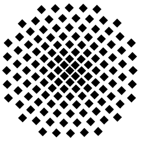


H L R I S

Institut für
Höchstleistungsrechnen

FORSCHUNGS- UND ENTWICKLUNGSBERICHT

OPTIMIZING I/O PERFORMANCE WITH MACHINE LEARNING SUPPORTED AUTO-TUNING

Ayşe Bağbaba

ISSN 0941 - 4665 Mai 2023 HLRS-25

Höchstleistungsrechenzentrum
Universität Stuttgart
Prof. Dr.-Ing. Dr. h.c. Dr. h.c. Prof. E.h. Michael M. Resch
Nobelstrasse 19 - 70569 Stuttgart
Institut für Höchstleistungsrechnen

OPTIMIZING I/O PERFORMANCE WITH MACHINE LEARNING SUPPORTED AUTO-TUNING

von der Fakultät Energie-, Verfahrens- und Biotechnik
der Universität Stuttgart zur Erlangung der Würde
eines Doktor-Ingenieurs (Dr.-Ing.) genehmigte Abhandlung
vorgelegt von

Ayşe Bağbaba
aus Ankara, Türkei

Hauptberichter: Prof. Dr.-Ing. Dr. h.c. Dr. h.c. Prof. E.h. Michael M. Resch
Mitberichter: Prof. Dr.-Ing. Stefan Wesner

Tag der Einreichung: 16. Mai 2022
Tag der mündlichen Prüfung: 16. Januar 2023

D93

ISSN 0941 - 4665 Mai 2023 HLRS-25

Declaration of Authorship

I hereby declare that I have written this dissertation entitled *Optimizing I/O Performance with Machine Learning Supported Auto-tuning* independently. I have clearly identified as quotes all passages and ideas from the literature or from other sources such as e.g. websites, and I have given the source.

Acknowledgements

There are many people whom I would like to thank for their contributions to this thesis.

In the beginning, I would like to thank my supervisor, Prof. Dr. Michael Resch, for all his guidance, support and understanding. His insightful comments pushed me to improve my thinking and my work. I am grateful to Dr. José Gracia, for the support and encouragement he gave me during my PhD study. I also appreciate the help of all colleagues at HLRS for the many inspiring discussions we have had, especially Christoph Niethammer, for his invaluable advice and feedback on my research. I am also thankful to Dr. Thomas Bönisch and Dr. Xuan Wang for always being so supportive of my work. I want to thank Dr. Loic Salles and Jiri Blahos from my secondment at Imperial College London for their collaboration.

I gratefully acknowledge the funding provided by the European Union's Horizon 2020 research and innovation program under the project EXPERTISE, the POP, and the HPC-EUROPA3.

I feel great gratitude towards my big family for their constant encouragement. I want to thank my husband, Ahmet, for always being there for me. And my mom and dad, Behiye and Mustafa, for their continuous support throughout my life. Without the love and help of you, this was not possible. Thank you!

Contents

Acknowledgements **iii**

Abstract **xi**

Zusammenfassung **xiii**

List of Figures **xv**

List of Tables **xix**

1. Introduction and Motivation **1**

1.1. Introduction 1

1.2. Motivation 2

1.2.1. Problem Definition 2

1.2.2. Related Work 4

1.2.3. The Proposed Approach 7

1.3. Organization of Dissertation 10

2. State of the Art **11**

2.1. User Applications 11

2.2. HPC Platforms 13

2.3. Distributed Parallel File Systems 13
2.3.1. Lustre 14
2.3.2. GPFS 16
2.3.3. BeeGFS 17
2.4. Parallel I/O 18
2.4.1. Overview 18
2.4.2. MPI-IO 19
2.4.3. Parallel I/O Optimizations 31

3. General I/O Auto-tuning Framework 35
3.1. Design Requirements 35
3.2. I/O Performance Factors 36
3.3. MPI and PMPI Wrapper 40
3.4. Running Modes 41

4. Heuristic Search Based I/O Auto-tuning 43
4.1. Heuristic Search 43
4.2. Architecture 45
4.2.1. IO_Optimizer: Configuration Search 46
4.2.2. IO_Tuner: Setting I/O Parameters at Runtime 47
4.3. Implementation 48
4.3.1. Benchmarks 48
4.3.2. System setup 49
4.3.3. Parameter space 49
4.3.4. Scale and data set sizes 50
4.4. Results 50

5. Performance Modelling Based I/O Auto-tuning 55
5.1. Performance Modelling 55
5.1.1. Performance Models 57
5.2. Architecture 60
5.2.1. IO_Tracer: Monitoring I/O Activity 61
5.2.2. IO_Predictor: Modelling I/O Performance 62

5.2.3. IO_Tuner: Setting I/O Parameters at Runtime 65
5.3. Implementation 65
5.3.1. Benchmarks 65
5.3.2. System setup 66
5.3.3. Parameter space 66
5.3.4. Scale and data set sizes 68
5.3.5. Log files and creating data set 68

6. Results 71
6.1. I/O Variability on Single Node 71
6.2. I/O Variability on Multiple Nodes 73
6.3. Training Process 74
6.4. Evaluations: I/O Benchmarks 79
6.5. Engineering Use Case : ls1 mardyn 84
6.5.1. Analyzing Application 84
6.5.2. Training 86
6.5.3. Optimization and Results 88
6.5.4. Conclusion 91

7. Conclusion and Future Work 93

A. Code and File Segments 97

Bibliography 107

Acronyms

ADIOS Adaptable I/O System

CFD Computational Fluid Dynamics

CPU Central Processing Unit

CSV Comma-Separated Values

DKRZ German Climate Computing Centre

FLOPS Floating Point Operations Per Second

GPFS General Parallel File System

HDF5 Hierarchical Data Format 5

HLRS High-Performance Computing Center Stuttgart

HPC High-Performance Computing

I/O Input and Output

IBM International Business Machines Corporation

IOR Interleaved-Or-Random

JSON JavaScript Object Notation

MDS MetaData Server

MDT MetaData Target

MGS Management Server

MPI Message Passing Interface

NetCDF Network Common Data Form

OSC Object Storage Client

OSS Object Storage Server

OST Object Storage Target

POSIX IEEE Portable Operating System Interface for UniX

RAID Redundant Array of Inexpensive Disks

SIOX Scalable I/O for extreme performance

UHH University of Hamburg

XML EXtensible Markup Language

ZIH Center for Information Services

Abstract

Data access is a considerable challenge because of the scalability limitation of I/O. In addition, some applications spend most of their total execution times in I/O. This causes a massive slowdown and wastage of useful computing resources. Unfortunately, there is not any one-size-fits-all solution to the I/O problems, so I/O becomes a limiting factor for such applications.

Parallel I/O is an essential technique for scientific applications running on high-performance computing systems. Typically, parallel I/O stacks offer many parameters that need to be tuned to achieve an I/O performance as good as possible. Unfortunately, there is no default best configuration of these parameters; in practice, these differ not only between systems but often also from one application use case to the other. However, scientific users might not have the time or the experience to explore the parameter space sensibly and choose a proper configuration for each application use case. I present a line of solutions to this problem containing a machine learning supported auto-tuning system which uses performance modelling to optimize I/O performance. I demonstrate the value of these solutions across applications and at scale.

Zusammenfassung

Der Datenzugriff ist aufgrund der Skalierbarkeitsbeschränkung von E/A eine beträchtliche Herausforderung. Darüber hinaus verbringen einige Anwendungen den größten Teil ihrer Gesamtausführungszeit mit E/A. Dies führt zu einer massiven Verlangsamung und Verschwendung nützlicher Rechenressourcen. Leider gibt es keine Einheitslösung für die I/O-Probleme, sodass I/O für solche Anwendungen zu einem einschränkenden Faktor wird.

Parallele I/O ist eine wesentliche Technik für wissenschaftliche Anwendungen, die auf Hochleistungscomputersystemen ausgeführt werden. Typischerweise bieten parallele I/O-Stacks viele Parameter, die abgestimmt werden müssen, um eine möglichst gute I/O-Leistung zu erzielen. Leider gibt es keine standardmäßige beste Konfiguration dieser Parameter; in der Praxis unterscheiden sich diese nicht nur zwischen Systemen, sondern oft auch von Anwendungsfall zu Anwendungsfall. Allerdings haben wissenschaftliche Benutzer möglicherweise nicht die Zeit oder die Erfahrung, den Parameterraum sinnvoll zu erkunden und eine geeignete Konfiguration für jeden Anwendungsfall auszuwählen. Ich stelle eine Reihe von Lösungen für dieses Problem vor, die ein durch maschinelles Lernen unterstütztes Autotuning-System enthalten, das die Leistungsmodellierung verwendet, um die E/A-Leistung zu optimieren. Ich demonstriere den Wert dieser Lösungen anwendungsübergreifend und in großem Maßstab.

List of Figures

1.1. Typical I/O stack of an HPC system. 2

2.1. System usage of different professional areas at HLRS in 2019. 12
2.2. Components of a Lustre file system. 14
2.3. Lustre file system striping mechanism. 16
2.4. Components of a GPFS. 17
2.5. Type maps of contiguous, vector and indexed datatypes for given base type. 23
2.6. Each process has its own view of the file, defined by a *displacement*, an *elementary type*, and a *file type*. 24
2.7. Writing contiguous data into a contiguous block defined by a file view. 24
2.8. Writing contiguous data into two separate blocks defined by a file view. 25
2.9. Collective I/O operations. 26
2.10. The abstracted ROMIO architecture. 27
2.11. The abstracted architecture of OMPIO frameworks and modules. 28
2.12. File write performances for contiguous, strided and indexed data layouts on OMPIO and ROMIO. 28

2.13. Collective and individual file read performances with the MPI-Tile-I/O benchmark. 30
2.14. Data sieving. 32
2.15. Reading a distributed array by using two-phase I/O. 33

3.1. I/O simulation results by applying different MPI hints. 38
3.2. I/O simulation results by applying different Lustre stripe size. 38
3.3. MPI wrapper for MPI_Init() flow chart. 40

4.1. Procedure of genetic algorithm. 44
4.2. Overall architecture of the heuristic search-based auto-tuning approach. 45
4.3. Overall architecture of the *IO_Search*. 47
4.4. Optimizing process. 48
4.5. Default vs. optimized write bandwidth on the IOR for various transfer sizes running on 240 cores and 1,200 cores of Vulcan. Y-axis represents I/O bandwidth in MBps and x-axis represents transfer sizes (in MB). The scales of the I/O bandwidth axes are different in the plots. 52
4.6. Default vs. optimized write bandwidth on the MPI-Tile-IO for various transfer sizes running on 64 cores and 256 cores of Vulcan. Y-axis represents I/O bandwidth in MBps and x-axis represents element sizes in number of tiles (in KB). The scales of the I/O bandwidth axes are different in the plots. 53

5.1. Basic structure of a decision tree [60]. 57
5.2. The schematic diagram of the random forest regression. . . . 60
5.3. Overall architecture of performance modelling-based I/O auto-tuning. 61
5.4. Tracing process of the *IO_Tracer*. 62
5.5. Tracing and optimizing processes. 63
5.6. Implementation of the performance model. 63

6.1. I/O performance variability and effect of interference on a single node writing to a file. 72
6.2. I/O performance variability and effect of interference on multiple nodes writing to a file. 73
6.3. Correlation between actual (observed) and predicted write bandwidths on training (70%) and testing subsets (30%). . 76
6.4. Default vs. optimized write bandwidth on the IOR for various transfer sizes. 82
6.5. Default vs. optimized write bandwidth on the MPI-Tile-IO for various transfer sizes. 83
6.6. ls1 mardyn performance output. 85
6.7. Default setup vs. optimizing for ls1 mardyn checkpointing time. 86
6.8. Default vs. optimized write times for checkpointing on ls1 Mardyn for various data sizes. 90

List of Tables

3.1. Configurations' search scope 39

4.1. A list of the tunable parameters and ranges used for experiments. The last column shows the number of distinct values used for each parameter. 50
4.2. I/O speedups of applications with optimized parameters over default parameters. 54

5.1. Configurations' searching scope for training process 67
5.2. Breakdown of training set for the I/O model. 68

6.1. Comparison of regression algorithms on I/O performance prediction. 74
6.2. The *IO_Predictor* performance modelling times and validation results. 77
6.3. Found good configurations for different file size groups. . . . 78
6.4. I/O speedups of applications with optimized parameters over default parameters. 81
6.5. Found some good configurations for the IOR and the MPI-Tile-IO benchmarks. 81

6.6. A fragment of a CSV file obtained after tracing process on ls1 mardyn. 88
6.7. Configurations' searching scope for training process ls1 mardyn. 88
6.8. Found optimal configurations after training process ls1 mardyn. 88

Chapter 1

Introduction and Motivation

1.1. Introduction

Data-intensive scientific applications running on high-performance computing (HPC) systems are correspondingly bottlenecked by the time that it takes to perform input and output (I/O) [1] of data from/to the file system. Moreover, some applications spend most of their total execution times in I/O [2]. This causes a massive slowdown and wastage of useful computing resources. Thus, I/O becomes probably the most limiting factor for such applications [3].

Figure 1.1 indicates a typical parallel I/O stack of many current HPC systems that is in order of user application; high-level I/O library, such as parallel Hierarchical Data Format version 5 (HDF5) [4]; I/O middleware, such as Message Passing Interface I/O (MPI-IO) [5]; low-level I/O library, such as Portable Operating System Interface I/O (POSIX-IO) [6]; parallel file systems, such as Lustre [7] and storage hardware. A parallel I/O stack

offers many optimizations that can help in improving the performance of I/O. Because of the theoretical hardware limits, various algorithms have been developed within each layer of the parallel I/O stack. Understanding the I/O behaviours of user applications and the specification of the parallel I/O stack optimizations help the HPC systems to increase their efficiency. However, analyzing the I/O activity and achieving efficient parallel I/O are challenging tasks due to the complex inter-dependencies between the layers of the parallel I/O stack.

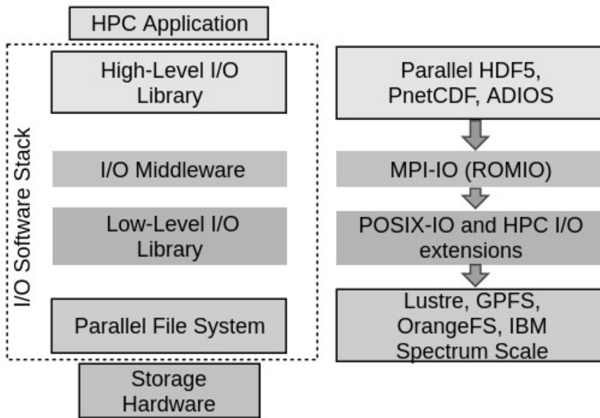


Figure 1.1.: Typical I/O stack of an HPC system.

1.2. Motivation

1.2.1. Problem Definition

Each layer of the parallel I/O stack offers a set of tunable parameters to improve I/O performance, such as Lustre file system stripe size and count, the optimization of collective I/O, the number of data aggregators, buffer size, and so on. The configuration of these parameters depends on diverse factors such as the application, storage hardware, problem size, and concurrency [8], [9], [10]. Unfortunately, there is no default best configuration of these

parameters; in practice, these differ not only between systems but often also from one application use case to the other.

When a parameter space gets larger, it becomes difficult to monitor the interactions between configuration options. Users who are not experts on the parallel I/O stack might have no time or experience for tuning their applications to the optimal level. Sometimes they might even drop down the I/O performance by mistake [8]. Application developers work on application code optimizations rather than I/O optimization [10]. In most cases, the default settings are used, often leading to poor I/O efficiency on parallel file systems [9]. As the complexity of large-scale applications and HPC systems increases, this brings more challenges in achieving high-performance I/O due to the lack of global optimizations. Available I/O profiling tools can not tell the optimal system default setups by easily monitoring and analyzing applications [9]. Identifying sources of I/O performance bottlenecks requires a multi-layer view of the I/O activity in user applications [11]. These issues make it increasingly difficult for users to find out the good configuration settings for their applications and use cases. Therefore, every achievement in I/O optimization in any layer of the parallel I/O stack is of the utmost importance to scientists and application users.

In the long benchmarking step with various I/O benchmarks, a lot of potential I/O optimization has been identified to improve the efficiency of HPC systems by avoiding unsuccessful tuning attempts. In order to increase the HPC system's efficiency, the following five requirements should be considered:

- Hiding the complexity of the parallel I/O stack layers from users
- Avoiding the configurations leading to a poor I/O performance
- Finding good configurations in a reasonable time
- Improving the application I/O performance transparently
- Considering the dynamic runtime conditions of a parallel I/O system

To address these requirements, auto-tuning can help users by automatically tuning good I/O parameters at various layers [12]. There are many

auto-tuning works to improve I/O performance which are based on various approaches such as heuristic search, analytical models, empirical models, etc. However, some of these approaches are time-consuming and not applicable. Thus, there is still a need for new studies that automatically improve the I/O performance of user applications and save core hours of HPC systems.

1.2.2. Related Work

Among various optimizing potentialities, the I/O optimization is mostly requested. Therefore, several approaches exist to determine good configurations in a large parameter search space through auto-tuning to improve the I/O performance.

Darshan[1] is designed as a scalable HPC I/O characterization tool to analyze application I/O behaviour and properties, such as access patterns within the files with minimum overhead [13], [14]. Prior Darshan versions (in versions before Darshan 3.0.0) have instrumented data from the POSIX, and MPI-IO layers of the I/O stack, as well as the high-level HDF5 and Parallel netCDF data interface layers [13]. Darshan can instrument I/O functions in executables that are both statically and dynamically. Darshan provides a post-processing utility to get helpful reports. To determine the good configurations for an HPC system, system administrators need to implement I/O applications, save the I/O configurations, and understand the Darshan reports. Darshan can be used for I/O monitoring and analysis; however, it cannot determine and tune values of I/O configurations.

Scalable I/O for extreme performance (SIOX) [15], developed by HLRS, ZIH[2], UHH[3], DKRZ[4] and IBM[5], offers a real-time parallel I/O optimization. SIOX assigns an I/O tracing thread running on each compute node to monitor real-time. SIOX modular architecture covers instrumentation of POSIX, MPI, and other high-level I/O libraries; the monitoring data is recorded

[1] https://www.mcs.anl.gov/research/projects/darshan/
[2] http://tu-dresden.de/zih
[3] http://www.uni-hamburg.de/
[4] http://www.dkrz.de/
[5] http://www.ibm.com/de-de/

asynchronously into a global database and recorded traces can be visualized [15]. However, the overhead produced by the MPI instrumentation is too high in production environment.

Behzad et al. approached the idea of pattern-driven parallel I/O tuning to optimize the I/O performance of HDF5 applications across platforms automatically [16]. This auto-tuning framework begins by tracing high-level I/O accesses and analyzing data write patterns. The framework then chooses the best-performing configurations at runtime based on these patterns and the available tuning parameters for similar patterns. If the previous history for a pattern is not available, the framework initiates model-based training to obtain an efficient set of tuning parameters. The framework includes a runtime system to apply the selected configurations using dynamic linking without the need for changing the application source code. This framework works only for HDF5 applications, while various engineering applications also use MPI-IO. A solution based on the MPI-IO library could be more widely used.

[8] presented a semi-automatically I/O-tuning solution for engineering applications. Standing upon the MPI-IO library allows it to be compatible with MPI-IO-based high-level I/O libraries, such as parallel HDF5 and parallel NetCDF. However, in auto-tuning, using a naïve strategy, running an application using all possible combinations of tunable parameters to find the best is like a bit of manual tweaking. It is an exhaustive search through the large parameter space with many thousands of combinations and is infeasible because of the long execution times of trial runs. This becomes a highly time and resource-consuming approach depending on the size of the parameter space. Using statistical methods in such a framework would improve I/O performance and save core hours. Investigating how to improve the ranking of configurations and improve I/O performance is still an essential issue in auto-tuning.

Several studies have attempted to include various approaches, such as heuristic search, machine learning, analytical models, etc., in the I/O analysis and optimization steps. However, due to the complexity of the state-of-the-art file systems, using analytical models is often inadequate and time-consuming

for expected predictive accuracy [17]. Upon this, many researchers have focused on empirical and machine learning approaches to model the I/O performance. In [18], Behzad et al. worked towards an auto-tuning framework for determining the parallel I/O parameters that achieves good I/O performance for different data write patterns. They characterized parallel I/O and discussed predictive models to reduce the parameter space effectively. Isaila et al. [19] integrated analytical and machine learning approaches to model the performance of ROMIO collectives. In [20], [21] Behzad et al. implemented a genetic algorithm-based I/O auto-tuning to traverse the search space systematically. In [22], manual-tuning HDF5 applications was studied; it is unmanageable for users and application developers. Megha et al. used Bayesian optimization and performance prediction for tuning parameters automatically [10]. However, due to application-specific parameter values, most of these approaches are time-consuming and could not be applied to each application use case.

In [23], Ryan et al. used supervised machine learning algorithms (i.e., decision trees, random forest, and k-nearest neighbours) to predict performance characteristics such as runtime and I/O traffic of batch jobs on high-end clusters, using only user job scripts as input. It is shown that decision trees can accurately predict the runtime of 73% of jobs. PRIONN [24] is another study that automates the prediction of per-job runtime and I/O resource usage, enabling IO-aware scheduling on HPC systems. The tool's novelty is the ability to feed entire job scripts into deep learning models, allowing for complete automation of runtime and I/O resource predictions. It shows the power of PRIONN by applying runtime and I/O resource predictions to IO-aware scheduling for real HPC data. In [25], a Gaussian process-based machine learning algorithm was implemented for I/O performance modelling and variability as a function of parallel file system and application characteristics. The results demonstrate that the presented sensitivity-based models are more promising at prediction when compared with application-partitioned or unpartitioned models. Furthermore, it shows modelling techniques robust to the outliers that may occur in production parallel file systems. This study provides insights into the file system metrics that have a significant impact

on I/O performance by using the developed metrics and modelling approach.

Several studies have attempted to include statistical modelling and machine learning in the I/O auto-tuning. For example, Anatoliy et al. developed a theory, methodology, and tools for automated program design, synthesis, and auto-tuning, based on Glushkov's systems of algorithmic algebras and term rewriting technique [26]. Furthermore, the TuningGenie auto-tuning framework and modelling for parallel programs were proposed in [26] to automate the adjustment of programs to a target platform. However, due to the empirical evaluation of many parameter combinations of an initial parallel program in a target environment, auto-tuning for complex and nontrivial parallel systems is usually time-consuming. In [27] and [28], they extend their approach with statistical modelling and neural network algorithms that allow reducing the space of possible parameter combinations significantly. The improvement automatically trains a neural network model on "traditional" tuning steps and replaces some auto-tuning calls with statistical model evaluations. While some of these studies are very complex, some have limitations in their applicability.

At this point, there is a demand for an auto-tuning solution to search I/O parameter space effectively and auto-tune good configurations transparently to the users for MPI-IO and Lustre parallel file systems, which are widely used in scientific applications. It would also offer optimization possibilities for the other file systems such as IBM Spectrum Scale and BeeGFS. Furthermore, standing upon the MPI-IO library allows it to be compatible with MPI-IO-based high-level I/O libraries, such as parallel HDF5 and parallel NetCDF.

1.2.3. The Proposed Approach

I researched and attempted to find a solution to solve the previously mentioned problems and fulfil the five requirements in Section 1.2.1. However, some solutions described in Section 1.2.2 can not be used for general engineering applications, and some can not run with production processes due to the high overhead. Furthermore, available I/O profiling tools can not tell the optimal system default setups and provide statistical information to system

administrators by easily monitoring and analyzing applications.

Finding good I/O configurations is challenging due to the large parameter space in today's HPC systems and the potentially long execution time of a trial run. In addition, the size of the parameter space can increase exponentially and become unmanageably large for brute-force approaches, depending on the granularity with which the parameter values are set.

A simple strategy for finding good I/O configurations is to run a user application or a representative I/O kernel with all possible combinations of tunable parameters for all layers of the I/O stack. However, because a typical parameter space has many thousands of combinations, this is an extremely time- and resource-consuming approach. In order to see if a combination gives good I/O performance, it is necessary to run the application with this configuration and evaluate the results. However, we also use resources for combinations that do not perform well for the I/O in this case. Therefore, instead of relying on the most straightforward approach, I have worked on two different approaches:

1. **Heuristic search with *IO_Search***: Adaptive heuristic search methods can traverse a parameter space and recommend a good set of parameters for a problem. The *IO_Search* uses a "genetic algorithm" for sampling the parameter space by testing a set of combinations and then adjusting the combination for the next runs to maximize the I/O performance with an objective function.

2. **Performance modelling with *IO_Predict***: Regression methods in machine learning can investigate the relationship between variables and outcomes. A regression model can be used for predictive modelling, in which an algorithm is used to predict outcomes. The *IO_Predict* uses a "random forest algorithm" to build a model for I/O performance. The good set of parameter values is determined by using predicted values by the *IO_Predict* rather than running the application.

Both approaches recommend a good set of parameter values for an application on a given system [29–31]. These approaches have been currently tested

with Lustre parameters and MPI-IO hints. The first approach runs the application to determine parameter values, whereas the second approach uses an I/O performance prediction model. With the second approach, the time is significantly reduced compared to the first approach for the training step. Furthermore, both models provide flexibility to focus on the improvement of I/O performance.

This study makes the following contributions:

- Design and implementation of an I/O auto-tuning system that can hide the complexity of multiple I/O stack layers from users and improve I/O performance
- Development of an approach to search the huge parameter space for good configurations with a small number of tests
- Development of an approach to generate an I/O performance model and extract expert knowledge from observations automatically in a negligible time
- Use of the models to reduce the parameter search space and save core hours
- Demonstration of how I/O benchmarks and engineering applications with different access patterns and problem sizes can benefit from the auto-tuning system
- Consideration of the dynamic runtime conditions of a parallel I/O system

The implementation of the auto-tuning system is currently based on the MPI-IO ROMIO library, which can be widely used and supports parallel HDF5 and parallel NetCDF applications. It is designed for Lustre parallel file system; it also offers optimization possibilities for the other file systems such as IBM Spectrum Scale and BeeGFS. The parameters discussed in this work are system-dependent, but new parameters can be easily integrated into the system.

1.3. Organization of Dissertation

This dissertation is organized as follows: Chapter 1 depicts the typical parallel I/O stack in an HPC system environment and introduces my research work. In Chapter 2, today's software technologies and their ability to accelerate parallel I/O operations are investigated in detail. Chapter 3 shows the conception of a general I/O auto-tuning system. The architecture, implementation details, and results are given in Chapter 4 for the heuristic search-based I/O auto-tuning system, while in Chapter 5 for the performance modelling-based I/O auto-tuning system. Chapter 6 presents the evaluation results and performance improvements using two popular I/O benchmarks, namely IOR and MPI-Tile-IO. Chapter 7 uses a real molecular dynamics code, namely ls1 Mardyn, as an engineering use case to present optimization results in a production environment. Finally, Chapter 8 summarizes my work and presents my future work to extend and improve the current approaches.

State of the Art

2.1. User Applications

I investigated the system usage at HLRS to determine which professional areas have the most potential to benefit from I/O optimizations. Among all professional areas, computational fluid dynamics[1] (CFD) consumed over 72% of annual computational capabilities at HLRS (Figure 2.1). For example, ANSYS Fluent[2] is a general-purpose CFD code which uses parallel HDF5 high-level I/O library, MPI-IO as well as POSIX I/O to read/write data. However, the software has few instructions for I/O performance because it is primarily concerned with solving computational problems rather than I/O optimizations. As another example, a molecular dynamics simulation code ls1 mardyn[3] is a highly scalable code that is optimized for parallel execution on supercomputing architectures aiming at investigating challenging use cases with up to trillions of molecules. The spatial distribution of the molecules may be heterogeneous and subject to rapid, unpredictable differences. This

[1] http://en.wikipedia.org/wiki/Computational_fluid_dynamics
[2] http://www.ansys.com/Products/Fluids/ANSYS-Fluent
[3] https://www.ls1-mardyn.de

is imaged by the algorithms and I/O structures as well as highly modular software engineering methods [32].

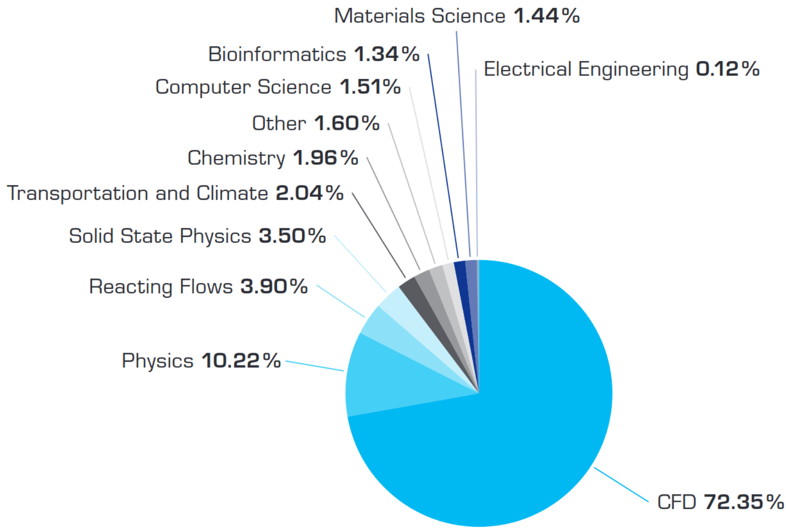


Figure 2.1.: System usage of different professional areas at HLRS in 2019.

Developers can utilize any I/O layer in self-implemented software to access parallel file systems based on their research needs. This type of software is sometimes developed on a single HPC platform but is intended to run on multiple HPC platforms. Users who are not experts might have no time or experience tuning their applications to the optimal level on different platforms. Application developers focus on application code optimizations rather than I/O optimization [10]. Most of the time, the default settings are used, which often results in poor I/O efficiency [9]. Identifying sources of I/O performance bottlenecks requires a multi-layer view of the I/O activity [11]. These issues make it more difficult for users to find out the good settings for their applications.

2.2. HPC Platforms

The experiments were conducted on Hazel Hen[1] (Cray XC40) and Vulcan[2] (the NEC Cluster platform) at HLRS[3] (High Performance Computing Center Stuttgart). The Hazel Hen was shut down on 25. February 2020.

With a peak performance of 7.42 PFLOPS, Hazel Hen was one of the most powerful HPC systems in the world (position 17 of Top500 list in June 2017). There are two Intel Xeon E5-2680 v3 CPUs on each compute node with 24 CPU cores and 128 GB shared memory installed. In addition, the Lustre file system is deployed on a Cray Sonexion scale-out Lustre storage system with 7 metadata targets (MDTs) and 54 object storage target (OSTs). The theoretical peak bandwidth of each Lustre OST is 3.75 GB/s, which leads to an aggregated 202.5 GB/s (3.75GB/s×54) peak bandwidth on the experimental Lustre file system. Vulcan consists of several front-end nodes for interactive access and several compute nodes of different types to execute parallel programs. It has 761 nodes with 24 cores, Centos 7 operating system, PBSPro Batch system, Infiniband + GigE node-node interconnect, 500 TB (Lustre) for Vulcan global disk, and the bandwidth is about 3 GB/sec. The Lustre file system consists of 54 OST storage targets, each of them one RAID6 lun, 8+PQ, 2 TB disks.

On Hazel Hen and Vulcan, various compilers (GNU and Intel), programming environments, MPI implementations, (parallel) HDF5 libraries, and (parallel) NetCDF libraries are available[4].

2.3. Distributed Parallel File Systems

To meet the demands of parallel I/O, a number of commercial and research file systems have been developed in recent years. I briefly describe some of them below.

[1] https://www.hlrs.de/de/systems/cray-xc40-hazel-hen/
[2] https://kb.hlrs.de/platforms/index.php/Vulcan
[3] https://www.hlrs.de/
[4] https://kb.hlrs.de/platforms/index.php/Platforms

2.3.1. Lustre

Lustre[1] is a GNU General Public Licensed, open-source distributed parallel file system developed and maintained by Sun Microsystems Inc. [33]. Lustre is one of the most popular distributed file systems thanks to its open architecture and high scalability. Because of these benefits, many supercomputers in the TOP500 list use this file system [34].

Figure 2.2 illustrates the Lustre architecture. Lustre clients are installed in the compute nodes or I/O nodes of an HPC system, connected with metadata servers and object storage servers via high-speed connecting networks.

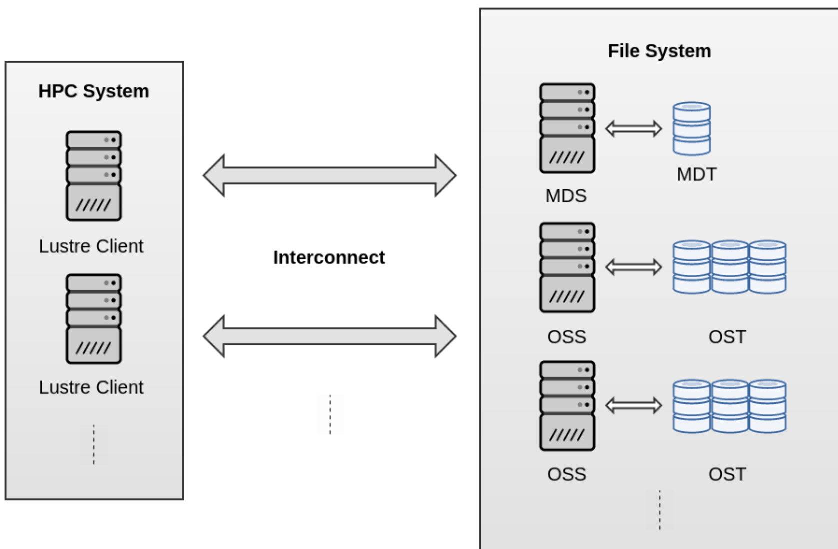


Figure 2.2.: Components of a Lustre file system.

- **MDS (metadata server)**: MDS is referred to metadata services, and a metadata client is a client of those services. One MDS per file system manages one (till Lustre software release 2.3) or multiple (since Lustre software release 2.4) metadata targets.

[1] http://www.lustre.org/

- **MDT (metadata target)**: Each MDT stores file metadata, such as file names, directory structures, and access permissions.
- **MGS (management server)**: MGS serves configuration information of the Lustre file system.
- **OSS (object storage server)**: OSS exposes block devices and serves data. Correspondingly, OSC (object storage client) is the client of the services. Each OSS controls one or more object storage targets (OSTs).
- **OST (object storage target)**: OSTs are like multiple disks connecting to OSSs where the application data are stored. The total data capacity of the Lustre parallel file system is the sum of all OST capacities.

Both MDT and OST can be constructed out of one disk or disk RAID to increase the capacity and I/O performance [8]. Users can choose how many OSTs to stripe their files over. The ability to stripe data across multiple OSTs in a round-robin fashion is one of the main factors contributing to Lustre file systems' high performance. Files can be divided into multiple chunks, which are then stored on different OSTs throughout the Lustre file system. This approach allows concurrent access to multiple OSTs, eventually accelerating the I/O requests. Besides the number of OSTs, users can also decide the stripe size, which specifies how many bytes can be stored in one OST stripe before moving to the next OST or next stripe. Different settings of these factors result in a significant difference in I/O performance.

The five segments of the linear sequence of bytes are striped across four OSTs in the physical view, as seen in Figure 2.3. Striping provides two advantages: 1) increased bandwidth because multiple processes can access the same file simultaneously, and 2) the capability to store large files that would take up more space than a single OST. On the other hand, striping has drawbacks: 1) increased network overhead and server contention; and 2) risk of file damage because of hardware failure.

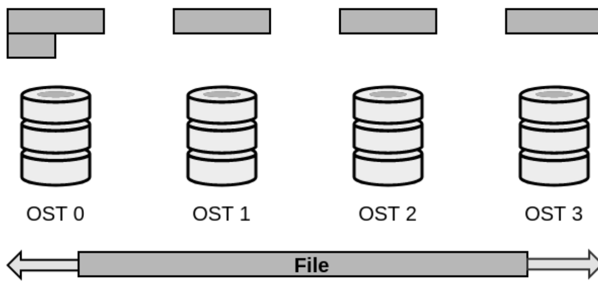


Figure 2.3.: Lustre file system striping mechanism.

2.3.2. GPFS

IBM's general parallel file system (GPFS)[1], a high-performance distributed parallel file system, was renamed Spectrum Scale. GPFS is a shared-disk file system for cluster computers. GPFS is used on many of the largest supercomputers in the world [35].

GPFS's extreme scalability is achieved through its shared-disk architecture [36]. A GPFS system is made up of cluster nodes that are connected to disks or disk subsystems via a switching fabric. All disks are accessible to all nodes in the cluster. Files are striped across all disks in the file system. Striping balances the load on the disks and achieves the maximum throughput of the disk. Figure 2.4 illustrates the GPFS architecture.

Files in GPFS are striped across all disks and divided into multiple blocks. Large files are striped and stored in blocks, while small files are stored in so-called sub-blocks. After the file system is established, the block size cannot be changed. Therefore, system administrators have to either deploy multiple file systems with varying block sizes or search for an acceptable compromise to maximize the throughput with one block size in one file system. GPFS also introduces another mechanism, page pool, for tuning its I/O performance. The page pool is an allocated part of physical memory that caches data and metadata. It supplies memory for buffering operations like prefetch (read)

[1] https://researcher.watson.ibm.com/researcher/view_group.php?id=4840

and write-behind. Its size can be very different depending on the types of nodes [8].

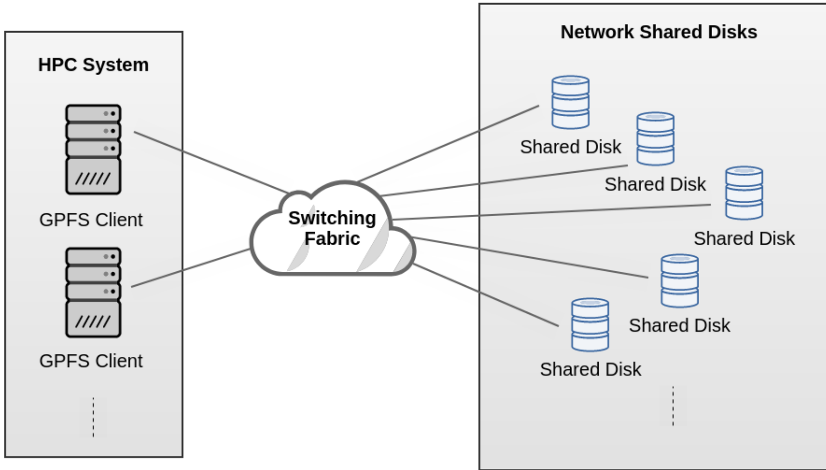


Figure 2.4.: Components of a GPFS.

2.3.3. BeeGFS

BeeGFS [1] is a parallel file system developed and optimized for high-performance computing at the Fraunhofer Center in Germany. BeeGFS is being used on a wide range of computer clusters, ranging from installations with only a few machines to several systems of the Top500 of the world's fastest supercomputers. Furthermore, the file system is a fundamental component of many research projects led by different research organizations and governmental institutions [37].

BeeGFS is a parallel cluster file system designed with a strong focus on performance and easy installation and management [36]. With the constantly increasing performance of modern processors and network technologies, processed data sets rapidly grow. In order to handle this vast amount of data

[1] https://www.beegfs.io/c/

and deliver it to the computing cores as fast as possible, the HPC community has been working on the parallel file system BeeGFS for several years now. The individual files are distributed across multiple servers chunk by chunk and, in doing so, can be read or written in parallel [37].

Users can simply scale the performance and capacity of the file system to the level that they require by increasing the number of servers and disks in the system, seamlessly scaling from small clusters to systems with thousands of nodes.

2.4. Parallel I/O

2.4.1. Overview

As the capabilities of HPC systems in achieving high-performance I/O increase, scientists use them more to solve large-scale problems that need a large amount of data and computing power [38]. For example, many parallel applications require access to large amounts of data stored in files for various reasons, including reading the initial input, writing the results, check-pointing for later restart, data analysis, and visualization [38]. In such cases, the I/O performance significantly impacts total execution time.

Prior research indicates that I/O behaviour can be a dominant factor in determining the overall performance of many HPC applications [38–44]. Understanding the parallel I/O operations is therefore critical to meeting the requirements for an HPC system and solving I/O performance issues.

Keeping a balance of computing and I/O performance is a significant challenge for future cost-effective HPC systems [45]. In recent years, significant progress has been achieved in parallel applications' computation and communication performances, whereas similar progress in I/O performance has not been achieved. Since I/O is slow compared to the CPU and communication performance, data-intensive scientific applications running on HPC systems are correspondingly bottlenecked by the I/O speed. Nevertheless, good I/O performance in parallel applications can be achieved by combining various conditions such as a sufficient amount of high-speed I/O hardware,

appropriate file-system software and an API for I/O, a high-performance implementation of the API, and using that API correctly [38].

Parallel I/O is essential for scientific applications running on HPC systems. To achieve high performance, many current HPC systems use a multi-layer parallel I/O stack, which consists of a high-level I/O library, I/O middleware, low-level I/O library, parallel file system, and storage hardware (Figure 1.1) [44]. The multi-layer parallel I/O stack offers a number of optimizations that can help in improving the performance of I/O within each layer. For example, the parallel file system typically stripes files across the disks by dividing the file into many smaller units called *striping units* and assigning the striping units to disks in a *round-robin manner* [38]. Thus, file striping enables multiple compute nodes to simultaneously access distinct portions of a file and provides higher bandwidth.

Parallel I/O is defined as concurrent requests from multiple processes of a parallel program for data stored in files [38]. At least two scenarios are possible:

- Each process accesses a separate file; no file is shared among processes: Although it can be thought of as parallel I/O because it represents I/O performed by a parallel program, it is just sequential I/O performed independently by a number of processes.
- All processes access a single, shared file: It can be considered as parallel I/O. That is to say; the I/O is parallel from the application's perspective.

2.4.2. MPI-IO

2.4.2.1. Background

MPI is a *message-passing library interface* specification. It primarily addresses a message-passing parallel programming model, in which data is moved from one process's address space to another process's address space via cooperative operations on each process [5]. The complexity of inter-process communication is hidden from application users and developers.

In April 1992, the basic features essential to a standard message-passing interface were discussed in a workshop of *Standards for Message-Passing in a Distributed Memory Environment* [5]. May 1994 witnessed the first version of the Message-Passing Interface standard released as MPI-1, followed by version 2.0, released in July 1997 as MPI-2, which introduced and standardized parallel I/O as MPI-IO [5]. MPI Forum has listed several MPI implementations such as MPICH[1], Open MPI[2], Cray MPI[3], Intel MPI[4], IBM Spectrum MPI[5] and so on.

MPI-IO aims for portability and optimization for parallel I/O, which cannot be achieved with the POSIX interface [5]. The MPI applications can be migrated easily among different HPC platforms without modifying the source codes. In addition to point-to-point communication, MPI defines collective communication, which results in significant I/O performance improvements for parallel I/O tasks with the proper algorithms. MPI-IO provides an interface for controlling file layout on the file system, partitioning file data among processes logically, issuing collective and asynchronous/non-blocking data access, and applying parallel I/O algorithms [8].

MPI supports and defines the blocking and non-blocking I/O routines [5]:

- *A blocking I/O* call blocks all processes until finishing reading/writing data; it will not return until it is completed.
- *A non-blocking I/O* call does not block other processes from running; starts reading/writing data but does not wait for it to finish.

Application developers can choose either blocking or non-blocking I/O. Due to synchronization and data consistency requirements, using non-blocking I/O is more complex in application design and implementation. The non-blocking I/O routines are named as *MPI_FILE_IXXX*, where the *I* stands for immediate [5].

MPI-IO supports three kinds of basic data-access functions based on [5]:

[1] http://www.mpich.org/
[2] http://www.open-mpi.org/
[3] http://www.cray.com/
[4] http://software.intel.com/en-us/intel-mpi-library
[5] http://www-03.ibm.com/systems/spectrum-computing/products/mpi/index.html

- *an explicit offset* which takes as an argument the offset in the file from which the read/write should begin.
- *an individual file pointer* that reads/writes data from the current location of a file pointer that is local to each process.
- *a shared file pointer* which reads/writes data from the location specified by a shared file pointer shared by a set of processes that together opened the file.

Users can specify a non-contiguous data layout in memory and file in all of these functions. These functions are available in both blocking and non-blocking modes. In addition, MPI-IO has collective versions of these functions that all processes that opened the file together must be called. The collective functions enable an implementation to perform collective I/O [5].

MPI-IO provides a mechanism, called *info*, that allows users to pass hints to the MPI-IO implementation or underlying parallel file system in a portable and extensible manner [38]. These hints have parameters for file striping, caching, access pattern information, etc., to improve I/O performance.

2.4.2.2. Non-contiguous Accesses in MPI-IO

Many parallel applications require each process to access a large number of relatively small pieces of data that are non-contiguously located throughout the file, such as distributed arrays [46], [47]. Thus, one of the main reasons for poor I/O performance is that I/O systems are optimized for large accesses, whereas many parallel applications make lots of small requests [47].

The amount of data that a function sends or receives is specified in terms of instances of a *datatype* [38]. In MPI, there are two kinds of datatypes:

- *Basic datatypes* corresponds to the basic data types such as integers, floating-point numbers, and so on.
- *Derived datatypes* consists of multiple basic data types that are either contiguously or non-contiguously.

MPI offers datatype-constructor functions to create derived datatypes consisting of multiple basic datatypes located either contiguously or non-contiguously [47].

The different types of datatype constructors in MPI are as follows:

- **contiguous**: The simplest derived type is the contiguous type, created with `MPI_Type_contiguous`. It creates a new datatype consisting of contiguous copies of a datatype.
- **vector/hvector**: The simplest non-contiguous datatype is the vector type, created with `MPI_Type_vector`. It creates a new datatype consisting of equally spaced copies of the existing datatype.
- **indexed/hindexed/indexedblock**: The indexed datatype, created with `MPI_Type_indexed`. It allows replicating a datatype into a sequence of blocks containing multiple copies of an existing datatype; the blocks may be unequally spaced.
- **struct**: It is created with `MPI_Type_create_struct`. The most general datatype constructor allows each block to consist of replications of different datatypes [48].
- **subarray**: It is created with `MPI_Type_create_subarray`. It creates a data type corresponding to a multidimensional array's subarray.
- **darray**: It is created with `MPI_Type_create_darray`. It creates a data type that describes a process's local array obtained from a regular distribution of a multidimensional global array.

As an example, *contiguous*, *vector*, and *indexed* are layouts that can quite easily be expressed in terms of existing MPI datatypes are shown in Figure 2.5.

MPI-IO uses MPI data types to define the data layout in the user's buffer in memory and also to define the data layout in the file. The data layout in memory is called *datatype* argument; while the data layout in the file is called *file view* in each I/O operation in MPI-IO [47].

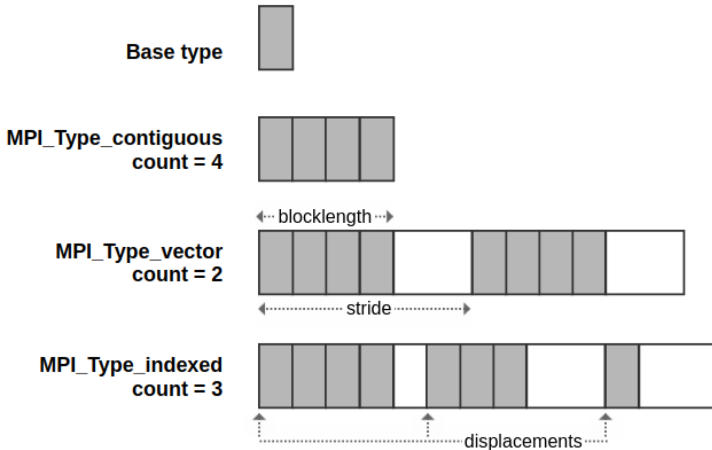


Figure 2.5.: Type maps of contiguous, vector and indexed datatypes for given base type.

When the file gets opened first, the whole file is the default file view of the process. Thus, the process can see the entire file and operate the I/O function contiguously by starting from the position defined by the I/O function. The file view of the process can be changed when needed by calling the function `MPI_File_set_view` with an MPI datatype parameter called the *file type*. That means the process can operate I/O operation on only those visible and accessible parts of the file defined by the file type; any holes are skipped [48] as shown in Figure 2.6.

A file view and a data layout in memory can be constructed using any MPI basic or derived datatype; hence, any non-contiguous access pattern can be represented [48].

Figure 2.7 indicates writing contiguous data into a contiguous block defined by a file view. A different file view is defined for each process using `MPI_File_set_view` so that together, the processes lay out a series of blocks in the file, one block per process (Listing 2.1).

Figure 2.8 indicates writing contiguous data into two separate blocks

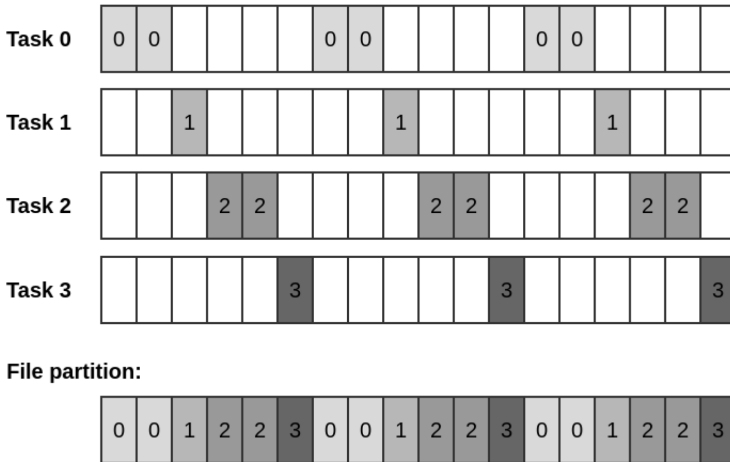


Figure 2.6.: Each process has its own view of the file, defined by a *displacement*, an *elementary type*, and a *file type*.

defined by a file view. Each block is a contiguous type in memory, but the pair of blocks is a vector type in the file view. Displacements are used again to lay out a series of blocks in the file so that two blocks are written per process in a repeating fashion (Listing 2.2).

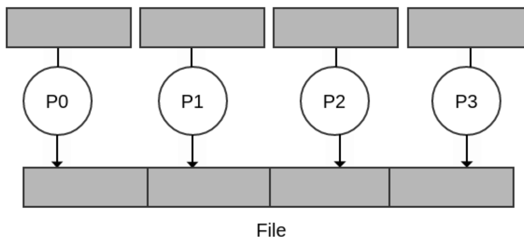


Figure 2.7.: Writing contiguous data into a contiguous block defined by a file view.

```
1    #define N 100
     MPI_Datatype arraytype;
3    MPI_Offset disp;
```

```
5      disp = rank*sizeof(int)*N; etype = MPI_INT;
       MPI_Type_contiguous(N, MPI_INT, &arraytype);
7      MPI_Type_commit(&arraytype);

9      MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
                     MPI_MODE_CREATE | MPI_MODE_RDWR,
11                   MPI_INFO_NULL, &fh);
       MPI_File_set_view(fh, disp, etype, arraytype, "native",
           MPI_INFO_NULL);
13     MPI_File_write(fh, buf, N, etype, MPI_STATUS_IGNORE);
```

Listing 2.1: Code segment for writing contiguous data into a contiguous block defined by a file view.

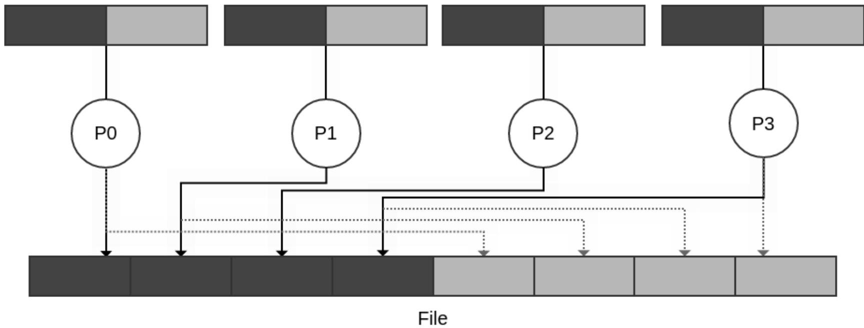


Figure 2.8.: Writing contiguous data into two separate blocks defined by a file view.

```
       int buf[NW*2];
2      MPI_File_open(MPI_COMM_WORLD, "/data",
                    MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
4      /* want to see 2 blocks of NW ints, NW*npes apart */
       MPI_Type_vector(2, NW, NW*npes, MPI_INT, &fileblk);
6      MPI_Type_commit(&fileblk);
       disp = (MPI_Offset)rank*NW*sizeof(int);
8      MPI_File_set_view(fh, disp, MPI_INT, fileblk,
                        "native", MPI_INFO_NULL);
10     /* processor writes 2 'ablk', each with NW ints */
       MPI_Type_contiguous(NW, MPI_INT, &ablk);
```

```
12      MPI_Type_commit(&ablk);
        MPI_File_write(fh, (void *)buf, 2, ablk, &status);
```

Listing 2.2: Code segment for writing contiguous data into two separate blocks defined by a file view.

Parallel I/O applications have constantly been challenged to efficiently store and retrieve the increasing amount of contiguous or non-contiguous data [49]. Therefore, it is essential that an access pattern can be expressed by the I/O interface, as it allows an implementation to improve the performance of I/O requests. A critical optimization in parallel I/O is to take advantage of collective operations instead of individual operations (Figure 2.9). Collective I/O operations typically allow users to specify multiple non-contiguous accesses with a single I/O function call by building large, contiguous blocks [39]; so that reads/writes will be more efficient.

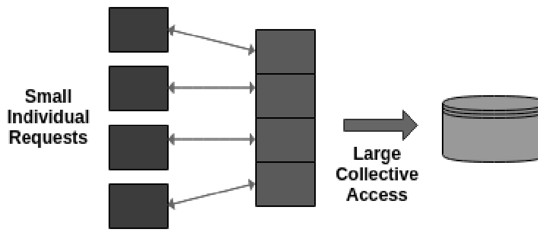


Figure 2.9.: Collective I/O operations.

2.4.2.3. MPI-IO Libraries

In parallel I/O, multiple processes of parallel program access any part of one file or multiple files, as contiguous or non-contiguous. As a result, the I/O access patterns of a parallel program often contain a large number of small and non-contiguous data accesses. This means that the application needs to make many small I/O requests, which degrades I/O performance drastically.

Parallel I/O systems or distributed memory architectures need a mechanism to define collective operations and non-contiguous data layouts in memory and the file. MPI-IO allows users to access non-contiguous data by

using MPI derived datatypes in a single collective I/O function call. This feature provides MPI-IO implementations with an opportunity to optimize data access [40], [50].

Two widely used MPI implementations *MPICH*[1] and *Open MPI*[2] have implemented different MPI-IO libraries, which are designed with different architectures. *ROMIO*, a portable MPI-IO library, has been integrated into most of the MPI implementations (Figure 2.10), while *OMPIO* [51] is a specialized MPI-IO library only for Open MPI and its derivatives (Figure 2.11). ROMIO and OMPIO are two coexisting but independent parallel I/O-libraries in Open MPI.

OMPIO provides two main advantages compared to ROMIO [51]:

- more fine-grained separation of functionality under the favour of different frameworks
- no modification needed on the end-user application with non-file system specific module selection

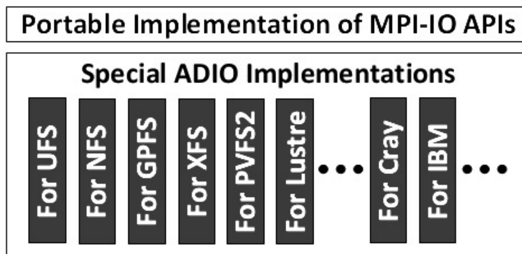


Figure 2.10.: The abstracted ROMIO architecture.

OMPIO has a highly modular approach to parallel I/O by dividing I/O functionality into smaller frameworks (*fs*, *fcoll*, *fbtl*, and *sharedfp*) and modules in each framework [51].

As parallel I/O algorithms, *data sieving* and *two-phase I/O* are integrated to achieve higher performance for non-contiguous I/O requests and small

[1]https://www.mpich.org/
[2]https://www.open-mpi.org/

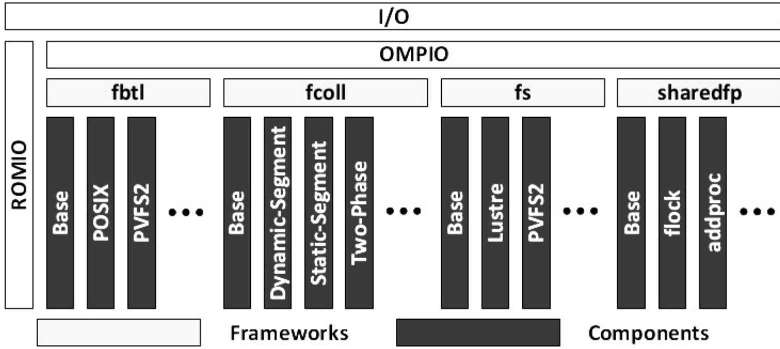


Figure 2.11.: The abstracted architecture of OMPIO frameworks and modules.

data accesses in ROMIO, while the *fcoll* framework in OMPIO has different algorithms: *two-phase I/O*, *static segmentation*, *dynamic segmentation*, *the individual algorithm* to use depending on the functionality of the framework and external parameters [51].

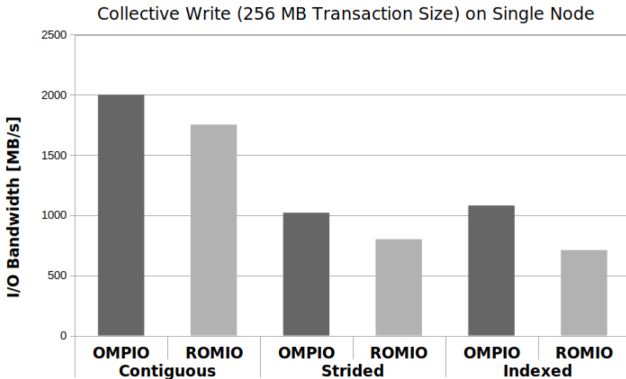


Figure 2.12.: File write performances for contiguous, strided and indexed data layouts on OMPIO and ROMIO.

To better understand, I want to compare two MPI-IO libraries, ROMIO

and OMPIO, based on the access pattern of an application and the underlying file system [52]. Particular focus is on collective I/O and its potential performance improvements over individual I/O operations. All evaluations were made on Hazel Hen (Cray XC40) with an InfiniBand connected Lustre file system at HLRS.

For the first experiment, a self-implemented I/O benchmark is performed to simulate collective write operations with contiguous and non-contiguous data layouts on Hazel Hen (Figure 2.12). Widely used contiguous and non-contiguous data layouts in user applications have been searched, and consequently, contiguous, strided and indexed data layouts are defined by derived MPI data types and file views for collective write operations in the benchmark. Type maps of contiguous, vector, and indexed data types for a given base type are shown in Figure 2.5. These data types are constructed in the benchmark accordingly by using `MPI_Type_contiguous` function for the contiguous layout, `MPI_Type_vector` function for the strided layout, and `MPI_Type_indexed` function for the indexed layout. The performance results of different data layouts and MPI-IO libraries, ROMIO and OMPIO, are investigated as seen in Figure 2.12. Single node experiments were performed on Hazel Hen using OMPIO default settings and ROMIO default settings. Each test case is executed multiple times for all measurements, taking the average achievable bandwidth across those runs.

Figure 2.12 shows the benefits of using the OMPIO library in all layouts for the given benchmark compared to the ROMIO library. When we look at data types, we see that choosing a contiguous data layout gives higher performance than the other data layouts in all experiments. Different transaction sizes are worked on in the first experiment. It is seen that small transaction sizes result in very poor I/O performance. If the transaction size is too small, the increased parallelism does not make up for the many small writes because overall latency increases simultaneously. In Figure 2.12, the use case is shown when the data transfer size is assigned to 256 MB. The contiguous data type seems x2 times faster than the vector and indexed data types with OMPIO default settings. It is understood that changing the I/O implementation of an application to favour large transaction sizes needs a

detailed investigation of the different data layouts. Choosing an appropriate MPI-IO library has a significant impact on I/O performance.

For the second experiment, collective and individual I/O read performance are compared for a different number of nodes on Hazel Hen. For both individual and collective operations, default settings are used for OMPIO and ROMIO. The MPI-Tile-I/O benchmark is used to implement tile accesses on two-dimensional dense data sets using 256 x 256 x 16 bytes per process (left), 256 x 256 x 4096 bytes per process (right) is presented in Figure 2.13.

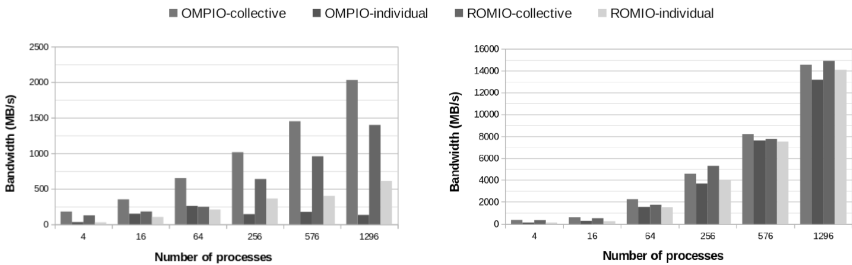


Figure 2.13.: Collective and individual file read performances with the MPI-Tile-I/O benchmark.

Figure 2.13 shows the performance improvements of collective I/O operations over individual I/O operations in both ROMIO and OMPIO for two different data transaction sizes. The results encourage using collective I/O operations in I/O intensive user applications. The comparison here shows the benefits of using the OMPIO library, especially its collective I/O framework, instead of individual I/O. Each test case is executed multiple times for all measurements, taking the average achievable bandwidth across those runs. Note that all evaluations were performed with default configurations of the Lustre file system on Hazel Hen; 4 OSTs and 1 MB striping unit. All data are written as stripe-aligned.

The results show that one cannot reliably choose a single data layout or approach and expect uniform performance portability among these two implementations. As it is understood, changing the I/O implementation of a

user application to favour larger transactions needs a detailed investigation of the data layouts and MPI-IO library.

Understanding MPI-IO libraries allows end-users to select the best I/O approach depending on the application's data layout and helps to increase their efficiency accordingly. The comparison here shows the benefits of using the OMPIO library, but the ROMIO library is widely used in today's engineering applications. Therefore, this study is currently based on the MPI-IO ROMIO library. It is designed for Lustre parallel file system; it also offers optimization possibilities for the other file systems such as IBM Spectrum Scale and BeeGFS.

2.4.3. Parallel I/O Optimizations

2.4.3.1. Data Sieving

It is critical to make a few requests to the file system as possible to reduce the impact of high I/O latency. *Data sieving* is a parallel I/O algorithm that was introduced in ROMIO to access non-contiguous I/O requests in large chunks [50]. It minimizes the network load, although at the cost of reading more data than needed. Instead of making independent requests for non-contiguous data, ROMIO uses data sieving to avoid separately accessing each contiguous data. The basic idea is to allocate a piece of local memory for caching the entire file or a relatively large portion of the file (the data transfer size of each non-contiguous I/O request) [8].

Assume that the user has made a single read request for five small non-contiguous data chunks. Without data sieving, each chunk is read separately, so the connection with the file system is established five times. Figure 2.14 shows an example of the data sieving. First, the data is read into the temp buffer in the local memory as a large and contiguous chunk. Then the requested portions are extracted from the temporary buffer and placed in the user's buffer [50]. The subsequent I/O operations are performed within the local memory, which requires almost no seek time. Accessing local memory for reading/writing operations is far faster than accessing a file system.

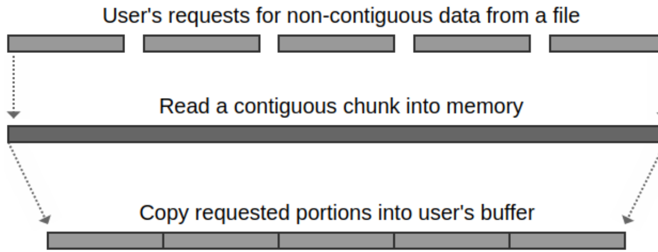


Figure 2.14.: Data sieving.

A potential problem with this algorithm is its memory requirement [50]. Reading a large chunk of data into memory and the locking mechanism to block other processes are the drawbacks of data sieving [8].

2.4.3.2. Collective I/O

Although each process may require access to several non-contiguous portions of a file in many parallel applications, the requests of different processes are frequently interleaved. As a result, they may collectively span large contiguous portions of the file [50]. If the MPI-IO implementation provides the complete non-contiguous access information of all processes, I/O performance can be significantly improved by merging the requests of different processes. Such optimization is referred to as *collective I/O*.

Collective I/O can be performed at the disk, server, or client levels. ROMIO performs collective I/O at the client level using a generalized version of two-phase I/O. It has been shown to improve performance significantly [47].

Two-phase I/O is a collective I/O algorithm at the client level for accessing distributed arrays from files. The basic idea of two-phase I/O is to avoid making lots of small I/O requests by dividing the entire I/O process into two phases: an I/O phase and a communication phase.

Figure 2.15 shows an example of the two-phase I/O. Assume that the user has requested reading a distributed array from a file using two-phase I/O. In the first phase, all processes access data according to a distribution

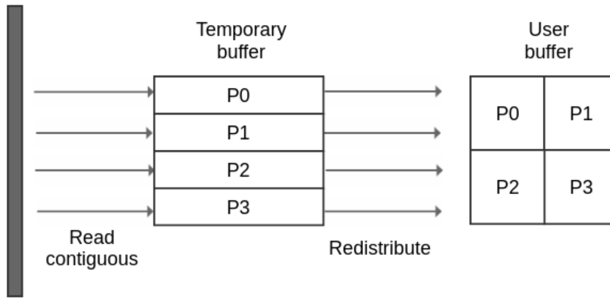


Figure 2.15.: Reading a distributed array by using two-phase I/O.

that results in each process making a single, large, contiguous access. In the second phase, data is redistributed among processes to achieve the desired distribution [47]. This method can significantly reduce I/O time by making all file accesses large and contiguous even though the additional cost of interprocess communication for redistribution [38].

The two-phase I/O abstraction may introduce an overhead due to the MPI communication required to organize the collection of segments. Additionally, modern distributed file systems often already provide similar internal optimizations, which may conflict with the collective buffering implementation [53].

2.4.3.3. MPI Hints

MPI info is an object that stores a set of (*key*, *value*) pairs, and they are passed to MPI functions [5]. MPI standard defines various MPI infos, such as communicator info, window info, and file info (also called *MPI file hints*), for users to provide information for direct optimization [8].

The following MPI functions can interpret its file info:

- `MPI_FILE_OPEN`
- `MPI_FILE_DELETE`
- `MPI_FILE_SET_VIEW`

- MPI_FILE_SET_INFO

Bypassing the MPI file info, file access information can be given from user applications to MPI-IO libraries or even the underlying distributed parallel file systems so that the parallel I/O performance can be improved [8]. Except MPI reserved file hints, different MPI implementations and MPI-IO libraries can define their own file hints/info [8]. For example, striping_factor and striping_unit reserved file hints are defined for the underlying Lustre file system striping mechanism, romio_cb_read and romio_cb_write file hints are defined by ROMIO to enable the collective I/O for MPI collective reading/writing functions.

Chapter 3

General I/O Auto-tuning Framework

3.1. Design Requirements

In Section 1.2.3, heuristic search-based and performance modelling-based I/O auto-tuning approaches to accelerate the I/O requests of engineering applications were introduced. The proposed I/O auto-tuning framework owns the following four abilities:

- **Compatibility**: This framework should be compatible with as many engineering applications as possible. Designing based on the MPI-IO library ensures the software compatibility with not only MPI but also parallel HDF5 or parallel NetCDF applications.
- **Scalability**: Time and resource consumption should be acceptable. The engineering applications usually scale out to hundreds of thousands of compute nodes to solve large-scale engineering problems.
- **Usability**: To encourage more scientists and engineers to use this

system, it should not require additional skills or knowledge and be easy to use. Scientists remain focused on their simulations and take little consideration about the I/O performance.

- **Portability**: The framework must be designed for not only one HPC platform. Sometimes there is more than one HPC system in an HPC centre, which is updated or upgraded regularly; therefore, it should also be able to run on multiple platforms.

To fulfil these abilities, the auto-tuning framework needs to follow the current MPI standard, run transparently to the users, produce acceptable little overhead and improve I/O performance automatically.

3.2. I/O Performance Factors

Many performance factors of the I/O stack are involved in the I/O efficiency. By researching the application characteristics, Lustre file system, and the MPI-IO ROMIO library, it can be seen that the following parameters affect I/O performance significantly:

- **number of cores**: The number of I/O processes has a significant impact on the I/O performance. Different numbers of processes participate to the I/O operations, such as one process, a subset of processes, or all processes. This performance factor can not be changed at runtime so that it can be used to identify log and configuration files.
- **problem size**: Proper configurations change depending on how many bytes of data are read/written by each MPI-IO process. However, it would be too expensive to find out a good configuration for every data transfer size. As suggested in [8], different data transfer sizes which may have similar good configurations can be grouped into file indexes. For example, in the prototype, 56 data transfer size groups are defined from 100 B to 2,000,000,00 B, and any data transfer size larger than 195.31 MB belongs to the 56th file size group.

- **MPI-IO subroutine**: Good configurations depend on the type of MPI-IO subroutines (collective vs individual access pattern). Saving the type of the MPI-IO subroutine used in applications can help users analyse the I/O bottlenecks of their applications. This performance factor can not be changed at runtime.
- **MPI info**: The MPI info has runtime I/O tuning options for MPI-IO libraries such as:
  - **romio_cb_read**: The collective buffering optimization (two-phase I/O) for reading operations can be enabled or disabled to accelerate applications' I/O requests.
  - **romio_cb_write**: The collective buffering optimization (two-phase I/O) for writing operations can be enabled or disabled to accelerate applications' I/O requests.
  - **striping_factor**: It specifies the number of Lustre OSTs to stripe new files (`stripe_count`).
  - **striping_unit**: It specifies the size (in bytes) of each Lustre file system OST stripe unit (`stripe_size`) used for new files.

The first challenge was determining how to choose the values of previously mentioned MPI info objects to test, particularly the Lustre striping parameters. I used Interleaved or Random (IOR)[1] benchmark to compare an efficient optimization to an inefficient one. It simulates the applications' I/O write requests in different problem sizes with 1,200 processes on Hazel Hen [2] with Lustre file system at HLRS. The IOR ran with various random configurations considering the characteristics of parallel I/O stack layers. The results of each configuration obtained from 10 running samples are presented in Figure 3.1 and Figure 3.2 through four box plot diagrams[3].

[1] http://github.com/LLNL/ior

[2] https://www.hlrs.de/systems/cray-xc40-hazel-hen/

[3] Box plots use graphic to illustrate groups of numerical data through their quartiles in descriptive statistics. They provide the maximum, median, and minimum results, as well as upper and lower quartiles.

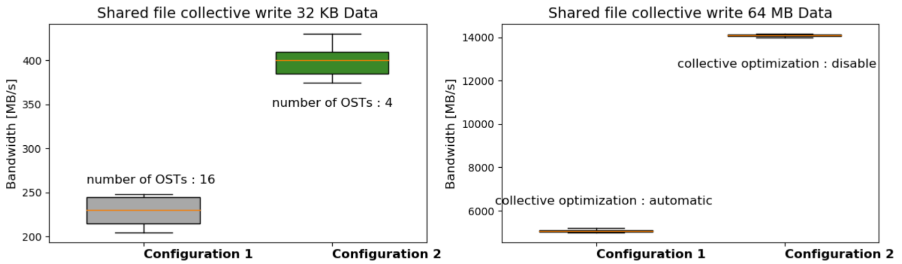


Figure 3.1.: I/O simulation results by applying different MPI hints.

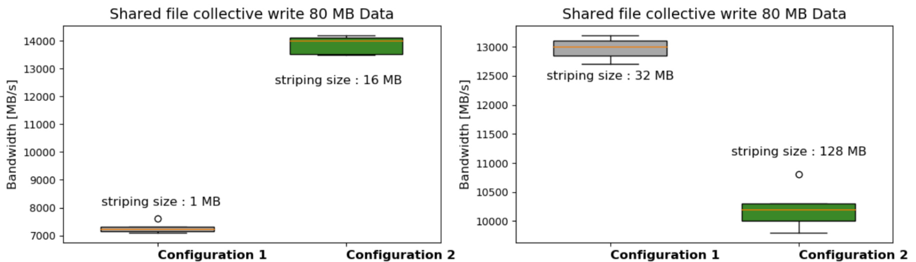


Figure 3.2.: I/O simulation results by applying different Lustre stripe size.

The I/O simulations ran with different configuration parameters such as data transfer size, the number of processes, Lustre striping values, and collective I/O optimizations in ROMIO. In Figure 3.1, the I/O simulations were configured with MPI collective I/O write operations that access a single shared file, using the same data transfer size on each process for each case and by using MPI hints to control the Lustre striping setups. In the left case of Figure 3.1, for a small data transfer size (32 KB) on Lustre, the writing performance of striping over 4 OSTs with 1 MB `stripe size` is about 71 % better than the performance of striping over 16 OSTs with 1 MB `stripe size`. Disabling the collective buffering optimization in ROMIO, which is usually not recommended, achieved about 269 % improvement in write performance in the right case of Figure 3.1 for a non-small data transfer size of 64 MB. Figure 3.2 shows that there is an optimal range for Lustre `stripe size` depending on data transfer size because I/O performance

does not keep rising together with `stripe size`. I want to highlight that, while in principle, there are many suggestions to improve IO performance, in practice, this is very challenging due to the complex interactions between different parameters. The problems shown in Figure 3.1, and Figure 3.2 are just the tip of the iceberg. The challenge is that those configuration parameters need to be carefully evaluated and tuned by an expert.

For the training process, I selected a few meaningful parallel I/O parameters for all the layers of the I/O stack based on previous research efforts and experience. Increasing the number of OSTs for large jobs improves the I/O performance accordingly, but the resources (Lustre OSTs and network connections) are limited. To avoid potential conflicts between concurrently running I/O applications, the maximum `striping_factor` was set as 16. Since large values of `striping_unit` may result in longer lock hold times, the values of `striping_unit` were selected from 1 MB to 16 MB with powers-of-two. As for the collective buffering, values are defined as automatic, disable and enable.

The search scope of the configuration parameters worked on in this study is given in Table 3.1. A user can set the parameter space by simply modifying the parameter list. The auto-tuning framework can help users and developers to explore how these parameters interact with each other.

Table 3.1.: Configurations' search scope

Name	**Value**
number of cores	24 - 2400
data transfer size	100 B - 195 MB
striping factor	1 - 16
striping unit	1 MB - 16 MB
collective I/O	automatic; disable; enable
access pattern	collective; individual

3.3. MPI and PMPI Wrapper

The two most widely used MPI implementations, MPICH and Open MPI are implemented in *C*. Both of them redirect the MPI subroutines in Fortran to the *profiling MPI (PMPI) interface* in C [8]. Every standard MPI function can be called with an `MPI_` or `PMPI_` prefix. The MPI standard lets one to write functions with the `MPI_` prefix that call the equivalent `PMPI_` function. Specifically, a function written in this manner has the standard function's behaviour plus any additional desired behaviour. This is useful for MPI performance analysis because it captures program MPI calls as well as important performance data.

The MPI wrapper is compiled as the shared library for C applications. It is implemented with the dynamic symbol in POSIX specification and can be dynamically loaded by setting the system environment variable `LD_PRELOAD` (Listing A.1).

MPI wrapper for `MPI_Init()` subroutine is depicted in Figure 3.3. The address of a symbol pointing to the profiling MPI subroutine `PMPI_Init()` is obtained by dynamic link function. The same input parameters used in user applications are passed to `PMPI_Init()` directly. By successfully executing the `PMPI_Init()` subroutine, modules of the auto-tuning framework starts according to the user running modes. It is analogous to implementing the MPI wrapper for other MPI or PMPI subroutines.

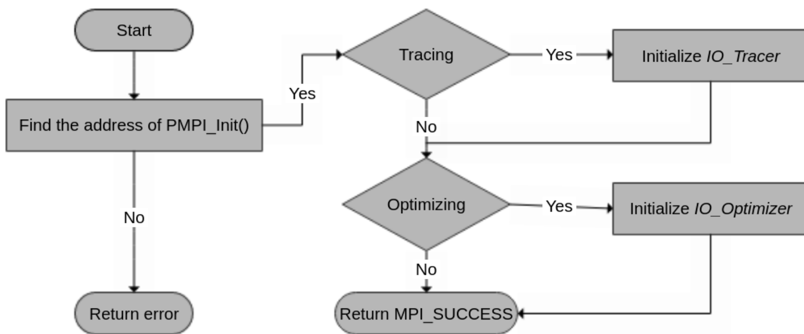


Figure 3.3.: MPI wrapper for MPI_Init() flow chart.

3.4. Running Modes

Depending on user needs, it is sometimes desirable to monitor the user application, sometimes optimize it, and sometimes both. In the auto-tuning framework, three running modes were designed. This way, more flexibility is given to end-users, and unnecessary overhead is also eliminated.

- **Optimizing**: This mode sets good configurations for all file operations transparently. Because the tracing component is not activated in this mode, no memory is allocated for tracing I/O operations.
- **Tracing**: This mode records the I/O-related information and gives users as well as system administrators a clear view to profile the applications' I/O behaviours. The log files generated in this mode are the sources of the learning module to search for good configurations.
- **Optimizing and tracing**: This mode initiates the tracing and optimizing modules together. It sets the good configurations and records their performance results into log files.

Chapter 4

Heuristic Search Based I/O Auto-tuning

This section describes the background of the heuristic search-based I/O auto-tuning approach developed in this thesis. I show how the approach determines and auto-tunes I/O parameters using the *genetic algorithm*.

4.1. Heuristic Search

A naïve strategy-based auto-tuning requires running an application with all possible combinations of I/O parameters to determine the best-performing parameter set. This approach is an exhaustive search through the huge parameter space due to the long execution times of trial runs. It consumes time and resources even for unsuccessful parameter sets for the given application and system. Adaptive heuristic search-based approaches can solve this problem by searching the parameter space with a few tests.

Genetic evolution algorithms, simulated annealing, and other adaptive heuristic search approaches can traverse the search space with fewer trials

[21]. I explore *genetic algorithm* for sampling the parameter space by testing a set of parameter combinations and then, based on the I/O performance, adjusting the combination of tunable parameters for further testing. As a result, better parameter combinations emerge through multiple generations (i.e., sets of tunable parameters with high I/O performance).

A genetic algorithm is a metaheuristic that reflects the process of natural evolution by modifying a population of individual solutions as a randomized search algorithm [54]. It randomly selects individuals of the initial population from the current population as parents. Then, it uses these individuals to generate the children for the next generation. The bad individuals are eliminated through iterative generations, and the good individuals are saved. Good parents produce good children.

A set of operators such as reproduction, crossover, and mutation is used to set the initial population to generate successive populations with time (Figure 4.1) [55]. Reproduction is a procedure based on the objective function (fitness function) of each individual to determine how "good" the individual is [55]. Thus, individuals with higher fitness values can contribute to the next generation. Crossover is a genetic operator in which members of the last population are mated at random in the mating pool [55]. The mutation is the random change portions of the individual with a small probability [55]. Random mutations provide a sampling of the remainder of the space [21]. A genetic algorithm is expected to converge to an optimal or near-optimal solution in fewer iterations [21].

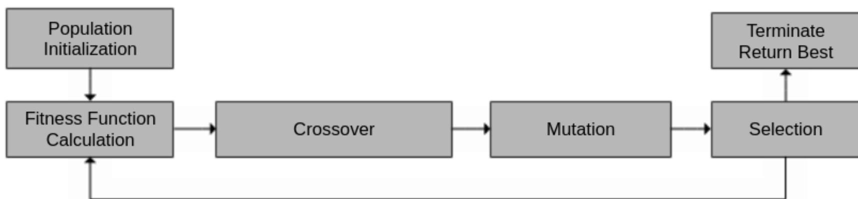


Figure 4.1.: Procedure of genetic algorithm.

4.2. Architecture

Figure 4.2 shows the overall architecture of heuristic search-based I/O auto-tuning approach that has two modules: *IO_Optimizer* and *IO_Tuner*. The *IO_Tuner* automatically tunes a good I/O parameter set suggested by a genetic algorithm engine; *IO_Search* that is located in the *IO_Optimizer* module. The *IO_Search* executes the I/O application with a preselected random initial set of tunable parameters. I/O parameter search space is also given as input. It searches the parameter space to find out the best-performing parameter set iteratively. When a successful parameter set is found, the *IO_Tuner* takes the set and dynamically links to MPI-IO calls of the I/O application. Then, I/O performance results can be used to refit the *IO_Search* with the dynamic conditions of a parallel I/O system adaptively for scientists and engineers to find out the latest good configurations.

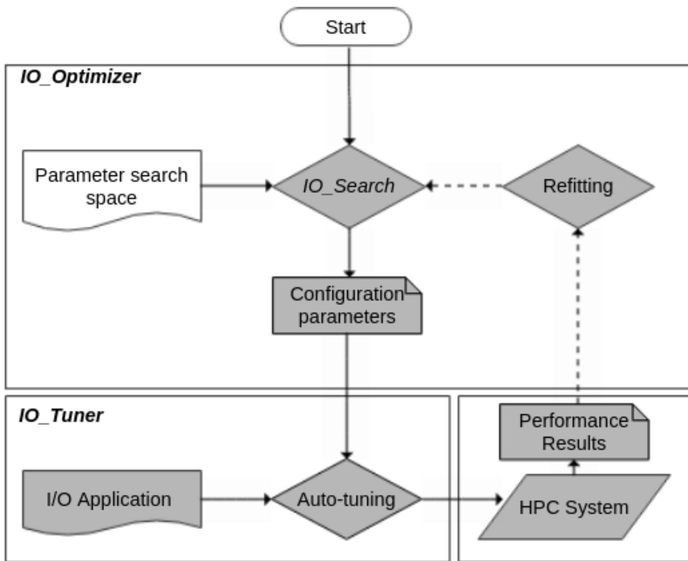


Figure 4.2.: Overall architecture of the heuristic search-based auto-tuning approach.

4.2.1. IO_Optimizer: Configuration Search

In genetic evolution algorithms, selecting fewer combinations of parameters and running a few tests is reasonable to search for huge and complex parameter space. The *IO_Search* is a genetic evolution algorithm engine included in the *IO_Optimizer* module to traverse the search space.

The *IO_Search* randomly selects individuals of the initial parameter set. Then, it modifies the values of parameters for further testing based on the I/O performance. Thus, over consecutive generations, the population can approach a parameter set that gives better I/O performance than the performance of the default settings.

Figure 4.3 shows the workflow of the genetic algorithm engine *IO_Search*. The *IO_Search* starts the genetic algorithm for a given concurrency and problem size with the I/O application. Predefined parameter space is given to the *IO_Search* as an input that includes all possible values of tunable parameters. At first, the *IO_Search* generates an initial population of I/O parameters and creates a configuration file containing the selected parameters to be used by the *IO_Tuner* in auto-tuning the I/O application.

In the experiments, population size is selected 10 by the *IO_Search*, it can be configured. The fitness value of an individual is defined as the I/O bandwidth. As the *IO_Search* passes through a new generation, it calculates the fitness of individuals, namely I/O bandwidth. The best individuals named as the *elite members* who give high I/O performance are transferred to the next generation. The rest of the population in the next generation is generated by applying crossovers and mutations to the current population. These steps are repeated for each generation until stop criteria are reached. The *IO_Search* defines the number of generations as 30 so that the maximum of 300 runs of the given I/O application can be executed by the *IO_Search*.

The mutation rate is defined as 15% in the *IO_Search*, which means mutation is applied to 15% of the current population for each generation. Finally, the I/O parameters found for the given application and scale are stored in the configuration file.

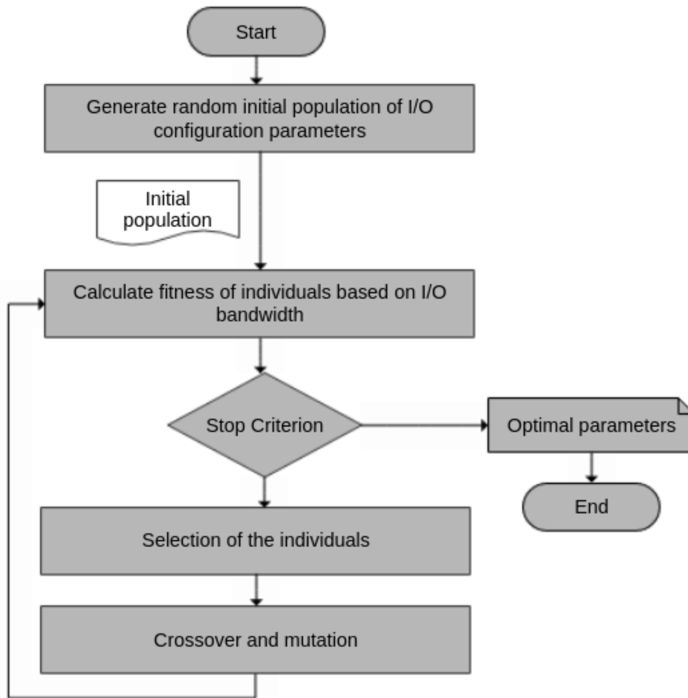


Figure 4.3.: Overall architecture of the *IO_Search*.

4.2.2. IO_Tuner: Setting I/O Parameters at Runtime

The *IO_Tuner* component is a parallel I/O tuning module that takes the I/O parameters found by the *IO_Search* included in the *IO_Optimizer* and then dynamically sets these parameters at different layers of the I/O stack. Currently, the *IO_Tuner* works on the MPI-IO ROMIO library and Lustre parallel file system.

At first, the *IO_Tuner* reads the configuration file, including good configurations found by the *IO_Search*. As soon as I/O applications or benchmarks call MPI-IO subroutines, the *IO_Tuner* is triggered to apply these good configurations before executing the I/O operation transparently. Finally, it passes the intercepted MPI-IO functions of the application or benchmark in the PMPI

wrapper. The good parameters are set at this step, and then the original MPI-IO function is called. In this way, auto-tuning can be done transparently to the users without source code modification.

Figure 4.4 shows how the *IO_Optimizer* module accesses the configuration pool, fetches the proper configuration and then passes it to the `MPI_FILE_OPEN` and `MPI_FILE_WRITE` subroutines through *IO_Tuner*.

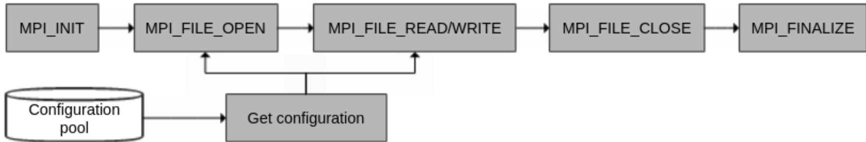


Figure 4.4.: Optimizing process.

The previously found good configurations could be outdated and no longer optimal. On the other hand, sometimes, users achieve better I/O performance with some "brand new" configurations. After executing I/O operations, performance results can be used to refit the *IO_Search* with the dynamic conditions of a parallel I/O system adaptively for scientists and engineers to find out the latest good configurations.

4.3. Implementation

4.3.1. Benchmarks

I chose two I/O benchmarks to evaluate the heuristic search-based I/O auto-tuning approach: *IOR* and *MPI-Tile-IO*. These represent different I/O write patterns with different problem sizes.

- **IOR** [56]: The IOR (LLNL 2015) is an I/O benchmark developed at Lawrence Livermore National Laboratory (LLNL). It is one of the main HPC I/O benchmarks because it is highly configurable and supports various APIs to simulate I/O load.
- **MPI-Tile-IO** [57]: The MPI-Tile-IO benchmark tests the I/O performance in a real-world scenario. The purpose of the MPI-Tile-IO is to

test the performance of an underlying MPI-IO and file system implementation under a non-contiguous access workload. It tests how it is performed when challenged with a dense 2D data layout.

4.3.2. System setup

The experiments were conducted on the NEC Cluster platform (Vulcan) at HLRS. Section 2.2 shows the technical details about the Vulcan. As for the experimental Lustre file system, the default setup of Lustre striping configuration is `striping_factor`=4 and `striping_unit`=1048576. OpenMPI version is from version 4.0.3.

4.3.3. Parameter space

The *IO_Search* can take arbitrary values as input for a parameter space. However, the evolution of the genetic algorithm will require more generations to search a parameter space with arbitrary values.

To shorten the search time, I selected a few meaningful parallel I/O parameters for all the layers of the I/O stack based on previous research efforts and experience. I have chosen most parameter values to be powers-of-two except for some parallel file system parameters. To avoid potential conflicts between concurrently running I/O applications, the maximum `striping_factor` was set as 16. Since large values of striping_unit may result in longer lock hold times, the values of `striping_unit` were selected from 1 MB to 16 MB with powers of two. As for the collective I/O operations, values are defined as automatic, disable and enable. Setting parameter values to reasonable ranges based on knowledge of page sizes, min/max striping ranges, and powers-of-two values can be done by someone with a basic understanding of the system. Furthermore, this task needs to be performed only once per-system basis.

Table 4.1 shows ranges of various parameter values. A user can set the parameter space by simply modifying the parameter list in the *IO_Search*. Adding new parameters to the search needs simple modifications to the

IO_Tuner. The following is a list of parameters that are used as part of the parameter space.

- Lustre:
  - Stripe count (`striping_factor`)
  - Stripe size (`striping_unit`)
- MPI-IO
  - The collective buffering optimization (`romio_cb_write`)

Table 4.1.: A list of the tunable parameters and ranges used for experiments. The last column shows the number of distinct values used for each parameter.

Name	Value	Quantity of Values
striping_factor	1; 2; 4; 8; 16	5
striping_unit	1 MB; 2 MB; 4 MB; 8 MB; 16 MB	5
romio_cb_write	automatic; disable; enable	3

4.3.4. Scale and data set sizes

I designed a weak-scaling configuration to test the performance of the auto-tuning framework at different concurrencies, i.e., 64 and 256 cores for the MPI-Tile-IO; 240 and 1,200 cores for the IOR benchmark. There are 24 CPU cores on each Vulcan compute node. The amount of data each core writes is constant for a given benchmark, i.e., the amount of data the benchmark writes increases proportional to the number of cores used.

4.4. Results

I conducted experiments for two benchmarks, the IOR and the MPI-Tile-IO, in different data transfer sizes on Vulcan. The experiments have been repeated multiple times, both default and tuned configurations, and plotted average

values. The default experiments are measured by applying the system default settings that system administrators define.

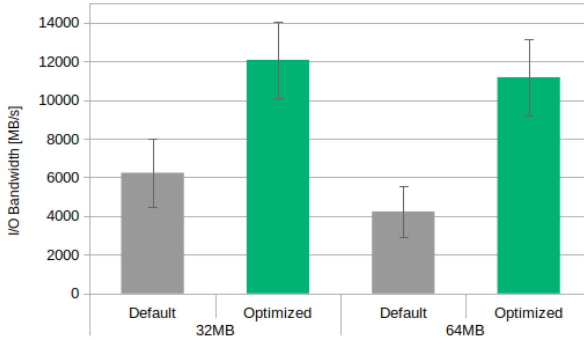
Performance improvements obtained by using the parameters that the auto-tuning system detected for the IOR benchmark are shown in Figure 4.5 on Vulcan at 240 and 1,200 cores concurrencies and for the MPI-Tile-IO benchmark in Figure 4.6 on Vulcan at 64 and 256 cores concurrencies. Y-axis represents I/O bandwidth in MB/s, and the x-axis represents data sizes. The scales of the I/O bandwidth axes are different in the plots. Note that only a subset of the combinations was run due to limited access to the platform. The default experiments correspond to the system default settings that a typical user of the HPC platform would encounter in the absence of an auto-tuning framework.

For all experimental tests, the I/O bandwidth is calculated as the ratio of the amount of data to be written into a file to the time taken to write the data. In the measured I/O time, opening, writing, and closing the file overhead is included. The overhead of MPI-IO call interception by the *IO_Tuner*, which is included in the time taken, was negligibly small, even at a high core count.

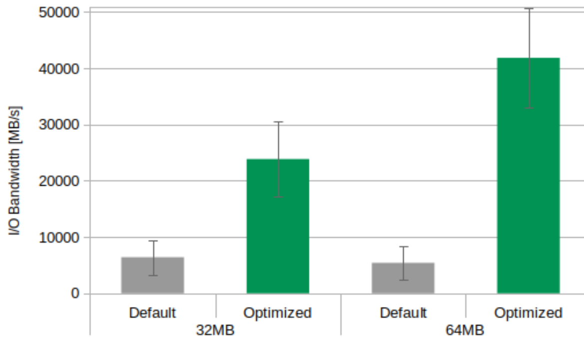
The *IO_Search* ran for $\sim$ 6.5 hours for the IOR, $\sim$ 2.5 hours for the MPI-Tile-IO to search through the parameter space of each experiment. In most cases, the *IO_Search* passed through 10 to 30 generations. It selects the configuration that achieves the best I/O performance through the course of the genetic algorithm evolution. The heuristic search-based auto-tuning approach achieved an increase in I/O bandwidth of up to 7.74$\times$over the default parameters for the IOR benchmark and 5.59$\times$over the default parameters bandwidth for the MPI-Tile-IO benchmark, as shown in Figure 4.5 and Figure 4.6.

Table 4.2 shows the I/O performances of the default, and the optimized experiments for two use cases in Figure 4.5 and Figure 4.6. I also show the speedup that the optimized settings achieved over the default settings for each experiment.

The heuristic search-based auto-tuning approach's runtime is proportional to the amount of time it takes to run the application. The benefit of using a genetic algorithm is selecting the parameter values for the next run. Thus,

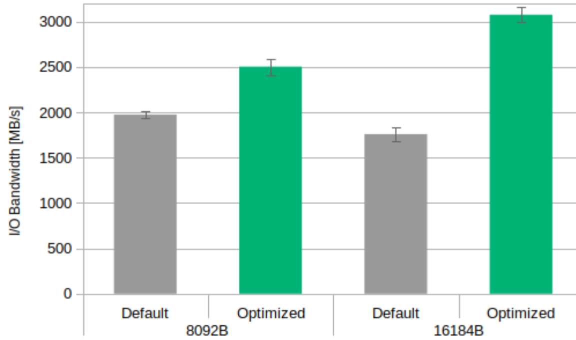


(a) 240 cores

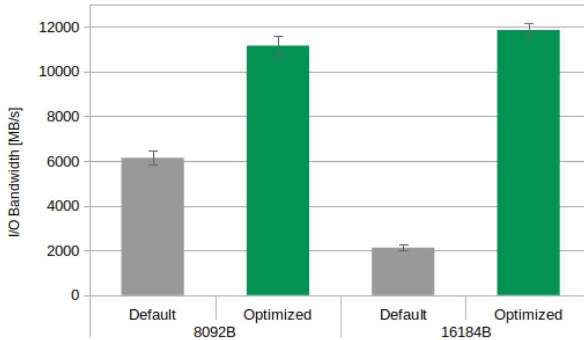


(b) 1200 cores

Figure 4.5.: Default vs. optimized write bandwidth on the IOR for various transfer sizes running on 240 cores and 1,200 cores of Vulcan. Y-axis represents I/O bandwidth in MBps and x-axis represents transfer sizes (in MB). The scales of the I/O bandwidth axes are different in the plots.



(a) 64 cores



(b) 256 cores

Figure 4.6.: Default vs. optimized write bandwidth on the MPI-Tile-IO for various transfer sizes running on 64 cores and 256 cores of Vulcan. Y-axis represents I/O bandwidth in MBps and x-axis represents element sizes in number of tiles (in KB). The scales of the I/O bandwidth axes are different in the plots.

Table 4.2.: I/O speedups of applications with optimized parameters over default parameters.

Application		IOR (MB/s)		MPI-Tile-IO (MB/s)	
#Cores		240	1200	64	256
Use case 1	Default	6238.56	6402.08	1974.462	6138.917
	Tuned	12075.01	23859.15	2503.22	11257.42
	Speedup	1.93	3.73	1.27	1.83
Use case 2	Default	4238.57	5412.91	1759.326	2122.027
	Tuned	11186.23	41859.44	3075.17	11859.23
	Speedup	2.64	7.74	1.75	5.59

it eliminates the need for large data sets for training, which are difficult to obtain due to the long runtime of application codes. However, running a few iterations of an I/O application on a few nodes can still take hours, especially for large amounts of data. Therefore, to reduce the overall time required, I propose a performance prediction model to predict I/O bandwidth and then further reduce the auto-tuning model runtime. The following section describes the performance modelling-based auto-tuning approach, which uses predictions to output the best parameters within a few seconds.

Chapter 5

Performance Modelling Based I/O Auto-tuning

This section describes the background of the performance modelling-based I/O auto-tuning approach that I propose for solving HPC I/O tuning problems. Then, I show how the auto-tuning approach determines and auto-tunes I/O parameters using the *random forest algorithm*.

5.1. Performance Modelling

In auto-tuning, a naïve strategy is to run an application using all possible combinations of tunable parameters to find the best. However, this is an exhaustive search through a huge parameter space and is infeasible because of the long execution times of trial runs on application codes. Furthermore, this becomes a highly time and resource-consuming approach depending on the size of the parameter space.

In the previous chapter, the heuristic search-based I/O auto-tuning achieved I/O write speedups between 1.27X and 7.74X. However, the overhead of

the genetic algorithm in heuristic search is significant that may severely limit the applicability of such an auto-tuning framework to general-purpose applications.

Rather than the approaches mentioned above, I thought I could use machine learning algorithms to solve this problem. I developed an I/O performance prediction model that can significantly reduce the search time and improve the I/O performance. Here, models are considered that can characterize the I/O performance (e.g., I/O bandwidth, I/O time) in terms of the parallel I/O stack characteristics. A model can be formally used to define the I/O performance of an application as follows:

$$\phi = f(\alpha, \zeta, \omega), \tag{5.1}$$

where α represents a set of observable parameters that describe application characteristics (problem size, I/O pattern, number of cores, etc.), ζ represents a set of observable parameters that describe file system and/or MPI-IO characteristics (Lustre parameters, MPI-IO hints, etc.), ω represents uncontrolled non-observable parameters, and ϕ represents I/O bandwidth or I/O time. In the modelling approach, I aim to understand the relationship between ϕ and the parameters (α, ζ). For a given set of input parameter values in (α, ζ), the function f should give a prediction.

Non-linear regression models, such as support vector machines, random forests, etc., can model the I/O performance in a reasonable amount of time for a given application. I explore *random forest algorithms* to predict the continuous-valued I/O performance because they have a higher classification accuracy than most tree-based algorithms and a higher noise and outlier tolerance.

5.1.1. Performance Models

5.1.1.1. Decision Tree

A *decision tree* is a kind of supervised learning algorithm. In practical applications, the decision tree algorithm is particularly suitable for analyzing discrete data and can be used to solve classification problems and regression problems [58]. The decision tree contains a root node, several decision nodes, and several leaf nodes. The leaf nodes correspond to the prediction results. The data set contained in each node is divided into child nodes according to the result of the attribute test. The path from the root node to each leaf node corresponds to a decision sequence as seen in Figure 5.1 [59].

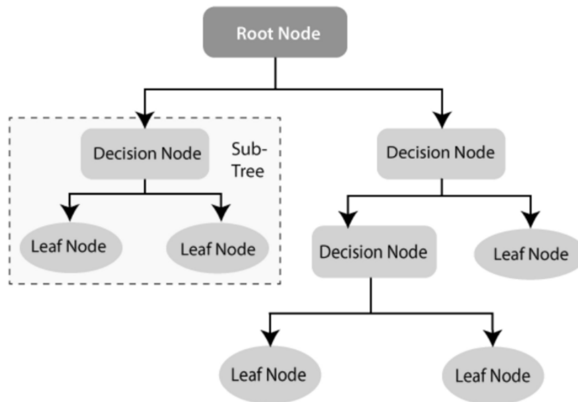


Figure 5.1.: Basic structure of a decision tree [60].

5.1.1.2. Regression Tree

A *regression tree* is basically a decision tree that is used for the regression that can be used to predict continuous-valued outputs. Its construction principle is to divide the input space of training set into subspaces recursively to construct the tree.

Assuming the data set D is defined as follows:

$$D = (x_1, y_1), (x_2, y_2), ..., (x_i, y_i), ..., (x_n, y_n) \tag{5.2}$$

where the independent parameters (e.g., the stripe count) in the model are denoted $x = [x_1, ..., x_n]$ and the scalar-valued output/dependent variable (e.g., the write time or bandwidth) associated with the configuration x by y(x).

The steps of generating a regression tree can be summarized as:

a) Selecting the optimal parameter value and solving the objective function:

$$\min_{j,s}[\min_{c_1} \sum\nolimits_{x_i \epsilon R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum\nolimits_{x_i \epsilon R_2(j,s)} (y_i - c_2)^2] \tag{5.3}$$

Where x_i, j and s are m-dimensional vectors with m features, optimal segmentation variables and optimal segmentation points, respectively. R_1 and R_2 are the two sub-regions after segmentation, and c_1 and c_2 are predicted values for each sub-region.

b) Dividing the area with the selected j, s and get the prediction value of each region:

$$R_1(j,s) = (x|x^j \leq s) \tag{5.4}$$

$$R_2(j,s) = (x|x^j > s) \tag{5.5}$$

$$c_m = \frac{1}{N_m} \sum\nolimits_{x_i \epsilon R_m(j,s)} y_i, x \epsilon R_m, m = 1, 2 \tag{5.6}$$

Where, x^j and N are the *j-th* feature and the total number of features, respectively.

c) Repeating the first two steps (a and b) until the stop condition is met.

d) Dividing the input space into N areas to generate a regression tree.

5.1.1.3. Random Forests

A *random forest* is a regression algorithm that trains and also analyzes previous samples through trees. A regression tree is the basic unit of *random forest*.

Random forests are a set of tree predictors that extract knowledge from numerous decision trees instead of producing a single decision tree [61]. The regression results are the joining of all the tree's output, which is more stable and more robust with respect to noise. Here, the selection of samples is random. The following steps describe random forest regression:

1. Randomly select the data points from the training data sets.
2. Construct a tree correlated with the chosen data points.
3. Step *1* and *2* are iterated m times to generate m decision trees.
4. Predict a data point value using m decision trees. Then calculate the average of the predicting values.

Random forest is a supervised learning algorithm utilizing the ensemble learning method to improve the predictive accuracy and control over-fitting. The trees in random forests can run in parallel. It works by building many decision trees during the training process and gives the individual trees' mean prediction (regression). These decision trees are aggregated into a random forest ensemble that combines their input. Then, results are aggregated so that they can outperform any individual decision tree's output [62]. The diagram of the random forest regression is shown in Figure 5.2.

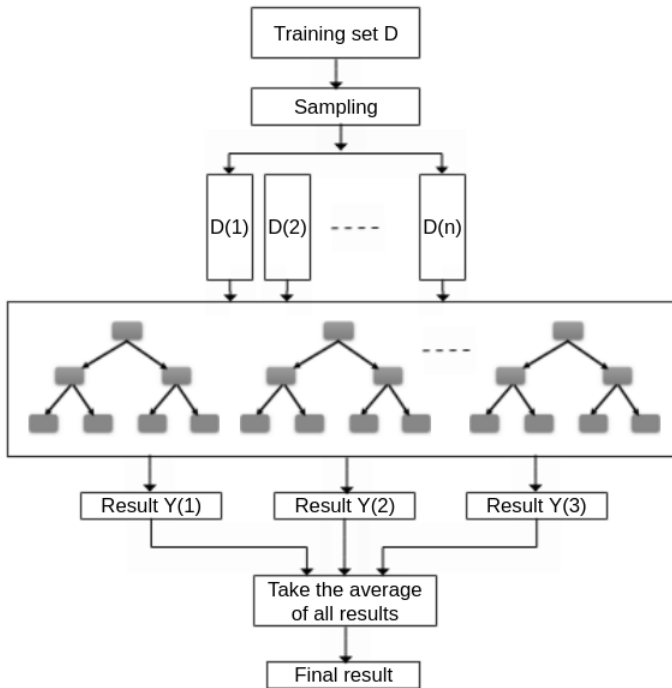


Figure 5.2.: The schematic diagram of the random forest regression.

5.2. Architecture

Figure 5.3 shows the overall architecture of performance modelling-based I/O auto-tuning. First, I/O performance data is collected for a variety of workloads/parameters, and then the I/O performance model *IO_Predictor* is built on the collected data. The model is trained and evaluated in the optimization module *IO_Optimizer*. All possible combinations of tunable parameters (the results from the cartesian product of the values) are given to the *IO_Predictor* as input. The *IO_Predictor* predicts how good I/O performance can be achieved if the configuration parameters in each combination are set. Then, the predictions are sorted from the highest, and the best-performing configuration settings among the predicted I/O performances

are selected for the given application and scale. The selected settings are saved as a configuration file.

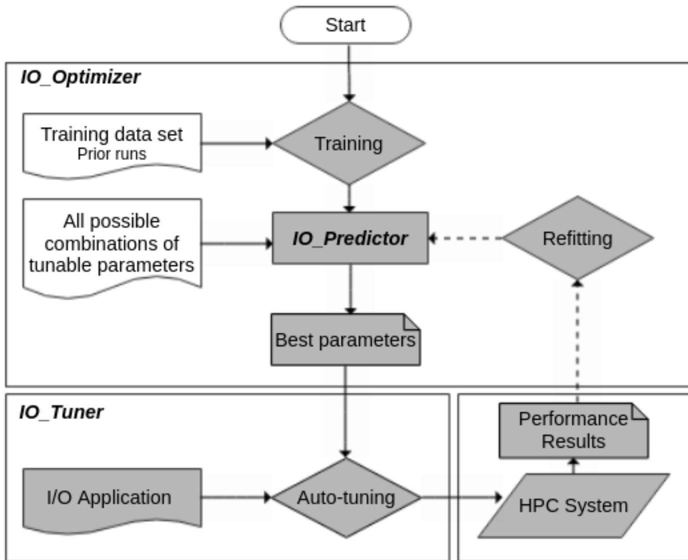


Figure 5.3.: Overall architecture of performance modelling-based I/O auto-tuning.

The tuning module *IO_Tuner* takes the parameters suggested by the *IO_Predictor* and dynamically passes them to the MPI-IO routines of the application or benchmark in the PMPI wrapper. After executing I/O operations, performance results are used to refit the *IO_Predictor* with the dynamic conditions of a parallel I/O system adaptively.

5.2.1. IO_Tracer: Monitoring I/O Activity

The *IO_Tracer* is implemented with *one process tracing* policy. Figure 5.4 shows its tracing process. The *rank 0* is responsible for collecting tracing results and storing them into its allocated local memory. Meanwhile, the other MPI processes stay idle after contributing to the duration of their I/O operations. As a result, all other MPI processes keep running while the *rank*

0 MPI process saves the tracing results into its local memory.

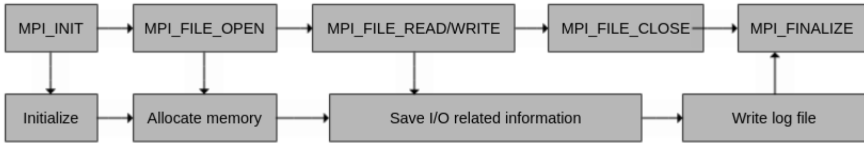


Figure 5.4.: Tracing process of the *IO_Tracer*.

After tracing the MPI writing operations, the *IO_Tracer* saves the I/O related information, such as operations' duration, data transfer sizes, operation bandwidths, names of MPI-IO subroutines, MPI info, objects, and so on, into the allocated memory. As soon as the application calls the `MPI_FINALIZE` subroutine, the *rank 0* MPI process writes all tracing results into the log file and finalizes the *IO_Tracer*.

5.2.2. IO_Predictor: Modelling I/O Performance

I collected a data set for the training phase with the tracing module *IO_Tracer* from multiple runs in different problem sizes and settings of tunable configuration parameters. Then, I applied several regression methods to construct a predictive performance model on the data set during the training phase. Remarkably, the *random forest* algorithm gave successful prediction results for such a non-linear relationship between parameters. The model built is called *IO_Predictor*. The *IO_Predictor* predicts I/O performance for all possible combinations of tunable parameters. It then sorts predictions and selects the best-performing configuration settings for the given scale.

Figure 5.5 shows tracing and optimizing processes together. For implementing the I/O performance model, I have collected the application log files by *IO_Tracer*. These log files are transformed into CSV files and the training data set is created. To model the *IO_Predictor* random forest algorithm is used on the training data set. Mainly, configuration parameters are considered the input variable (independent variable), and I/O bandwidth or I/O time are considered output variables (dependent variables). The *IO_Predictor* predicts the output variables from the input variables.

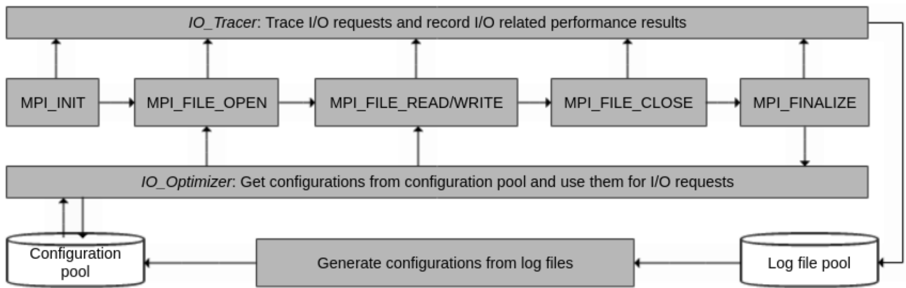


Figure 5.5.: Tracing and optimizing processes.

Implementation of the model is described in the following flowchart (Figure 5.6).

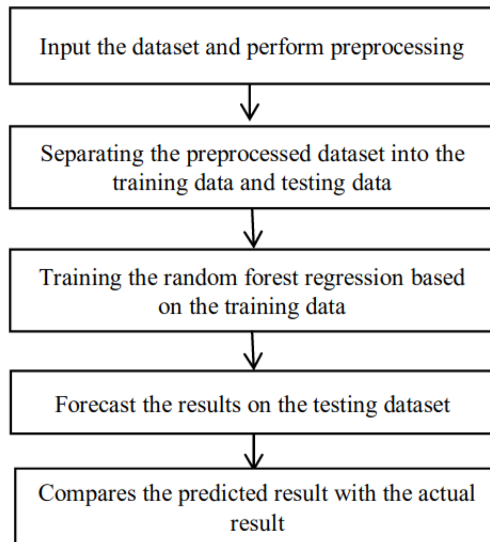


Figure 5.6.: Implementation of the performance model.

A detailed description of Figure 5.6 is given below.

- **Input the data set and perform preprocessing:** I have used mainly

the previous two years' data as an input from the year 2019 to 2021. Data set is needed to preprocess before the training phase, such as identifying and handling the missing values in order to ensure performance.

- **Separating the preprocessed data set into training and testing data set:** The train-test split procedure allows us to compare the performance of machine learning algorithms for our predictive modelling problem. Training data set is used in order to fit the machine learning model. Testing data set is used to evaluate the fit machine learning model. For the *IO_Predictor*, the results are based on the split of training (70%) and testing subsets (30%).
- **Training the *IO_Predictor* using the training data set:** To build the *IO_Predictor* random forest algorithm is implemented on the training data set. In the implementation of the *IO_Predictor*, there are some important parameters:
  - *n_estimators*: It indicates the number of trees employed in the regression forest. It manages the number of trees to compose and create various decision trees. In this study, it is assigned and evaluated up to 100.
  - *criterion*: It is a function to measure the quality of a split such as "squared_error", "absolute_error", "poisson". In this study, "squared_error" and "absolute_error" are evaluated.
  - *max_depth*: It indicates the maximum depth of the tree, namely the longest path between the leaf node and the root node. By using the max_depth parameter, we can specify what depth we want every tree in the random forest to grow. In this study, it is assigned and evaluated up to 5.
- **Result predicting on test data:** Predicting is done on testing data at first for evaluation of the model.
- **Result comparison with the actual results:** Here, mainly predicted results and the actual results are compared.

5.2.3. IO_Tuner: Setting I/O Parameters at Runtime

The *IO_Tuner* component is a parallel I/O tuning module that takes the best-performing I/O parameters found by the *IO_Predictor* included in the *IO_Optimizer* and then dynamically sets these parameters at different layers of the I/O stack. Currently, the *IO_Tuner* works on the MPI-IO ROMIO library and Lustre parallel file system parameters.

At first, the *IO_Tuner* reads the configuration file, including good configuration settings found by the *IO_Predictor*. Then, as soon as I/O applications call MPI-IO subroutines, the *IO_Tuner* is triggered to apply these configurations before executing the I/O operation transparently. Finally, it passes the intercepted MPI-IO functions of the application in the PMPI wrapper. The good parameters are set at this stage, and then the original MPI-IO function is called. In this way, auto-tuning can be done transparently to the users without source code modification.

The previously found good configuration settings could be outdated and no longer optimal. Sometimes users achieve better I/O performance with some "brand new" configurations. After executing I/O operations, performance results can be used to refit the *IO_Predictor* with the dynamic conditions of a parallel I/O system adaptively for scientists and engineers to find out the latest good configurations.

5.3. Implementation

5.3.1. Benchmarks

I chose two I/O benchmarks and a real molecular dynamics code to evaluate performance modelling-based I/O auto-tuning approach: *IOR, MPI-Tile-IO* and *ls1 Mardyn*. These represent different I/O write motifs with different problem sizes.

- **IOR** [56]: The IOR (LLNL 2015) is an I/O benchmark developed at Lawrence Livermore National Laboratory (LLNL). It is one of the main HPC I/O benchmarks because it is highly configurable and supports

various APIs to simulate I/O load.

- **MPI-Tile-IO** [57]: The MPI-Tile-IO benchmark tests the I/O performance in a real-world scenario. The purpose of the MPI-Tile-IO is to test the performance of an underlying MPI-IO and file system implementation under a non-contiguous access workload. It tests how it is performed when challenged with a dense 2D data layout.
- **ls1 Mardyn** [11]: ls1 Mardyn is a molecular dynamics simulation program which is optimized for massively parallel execution on supercomputing architectures. It is a highly scalable code. In this study, it is used for writing check-points in the different domain sizes to scale the number of particles.

5.3.2. System setup

The experiments were conducted on the NEC Cluster platform (Vulcan) at HLRS. Section 2.2 shows the technical details about the Vulcan. As for the experimental Lustre file system, the default setup of Lustre striping configuration is `striping_factor`=4 and `striping_unit`=1048576. OpenMPI version is from version 4.0.3.

5.3.3. Parameter space

The α configuration parameters which are worked on are a number of cores, problem size, and I/O pattern (independent or collective), while the ζ configuration parameters include Lustre file system parameters (striping size and striping count), MPI-IO parameters (whether or not to perform collective I/O). [8], [10], [17] and [19] show that these parameters have an essential effect on the parallel I/O performance.

The optimization criteria ϕ is the I/O bandwidth or I/O time. In the modelling approach, the aim is to understand the relationship between ϕ and the parameters (α, ζ). For a given set of input parameter values in (α, ζ), the function f should give a prediction. If it provides distributional information (such as standard deviation), the variability in ϕ and the ω parameters can

also be captured. The ω parameters, such as other processes' load on the file system, are also potentially important but not easily observable. Thus these parameters have been ignored for simplicity in this study.

For the training process of the I/O performance model, I selected a few meaningful parallel I/O parameters for parallel I/O stack layers based on previous research efforts and experience. I have chosen most of the parameter values to be powers-of-two except for some parallel file system parameters. To avoid potential conflicts between concurrently running I/O applications, the maximum `striping_factor` was set as 16. Since large values of `striping_unit` may result in longer lock hold times, the values of `striping_unit` were selected from 1 MB to 16 MB with powers-of-two. As for the collective I/O operations, values are defined as automatic, disable and enable. Table 5.1 shows the different settings that comprised the set of training configurations in the set of experiments. This leads to 5 x 5 x 3 = 75 configurations used for training the model at each scale from 24 to 2400 for each I/O pattern.

A user can update the parameter space by simply modifying the parameter list as needed. Adding new parameters to the search needs simple modifications to *IO_Tuner*.

Table 5.1.: Configurations' searching scope for training process

Name	**Value**	**Quantity**
number of cores	24; 120; 240; 1200; 2400	5
striping_factor	1; 2; 4; 8; 16	5
striping_unit	1 MB; 2 MB; 4 MB; 8 MB; 16 MB	5
romio_cb_write	automatic; disable; enable	3
access pattern	independent; collective	2

The sum of different combinations of configuration settings listed in Table 5.1 is 375 ($5 \times 5 \times 3 \times 5$) for independent or collective I/O separately.

5.3.4. Scale and data set sizes

Experimentally, I ran tests using different file sizes and different core counts on Vulcan. I collected I/O performance data at different concurrencies, 24; 120; 240, 1,200, and 2,400 cores. I consider models that could be employed in tuning for multiple different file sizes simultaneously. I defined training sets depending on the core counts and file sizes to use HPC resources effectively.

The data transfer sizes were chosen from 100 bytes to 2,000,000,00 bytes (56 different sizes) for training at 24; 120, and 240 processes. I have chosen to decrease the size of the training set as the core counts (and hence file sizes) increase because of the corresponding increase in computational resources required. Therefore, the data transfer sizes were chosen from 100 bytes to 61,440,000 bytes (48 different sizes) for training at 1200 and 2400 processes.

The training set for each of the experiments and their file sizes are shown in Table 5.2. This training process is performed separately for independent and collective access patterns.

Table 5.2.: Breakdown of training set for the I/O model.

# of Cores	# of Sizes	# of Configurations	Training Set Size
24	56	75	4200
120	56	75	4200
240	56	75	4200
1200	48	75	3600
2400	48	75	3600

5.3.5. Log files and creating data set

The I/O performance data set was obtained from the benchmark runs. Developers typically run their application code on a system for an extended period. They also play around with many different experiments by applying various configuration settings and other parameters. They typically save their runs and their results in a database. Thus, in some cases, log files from

previous runs can also be used in training phase.

A simple log file processing utility is implemented in this study. It converts log files generated by the tracing module *IO_Tracer* to CSV files. Thus, system administrators or engineers can easily process the files with their preferred tools, such as Microsoft Office, to analyze the MPI-IO operations. This part can be improved with a visualization utility in future work. I/O behaviours become easier to interpret for system administrators or engineers in this way.

Chapter 6

Results

6.1. I/O Variability on Single Node

I begin by investigating the issue of building a model on a single node for write bandwidths when Lustre and MPI-IO settings are changed. In order to focus on writing a single shared file on the Lustre file system, I use a self-implemented program that uses MPI-IO (Listing A.2). Two different programs are implemented to separately test collective I/O and independent I/O access patterns.

The data transfer size is fixed to about 20 MB per process (20 * 24 = 480 MB file size for a single node). Table 5.1 shows the different settings that comprised the set of training configurations in the set of experiments. This leads to 5 x 5 x 3 = 75 configurations used to train the model at each scale and the access pattern.

One of my objectives is to analyze write bandwidth variability in simple settings for a single node. Therefore, I tested all 75 training configurations in three different experiments (each taking place on different days of a week) to increase the possibility of exposure to different levels of interference from the I/O activity of other jobs running on a shared system such as Vulcan.

Figure 6.1 shows the 225 write bandwidths obtained as part of these three experiments. The 75 training configurations are sorted by the minimum write bandwidth across the three experiments. Variability within a particular configuration is illustrated by a vertical line connecting the three write bandwidths for that experiment. It can be seen that, even in this single-node setting, interference/noise can have a significant impact on performance.

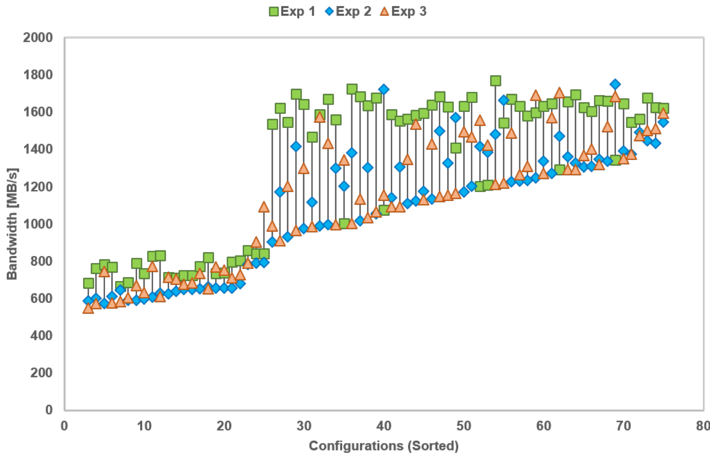


Figure 6.1.: I/O performance variability and effect of interference on a single node writing to a file.

This variability can significantly complicate the modelling process since it necessitates a more careful definition of the modelling objectives prior to performing experiments. For example, if one wishes to model "average" I/O performance, then experimental setups would need sufficient samples across different system states/sources of the variability. Furthermore, since this variance is nonstationary (having different magnitudes from configuration to configuration), accurately modelling performance across the entire configuration space can be a daunting task, requiring one to also model the variability over the configuration space.

In the context of this study, I am looking to identify sets of high-performing configurations (that are not already in the training set) for subsequent

evaluation. Figure 6.1 shows, for example, that it is observed that the highest-performing configurations tend to be less sensitive to noise; reordering the configurations based on the mean or median of the three experiments has little effect on the constituents of the highest-performing quartile. As a result, I have decided to use the minimum bandwidth of each of the experiments in building performance models.

6.2. I/O Variability on Multiple Nodes

Having observed that I/O performance variability when one node writes to one file, I want to investigate when writing to shared files from multiple nodes. I use my micro-benchmark with 50 nodes and file size of 20 GB.

I performed three different experiments on each of the 75 configurations, with the training data is again taken as the minimum write bandwidth over these runs Figure 6.2.

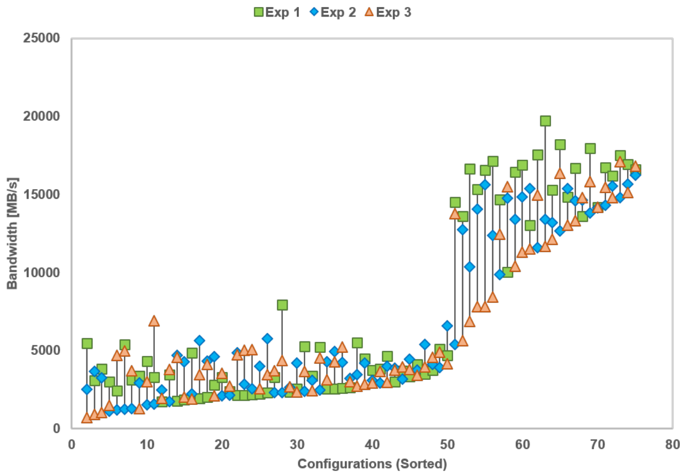


Figure 6.2.: I/O performance variability and effect of interference on multiple nodes writing to a file.

Note that the empirical data suggests that the variability is smaller for

the configurations with lower write bandwidths. Furthermore, although the models considered in this study do not directly account for the variability, I observe that the realized predictions tend to yield more accurate predictions for those configurations where little variability is seen.

6.3. Training Process

I applied several non-linear regression methods to construct a predictive I/O performance model on the training data set. Table 6.1 gives a comparison of three popular algorithms; *support vector machines*, *multi-layer perceptron* and *random forest* over correlation coefficient, mean absolute percentage error, root mean square percentage error, time taken to build model and time taken to test model. Remarkably, the *random forest regression model* gave successful prediction results for such a non-linear relationship between parameters in both accuracy and modelling time.

Table 6.1.: Comparison of regression algorithms on I/O performance prediction.

	Support vector machine	**Multi-layer perceptron**	**Random forest**
Correlation coefficient	0.653	0.942	0.95
Mean absolute percentage error	139.357	49.957	5.231
Root mean square percentage error	999.954	357.882	29.591
Time taken to build model	998.37 seconds	4.25 seconds	2.27 seconds
Time taken to test model	0.03 seconds	0.01 second	0.31 seconds

The training process used a self-implemented MPI program to test different combinations of configuration settings. The training set for each of the scales and file sizes are shown in Table 5.2. For example, the data transfer sizes were chosen from 100 bytes to 2,000,000,00 bytes (56 different sizes) for training at 24 processes; therefore, 56 x 75 = 4200 data of possible combinations in the set. This training process was performed separately for independent and collective access patterns at each scale.

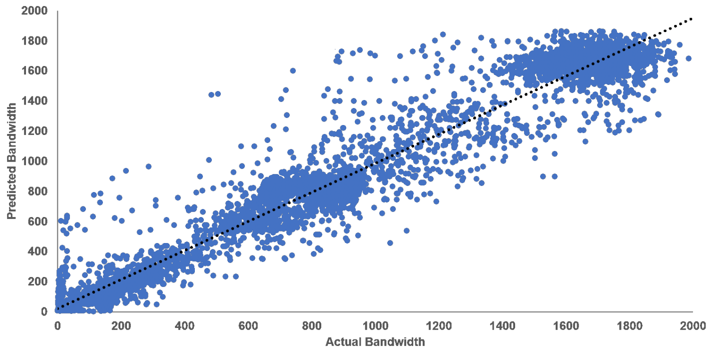
I collected a training data set with the tracing module *IO_Tracer* from runs in multiple problem sizes and scales (Section 5.3.4). For example, the

self-implemented MPI program (Listing A.2) for the training process created 3,600 (48 × 75) files for a 2400PE experiment and deleted the training files after closing them. Therefore, the evaluation of the training process only occupied 0.31 TB of storage space for the most extensive file created by 2,400 processes.

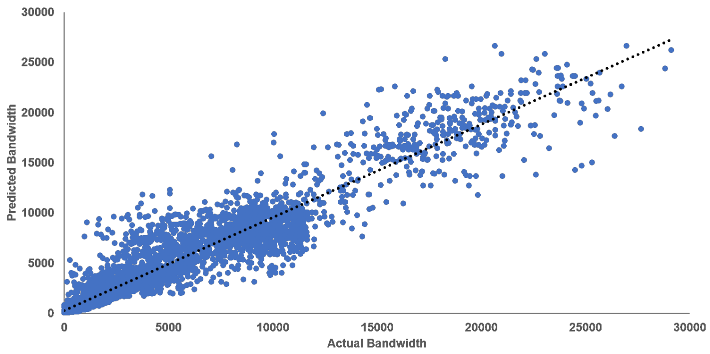
Running the training process once for this search space consumed ≈ 37.8 hours of wall time in total. However, the computing resource consumption for training specific applications was very little compared to the saved core hours achieved.

I ran the *IO_Predictor* for a 70/30 train/test split to predict the write bandwidth. The model it took around 4.5 seconds to train for a single node. The resulting model is then evaluated on the testing subset. The primary objective is to keep these predicted values closer to actual values. Figure 6.3 shows the correlation between actual values of write bandwidths and predicted values of write bandwidths for a 30/70 split of train/test data (training (70%) and testing (30%)) by using scatter plots. The ideal graph should be a dashed black line. The plots are well-centred around the black dotted line (ideal case with 100% accuracy).

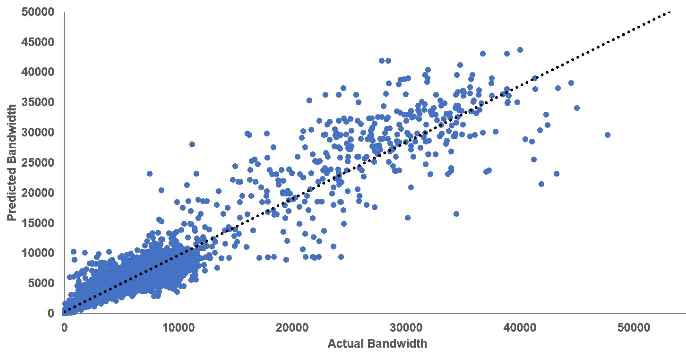
Mean absolute error is the sum of all absolute/positive errors. So we add the positive values of all errors to find their mean. This is the sum of absolute differences between actual (observed) and predicted values. It does not consider the direction, namely, positive or negative. Mean square error is always positive, and if a value is closer to 0 or a lower value is better. The square root of mean square error yields root means square error. It is the standard deviation of the error (residual error). It shows the spread of the residual errors. It is always positive, and a lower value indicates better performance. The ideal value would be 0, but it is never achieved. The effect of each error on root mean square error is directly proportional to the squared error; therefore, root mean square error is sensitive to outliers and can exaggerate results if there are outliers in the data set. It is observed that higher root mean square error for higher core counts because of the high I/O performance variability. Table 6.2 shows performance modelling times and validation results.



(a) Single node



(b) Multiple nodes - 50 nodes



(c) Multiple nodes - 100 nodes

Figure 6.3.: Correlation between actual (observed) and predicted write bandwidths on training (70%) and testing subsets (30%).

Table 6.2.: The *IO_Predictor* performance modelling times and validation results.

	1 node	**50 nodes**	**100 nodes**
Time taken to build model	4.5 seconds	2.61 seconds	2.47 seconds
Time taken to test model on test split	0.71 seconds	0.39 seconds	0.29 seconds
Correlation coefficient	0.9758	0.9526	0.95
Mean absolute percentage error	4.2186	3.3096	5.2312
Root mean square percentage error	15.1091	30.204	29.5911
Mean absolute error	78.2782	832.1251	980.2587
Root mean squared error	140.708	1604.4709	2349.391

As a result, the training process on a single node (24PE) managed to find out 56 different configuration sets, and 48 different configuration sets on 100 nodes (2400PE) with acceptable accuracy and error values.

Table 6.3 lists the most of the good configurations found based on the different file size groups. Listing A.3 shows how these configurations are stored in the configuration file. Maximum and minimum data transaction sizes of file size groups are also provided in Table 6.3.

Table 6.3.: Found good configurations for different file size groups.

group	Min. Size (B)	Max. Size (B)	striping factor	striping unit	collective I/O
1	100	256	4	8 MB	automatic
2	256	384	1	2 MB	automatic
3	384	512	4	8 MB	enable
4	512	640	8	4 MB	automatic
5	640	768	4	1 MB	enable
6	768	896	4	4 MB	automatic
7	896	1024	4	1 MB	enable
8	1024	2048	4	4 MB	automatic
9	2048	3072	4	1 MB	enable
10	3072	4096	4	8 MB	enable
11	4096	5120	4	1 MB	enable
12	5120	6144	4	1 MB	enable
13	6144	7168	4	1 MB	enable
14	7168	8192	4	1 MB	enable
15	8192	9216	6	1 MB	automatic
16	9216	10240	4	1 MB	enable
17	10240	20480	8	1 MB	enable
18	20480	30720	12	1 MB	automatic
19	30720	40960	12	1 MB	automatic
20	40960	51200	12	1 MB	automatic
21	51200	61440	8	2 MB	automatic
22	61440	71680	8	2 MB	automatic
23	71680	81920	16	2 MB	automatic
24	81920	92160	12	2 MB	automatic
25	92160	102400	12	1 MB	automatic
26	102400	204800	16	1 MB	automatic
27	204800	307200	12	8 MB	automatic
28	307200	409600	16	2 MB	enable
29	409600	512000	16	2 MB	automatic
30	512000	614400	16	1 MB	enable
31	614400	716800	16	2 MB	automatic
32	716800	819200	16	2 MB	automatic
33	819200	921600	16	1 MB	automatic
34	921600	1024000	16	2 MB	automatic
35	1024000	2048000	16	1 MB	automatic
36	2048000	3072000	16	1 MB	automatic
37	3072000	4096000	16	1 MB	automatic
38	4096000	5120000	16	1 MB	automatic
39	5120000	6144000	16	2 MB	automatic
40	6144000	7168000	16	1 MB	enable
41	7168000	8192000	16	2 MB	enable
42	8192000	9216000	16	1 MB	enable
43	9216000	10240000	16	8 MB	automatic
44	10240000	20480000	16	2 MB	automatic
45	20480000	30720000	16	1 MB	automatic
46	30720000	40960000	16	8 MB	automatic
47	40960000	51200000	16	4 MB	automatic
48	51200000	61440000	16	1 MB	disable
49	61440000	71680000	8	4 MB	disable
50	71680000	81920000	16	1 MB	disable
51	81920000	92160000	16	2 MB	disable
52	92160000	102400000	12	8 MB	disable

6.4. Evaluations: I/O Benchmarks

In this section, the training results were evaluated using two I/O benchmarks, the IOR and the MPI-Tile-IO on Vulcan (Section 5.3.2). The IOR benchmark was used to simulate collectively writing a single-shared file with 1 MB, 32 MB, and 64 MB data block sizes. On the other hand, the MPI-Tile-IO benchmark simulated collectively writing a single-shared file with 4 x 4 tiles, 8 x 8 tiles, 16 x 16 tiles, and each tile is 256 x 256 elements, 4096 bytes per element. The experiments have been repeated multiple times, and average values are presented.

Proper configurations change depending on how many bytes of data are written by each MPI-IO process. However, it would be too expensive to find out a good configuration for every data transfer size as suggested in [8]. Instead, different data transfer sizes which may have similar good configurations can be grouped into file indexes. For example, 56 data transfer size groups in the prototype are defined from 100 B to 2,000,000,00 B (195.31 MB), and any data transfer size larger than 195.31 MB belongs to the 56th file size group. The model's optimal configurations of unknown data sizes (untrained explicitly) were automatically selected based on the number of processes and data size group.

The default setup of Lustre striping configuration on the experimental file system was `striping_factor`=4 and `striping_unit`=1048576. The bottleneck of one Lustre OST for collective writing operation was quickly reached. The found good configurations in the training phase were implemented here for accelerating MPI applications.

Figure 6.4 presents the performance improvements obtained for the IOR benchmark by collectively writing a single-shared file with 1 MB, 32 MB, and 64 MB data block sizes, respectively. Y-axis represents I/O bandwidth in MBps, and the x-axis represents transfer sizes (in MB). The scales of the I/O bandwidth axes are different in the plots. The improvement kept rising when the benchmarks scaled out with the found good configurations—for (240PE), writing performance increased by collective buffering and the larger size of Lustre `striping_unit`. This change brought about a 2.34x improvement

in I/O performance (240PE). The writing performance improved for the large jobs such as (1200PE) and (2400PE) by disabling the collective buffering, increasing the Lustre `striping_factor` and Lustre `striping_unit`. As a result, 5.85x and 18.95x improvements over the default settings are achieved for (1200PE) and (2400PE), respectively. Disabling the collective buffering to avoid the inter-processes communication of collective I/O algorithm like two-phase I/O (Section 2.4.3.2) improved the I/O performance. The training process showed that disabling the collective buffering accelerated large-scaled write operations collectively.

Figure 6.5 presents the performance improvements obtained for the MPI-Tile-IO benchmark by collectively writing a single-shared file with 4 x 4 tiles, 8 x 8 tiles, and 16 x 16 tiles; each tile is 256 x 256 elements. 4096 bytes, 8092 bytes, and 16184 bytes per element were selected, respectively. Y-axis represents I/O bandwidth in MBps, and the x-axis represents element sizes of core times the core number of tiles (in KB). The scales of the I/O bandwidth axes are different in the plots. The improvement kept rising when the benchmarks scaled out, even though the number of Lustre OSTs did not change since 64PE—for (16PE), writing performance increased by collective buffering and the larger size of Lustre `striping_unit` about 1.07x improvement. The writing performance increased for the other larger jobs such as (64PE) and (256PE) by disabling the collective buffering and increasing Lustre `striping_factor`. As a result, 1.61x and 5.07x improvements over the default settings are achieved for (64PE) and (256PE), respectively.

Table 6.4 shows the I/O performances of the default and the optimized experiments for three use cases (in different data sizes) presented in Figure 6.4 and Figure 6.5. Table 6.4 also shows the speedup that the auto-tuned settings achieved over the default settings for each experiment.

The optimized results were obtained in all experiments by the proper configurations promising good performance found by performance-modelling based I/O auto-tuning system. Using the *IO_Predictor* model, an increase in I/O bandwidth of up to 18.9×over the default parameters for the IOR benchmark and 5.1×over the default parameters bandwidth for the MPI-

Table 6.4.: I/O speedups of applications with optimized parameters over default parameters.

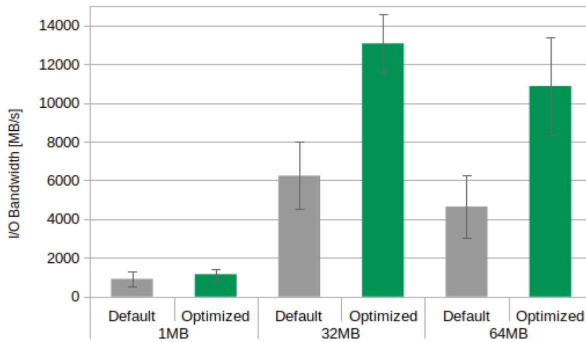
Application		IOR (MB/s)			MPI-Tile-IO (MB/s)		
#Cores		**240**	**1200**	**2400**	**16**	**64**	**256**
Use case 1	**Default**	913.89	1659.64	1483.18	854.71	2400.53	7530.95
	Tuned	1155.89	2440.42	3654.74	1012.69	3099.70	10078.97
	Speedup	1.26	1.47	2.46	1.18	1.29	1.33
Use case 2	**Default**	6238.56	6400.08	2849.65	875.50	1974.46	6138.91
	Tuned	13067.73	25121.44	41475.43	989.94	2792.39	10384.21
	Speedup	2.09	3.93	14.55	1.13	1.41	1.69
Use case 3	**Default**	4645.64	6254.05	2162.98	726.43	1759.32	2122.02
	Tuned	10866.29	36557.57	40980.18	783.73	2833.29	10771.1
	Speedup	2.34	5.85	18.95	1.07	1.61	5.07

Table 6.5.: Found some good configurations for the IOR and the MPI-Tile-IO benchmarks.

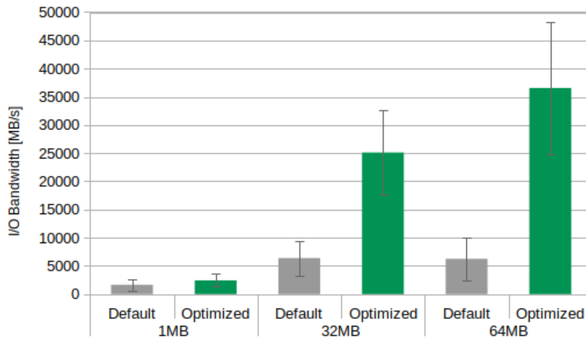
Application	#Cores	striping factor	striping unit	collective I/O
IOR	240	4	4194304	automatic
	1200	16	16777216	disable
	2400	16	16777216	disable
MPI-Tile-IO	16	4	4194304	automatic
	64	16	1048576	disable
	256	16	4194304	disable

Tile-IO benchmark were achieved. The found good configurations of the most successful cases in experimental results for the IOR and the MPI-Tile-IO benchmarks are given in Table 6.5 as an example. I note bigger improvements for larger transfer sizes because tuning the default parameters matter more for large data transfers.

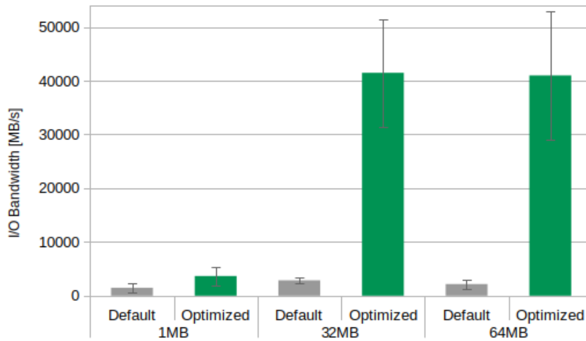
Note that only a subset of the combinations was run due to limited access to the platform.



(a) 240 cores

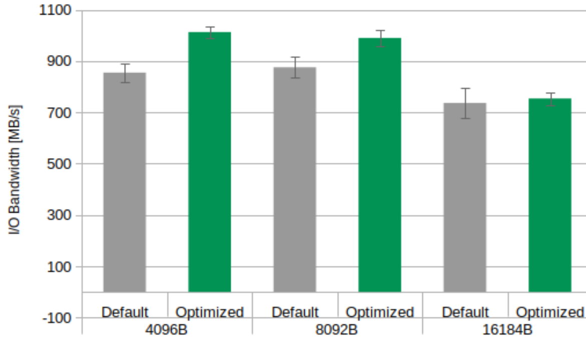


(b) 1200 cores

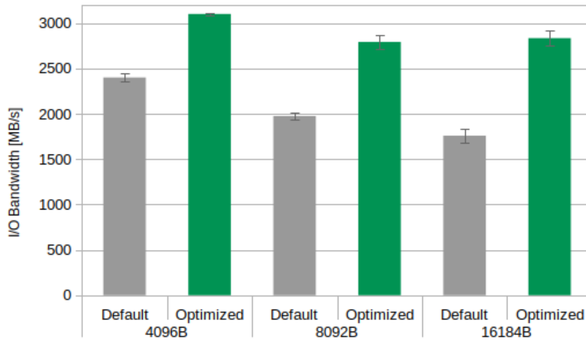


(c) 2400 cores

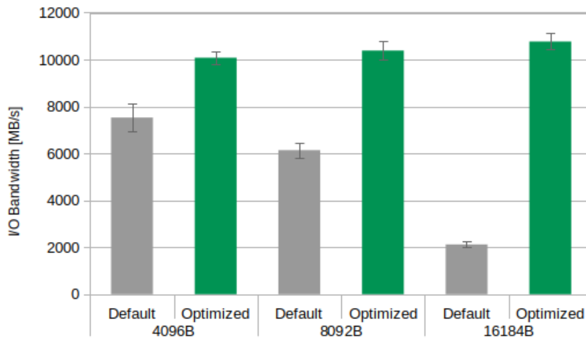
Figure 6.4.: Default vs. optimized write bandwidth on the IOR for various transfer sizes.



(a) 16 cores



(b) 64 cores



(c) 256 cores

Figure 6.5.: Default vs. optimized write bandwidth on the MPI-Tile-IO for various transfer sizes.

6.5. Engineering Use Case : ls1 mardyn

Today, molecular dynamics simulations are essential in many areas of research and industry, including biochemistry, solid-state physics, chemical engineering, etc. The increasing computing power of supercomputers allows them to handle increasingly complex problems. It enables more sophisticated molecular models with an increasing number of previously computationally intensive particles.

A molecular dynamics simulation code ls1 mardyn[1] is a highly scalable code that is optimized for parallel execution on supercomputing architectures aiming at investigating challenging scenarios. In the regarded systems, the spatial distribution of the molecules may be heterogeneous and subject to rapid, unpredictable differences. This is imaged by the algorithms and data structures as well as a highly modular software engineering method [32]. The source code of ls1 mardyn is made publicly available as free software under a two-clause BSD license.

The following sections will show how to analyze the I/O bottleneck in the ls1 mardyn, describe how to use the proposed performance-modelling based auto-tuning solution, and give the optimization results achieved.

6.5.1. Analyzing Application

There are different bottlenecks in the ls1 mardyn. As seen, communication and I/O are the main problems for limited scalability. The I/O part is a significant point for this study, e.g., used for writing checkpoints. After examining the number of writing operations indicated by the tracing log files, I realized that the checkpointing process creates a bottleneck.

ls1 mardyn MPI checkpoint writer writes checkpoint files that can be used to continue the simulation using MPI-IO. A self-programmed data processing application (written in C++) uses the MPI-IO library to process the results. The application uses an MPI parallel programming model, while the I/O requests use MPI-IO.

[1] https://www.ls1-mardyn.de

Users have been complaining about slow I/O performance, particularly during checkpointing operations. After speaking with project users, I discovered that they use the default MPI info setup, `striping_factor`=4, `striping_unit`=1048576 and let the system decide whether to disable or enable collective buffering for writing operations. On the other hand, when analyzing application codes, another critical parameter caught my attention to optimize: `ParticlesBufferSizeMPI`, which is the buffer size for writing particles per rank.

ls1 mardyn plugin "MPICheckpointWriter" gave the performance results as shown in Figure 6.6 using default configurations without setting the parameter of `ParticlesBufferSizeMPI`. Since the output of the plugin MPICheckPointer is I/O time in seconds, the results will be displayed over time for clarity.

```
INFO:   20201214T140720  5276.42 [0]    Computation in main loop took: 4749.37 sec
INFO:   20201214T140720  5276.42 [0]            Decomposition took: 147.266 sec
INFO:   20201214T140720  5276.42 [0]                    Communication took: 79.484 sec
INFO:   20201214T140720  5276.42 [0]                            initSend() took: 13.6711 sec
INFO:   20201214T140720  5276.42 [0]                            testRecv() took: 30.4231 sec
INFO:   20201214T140720  5276.42 [0]            Computation took: 822.452 sec
INFO:   20201214T140720  5276.42 [0]                    Force calculation took: 556.949 sec
INFO:   20201214T140720  5276.42 [0]            IO in main loop took: 3432.73 sec
INFO:   20201214T140720  5276.42 [0]                    Plugin SysMonOutput took: 0.093106 sec
INFO:   20201214T140720  5276.42 [0]                    Plugin MPICheckpointWriter took: 3774.91 sec
INFO:   20201214T140720  5276.42 [0]    Timer SIMULATION_IO took: 0 sec
INFO:   20201214T140720  5276.42 [0]            Phasespace creation took: 9.87826 sec
INFO:   20201214T140720  5276.42 [0]            Final IO took: 511.353 sec
```

Figure 6.6.: ls1 mardyn performance output.

The case analyzed for the ls1 mardyn writes 1.9 GB checkpoint files, with 1000 time steps and 100 write frequency. It writes the data starting at the location specified by the individual file pointer (blocking, non-collective) by calling the `MPI_File_write` subroutine. Another subroutine is the `MPI_File_write_at` that writes a file at an explicitly specified offset (blocking, non-collective). In the first step, I chose one of the found good configurations in a similar case from the evaluation's training process for writing operations to set these MPI info objects as below.

- `striping_factor` = 16
- `striping_unit` = 4194304

- `romio_cb_unit` = automatic

For all cases, I set the application buffer size parameter as below.

- `ParticlesBufferSizeMPI`=33554432

According to the results, playing with configuration parameters was successful in reducing the I/O time of a test process by approximately 71.92% for (240PE), as seen in Figure 6.7. Note that again, since the output of the plugin MPICheckPointer is I/O time in seconds, the results will be displayed over time for clarity.

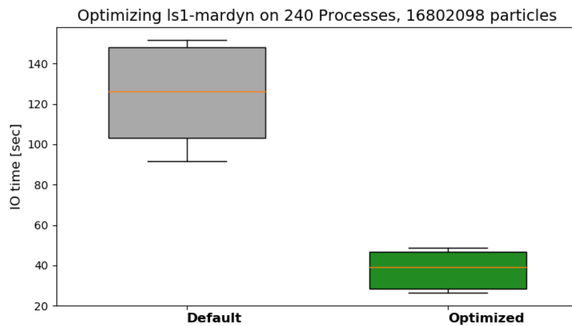


Figure 6.7.: Default setup vs. optimizing for ls1 mardyn checkpointing time.

Then, we decided to deploy a performance-modelling based I/O autotuning system for ls1 mardyn to search configuration space. Furthermore, the application parameter `ParticlesBufferSizeMPI` was also included in the performance modelling process that enables us to find out a good value for the buffer size to be used in the application.

6.5.2. Training

First, the data transfer size of each MPI process must be known to determine the good configurations. To obtain this information, the application is executed with "tracing mode" (Section 3.4).

The *IO_Tracer* generated a log file of 36K, including the tracing results of MPI-IO subroutines in ls1 mardyn. Besides the configuration parameters,

the *IO_Tracer* logs the time stamp when the MPI-IO operations are called. The timestamp information gives a timeline of ls1 mardyn I/O requests. This information can be used in performance modelling for analyzing log file searching range. To understand the I/O behaviour of the application, the *IO_Tracer* saves the aggregated data transfer size for all I/O processes in bytes (*bytes*) and the duration of the slowest process in seconds (*duration*). Based on *bytes* and *duration*, the I/O bandwidth (*bandwidth*) is also calculated and recorded for each checkpoint file.

The log file in JavaScript Object Notation (JSON)-like format has been transformed into a CSV file with the help of the file processing utility (Section 5.3.5). Table 6.6 presents a fragment of tracing results from ls1 mardyn log file for 1,200 processes. According to this, it has extracted 6 data transfer sizes for writing. Six data transfer sizes, 3 (3,600 ÷ 1,200), 4 (4,800 ÷ 1,200), 7 (8,400 ÷ 1,200), 8 (9,600 ÷ 1,200), 20 (24,000 ÷ 1,200) and 1,607,876 (1,929,451,200 ÷ 1,200) bytes, are assigned in the 1st and the 35th file size groups. In training process three of them; 8 (9,600 ÷ 1,200), 20 (24,000 ÷ 1,200) and 1,607,876 (1,929,451,200 ÷ 1,200) bytes have been used. It takes about 3.72 seconds for the `MPI_File_write` subroutine to finish writing 1.9 GB (1,929,451,200 B) data with the given MPI info objects as default. This example shows that the I/O performance of writing 1.9 GB data striped over 4 OSTs with 1 MB striping unit reaches about 494.828 MB/S. Thus, understanding the I/O behaviour of the ls1 mardyn has become more effortless by using *IO_Tracer*.

Table 6.7 presents configurations' searching scope used in training process ls1 mardyn. Besides the previously mentioned factors, MPI info objects are chosen based on the experience of the evaluations in Section 6.4. The maximal number of `striping_factor` was set to 16, considering the I/O performance and the Lustre OSTs' resource competition. 16 MB was chosen for maximal `stripe_size` because of the evaluations in Section 6.4.

Since no training process was done with independent I/O operations, I implemented a self-implemented program that uses MPI-IO, an independent version of the code given in (Listing A.2). Executing the training process

Table 6.6.: A fragment of a CSV file obtained after tracing process on ls1 mardyn.

group	*time_stamp*	*operation*	*bytes*	*duration*	*bandwidth*	*str_factor*	*str_unit*	*romio_cb_write*
1	18.717881	MPI_File_write	24000	0.048611	0.471	4	1048576	automatic
1	18.717947	MPI_File_write	4800	5E-06	917.914	4	1048576	automatic
1	18.718002	MPI_File_write	9600	3E-06	3452.215	4	1048576	automatic
1	18.718037	MPI_File_write	8400	3E-06	3178.914	4	1048576	automatic
1	18.718071	MPI_File_write	3600	2E-06	1590.930	4	1048576	automatic
1	18.718112	MPI_File_write	9600	2E-06	4172.868	4	1048576	automatic
1	19.076904	MPI_File_write_at	9600	0.357778	0.026	4	1048576	automatic
1	19.116025	MPI_File_write_at	9600	0.039074	0.234	4	1048576	automatic
1	19.166615	MPI_File_write_at	9600	0.050565	0.181	4	1048576	automatic
1	19.31351	MPI_File_write_at	9600	0.146863	0.062	4	1048576	automatic
1	19.358743	MPI_File_write_at	9600	0.045212	0.203	4	1048576	automatic
1	19.456525	MPI_File_write_at	9600	0.097745	0.094	4	1048576	automatic
1	19.506593	MPI_File_write_at	9600	0.050042	0.183	4	1048576	automatic
1	19.627125	MPI_File_write_at	9600	0.120202	0.077	4	1048576	automatic
35	23.351185	MPI_File_write	1929451200	3.718603	494.828	4	1048576	automatic

Table 6.7.: Configurations' searching scope for training process ls1 mardyn.

Name	**Value**	**Quantity**
number of processes	1200	1
data transfer size (bytes)	8; 20; 1,607,876	3
striping_factor	1; 2; 4; 8; 16	5
striping_unit	1 MB; 2 MB; 4 MB; 8 MB; 16 MB	5
romio_cb_write	automatic; disable; enable	3
particles' buffer size	1; 4; 8; 16; 32	5

took about 67,8 seconds (about 22,6 core hours) for each case and generated 1125 files for write (3×5×5×3×5= 1125 files) operations in total. Table 6.8 lists the optimal configurations found by the training utility.

Table 6.8.: Found optimal configurations after training process ls1 mardyn.

data size (B)	**striping factor**	**striping unit**	**collective write**	**particles' buffer size**
9600	2	2097152	enable	16777216
24000	2	2097152	enable	16777216
1929451200	16	2097152	disable	33554432

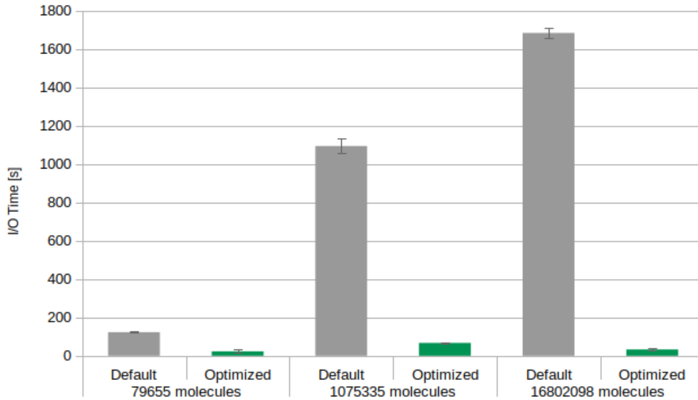
6.5.3. Optimization and Results

The training process results presented that disabling the collective buffering accelerated large-scaled I/O requests writing large data collectively. I

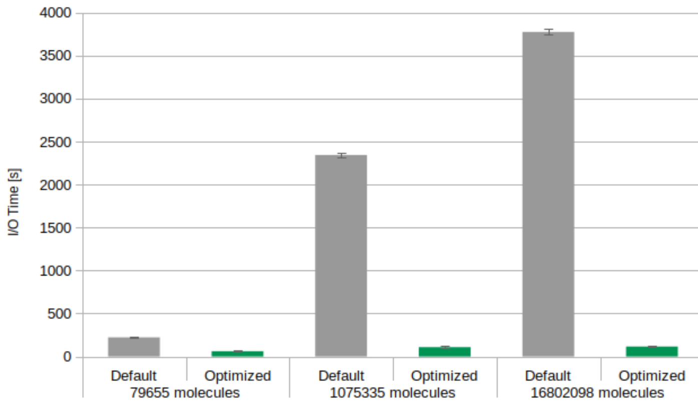
investigated the two-phase I/O algorithm again, and realized that it aimed to optimize the I/O requests for small data transfer sizes, which was also confirmed by the training process. For small data transfer sizes, better performances were achieved by "enabling" (enable) or "letting the system decide to enable or disable" (automatic) the collective buffering. For large data transfer sizes, better performances were achieved by "disabling" (disable) the collective buffering when applications scaled out.

Default vs optimized write I/O times on ls1 mardyn for various transfer sizes running on 240 cores and 1200 cores of Vulcan is given in Figure 6.8. Y-axis represents I/O time in seconds, and the x-axis represents different numbers of molecules. Lower I/O time is better. The scales of the I/O time axes are different in the plots. Note that since the output of the plugin MPICheckPointWriter is I/O time in seconds, the results are displayed over time for clarity.

The optimization effects are represented with the output of the plugin MPICheckPointWriter is I/O time in seconds for the ls1 mardyn process. It took the original setup 3774,91 seconds to complete checkpointing with default configurations and without setting `ParticlesBufferSizeMPI`. Only adding the parameter `ParticlesBufferSizeMPI` to the modelling has successfully accelerated the process enormously. Writing target files has managed to accelerate ls1 mardyn with the found good configuration settings including `ParticlesBufferSizeMPI`. Optimized ls1 mardyn took about 126.4 seconds for the same case in this way.



(a) 240 cores



(b) 1200 cores

Figure 6.8.: Default vs. optimized write times for checkpointing on ls1 Mardyn for various data sizes.

In all experiments, the optimized results were obtained by the configuration promising the best performance predicted by the *IO_Predictor*. Using the *IO_Predictor* for performance modelling on parameters such as application (`ParticlesBufferSizeMPI`), Lustre (`striping_factor` and `striping_unit`) and MPI-IO (`romio_cb_write`), the ls1 mardyn

I/O time got to 32×improvements compared to the default I/O time.

6.5.4. Conclusion

The performance modelling-based auto-tuning system has been shown to be capable of accelerating engineering applications in a production environment in addition to being an I/O monitoring and analyzing tool. In this use case, I used a self-implemented training utility written in C and implemented it with the MPI-IO library to successfully find the good configuration settings for ls1 mardyn, a parallel MPI-IO application written in C++. The I/O time is used here as the performance criterion. Setting the found configurations has saved computing resources each time ls1 mardyn run. Creating files using the recommended setups would improve the reading performance of the following processes.

CHAPTER 7

Conclusion and Future Work

In this dissertation, I have presented and evaluated two different I/O auto-tuning approaches to auto-tune the parallel I/O stack parameters in engineering applications; heuristic search-based and performance modelling-based auto-tuning. These approaches can be understood by users with little knowledge of parallel I/O without any post-processing step. They are implemented upon the MPI-IO library, widely used in modern HPC systems.

Following the current MPI standard allows compatibility with MPI-based scientific and engineering applications while being portable to different HPC platforms deploying different MPI implementations. Evaluating approaches with widely used I/O benchmarks (the IOR and the MPI-Tile-IO) has proved compatibility, scalability, and portability. Tracing utility provides less detailed I/O tracing information than Darshan dose. Therefore, engineers or scientists can easily understand the tracing results without any post-processing utility. Furthermore, its intuitive log files present enough I/O tuning information for the users with little knowledge of parallel I/O to

carry on the analysis and optimizations.

I presented an I/O performance model; the *IO_Predictor* to estimate I/O performance based on the results of the previous runs. It achieves I/O performance improvements for write performance in the popular I/O benchmarks and a real molecular dynamics code on Vulcan. Thereby, the training time to find the best parameters is drastically reduced from hours (application-dependent) for a naïve strategy to only several seconds(data-dependent). This is an enormous improvement in training time over past models for auto-tuning. Furthermore, it increases I/O bandwidth by a factor of up to 18 over the default parameters for collective I/O in the IOR and a factor of up to 5 for the non-contiguous write in the MPI-Tile-IO. The ls1 mardyn I/O time got to 32×improvements compared to the default I/O time. Thus, I demonstrate that this approach can indeed be helpful for the I/O tuning of parallel applications in HPC. Furthermore, the model can be trained with negligible effort for any benchmark or I/O application.

The *IO_Predictor* uses random forest regression and obtains less than 10% median prediction errors for most cases. Looking at the parameters of the random forest regression algorithm, for example, the value of the depth of the tree can be increased, and accuracy can be higher. However, it seems the model is subject to overfitting in this case. A more comprehensive training data set can give better prediction results. My future efforts will further explore more accurate model generations. I plan to feed the *IO_Predictor* with more input data in the future to learn in case of various applications comprehensively. Furthermore, this work can be extended to different data layouts and I/O libraries; such as OMPIO in order to cover more applications.

The parameters discussed are system-dependent, but new parameters can be easily integrated into the configuration files. With a fast and straightforward training utility, searching for new suitable configurations on another HPC platform allows covering more applications and more HPC platforms. Moreover, with the help of statistic utility and log files, scientists and engineers can analyze the I/O behaviours of their applications easily.

Future efforts will further explore more accurate configuration parameters and statistical methods representations. In future work, the auto-tuning

framework will be tested on engineering applications in different professional areas to show usability.

CODE AND FILE SEGMENTS

```
2 int MPI_File_write_all(MPI_File fh, const void *buf, int count
      ,
      MPI_Datatype datatype, MPI_Status *status) {
4   int ret;
    double time_stamp_1, time_stamp_2;
6
    if (running_mode == OPTIMIZE|| running_mode ==
        OPTIMIZE_TRACE) {
8     MPI_Info info;
      PMPI_File_get_info(fh, &info);
10    set_mpi_info_for_write(info, count, datatype,
          mpi_rank_size);
      PMPI_File_set_info(fh, info);
12
      MPI_Offset disp;
14    MPI_Datatype etype;
      MPI_Datatype filetype;
16    char datarep[32];
      int rc = 0;
18    rc = PMPI_File_get_view(fh, &disp, &etype, &filetype,
          datarep);
```

```
    if (rc == 0) {
20      PMPI_File_set_view(fh, disp, etype, filetype, datarep,
            info);
    }
22  }

24  // real MPI function call
  __real_PMPI_File_write_all = dlsym(RTLD_NEXT, "
      PMPI_File_write_all");
26  time_stamp_1 = PMPI_Wtime();
  ret = __real_PMPI_File_write_all(fh, buf, count, datatype,
      status);
28  time_stamp_2 = PMPI_Wtime();

30  if (ret != MPI_SUCCESS) {
    fprintf(stderr, "PMPI function call with MPI error number:
        %d\n", ret);
32    return ret;
  }
34
  double write_time = time_stamp_2 - time_stamp_1,
      longest_write_time;
36  PMPI_Allreduce(&write_time, &longest_write_time, 1,
      MPI_DOUBLE, MPI_MAX, mpi_comm);

38  if (rank == MASTER_RANK) {
      ret = record_IO_info(mpi_comm, info, count, datatype,
40          mpi_rank_size, "MPI_File_write_all",
              longest_write_time);
    }
42  return ret;
}
```

Listing A.1: Code segment for MPI_File_write_all() wrapper.

```
2 int train_collective_write(char *file_name, int local_size,
      int offset, int my_rank, MPI_Info info) {
  MPI_File fh;
4  int *local_array;
  int i, ret;
```

```
  ret = MPI_File_delete(file_name, MPI_INFO_NULL);
  local_array = (int*) malloc((size_t) (local_size * sizeof(
      int)));
  if (local_array == NULL) {
    return -1;
  }
  for (i = 0; i < local_size; i++) {
    local_array[i] = offset + i;
  }
  ret = MPI_File_open(MPI_COMM_WORLD, file_name,
      MPI_MODE_CREATE | MPI_MODE_WRONLY, info, &fh);
  if (ret != MPI_SUCCESS) {
    fprintf(stderr, "Could not open training file\n");
    MPI_Abort(MPI_COMM_WORLD, ret);
  }
  ret = MPI_File_set_view(fh, my_rank * (MPI_Offset)
      local_size * sizeof(int), MPI_INT, MPI_INT, "native",
      info);
  if (ret != MPI_SUCCESS) {
    fprintf(stderr, "Could not set view\n");
    MPI_Abort(MPI_COMM_WORLD, ret);
  }
  ret = MPI_File_write_all(fh, local_array, local_size,
      MPI_INT, MPI_STATUS_IGNORE);
  if (ret != MPI_SUCCESS) {
    fprintf(stderr, "Could not write training file\n");
    MPI_Abort(MPI_COMM_WORLD, ret);
  }
    ret = MPI_File_close(&fh);
  if (ret != MPI_SUCCESS) {
    fprintf(stderr, "Could not close training file\n");
    MPI_Abort(MPI_COMM_WORLD, ret);
  }
  free(local_array);
  return ret;
```

Listing A.2: Self-implemented training program for collective writing.

```

{"group":6,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
```

```
    },{"cb_nodes":"4"},{"striping_factor":"4"},{"striping_unit
    ":"4194304"}]}
{"group":8,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
    },{"cb_nodes":"4"},{"striping_factor":"4"},{"striping_unit
    ":"4194304"}]}
{"group":1,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
    },{"cb_nodes":"4"},{"striping_factor":"4"},{"striping_unit
    ":"8388608"}]}
{"group":15,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
    },{"cb_nodes":"6"},{"striping_factor":"6"},{"striping_unit
    ":"1048576"}]}
{"group":2,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
    },{"cb_nodes":"1"},{"striping_factor":"1"},{"striping_unit
    ":"2097152"}]}
{"group":21,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
    },{"cb_nodes":"8"},{"striping_factor":"8"},{"striping_unit
    ":"2097152"}]}
{"group":22,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
    },{"cb_nodes":"8"},{"striping_factor":"8"},{"striping_unit
    ":"2097152"}]}
{"group":4,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
    },{"cb_nodes":"8"},{"striping_factor":"8"},{"striping_unit
    ":"4194304"}]}
{"group":18,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
    },{"cb_nodes":"12"},{"striping_factor":"12"},{"
    striping_unit":"1048576"}]}
{"group":19,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
    },{"cb_nodes":"12"},{"striping_factor":"12"},{"
    striping_unit":"1048576"}]}
{"group":20,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
    },{"cb_nodes":"12"},{"striping_factor":"12"},{"
```

```
        striping_unit":"1048576"}]}
13 {"group":25,"mpi_info":[{"cb_buffer_size":"16777216"},{"
        romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
        },{"cb_nodes":"12"},{"striping_factor":"12"},{"
        striping_unit":"1048576"}]}
   {"group":24,"mpi_info":[{"cb_buffer_size":"16777216"},{"
        romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
        },{"cb_nodes":"12"},{"striping_factor":"12"},{"
        striping_unit":"2097152"}]}
15 {"group":23,"mpi_info":[{"cb_buffer_size":"16777216"},{"
        romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
        },{"cb_nodes":"16"},{"striping_factor":"16"},{"
        striping_unit":"2097152"}]}
   {"group":5,"mpi_info":[{"cb_buffer_size":"16777216"},{"
        romio_cb_read":"automatic"},{"romio_cb_write":"enable"},{"
        cb_nodes":"4"},{"striping_factor":"4"},{"striping_unit":"
        1048576"}]}
17 {"group":9,"mpi_info":[{"cb_buffer_size":"16777216"},{"
        romio_cb_read":"automatic"},{"romio_cb_write":"enable"},{"
        cb_nodes":"4"},{"striping_factor":"4"},{"striping_unit":"
        1048576"}]}
   {"group":11,"mpi_info":[{"cb_buffer_size":"16777216"},{"
        romio_cb_read":"automatic"},{"romio_cb_write":"enable"},{"
        cb_nodes":"4"},{"striping_factor":"4"},{"striping_unit":"
        1048576"}]}
19 {"group":12,"mpi_info":[{"cb_buffer_size":"16777216"},{"
        romio_cb_read":"automatic"},{"romio_cb_write":"enable"},{"
        cb_nodes":"4"},{"striping_factor":"4"},{"striping_unit":"
        1048576"}]}
   {"group":13,"mpi_info":[{"cb_buffer_size":"16777216"},{"
        romio_cb_read":"automatic"},{"romio_cb_write":"enable"},{"
        cb_nodes":"4"},{"striping_factor":"4"},{"striping_unit":"
        1048576"}]}
21 {"group":7,"mpi_info":[{"cb_buffer_size":"16777216"},{"
        romio_cb_read":"automatic"},{"romio_cb_write":"enable"},{"
        cb_nodes":"4"},{"striping_factor":"4"},{"striping_unit":"
        1048576"}]}
   {"group":14,"mpi_info":[{"cb_buffer_size":"16777216"},{"
        romio_cb_read":"automatic"},{"romio_cb_write":"enable"},{"
        cb_nodes":"4"},{"striping_factor":"4"},{"striping_unit":"
        1048576"}]}
```

```
23 {"group":16,"mpi_info":[{"cb_buffer_size":"16777216"},{"
     romio_cb_read":"automatic"},{"romio_cb_write":"enable"},{"
     cb_nodes":"4"},{"striping_factor":"4"},{"striping_unit":"
     1048576"}]}
   {"group":10,"mpi_info":[{"cb_buffer_size":"16777216"},{"
     romio_cb_read":"automatic"},{"romio_cb_write":"enable"},{"
     cb_nodes":"4"},{"striping_factor":"4"},{"striping_unit":"
     8388608"}]}
25 {"group":3,"mpi_info":[{"cb_buffer_size":"16777216"},{"
     romio_cb_read":"automatic"},{"romio_cb_write":"enable"},{"
     cb_nodes":"4"},{"striping_factor":"4"},{"striping_unit":"
     8388608"}]}
   {"group":17,"mpi_info":[{"cb_buffer_size":"16777216"},{"
     romio_cb_read":"automatic"},{"romio_cb_write":"enable"},{"
     cb_nodes":"8"},{"striping_factor":"8"},{"striping_unit":"
     1048576"}]}
27 {"group":27,"mpi_info":[{"cb_buffer_size":"16777216"},{"
     romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
     },{"cb_nodes":"12"},{"striping_factor":"12"},{"
     striping_unit":"8388608"}]}
   {"group":33,"mpi_info":[{"cb_buffer_size":"16777216"},{"
     romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
     },{"cb_nodes":"16"},{"striping_factor":"16"},{"
     striping_unit":"1048576"}]}
29 {"group":35,"mpi_info":[{"cb_buffer_size":"16777216"},{"
     romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
     },{"cb_nodes":"16"},{"striping_factor":"16"},{"
     striping_unit":"1048576"}]}
   {"group":36,"mpi_info":[{"cb_buffer_size":"16777216"},{"
     romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
     },{"cb_nodes":"16"},{"striping_factor":"16"},{"
     striping_unit":"1048576"}]}
31 {"group":26,"mpi_info":[{"cb_buffer_size":"16777216"},{"
     romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
     },{"cb_nodes":"16"},{"striping_factor":"16"},{"
     striping_unit":"1048576"}]}
   {"group":37,"mpi_info":[{"cb_buffer_size":"16777216"},{"
     romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
     },{"cb_nodes":"16"},{"striping_factor":"16"},{"
     striping_unit":"1048576"}]}
33 {"group":38,"mpi_info":[{"cb_buffer_size":"16777216"},{"
```

```
    romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
    },{"cb_nodes":"16"},{"striping_factor":"16"},{"
    striping_unit":"1048576"}]}
   {"group":39,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
    },{"cb_nodes":"16"},{"striping_factor":"16"},{"
    striping_unit":"2097152"}]}
35 {"group":32,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
    },{"cb_nodes":"16"},{"striping_factor":"16"},{"
    striping_unit":"2097152"}]}
   {"group":29,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
    },{"cb_nodes":"16"},{"striping_factor":"16"},{"
    striping_unit":"2097152"}]}
37 {"group":31,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
    },{"cb_nodes":"16"},{"striping_factor":"16"},{"
    striping_unit":"2097152"}]}
   {"group":34,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
    },{"cb_nodes":"16"},{"striping_factor":"16"},{"
    striping_unit":"2097152"}]}
39 {"group":43,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
    },{"cb_nodes":"16"},{"striping_factor":"16"},{"
    striping_unit":"8388608"}]}
   {"group":30,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"enable"},{"
    cb_nodes":"16"},{"striping_factor":"16"},{"striping_unit":
    "1048576"}]}
41 {"group":42,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"enable"},{"
    cb_nodes":"16"},{"striping_factor":"16"},{"striping_unit":
    "1048576"}]}
   {"group":40,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"enable"},{"
    cb_nodes":"16"},{"striping_factor":"16"},{"striping_unit":
    "1048576"}]}
43 {"group":28,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"enable"},{"
```

```
    cb_nodes":"16"},{"striping_factor":"16"},{"striping_unit":
    "2097152"}]}
  {"group":41,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"enable"},{"
    cb_nodes":"16"},{"striping_factor":"16"},{"striping_unit":
    "2097152"}]}
45 {"group":45,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
    },{"cb_nodes":"16"},{"striping_factor":"16"},{"
    striping_unit":"1048576"}]}
  {"group":44,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
    },{"cb_nodes":"16"},{"striping_factor":"16"},{"
    striping_unit":"2097152"}]}
47 {"group":47,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
    },{"cb_nodes":"16"},{"striping_factor":"16"},{"
    striping_unit":"4194304"}]}
  {"group":46,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"automatic"
    },{"cb_nodes":"16"},{"striping_factor":"16"},{"
    striping_unit":"8388608"}]}
49 {"group":48,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"disable"},{
    "cb_nodes":"16"},{"striping_factor":"16"},{"striping_unit"
    :"1048576"}]}
  {"group":49,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"disable"},{
    "cb_nodes":"8"},{"striping_factor":"8"},{"striping_unit":"
    4194304"}]}
51 {"group":50,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"disable"},{
    "cb_nodes":"16"},{"striping_factor":"16"},{"striping_unit"
    :"1048576"}]}
  {"group":51,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"disable"},{
    "cb_nodes":"16"},{"striping_factor":"16"},{"striping_unit"
    :"2097152"}]}
53 {"group":0,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"disable"},{
    "cb_nodes":"12"},{"striping_factor":"12"},{"striping_unit"
```

```
    :"4194304"}]}
{"group":53,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"disable"},{
    "cb_nodes":"12"},{"striping_factor":"12"},{"striping_unit"
    :"4194304"}]}
55 {"group":52,"mpi_info":[{"cb_buffer_size":"16777216"},{"
    romio_cb_read":"automatic"},{"romio_cb_write":"disable"},{
    "cb_nodes":"12"},{"striping_factor":"12"},{"striping_unit"
    :"8388608"}]}
```


Listing A.3: A configuration file example.

Bibliography

[1] X. Ma, X. Ma. 'I/O'. In: *Encyclopedia of Parallel Computing*. Ed. by D. Padua. Boston, MA: Springer US, 2011, pp. 975–984. URL: https://doi.org/10.1007/978-0-387-09766-4_290 (cit. on p. 1).

[2] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, K. Riley. '24/7 Characterization of petascale I/O workloads'. In: *2009 IEEE International Conference on Cluster Computing and Workshops*. 2009, pp. 1–10 (cit. on p. 1).

[3] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, Y. Yao. 'A Multiplatform Study of I/O Behavior on Petascale Supercomputers'. In: *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '15. Portland, Oregon, USA: Association for Computing Machinery, 2015, pp. 33–44. URL: https://doi.org/10.1145/2749246.2749269 (cit. on p. 1).

[4] M. Folk, A. Cheng, K. Yates. 'HDF5: A file format and I/O library for high performance computing applications'. In: *Proceedings of Supercomputing*. Vol. 99. 1999, pp. 5–33 (cit. on p. 1).

[5] M. P. I. Forum. *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015*. High-Performance Computing Center Stuttgart, University of Stuttgart, 2015. URL: https://books.google.de/books?id=Fbv7jwEACAAJ (cit. on pp. 1, 19–21, 33).

[6] URL: https://www2.opengroup.org/ogsys/catalog/c165 (cit. on p. 1).

[7] P. Schwan. 'Lustre: Building a File System for 1,000-node Clusters'. In: (Jan. 2003) (cit. on p. 1).

[8] X. Wang. 'A light weighted semi-automatically I/O-tuning solution for engineering applications'. PhD thesis (cit. on pp. 2, 3, 5, 15, 17, 20, 31–34, 36, 40, 66, 79).

[9] B. Behzad, S. Byna, S. M. Wild, M. Prabhat, M. Snir. 'Improving Parallel I/O Autotuning with Performance Modeling'. In: *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '14. Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 253–256. URL: https://doi.org/10.1145/2600212.2600708 (cit. on pp. 2, 3, 12).

[10] M. Agarwal, D. Singhvi, P. Malakar, S. Byna. 'Active Learning-based Automatic Tuning and Prediction of Parallel I/O Performance'. In: *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*. 2019, pp. 20–29 (cit. on pp. 2, 3, 6, 12, 66).

[11] H. Luu, B. Behzad, R. Aydt, M. Winslett. 'A multi-level approach for understanding I/O activity in HPC applications'. In: *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. 2013, pp. 1–5 (cit. on pp. 3, 12, 66).

[12] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, R. Vuduc. 'Autotuning in High-Performance Computing Applications'. In: *Proceedings of the IEEE* 106.11 (2018), pp. 2068–2083 (cit. on p. 3).

[13] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, N. J. Wright. 'Modular HPC I/O Characterization with Darshan'. In: *2016 5th Workshop on Extreme-Scale Programming Tools (ESPT)*. 2016, pp. 9–17 (cit. on p. 4).

[14] S. Snyder, P. Carns, K. Harms, R. Latham, R. Ross. 'Performance Evaluation of Darshan 3.0.0 on the Cray XC30'. In: (Apr. 2016). URL: https://www.osti.gov/biblio/1250469 (cit. on p. 4).

[15] J. M. Kunkel, M. Zimmer, N. Hübbe, A. Aguilera, H. Mickler, X. Wang, A. Chut, T. Bönisch, J. Lüttgau, R. Michel, et al. 'The SIOX Architecture – Coupling Automatic Monitoring and Optimization of Parallel I/O'. In: *Lecture Notes in Computer Science Supercomputing* (2014), pp. 245–260 (cit. on pp. 4, 5).

[16] B. Behzad, S. Byna, M. Prabhat, M. Snir. 'Pattern-driven parallel I/O tuning'. In: Nov. 2015, pp. 43–48 (cit. on p. 5).

[17] S. Madireddy, P. Balaprakash, P. Carns, R. Latham, R. Ross, S. Snyder, S. Wild. 'Machine Learning Based Parallel I/O Predictive Modeling: A Case Study on Lustre File Systems'. In: Jan. 2018, pp. 184–204 (cit. on pp. 6, 66).

[18] B. Behzad, S. Byna, S. M. Wild, M. Prabhat, M. Snir. 'Improving Parallel I/O Autotuning with Performance Modeling'. In: *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '14. Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 253–256. URL: https://doi.org/10.1145/2600212.2600708 (cit. on p. 6).

[19] F. Isaila, P. Balaprakash, S. M. Wild, D. Kimpe, R. Latham, R. Ross, P. Hovland. 'Collective I/O Tuning Using Analytical and Machine Learning Models'. In: *2015 IEEE International Conference on Cluster Computing*. 2015, pp. 128–137 (cit. on pp. 6, 66).

[20] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, M. Snir. 'Taming Parallel I/O Complexity with Auto-Tuning'. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado: Association for Computing Machinery, 2013. URL: https://doi.org/10.1145/2503210.2503278 (cit. on p. 6).

[21] B. Behzad, S. Byna, Prabhat, M. Snir. 'Optimizing I/O Performance of HPC Applications with Autotuning'. In: *ACM Trans. Parallel Comput.* 5.4 (Mar. 2019). URL: https://doi.org/10.1145/3309205 (cit. on pp. 6, 44).

[22] M. Howison, Q. Koziol, D. Knaak, J. Mainzer, J. Shalf. 'Tuning HDF5 for Lustre File Systems'. In: (Jan. 2012) (cit. on p. 6).

[23] R. McKenna, S. Herbein, A. Moody, T. Gamblin, M. Taufer. 'Machine Learning Predictions of Runtime and IO Traffic on High-End Clusters'. In: *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. 2016, pp. 255–258 (cit. on p. 6).

[24] M. R. Wyatt, S. Herbein, T. Gamblin, A. Moody, D. H. Ahn, M. Taufer. 'PRIONN: Predicting Runtime and IO Using Neural Networks'. In: ICPP 2018. Eugene, OR, USA: Association for Computing Machinery, 2018. URL: https://doi.org/10.1145/3225058.3225091 (cit. on p. 6).

[25] S. Madireddy, P. Balaprakash, P. Carns, R. Latham, R. Ross, S. Snyder, S. Wild. 'Machine Learning Based Parallel I/O Predictive Modeling: A Case Study on Lustre File Systems'. In: Jan. 2018, pp. 184–204 (cit. on p. 6).

[26] P. Ivanenko, A. Doroshenko, K. Zhereb. 'TuningGenie: Auto-Tuning Framework Based on Rewriting Rules'. In: vol. 469. June 2014, pp. 139–158 (cit. on p. 7).

[27] A. Doroshenko, P. Ivanenko, O. Novak, O. Yatsenko. 'A Mixed Method of Parallel Software Auto-Tuning Using Statistical Modeling and Machine Learning'. In: *Information and Communication Technologies in Education, Research, and Industrial Applications Communications in Computer and Information Science* (2019), pp. 102–123 (cit. on p. 7).

[28] A. Doroshenko, P. Ivanenko, O. Novak, O. Yatsenko. 'Parallel software auto-tuning using statistical modeling and machine learning'. In: *Problems In Programming* 2-3 (2018), pp. 046–053 (cit. on p. 7).

[29] A. Bağbaba. 'Improving Collective I/O Performance with Machine Learning Supported Auto-tuning'. In: *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2020, pp. 814–821 (cit. on p. 8).

[30] A. Bağbaba, X. Wang, C. Niethammer, J. Gracia. 'Improving the I/O Performance of Applications with Predictive Modeling based Auto-tuning'. In: *2021 International Conference on Engineering and Emerging Technologies (ICEET)*. 2021, pp. 1–6 (cit. on p. 8).

[31] A. Bağbaba, X. Wang. 'Improving the MPI-IO Performance of Applications with Genetic Algorithm based Auto-tuning'. In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2021, pp. 798–805 (cit. on p. 8).

[32] C. Niethammer, S. Becker, M. Bernreuther, M. Buchholz, W. Eckhardt, A. Heinecke, S. Werth, H.-J. Bungartz, C. Glass, H. Hasse, J. Vrabec, M. Horsch. 'ls1 mardyn: The Massively Parallel Molecular Dynamics Code for Large Systems'. In: *Journal of Chemical Theory and Computation* 10 (Aug. 2014) (cit. on pp. 12, 84).

[33] URL: https://wiki.old.lustre.org/lid/ulfi/complete/ulfi_complete.html (cit. on p. 14).

[34] V. O. Rybintsev. 'Optimizing the parameters of the Lustre-file-system-based HPC system for reverse time migration'. In: *The Journal of Supercomputing* 76.1 (Jan. 2020), pp. 536–548. URL: https://doi.org/10.1007/s11227-019-03054-7 (cit. on p. 14).

[35] F. Schmuck, R. Haskin. 'GPFS: A Shared-Disk File System for Large Computing Clusters'. In: *Proceedings of the 1st USENIX Conference on File and Storage Technologies*. FAST '02. Monterey, CA: USENIX Association, 2002, 19–es (cit. on p. 16).

[36] M. Soysal, M. Berghoff, T. Zirwes, M.-A. Vef, S. Oeste, A. Brinkmann, W. E. Nagel, A. Streit. 'Using On-Demand File Systems in HPC Environments'. In: *2019 International Conference on High Performance Computing Simulation (HPCS)*. 2019, pp. 390–398 (cit. on pp. 16, 17).

[37] *Fraunhofer Parallel File System – BeeGFS - Fraunhofer ITWM*. June 2021. URL: https://www.itwm.fraunhofer.de/en/departments/hpc/fraunhofer-parallel-file-system-beegfs.html (cit. on pp. 17, 18).

[38] R. Thakur, W. Gropp. 'Parallel I/O'. In: 2003 (cit. on pp. 18, 19, 21, 33).

[39] A. Ching, A. Choudhary, K. Coloma, W.-k. Liao, R. Ross, W. Gropp. 'Non-contiguous I/O accesses through MPI-IO'. In: *CCGrid 2003. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2003. Proceedings.* 2003, pp. 104–111 (cit. on pp. 18, 26).

[40] R. Thakur, W. Gropp, E. Lusk. 'Data sieving and collective I/O in ROMIO'. In: Mar. 1999, pp. 182–189 (cit. on pp. 18, 27).

[41] K. Coloma, A. Choudhary, W. Liao, L. Ward, E. Russell, N. Pundit. 'Scalable high-level caching for parallel I/O'. In: *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* 2004, pp. 96– (cit. on p. 18).

[42] H. Shan, K. Antypas, J. Shalf. 'Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark'. In: *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. 2008, pp. 1–12 (cit. on p. 18).

[43] Y. Chen, X.-H. Sun, R. Thakur, H. Song, H. Jin. 'Improving Parallel I/O Performance with Data Layout Awareness'. In: *2010 IEEE International Conference on Cluster Computing*. 2010, pp. 302–311 (cit. on p. 18).

[44] S. J. Kim, Y. Zhang, S. W. Son, M. Kandemir, W.-k. Liao, R. Thakur, A. Choudhary. 'IOPro: a parallel I/O profiling and visualization framework for high-performance storage systems'. In: *The Journal of Supercomputing* 71.3 (Mar. 2015), pp. 840–870. URL: https://doi.org/10.1007/s11227-014-1329-0 (cit. on pp. 18, 19).

[45] S. El Sayed, M. Bolten, D. Pleiter. 'Parallel I/O Architecture Modelling Based on File System Counters'. In: *High Performance Computing*. Ed. by M. Taufer, B. Mohr, J. M. Kunkel. Cham: Springer International Publishing, 2016, pp. 627–637 (cit. on p. 18).

[46] E. Smirni, D. Reed. 'Lessons from characterizing the input/output behavior of parallel scientific applications'. In: *Performance Evaluation* 33.1 (1998). Tools for Performance Evaluation, pp. 27–44. URL: https://www.sciencedirect.com/science/article/pii/S0166531698000091 (cit. on p. 21).

[47] R. Thakur, W. Gropp, E. Lusk. 'Optimizing noncontiguous accesses in MPI–IO'. In: *Parallel Computing* 28 (Aug. 2001), pp. 83–105 (cit. on pp. 21, 22, 32, 33).

[48] R. Thakur, W. Gropp, E. Lusk. 'A Case for Using MPI's Derived Datatypes to Improve I/O Performance'. In: *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. 1998, pp. 1–1 (cit. on pp. 22, 23).

[49] C. Jin, S. Sehrish, W.-k. Liao, A. Choudhary, K. Schuchardt. *Improving the Average Response Time in Collective I/O* (cit. on p. 26).

[50] R. Thakur, W. Gropp, E. Lusk. 'On Implementing MPI-IO Portably and with High Performance'. In: *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*. IOPADS '99. Atlanta, Georgia, USA: Association for Computing Machinery, 1999, pp. 23–32. URL: https://doi.org/10.1145/301816.301826 (cit. on pp. 27, 31, 32).

[51] M. Chaarawi, E. Gabriel, R. Keller, R. Graham, G. Bosilca, J. Dongarra. 'OMPIO: A modular software architecture for MPI I/O'. In: vol. 6960. Sept. 2011, pp. 81–89 (cit. on pp. 27, 28).

[52] A. Bagbaba. 'A Comparative Study of MPI-IO Libraries for Offloading of Collective I/O Tasks'. In: *2021 International Conference on Engineering and Emerging Technologies (ICEET)*. 2021, pp. 1–6 (cit. on p. 29).

[53] A. Chiusole, S. Cozzini, D. van der Ster, M. Lamanna, G. Giuliani. 'An I/O Analysis of HPC Workloads on CephFS and Lustre'. In: *High Performance Computing*. Ed. by M. Weiland, G. Juckeland, S. Alam, H. Jagode. Cham: Springer International Publishing, 2019, pp. 300–316 (cit. on p. 33).

[54] H. Son, G. Lee, K. Kang, Y.-J. Kang, B. D. Youn, I. Lee, Y. Noh. 'Industrial issues and solutions to statistical model improvement: a case study of an automobile steering column'. In: *Structural and Multidisciplinary Optimization* 61 (Apr. 2020) (cit. on p. 44).

[55] W. Roetzel, X. Luo, D. Chen. 'Optimal design of heat exchanger networks'. In: Jan. 2020, pp. 231–317 (cit. on p. 44).

[56] Llnl. *LLNL/ior: Parallel filesystem I/O benchmark*. URL: http://github.com/LLNL/ior (cit. on pp. 48, 65).

[57] URL: https://www.mcs.anl.gov/research/projects/pio-benchmark (cit. on pp. 48, 66).

[58] T. S. Tamir, G. Xiong, Z. Li, H. Tao, Z. Shen, B. Hu, H. M. Menkir. 'Traffic Congestion Prediction using Decision Tree, Logistic Regression and Neural Networks'. In: *IFAC-PapersOnLine* 53.5 (2020). 3rd IFAC Workshop on Cyber-Physical & Human Systems CPHS 2020, pp. 512–517. URL: https://www.sciencedirect.com/science/article/pii/S2405896321002627 (cit. on p. 57).

[59] E. Dumitrescu, S. Hué, C. Hurlin, S. Tokpavi. 'Machine learning for credit scoring: Improving logistic regression with non-linear decision-tree effects'. In: *European Journal of Operational Research* (2021). URL: https://www.sciencedirect.com/science/article/pii/S0377221721005695 (cit. on p. 57).

[60] I. H. Sarker. 'Machine Learning: Algorithms, Real-World Applications and Research Directions'. In: *SN Computer Science* 2.3 (Mar. 2021), p. 160. URL: https://doi.org/10.1007/s42979-021-00592-x (cit. on p. 57).

[61] L. Breiman. 'Random Forests'. In: *Machine Learning* 45.1 (Oct. 2001), pp. 5–32. URL: https://doi.org/10.1023/A:1010933404324 (cit. on p. 59).

[62] S. Benedict, R. Rejitha, P. Gschwandtner, R. Prodan, T. Fahringer. 'Energy Prediction of OpenMP Applications Using Random Forest Modeling Approach'. In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. 2015, pp. 1251–1260 (cit. on p. 59).