored in red and TWINS (distinct windows) are in blue. The y-axis is different in each plot.



(a) `READ`, 8 forwarding nodes (b) `WRITE`, 2 forwarding nodes

Source: Author

To demonstrate the applicability of the access pattern detection techniques described in Sections 3.2.1 (decision trees), 3.2.2 (random forests), and 3.2.3 (neural networks), we evaluate those strategies in a case study in which the accurate detection of the access pattern is fundamental: selecting the TWINS window size. For that, metrics are collected at the I/O forwarding layer and are used by the detection mechanism to identify the access pattern and tune the scheduler's configuration.

Figure 3.8 illustrates the benefits of correctly picking a window based on the access pattern, but also the overhead a wrong choice could introduce. In the situation presented in Figure 3.8(a), TWINS provides performance improvements between 10% and 48%, depending on the selected window duration. On the other hand, in Figure 3.8(b), the performance *decreases* approximately 35% with windows of 8ms, which could be avoided by using a smaller window. Therefore, to achieve the best performance with TWINS, we need to tune its time window duration to the current situation.

3.5 Applying the I/O Access Pattern Detection

When applied to a real tuning mechanism, mispredictions could have a higher or lower impact, depending on the class and on how optimizations applied to that given class behave in such non-optimal scenarios. Therefore, we complete our investigation by implementing the **decision tree**, **random forest**, and **neural network** methods to the case study presented in Section 3.4, where **we seek to tune the window size of the TWINS scheduler**. We applied the detection mechanism to each observation in the entire dataset of over one million entries and used this detection to determine the best window at each instant. In this section, we assume that if a pattern is correctly detected, the best window is always selected. Since we have a comprehensive set of experiments with performance metrics of multiple access patterns using distinct window sizes, we can evaluate it offline.

Figure 3.9: The number of patterns where performance was increased considering different policies to tune the I/O scheduler parameter. Results are grouped by the number of I/O nodes. The y-axis is not the same in all the plots. *O = Oracle, S = Static, T = Decision tree, F = Random forest, and N = Neural network.*

(a) 1 forwarding node (b) 2 forwarding nodes (c) 4 forwarding nodes (d) 8 forwarding nodes

Source: Author

By comparing the performance results with a baseline, we count the number of decisions that resulted in performance improvements or decreases, as presented in Figure 3.9 and Figure 3.10. The baseline to compare performance was using a $1\,\mathrm{ms}$ windows, a conservative value that decreases (and increases) performance for the least number of scenarios. This value would represent the best decision if we had to select a single fixed window to accommodate all the tested workloads. We compare the results of the three mechanisms with an oracle and a static solution. Because we have performance results with all the seven TWINS window sizes considered in our experimental campaign, we built the oracle by selecting the window size that yielded the highest bandwidth in each scenario. For the static solution, a $125\,\mu\mathrm{s}$ window is always used. This value was chosen as it increases performance for the highest number of scenarios (BEZ, 2016).

Results, where performance was increased, are presented in Figure 3.9 and are

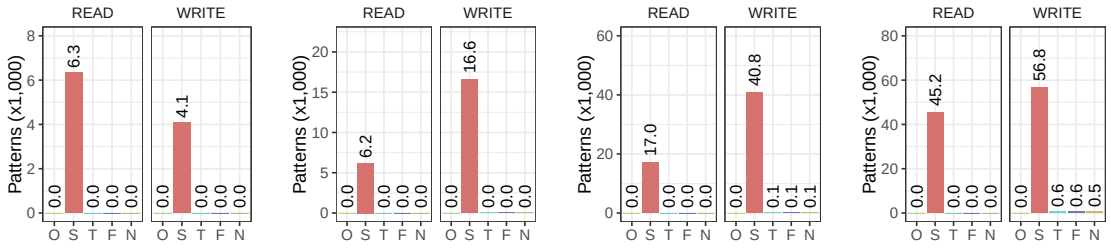
grouped by the number of I/O forwarding nodes and operation. The y-axis of each plot is on a different scale, and they all represent the number of patterns $\times 10^3$. All approaches to detect the access pattern at runtime were able to perform better than the static solution for all scenarios. Table 3.6 summarizes the differences with higher precision.

For read operations, the decision tree was able to improve performance for all the situations where the oracle did when 1, 2, and 4 I/O forwarding nodes are used, respectively. However, for 8 I/O nodes, it improved performance for 99.9% of the cases. That represents 9.7% (1 I/O node), 6.5% (2 I/O nodes), 11.0% (4 I/O nodes), and 16.9% (8 I/O nodes) more than the static solution. The random forest approach increased performance by the same amount as the decision tree. The neural network behaved similarly. For write operations, all detection approaches are comparable to the oracle, with 1 and 2 I/O nodes. With 4 and 8, the decision tree achieves 99.9% and 99.8% of the oracle, respectively. That represent 6.0% (1 I/O node), 19.7% (2 I/O nodes), 31.2% (4 I/O nodes), and 22.0% (8 I/O nodes) more than using a fixed $125\,\mu s$ window size. In summary, using such detection approaches, we can increase performance by 17% over using a statically defined window.

Figure 3.10 depicts all the scenarios with performance decrease. Results are also grouped by the number of I/O forwarding nodes and operation. The y-axis of each plot is on a different scale, and they all represent the number of patterns $\times 10^3$. The three detection approaches were able to outperform the static choice, for all scenarios, guiding the scheduler to avoid using windows sizes that would be harmful to the I/O performance. Table 3.7 compares the results with higher precision to detail the differences.

For read operations, the decision tree was able to avoid performance decrease for all the situations where the oracle did when 1, 2, and 4 I/O forwarding nodes are used, respectively. Minor impacts were seen using 8 I/O nodes. The random forest and the

Figure 3.10: The number of patterns where performance was decreased considering different policies to tune the I/O scheduler parameter. Results are grouped by the number of I/O nodes. The y-axis is not the same in all the plots. *O = Oracle, S = Static, T = Decision tree, F = Random forest, and N = Neural network.*



(a) 1 forwarding node (b) 2 forwarding nodes (c) 4 forwarding nodes (d) 8 forwarding nodes

Source: Author

Table 3.6: Number of patterns where performance was increased.

Operation	Approach	1 ION	2 IONs	4 IONs	8 IONs
READ	**Oracle**	71,672	100,653	170,833	311,322
	Static	65,326	94,438	153,824	266,126
	Tree	71,672	100,653	170,833	311,316
	Forest	71,672	100,653	170,833	311,316
	Network	71,672	100,653	170,830	311,310
WRITE	**Oracle**	71,677	100,653	170,834	311,314
	Static	67,575	84,063	130,038	254,552
	Tree	71,676	100,636	170,687	310,757
	Forest	71,676	100,637	170,691	310,756
	Network	71,676	100,644	170,727	310,782

Source: Author

Table 3.7: Number of patterns where performance was decreased.

Operation	Approach	1 ION	2 IONs	4 IONs	8 IONs
READ	**Oracle**	0	0	0	0
	Static	6,346	6,215	17,009	45,196
	Tree	0	0	0	6
	Forest	0	0	0	6
	Network	0	0	3	12
WRITE	**Oracle**	0	0	0	0
	Static	4,102	16,590	40,796	56,762
	Tree	1	17	147	557
	Forest	1	16	143	558
	Network	1	9	107	532

Source: Author

neural network behaved similarly. For write operations, incorrect detections at runtime causes loss of performance for some patterns. Table 3.7 summarizes these differences.

Nonetheless, all detection approaches were able to avoid most of those scenarios. With 4 and 8, the decision tree decreased performance for 0.36% and 0.98% of the patterns where the static solution did. When applying the random forest instead, similar behavior is observed: 0.35% and 0.98%. Finally, with the neural network to make the detection, identical behavior is observed with slight differences between 0.26% and 0.93%.

3.6 Final Remarks

Different optimization techniques have been proposed to improve I/O operations' performance at many levels of the I/O stack. These techniques typically achieve their goals in situations they were designed to improve performance, but not for all possible

scenarios. Furthermore, they often require fine-tuning of parameters to yield better results.

In this chapter, we demonstrated the applicability of machine learning techniques to automatically detect the I/O access pattern of HPC applications at runtime. We investigated decision trees, random forests, and neural networks to classify runtime metrics into common access patterns. We demonstrated that all three approaches have an equivalent accuracy but different training and usage overheads. Furthermore, to illustrate its applicability, we evaluated these strategies by investigating the benefits of correctly selecting the TWINS scheduler's window size at the forwarding layer. We used some performance measurements obtained with TWINS to make an offline adaptation depending on the detected access pattern. However, that adaptation was simply an oracle as the only goal was to evaluate how well the access pattern detection was working. We detail and evaluate a practical solution for this adaptation problem, using TWINS and applying one detection mechanism, in Chapter 4.

The proposed approaches to detecting the access pattern are not specific to tuning the TWINS window size parameter. They can be applied in the context of other optimization techniques that also require or benefit from runtime knowledge of the current I/O access pattern such as request schedulers, MPI-IO collective operations, and HDF5 tuning.

4 DYNAMIC TUNING OF I/O FORWARDING SCHEDULER

In this chapter, we propose a practical approach to adapt the I/O system to a changing workload by selecting appropriate values for parameters, which depend on the current access pattern. One way of adapting the stateless I/O system would be to observe the current access pattern over some time, detected by the techniques described in Chapter 3, and combine it with a tuning strategy. A possible tuning strategy could be a supervised technique (a decision tree, for example), which would require a previous training step. However, considering I/O performance is sensitive to many parameters, creating a training set to represent all applications (and all interactions of concurrent requests) and executing it would be both difficult and time-consuming. Therefore the best option is to learn the best choice for each situation during the execution of applications.

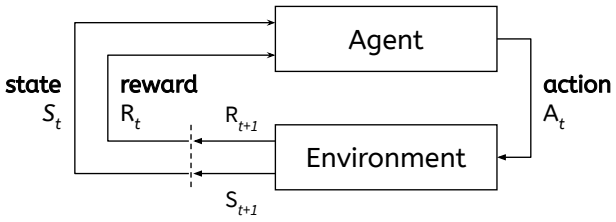
Though we use TWINS in this work as a case study to demonstrate our adaptation technique's effectiveness, it is **not restricted to it**. The approach we propose in this work could be adopted to tune any other parameter at runtime that depends on the observed application's access pattern, such as quantum-based request scheduling algorithms (Handle-Based Round-Robin (OHTA et al., 2010) or aIOLi (LEBRE et al., 2006)) or middleware tunable parameters such as the ones available for MPI-IO collective operations.

4.1 Adaptive I/O Forwarding

We require a solution where the I/O forwarding layer learns the best choice for different situations while being observed, but without prior training due to its high cost and complexity. Thus, we approach this as a Reinforcement Learning (RL) problem (SUTTON; BARTO, 2017) where we take actions in an environment and use the feedback from these actions and experiences to maximize the notion of cumulative reward, as illustrated by Figure 4.1. In it, the learner and decision maker is depicted as the *agent*. The thing it interacts with, comprising everything outside the agent, is called the *environment*. These two interact continually in a sequence of discrete time steps. At each time step t, the agent receives some representation of the environment's state S_t, and on that basis selects an action A_t. In the following step, it receives a numerical reward R_{t+1} for taking the action and it finds itself in a new state S_{t+1}.

We approach our need for adaptation at the I/O forwarding layer as a *k-armed bandit problem* (BERRY; FRISTEDT, 1985), where at each step, an agent takes one of

Figure 4.1: The agent-environment interaction in Reinforcement Learning (RL).



Source: Sutton and Barto (2017)

the k possible actions (for TWINS each action is a different time window duration) and receives a reward (performance represented by the bandwidth). The expected reward from one action is called its *value*. Since the value of an action a, denoted by $q_*(a) \doteq \mathbb{E}[R_t|A_t = a]$, is not known beforehand, at the time step t, we only have an estimate of it represented as $Q_t(a)$. Ideally $Q_t(a)$ would be close $q_*(a)$. This estimate is based on the rewards previously obtained by taking that action. The algorithm selects actions with the highest estimated values (*exploitation*), but also occasionally chooses other actions to attain better estimates of their values (*exploration*).

The values (performance with each parameter) change as the access pattern changes. To avoid having to "reset" the learning process whenever that happens, we model our problem as a *contextual bandit* (or *associative search* task) (SUTTON; BARTO, 2017): we have multiple concurrent "instances" of the k-armed bandit, one for each different access pattern. At each iteration, only one of these instances will be active. This choice carries the assumption that an action does not have an impact on the following observation. Tuning the parameter will impact the current I/O phase's performance and, therefore, slightly anticipate or delay the next I/O phase. Still, we consider this effect as negligible because the applications primarily dictate the access pattern.

We implemented each of the concurrent armed bandit instances as an ε-*greedy* algorithm (WATKINS, 1989; SUTTON; BARTO, 2017), that at step t takes the action a of the highest estimated value $Q_t(a)$, with probability $(1-\varepsilon)$, or with probability ε takes a randomly selected action. Value estimates use *incrementally computed sample averages*, i.e., after obtaining this step's reward R, the estimate for a is updated as Equation 4.1, where $N(a)$ is the number of times a has been taken:

$$Q_{t+1}(a) = Q_t(a) + \frac{1}{N(a)}[R - Q_t(a)] \tag{4.1}$$

The pseudo-code for a complete bandit algorithm using the incrementally computed sample averages and the ε-greedy action selection is detailed by Algorithm 2. The

Algorithm 2 Simple bandit algorithm using ε-greedy by Sutton and Barto (2017)

1: **for** $a \leftarrow 1$ to k **do**
2: $\quad Q(a) \leftarrow 0$
3: $\quad N(a) \leftarrow 0$
4: **end for**
5: **while** $true$ **do**
6: $\quad A \leftarrow \begin{cases} \underset{a}{\operatorname{argmax}}\, Q(a) & \text{with probability } 1-\varepsilon \quad \text{(breaking ties randomly)} \\ \text{random action} & \text{with probability } \varepsilon \end{cases}$
7: $\quad R \leftarrow bandit(A)$
8: $\quad N(A) \leftarrow N(A) + 1$
9: $\quad Q(A) \leftarrow Q(A) + \frac{1}{N(A)}[R - Q(A)]$
10: **end while**

$bandit(A)$ function takes action A in the system and returns the corresponding observed reward. Adapting it for the *contextual bandit* scenario would comprise identifying it uniquely prior to activating the corresponding k-armed bandit instance.

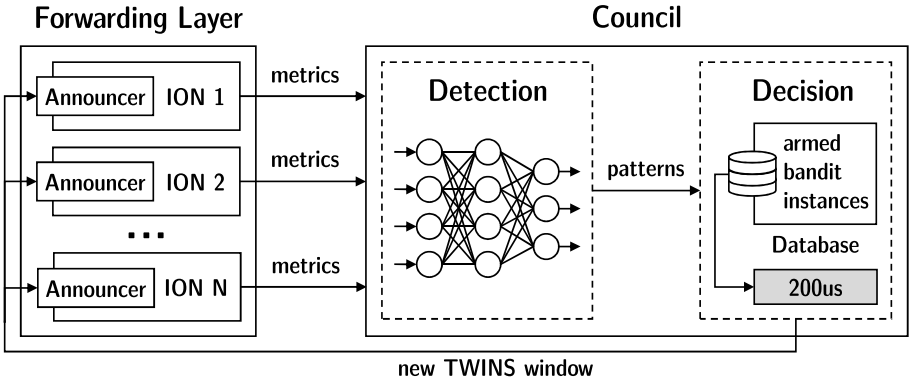
Since the proposed mechanism will run in the I/O nodes, its lifetime will **not** be constrained to the jobs' execution. Consequently, after observing an access pattern multiple times, it will be capable of consistently providing performance improvements by making the best decision for that given pattern on any applications that use it.

4.1.1 Architecture of the proposed mechanism

Our case study (TWINS) described in Section 3.4 aims to achieve global coordination. Thus although request scheduling happens independently on each I/O node, decisions about the window size should **not**, as multiple I/O nodes should use the same parameter value for it to make sense. Hence, we included a centralized agent, the *Council*, depicted by Figure 4.2, located on a separate node. In each I/O node, the *Announcer* is responsible for collecting and asynchronously sending metrics about the observed access pattern to the *Council* and receiving the selected window size the node should use. The *Council*'s workflow consists of two steps: the *detection* and *decision* phases. The *detection* phase is responsible for classifying the observed access pattern from the metrics collected on each I/O node using an access pattern detection technique such as the ones proposed by Bez et al. (2019) and Boito et al. (2019). A new *decision* is only made if the metrics allow for detection (i.e. if sufficient requests are flowing through this layer).

Algorithm 3 details the *Council* interactive decision and learning process depicted in Figure 4.2. Once a set of metrics is received (line 4), the access pattern is detected for

Figure 4.2: The proposed architecture includes the *Announcers*, at the I/O nodes, and the centralized *Council* (on a separated node) where detection and decision take place.



Source: Author

Algorithm 3 Council

Input:

ion_number is the number of connected I/O nodes
ε is the greedy exploration ratio

1: $S \leftarrow \varnothing$, $window \leftarrow default\_window$
2: **while** $true$ **do**
3: **while** $S.size() < ion\_number$ **do**
4: $host, metrics \leftarrow receiveMetrics()$
5: $reward \leftarrow getBandwidth(metrics)$
6: ▷ Use any access pattern detection mechanism
7: $pattern \leftarrow detectAccessPattern(metrics)$
8: ▷ Update N and Q with the reward, i.e. bandwidth
9: $increment(N[pattern][window])$
10: $update(Q, N, pattern, window, reward)$ ▷ Eq. 4.1
11: ▷ Select the recommended window based on the pattern
12: $choices[host] = \underset{window}{\operatorname{argmax}}\, Q[pattern][window]$
13: $S.add(host)$
14: **end while**
15: ▷ Define the window size for all I/O nodes
16: **if** $random(0, 1) \leqslant \varepsilon$ **then**
17: $window \leftarrow randomWindow()$ ▷ Explore
18: **else**
19: $window \leftarrow mostOccurringInList(choices)$ ▷ Exploit
20: **end if**
21: $announceWindow(window)$
22: $S.reset()$
23: **end while**

each I/O node separately (line 7), and the corresponding instance of the armed bandit is selected. Then the value estimate of the previously taken action is updated according to Equation 4.1 (line 10), using the observed performance as a reward. The estimates are used to determine the window size suitable for that I/O node (line 12). Upon making

the centralized decision, the *Council* must decide between *exploration* or *exploitation* (line 16). For the latter, a consensus is required (line 19) between the possible divergent recommendations for the different I/O nodes. In this work, we make the best choice for the majority of the I/O nodes.

TWINS requires the centralized *Council*. However, if the optimization tolerates different decisions by the I/O nodes, the detection and decision steps would instead happen on each I/O node, and no centralized agent would be needed. In this case study, the centralized approach has the additional benefit of accelerating the learning process, as parallel experiences are combined in the value estimates. A compromise to decrease the time required to reach a decision would be a hierarchical approach, where the ability to reach global consensus is sacrificed for a reduced overhead.

It is important to notice that only the I/O nodes will interact with the *Council*, and not all the compute nodes. For instance, if we consider the TaihuLight machine, only the 240 I/O nodes interact with the *Council*. In our experimental environment, for such a scale, that interaction would take < 250ms (Figure 4.12). Furthermore, since the interaction with the centralized agent is completely asynchronous for the I/O nodes' perspective, the *Council* should not become a bottleneck. Further discussion on our solution's overhead and scalability is presented in Section 4.2.5.

4.1.2 Required access pattern detection mechanism

Our solution requires a server-side classification of observed access patterns to identify an armed bandit instance. Hence, that classification needs to cover all aspects that impact the behavior of the tuned optimization. Otherwise, each instance would not be able to learn correctly. On the other hand, redundant classes slow down the learning, as multiple armed bandit instances cover the same behavior. Specifically to our case study, we classified the access pattern regarding the operation (read or write), spatiality (contiguous or 1D-strided), number of accessed files, and request size aspects, since our extensive performance evaluation of TWINS showed these to identify the different behaviors uniquely.

From the approaches presented in Chapter 3, we chose to use a neural network for the detection of spatiality (Section 3.2.3). As new patterns could be added in the future to represent the workload of different types of applications, we believe a NN would be a more flexible solution. Moreover, choosing this approach, that presented the high-

est prediction time (~9ms), would serve as an upper bound time of the detection phase. Nonetheless, any other detection strategy could be plugged into this step.

It is important to remember this is a server-side classification, where information from user-side libraries is not available, and very little is known about the applications. When applying our proposal to other optimization techniques, the server-side access pattern detection must be adapted accordingly. If the set of relevant aspects is not known (often the case), a generic classification is required. In a previous work (BOITO et al., 2019), we proposed a classification strategy that covers all aspects. We represented access patterns as time series and used pattern matching to compare them. Our results have shown precision of up to 93% and recall of up to 99%. Mainly because of its high recall, we consider that approach to be a good strategy given the long life of the system (years) and taking into account that the alternative is not to adapt to the workload, which means giving up on possible performance improvements.

4.2 Results and Discussion

In this section, we evaluate our proposal applied to the TWINS case study. Section 4.2.1 discusses the methodology used for all experiments. We conduct an offline evaluation to show the learning ability in Section 4.2.2, and an online one to show how our architecture works in practice and leads to performance improvements in Section 4.2.3. We evaluate overhead and scalability in Section 4.2.5. Finally, we discuss some other relevant aspects of our proposal in Section 4.2.6.

4.2.1 Experimental Methodology

All experiments in this section were carried out in two clusters from the Grid'5000 platform (BOLZE et al., 2006) located at the Nancy site: Grimoire and Grisou. For our evaluation, we used the PVFS file system and the IOFSL forwarding framework. Regarding the file-system choice, PVFS is one of the most used in parallel I/O research (BOITO et al., 2018) as it is easy to deploy and modify for various purposes. Additionally, IOFSL has an integration with PVFS for improved performance. Regardless, it is important to notice that our approach does not rely on the file system nor a particular forwarding tool.

We deployed four PVFS2 servers in Grimoire nodes, 32 clients, and multiple

IOFSL (I/O forwarding) nodes in separated Grisou nodes. Each node of both clusters is powered by an Intel Xeon E5-2630 v3 processor (Haswell, 2.40GHz, 2 CPUs per node, 8 cores per CPU) and 128GB of memory[1]. The parallel file system servers use a 600GB HDD SCSI Seagate ST600MM0088. Nodes are connected by a 10Gbps Ethernet interconnection, and there are four 10Gbps links between the clusters. Both clusters were entirely reserved during the experiments to minimize network interference.

PVFS version 2.8.2 was deployed with default 64KB stripe size and striping through all data servers. They write directly to their disks, bypassing caches, to ensure the scale of tests would be enough to see access pattern impact on performance. Clients are equally distributed among the I/O nodes that communicate directly with the file system through the IOFSL dispatcher. The IOFSL daemon was launched with all its default parameters, aggregating up to 16 requests before dispatch, and using 4 to 16 threads to handle the I/O.

We executed the MPI-IO Test benchmarking tool and generated traces of all metrics collected by the I/O nodes during all executions. These traces are used for our offline analysis. When using the benchmarking tool, we sought to cover the most common access patterns of HPC applications. We varied the number of processes (128, 256, or 512), the file layout (shared-file or file-per-process), spatiality (contiguous or 1D-strided access), operation (reads or writes), and request sizes (32 or 256KB — smaller than the stripe size or large enough so that all servers are accessed). Each experiment writes/reads a total of 4GB of data. To consider multiple deployment scenarios, we used a different number of available I/O nodes (1, 2, 4, or 8). These 144 different situations (we excluded the unusual 1D-strided file-per-process) were executed with seven different values for the time window parameter, for a total of 1,008 experiments. Metrics were collected from all I/O nodes every second composing a dataset of over one million observations.

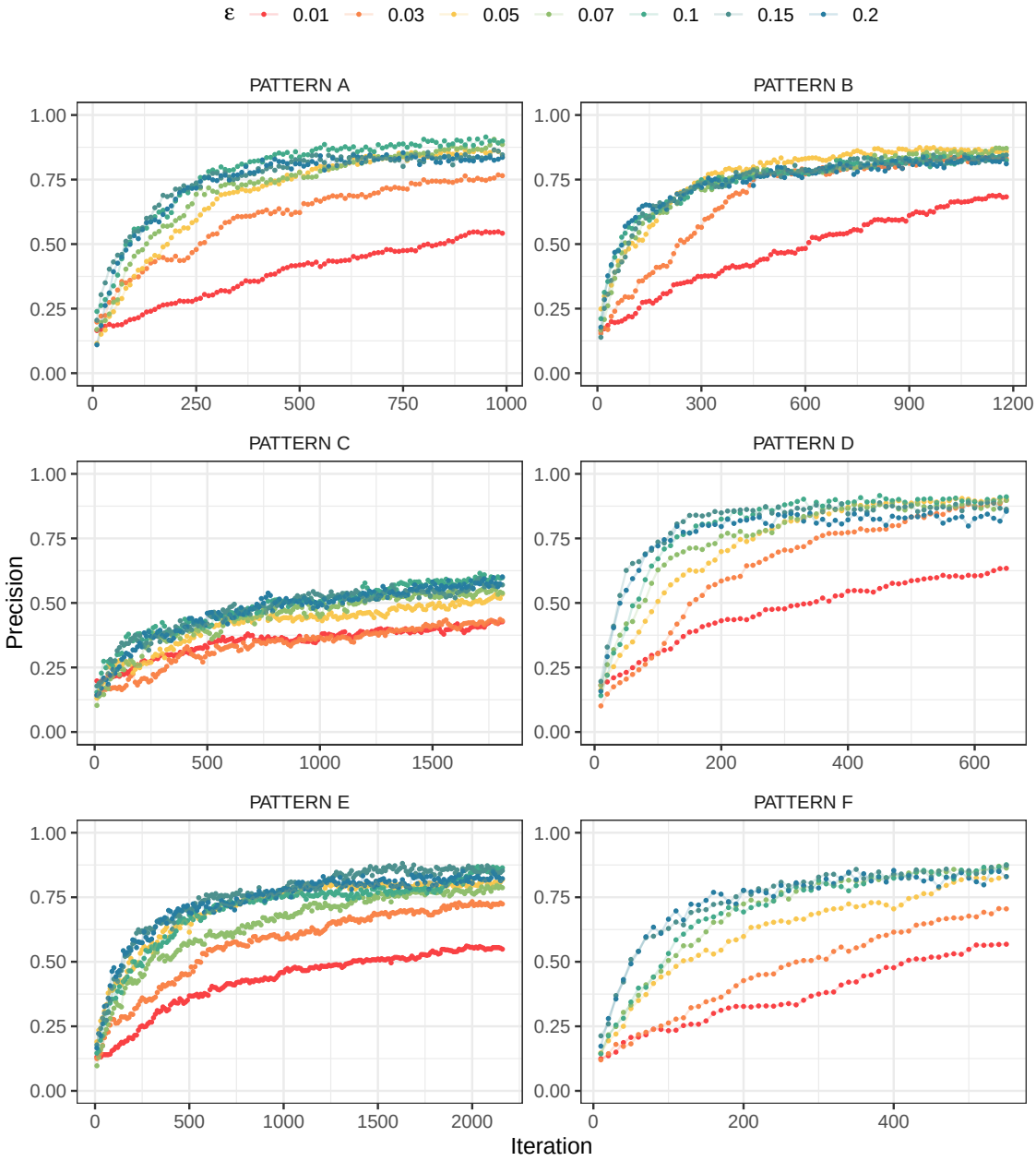
4.2.2 Offline Evaluation

To evaluate our ability to learn the best parameter value without prior knowledge, we conducted simulations of the ε-greedy approach described in Section 4.1, assuming a perfect access pattern detection, separately for each of the 144 scenarios. The algorithm has seven possible actions (window sizes) to choose from: 125 µs, 250 µs, 500 µs, 1 ms, 2 ms, 4 ms, and 8 ms. We have chosen these options based on our previous evaluation (BEZ, 2016). For the learning process, all of the actions start with a value estimate of

[1] <https://www.grid5000.fr/w/Nancy:Hardware>

zero. After deciding on an action, to determine its reward (performance), we randomly sample a dataset of previously collected real measurements with that window size in that given scenario. Thus, the simulation duration is limited by the number of available measurements. We repeat each simulation 100 times to account for the sampling variability.

Figure 4.3: Achieved precision, i.e., how often our approach chooses the correct window size, depicted in bins of 10 observations for simulations with different ε. Table 4.1 details the characteristics of the six patterns selected for this analysis. The x-axis of each plot, limited by the number of measurements of the experiments, is described in Section 4.2.1.



Source: Author

To calculate the precision, we grouped iterations into fixed-size bins. That way, it is possible to count how often we chose the window duration previously observed to be

the best for that given pattern. We then summarize the 100 simulations by their average to account for the random sampling of the dataset. We have chosen a bin size of 10 to aid in the visualization of the trends.

From the 144 scenarios, we have selected six to illustrate the learning process. They are detailed in Table 4.1. Figure 4.3 depicts the precision, i.e., how often the correct value is selected, during simulations with different ε.

Table 4.1: Selected patterns concurrently simulated in Figure 4.4

	I/O Nodes	Processes	File Layout	Request Spatiality	Request Size	Operation
A	8	128	Shared	1D-strided	32KB	read
B	2	128	Shared	contiguous	32KB	write
C	8	512	Shared	contiguous	32KB	read
D	1	128	Shared	1D-strided	32KB	write
E	1	128	Individual	contiguous	32KB	write
F	4	128	Shared	1D-strided	32KB	read

Source: Author

Table 4.2: Achieved precision and performance for the six patterns with ε-greedy.

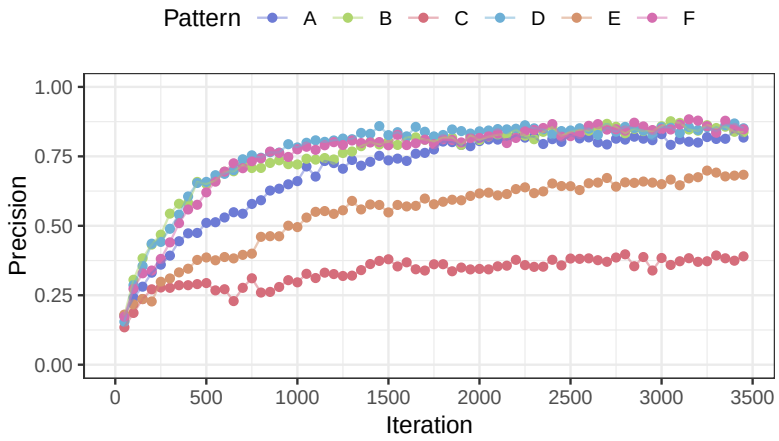
Pattern	**A**	**B**	**C**	**D**	**E**	**F**
ε	0.10	0.07	0.20	0.10	0.10	0.10
Precision	0.89	0.87	0.60	0.91	0.86	0.87
Performance	0.98	0.96	0.97	0.98	0.99	0.98

Source: Author

As the algorithm reaches better estimates for performance with different windows, it selects the best value for the parameter more frequently, thus increasing precision. The smaller the ε (probability of exploration), the slower the convergence. In the long-term, an ε of 0.15, for instance, will choose the best action only at 85% of the time. An alternative would be to start with a high value for ε and decrease it over time. Table 4.3 compiles the precision and performance achieved in the last bin (10 last iterations). Performance is normalized by what would be obtained if statically using the best possible window.

We can see our approach achieves better precision for some patterns (**A**, **B**, and **D**) than others (**C**, **E**, and **F**). That can be explained by the fact that it is easier to learn in situations where there is a distinct better choice for the parameter, with a large performance difference to other possible values. On the other hand, if multiple parameter values yield similar performance (e.g., pattern **C**), the algorithm might not converge to the correct window. It is important to notice that though precision might be low for pattern **C**

Figure 4.4: Observed precision during the simulations of the six distinct concurrent access patterns detailed in Table 4.1, depicted in bins of 50 observations, using a ε-greedy policy with $\varepsilon = 0.15$. The x-axis indicates the learning iteration of each concurrent pattern.



Source: Author

(0.60), due to the presented reasons, selecting a value somewhat identical to the best (at least for the TWINS case study) will still lead to improvements in performance. This is demonstrated by the 0.97 precision achieved in this scenario.

In practice, the different armed bandit instances will learn as the system observes different access patterns over time. To illustrate, we executed the simulation with $\varepsilon = 0.15$, but this time, at each iteration, we randomly chose one of the six patterns. Figure 4.4 presents the precision results, further detailed by Table 4.3, and confirms that the bandit can indeed learn and reach similar precision when seeing each pattern separately.

Table 4.3: Achieved precision and performance for the six patterns.

Pattern	**A**	**B**	**C**	**D**	**E**	**F**
Precision	0.88	0.88	0.49	0.87	0.59	0.59
Performance	0.99	0.96	0.96	0.97	0.98	0.92

Source: Author

Exploration plays a vital role in reducing the uncertainty about the accuracy of the action-value estimate. The ε-greedy action selection forces the non-greedy actions to be tried indiscriminately. An alternative approach would be to select among the non-greedy actions according to their potential to be optimal, considering the accuracy and the uncertainty of those estimates. In such an approach, we could employ another non-parametric algorithm named *Upper Confidence Bound* (UCB1), proposed by Auer, Cesa-

Bianchi and Fischer (2002), where actions are selected according to:

$$A_t \doteq \underset{a}{\operatorname{argmax}} \left[Q_t(a) + \sqrt{\frac{2 \ln t}{N_t(a)}} \right] \tag{4.2}$$

In Equation 4.2, $\ln t$ denotes the natural logarithm of t (the current timestep), $N_t(a)$ is the number of times that action a has been taken before time t. Upon starting, each action is considered to be a maximizing action, so it is explored at least once. The square-root term in the formula seeks to measure the uncertainty or variance in estimating an action's value. The maximization serves as an upper bound on the possible true value of a. The use of natural logarithm translates into smaller increases over time. Though UCB1 will eventually select all actions, those with lower value estimates or that have been frequently selected will be less favored over time.

Table 4.4: Achieved precision and performance for the six patterns with UCB1.

Pattern	**A**	**B**	**C**	**D**	**E**	**F**
Precision	0.92	0.97	0.73	0.97	0.68	0.89
Performance	0.99	1.00	0.98	1.00	0.99	0.98

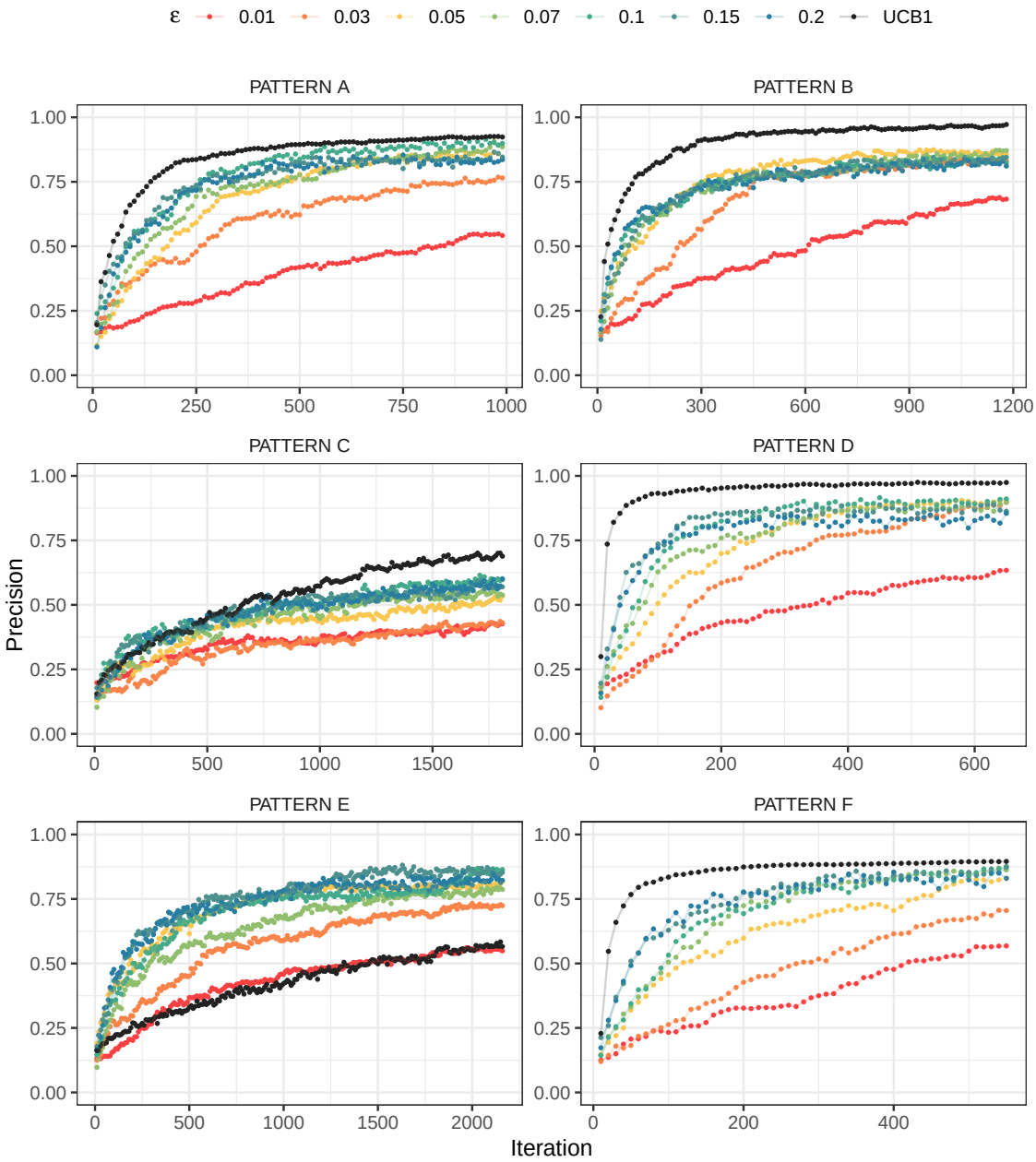
Source: Author

The achieved precision and performance when using UCB1 for the six access patterns are depicted in Figure 4.5 that also compares to the previous ε-greedy results. Table 4.4 compiles the best precision achieved in the simulation's last iteration, alongside the corresponding performance value. For four of the six patterns, a precision ≥ 0.89 can be observed, which translates into performance results that are very close to the best in those scenarios. For pattern **C**, with 0.73 precision, and pattern **E**, with 0.68 precision, we still get near to the full performance. These two cases indicate that there is not just a single best window that could be used to optimize for this pattern, but rather a subset.

4.2.3 Online Evaluation

To test our proposal in practice, we performed an evaluation in the environment described in Section 4.2.1, using four I/O nodes. We executed a benchmark using 128 processes to write and then read a 4GB shared file with 32KB 1D-strided requests using MPI-IO. This scenario (the read portion is represented by scenario **F** in the previous section) was chosen because the best window size for write requests under that pattern is the worst for read requests, and vice-versa. Thus, this particular scenario allows us to

Figure 4.5: Achieved precision, depicted in bins of 10 observations for simulations of the UCB1 policy compared to the ε-greedy alternative.

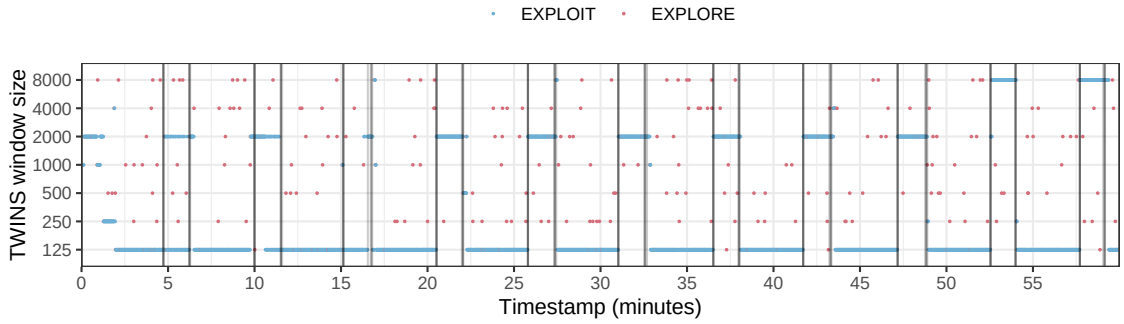


Source: Author

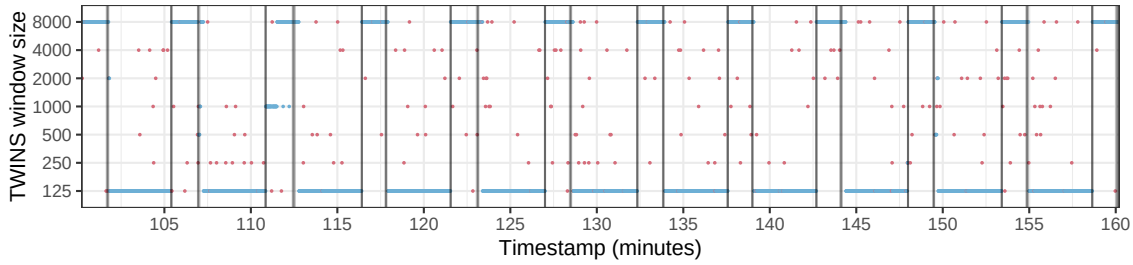
visualize the impact of learning.

We executed the benchmark (interleaving write and read phases) while keeping track of the achieved bandwidth, council metrics, and the selected windows. Figure 4.6 illustrates this exploration/exploitation process using the ε-greedy policy, by depicting the decisions took during the first and the last 60 minutes of the whole experiment. The vertical gray lines denote a change in the detected access pattern (in this scenario, between read and write phases). We can see the algorithm stabilizes at the choice of 125 µs as the

Figure 4.6: The selected window sizes during the online adaptation experiment with ε-greedy. The gray lines separate the write (wide) and read (narrow) phases. It is crucial to notice that there is a single choice at each second, i.e., there are no overlapping decisions (points), despite what the scale of the plots might suggest.



(a) First 60 minutes of execution



(b) Last 60 minutes of execution

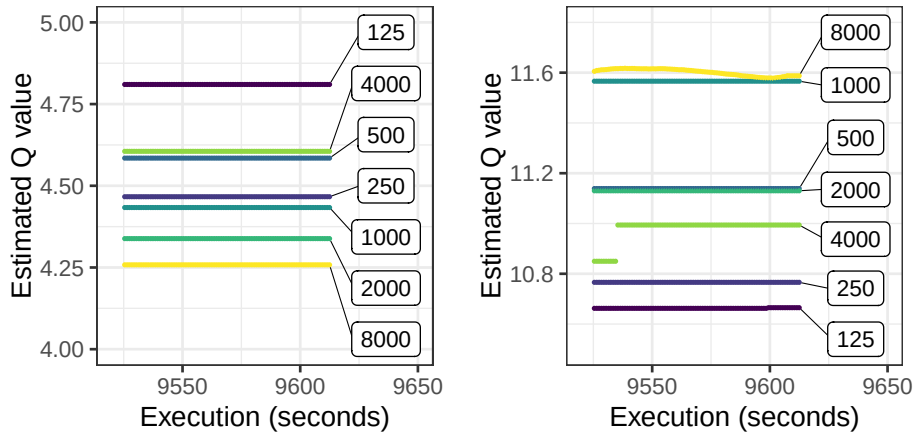
Source: Author

best TWINS window duration for write phases (the wide ones) from the first execution onward. In contrast, the choice of $8\,\mathrm{ms}$ for read phases (the narrow ones) does not happen in the first nine executions. Figure 4.7 depicts the value estimates for the possible actions (window duration) towards the end of the experiment for write (Figure 4.7(a)) and read (Figure 4.7(b)) access patterns.

Nonetheless, we still keep exploring at the same rate as we do not decrease ϵ over time. It is important to remember that at the beginning of these experiments, the system has value estimates of zero for all actions. The depicted learning process only has to happen once. The system will then be able to improve performance not only for the same application but for all applications that share the same access pattern.

Figure 4.8 presents the performance observed by the client (the bandwidth) during repeated executions. The red dashed line represents the first run, the green line indicates the highest achieved bandwidth (best result), and the blue line presents the trend obtained with linear regression. Our approach was able to reach a choice for the write access pattern (Figure 4.8(a)) already on the first execution of the benchmark (in the first $260\,\mathrm{s}$) and yielded good results for the next ones. This learning process was slower for the

Figure 4.7: Estimates for the actions (window sizes) at the end of the online experiment. Each action is shown by its value.



(a) WRITE (b) READ

Source: Author

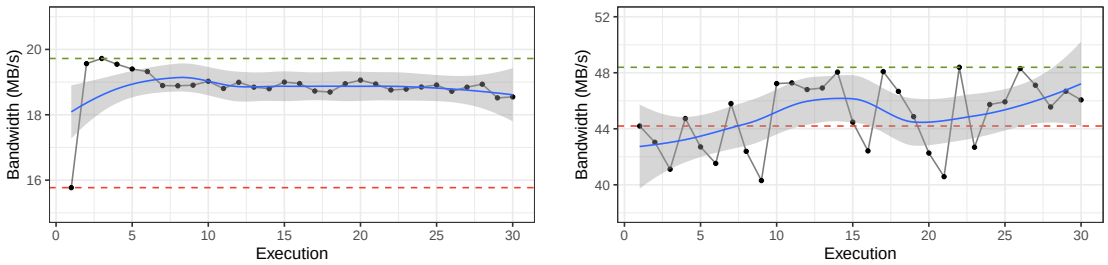
read phases (Figure 4.8(b)) for many reasons. First, read phases are $\approx 60\%$ shorter and consequently comprised of fewer learning iterations. Secondly, it is important to notice that consecutive write/read phases are separated by read/write ones. There is a delay of at least one second (due to the interval in which metrics are reported) before detecting the new access pattern and acting accordingly. Until then, we keep using the previous window size, that might not be optimal, which is the case here. Third, the execution time of the read phase of the benchmark usually presents a higher variability, and that is reflected in the metrics observed by the *Council*, which adds noise to the learning. Finally, the performance impact of bad decisions (mainly due to exploration phases) is higher for these read phases than for the write phases. For this scenario, the best window size for write operations is the worst for reads, and the inverse is also true, as depicted by Figure 4.7. Despite these adversities, we can see an increasing trend for the read bandwidth as the system learns to adapt to that I/O workload.

Moreover, as discussed in the previous section, performance improvements are somewhat limited by the exploration phases. In these experiments, with $\varepsilon = 0.07$, the best value will be chosen 93% of the times. Decreasing ε over time would allow for better performance improvements. On the other hand, exploration can be important to adapt to changes in the system (if the best value for the parameter changes over time due to system performance degradation or system updates and upgrades). The choice depends on the situation at hand.

If we were to apply the UCB1 policy instead of the ε-greedy for the same live experiment, we would discover the best window size much faster. However, we would

not explore as much in future actions, as depicted by the selected windows during the experiment in Figure 4.9. Unlike the previous policy, the exploration or exploitation action is now given by Equation 4.2. Once more, for the write portion of the benchmark, we find the best window size in the application's first execution. For the read phases, we need more observations for the reasons previously described. Nevertheless, we learn it much faster (in the second run of the application) than when using the ε-greedy policy by giving

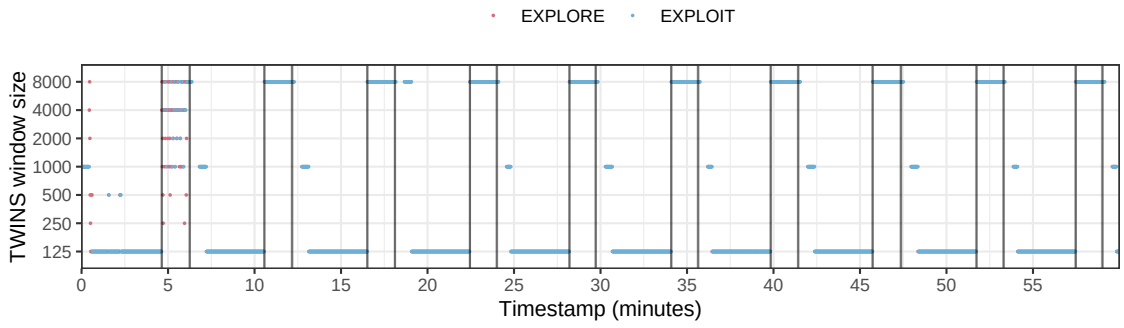
Figure 4.8: Bandwidth of the benchmark during the learning process. The red dashed line shows the execution time of the first iteration, the green line indicates the shortest execution time, and the blue one presents the trend obtained through linear regression. The y-axis is different in each plot.



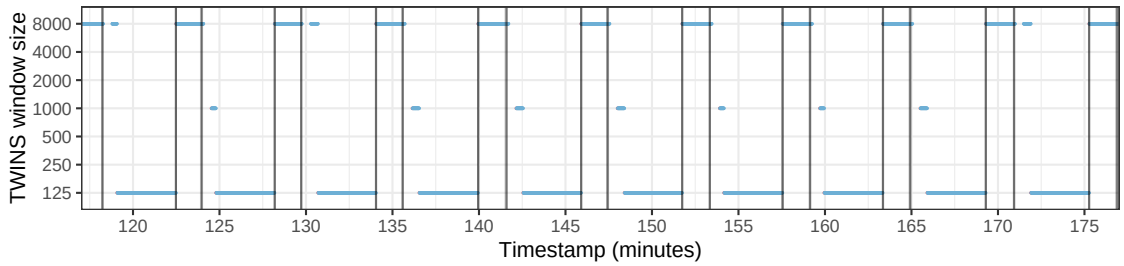
(a) `WRITE` phases (b) `READ` phases

Source: Author

Figure 4.9: The selected window sizes during the online adaptation experiment with UCB1. The gray lines separate the write (wide) and read (narrow) phases. It is crucial to notice that there is a single choice at each second, i.e., there are no overlapping decisions (points), despite what the scale of the plots might indicate.



(a) First 60 minutes of execution



(b) Last 60 minutes of execution

Source: Author

up exploration opportunities as the use of the natural logarithm in UCB1 means that the increases get smaller over time, but are unbounded. UCB1 will eventually select all actions, but actions with lower value estimates or that have already been selected frequently for exploration purposes will receive a decreasing priority over time. In the next section, we choose to restrict our evaluation to the ε-greedy policy as it will better reflect changes due to its frequent exploration.

4.2.4 Results with MADspec

We also evaluated our approach using the MADbench2 (Borrill et al., 2007) I/O kernel extracted from the MADspec application. MADbench2 allows testing the integrated performance of the I/O, communication, and calculation subsystems of massively parallel architectures under the stresses of a real scientific application. It is derived directly from a large-scale Cosmic Microwave Background (CMB) data analysis package. It calculates the maximum likelihood angular power spectrum of the CMB radiation from a noisy pixelized map of the sky and its pixel-pixel noise correlation matrix.

The application has three component functions, each with different access patterns, named *S*, *W*, and *C*. Table 4.5 describes them. In our evaluation we used $N_p = 256$, $N_{pix} = 1280$, $N_{bin} = 80$, and $N_{gang} = 1$. N_p defines the number of processes. N_{pix} sets the size of the pseudo-data, where all the component matrices have $N_{pix} \times N_{pix}$ elements. N_{bin} sets the size of the pseudo-dataset composed on N_{bin} component matrices. Finally, N_{gang} sets the level of gang-parallelism allowing the MADbench2 to run as a single or multi-gang. In former all the matrix operations are carried out distributed over all of the processors. The application uses the MPI-IO interface to issue its I/O operations to a single shared file synchronously.

Table 4.5: I/O characteristics of the MADcode.

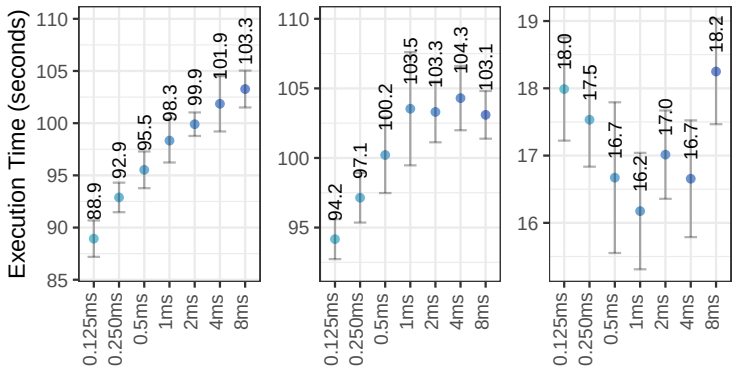
	Input / Output
S	N_{bin} writes each of ${N_{pix}}^2$ bytes on N_p processors.
W	N_{bin} reads each of ${N_{pix}}^2$ bytes on N_p processors. N_{bin} writes each of ${N_{pix}}^2$ bytes on N_p/N_{gang} processors.
C	${N_{bin}}^2/N_{gang}$ reads each of ${N_{pix}}^2$ bytes on N_p/N_{gang} processors.

Source: Author

To find the evaluation baselines, we had to measure the impact of the parameter

value for this application. Hence we conducted an exploratory investigation measuring the execution time when using a fixed TWINS window size during the execution. Figure 4.10 presents the mean of five repetitions for each component using different window sizes. The distance from the average to the extremities of the error bars was calculated using a 95% confidence interval, i.e., $\frac{2 \cdot \sigma}{\sqrt{N}}$ (where σ represents the standard deviation, and N the number of measurements). For the *W* and *S* phases, a small window size yields better results. On the other hand, for *C*, a larger window is better. For the latter, results are less conclusive due to higher variability.

Figure 4.10: Execution time of the *W*, *S*, and *C* components of MADspec with different window duration. The y-axes are different and do not start at zero.



(a) *W* phase (b) *S* phase (c) *C* phase

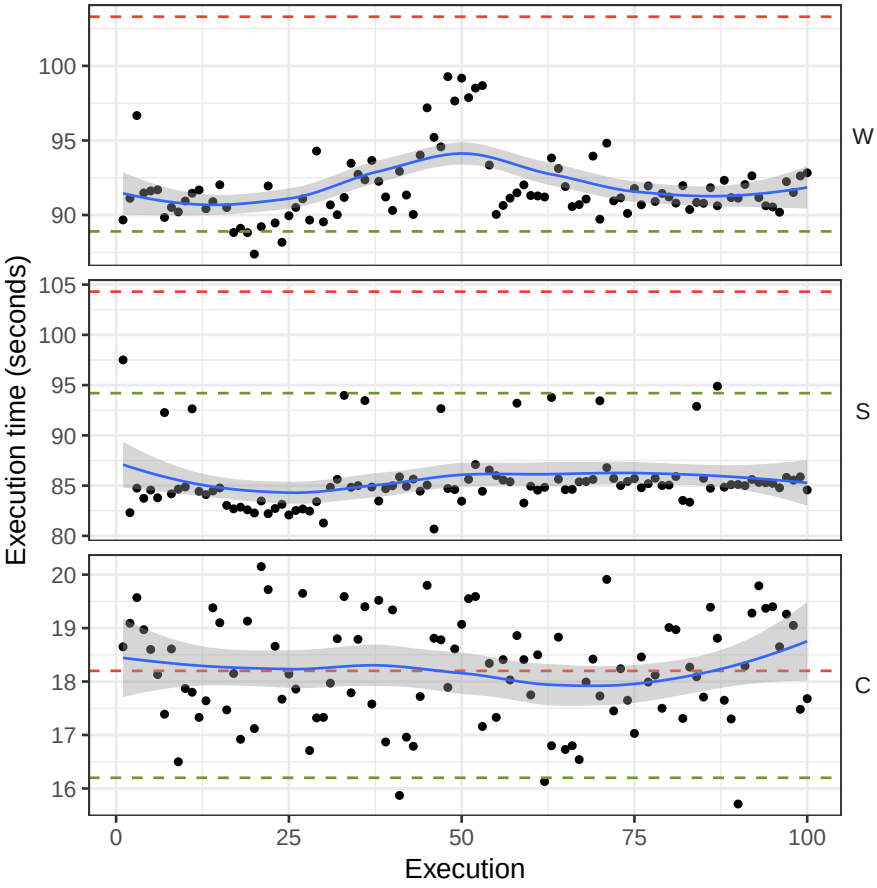
Source: Author

Figure 4.11 presents the three phases' execution time when executed repeatedly while our approach works to adapt the window size. The red and the green dashed lines come from Figure 4.10 and show, for each case, the result previously obtained with the worst and with the best values for the parameter, respectively. For the *W* component, the window choice gets closer to the best (though in some executions, due to the continuous exploration, we get higher times). For *S*, all times are below the previous best because *S* is a mixture of different shorter access patterns. The adaptation mechanism can tune the parameter to them separately, which is better than using a static window for the whole phase. Finally, for *C*, the learning mechanism would require more iterations before successfully adapting due to the read pattern being shorter and more variable.

4.2.5 Overhead and Time-to-decision

In our proposal, an *Announcer* thread in each I/O node interacts with the *Council* located on a remote node. As previously discussed, this configuration is not a requirement

Figure 4.11: Execution time for *W*, *S*, and *C* while adapting the TWINS window size. The dashed lines indicate the previously measured times without adaptation. In red, the worst window size, and in green, the best one for each scenario. The blue line represents the trend using a Local Polynomial Regression Fitting function with 95% confidence.



Source: Author

of our proposal but for our case study. Still, in this section, we analyze its performance.

Our architecture allows for client requests to continue to be received and processed by the I/O nodes. Simultaneously, metrics are asynchronously sent from the *Announcers* to the *Council*. That means the I/O nodes do not stop and wait for new decisions. Therefore, our proposal's overhead is related to the cost of collecting and keeping metrics about the current access pattern and changing the parameter value. To quantify this overhead, we repeated all 144 experiments described in Section 4.2.1 using the proposed architecture but ignoring the decisions. New values for the window parameter are chosen and announced, but TWINS continues to use the same window as before. Therefore we can compare the execution time to a static solution without accounting for performance improvements the adaptation could cause.

Out of the 144 scenarios, we only observed overhead for 65 of them. Table 4.6 summarizes the results for these 65 scenarios. The minimum observed overhead was

0.02%, the median, 1.8%, and the maximum of 32%. The latter, when a single I/O node was used by 512 processes to read contiguously from a shared file using with small requests (32KB). Furthermore, 91.6% of the 144 scenarios have an overhead of less than 5%. We conclude that, in general, our proposal imposes a low overhead (median $< 2\%$).

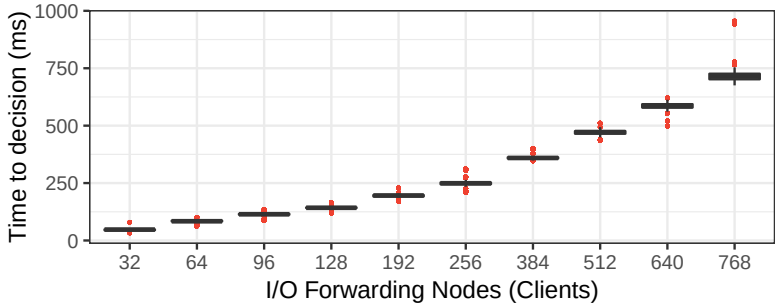
Table 4.6: Overall overhead ($\%$) of our approach for the 144 scenarios, excluding the 79 ones where the overhead was zero.

Min.	**1st Qu.**	**Median**	**Mean**	**3rd Qu.**	**Max.**
0.0199	0.8378	1.7780	3.8354	4.1218	32.2950

Source: Author

We also evaluate the *Council*'s ability to handle a large number of I/O nodes reporting metrics. To do that, we estimate the total time to decision, which is measured by taking the time before the *Announcer* starts to send its metrics up to right after it receives the new parameter value from the *Council*. We executed an increasing number of *Announcer* processes (up to the largest number of I/O nodes in Table 1.1) to report to the centralized *Council* every second. Results (60 measurements of each scenario) are presented in Figure 4.12. Once more, if we consider the TaihuLight, only the 240 I/O nodes would interact with the *Council*. In our experimental environment, for such a scale, the interaction would take < 250ms.

Figure 4.12: Time to decision when I/O nodes are asynchronously reporting metrics every second to the centralized *Council* located on a remote node.



Source: Author

The centralized decision-making agent can work under a heavier workload, with an expected degradation of the time to decision. In this work, we used a somewhat naive communication strategy and a single-threaded prototype of the *Council*, so there is room for improvement. This time does **not** directly impose overhead because, as previously discussed, the adaptation mechanism happens asynchronously, while the I/O node continues to work as expected.

The time to decision is important, however, when selecting the adaptation frequency. It is essential to give the system enough time after adapting to observe an impact on performance. On the other hand, we want to make it as often as possible, so shorter I/O phases can still benefit. In the experiments, the *Announcer* sent metrics one second after the previous decision, and the median time to decision was measured to be of ≈ 130ms. A degraded time to solution would mean a lower adaptation frequency. An alternative, in this case, would be to separate the I/O nodes in groups, and assign one *Council* per group. That would be a compromise where we sacrifice some ability to make global decisions. It is crucial to notice that this discussion only concerns optimization techniques that require global decisions; otherwise, a centralized *Council* is not needed.

4.2.6 Discussion and Limitations

As with every approach, our proposed adaptation for the I/O forwarding layer method has its limitations. We should take the frequency in which we collect metrics and act upon our observations into account. Higher frequency of metrics reports and adaptation decisions might not provide enough data to observe the impact of changes. On the other hand, lower frequencies might miss shorter I/O behaviors in the system and fail to adapt. In our experimental evaluation with TWINS, a one-second observation window proved to be meet the requirements. Furthermore, for the particular case of TWINS, the use of a centralized Council could be considered a single point of failure. Adding a standby node and periodically checkpoint what we have learned so far with newly collected observations could mitigate the impact of failures. In other cases that do not require a consensus on decisions, the learning process is not required to happen in a centralized fashion.

Moreover, for the case of TWINS, we represented the continuous numerical parameter "window size" as a set of discrete values to apply the bandit strategy to our case study. We believe this to be an appropriate strategy because when optimization techniques are proposed, they are evaluated with a set of values for their parameters that developers believe would be reasonable. For instance, for TWINS, it is evident that a window of several milliseconds or a few seconds would cause requests to starve. The real optimal values may lie between classes, but reaching the performance of the best among the observed classes is already an improvement over having no adaptation at all.

Still, if the tuned optimization allows, it could help to provide fewer options to the

armed bandit. That would accelerate learning and could be combined with a smaller value of ε without losing the ability to find the best option quickly. Further, in Section 4.2.2, we attested that having similar values options slows down learning.

Regarding the choice of using the bandwidth as a reward, the caveat is that a low demand can result in a low bandwidth not related to the optimization technique's success. Consequently, that could add noise to the learning process. This strategy was not a problem in our experiments because measurements had the same "intensity" of access. Furthermore, we ignored rewards when no I/O was done, i.e., when the observed bandwidth was zero. To mitigate this problem in practice, we must either take into account load metrics (like the number of bytes requested) or apply some bandwidth normalization. The server-side access pattern classification approach proposed by Boito et al. (2019) solves the issue by accounting for the load.

Another issue might arise if the learning mechanism were to observe too many different access patterns at a given time or have numerous tuning options to choose from, as it would take considerably more time to explore the space and learn the best choice for each pattern and system. Nevertheless, once it learns the best choice for a given I/O pattern, the knowledge we acquired for a given pattern will be used by other applications that share the same pattern.

Finally, for the access pattern detection phase, we could re-train or upgrade the model asynchronously, and once complete, update the model used by the learning mechanism while it keeps learning what the best choice for a given pattern is. That would allow us to promptly expand the number of classes without further modifications in the learning process.

4.3 Final Remarks

Different I/O optimization techniques (including but not limited to the I/O forwarding layer) typically provide improvements for specific system configurations and application access patterns, but not for all of them. Moreover, they often require fine-tuning of parameters. This chapter proposed a novel approach to adapt the I/O forwarding layer to the current I/O workload. By periodically observing access pattern metrics collected by the I/O nodes and applying a reinforcement learning technique – contextual bandits – we demonstrate that the system can learn the best choice for each access pattern at runtime, removing the tuning responsibility from the users.

Our case study was TWINS, a request scheduler that provides improvements over other algorithms, but that depends on selecting the proper window size parameter. Our approach collects metrics on the observed access pattern and periodically use these metrics to detect it. Based on the detected access pattern, a contextual bandit is used to learn the best window to each pattern during execution. This mechanism learns while making decisions and does not require previous training, which is a challenging, error-prone, and time-consuming task because it involves observing all possible patterns and their interactions. Finally, the approach we proposed is not specific to tuning the TWINS window size parameter, and it can be applied to other optimization strategies.

5 DYNAMIC RECONFIGURATION OF I/O FORWARDING LAYER

As briefly discussed in Chapter 1, the forwarding layer is traditionally physically deployed on special nodes, and the mapping between clients and I/O nodes is static. Consequently, a subset of compute nodes will only forward requests to a single fixed I/O node, which ends up forcing applications to use forwarding with a statically pre-defined configuration, even if that decision might not be in the best interest of a given workload. Though this setup seeks to distribute I/O nodes between compute nodes evenly, it lacks the flexibility to adjust to applications' I/O demands, and it can even cause the misallocation of forwarding resources and an I/O load imbalance, as demonstrated by Yu et al. (2017c) on the Sunway TaihuLight and Bez et al. (2020) on MareNostrum 4. In this chapter, we argue in favor of a dynamic allocation of I/O nodes considering the application's workload characteristics and the number of available forwarding nodes.

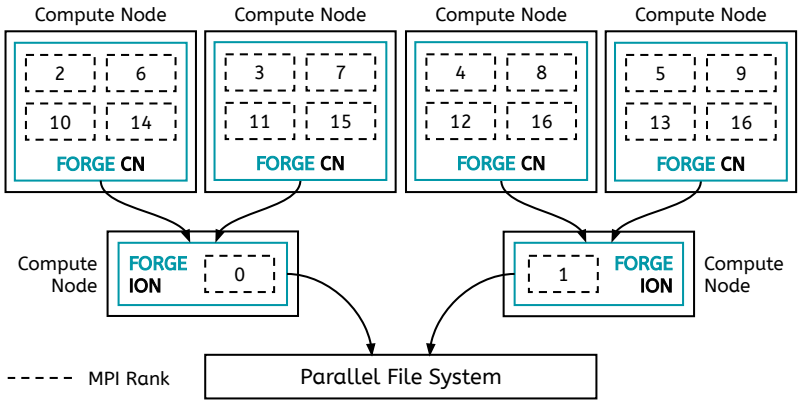
5.1 Impact of I/O Node Allocation

We first seek to investigate the impact of I/O forwarding on performance, taking into account the application's I/O demands (i.e., their access patterns) and the overall system characteristics. We seek to explore when forwarding is the best choice (and how many I/O nodes an application would benefit from), and when not using it is the most suitable alternative. We hope that the design of the forwarding layer in future machines will build upon this knowledge, leading towards a more re-configurable I/O forwarding layer. The Sunway TaihuLight (YANG et al., 2019b) and Tianhe-2A (XU et al., 2014) already provide an initial support towards this plasticity.

Due to the physically static deployment of current I/O forwarding infrastructures, it is usually not possible to run experiments with varying numbers of I/O nodes. Furthermore, any reconfiguration of the forwarding layer typically requires acquiring, installing, and configuring new hardware. Thus, most machines are not easily re-configurable, and, in fact, end-users are not allowed to make modifications to this layer to prevent impacting a production system. We argue that a research/exploration alternative is required to enable obtaining an overview of the impact that different I/O access patterns might have under different forwarding configurations. We seek a portable, simple-to-deploy solution, capable of covering different deployments and access patterns.

Existing I/O forwarding solutions in production today, such as IBM's CIOD (ALMÁSI

Figure 5.1: Overview of the architecture used by FORGE to implement the I/O forwarding technique at user-space. I/O nodes and computes nodes are distributed according to the hostfile configuration.



Source: Author

et al., 2003) or Cray DVS (SUGIYAMA; WALLACE, 2008) require dedicated hardware for I/O nodes and/or a restrictive software stack. They are, hence, not a straightforward solution to explore and evaluate I/O forwarding in machines that do not yet have this layer deployed or that may seek to modify existing deployments. Furthermore, modifications on deployment or tuning of these solutions would have a system-wide impact, even if just for exploration/testing purposes. Since many supercomputers such as the MareNostrum 4 and the Santos Dumont still do not have an I/O forwarding layer, we implemented such layer in user-space to allow sysadmins to evaluate the benefits and impact of using different number of I/O nodes under different workloads, without the need for production downtime or new hardware.

We implement a lightweight I/O forwarding layer that can be deployed as a user-level job to gather performance metrics and aid in understanding the impact of forwarding in an HPC system. The goal of the **I/O Forwarding Explorer** (FORGE) is to evaluate new I/O optimizations (such as new request schedulers) as well as modifications on I/O forwarding deployment and configuration on large-scale clusters and supercomputers. Our solution is designed to be flexible enough to represent multiple I/O access patterns (number of processes, file layout, request spatiality, and request size) derived from Darshan (CARNS et al., 2011). As mentioned, it can be submitted to HPC queues as a normal user-space job to provide insights regarding the I/O node's behavior on the platform. FORGE is open-source and can be downloaded from <https://github.com/jeanbez/forge>.

We designed FORGE based on the general concepts applied in I/O forwarding tools such as IOFSL, IBM CIOD, and Cray DVS. The common underlying idea present in these solutions is that clients emit I/O requests which are then intercepted and forwarded

to one I/O node in the forwarding layer. This I/O node will receive requests from multiple processes of a given application (or even many applications), schedule those requests, and reshape the I/O flow by reordering and aggregating the requests to better suit the underlying PFS. The requests are then dispatched to the back-end PFS, which executes them and replies to the I/O node. Only then, the I/O node responds to the client (with the requested data, if that is the case) whether the operation was successful. I/O nodes typically have a pool of threads to handle multiple incoming requests and dispatch them to the PFS concurrently.

FORGE was built as an MPI application where M ranks are split into two subsets: compute nodes and I/O nodes. The first N processes act as *I/O forwarding servers*, and each should be placed exclusively on a compute node. The remaining $M - N$ processes act as an application, and are evenly distributed between the nodes, allocating more than one process per node if necessary, as depicted by Figure 5.1. The compute nodes issue I/O requests according to a user-defined input file describing the I/O phases of an application. The input file is in the JSON[1] (JavaScript Object Notation) format, a lightweight data-interchange format that is easy for humans to read and write, and easy for machines to parse and generate. All I/O requests of a given compute node are sent by MPI messages to the defined I/O node. The operations are synchronous from the client's perspective, i.e., FORGE waits until the I/O operation to the PFS is complete to reply back to the client.

Each I/O node has a thread that listens to incoming requests and a pool of threads to dispatch these requests to the PFS and reply to the clients. Note that any metadata-related requests, such as those in `open()` and `close()` operations, are immediately executed. For `write()` and `read()` operations, the requests are placed on a queue of incoming requests that are scheduled using the AGIOS (BOITO et al., 2013) scheduling library, aggregated, and dispatched. AGIOS was chosen as it can be plugged into other forwarding solutions, and it contains several available schedulers. Regarding aggregation, in FORGE, requests to the same file and operation are handled in succession. Existing solutions, such as IOFSL, have an interface to issue list operations if the PFS supports it. We opted for a more general approach without relying on such feature being available.

Our experiments were conducted on MareNostrum 4[2] at Barcelona Supercomputing Center (BSC), Spain, and on Santos Dumont[3] supercomputer at the National Laboratory for Scientific Computation (LNCC), Brazil:

[1] https://www.json.org
[2] https://www.bsc.es/marenostrum/marenostrum
[3] https://sdumont.lncc.br

- **MareNostrum 4** supercomputer uses $3,456$ Lenovo ThinkSystem SD530 compute nodes on 48 racks. Each node uses two Intel Xeon Platinum 8160 24C chips with 24 processors each at 2.1 GHz which totals to $165,888$ processes and 390TB of main memory. Each node provides an Intel SSD DC S3520 Series with 240 GiB of available storage, usable within a job. A 100 Gb Intel Omni-Path Full-Fat Tree is used for the interconnection network and a total of 14 PB of storage capacity is offered by IBM's GPFS. There are 7 data servers and 2 metadata servers.

- **Santos Dumont** has a total of $36,472$ CPU cores. The base system has 756 thin compute nodes with two Intel Xeon E5-2695v2 Ivy Bridge 2.40GHz 12-core processors, 64GB DDR3 RAM, and one 128GB SSD. There are three types of thin nodes (BullX B700 family), one fat node (BullX MESCA family), and one AI/ML/DL dedicated node (BullSequana X family). Its latest upgrade added 376 X1120 nodes with two Intel Xeon Cascade Lake Gold 6252 2.10GHz 24-core processors, and one 1TB SSD. 36 of these nodes have a 764GB DDR3 RAM, whereas the remainder have a 384GB DDR3 RAM. Compute nodes are connected to the LustreFS directly through a fat-tree non-blocking Infiniband FDR network. Each OSS has one OST made of 40 6TB HDD disks in a RAID6 (same for the MDS–MDT).

We explore FORGE with multiple access patterns and forwarding deployments in MareNostrum in Section 5.1.1. We expand our analysis using a subset of patterns in Santos Dumont in Section 5.1.2. We covered 189 scenarios, with at least 5 repetitions of each, considering the following factors:

- 8, 16, and 32 compute nodes;
- 12, 24, and 48 client processes per compute node;
- 96, 192, 384, 768, and 1536 processes depending on the scenario;
- File layout: file-per-process or shared-file;
- Spatiality: contiguous or 1D-strided;
- Operation: writes with `O_DIRECT` enabled to account for caching effects;
- Request sizes of 32KB, 128KB, 512KB, 1MB, 4MB, 6MB, and 8MB synchronously issued until a given total size is transferred or a stonewall is reached.

As the compute nodes of both supercomputers' have 48 cores per node, we evaluate a scenario where an application uses all, half, or a quarter of its cores. Regarding file layout and spatiality, we opted for configurations commonly tested with benchmarks such as IOR and MPI-IO Test.

Table 5.1: Access patterns described with FORGE for the experiments executed on the MareNostum (Figure 5.3(a)) and SDumont (5.4) supercomputers.

#	Nodes	Processes	File Layout	Request Spatiality	Request (KB)
A	32	1536	File-per-process	Contiguous	1024
B	32	1536	File-per-process	Contiguous	128
C	32	1536	Shared	Contiguous	1024
D	32	1536	Shared	Contiguous	4096
E	32	1536	Shared	1D-strided	512
F	16	192	Shared	Contiguous	32
G	16	192	Shared	1D-strided	128
H	8	192	File-per-process	Contiguous	8192
I	8	192	Shared	1D-strided	8192
J	16	384	Shared	Contiguous	128
K	16	384	Shared	Contiguous	8192
L	32	384	Shared	1D-strided	4096
M	32	384	Shared	1D-strided	512
N	8	384	Shared	Contiguous	4096
O	16	768	Shared	Contiguous	1024
P	32	768	Shared	Contiguous	1024
Q	16	768	Shared	1D-strided	1024
R	8	96	File-per-process	Contiguous	8192
S	8	96	Shared	1D-strided	6144
T	8	96	Shared	Contiguous	512

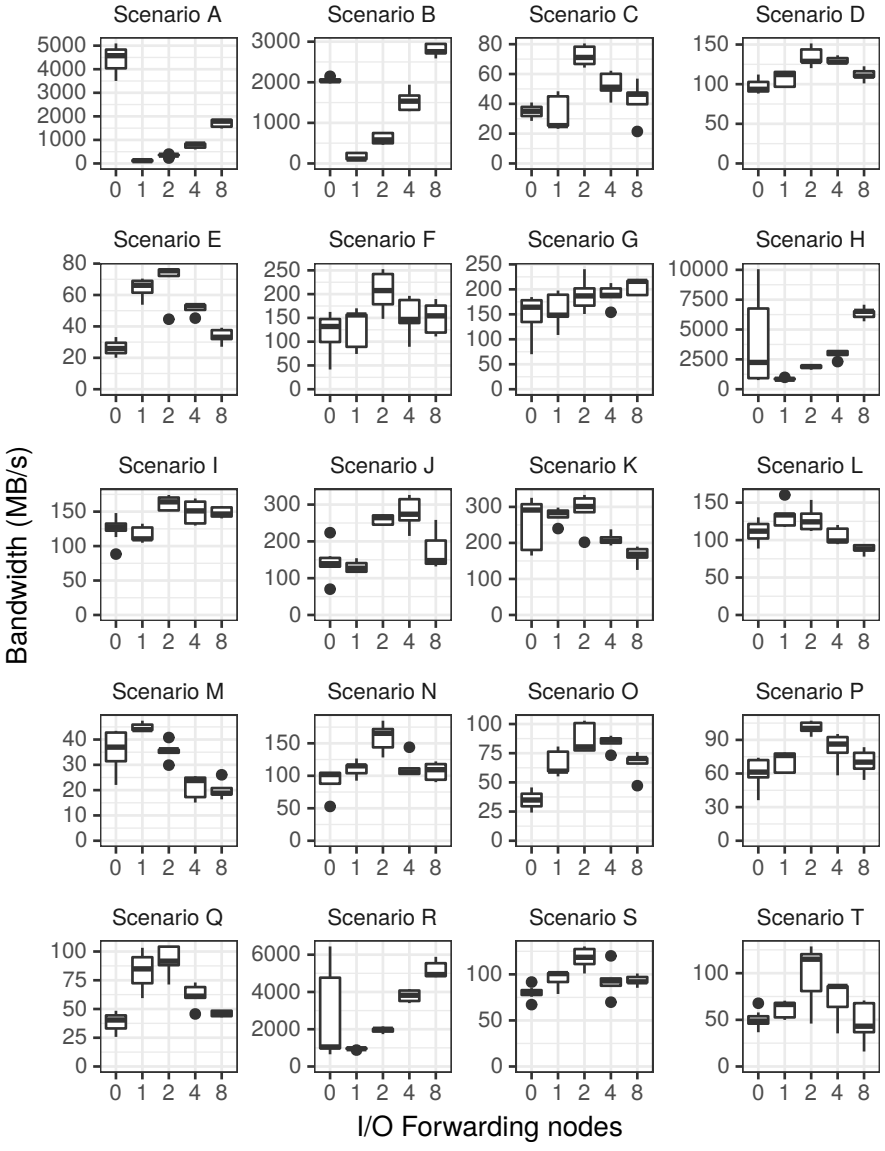
Source: Author

5.1.1 I/O Forwarding on MareNostrum 4

Figure 1.1 depicts the bandwidth measured at client-side (makespan), when multiple clients issue their requests following each access pattern and taking into account the number of available I/O nodes (0, 1, 2, 4, and 8). Each experiment was repeated at least 5 times, in random order, and spanning different days and periods of the day. Table 5.1 describes each depicted pattern. In Figure 1.1, only for scenario **A** directly accessing the PFS translates into higher I/O bandwidth. For **L** and **M**, the right choice would be to allocate one I/O node each. Patterns **J** and **O** would achieve higher bandwidth when four I/O nodes

are given to the application. On the other hand, eight are required by scenarios **B**, **G**, **H**, and **R**. For the remaining scenarios, two I/O nodes is the ideal choice. The complete evaluation of the 189 experiments is available at <https://doi.org/10.5281/zenodo.4016899>.

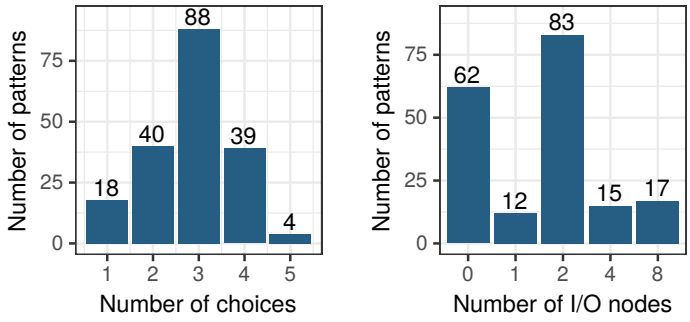
Figure 5.2: I/O bandwidth of distinct write access patterns and I/O nodes in the MareNostrum 4 supercomputer. The y-axis is not the same.



Source: Author

For each one of 189 scenarios, it is possible to determine the different options an application (or access pattern) has to choose regarding how many I/O nodes it could use. For options that translate into similar achieved bandwidth, the smallest number of I/O nodes is the most reasonable choice. To verify how many choices we have for each pattern, we compute Dunn's test (DUNN; DUNN, 1961) for stochastic dominance, which reports the results among multiple pairwise comparisons after a Kruskal-Wallis

Figure 5.3: (a) Number of choices each access pattern has which translates into statistically distinct I/O performance. (b) Access patterns grouped by the number of I/O nodes that would translates to the best performance.



(a) (b)

Source: Author

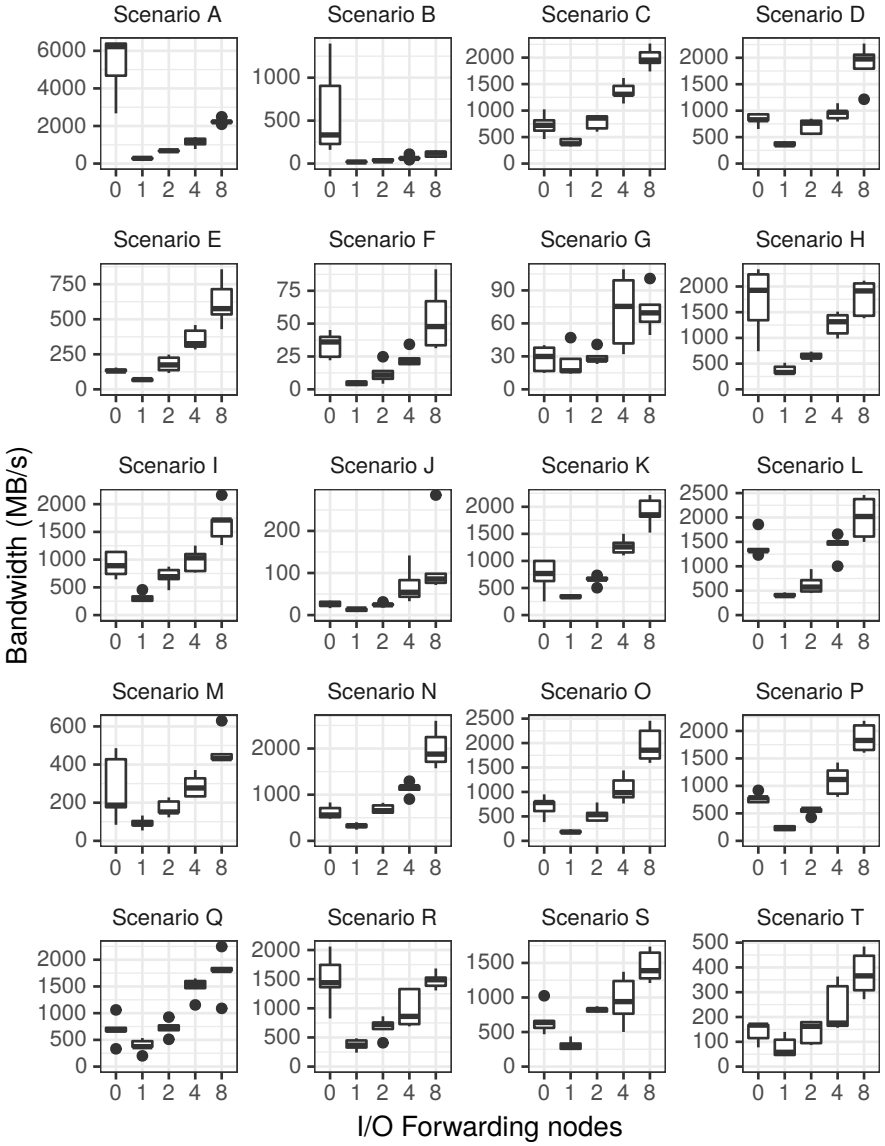
(KRUSKAL; WALLIS, 1952) test for stochastic dominance among groups. Dunn is a non-parametric test that can be used to identify which specific means are significant from the others. The null hypothesis (H_0) for the test is that there is no difference between groups. The alternate hypothesis is that there is a difference. We used a level of significance of $\alpha = 0.05$, thus we reject H_0 if $p \leq alpha/2$.

Not using forwarding is always an option to be considered as it yields different bandwidth than when using forwarding, though it might not be the best choice for many cases. Figure 5.3(a) illustrates how many options an allocation policy would have to consider. For 88 ($\approx 46\%$) of the patterns, three possible choices would impact performance.

The optimal number of I/O nodes for each of the 189 scenarios, considering the available choices of I/O nodes, is different, as depicted by Figure 5.3(b). For 12 (6%), 83 (44%), 15 (8%), and 17 (9%) scenarios, the largest bandwidth is achieved by using 1, 2, 4, and 8 I/O nodes respectively. Whereas, for 62 scenarios (33%), not using forwarding is instead the best alternative. There does not seem to be a simple rule to fit all applications and system configurations, which is to be expected given the complexity of factors that can influence I/O performance. For instance, consider patterns **A** and **B** in MareNostrum. The size of requests makes not using forwarding the best for pattern **A**, whereas for **B**, using 8 I/O nodes translates more bandwidth. For patterns **A** and **C**, where the only difference is the file layout, **C** should use 2 I/O nodes. In Santos Dumont, 8 I/O nodes should be chosen for **C** instead of 0 as in **A**.

Considering the static physical setup of platforms where forwarding is present, applications often are not allowed to have direct access to the PFS other than by an I/O node. We can compare the maximum attained bandwidth one could achieve by forcibly

Figure 5.4: I/O bandwidth of distinct write access patterns and I/O nodes in the Santos Dumont supercomputer. The y-axis is not the same.



Source: Author

using forwarding in all the cases (and the optimal number of I/O nodes) to not using this technique. For 90.5% (171 scenarios) using forwarding, for 9.5% (18 scenarios), directly accessing the PFS would instead increase performance. Hence, both options should be available to applications to optimize I/O performance properly.

5.1.2 I/O Forwarding on Santos Dumont

We conducted a smaller evaluation on Santos Dumont (due to allocation restrictions) comprised of 20 patterns described in Table 5.1. Figure 5.4 depicts the I/O band-

width of distinct write access patterns with a varying number of I/O nodes. One can notice that the forwarding layer setup's impact is not the same on MareNostrum and SDumont.

A common behavior observed in Santos Dumont is that using more I/O nodes yields better performance. Regardless, the choice of using forwarding or not in this machine is still relevant. For instance, for the scenarios **A**, **B**, and **H**, direct access to the Lustre PFS translates into higher I/O bandwidth. Whereas, for some other scenarios, the benefits of using forwarding are perceived by the application after more than one I/O node is available. It is possible to notice that using a single I/O node is not the best choice for none of the tested patterns. If two were available, scenarios **C**, **E**, **N**, **Q**, and **S** would already see benefits. For **D**, **G**, **I**, **J**, **K**, **L**, **M**, **O**, **P**, and **T**, only when four I/O nodes are given to the application, in this machine, performance would increase. Other patterns, such as **F** and **R**, require eight I/O nodes instead to achieve the best performance. In practice, there will be an upper bound on how many of these resources are available to applications. This information could guide allocation policies to arbitrate the I/O nodes by allocating the resources to those that would achieve the highest bandwidth.

5.1.3 Discussion

As HPC systems tend to get more complex to accommodate new performance requirements and different applications, the forwarding layer might be re-purposed from a must-use to an on-demand approach. This layer has the potential to not only coordinate the access to the back-end PFS or avoid contention but also to transparently reshape the flow of I/O requests to be more suited to the characteristics and setup of the storage system. Our results demonstrate the intuitive notion that access patterns are impacted differently by using forwarding and that the choice in the number of I/O nodes also plays an important role. Consequently, taking into account this information, novel forwarding mechanisms could benefit from this knowledge to allocate I/O nodes based on the demand. Our initial experiments with FORGE demonstrate that an idle compute node (or even a set of compute nodes) could be temporarily allocated to act as a forwarder and improve application and system performance.

Furthermore, as I/O nodes could be given only to applications that would benefit the most from them, interference on sharing these resources would be reduced or even eliminated. Instead, in today's setups, I/O nodes have to handle all requests from a subset of compute nodes, where concurrent applications (with very distinct characteristics) could

be running. Yildiz et al. (2016) demonstrate the impact of interference in the PFS, and Bez et al. (2017) attested this when forwarding is also present. Moreover, Yu et al. (2017c) investigated the load imbalance on the forwarding nodes, where recruiting idle I/O nodes would be the solution. Instead, we argue that a shift of focus from this layer's physical global deployment to the I/O demands of applications should instead guide this layer's future usage and distribution. Hence, I/O forwarding could be seen as an on-demand service for applications that would benefit from it, possibly recruiting temporary I/O nodes to avoid interference present when sharing these resources.

5.2 Problem Statement

As demonstrated in previous sections, there does not seem to be a simple rule regarding the number of I/O nodes to fit all applications and system configurations, which is to be expected given the complexity of factors that can influence I/O performance. Furthermore, some patterns seem to benefit the most from having access to more I/O nodes than others. Consequently, a static mapping of I/O nodes to compute nodes without considering an application's workload does not always result in the best performance. Hence, the need for appropriate allocation policies that take into account these issues to maximize *globally-perceived* I/O performance.

Since no simple rules allow to allocate I/O forwarding resources to best suit all applications, I/O node arbitration needs to be considered as an optimization problem. As an optimization problem, the I/O node allocation may informally be thought of as the following: given a set of jobs to run and a fixed number of I/O nodes, determine how many forwarding nodes each of them should receive to maximize the aggregated global bandwidth. Thus, every time the set of running applications changes, the decisions have to be reevaluated. Based on these requirements, this section describes our proposed solution to the allocation problem. To prove that we are on the right track, we evaluate its pre-implementation by measuring the achieved I/O performance through simulation, comparing it to different baselines.

5.3 The Multiple-Choice Knapsack Problem (MCKP) Allocation Policy

The Multiple-Choice Knapsack Problem (MCKP) is an optimization problem, derived from the 0-1 Knapsack, where the items are subdivided into k classes, each having N_i items. The binary choice of taking an item in the 0-1 problem is replaced by selecting exactly one item from each class. Formally, the problem is described by (5.1). Where the variables x_{ij} take on value 1 if and only if item j is chosen in class N_i. While the problem is $\mathcal{NP}$-hard, the time complexity of its Dynamic Programming solution, which is pseudo-polynomial, is $O(W \sum_{i=1}^{k} N_i)$.

$$\begin{aligned} \text{maximize } & \sum_{i=1}^{k} \sum_{j \in N_i} p_{ij} x_{ij} \\ \text{subject to } & \sum_{i=1}^{k} \sum_{j \in N_i} w_{ij} x_{ij} \leq W, \\ & \sum_{j \in N_i} x_{ij} = 1, \forall i \in \{1, ..., k\} \\ & x_{ij} \in \{0, 1\}, \forall i \in \{1, ..., k\}, \forall j \in N_i. \end{aligned} \tag{5.1}$$

We chose to model our problem after the MCKP as a global goal (i.e., the aggregated bandwidth) needs to be maximized based on a set of options available to choose from (i.e., the number of I/O nodes an application could use). We focus on optimizing how many forwarding nodes an application should use rather than where it should run.

For the I/O node allocation policy, each class represents an application, and the items of a class denote the number of I/O nodes that the application could use. These items can be different for each class, as long as the number of compute nodes used by the application is divisible by the number of I/O nodes. This constraint is to improve load balancing. Furthermore, it should be limited by the total number of I/O nodes. The weight w_i of each item represents the number of I/O nodes, and the value p_i the bandwidth. We must pick one choice for each application, seeking to maximize the global bandwidth, taking into account a pool of available forwarders, represented by the capacity W.

We assume that we have information about an application's I/O performance using different numbers of forwarding nodes. One can obtain this data from exploratory executions, though more detailed information can also be extracted from Darshan (CARNS et al., 2011) traces, which are already transparently collected at many supercomputers.

These traces can be used to identify the base access patterns (e.g., file approach, spatiality, and request sizes), the number of processes making I/O requests, and the total transferred data volume. Combined with performance metrics of short benchmark runs using those base I/O patterns with different number of I/O nodes allows us to estimate the complete application's I/O performance. Details of such approach are described by Boito (2020), as we focus on the allocation policy and not on application profiling. Thus, it would not be required for each application to be run using different forwarding setups in order to attain the necessary information. When no such application data is available, i.e., on its first execution, MCKP is provided with the default number of I/O nodes the application would receive for that particular system setup, hence avoiding a negative impact.

In our experiments, we allow applications to not use forwarding, which implies no need for sharing I/O nodes as the number of available forwarding resources is always enough. An additional option could be given to the applications to accommodate a scenario where sharing is inevitable: using a shared I/O node. Nevertheless, we seek to avoid that whenever possible as it brings performance interference. When considering sharing, we use a naive estimation based on the bandwidth of using a single node (for that application) divided by the total number of running applications. There are two caveats to this approach. The first is that estimating the impact of interference is not simple. The second is that the number of applications sharing the I/O node will be smaller than the number of running applications. Nonetheless, such an estimate does not present an issue as it will be a low-bandwidth option that the policy will take only for less performant applications. The remaining $(N - 1)$ forwarding nodes could then be given to MCKP to arbitrate, as one I/O node is reserved for shared allocations.

5.4 Evaluation of MCKP Applicability

From the 189 patterns executed at the MN4 machine, as described in Section 5.1, we randomly sampled sets of 16 to simulate each policy considering N available I/O nodes. For this experiment, each pattern is an application that is ready to run. We generated $10,000$ sets, to cover multiple combinations of those patterns running at the same time, having up to 128 forwarding nodes to allocate among the 16 applications – eight per application, the maximum I/O node number for which we have results. In those sets, the median number of compute nodes used by all applications was 256, with a minimum of 88 and a maximum of 512 nodes. Results are then obtained by Equation 5.2, taking the

sum of the 16 applications' bandwidth. The W and R in the equation represent the total transferred size by write and read operations for each application a.

$$aggregate\ BW = \sum_{a=1}^{16} \left(\frac{W_a + R_a}{runtime_a} \right) \tag{5.2}$$

We compare our MCKP solution to alternative policies:

- **ZERO** and **ONE Policies**: each application is assigned zero or one I/O nodes. These policies demonstrate the initial impact of using I/O forwarding.

- **STATIC Policy**: the total number of I/O nodes is divided between the applications based on the number of compute nodes (CN) each one requires. The number of I/O nodes is given by $ceil(\frac{C}{R})$, where C is the number of compute nodes required by the application and $R = \frac{CN}{F}$. F is the number of I/O nodes in the system. This is the policy used by some supercomputers that have forwarding.

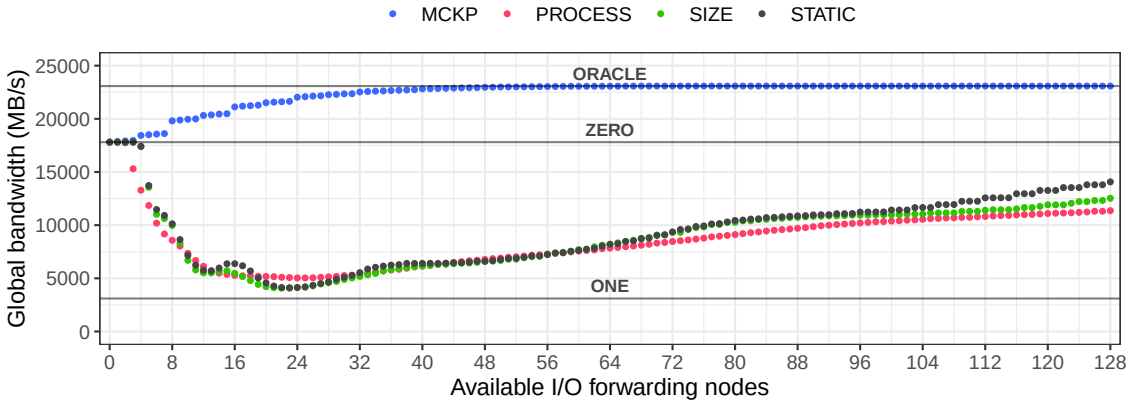
- **SIZE** and **PROCESS Policies**: the total number of I/O nodes is proportionally divided between the running applications based on their sizes (number of compute nodes or processes). The number of I/O nodes is equal to $round\left(F \times \frac{c_a}{\sum_{a=0}^{A} c_a}\right)$, where F is the number of forwarding nodes, A the number applications, and c_a the size of application a.

- **ORACLE Policy**: each application is assigned the number of I/O nodes that achieved the highest bandwidth, obtained from our performance evaluation. This is a *fictitious policy* that disregards the limited number of I/O nodes in the system. It is intended to provide an upper bound of the gains.

If we consider the ONE policy, where we allocate a single non-shared I/O node to each application and compare it to not using forwarding at all (ZERO policy), in our simulations, we observed a median slowdown of 82.11%. As the majority of the tested workloads benefit from using more than a single I/O node, or not using I/O forwarding at all, this policy is not well suited. On the other hand, if we compare the ZERO to the ORACLE policy, it is possible to better grasp the potential improvements of using forwarding. In this comparison, we observed a minimum performance boost of 0.83% and a maximum of 121.68%. The median was of 25.63% just by using forwarding correctly.

Figure 5.5 compares the median aggregated bandwidth, computed by Equation 5.2, of the $10,000$ experiments for each number of I/O forwarding nodes using the arbitration policies. The MCKP policy achieves the same aggregated bandwidth as the

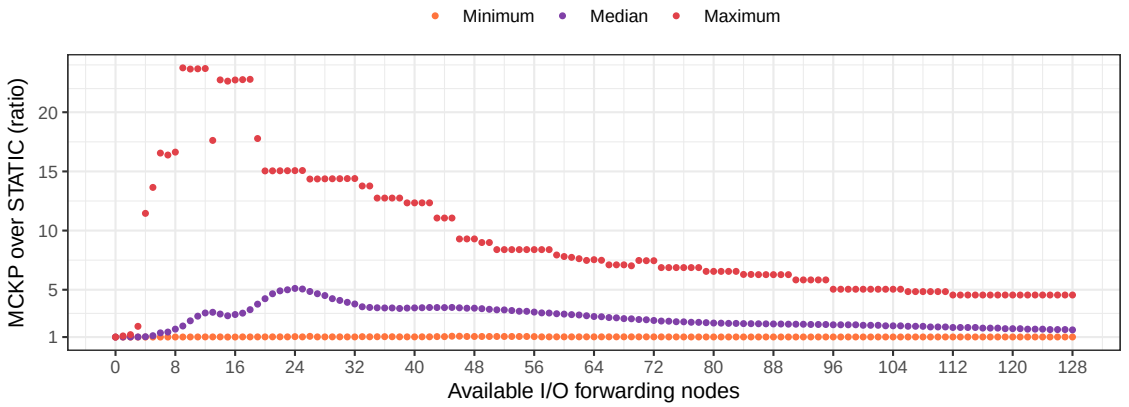
Figure 5.5: Median global bandwidth observed in the $10,000$ sets of 16 randomly selected applications from the 189 scenarios collected at MN4 supercomputer.



Source: Author

ORACLE policy when 56 forwarding nodes are allocated between the 16 random running applications. However, the ORACLE is not limited by the number of available I/O nodes. Moreover, it demonstrates that allocating I/O nodes solely based on application size (i.e., number of required compute nodes or processes) is not the best solution. Compared to the STATIC policy, when the optimal number of 56 I/O nodes is available, the MCKP policy achieved a minimum performance boost of 4.08% and a maximum of 739.22%. The median was 211.38%.

Figure 5.6: Improvement of MCKP compared to STATIC policy observed in the $10,000$ sets of 16 applications randomly selected from the 189 scenarios collected at MN4, with different numbers of I/O nodes.



Source: Author

5.5 GekkoFWD: On-Demand I/O Forwarding

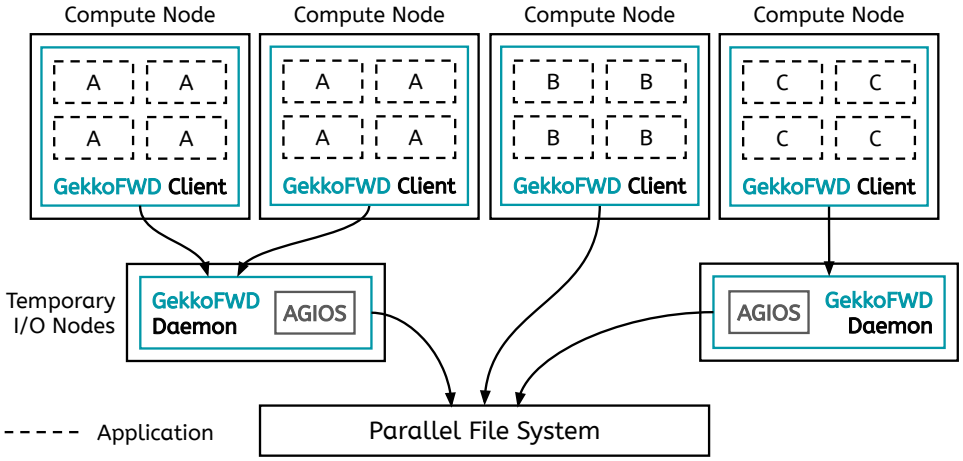
Though FORGE allows replaying I/O profiles from applications to rapidly explore different I/O forwarding deployments, it lacks the support to actually run applications themselves. Besides, once an I/O node mapping selected, it is impossible to dynamically change the number of allocated I/O nodes at runtime. Therefore, we propose a full-fledged user-level I/O forwarding solution that is adequate and easy to run in production machines. To achieve this goal, we enriched an existing ad-hoc file system called GekkoFS (VEF et al., 2018; VEF et al., 2020) by implementing a *forwarding mode* for it. GekkoFS creates a temporary file system on compute nodes using their local storage capacity as a burst-buffer to alleviate I/O peaks. It ranked 4^{th} in the overall 10-node challenge of IO500[4] in November 2019, as well 2^{nd} concerning metadata performance in the same challenge.

GekkoFS uses the local storage available on compute nodes to provide a global namespace accessible to all participating nodes. The **GekkoFWD** extension mode modifies it to use the shared PFS (e.g., Lustre, GPFS) for storage instead. Moreover, data operations in GekkoFS are typically distributed across all nodes using the Mercury HPC RPC framework (SOUMAGNE et al., 2013). Once an I/O operation is intercepted, the client forwards that request to the responsible server, determined by hashing the file's path. To achieve a balanced data distribution, each file is split into equally sized chunks by the client and distributed among the servers. Conversely, GekkoFWD enables GekkoFS servers to act as intermediate I/O nodes between the compute nodes and the PFS data servers. To achieve this, we leverage the system call interception infrastructure in each GekkoFS client to transparently capture all application I/O requests in each compute node. Then GekkoFWD forwards those requests to a single server, which will now act as an I/O node as determined by a pre-defined allocation policy. To conform to such a policy, we included a thread in the GekkoFS client that checks for mapping updates and respond to any modification.

Since forwarding layers are transparent to applications, they usually are a perfect target to implement I/O optimizations such as file-level request scheduling (VISHWANATH et al., 2010; OHTA et al., 2010; BEZ et al., 2017). For that reason, we integrated the AGIOS scheduling library into GekkoFWD. AGIOS provides several schedulers, giving GekkoFWD the flexibility to prototype new scheduling solutions. Once a request is received by a GekkoFWD I/O node, it is sent to AGIOS to determine when

[4] <https://www.vi4io.org/io500/list/19-11/10node>

Figure 5.7: GekkoFWD deployment uses an interception library at the client side and a daemon on the nodes that will act as temporary I/O nodes.



Source: Author

it should be processed. Once scheduled, it is then dispatched to the PFS and executed following the normal flow of requests in GekkoFS. Figure 5.7 depicts a deployment of GekkoFWD, where some applications use forwarding and others do not, depending on the pre-selected policy. GekkoFWD is open source and is available in the official GekkoFS[5] under an MIT license.

5.6 Experimental Evaluation

Since we needed fine control on allocation decisions, our evaluation was conducted on the Grid 5000 (G5K) (BALOUEK et al., 2013) platform, a large-scale testbed for experiment-driven research. Our experiments used two clusters from the Nancy site: Grimoire (8 nodes) and Gros (124 nodes). Grimoire nodes are powered by an Intel Xeon E5-2630 v3 processor (Haswell, 2.40 GHz, 2 CPUs per node, 8 cores per CPU) and 128 GB of memory. The Lustre parallel file system servers deployed in Grimoire nodes use a 600 GB HDD SCSI Seagate ST600MM0088. Gros nodes are powered by an Intel Xeon Gold 5220 processor (Cascade Lake-SP, 2.20 GHz, 1 CPU/node, 18 cores/CPU) and 96 GB of memory. Each node of Gros is connected to two switches with 2×10Gbps Ethernet links. The two switches are connected to another one with 2×40Gbps links each. The latter is connected to Grimoire's nodes with 4×10Gbps Ethernet to each node.

[5] <https://storage.bsc.es/gitlab/hpc/gekkofs/-/releases>

5.6.1 Application

To demonstrate the applicability of MCKP under mixed I/O workloads, we ran five different application kernels and the IOR[6] micro-benchmark on top of GekkoFWD. We briefly describe them in the following items, and highlight their access patterns.

Table 5.2: Setup and I/O characteristics of the applications.

Label	Application	Operation	File Approach	Write (GB)	Read (GB)	Total (GB)	Nodes	Processes
BT-C	NAS BT-IO (Class C)	write / read	Single shared file	6.3	6.3	12.6	32	128
BT-D	NAS BT-IO (Class D)	write / read	Single shared file	126.5	126.5	253.0	64	512
HACC	HACC-IO	write	File-per-process	1.8	0	1.8	8	64
IOR-MPI	IOR (MPI-IO)	write / read	Single shared file	16.0	16.0	32.0	16	128
POSIX-S	IOR (POSIX)	write / read	Single shared file	16.0	16.0	32.0	16	128
POSIX-L	IOR (POSIX)	write / read	File-per-process	32.0	32.0	64.0	64	512
MAD	MADBench2	write / read	Single shared file	16.2	16.2	32.4	32	64
SIM	S3DSIM	write	Single shared file	19.6	0	19.6	16	16
S3D	S3D-IO	write	Multiple shared files	33.7	0	33.7	64	512

Source: Author

- The **S3D I/O Kernel** (CHEN et al., 2009b) is a continuum scale first principles direct numerical simulation code which solves the compressible governing equations of mass continuity, momenta, energy and mass fractions of chemical species including chemical reactions. It performs N checkpoints at regular intervals, where it writes three and four-dimensional arrays of doubles into a newly created file. We configured $N = 5$, thus S3D generates five files. All three-dimensional arrays are partitioned among the MPI processes, whereas the fourth dimension (the most significant one) is not partitioned. We configured it to use PnetCDF (LATHAM et al., 2003) nonblocking APIs, where each checkpoint has four nonblocking write calls, followed by a wait and flush the requests (LIAO; CHOUDHARY, 2008).

- **MADBench2** (Borrill et al., 2007) is based on the MADspec code, which calculates the maximum likelihood angular power spectrum of the Cosmic Microwave Background radiation from a noisy pixelized map of the sky and its pixel-pixel noise correlation matrix. It has three I/O component functions, each with different access patterns, named *S*, *W*, and *C*. The *S* component consists of writes by a subset of the processes. In *W*, that data is read back, and a smaller subset writes new data. Finally, in component *C*, that data is read back. MADBench2 uses the MPI-IO interface to issue its I/O operations synchronously to a single shared file.

[6]<https://github.com/hpc/ior>

- **HACC-IO**[7] is an I/O kernel of the Hardware/Hybrid Accelerated Cosmology Code (HACC) application (Gordon Bell Award Finalist 2012, 2013) used to simulate collision-less fluids in space using N-Body. In our tests, we used $N = 100$K particles. Each process writes $N \times 38$ bytes and a 24 MB header using POSIX and a file-per-process.

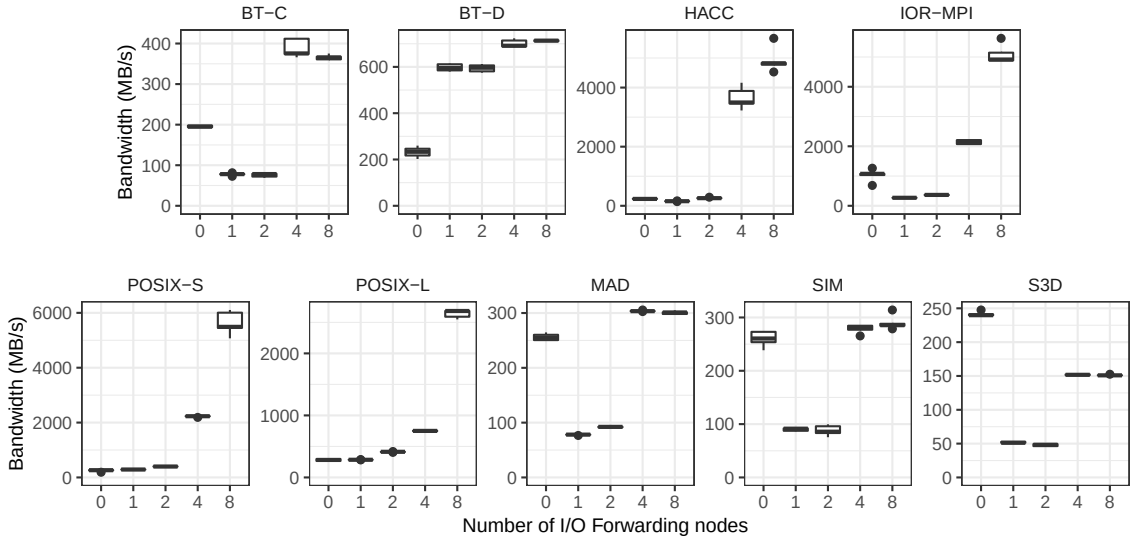
- **S3aSim** (Ching et al., 2006) is a sequence similarity search framework. It uses a parallel programming model with database segmentation, which mimics the mpiBLAST access pattern. Given input query sequences, it divides up the database sequences into fragments. Workers request a query and fragment information from the master and search the query against the database fragment assigned. The results are sent to the master to be sorted and then written to a single shared file. We configured S3aSim to issue 100 queries varying from 1 to 10 to a database sequence of sizes ranging from 6 to $45,088,768$, both using a simple uniform random distribution and 128 fragments. Each query writes from $\approx$ 4MB to 328MB of data, $\approx$ 100MB on average. This application uses individual I/O operations, without synchronizing after writing every query.

- **NAS BT-IO** (NASA Advanced Supercomputing Division, 2003) is based on the Block-Tridiagonal (BT) problem of the NAS Parallel Benchmarks (NPB). After every five time steps, the entire solution, consisting of five double-precision words per mesh point, must be written to file. In the end, all data belonging to a single time step must be stored in the same file and must be sorted by vector component, x, y, and z-coordinate. We used the BT-IO MPI version with collective buffering, where data scattered among the processors is collected on a subset of the participating processors and rearranged before written to file in order of increasing granularity. The C class with 128 processes issues MPI-IO requests of 1.34MB and POSIX requests of 5.23MB, according to Darshan logs. The D class with 512 processes issues larger requests of 5.35MB and 12.31MB, for MPI-IO and POSIX, respectively.

Table 5.2 labels each application based on its configuration. We detail the parameters used to run each one of them in our companion repository. We compute the bandwidth for each application by measuring the application's execution time at the client-side (i.e., the makespan). Figure 5.8 confirms FORGE's results in Figure 1.1, where in neither case there is a single allocation solution that best fits all applications.

[7] <https://github.com/glennklockwood/hacc-io/>

Figure 5.8: I/O bandwidth, measured at client-side, of five repetitions of each application described in Table 5.2. The x-axis represents the number of I/O forwarding nodes exclusively used by the job. The y-axis is not the same for each plot.



Source: Author

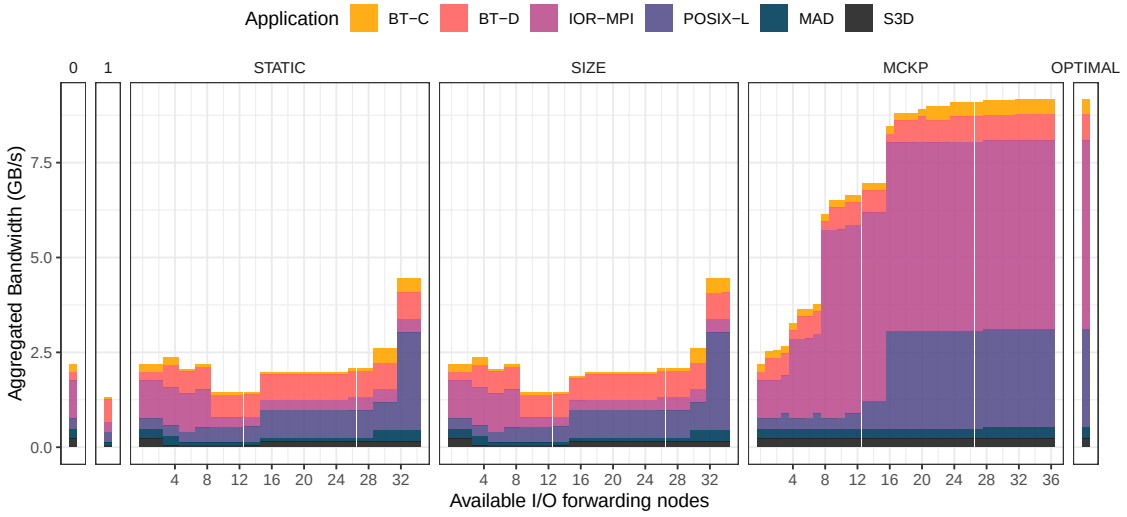
As we executed all applications using C compute nodes and P processes, where both C and P are a power of two, we also considered the number of available I/O nodes each application can use as powers of two. For our evaluation, the policies can choose between 0, 1, 2, 4, and 8 I/O nodes for each application. In practice, these options would comprise numbers divisible by the number of compute nodes used by each application to improve load balancing.

5.6.2 Allocation Decisions

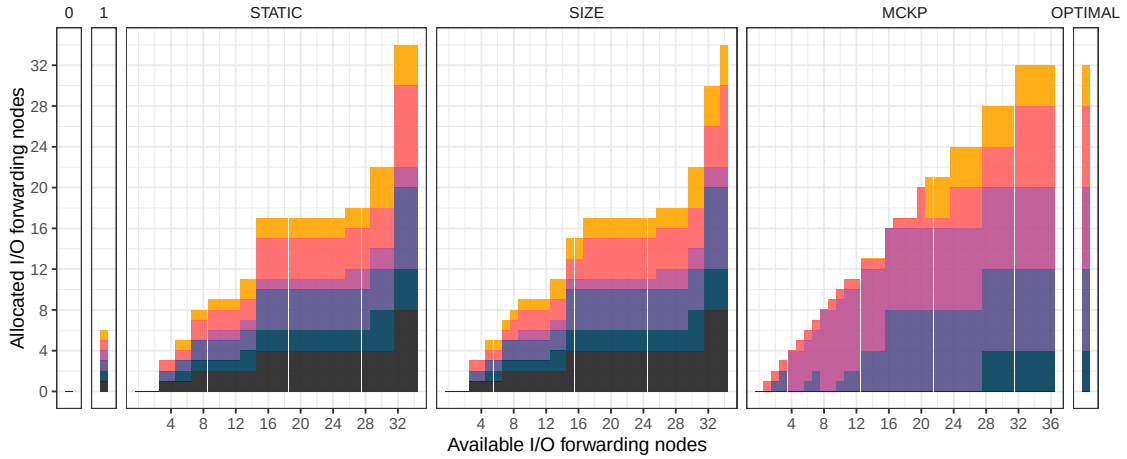
In this section, we investigate the allocation of I/O nodes with a subset of the applications described in Section 5.6.1. The aggregated bandwidth is computed as the sum of the bandwidth achieved by each application when using the number of I/O nodes allocated to it by the arbitration policy. We focus on a set of jobs composed of **BT-C**, **BT-D**, **IOR-MPI**, **POSIX-L**, **MAD**, and **S3D**. In total, these applications require 72 compute nodes. A complete experiment with all applications, dynamically changing the allocated I/O nodes, is presented in Section 5.6.3.

Figure 5.9(a) details the results. The x-axis represents the number of available I/O nodes to arbitrate, and each box groups a policy. The first group represents direct access to Lustre, whereas the second is the ONE policy. The last box represents the ORACLE

Figure 5.9: Global aggregated bandwidth computed by Equation 5.2 and I/O nodes allocation for the six applications under different I/O policies. The x-axis of both plots represents the number of available I/O nodes. Colors differentiate applications.



(a) Bandwidth achieved by each application using the allocated I/O forwarding nodes of Figure 5.9(b).



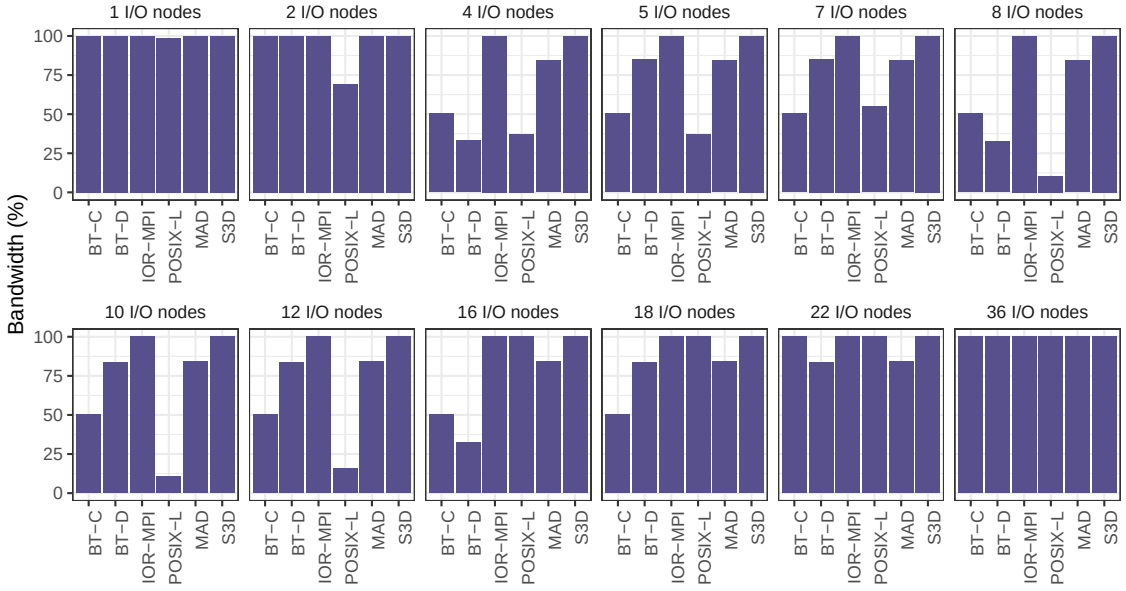
(b) Number of allocated I/O forwarding nodes when the applications are running concurrently in the offline simulation. Each bar's height depicts the amount of I/O nodes given to that application.

Source: Author

policy, as detailed in Section 5.4. As demonstrated by the results with FORGE at MN4, the ONE policy represents a global slowdown (39.17%) compared to directly accessing the PFS servers, even though some applications such as **S3D** would benefit from this choice. The STATIC, SIZE, and PROCESS (the latter not depicted) cannot achieve the same aggregated bandwidth as the MCKP policy that is $4.59\times$, $4.59\times$, and $4.1\times$ better than the alternatives. MCKP achieves the same performance of the ORACLE (the upper bound) when 36 nodes are available to be arbitrated among the 6 running applications (Figure 5.9(a)).

Regarding the number of allocated I/O nodes, the STATIC and SIZE policies dis-

Figure 5.10: Bandwidth achieved by individual applications using the assigned number of I/O nodes by our MCKP policy, compared to each application running alone under the same I/O node number constraint, i.e., the best result that application could achieve.



Source: Author

tribute the I/O nodes in a non-optimal way. Under the constraint of 12 available I/O nodes, for instance, applications **BT-C**, **MAD**, and **S3D** should not use forwarding, as detailed by Table 5.3. Instead, **IOR-MPI** should receive more I/O nodes as it can achieve a bandwidth that is $18.96\times$ higher when using eight forwarders instead of one. The MCKP policy does not give any I/O nodes for **S3D** as having a direct access to the PFS is the best option instead.

Table 5.3: Allocated forwarders and achieved bandwidth using the STATIC, SIZE, and MCKP policies when 12 I/O nodes are available to be arbitrated.

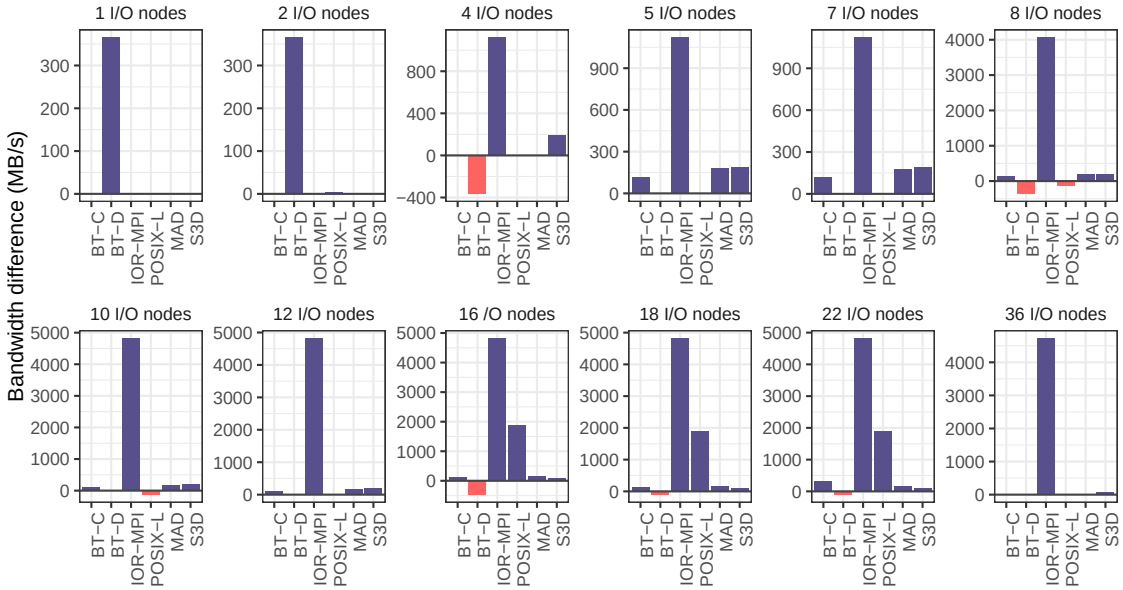
	STATIC		**SIZE**		**MCKP**	
	I/O Nodes	BW (MB/s)	I/O Nodes	BW (MB/s)	I/O Nodes	BW (MB/s)
BT-C	1	77.6	1	77.6	0	195.7
BT-D	2	594.2	2	594.2	1	597.2
IOR-MPI	1	268.4	1	268.4	8	5089.9
POSIX-L	2	411.9	2	411.9	2	411.9
MAD	1	77.8	1	77.8	0	255.9
S3D	2	48.1	2	48.1	0	241.3

Source: Author

We analyzed the penalty to the performance of individual applications caused by our MCKP policy, which aims at maximizing the *global* bandwidth, in Figure 5.10. For each total number of available I/O nodes (the boxes), we show the performance of each

application (x-axis) with the assigned number of I/O nodes, compared to the best possible result for that application running alone under the same number of I/O nodes. With four I/O nodes, applications **IOR-MPI** and **S3D** manage to achieve the same performance they would attain when running alone under this constraint. For both, choosing between 1, 2, or 4 I/O nodes, the latter is always the best choice. However, for the remaining applications, such as **BT-C** or **BT-D**, where 4 is also the best choice, they reach only 50% and 33% of the bandwidth they could achieve if running alone under that constraint. When running with other applications, e.g., **IOR-MPI** and **S3D**, they are not prioritized by the policy because they do not gain performance as the first two when using more I/O nodes.

Figure 5.11: Bandwidth difference between applications running under STATIC and MCKP. Positive (in purple) means MCKP was faster than STATIC. The y-axis is not the same in all the plots.



Source: Author

In Figure 5.11, we depict the bandwidth differences between the STATIC and MCKP policies for each application. Positive values mean that the MCKP was able to yield improvements, whereas negatives indicate that the STATIC policy was a better alternative for that particular application. Improving global bandwidth might often come from impairing specific applications. For instance, the MCKP policy sacrifices **BT-D** by giving fewer I/O nodes than what the STATIC policy would allocate to it. The reason is that **BT-D** has a lower bandwidth, and the increase in performance for the remaining applications is higher than what is lost by **BT-D**, if observed individually.

5.6.3 Dynamic Allocation Policy

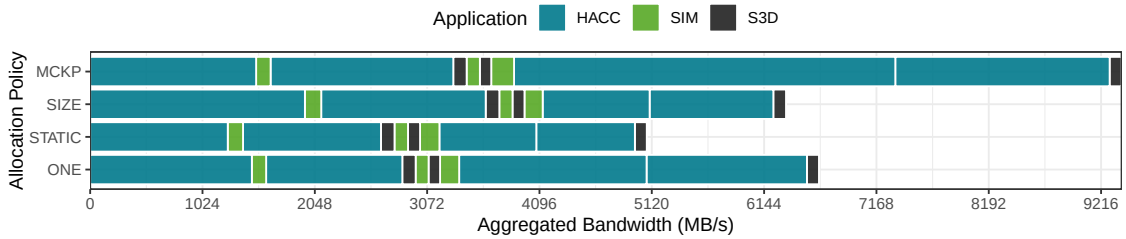
In this section, we use GekkoFWD with MCKP to dynamically arbitrate the I/O nodes between the changing set of running applications in the G5K platform. We split the Gros cluster nodes into two groups: 96 compute nodes and 12 I/O nodes. We deployed Lustre in the Grimoire cluster with one MGS/MDS node and two OSS with one OST of 500GB each. Lustre was configured with a stripe size of 1MB and striping over the available OSTs. We do not consider directly accessing the PFS, i.e., not using forwarding for this test to mimic platforms with this restriction.

In this experiment, we have a predefined queue of jobs to be executed following a strict FIFO order. Once one or more applications are scheduled, MCKP is invoked to choose the number of I/O nodes each should use. The decision considers all running jobs and may change the number of I/O nodes used by some of them. The policy is also invoked when jobs finish but new ones cannot be scheduled as there are not enough compute nodes yet. The only exception is the STATIC policy, that is invoked but will *not reallocate* resources for already running applications.

Figure 5.12 depicts the aggregated bandwidth comparison between a queue comprised of jobs from applications **HACC**, **SIM**, and **S3D**. The global aggregated performance is improved by $1.85\times$ by using MCKP rather than the STATIC policy, from 4.95GB/s to 9.18GB/s. Taking a close look at the bandwidth achieved by **HACC** in its four instances in the job queue, we observed improvements of $1.2\times$ (from 1.25GB/s to 1.51GB/s), $1.32\times$ (from 1.25GB/s to 1.66GB/s), $3.93\times$ (from 883.37MB/s to 3.47GB/s), and $2.17\times$ (from 898.7MB/s to 1.9GB/s) depending on the set of concurrently running applications. On the other hand, **SIM** and **S3D** applications observed a slight slowdown on their first executions, as the priority when using more I/O nodes was given to **HACC** that could benefit the most from the additional allocated resources. **SIM** attained $0.94\times$ (from 139.1MB/s to 132.0MB/s), $0.97\times$ (from 123.1MB/s to 120.6MB/s), and $1.18\times$ (from 176.2MB/s to 208.1MB/s) of the bandwidth achieved under the STATIC allocation. **S3D** attained $0.98\times$ (from 123.3MB/s to 121.7MB/s), $0.94\times$ (from 110.0MB/s to 103.5MB/s), and $1.00\times$ (from 108.3MB/s to 108.9MB/s) of the bandwidth using MCKP.

We generated random queues of jobs using the applications described in Section 5.6.1. We selected one queue whose metrics indicate a high number of concurrently running jobs to observe the arbitration of forwarding resources and the decisions' impact.

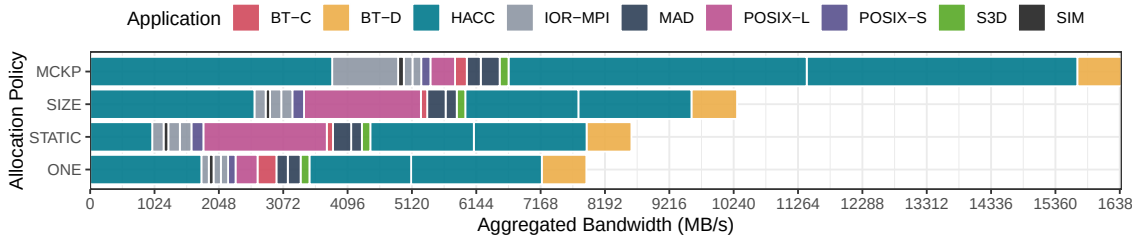
Figure 5.12: Aggregate bandwidth achieved by the arbitration policies while running the **HACC**, **SIM**, and **S3D** applications on the G5K platform using GekkoFWD.



Source: Author

The source code of the queue generator is open-source[8].

Figure 5.13: Aggregate bandwidth achieved by the arbitration policies while running at least one job for each application on the G5K platform using GekkoFWD.



Source: Author

The selected queue has at least one job of each application, in the following order: **HACC**, **IOR-MPI**, **SIM**, **IOR-MPI**, **IOR-MPI**, **POSIX-S**, **POSIX-L**, **BT-C**, **MAD**, **MAD**, **S3D**, **HACC**, **HACC**, and **BT-D**. Figure 5.13 illustrates the bandwidth achieved by each application (and the aggregated bandwidth given by Equation~5.2) under the ONE, STATIC, SIZE, and MCKP policies. Despite invoking the arbitration policy before applications start or after they finish, the STATIC will not change the number of I/O nodes (nor the mapping) that were first assigned to an application.

The first job of the **HACC** application is given 1 I/O node by the STATIC policy due to its size. In contrast, MCKP initially allocates 8 I/O nodes and then reduces to 4 as new jobs from **IOR-MPI** and **SIM** applications begin to execute. From the application's perspective, increasing the number of I/O nodes here translates into a bandwidth that is $3.9\times$ higher than on the STATIC allocation (from 987.3MB/s to 3850.7MB/s). For **POSIX-L**, the STATIC policy allocated 8 I/O nodes reaching $1,963.9$MB/s, whereas MCKP only allows the application to use 8 I/O nodes during 9.7% of the time, and 2 I/O nodes for 90.3% of the time, which limits the bandwidth to 391.7MB/s. For the **POSIX-L** application, using 4 I/O nodes (MCKP) instead of 2 (SIZE), bandwidth is improved by $5.8\times$, from 180.5MB/s to $1,049.9$MB/s with MCKP. It is possible to see that the latter

[8]<https://doi.org/10.5281/zenodo.3875176>

prioritizes applications that can reach high bandwidth by giving them more I/O nodes. Moreover, if we compare the STATIC solution to our dynamic MCKP arbitration policy, the latter improves global performance by $1.9\times$ in this scenario — from 8.41GB/s to 16.02GB/s.

The dynamic remapping of I/O nodes to the compute nodes does not require any synchronization between the nodes of the forwarding layer, which could impact performance. Instead, the GekkoFWD clients check whether the mapping changed periodically (every 10 seconds by default). Thus, there might be a brief period where I/O nodes are shared by more than one application, especially if compute node clocks are not synchronized. We believe this should not pose an issue as jobs run in higher orders of magnitude.

5.7 Discussions and Limitations

In this chapter, we argued in favor of a dynamic on-demand allocation of I/O nodes considering the application's I/O characteristics. We demonstrate that the forwarding layer's global deployment combined with the existing static allocation policy based solely on application size (i.e., on the number of compute nodes it requires) is not suitable to accommodate the increasingly heterogeneous workloads entering HPC installations. Instead, an application's I/O characteristics should be considered when arbitrating these resources among concurrently running applications to improve global performance.

To understand the impact of I/O node allocation under different access patterns, we proposed FORGE, a lightweight I/O forwarding explorer. FORGE collects performance metrics to aid in understanding the effect of forwarding in an HPC system. Additionally, we presented a user-level I/O forwarding solution named GekkoFWD that does not require application modifications and allows a dynamic remapping of forwarding resources to compute nodes. GekkoFWD is simple to run in production machines, where this layer is not present, targeting applications that would benefit from it.

We proposed a novel I/O forwarding allocation policy based on the Multiple-Choice Knapsack Problem. We demonstrate our dynamic MCKP allocation policy's applicability to arbitrate the available I/O forwarding resources through extensive evaluation and experimentation. We show it could transparently improve global I/O bandwidth by up to $23\times$ compared to the existing static policy, though improving global bandwidth might often come from impairing some applications. We also observed improvements of up to 85% in a live experiment using GekkoFWD and a queue of nine distinct applications.

Finally, to wrap up our discussion, our proposed dynamic allocation strategy has a caveat: the frequency to which we should react to changes. If we respond too fast, we might unnecessarily re-configure the system based on short-lived behaviors. On the other hand, if we react too slowly, we might not adapt to all the I/O workloads. In our evaluation, re-evaluating allocation decisions when the set of running applications change (i.e., a new job starts to execute or a running stop finishes) has proven sufficient to attain considerable gains over a static policy. We could change that frequency to accommodate systems with a high number of short or small jobs, or those could be considered ineligible to receiving forwarding resources as they would probably not benefit from them. Nonetheless, our approach gives system administrators the means to enforce different dynamic policies than the static solution available in those systems.

6 RELATED WORK

The I/O forwarding layer has been the focus of multiple research efforts to improve its performance and transparently benefit applications. Vishwanath et al. (2010) improved the I/O performance of an IBM Blue Gene/P supercomputer by up to 38% by updating this layer. Their modifications allowed for asynchronous operations in the I/O nodes and included a simple FIFO request scheduler to coordinate accesses from the multiple threads. The same authors later optimized data movement between layers through a topology-aware approach (VISHWANATH et al., 2011), whereas Isaila et al. (2011) proposed a two-level prefetching scheme. Ohta et al. (2010) implemented a FIFO, and the quantum-based HBRR request schedulers for the IOFSL framework. The latter aims at reordering and aggregating requests. TWINS (BEZ et al., 2017), a novel scheduler proposed for the forwarding layer, aims at coordinating accesses to the data servers to avoid contention. It was able to improve performance and alleviate interference when compared to FIFO and HBRR.

We explore related work on access pattern detection in Section 6.1. Current approaches to the dynamic tuning of HPC I/O are discussed in Section 6.2. Finally, in Section 6.3, we comment on resource allocation policies, more specifically those related to I/O nodes.

6.1 On Access Pattern Detection

Detecting access patterns is an essential topic as it allows adapting the I/O system to the workload. For that, both *postmortem* and *runtime* approaches are popular. In the case of postmortem, after the execution, information is often obtained from traces and applied to future executions of the same applications (LIU et al., 2014; BOITO et al., 2016), targeting repeated patterns with similar characteristics. For instance, Behzad et al. (2019) define high-level I/O patterns to characterize write operations by collecting high-level I/O calls, such as the HDF5 data model definition and write calls, using the Recorder (Luu et al., 2013) tracer library. They then use H5Analyze to execute an analysis on the trace files to come up with the patterns to guide parameter tuning for HDF5 applications.

In this work, we favor a runtime technique to avoid imposing the profiling effort and also to benefit from similarities between different applications. Furthermore, runtime detection allows the applications to start profiting from tuning optimizations more quickly,

differently from postmortem analysis.

At runtime, techniques can typically only use information from operations already performed. To predict future accesses, the technique proposed by Dorier et al. (2014) named Omnisc'IO, intercepts I/O operations and builds a grammar. The approach employed by Tang et al. (2014) periodically analyzes previous accesses and applies a rule library to predict future accesses (for prefetching). They collect metrics regarding spatiality of read requests from the MPI-IO library.

Other techniques benefit from information obtained from I/O libraries. Ge, Feng and Sun (2012) collect data from MPI-IO covering the type of operation, data size, spatiality, and whether or not operations are collective and synchronous. Liu, Chen and Zhuang (2013b) collect the number of processes, the number of aggregators, and binding between nodes and processes. Lu et al. (2014) use the offsets accessed by each process during collective operations. The access spatiality of the application's processes is used in the approach proposed by Song et al. (2011a).

These are client-side techniques. Consequently, they would not work at the forwarding layer or at the server-side, where less information is available, and the observed pattern is the interaction of multiple concurrent patterns. Conversely, in this work, we focus our efforts on detecting the access pattern at the I/O forwarding layer as it is a perfect place to apply optimization since this layer is transparent to applications.

6.2 On Dynamic Tuning of Parameters

Numerous parameters – in different levels of the HPC I/O stack – can affect I/O performance, thus tuning the system normally requires a large number of experiments to properly understand the interplay of factors and their impact on the performance. Table 6.1 compares our approach to related work that has been directed to aid in the configuration of the I/O stack. We classify each research based on: whether or not previous training is required; if any access pattern detection technique is used; if the tuning approach is application and file system agnostic; and if real-time metrics are used for the learning and tuning process. The dash on the work of Chen et al. (2009a) and Zhang et al. (2011) for application and file system agnostic is used as their research does not target HPC applications, but rather on distributed systems such as the web.

Chen et al. (2009a) use the Markov Decision Process theory and a reinforcement learning strategy to discover a relationship between the system workload and the optimal

Table 6.1: Comparison of features in our proposal to related work.

Related Work	Previous Training	Access Pattern Detection	Application Agnostic	File System Agnostic	Real-time Metrics
Chen et al. (2009a)	✔	✖	-	-	✔
Zhang et al. (2011)	✔	✖	-	-	✔
Agarwal et al. (2019)	✔	✖	✔	✖[1]	✖
Behzad et al. (2013), Behzad et al. (2019)	✔	✔	✔	✔	✖
Boito et al. (2016)	✔	✔	✔	✔[2]	✖
Li et al. (2017)	✖	✖	✔	✖[3]	✔
Our approach	✖	✔	✔	✔	✔

[1] Works on Lustre and requires the number of OSTs and stripe count.
[2] Requires AGIOS integration on the data server.
[4] Requires modification on the file system client.

Source: Author

configuration. Their solution uses an actor and a critic. The first follows the stochastic policy that maps system states to configuration settings, and the latter uses a value function to provide feedback to the actor. Both are implemented by multiple-layer neural networks. Zhang et al. (2011) propose an ordinal optimization-based strategy combined with a back-propagation neural network for the auto-tuning of configuration parameters in distributed systems. They demonstrate that their approach reduces the number of required measurements in a real system compared to the use of traditional evolutionary optimization strategies in the search for the optimal policy. Their method searches for a balance between performance and simulation time. Both approaches require a previous training phase and do not use any access pattern detection mechanism. Chen et al. focus their approach tuning a web server where a change in the workload is represented by the volume of accesses (maximum clients and time a connection should remain open). Zhang et al. share the same focus, but considering more parameters that were relevant to represent a web system's performance.

In the context of HPC, Agarwal et al. (2019) propose two auto-tuning models, based on active learning, to recommend a set of parameter values for MPI-IO hints and Lustre configuration for an application on a given system. They employ Bayesian optimization to find the parameter values. Though their approach still requires training, due to their separation of real-application execution and I/O prediction model, training time is reduced compared to other methods. Nevertheless, as they seek to minimize loss for the overall execution, mixed workloads of read and write requests might not achieve

the optimal performance without a fine-grain tuning. Moreover, their approach requires re-training the model with additional data to be able to target new applications' access patterns. Our runtime proposal, on the other hand, monitor metrics in a finer-grain to adapt to the application access pattern, learning with new decisions during its lifetime.

Behzad et al. (2013), Behzad et al. (2019) proposed an auto-tuning framework to set the parameters of the HDF5, MPI-IO, and parallel file systems. It extracts the I/O kernel of an application using tracing tools and runs the kernel with a pre-selected training set of tunable parameters. The authors employ adaptive heuristic search approaches – genetic evolution algorithms and simulated annealing – to transverse the search space in a reasonable amount of time. Furthermore, they use a regression model to predict the top 20 tunable parameter values that would improve I/O performance and run the kernel once more to select the best parameters under those conditions. Their approach decreases the number of experiments to be executed to tune the parameters, but it does not eliminate the profiling phase. Instead, our method can continuously receive metrics at runtime and adapt to the changing workload. Behzad et al. (2019) also have a key-value store to associate patterns with their I/O performance model, and a modeling component, for when no previously trained model is available. Nonetheless, they might require more than 10 hours to come up with a nonlinear regression model for the patterns the framework is not able to find a match. In contrast, our approach can demonstrate improvements a few minutes after a new access pattern is first observed by dynamically exploring the space using the armed-bandit RL technique. Another approach that requires previous training but uses access pattern detection is proposed by Boito et al. (2016). Their approach is application and file system agnostic and rely on decision trees.

CAPES, the tuning system proposed by Li et al. (2017), takes periodic measurements of a machine and train (online) a deep neural network that uses Q-learning to change parameters. They applied it to tune the congestion window size and the I/O rate limit on a Lustre PFS, improving write performance by up to 45%. Their approach requires previous training that can take over 24 hours for the synthetic workloads considered in the paper, generated with the Filebench benchmark. Conversely, as our system does not require an initial training phase, it can start to learn and benefit from it a few minutes after a new access pattern is first seen. Table 6.2 summarizes the learning methods adopted by each approach, and the target location employed by each one to demonstrate its applicability.

Building models to represent the impact of parameters is a usual strategy, as done

Table 6.2: Learning methods and targeted tuned locations used by each related work when compared to our proposal.

Related Work	Location	Learning Methods
Chen et al. (2009a)	Web Servers	Markov Decision Process Multiple-Layer Neural Networks Reinforcement Learning
Zhang et al. (2011)	Web Servers	Ordinal Optimization Back-propagation Neural Network
Agarwal et al. (2019)	Lustre MPI-IO	Active Learning Bayesian Optimization Extreme Gradient Boosting
Behzad et al. (2013), Behzad et al. (2019)	HDF5 MPI-IO Lustre Servers	Genetic Algorithms Simulated Anneling
Boito et al. (2016)	Data Servers	Decision Trees
Li et al. (2017)	Lustre Clients	DNN Q-learning
Our approach	I/O Nodes	Armed Bandits, Reinforcement Learning

Source: Author

by McLay et al. (2014) to optimize MPI-IO collective writes to Lustre. They propose a set of heuristics based on exploratory experiments on three machines to explain the impact of the choice of the tuned parameters. Focusing on block-level local storage, in the context of a single data server, Nou, Giralt and Cortes (2012) used pattern matching to record known patterns and their performance with different disk schedulers, using this information to adapt. Both approaches require previous experiments to explore the outcomes when using different parameters and to build models that can be used to guide tuning decisions. Conversely, we targeted a dynamic and transparent approach that would allow our tuning mechanism to learn the best choice for each parameter, by observing system metrics and reacting accordingly to changes in the access pattern and system.

Regarding the access pattern detection used in this work, Bez et al. (2019) compare techniques to classify (at the forwarding level) the spatiality of accesses. Those techniques were used in the experiments presented in Chapter 4 , but other solutions could be plugged-in to make this detection. Furthermore, Boito et al. (2019) proposes a more generic server-side access pattern classification approach that would be suitable when adapting any optimization mechanism, not only TWINS. In that approach, we employed some performance measurements obtained with TWINS to make an offline adaptation depending on the detected access pattern. However, that adaptation represented an oracle as the only goal was to evaluate the accuracy of the proposed pattern detection technique.

6.3 On I/O Forwarding Allocation

Yu et al. (2017c) address the load imbalance problem of the I/O forwarding layer. They argue that the bursty I/O traffic of HPC applications and the commonly rank 0 I/O pattern make the I/O nodes highly unbalanced. As some I/O nodes become hot spots, they hinder performance. Thus, they propose to recruit idle I/O nodes to alleviate this problem by giving additional nodes to the applications. In their approach, they strip files across a number of I/O nodes. Hence each I/O node is responsible for different portions of the file, and each storage server only servers one I/O node in the group (Yu et al., 2017a). The mapping of additional I/O nodes and primary I/O nodes are exclusive for an application. Thus, different applications can have different mappings, which adds some flexibility to the default static solution. However, the I/O nodes temporarily allocated to aid others might be required by their subset of compute nodes. In that case, global performance would be impacted. Furthermore, once the mapping is done for an application, it cannot be changed to accommodate new jobs that would benefit from more from additional nodes. Differently, we propose a dynamic approach to the problem, reviewing I/O node allocations as new jobs start or end their executions. Moreover, we employ a resource pool of available I/O nodes that we need to arbitrate among the running applications, instead of recruiting idle resources from this layer.

The Tianhe-2 (Milkyway-2) supercomputer has a hybrid hierarchy storage system named H^2FS, that merges the local storage of I/O nodes and the disks in the object storage servers (XU et al., 2014). This supercomputer can be configured with as few as one I/O per 64 compute nodes, or as many as one I/O node per $eight$ compute nodes. The H^2FS has an I/O path manager that maps compute nodes to a group of I/O nodes. Two mapping modes are supported: static and dynamic. The first is determined by network topology (system deployment) when the system is initialized, or the applications specify it. The latter selects the I/O path based on the real-time overhead of the DPUs, seeking to reduce congestion. If the overhead difference of the DPUs is within a pre-defined range, a round-robin policy is taken to allocate the I/O nodes. Otherwise, the dynamic mapping adjusts the I/O path and allocates DPUs for every file dynamically. Conversely, our approach does not focus on such a small granularity, i.e., file level, but rather on the whole application behavior. Moreover, our policy is not fixed for a given application, but instead, it takes into account concurrent jobs, adapting based on the workload.

Ji et al. (2019) propose a Dynamic Forwarding Resource Allocation (DRFA),

which estimates the number of forwarding nodes needed by a certain job based on its I/O history records. Their approach leverages automatic and online I/O subsystem monitoring and performance data analysis to make such decisions. DFRA works by remapping a group of compute nodes (scheduled to start executing an application) to other than their default I/O node assignments. They either grant more forwarding nodes (for capacity) or unused forwarding nodes (for isolation). Only dedicated nodes from an idle pool are used, where they expect no further interference. Nonetheless, their allocation remains fixed once the job starts and do not adapt or allow a remapping when new applications start or finish to run. Conversely, we argue for a dynamic policy that can evaluate the set of running jobs to arbitrate the available I/O nodes. Moreover, their strategy relies on an over-provisioning of I/O nodes and assumes that there are enough idle resources to satisfy all allocation upgrades. On the other hand, our approach does not rely on that assumption but seeks to arbitrate among the already existing forwarding resources. Table 6.3 summarizes and compares the features in our proposal to previous related works.

Table 6.3: Comparison of features in our proposal to related work.

Related Work	**Focus**	**Policy**	**Static Mapping**	**Dynamic Mapping**	**Mapping Reevaluation**	**Source-code Modification**
Xu et al. (2014)	Network Topology Application	Network Topolgy	✔	✖	✖	✔[1]
Xu et al. (2014)	File	Static (Round-Robin) Overhead	✖	✔	✖	✖
Yu et al. (2017c)	Application	Stripping	✔	✖	✖	✖
Ji et al. (2019)	Application	Capacity Upgrade Isolation Upgrade	✖	✔	✖	✖
Our approach	Application	One, Static, Size, Process, MCKP	✖	✔	✔	✖

[1] For a mapping focused on the application, source-code modification is required.

Source: Author

In light of recent trends in storage system design using compute node local storage, we believe our approach is complementary. Local storage on the compute nodes is often temporary and eventually needs to be flushed to the PFS, flowing through the forwarding layer (if present). Furthermore, local storage is often limited and might not be enough to hold the applications' data. If used solely as a cache for reads or writes (similar to a burst buffer), it would still need to eventually reach the PFS, once more flowing through the forwarding layer. In those scenarios where an application does not issue I/O requests

directly to the PFS, MCKP would not allocate nodes for it. In contrast, the default SIZE policy would misallocate those resources that could be more useful to other applications.

Mingming et al. (2020) present an initial work towards an application forwarding layer. They seek to use dedicated compute nodes from a Tianhe 2 supercomputer cluster to process the requests of different applications and optimize I/O performance. By manually forwarding requests from the IOR framework to the AGIOS library, located on a remote node, which instead issues the request to the file system, they mimic a forwarding layer's behavior. However, their approach strongly relies on modifying the application's source code to interface with this logical layer. Furthermore, their initial evaluation does not consider allocation aspects. We believe the solutions we have proposed in this work with GeekkoFWD and the reallocation policies provide a more suitable and feasible alternative.

Finally, using the I/O information to make job allocation decisions is not comprised by this work. Different related works (ZHOU et al., 2015; HERBEIN et al., 2016; AUPY; GAINARU; FèVRE, 2019; Miranda et al., 2019) cover this topic. They are considered complementary to our proposal.

7 CONCLUSION

Different I/O optimization techniques (including but not limited to the I/O forwarding layer) provide improvements for specific system configurations and application access patterns, but not for all of them. Furthermore, they often require fine-tuning of parameters. In this research, we sought to dynamically tune the I/O forwarding layer in HPC platforms to improve global performance. We explored two approaches that use the application's access patterns as guidelines to make decisions: tuning scheduling parameters at the forwarding layer and arbitrating the available I/O nodes between the ever-changing set of running applications.

We demonstrated the applicability of different machine learning techniques to automatically detect the I/O access pattern of HPC applications at runtime. We investigated decision trees, random forests, and neural networks to classify runtime metrics into common access patterns. To illustrate its applicability, we evaluated these strategies by estimating the impact of correctly detecting the application's access pattern to tune the TWINS scheduler's window size at the forwarding layer. Our results have shown that all detection approaches covered in this work can correctly detect the access pattern. The detection mechanisms achieved up to 99% of an Oracle solution's performance. We also demonstrated improvements of approximately 17%, on average, over a statically defined window. Lastly, by correctly identifying the access pattern at runtime, we were able to avoid performance slowdowns caused by bad parameter choices.

We proposed a novel approach to adapt the forwarding layer to the current I/O workload. We periodically observe access pattern metrics collected by the I/O nodes to detect the access pattern and apply a reinforcement learning technique named contextual bandits. We showed that the system could learn the best choice for each access pattern at runtime, removing the tuning responsibility from the users. For our case study (TWINS), the results for the offline evaluation of 144 scenarios showed that our approach is capable of reaching a precision of $\approx 88\%$ (while providing $\approx 99\%$ of the best option's performance) in the first hundreds of observations of a given access pattern. Our online evaluation showed in practice that our approach is capable of discovering the correct window sizes and of showing runtime improvements of up to 19.3% by avoiding window sizes that could harm performance.

Additional experiments with an application, the MADspec I/O workload, demonstrated our learning mechanism's applicability by reducing the impact of a wrong window

choice by up to 17%. Finally, the median overhead imposed by our proposal was inferior to 2%, and the time required to announce metrics and reach a decision was short enough to make adaptation viable. It is vital to notice that the approach we proposed in this thesis is not particular to tuning the TWINS window size parameter. It can be applied to other situations where the current access pattern information is relevant to adjust a given configuration parameter in the I/O stack.

Though I/O forwarding is an established and widely-adopted HPC technique to reduce contention and improve I/O performance in the access to shared storage infrastructure, it is not always possible to explore its advantages under different setups without impacting or disrupting production systems. In this work, we have also investigated I/O forwarding by considering particular applications I/O access patterns and system configuration, rather than trying to guess a one-size-fits-all configuration for all applications. By determining when forwarding is the best choice for an application and how many I/O nodes it would benefit from, we can guide allocation policies to reach better decisions.

To understand the impact of forwarding I/O requests of different access patterns, we implemented FORGE, a lightweight forwarding layer in user-space. We explored 189 scenarios covering distinct access patterns and demonstrated that, as expected, the optimal number of I/O nodes varies depending on the application. While for 90.5% patterns forwarding would be the correct course of action, allocating two I/O nodes would only bring performance improvements for 44% of the scenarios. Our results on the MareNostrum and Santos Dumont supercomputers demonstrate that shifting the focus from a static system-wide deployment to an on-demand reconfigurable I/O forwarding layer dictated by application demands can improve I/O performance on future machines.

Regarding I/O node resource arbitration, we argued in favor of a dynamic on-demand allocation of I/O nodes considering the application's I/O characteristics. We demonstrate that the forwarding layer's global deployment combined with the existing static allocation policy based solely on application size is not suitable to accommodate the increasingly heterogeneous workloads entering HPC installations. Instead, an application's I/O characteristics should also be considered when arbitrating forwarding resources among concurrently running applications to improve global performance.

We presented a user-level I/O forwarding solution named GekkoFWD that does not require application modifications and allows a dynamic remapping of forwarding resources to compute nodes. GekkoFWD is simple to run in production machines, where this layer is not yet present, targeting applications that would benefit from it. We pro-

posed a novel I/O forwarding allocation policy based on the Multiple-Choice Knapsack Problem. In this context, we have multiple classes representing each running application, and the items on each class represent the number of I/O nodes we could choose. We must elect one item for each application, seeking to maximize the global bandwidth.

We demonstrate our dynamic MCKP allocation policy's applicability to arbitrate on the available I/O forwarding resources through extensive evaluation and experimentation. We show it could transparently improve global I/O bandwidth by up to $23\times$ compared to the existing static policy, though improving global bandwidth might often come from impairing specific applications. Furthermore, we observe improvements of up to 85% in a live experiment using GekkoFWD and a queue of nine different applications.

7.1 Future Work

The tuning approach we proposed in this work is not particular for the test case of TWINS that we used to demonstrate its applicability. We could also employ it in other situations that require knowledge about the application's access pattern to make better decisions. A couple of examples would be parameters at the MPI-IO layer to tune collective buffering or data sieving (by setting the number of collective buffering nodes, or the size of an intermediate buffer on an aggregator), or even on the HDF5 library (to tune the chunk and cache sizes). Future work will focus on extending our approach to other tunable parameters of the HPC I/O stack.

A demonstration of our tuning solution in a large scale machine is also in our plans, although made difficult by having to update the I/O forwarding software – as we required changes to enable communication of metrics and mechanism to apply and enforce the new decisions. Nonetheless, this does not invalidate prototyping efforts, and initial results in smaller platforms serve as a proof-of-concept.

The new forwarding features included in GekkoFS are definitely a step toward allowing such tunable techniques to be applied in a system. Furthermore, as it adds flexibility on the allocation and reallocation of forwarding resources, it opens up new research perspectives to further consider the increase in depth of the I/O stack (e.g., using local SDD devices, NVRAM, I/O nodes, the file system servers to store the data). Integration with existing job schedulers could also consider the allocation of forwarding resources among the application in a more pro-active fashion.

Finally, as we have presented two approaches to optimizing I/O performance based

on the access pattern detection, using that information to make tuning (finer-grain) and allocation (coarse-grain) decisions, we plan to investigate merging these two techniques. That would require sharing knowledge between the two methods to prioritize optimizations according to system policies and applications' needs.

7.2 Publications

The research presented in this document resulted in the following contributions:

- **J. L. Bez**, A. Miranda, R. Nou, F. Z. Boito, T. Cortes, P. O. A. Navaux, **Arbitration Policies for On-Demand User-Level I/O Forwarding on HPC Platforms**, *35th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2021. (*Accepted*).

- **J. L. Bez**, F. Z. Boito, R. Nou, A. Miranda, T. Cortes, P. O. A. Navaux, **Towards On-Demand IO Forwarding in HPC Platforms**, *2020 IEEE/ACM Fifth International Parallel Data Systems Workshop (PDSW)*, 2020.

- **J. L. Bez**, F. Z. Boito, R. Nou, A. Miranda, T. Cortes, P. O. A. Navaux, **Adaptive Request Scheduling for the I/O Forwarding Layer**, *Future Generation Computer Systems*, 2020.

- **J. L. Bez**, F. Z. Boito, R. Nou, A. Miranda, T. Cortes and P. O. A. Navaux, **Detecting I/O Access Patterns of HPC Workloads at Runtime**, *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Campo Grande, Brazil, 2019, pp. 80-87.

- P. J. Pavan, **J. L. Bez**, M. S. Serpa, F. Z. Boito and P. O. A. Navaux, **An Unsupervised Learning Approach for I/O Behavior Characterization**, *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Campo Grande, Brazil, 2019, pp. 33-40.

- F. Z. Boito, R. Nou, L. L. Pilla, **J. L. Bez**, J. F. Meháut, T. Cortes, and P. O. A. Navaux, **On server-side file access pattern matching**, *HPCS 2019 - 17th International Conference on High Performance Computing & Simulation*.

- A. R. Carneiro, **J. L. Bez**, F. Z. Boito, B. A. Fagundes, C. Osthoff and P. O. A. Navaux, **Collective I/O Performance on the Santos Dumont Supercomputer**, *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, Cambridge, 2018, pp. 45-52.

- **J. L. Bez**, A. R. Carneiro, F. Z. Boito, B. A. Fagundes, C. Osthoff and P. O. A. Navaux, **I/O Performance of the Santos Dumont Supercomputer**, *International Journal of High Performance Computing Applications (IJHPCA)*.

- F. Z. Boito, E. C. Inacio, **J. L. Bez**, P. O. A. Navaux, M. A. R. Dantas, and Y. Denneulin, 2018, **A Checkpoint of Research on Parallel I/O for High-Performance Computing**, *ACM Computing Surveys* 51, 2, Article 23 (March 2018), 35 pages.

- P. J. Pavan, R. K. Lorenzoni, V. R. Machado, **J. L. Bez**, E. L. Padoin, F. Z. Boito, P. O. A. Navaux, J. Méhaut, **Energy efficiency and I/O performance of low-power architectures**, *Concurrency and Computation: Practice and Experience*, 2019, 31:e4948.

We presented a poster with the initial concepts of I/O node arbitration:

- **J. L. Bez**, F. Z. Boito, R. Nou, A. Miranda, T. Cortes, P. O. A. Navaux, **Towards the Reconfiguration of the I/O Forwarding Layer**, *33rd IEEE International Parallel & Distributed Processing Symposium*, Rio de Janeiro, Brazil, 2019.

Finally, we presented research perspectives and progress in workshops:

- **J. L. Bez**, F. Z. Boito, R. Nou, A. Miranda, T. Cortes, P. O. A. Navaux, **Dynamic I/O Forwarding Reconfiguration**, *JLESC Workshop – Joint Laboratory on Extreme Scale Computing*, Barcelona, Spain, 2018.

- **J. L. Bez**, F. Z. Boito, R. Nou, A. Miranda, T. Cortes, P. O. A. Navaux, **A Reinforcement Learning Strategy to Tune Request Scheduling at the I/O Forwarding Layer**, *HPC-IODC: HPC I/O in the Data Center Workshop*, ISC High Performance, 2020.

REFERENCES

ABADI, M. et al. **TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems**. 2015. Available from Internet: <https://tensorflow.org>.

Agarwal, M. et al. Active Learning-based Automatic Tuning and Prediction of Parallel I/O Performance. In: **2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)**. [S.l.: s.n.], 2019. p. 20–29.

ALI, N. et al. Scalable I/O Forwarding Framework for High-Performance Computing Systems. In: IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING AND WORKSHOPS, 2009, Berkeley, USA. **2009 IEEE International Conference on Cluster Computing and Workshops**. [S.l.]: IEEE, 2009. p. 1–10.

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

AUER, P.; CESA-BIANCHI, N.; FISCHER, P. Finite-time analysis of the multiarmed bandit problem. **Mach. Learn.**, Kluwer Academic Publishers, USA, v. 47, n. 2–3, p. 235–256, may 2002. ISSN 0885-6125. Available from Internet: <https://doi.org/10.1023/A:1013689704352>.

AUPY, G.; GAINARU, A.; FèVRE, V. L. I/o scheduling strategy for periodic applications. **ACM Transactions on Parallel Computing**, Association for Computing Machinery, New York, NY, USA, v. 6, n. 2, jul. 2019. ISSN 2329-4949. Available from Internet: <https://doi.org/10.1145/3338510>.

BALI, R.; SARKAR, D.; LANTZ, B. **R: Unleash Machine Learning Techniques**. Packt Publishing, 2017. (Learning path). ISBN 9781787127340. Available from Internet: <https://books.google.com.br/books?id=7dMvswEACAAJ>.

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

Bağbaba, A. Improving collective i/o performance with machine learning supported auto-tuning. In: **2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)**. [S.l.: s.n.], 2020. p. 814–821.

BEHZAD, B. et al. Optimizing I/O Performance of HPC Applications with Autotuning. Association for Computing Machinery, New York, NY, USA, v. 5, n. 4, 2019.

BEHZAD, B. et al. Taming parallel i/o complexity with auto-tuning. In: **Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis**. New York, NY, USA: Association for Computing Machinery, 2013. (SC '13). ISBN 9781450323789. Available from Internet: <https://doi.org/10.1145/2503210.2503278>.

BERRY, D. A.; FRISTEDT, B. Bandit problems: sequential allocation of experiments (monographs on statistics and applied probability). **London: Chapman and Hall**, Springer, v. 5, p. 71–87, 1985.

BEZ, J. L. **Evaluating I/O Scheduling Techniques at the Forwarding Layer and Coordinating Data Server Accesses**. Dissertation (Master) — PPGC - Federal University of Rio Grande do Sul, 2016.

BEZ, J. L. et al. Towards On-Demand I/O Forwarding in HPC Platforms. In: **Int. Parallel Data Systems Workshop**. [S.l.]: IEEE, 2020.

BEZ, J. L. et al. Detecting I/O Access Patterns of HPC Workloads at Runtime. In: **International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)**. Brazil: [s.n.], 2019. p. 1–8.

BEZ, J. L. et al. TWINS: Server Access Coordination in the I/O Forwarding Layer. In: **2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)**. [S.l.: s.n.], 2017. p. 116–123.

BOITO, F. Z. **Estimation of the impact of I/O forwarding on application performance**. [S.l.], 2020. 20 p. Available from Internet: <https://hal.inria.fr/hal-02969780>.

BOITO, F. Z. et al. A Checkpoint of Research on Parallel I/O for High-Performance Computing. **ACM Comput. Surv.**, ACM, v. 51, n. 2, p. 23:1–23:35, 2018.

BOITO, F. Z. et al. Towards fast profiling of storage devices regarding access sequentiality. In: **Proceedings of the 30th Annual ACM Symposium on Applied Computing**. New York, NY, USA: Association for Computing Machinery, 2015. (SAC '15), p. 2015–2020. ISBN 9781450331968. Available from Internet: <https://doi.org/10.1145/2695664.2695701>.

BOITO, F. Z. et al. AGIOS: Application-guided I/O scheduling for parallel file systems. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED SYSTEMS (ICPADS). **2013 International Conference on Parallel and Distributed Systems**. Seoul, South Korea: IEEE, 2013. p. 43–50. ISSN 1521-9097.

BOITO, F. Z. et al. Automatic I/O scheduling algorithm selection for parallel file systems. **Concurrency and Computation: Practice and Experience**, 2016.

BOITO, F. Z. et al. On server-side file access pattern matching. In: **2019 International Conference on High Performance Computing Simulation (HPCS)**. [S.l.: s.n.], 2019. p. 217–224.

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

Borrill, J. et al. Investigation of leading HPC I/O performance using a scientific-application derived benchmark. In: **SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing**. [S.l.: s.n.], 2007. p. 1–12.

BOX, G. E. P.; COX, D. R. An analysis of transformations. **Journal of the Royal Statistical Society: Series B (Methodological)**, v. 26, n. 2, p. 211–243, 1964. Available from Internet: <https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/j.2517-6161.1964.tb00553.x>.

BREIMAN, L. Random Forests. **Machine Learning**, v. 45, n. 1, p. 5–32, Oct 2001.

CARNS, P. et al. Understanding and Improving Computational Science Storage Access Through Continuous Characterization. **Trans. Storage**, ACM, New York, NY, USA, v. 7, n. 3, p. 8:1–8:26, oct. 2011. ISSN 1553-3077.

CARNS, P. et al. 24/7 characterization of petascale i/o workloads. In: IEEE. **2009 IEEE International Conference on Cluster Computing and Workshops**. [S.l.], 2009. p. 1–10.

CARNS, P. H. **ALCF I/O Data Repository**. 2013. <http://ftp.mcs.anl.gov/pub/darshan/data/>.

CHEN, H. et al. Boosting the Performance of Computing Systems through Adaptive Configuration Tuning. In: . [S.l.: s.n.], 2009.

CHEN, J. H. et al. Terascale direct numerical simulations of turbulent combustion using s3d. **Computational Science & Discovery**, IOP Publishing, v. 2, n. 1, p. 015001, jan 2009.

Ching, A. et al. Exploring I/O Strategies for Parallel Sequence-Search Tools with S3aSim. In: **2006 15th IEEE International Conference on High Performance Distributed Computing**. Paris, France: IEEE, 2006. p. 229–240. ISSN 1082-8907.

CHOLLET, F. et al. **Keras**. 2015. <https://keras.io>.

COHEN, J. A coefficient of agreement for nominal scales. **Educational and Psychological Measurement**, v. 20, n. 1, p. 37–46, 1960.

CONGIU, G. et al. Improving Collective I/O Performance Using Non-volatile Memory Devices. In: **2016 IEEE International Conference on Cluster Computing (CLUSTER)**. [S.l.]: IEEE, 2016. p. 120–129.

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

CUTLER, D. R. et al. Random forests for classification in ecology. **Ecology**, v. 88, n. 11, p. 2783–2792, 2007.

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

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

DORIER, M. et al. Omnisc'IO: a grammar-based approach to spatial and temporal I/O patterns prediction. In: **Proceedings of the Int. Conference for High Performance Computing, Networking, Storage and Analysis**. [S.l.: s.n.], 2014. p. 623–634.

DUNN, J.; DUNN, O. J. Multiple comparisons among means. **American Statistical Association**, p. 52–64, 1961.

GAINARU, A. et al. Scheduling the i/o of hpc applications under congestion. In: **2015 IEEE International Parallel and Distributed Processing Symposium**. [S.l.: s.n.], 2015. p. 1013–1022.

GE, R.; FENG, X.; SUN, X. H. SERA-IO: Integrating energy consciousness into parallel I/O middleware. In: **CCGRID '12 Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing**. [S.l.]: IEEE, 2012. p. 204–211. ISBN 9780769546919.

GORINI, S.; CHESI, M.; PONTI, C. **CSCS Site Update**. 2017. <http://opensfs.org/wp-content/uploads/2017/06/Wed11-GoriniStefano-LUG2017_20170601.pdf>. [Online; Accessed 7-January-2020].

HAHNLOSER, R. et al. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. **Nature**, v. 405, p. 947–51, 07 2000.

HE, J. et al. I/o acceleration with pattern detection. In: **Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing**. New York, NY, USA: Association for Computing Machinery, 2013. (HPDC '13), p. 25–36. ISBN 9781450319102. Available from Internet: <https://doi.org/10.1145/2493123.2462909>.

He, S.; Wang, Y.; Sun, X. Improving Performance of Parallel I/O Systems through Selective and Layout-Aware SSD Cache. **IEEE Transactions on Parallel and Distributed Systems**, v. 27, n. 10, p. 2940–2952, 2016.

HERBEIN, S. et al. Scalable I/O-Aware Job Scheduling for Burst Buffer Enabled HPC Clusters. In: **Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing**. New York, NY, USA: Association for Computing Machinery, 2016. (HPDC '16), p. 69–80. ISBN 9781450343145. Available from Internet: <https://doi.org/10.1145/2907294.2907316>.

HU, W. et al. Storage wall for exascale supercomputing. **Frontiers of Information Technology & Electronic Engineering**, v. 17, n. 11, p. 1154–1175, Nov 2016. ISSN 2095-9230. Available from Internet: <https://doi.org/10.1631/FITEE.1601336>.

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

ISAILA, F.; CARRETERO, J.; ROSS, R. CLARISSE: A middleware for data-staging coordination and control on large-scale HPC platforms. In: **2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)**. [S.l.: s.n.], 2016. p. 346–355.

ISAILA, F. et al. Making the case for reforming the i/o software stack of extreme-scale systems. **Preprint ANL/MCS-P5103-0314, Argonne National Laboratory**, 2014.

ISKRA, K. et al. ZOID: I/O forwarding infrastructure for petascale architectures. In: **in Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming**. [S.l.: s.n.], 2008. p. 153–162.

JI, X. et al. Automatic, Application-Aware I/O Forwarding Resource Allocation. In: **Proceedings of the 17th USENIX Conference on File and Storage Technologies**. USA: USENIX Association, 2019. (FAST'19), p. 265–279. ISBN 9781931971485.

Kougkas, A. et al. Leveraging burst buffer coordination to prevent i/o interference. In: **2016 IEEE 12th International Conference on e-Science (e-Science)**. [S.l.: s.n.], 2016. p. 371–380.

KRUSKAL, W. H.; WALLIS, W. A. Use of ranks in one-criterion variance analysis. **Journal of the American Statistical Association**, [American Statistical Association, Taylor & Francis, Ltd.], v. 47, n. 260, p. 583–621, 1952.

KUHN, M.; JOHNSON, K. **Applied Predictive Modeling**. [S.l.]: Springer New York, 2013. (SpringerLink : Bücher). ISBN 9781461468493.

KUHN, M.; JOHNSON, K. **Feature Engineering and Selection: A Practical Approach for Predictive Models**. CRC Press, 2019. (Chapman & Hall/CRC Data Science Series). ISBN 9781351609463. Available from Internet: <https://books.google.com.br/books?id=q5alDwAAQBAJ>.

KUMAR, S. et al. Characterization and modeling of PIDX parallel I/O for performance optimization. In: **SC '13 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis**. [S.l.]: ACM Press, 2013. p. 1–12.

KUO, C.-S. et al. How file access patterns influence interference among cluster applications. In: **2014 IEEE International Conference on Cluster Computing (CLUSTER)**. [S.l.]: IEEE, 2014. p. 185–193. ISSN 1552-5244.

LANDIS, J. R.; KOCH, G. G. The measurement of observer agreement for categorical data. **Biometrics**, [Wiley, International Biometric Society], v. 33, n. 1, p. 159–174, 1977.

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

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

LATHAM, R. et al. Parallel netCDF: A High-Performance Scientific I/O Interface. In: **SC Conference**. Los Alamitos, CA, USA: IEEE Computer Society, 2003. p. 39. Available from Internet: <https://doi.ieeecomputersociety.org/10.1109/SC.2003.10053>.

LEBRE, A. et al. I/O scheduling service for multi-application clusters. In: **2006 IEEE International Conference on Cluster Computing**. [S.l.: s.n.], 2006. p. 1–10.

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

LI, Y. et al. Capes: Unsupervised storage performance tuning using neural network-based deep reinforcement learning. In: **Supercomputing '17**. [S.l.: s.n.], 2017.

LIAO, W.-k.; CHOUDHARY, A. Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols. In: **Proceedings of the 2008 ACM/IEEE Conference on Supercomputing**. Austin, Texas: IEEE Press, 2008. (SC '08). ISBN 9781424428359.

LIU, J.; CHEN, Y.; ZHUANG, Y. Hierarchical i/o scheduling for collective i/o. In: **Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing**. IEEE Press, 2013. (CCGRID '13), p. 211–218. ISBN 9780768549965. Available from Internet: <https://doi.org/10.1109/CCGrid.2013.30>.

LIU, J.; CHEN, Y.; ZHUANG, Y. Hierarchical I/O scheduling for collective I/O. In: **Proceedings of the 13th International Symposium on Cluster, Cloud and Grid Computing**. [S.l.]: IEEE, 2013. p. 211–218. ISBN 978-0-7695-4996-5.

LIU, N. et al. On the role of burst buffers in leadership-class storage systems. In: **2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)**. [S.l.: s.n.], 2012. p. 1–11.

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

Liu, W.; Wu, L.; Xu, X. Topology aware algorithm for two-phase i/o in clusters with tapered hierarchical networks. **IEEE Access**, v. 8, p. 66917–66930, 2020.

LIU, Y. et al. Automatic Identification of Application I/O Signatures from Noisy Server-Side Traces. In: **FAST'14 Proceedings of USENIX conference on File and Storage Technologies**. [S.l.: s.n.], 2014. p. 213–228. ISBN ISBN 978-1-931971-08-9.

LIU, Y. et al. Server-side log data analytics for I/O workload characterization and coordination on large shared storage systems. In: IEEE. **High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for**. [S.l.], 2016. p. 819–829.

LIU, Z. et al. Profiling and Improving I/O Performance of a Large-Scale Climate Scientific Application. In: **2013 22nd International Conference on Computer Communication and Networks (ICCCN)**. [S.l.: s.n.], 2013. p. 1–7.

LOFSTEAD, J. et al. Six degrees of scientific data: Reading patterns for extreme scale science io. In: **Proceedings of the 20th International Symposium on High Performance Distributed Computing**. New York, NY, USA: Association for Computing Machinery, 2011. (HPDC '11), p. 49–60. ISBN 9781450305525. Available from Internet: <https://doi.org/10.1145/1996130.1996139>.

Los Alamos National Laboratory. **MPI-IO Test Bechmark**. 2008. <bitbucket.org/jeanbez/mpi-io-test>.

LU, Y. et al. Revealing applications' access pattern in collective I/O for cache management. In: **Proceedings of the 28th ACM International Conference**

on Supercomputing. [S.l.]: ACM Press, 2014. (ICS '14), p. 181–190. ISBN 9781450326421.

Luu, H. et al. A multi-level approach for understanding I/O activity in HPC applications. In: **2013 IEEE International Conference on Cluster Computing (CLUSTER)**. [S.l.: s.n.], 2013. p. 1–5.

MCLAY, R. et al. A User-friendly Approach for Tuning Parallel File Operations. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. IEEE Press, 2014. (SC '14), p. 229–236. Available from Internet: <https://doi.org/10.1109/SC.2014.24>.

MINGMING, H. et al. Improving I/O Performance for High Performance Computing with Application Forwarding Layer. In: ____. New York, NY, USA: Association for Computing Machinery, 2020. p. 1–5. ISBN 9781450375603. Available from Internet: <https://doi.org/10.1145/3409501.3409511>.

Miranda, A. et al. NORNS: Extending Slurm to Support Data-Driven Workflows through Asynchronous Data Staging. In: **2019 IEEE International Conference on Cluster Computing (CLUSTER)**. Albuquerque, NM, USA: IEEE, 2019. p. 1–12. ISSN 2168-9253.

NASA Advanced Supercomputing Division. **NAS Parallel Benchmarks**. 2003. Available from Internet: <https://www.nas.nasa.gov/publications/npb.html>.

NOU, R.; GIRALT, J.; CORTES, T. Automatic I/O scheduler selection through online workload analysis. In: IEEE. **9th International Conference on Autonomic and Trusted Computing**. [S.l.], 2012. p. 431–438.

OHTA, K. et al. Optimization Techniques at the I/O Forwarding Layer. In: INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING, 2009, Heraklion, Crete. **2010 IEEE International Conference on Cluster Computing**. [S.l.]: IEEE, 2010. p. 312–321.

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

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

REN, K. et al. Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In: **SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. [S.l.: s.n.], 2014. p. 237–248.

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

ROSS, R. et al. Storage systems and input/output: Organizing, storing, and accessing data for scientific discovery. report for the doe ascr workshop on storage systems and i/o. [full workshop report]. 9 2018. Available from Internet: <https://www.osti.gov/biblio/1491994>.

SCHMUCK, F.; HASKIN, R. Gpfs: A shared-disk file system for large computing clusters. In: **Proceedings of the 1st USENIX Conference on File and Storage Technologies**. USA: USENIX Association, 2002. (FAST '02), p. 19–es.

SONG, H. et al. A cost-intelligent application-specific data layout scheme for parallel file systems. In: **Proceedings of the 20th International Symposium on High Performance Distributed Computing**. [S.l.]: ACM, 2011. (HPDC '11), p. 37–48. ISSN 1386-7857.

SONG, H. et al. Server-Side I/O Coordination for Parallel File Systems. In: **Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis**. New York, NY, USA: Association for Computing Machinery, 2011. (SC '11). ISBN 9781450307710. Available from Internet: <https://doi.org/10.1145/2063384.2063407>.

SOUMAGNE, J. et al. Mercury: Enabling remote procedure call for high-performance computing. In: **2013 IEEE International Conference on Cluster Computing (CLUSTER)**. [S.l.: s.n.], 2013. p. 1–8. ISSN 1552-5244.

SPEARMAN, C. The Proof and Measurement of Association between Two Things. **The American Journal of Psychology**, University of Illinois Press, v. 15, n. 1, p. 72–101, 1904. Available from Internet: <http://www.jstor.org/stable/1412159>.

STENDER, J. et al. Striping without sacrifices: Maintaining posix semantics in a parallel file system. In: **First USENIX Workshop on Large-Scale Computing**. USA: USENIX Association, 2008. (LASCO'08).

STROBL, C. et al. Conditional variable importance for random forests. **BMC Bioinformatics**, v. 9, n. 1, p. 307, Jul 2008.

SUGIYAMA, S.; WALLACE, D. Cray DVS: Data Virtualization Service. 2008.

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

SUTTON, R. S.; BARTO, A. G. **Reinforcement Learning: An Introduction**. 2017. <http://incompleteideas.net/book/the-book-2nd.html>. Accessed: August 2018.

TANG, H. et al. Improving Read Performance with Online Access Pattern Analysis and Prefetching. In: **Euro-Par 2014**. [S.l.]: Springer, 2014. p. 246–257.

TESSIER, F.; VISHWANATH, V.; JEANNOT, E. TAPIOCA: An I/O Library for Optimized Topology-Aware Data Aggregation on Large-Scale Supercomputers. In: **2017 IEEE International Conference on Cluster Computing (CLUSTER)**. [S.l.]: IEEE, 2017. p. 70–80.

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

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

VEF, M.-A. et al. GekkoFS - a temporary distributed file system for HPC applications. In: IEEE. **2018 IEEE International Conference on Cluster Computing (CLUSTER)**. Belfast, UK: IEEE, 2018. p. 319–324. ISSN 2168-9253.

VEF, M.-A. et al. GekkoFS—A temporary burst buffer file system for HPC applications. **Journal of Computer Science and Technology**, Springer, v. 35, n. 1, p. 72–91, 2020.

VIGIL, M. **Trinity Platform Introductioon and Usage Model**. 2015. <https://www.lanl.gov/projects/trinity/_assets/docs/trinity-usage-model-presentation.pdf>. [Online; Accessed 7-January-2020].

VISHWANATH, V. et al. Accelerating i/o forwarding in ibm blue gene/p systems. In: **Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis**. USA: IEEE Computer Society, 2010. (SC '10), p. 1–10. ISBN 9781424475599. Available from Internet: <https://doi.org/10.1109/SC.2010.8>.

VISHWANATH, V. et al. Topology-aware data movement and staging for i/o acceleration on blue gene/p supercomputing systems. In: **Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis**. New York, NY, USA: Association for Computing Machinery, 2011. (SC '11). ISBN 9781450307710. Available from Internet: <https://doi.org/10.1145/2063384.2063409>.

WALKO, R. L.; AVISSAR, R. The Ocean–Land–Atmosphere Model (OLAM). Part I: Shallow-Water Tests. **Monthly Weather Review**, v. 136, n. 11, p. 4033–4044, 2008.

WANG, Z. et al. Iteration based collective I/O strategy for Parallel I/O systems. In: **CCGRID '14 Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing**. [S.l.]: IEEE, 2014. p. 287–294. ISBN 9781479927838.

WATKINS, C. J. C. H. **Learning from Delayed Rewards**. Thesis (PhD) — King's College, Cambridge, UK, May 1989. Available from Internet: <http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf>.

WELCH, B. et al. Scalable performance of the panasas parallel file system. In: **Proceedings of the 6th USENIX Conference on File and Storage Technologies**. USA: USENIX Association, 2008. (FAST'08).

XU, W. et al. Hybrid hierarchy storage system in MilkyWay-2 supercomputer. **Frontiers of Computer Science**, Higher Education Press, v. 8, n. 3, p. 367–377, 2014. ISSN 2095-2228.

YANG, B. et al. End-to-End I/O Monitoring on a Leading Supercomputer. In: **Proceedings of the 16th USENIX Conference on Networked Systems Design and**

Implementation. USA: USENIX Association, 2019. (NSDI'19), p. 379–394. ISBN 9781931971492.

YANG, B. et al. End-to-end I/O Monitoring on a Leading Supercomputer. In: **16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)**. Boston, MA: USENIX Association, 2019. p. 379–394. ISBN 978-1-931971-49-2. Available from Internet: <https://www.usenix.org/conference/nsdi19/presentation/yang>.

YEO, I.-K.; JOHNSON, R. A. A New Family of Power Transformations to Improve Normality or Symmetry. **Biometrika**, [Oxford University Press, Biometrika Trust], v. 87, n. 4, p. 954–959, 2000. Available from Internet: <http://www.jstor.org/stable/2673623>.

Yildiz, O. et al. On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems. In: **2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. Chicago, IL, USA: IEEE, 2016. p. 750–759.

YIN, Y. et al. Pattern-direct and layout-aware replication scheme for parallel i/o systems. In: **Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing**. USA: IEEE Computer Society, 2013. (IPDPS '13), p. 345–356. ISBN 9780769549712. Available from Internet: <https://doi.org/10.1109/IPDPS.2013.114>.

Yu, J. et al. Further exploit the potential of i/o forwarding by employing file striping. In: **2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)**. Guangzhou, China: IEEE, 2017. p. 322–330.

Yu, J. et al. Further exploit the potential of I/O forwarding by employing file striping. In: **2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)**. Guangzhou, China: IEEE, 2017. p. 322–330.

Yu, J. et al. On the load imbalance problem of I/O forwarding layer in HPC systems. In: **2017 3rd IEEE International Conference on Computer and Communications (ICCC)**. Chengdu, China: IEEE, 2017. p. 2424–2428.

YU, J. et al. Cross-layer coordination in the I/O software stack of extreme-scale systems. **Concurrency and Computation: Practice and Experience**, v. 30, n. 10, 2018. Available from Internet: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4396>.

ZHANG, F. et al. Performance Improvement of Distributed Systems by Autotuning of the Configuration Parameters. **Tsinghua Science & Technology**, v. 16, n. 4, p. 440 – 448, 2011. ISSN 1007-0214.

ZHANG, X.; DAVIS, K.; JIANG, S. Iorchestrator: Improving the performance of multi-node i/o systems via inter-server coordination. In: **Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis**. USA: IEEE Computer Society, 2010. (SC '10), p. 1–11. ISBN 9781424475599. Available from Internet: <https://doi.org/10.1109/SC.2010.30>.

ZHOU, Z. et al. I/O-Aware Batch Scheduling for Petascale Computing Systems. In: **Proceedings of the 2015 IEEE International Conference on Cluster Computing**. USA: IEEE Computer Society, 2015. (CLUSTER '15), p. 254–263. ISBN 9781467365987. Available from Internet: <https://doi.org/10.1109/CLUSTER.2015.45>.

ZIMMER, C.; GUPTA, S.; LARREA, V. G. V. Finally, A Way to Measure Frontend I/O Performance. In: CRAY USER GROUP MEETING, 2016, London, UK. **Cray User Group Proceedings**. [S.l.], 2016. p. 1–8.

APPENDIX A — RESUMO EXPANDIDO EM PORTUGUÊS

In this appendix, we present a summary of this Ph.D. Thesis, in Portuguese, as required by the PPGC Graduate Program in Computing at UFRGS.

Neste apêndice, apresentamos um resumo desta Tese de Doutorado, em Português, conforme determinado pelo Programa de Pós-Graduação em Computação da UFRGS.

A.1 Motivação

Aplicações científicas demandam cada vez mais desempenho dos sistemas de Processamento de Alto Desempenho (PAD). Além disso, cargas de trabalho cada vez mais heterogêneas estão presentes em plataformas PAD, de aplicações de Big Data até aplicações que utilizam aprendizado de máquina, tornando os sistemas mais complexos do que nunca. Esses novos requisitos justificam as atualizações contínuas e a instalação de novas plataformas de grande escala. No entanto, na medida que a complexidade desses sistemas tende a crescer, também aumenta o número de parâmetros e fatores que podem afetar direta ou indiretamente o desempenho das aplicações e do sistema. O relatório "Oportunidades e Desafios da Computação Exascale"[1] (tradução nossa) apresentado pelo Departamento de Energia dos EUA (DOE) afirmou que o problema *Exascale* (criar sistemas de computação capazes de calcular pelo menos 10^{18} operações de ponto flutuante por segundo) é mais do que apenas uma questão de escala. O comportamento e o desempenho das aplicações serão determinados por uma interação complexa do código, processador, memória, rede de interconexão e operações de entrada e saída (E/S) (DOE, 2010). Portanto, obter desempenho em escala requer a capacidade de orquestar estes componentes de forma otimizada, além de possuir uma visão completa do sistema para entender as causas de possíveis perdas de desempenho.

Com relação ao gerenciamento de sistemas de armazenamento, o relatório "Sistemas de Armazenamento e E/S: Organização, Armazenamento e Acesso a Dados para Descoberta Científica"[2] (tradução nossa) (ROSS et al., 2018) reconhece a crescente popularidade do aprendizado de máquina como uma ferramenta dentro da comunidade de sistemas, que introduz novas possibilidades na aplicação de modelos estatísticos para

[1] *"The Opportunities and Challenges of Exascale Computing"*
[2] *"Storage Systems and I/O: Organizing, Storing, and Accessing Data for Scientific Discovery"*

gerenciamento de sistemas de armazenamento. O relatório menciona esforços que podem contribuir para o uso mais eficaz do hardware de armazenamento atual e futuro. Entre eles, destacamos os sistemas de armazenamento capazes de responder às demandas de carga de trabalho cíclicas, conjuntos de ferramentas orientados por políticas capazes de gerenciar o compartilhamento de recursos e novos recursos para permitir a rápida reorganização de dados. A pesquisa apresentada neste documento está alinhada com essas tendências emergentes.

Como as operações de E/S são um gargalo para um número cada vez maior de aplicações científicas, elas têm o potencial de impactar criticamente o desempenho das aplicações na próxima geração de supercomputadores. Enquanto os clusters de larga escala normalmente dependem de uma infraestrutura de armazenamento compartilhada gerenciada por um Sitema de Arquivos Paralelos (SAP), como o Lustre (SUN, 2007), GPFS (SCHMUCK; HASKIN, 2002) ou Panasas (WELCH et al., 2008), as crescentes demandas de E/S das aplicações provenientes de áreas do conhecimento fundamentalmente distintas exigem muito desta infraestrutura compartilhada. Conforme os sistemas aumentam em número de nós de computação para acomodar aplicações maiores e mais execuções simultâneas, os sistemas de arquivos não são capazes de continuar fornecendo desempenho devido ao aumento da contenção e interferência (XU et al., 2014; Kougkas et al., 2016; Yildiz et al., 2016; YU et al., 2018; YANG et al., 2019a).

Para mitigar esse problema, a técnica de encaminhamento de E/S (também conhecida como *I/O forwarding*) (ALMÁSI et al., 2003) busca reduzir o número de nós que acessam simultaneamente os servidores de dados do sistema de arquivos, criando uma camada adicional entre os nós de computação e os servidores de dados. Assim, ao invés das aplicações acessarem o SAP diretamente, a técnica de encaminhamento de E/S define um conjunto de *nós de I/O* que são responsáveis por receber requisições de E/S das aplicações e encaminhá-los para o sistema de arquivos paralelo de maneira controlada. Isso permite a aplicação de técnicas de otimização, como escalonamento de requisições, agregação e compactação. Além disso, sua presença em um sistema de larga escala é transparente para as aplicações e independente do sistemas de arquivos existentes na plataforma. Devido a esses benefícios, a técnica de encaminhamento é aplicada por vários supercomputadores do Top 500[3] conforme detalhado na Tabela A.1.

As técnicas de otimização de E/S (incluindo, mas não se limitando à camada de encaminhamento de E/S) geralmente fornecem melhorias de desempenho para sistemas

[3]Novembro 2019 TOP500: <https://www.top500.org/lists/2019/06/>.

Table A.1: TOP 500 supercomputadores que utilizam encaminhamento de E/S.

Rank	Supercomputador	Nodos de Computação	Nodos de E/S
3	Sunway TaihuLight (YANG et al., 2019b)	40,960	240
4	Tianhe-2A (XU et al., 2014)	16,000	256
6	Piz Daint (GORINI; CHESI; PONTI, 2017)	6,751	54
7	Trinity (VIGIL, 2015)	19,420	576
13	Sequoia (PRABHAT; KOZIOL, 2014)	98,304	768

Fonte: TOP 500 Novembro 2019, e Ji et al. (2019).

com determinadas configurações ou para aplicações que usam determinados padrões de acesso, mas não para todos eles. Chamamos de *padrão de acesso* a forma como uma aplicação faz suas operações de E/S: número de arquivos acessados, espacialidade dos acessos (contíguo, 1D-strided, etc) e o tamanho das requisições. Além disso, tais otimizações frequentemente dependem da escolha correta de parâmetros (por exemplo, o tamanho do buffer para operações coletivas utilizando MPI-IO). No entanto, a responsabilidade de fazer essa escolha geralmente é do usuário final. As técnicas não fornecem melhorias para todos os padrões porque são projetadas para explorar características específicas de sistemas e cargas de trabalho (MCLAY et al., 2014). Boito et al. (2016) e Bez et al. (2017) demonstram isso para o escalonamneto de requisições em diferentes níveis da pilha de E/S. Portanto, considerando que em tais sistemas de grande escala temos um conjunto variável de aplições executando em um dado momento (com características e demandas distintas) para melhorar o desempenho com sucesso, **é essencial que o sistema seja capaz de se adaptar dinamicamente a uma carga de trabalho que também muda**. Assim, retiramos a responsabilidade da configuração de parâmetros (também conhecida por *tuning*) dos usuários, tornando o sistema capaz de se adaptar para entregar o melhor desempenho possível.

Propomos uma nova abordagem para adaptar a camada de encaminhamento de E/S à carga de trabalho de E/S que está sendo observada em um dado momento. Em nossa proposta, analisamos periodicamente as métricas de padrão de acesso coletadas pelos nós de E/S. Aplicamos uma técnica de aprendizado por reforço, chamada de Bandidos Contextuais (*Armed Bandits*) (SUTTON; BARTO, 2017), para que o sistema possa aprender a melhor escolha para cada padrão de acesso, em tempo de execução. Depois de observar um padrão por um número suficiente de vezes, o conhecimento adquirido será usado para melhorar o desempenho durante toda a vida do sistema, ou seja, por anos. Além disso, na medida que o mecanismo de aprendizagem continua atualizando seu conhecimento, ele

pode se adaptar às mudanças no sistema. A nossa proposta inova ao utilizar uma variante desta técnica denominada k-*armed bandit* juntamente com a detecção do padrão de acesso das aplicações para ajustar de forma automática e transparente os parâmetros que impactam o desempenho de E/S, na camada de encaminhamento, durante a execução das aplicações.

Ao tornar o sistema capaz de aprender em tempo de execução, eliminamos a necessidade de uma etapa de treinamento anterior. Isso é essencial, pois projetar e executar um conjunto de treinamento para representar um conjunto diversificado de aplicações que serão executados em um supercomputador, e as interações entre eles (considerando a infraestrutura de E/S compartilhada), é difícil, sujeito a erros (como poderíamos não cobrir com precisão todos os padrões e suas interações) e muito demorado.

Além disso, nessas máquinas, a camada de encaminhamento é tradicionalmente instalada em nós especiais e o mapeamento entre os nós de computação e esses nós de E/S é estático. Consequentemente, um subconjunto de nós de computação encaminhará suas requisições de E/S apenas para um único nó de E/S fixo, o que acaba forçando as aplicações a utilizarem o encaminhamento de E/S com um número predefinido de nós de E/S, mesmo que essa decisão não represente a melhor escolha para uma determinada carga de trabalho. Embora essa configuração tenha como objetivo distribuir os nós de E/S entre nós de computação de maneira uniforme, ela carece de flexibilidade para se ajustar às demandas de E/S das aplicações, podendo até mesmo causar a má alocação de recursos de encaminhamento e um desequilíbrio de carga de E/S, conforme demonstrado por Yu et al. (2017c) no supercomputador Sunway TaihuLight[4] e Bez et al. (2020) no supercomputador MareNostrum 4[5].

Argumentamos a favor de uma alocação dinâmica e sob demanda de nós de E/S que considere as características de carga de trabalho das aplicações (seu padrão de acesso). Consequentemente, dado um conjunto de aplicações prontos para executar e um número fixo de recursos de encaminhamento, resolver o problema de alocação consistiria em determinar quantos nós de E/S cada um deles deveria receber para maximizar a largura de banda global agregada. A política de alocação deve ser chamada antes que novas aplicações comecem a ser executados e quando o conjunto de aplições em execução for alterado.

No entanto, devido à natureza estática das infraestruturas de encaminhamento de E/S tradicionais e às limitações inerentes envolvidas em executar aplicações em um su-

[4]<https://www.top500.org/system/178764>
[5]<https://www.top500.org/system/179067>

percomputador de produção, nem sempre é possível para os administradores do sistema explorar diferentes estratégias de alocação de E/S sem impactar negativamente as execuções dos usuários. Assim, é necessária uma solução de pesquisa/exploração que permita aos pesquisadores de E/S e administradores de sistema obter uma visão geral das vantagens ou desvantagens dos diferentes padrões de acesso em diferentes configurações de encaminhamento de E/S. Para que tal solução seja útil, ela deve ser portátil, permitir que os aplicativos existentes sejam executados sem modificações em seu código-fonte e, se possível, ser executado como um serviço em nível de usuário para simplificar a sua implantação. Como defendemos a alocação dinâmica, essa solução também deve permitir a alteração do número de nós de E/S atribuídos à uma aplicação durante sua execução, sem interrompê-la.

A.2 Contribuições

O principal objetivo de nossa pesquisa é **ajustar dinamicamente a camada de encaminhamento de E/S em plataformas de larga escala para melhorar o desempenho global**. Exploramos duas frentes que usam os padrões de acesso das aplicações como guia para tomar decisões: ajustar os parâmetros de escalonamento na camada de encaminhamento e arbitrar os nós de E/S entre o conjunto de aplicações em execução. Considerando esses objetivos, nossas contribuições são as seguintes:

- Investigamos e demonstramos como as técnicas de aprendizado de máquina (ML) (árvore de decisão, florestas aleatórias e redes neurais) podem ajudar na detecção automática dos padrões de acesso de E/S mais comuns em aplicações PAD;
- Demonstramos a aplicabilidade das estratégias de detecção na camada de encaminhamento de E/S ao ajustar um parâmetro do escalonador de E/S em que a detecção precisa do padrão de acesso é fundamental para atingir alto desempenho;
- Propomos uma nova abordagem para adaptar a camada de encaminhamento de E/S à carga de trabalho de atual, combinando Aprendizado por Reforço (Reinforcement Learning) e detecção de padrão de acesso para ajustar de forma automática e transparente os parâmetros relacionados a E/S em tempo de execução;
- Propomos uma ferramenta leve capaz de explorar a técnica de encaminhamento de E/S chamada FORGE para coletar métricas de desempenho e auxiliar na compreen-

são do impacto do encaminhamento de E/S em um sistema;

- Propomos uma política de alocação de encaminhamento de E/S baseada em uma variante do problema da mochila, o Multiple-Choice Knapsack Problem (MCKP), para arbitrar de forma otimizada os nós de E/S entre as aplicações;

- Avaliamos diferentes políticas de alocação de encaminhamento de E/S e demonstramos que uma abordagem de alocação dinâmica pode melhorar a largura de banda global e o uso do sistema, ao mesmo tempo que usa de forma eficiente os nós de E/S disponíveis;

- Apresentamos um serviço de encaminhamento de E/S denominado GekkoFWD que atua como uma camada de encaminhamento sob demanda e implementa a política de alocação MCKP. GekkoFWD é construído em cima de um sistema de arquivos ad-hoc em nível do usuário, enriquecendo-o para permitir a exploração de diferentes configurações de encaminhamento. Esta solução não requer modificações nas aplicações, além de ser simples de executar em produção.

A.3 Conclusões

Diferentes técnicas de otimização de E/S (incluindo, mas não se limitando à camada de encaminhamento de E/ S fornecem melhorias para algumas configurações de sistema e padrões de acesso, mas não para todos eles. Além disso, eles geralmente requerem um ajuste mais detalhado dos parâmetros. Nesta pesquisa, procuramos ajustar dinamicamente a camada de encaminhamento de E/S em plataformas de larga escala para melhorar o desempenho global. Exploramos duas abordagens que usam o padrão de acesso das aplicações como diretrizes para a tomada de decisões: ajustar os parâmetros de um escalonador de requisições na camada de encaminhamento e arbitrar os nós de E/S disponíveis entre o conjunto de aplicações que estão executando em um dado momento.

Demonstramos a aplicabilidade de diferentes técnicas de aprendizado de máquina para detectar automaticamente o padrão de acesso de E/S de aplicações PAD, em tempo de execução. Investigamos árvores de decisão, florestas aleatórias e redes neurais para classificar métricas coletadas em tempo de execução nos padrões de acesso mais comuns. Para ilustrar a aplicabilidade das técnicas, avaliamos essas estratégias estimando o impacto da detecção correta do padrão de acesso para ajustar o tamanho da janela do

escalonador TWINS na camada de encaminhamento. Nossos resultados mostraram que todas as abordagens de detecção avaliadas neste trabalho podem detectar corretamente o padrão de acesso. Os mecanismos de detecção alcançaram até 99% do desempenho de uma solução oráculo. Também demonstramos melhorias de aproximadamente 17%, em média, ao comparar com o uso de uma janela definida estaticamente. Por último, ao identificar corretamente o padrão de acesso em tempo de execução, fomos capazes de evitar quedas no desempenho causada por escolhas erradas de parâmetros.

Também propusemos uma nova abordagem para adaptar a camada de encaminhamento à carga de trabalho de E/S observada em um dado momento. Periodicamente, coletamoos métricas nos nós de E/S para detectar o padrão de acesso e aplicar uma técnica de aprendizado por reforço chamada bandidos contextuais. Mostramos que o sistema pode aprender a melhor escolha para cada padrão de acesso em tempo de execução, retirando a responsabilidade de configuração dos usuários. Para nosso estudo de caso (TWINS), os resultados da avaliação offline de 144 cenários mostraram que nossa abordagem é capaz de alcançar uma precisão de $\approx 88\%$ (atingindo $\approx 99\%$ do desempenho da melhor opção) nas primeiras centenas de observações de um determinado padrão de acesso. Nossa avaliação online mostrou na prática que nossa abordagem é capaz de descobrir os tamanhos de janela corretos e mostrar melhorias de tempo de execução de até $19,3\%$, evitando tamanhos de janela que podem prejudicar o desempenho.

Experimentos adicionais com a carga de E/S da aplicação MADspec, demonstraram a aplicabilidade de nosso mecanismo de aprendizagem reduzindo o impacto de uma escolha de janela errada em até 17%. Finalmente, a sobrecarga média imposta por nossa proposta foi inferior a 2%, e o tempo necessário para anunciar as métricas e chegar a uma decisão foi curto o suficiente para viabilizar a adaptação. É vital notar que a abordagem proposta nesta tese não é específica para ajustar o parâmetro de tamanho da janela TWINS. Ele pode ser aplicado a outras situações em que as informações do padrão de acesso atual são relevantes para ajustar um determinado parâmetro de configuração na pilha de E/S.

Embora o encaminhamento de I/O seja uma técnica estabelecida e amplamente adotada para reduzir a contenção e melhorar o desempenho das operações de E/S no acesso à infraestrutura de armazenamento compartilhado, nem sempre é possível explorar suas vantagens em configurações diferentes sem impactar ou interromper os sistemas de produção. Neste trabalho, também investigamos o encaminhamento de E/S considerando os padrões de acesso de aplicações e a configuração do sistema, em vez de tentar adivinhar

ou propor uma configuração única para todos as cargas de trabalho. Ao determinar quando o encaminhamento é a melhor escolha para uma dada aplicação e de quantos nós de E/S ela se beneficiaria, podemos orientar as políticas de alocação para chegar a melhores decisões.

Para entender o impacto do encaminhamento de requisições de E/S em diferentes padrões de acesso, implementamos FORGE, uma camada de encaminhamento leve em espaço do usuário. Exploramos 189 cenários diferentes, cobrindo padrões de acesso distintos e demonstramos que, o número ideal de nós de E/S varia dependendo da carga da aplicação. Enquanto para $90,5\%$ o uso encaminhamento seria a melhor opção, a alocação de apenas dois nós de E/S só traria melhorias de desempenho para 44% dos cenários. Nossos resultados nos supercomputadores MareNostrum e Santos Dumont demonstram que mudar o foco de uma implantação estática considerando todo o sistema para uma camada de encaminhamento de E/S reconfigurável e sob demanda guiada pelas demandas das aplicações pode melhorar o desempenho de E/S em máquinas futuras.

Com relação à arbitragem de recursos dos nós de E/S, argumentamos a favor de uma alocação dinâmica e sob demanda considerando as características de E/S da aplicação. Demonstramos que a implantação global da camada de encaminhamento combinada com a política de alocação estática existente baseada exclusivamente no tamanho das aplicações não é adequada para acomodar as cargas de trabalho cada vez mais heterogêneas que entram nas plataformas de larga escala. Em vez disso, as características de E/S de uma aplicação também devem ser consideradas ao arbitrar recursos de encaminhamento entre aplicações em execução para melhorar o desempenho global.

Apresentamos uma solução de encaminhamento em nível de usuário chamada GekkoFWD que não requer modificações nas aplicações, além de permitir um remapeamento dinâmico de recursos de encaminhamento para nós de computação. O GekkoFWD é simples de executar em máquinas de produção, onde essa camada ainda não está presente, visando aplicações que se beneficiariam dela. Propusemos uma nova política de alocação de encaminhamento de E/S baseada no problema da mochila de múltipla escolha (*Multiple-Choice Knapsack Problem*) (MCKP). Nesse contexto, temos várias classes que representam cada aplicação em execução e os itens em cada classe representam o número de nós de E/S que poderíamos escolher. Devemos eleger um item para cada aplicação, visando maximizar a largura de banda global.

Demonstramos a aplicabilidade de nossa política de alocação dinâmica MCKP para arbitrar os recursos de encaminhamento de E/S disponíveis por meio de extensa avali-

ação e experimentação. Mostramos que nossa solução pode melhorar de forma transparente a largura de banda de E/S global em até $23\times$ em comparação com a política estática existente, embora a melhoria da largura de banda global muitas vezes possa resultar do comprometimento do desempenho de determinadas aplicações. Além disso, observamos melhorias de até 85% em um experimento completo usando GekkoFWD e uma fila de nove aplicações científicas diferentes.