Ollscoil na Gaillimhe
University of Galway

Provided by the author(s) and University of Galway in accordance with publisher policies. Please cite the published version when available.

Title	HPC IO and seismic data performance optimization using ANNs prediction based auto-tuning
Author(s)	Tipu, Abdul Jabbar Saeed
Publication Date	2023-06-19
Publisher	NUI Galway
Item record	http://hdl.handle.net/10379/17823

Downloaded 2024-05-18T12:38:35Z

Some rights reserved. For more information, please see the item record link above.

HPC IO and Seismic Data Performance Optimization using ANNs Prediction based Auto-tuning

By
Abdul Jabbar Saeed Tipu
17235774

Supervisor
Dr. Enda Howley
School of Computer Science, University of Galway

A thesis submitted as the fulfilment of the requirements for the degree of PhD. in Computer Science

In
School of Computer Science,
University of Galway,
Galway, Ireland.

(May 30, 2023)

Abstract

HPC or super-computing clusters are designed for executing computationally intensive operations that typically involve large scale IO operations. This most commonly involves using a standard MPI library implemented in C/C++.

The MPI-IO performance in HPC clusters tends to vary significantly over a range of configuration parameters that are generally not considered by the algorithm. It is commonly left to individual practitioners to optimise IO on a case-by-case basis at code level. This can often lead to a range of unforeseen outcomes, specifically when it comes to the manual tuning of the configuration parameters.

The ExSeisDat utility is built on top of the native MPI-IO library comprising of Parallel IO and Workflow Libraries to process seismic data encapsulated in SEG-Y file format. The SEG-Y File data structure is complex in nature, due to the alternative arrangement of trace header and trace data. Its size scales to petabytes and the chances of IO performance degradation are further increased by ExSeisDat.

The aim of this research is to auto-tune the Parallel IO configurations which determine the gain in bandwidth performance, without requiring the user programmer's intervention. This research thesis presents a novel study of the changing IO performance in terms of bandwidth, with the use of parallel plots against various MPI-IO, Lustre (Parallel) File System and SEG-Y File configuration parameters.

Another novel aspect of this research is the predictive modelling of parallel (MPI) IO, and ExSeisDat's SEG-Y IO and file sorting bandwidth performance behaviour using Artificial Neural Networks (ANNs). In continuation to this, the auto-tuning parameters strategy is designed, which is based on the ANN models predictions. This is an innovative approach to optimize Parallel IO bandwidth performance for MPI-IO, seismic (SEG-Y) data IO and file sorting operations.

The results presented in the thesis, show significant performance gain through statistical analysis, in the Parallel IO bandwidth within the HPC cluster. Additionally, this research has highlighted the common most useful configuration settings which give the highest probability of improving the IO performance, in the event ML models are unavailable for bandwidth predictions.

Furthermore, the different ANN hidden layer node configurations have also been discussed with respect to identified SEG-Y operations, for separate and combined READ-/WRITE executions, which show maximum average bandwidth performances.

List of Publications

This is an article-based thesis and following is the list of published articles in the journals:

1. Tipu, A.J.S., Conbhuí, P.Ó. and Howley, E., 2022. Applying neural networks to predict HPC-IO bandwidth over seismic data on lustre file system for ExSeisDat. *Cluster Computing*, 25(4), pp.2661-2682.

2. Tipu, A.J.S., Conbhuí, P.Ó. and Howley, E., 2022. Artificial Neural Networks based Predictions towards the Auto-tuning and Optimization of Parallel IO Bandwidth in HPC System. *Cluster Computing*, pp.1-20.

3. Tipu, A.J.S., Conbhuí, P.Ó. and Howley, E., 2022. Seismic data IO and sorting optimization in HPC through ANNs prediction based auto-tuning for ExSeisDat. *Neural Computing and Applications*, pp.1-34.

Acknowledgment

Up and above everything all glory to **ALMIGHTY ALLAH**. The Beneficent, The most Merciful and Most Compassionate. It's a great blessing from Almighty Allah that gives me the health and strength to do this research work.

I am sincerely grateful to my loving parents and family members for their consistent prayers, wishes and moral support throughout this journey.

I would specially like to thank my **Supervisor** Dr. Enda Howley for supporting me during the challenges faced throughout the stages of this Structured PhD Programme and pandemic as well!

I am also immensely thankful to my **Irish Centre for High-End Computing (ICHEC) team**: Prof. Jean Christophe Desplat, Dr. Pádraig Ó Conbhuí and all my colleagues for giving me the opportunity to work under the umbrella of an industrial project as part of my PhD. They, supported and provided me with the HPC facility, tools to carry out my research, also resolving issues which arouse on the technical side during this period and an excellent platform for nurturing and building my personal and professional development.

Abdul Jabbar Saeed Tipu

Funding Information

ExSeisDat is designed by Irish Centre for High-End Computing (ICHEC) in partnership with Tullow Oil plc and DataDirect Networks (DDN). ExSeisDat and this PhD research thesis is supported, in part, by Science Foundation Ireland (SFI) grant 13/RC/2094 and co-funded under the European Regional Development Fund through the Southern & Eastern Regional Operational Programme to Lero - the Irish Software Research Centre (www.lero.ie), and further supported by the Higher Education Authority (HEA), Ireland.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Research Questions 4
1.3 Thesis Contributions 4
1.4 Thesis Structure 5

2 High Performance Computing & Parallel IO **7**
2.1 Introduction 7
2.2 HPC Parallel Programming Memory Models 9
2.2.1 Distributed Memory Model 9
2.2.2 Shared Memory Model 11
2.2.3 Hybrid Memory Model 12
2.2.4 GPU Programming Model 14
2.3 HPC Parallel Storage 18
2.3.1 Parallel IO 18
2.3.2 Parallel File System 22
2.3.3 Previous Related Work 26
2.4 Summary 30

3 Seismic Data & ExSeisDat **32**
3.1 Introduction 32
3.2 Seismic Data Format in SEG-Y File 32
3.3 ExSeisDat 35
3.3.1 Motivation 35
3.3.2 Basic Design Concept 35
3.3.3 ExSeisPIOL 36
3.3.4 ExSeisFlow 37
3.3.5 Overview of ExSeisDat Benchmarking Experimental Setup 39
3.3.6 ExSeisDat Benchmarking Results 40
3.4 Summary 41

4 Machine Learning **45**
4.1 Introduction 45
4.2 Data Preparation 47
4.2.1 Identification of Input and Output Feature Parameters 48
4.2.2 Data generation 49
4.2.3 Data Scaling 50
4.2.4 Data Shuffling 51

4.2.5 Data Splitting . . . 53
4.3 Supervised vs. Unsupervised Learning . . . 54
4.3.1 Supervised Learning . . . 54
4.3.2 Unsupervised Learning . . . 56
4.4 Model Training . . . 57
4.4.1 Linear Regression (`for Regression problems`) . . . 57
4.4.2 Logistic Regression (`for Classification problems`) . . . 60
4.4.3 Multi-class Classification . . . 63
4.4.4 Regularization - a solution to Overfitting . . . 63
4.5 Artificial Neural Network . . . 65
4.5.1 Forward Propagation Pass . . . 67
4.5.2 Backward Propagation Pass . . . 68
4.6 Model Testing . . . 71
4.6.1 Regression Model Testing . . . 71
4.6.2 Classification Model Testing . . . 72
4.7 Recent ML Research for Optimizing HPC IO . . . 74
4.8 Summary . . . 76

5 Applying Neural Networks to predict HPC-IO bandwidth over seismic data on Lustre File System for ExSeisDat 77
5.1 Introduction . . . 77
5.2 Related Work . . . 78
5.3 Design and Implementation . . . 80
5.3.1 Research Methodology . . . 80
5.3.2 Key Parameters Identified . . . 80
5.3.3 Development of Benchmarks . . . 81
5.3.4 Development of ML based ANN models . . . 83
5.3.5 Prediction Accuracy Evaluation of ANN Models . . . 86
5.4 Experimental Result Analysis . . . 87
5.4.1 Benchmarks Results . . . 88
5.4.2 Prediction Results Analysis . . . 95
5.5 Discussion . . . 98
5.6 Conclusion . . . 100

6 Artificial Neural Networks based Predictions towards the Auto-tuning and Optimization of Parallel IO Bandwidth in HPC System 101
6.1 Introduction . . . 101
6.2 Background and Related Research . . . 103
6.3 Experimental Design and Implementation . . . 104
6.3.1 Identification of Key and Tunable Parameters . . . 104
6.3.2 Generating MPI-IO Benchmarks Results Data . . . 106
6.3.3 Creation and Development of ANN models . . . 107
6.3.4 Complete Auto-tuning Design Applied to Test Cases . . . 111
6.4 Experimental Results and Evaluation . . . 116
6.4.1 Runtime Complexity Analysis of ANNs Predictions . . . 117
6.4.2 READ Auto-tuning and Common Configurations Analysis . . . 118
6.4.3 WRITE Auto-tuning and Common Configurations Analysis . . . 120
6.5 Summary of the Work . . . 122

6.6 Conclusion 123

7 Seismic Data IO and Sorting Optimization in HPC through ANNs Prediction based Auto-tuning for ExSeisDat 125

7.1 Introduction 125
7.2 Related Work 127
7.3 Design and Implementation 128
7.3.1 Research Methodology 128
7.3.2 Key Parameters Identified 128
7.3.3 Generating SEG-Y IO and file sorting Benchmarks Results Data . 130
7.3.4 Predictive IO bandwidth Modelling with ANNs 133
7.3.5 Full Design Strategy for Auto-tuning Parameters 134
7.4 Experimental Results Analysis 142
7.4.1 ANNs Performance Analysis 144
7.4.2 Auto-tuning Results Analysis 152
7.5 Discussion 157
7.6 Conclusion 158

8 Summary of Thesis 166

9 Conclusion & Future work 169

List of Abbreviations

Abbreviations	Descriptions
ExSeisDat	Extreme-Scale Seismic Data Library
HPC	High Performance Computing
MPI	Message Passing Interface
IO	Input Output
ROMIO	A High-Performance, Portable MPI-IO Implementation
SEG-Y	File format for storing seismic data
PFS	Parallel File System
LFS	Lustre File System
PVFS	Parallel Virtual File System
GPFS	General Parallel File System
HDFS	Hadoop Distributed File System
HDF	Hierarchical Data Format
NHI	Network Hardware Infrastructure
PCIe	Peripheral Component Interconnect Express
SSD	Solid State Drive
HDD	Hard Disk Drive
MDS	Meta Data Server
MDT	Meta Data Target
OSS	Object Storage Server
OST	Object Storage Target
ML	Machine Learning
ANN	Artificial Neural Networks
MSE	Mean Square Error
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error
NASA	National Aeronautics and Space Administration
DDN	DataDirect Networks
IME	Infinite Memory Engine
CRS	Coordinates Reference System
DMS	Degrees-Minutes-Seconds

List of Figures

2.1 The basic view of HPC cluster system. 8
2.2 The Distributed Memory Model Overview. 9
2.3 MPI Example - Sum of integers in array. 10
2.4 The Shared Memory Model Overview. 11
2.5 OpenMP Example - Sum of integers in array. 13
2.6 The Hybrid Memory Model Overview. 14
2.7 Hybrid Model Example - Sum of integers in array. 15
2.8 The GPU Memory Model Overview. 16
2.9 Parallel IO. 19
2.10 View of IO Stack in HPC cluster for handling parallel IO requests. 19
2.11 Data Sieving in parallel read or write operation. 20
2.12 Aggregation READ. 21
2.13 Aggregation WRITE. 21
2.14 Two Phase Collective IO. 22
2.15 Lustre Architecture. 23
2.16 Lustre command to list available MDTs and OSTs. 24
2.17 Lustre command to get File Distribution Pattern. 25
2.18 File Striping. 25
2.19 File Striping with stripe count 4 and stripe size 2MB. 26
2.20 File Striping with stripe count 4 and stripe size 8MB. 26
2.21 File Striping with stripe count 2 and stripe size 4MB. 27
2.22 Data aligning at Lustre File System level. 28

3.1 IO Stack involving ExSeisDat. 33
3.2 Structure of SEG-Y file format. Reprinted from "ExSeisDat: A set of parallel I/O and workflow libraries for petroleum seismology" by Fisher, M.A., Conbhuí, P.Ó., Brion, C.Ó., Acquaviva, J.T., Delaney, S., O'brien, G.S., Dagg, S., Coomer, J. and Short, R., 2018, *Oil & Gas Science and Technology–Revue d'IFP Energies nouvelles, 73*, p.74 [1]. 33
3.3 LFS runtime and LFS+IME runtime. Reprinted from "ExSeisDat: A set of parallel I/O and workflow libraries for petroleum seismology" by Fisher, M.A., Conbhuí, P.Ó., Brion, C.Ó., Acquaviva, J.T., Delaney, S., O'brien, G.S., Dagg, S., Coomer, J. and Short, R., 2018, *Oil & Gas Science and Technology–Revue d'IFP Energies nouvelles, 73*, p.74 [1]. 41

3.4 Throughput comparison between LFS and IME reads/writes. Reprinted from "ExSeisDat: A set of parallel I/O and workflow libraries for petroleum seismology" by Fisher, M.A., Conbhuí, P.Ó., Brion, C.Ó., Acquaviva, J.T., Delaney, S., O'brien, G.S., Dagg, S., Coomer, J. and Short, R., 2018, *Oil & Gas Science and Technology–Revue d'IFP Energies nouvelles, 73*, p.74 [1]. 42
3.5 Fraction of runtime spent on IO and SEG-Y computations for LFS and LFS-supported IME. Reprinted from "ExSeisDat: A set of parallel I/O and workflow libraries for petroleum seismology" by Fisher, M.A., Conbhuí, P.Ó., Brion, C.Ó., Acquaviva, J.T., Delaney, S., O'brien, G.S., Dagg, S., Coomer, J. and Short, R., 2018, *Oil & Gas Science and Technology–Revue d'IFP Energies nouvelles, 73*, p.74 [1]. 43

4.1 Basic Machine Learning Process steps. 46
4.2 k-Folds Cross-Validation. 54
4.3 Basic Structures of ANNs. 66

5.1 Research Flow 80
5.2 Hierarchy of parameter values nested loops generating all possible configurations to benchmark. 84
5.3 Benchmarks Results Graphs. 89
5.4 Data-aligning strategy showing Low IO bandwidths. **Note:-** stripe_size is represented in the units of Mega Bytes (MBs), chunk_size and file_size are represented in the units of Bytes. 90
5.5 High bandwidth configurations settings for SEG-Y File READ operations. **Note:-** stripe_size is represented in the units of Mega Bytes (MBs). 92
5.6 High bandwidth configurations settings for SEG-Y File WRITE operations. **Note:-** stripe_size is represented in the units of Mega Bytes (MBs). 93
5.7 High bandwidth configurations settings for SEG-Y file sorting. **Note:-** stripe_size is represented in the units of Mega Bytes (MBs). 94
5.8 Predictions Results using Trained ANN Models. 97

6.1 Research Methodology of the Experimental Setup to train the ANN ML models and auto-tune the configuration parameters settings based on the IO bandwidth predictions. 105
6.2 MPI-IO Benchmarks Results. 108
6.3 MPI-IO Bandwidth Predictions. 109
6.4 READ Improvements. 119
6.5 WRITE Improvements. 121

7.1 Research Flow for Auto-tuning SEG-Y operations. 129
7.2 SEG-Y Benchmarks Results. 133
7.3 SEG-Y IO ANNs Prediction Accuracy. 145
7.4 SEG-Y IO READ Predictions. 146
7.5 SEG-Y IO WRITE Predictions. 147
7.6 SEG-Y file sorting ANNs Prediction Accuracy. 148
7.7 SEG-Y Sorting Contiguous READ Predictions. 149
7.8 SEG-Y Sorting Contiguous WRITE Predictions. 150
7.9 SEG-Y IO READ Improvements. 153

7.10 SEG-Y IO WRITE Improvements. 161
7.11 SEG-Y file sorting Improvements via `contiguous` READ. 163
7.12 SEG-Y file sorting Improvements via `contiguous` WRITE. 165

List of Tables

3.1 ExSeisDat Benchmarking Settings. . . . 39

4.1 Example - Estimation of House Prices. . . . 48
4.2 Example - Forecasting Temperature on Hourly basis. . . . 49
4.3 Example - Estimation of CPU time and IO time. . . . 49
4.4 Sample 1 of shuffled House Prices data by rows. . . . 52
4.5 Sample 2 of shuffled House Prices data by fraction. . . . 53

5.1 Configuration Parameters and their values. . . . 81
5.2 ANN's Description Table. . . . 84
5.3 ANN's Hyper-parameters. . . . 85
5.4 Errors during Training and Testing. . . . 95
5.5 Accuracy of applied ANN models. . . . 96
5.6 Accuracy breakdown for MPI-IO benchmarks prediction models. . . . 98
5.7 Accuracy breakdown for SEG-Y IO benchmarks prediction models. . . . 98
5.8 Accuracy breakdown for SEG-Y Sorting benchmarks prediction models. . 99

6.1 Tunable/Non-tunable Parameters with possible values settings. . . . 106
6.2 ANNs Description table to Model READ/WRITE bandwidth behaviour. 109
6.3 ANNs Input and Output Nodes Mapping to Configuration and Output Parameters for READ and WRITE Bandwidth Predictions. . . . 109
6.4 Hyper parameters applied to ANNs. . . . 110
6.5 Default Configurations Test Cases for Auto-tuning. . . . 116
6.6 Matrices Description and Memory Usage During ANNs Feed Forward Propagation Pass. . . . 118
6.7 ANNs Feed Forward Propagation Pass to predict bandwidth. . . . 118
6.8 READ Auto-tuning Improvement Results. . . . 119
6.9 Common READ Auto-Tuned Configurations. Note: - **c_s** denotes chunk size for each MPI process and **f_a_p** denotes file access pattern. . . . 120
6.10 WRITE Auto-tuning Improvement Results. . . . 121
6.11 Common WRITE Auto-Tuned Configurations. Note: - **s_c** denotes lustre stripe count, **s_s** denotes lustre stripe size, **c_s** denotes chunk size for each MPI process and **f_a_p** denotes file access pattern. . . . 122

7.1 SEG-Y IO Benchmarks Configuration Parameters and their values. . . . 130
7.2 SEG-Y file sorting Benchmarks Configuration Parameters and their values. 130
7.3 Training and Testing Sets Partitions from Benchmarks Results. . . . 134
7.4 ANNs Structure Configuration. . . . 135

7.5 Mapping of ANNs Input/Output nodes to Configuration Parameters for SEG-Y IO and File Sort Bandwidth Modelling. 135
7.6 Hyper-parameters for SEG-Y IO and file sorting prediction models. . . . 135
7.7 Default SEG-Y IO Configurations Test Cases to Auto-tune. 142
7.8 Default SEG-Y Sorting Configurations Test Cases to Auto-tune. 144
7.9 SEG-Y IO Prediction Accuracy values. 146
7.10 SEG-Y file sorting Prediction Accuracy values. 148
7.11 Matrices Detail and Memory Space in a Feed Forward Propagation Pass of an ANN. 151
7.12 Sub-passes of Feed Forward Propagation Pass for a bandwidth prediction. 151
7.13 Description of Statistical Metrics used. 159
7.14 SEG-Y READ Improvements. 160
7.15 SEG-Y WRITE Improvements. 160
7.16 Combined SEG-Y IO Improvements. 162
7.17 SEG-Y file sorting with `contiguous` READ Improvements. 162
7.18 SEG-Y file sorting with `contiguous` WRITE Improvements. 164
7.19 Combined SEG-Y file sorting Improvements. 164

Chapter 1

Introduction

1.1 Motivation

Seismic data is one of the most critical factors for geophysicists to study and understand the earth structure beneath its surface or seabed. The critical importance of seismic data is not only restricted to understanding the earth's sub-surface but also when earthquakes might jeopardise human life [2]. The Oil and Gas industry also extensively relies on the seismic data as a critical factor for processing and visualizing the sub-structure of the earth or seabed before oil/gas exploration [3]. The seismic data is normally stored and encapsulated in SEG-Y format files which is a global standard [4]. The SEG-Y file format is a complex structure layout used to store the seismic data in a file. It contains trace headers and actual traces data on alternate positions in the file. Usually the SEG-Y or seismic data scales to petabytes when written to files on disks. Therefore, it increases the need of the high performance computing (HPC) or super-computing system facilities to perform large scale IO processing.

Modern HPC clusters are normally designed to perform exascale operations in a time efficient manner. They usually contain a high number of computing nodes and parallel storage disks connected via fast network hardware infrastructure (NHI) [5, 6]. The basic HPC cluster is normally composed of many compute nodes and storage disks connected over Network Hardware Infrastructure (NHI) fabric. At software side a programming paradigm is also required to exploit the clusters potential by writing and executing parallel applications, which in this case is the standard Message Passing Interface (MPI) library to serve the purpose [7]. The library also contains the application programming interface (API) for carrying out parallel IO processing tasks by running parallel processes communicating with multiple storage disks. These disks are normally controlled and managed by a parallel file system (PFS) protocol. In this research, the Lustre File System (LFS) is the PFS running on the targeted machine disks [8].

The LFS is a file system specifically designed for Linux based clusters for large-scale computing. The PFS or LFS are responsible for distributing and storing the file data across the multiple disks which are connected over the network. The file distribution pattern is normally decided and executed at this file system layer. This is based on the number of disks on which the file is stored, and stripe unit which is a splitting unit to write file chunks across the disks. The purpose of this LFS functionality is to support the parallel IO processing and enhance the bandwidth performance during the application execution. However, this comes with overheads which are discussed later in detail.

This thesis addresses the problem of poor IO performance within the parallel applica-

tions, which leads to the overall program performance degradation. One of the reasons is with respect to advancement in storage hardware which has been slower over the decades as compared to compute hardware. Therefore, the computational efficiency of large-scale applications has been immensely increased however, the IO overheads are still the main cause of overall poor application performance. Consequently, the IO performance improvement is usually left to researchers and experts to devise any methods, to overcome the performance issues. The second reason for performance issues from the software side is that in the HPC system the parallel IO is usually optimized by manually tuning the configuration settings related to MPI-IO, LFS and file properties. However, the performance outcome of manual tuning, is subject to case-to-case situations. Manual tuning of the related configuration settings, on application's each execution run, is often ignored from the user programmer end as they pose an extra overhead in the productivity. If user considers to manually tune the settings, the previous research studies suggest the data-alignment strategy to apply as the configuration setting [9, 10, 11, 12]. This sets the number of processes equal to number of disks (stripe count) and each process file chunk size to READ/WRITE, equal to stripe unit size. Despite the data-alignment is useful in several cases to optimize HPC IO performance however, it is still not promising for all types of cases or applications, as explained later.

To process the SEG-Y files data, the Extreme-Scale Seismic Data (ExSeisDat) library is already developed based on the existing MPI-IO APIs, to further develop and execute its application on the HPC system [1, 13]. The ExSeisDat contains its own parallel IO library, namely PIOL, and Workflow library to perform seismic data related functionalities. Despite this parallel processing library, it faces a degradation in program execution performance. The reason is multiple MPI processes get restricted by the LFS protocol when they try to access a disk at a same instance in parallel. This is because the data-alignment is not applied since it is challenging to be configured for the SEG-Y format file, due to having the traces data being arranged at alternate positions in a file rather than being placed consecutively. Furthermore, data-alignment is not applied in ExSeisDat library.

From this scenario, it can be clearly seen that the magnitude of parallel IO bandwidth for MPI-IO operations or ExSeisDat's SEG-Y operations, critically relies on certain configuration parameters related to MPI, LFS, file properties or SEG-Y file properties. Therefore, it is necessary to find and tune the suitable parameters settings that can improve the bandwidth performance from the current existing value settings. This can be only achieved if the right configuration settings that give maximum possible IO bandwidth, are already known to be tuned or applied. This has significantly motivated the aim of this research, to devise an auto-tuning software logic support that does not require user interference to manually tune the configuration settings. It should be noted that the auto-tuning is required to be executed before any IO operation during the application execution. The auto-tuning is supposed to be based on the parallel IO bandwidth prediction of the basic parallel (MPI) IO operations, and the ExSeisDat's seismic data IO and file sorting operations. The execution details of these types of operations, are discussed later in the thesis.

As the aim of this research is to keep the user free from the manual tuning of the parameters therefore, the proposed method of IO optimization is auto-tuning based on the Machine Learning (ML) predictions. The recent studies have shown the notable advantages of predictive IO bandwidth or performance modelling through Machine Learning (ML), for auto-tuning parameters in different HPC problem scenarios [14, 15, 16, 17, 18].

Consequently, the Artificial Neural Network (ANN) is chosen as the ML technique for predictive IO modelling [19, 20]. Some of the recent studies, are the source of motivation in this research to implement ANN based ML models, using the Python PyTorch package due to its significant accuracy in predicting outputs [21, 22, 23, 24]. The scope of this research covers the parallel IO and seismic data operations as stated earlier, for their optimization based on auto-tuning using the ANNs ML predictions. Once the ANNs are trained, they are validated over the test set of IO bandwidth profiling benchmarks results against and the related configuration parameters.

This research is divided into three main contributions. The first contribution presents the graph representation of benchmarks data of high bandwidth on parallel plots (HiPlot [25]), and the ANN ML prediction models, for MPI-IO and SEG-Y operations. The error metrics such as: MAE, MSE and MAPE have been used to portray the prediction accuracy of all the models. The second contribution presents the auto-tuning design strategy for MPI-IO operations, statistical analysis of bandwidth performance improvements on test cases, and highlights the most selected configuration settings. Lastly, the third contribution presents the auto-tuning design for SEG-Y IO and SEG-Y file sorting operations, prediction analysis on range of ANNs for each operation type, varying with hidden layers nodes, and the statistical analysis of bandwidth performance improvements on test cases with changing hidden layers nodes of ANNs.

In the third and last contribution of this research, the range of ANNs are tested in the auto-tuning process, with variation in hidden layers nodes, to observe the impact on prediction accuracies and IO bandwidth improvements. The purpose is also to detect the most suitable ANN for the separate READ/WRITE operations in SEG-Y IO and SEG-Y file sorting, and for their combined READ/WRITE operations executions.

This research has been mainly emphasized on the generation of accurate ANN based ML models to predict IO bandwidth beforehand, and auto-tuning the MPI-IO, SEG-Y IO and SEG-Y file sorting operations configurations, using those bandwidth predictions. Then in addition to this, it presents IO bandwidth parallel plots, prediction accuracy analysis, statistical analysis of bandwidth improvements in MPI-IO and SEG-Y operations, highlighted the mostly selected configurations settings for MPI-IO and, impact of changing ANNs on predictions and bandwidth performance improvement data.

As evident from the results in chapter 5, 6 and 7, the ANN models have shown significant prediction accuracy, and have been proven their importance in auto-tuning for optimizing the IO bandwidth performance for MPI-IO and SEG-Y operations.

This thesis follows on with the discssion of high performance computing & parallel IO in chapter 2. Then seismic data & ExSeisDat are explained in chapter 3. Afterwards, machine learning process and some algorithms are elaborated in chapter 4. Later in the thesis, chapter 5 represents the first publication and research contribution about the application of neural networks to predict HPC-IO and seismic data bandwidth. This follows on with the next two research contributions chapters 6 and 7, which explain the auto-tunning approach for optimizing parallel HPC IO and seismic data bandwidth performance, respectively. Then chapter 8 presents summary of thesis. In last, conclusion & future work are discussed in chapter 9.

1.2 Research Questions

The research work conducted in the thesis further extends the existing research in the area of parallel HPC IO bandwidth performance optimization. Specifically, it approaches the IO optimization from learning the pattern of changing bandwidth over configuration parameters to auto-tuning the parameters, in relation to basic parallel MPI IO and seismic data operations. In particular, the following are the research questions as the theme of problems stated and addressed in this thesis:

1. RQ1: What are the configurations that determine high bandwidth for parallel IO and seismic data operations? What is the most suitable technique to accurately predict the changing bandwidth pattern in basic parallel IO, and ExSeisDat's parallel seismic data IO and sorting operations?

2. RQ2: What is the best approach to optimize the basic parallel IO operations performance?

3. RQ3: What approach can be used to optimize the ExSeisDat's seismic data IO and sorting operations performance?

1.3 Thesis Contributions

To address the research questions stated above, this thesis has been mainly comprised of three parts which corresponds to RQ1, RQ2 and RQ3, as the contributions of this work. The following are the research contributions of this thesis, as listed with respect to each research question posed earlier:

1. **Applying Neural Networks to predict HPC-IO bandwidth over seismic data on Lustre File System for ExSeisDat**

   This research corresponds to RQ1, which is the first contribution of this thesis work, presented in chapter 5. In this research, the data-alignment and high bandwidth values against the identified configuration parameters, have been highlighted using the parallel plots, for parallel IO and seismic data operations. Afterwards, the most suitable and accurate bandwidth prediction technique is identified which is the ANN based ML modelling. The IO bandwidth profiling benchmarks data is used for training and testing the models, against the configuration value settings of parameters. The prediction accuracy values are evaluated using the error metrics such as MAE, MSE and MAPE. This research has been published in [26].

2. **Artificial Neural Networks based Optimization of Parallel IO Bandwidth in HPC System**

   This research corresponds to RQ2, which is the second contribution of the thesis and presented in chapter 6. In this research, the auto-tuning design strategy is proposed and explained, to optimize the parallel (MPI) IO operations performance, based on the bandwidth predictions by ANN models, against the identified configuration parameters. The significant percentage of improvement in overall bandwidth performance, is observed, ranging from 65-83% in reading and writing data. Additionally, 3-5 configuration settings are noted to have higher probabilities

in increasing IO performance for disk reads and writes. This research has been published in [27].

3. **Seismic Data IO and Sorting Optimization in HPC through ANNs Prediction based Auto-tuning for ExSeisDat**.

   This research corresponds to RQ3, the third and last contribution of this thesis, which is presented in chapter 7. In this research, the auto-tuning design for parallel seismic data IO and sorting is proposed to optimize the ExSeisDat's application performance based on their ANNs prediction models, against the identified related parameters. The auto-tuning is executed over the range of ANNs having different hidden layers nodes configuration. This shows the impact of increasing ANN hidden nodes on the seismic data bandwidth prediction accuracy and the overall bandwidth improvement. Additionally, the specific ANN configurations are clearly visible that yields to maximum possible bandwidth improvement for each identified seismic data operation. This research has been published in [28].

1.4 Thesis Structure

The structure and layout of work presented in this thesis has been outlined in this section. The first four chapters present the introduction and background topics relevant to this thesis, along with a related literature review. Then next three chapters present the main three primary contributions of this thesis, which correspond to addressing the three research questions, as stated earlier. The IO bandwidth prediction technique discussed in chapter 5, and based on that, the IO optimization by auto-tuning approach elaborated in chapter 6, are together applied to the usecase of seismic data IO and sorting operations. This indicates the usefulness of the novel approach that it can be applied to different domains of applications running on the HPC cluster via MPI standard with a parallel file system based disk IO. Then, in the final two chapters, the work of this research thesis is summarized and concluded. The following is the outlined description of these chapters:

1. chapter 2 presents the background of HPC area and initial research studies in regards to overcoming performance issues of parallel IO within the cluster environment. Mainly, it elaborates the usage of different parallel programming memory models in the HPC framework, and the parallel IO and file system techniques.

2. chapter 3 presents the background of seismic data in SEG-Y file format and its HPC application ExSeisDat. This explains the main features of ExSeisDat library along with the seismic data format structure in SEG-Y file.

3. chapter 4 presents the background study of Machine Learning and its application in various types of applications and research studies to improve HPC IO. This chapter mainly demonstrates the different steps involved in a complete ML process.

4. chapter 5 presents an ANN ML based bandwidth prediction technique for parallel (MPI) IO, and seismic data IO and sorting operations, which corresponds to RQ1.

5. chapter 6 presents the parallel IO optimization approach, based on the bandwidth predictions by ANN based ML models, corresponding to RQ2.

6. chapter 7 presents the seismic data IO and sorting operations optimization, based on the bandwidth predictions by their respective ANN ML models. This corresponds to RQ3.

7. chapter 8 presents the summary of all the research work done in this thesis, from the background study chapters to the research chapters.

8. chapter 9 finally concludes on the critical research and outcomes of this thesis, along with the suggestions for the future work.

Chapter 2

High Performance Computing & Parallel IO

2.1 Introduction

High Performance Computing (HPC) generally refers to the ability of application programs to efficiently execute complex tasks operating on extremely large data volumes. HPC tasks are executed by harnessing the power of many compute, memory and storage resources, in parallel [29]. This is a specific domain of computer science which requires a powerful computing hardware infrastructure usually referred to as clusters or super-computers, rather than the personal general computer systems. HPC cluster or a super-computer can normally comprise of numerous compute nodes having high frequency many-core CPUs, equipped with large size shared and distributed memory, and multiple parallel storage devices. It is also the reason that usually the term HPC is interchangeably used with super-computing or cluster-computing and that it has many sub-forms i.e., distributed computing, parallel computing, GPU programming, etc. Since it involves clusters of high-powered computing resources, the medium connecting them is considered a important component for communication among processes. It is referred to as Network Hardware Infrastructure (NHI), usually involves fast Local Area Network (LAN) cables for high-speed data transfers among processes. The ones most widely used in academia and industry are Intel Omnipath and Infiniband [5, 6]. HPC clusters highly depend on multiple IO storage devices for parallel IO among processes and disks. Therefore, this thesis focuses on addressing the issues in IO performance. Since the cluster servers are used for large scale computations on the data stored in a number of storage devices, they are also connected in parallel with the same network hardware to READ/WRITE data from compute nodes. The Figure 2.1 shows the internal structure of an HPC cluster with multiple compute nodes connected to the multiple storage devices via NHI.

In order to perform computations within the cluster by utilizing the compute resources, different programming models and libraries or frameworks exist [7, 30, 31]. It depends on what level the computations are required to be executed. The basic theme of HPC is to split and distribute the large problem into small problems and execute them in parallel by processes or threads. If the tasks are required to be executed by processes running on a number of compute nodes, then a distributed memory programming model such as the standard Message Passing Interface (MPI) [7, 13], is normally used. Despite this, an MPI program can also be used to run on a single compute node or a general computer system with single or multiple CPU cores. If the tasks are just required to

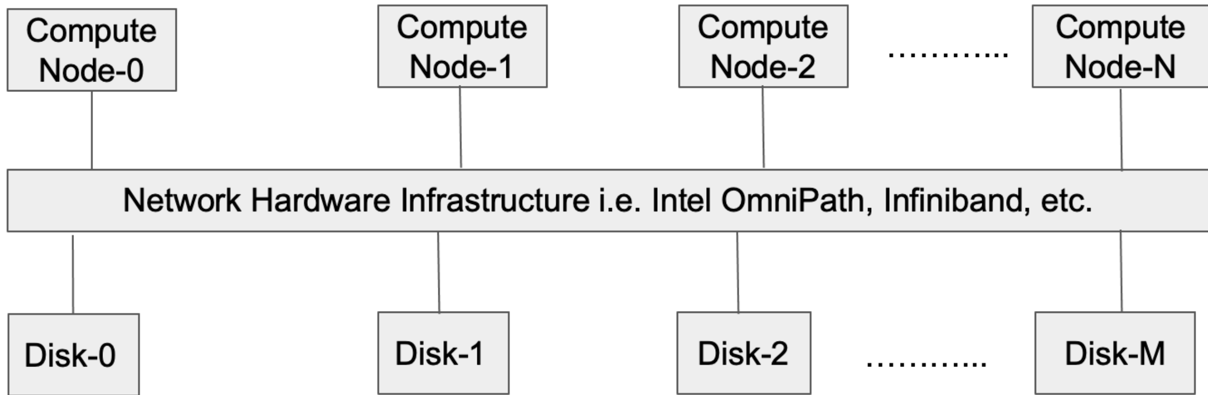


Figure 2.1: The basic view of HPC cluster system.

be executed at any particular compute nodes by threads or CPU cores, then a shared memory programming model is used, such as the OpenMP standard [30]. The hybrid (distributed-shared) programming model can also be used to split the small problem at compute node in further chunks for CPU cores or threads. On a compute node level, GPU programming is also frequently used to process compute intensive graphics related tasks i.e., CUDA [31]. Since modern efficient computer systems come with a specific graphics card for this purpose, it can be integrated in a hybrid model. The CUDA library itself provides a number of Application Programming Interfaces (APIs) with MPI semantics to use distributed memory transfers easily and efficiently.

Apart from these programming frameworks providing a platform for efficient computations, the IO storage processing in clusters play a key role. As the computations are performed on data, it is frequently needed to read/write from hard disks. For achieving high IO throughput in clusters, they are normally provided with a Parallel File System (PFS) to manage multiple parallel networked storage devices i.e., Lustre File System (LFS) [8]. As the parallel running processes may perform IO on their particular chunks of file(s) on disks, data should be distributed and managed among the storage devices connected to the cluster network for an efficient access. The data splitting, striping or distributing of a single or multiple files are common operations of any PFS. The widely used PFSs other than LFS are Parallel Virtual File System (PVFS), General Parallel File System (GPFS), and Hadoop Distributed File system (HDFS) for Hierarchical Data Format (HDF) [32, 33, 34, 35].

The performance issues in IO storage processing are very common challenges for researchers, addressed with different approaches [36, 37]. The IO performance has always been the concern because it has proven to be the main source of overall application performance degradation. The data read/write from disks already wastes a lot of CPU cycles apart from actual computations. The storage side in computer systems has not shown much advancement throughout the evolution of computer systems with the passage of time. An algorithm focused group of programmers, researchers and computer scientists also often ignore making any IO optimizations at the software side. This leads to program performance degradation and IO issues that are left to the researchers of this domain to be addressed with different approaches and techniques. Research studies in the past have produced numerous techniques for optimizing the IO performance. However, they all pose limitations as well, due to the application specific nature of problems.

In this chapter, different forms of HPC programming techniques and memory models

will be discussed in detail. Furthermore, the parallel IO, file system with the perspective of performance issues and past research studies suggesting different approaches to overcome poor IO will also be discussed.

2.2 HPC Parallel Programming Memory Models

2.2.1 Distributed Memory Model

In HPC, the distributed memory programming model enables a user to write a parallel application that can contain multiple end processes executing across the number of cluster compute nodes, as mentioned by Walker [38]. There can be communication among the processes over the network for passing important information or data to each other as a requirement to solve a particular problem.

Figure 2.2 depicts an overview of programming model where compute nodes has their own local memory, connected via network for communication of data among their processes. Similarly, when application code is replicated on the required number of processes, all have the separate copy of the memory variables regardless of being declared as local or global variables. So, a process needs to explicitly send/receive the data to/from other processes. Only in case of storage IO the processes can have concurrent access to the same file. The processes do not require to explicitly pass file data to each other for sharing before any processing.

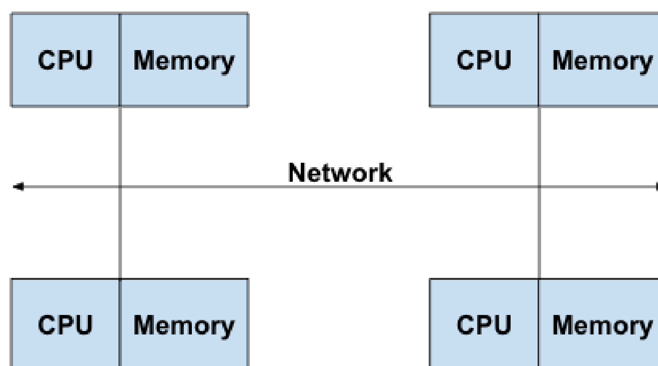


Figure 2.2: The Distributed Memory Model Overview.

The MPI library is a widely used framework for such type of distributed applications, originally implemented in the C language. It is originally supported by the standard C/C++ programming platform. However, it is also available in Java and Python, as mentioned in the works of Baker et al. and Dalcin et al. [39, 40]. Additionally, MPI is integrated with CUDA as well to extend GPU programming to support heterogeneous computing, as elaborated in the work of Awan et al. [41]. The MPI processes are often referred as MPI `ranks` which denote the unique identity for each process. For the understanding of this paradigm, a simple example is the sum of array elements computed by all the processes equally. Figure 2.3 presents the working of this example. As can be seen from the diagram, there are total 4 MPI processes (`P0 - P3`) executing this task. First, the head process `P0` with MPI `rank 0` creates the array of integers of size 12 and splits it into 4 chunks of equal size of 3. Then `P0` keep the first chunk for itself to compute the sum of elements in its array chunk.

The remaining MPI processes first receive their data chunks. Once all processes get their respective data chunks then they compute the local sum of elements in parallel. Then `P1 - P3` having ranks 1 to 3 send their results of sum values back to the `P0` head process. The `P0` process receive the sum values results from the other processes and then finally compute the total sum of all values with its own sum value as a final answer. The `MPI_Send()` and `MPI_Recv()` are the very basic frequently used APIs that provides the functionality of communication among the MPI processes, implemented on top of the sockets interface. MPI also provides the API to apply reduction for summation of all values received at the head process i.e. `MPI_Reduce()` would be suitable for this scenario. The other frequently used APIs are `MPI_Bcast()`, `MPI_Scatter()` and `MPI_Gather()`, which are useful for collectively distributing and gathering data from cluster compute nodes [13]. Ultimately, the MPI library provides a very broad range of APIs to implement distributed memory programming model based parallel applications to execute via exploiting the potential of HPC clusters.

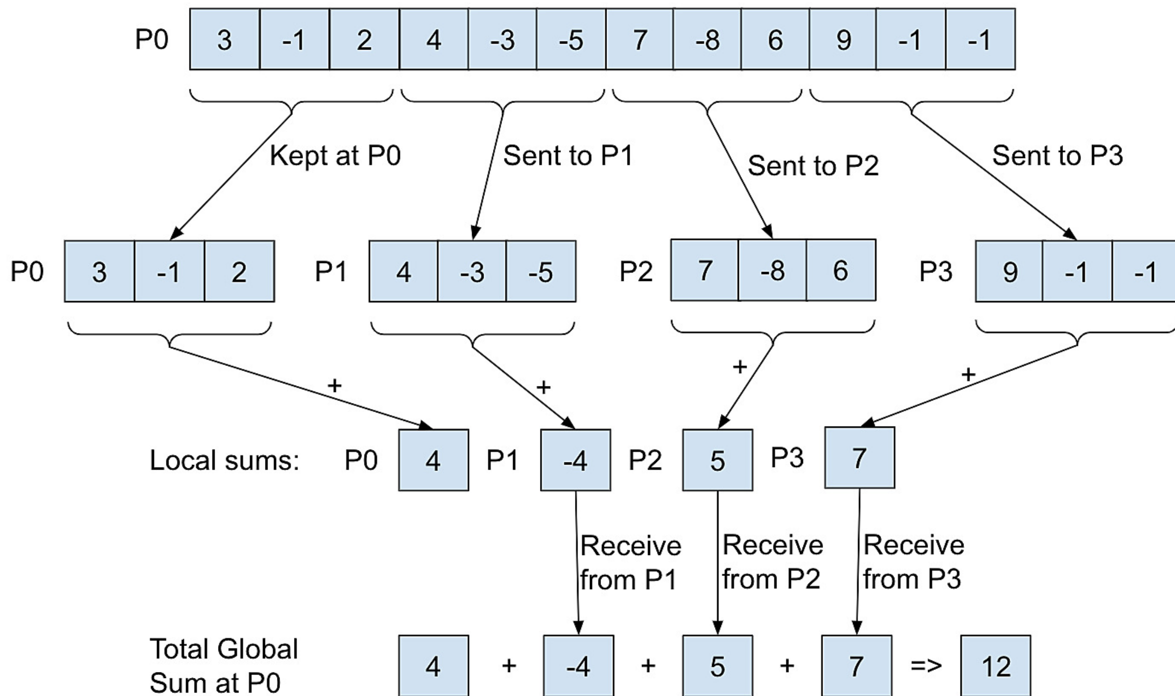


Figure 2.3: MPI Example - Sum of integers in array.

Since there is a necessity of communication among processes and compute nodes involved in this memory model, NHI also plays important role. The efficiency of an application relies on high speed data transfers for communication. If an application is extensively managing the communication or data transfers among the processes, then it might also affect the overall program performance. This is a case where specially NHI is not providing the ideal transfer speed among the multiple compute and storage nodes connected in the cluster. However, as stated earlier, Infiniband and Intel Omnipath are the latest NHIs known for their ideal transfer speed, and they are widely used in modern HPC clusters.

The parallel MPI-IO is another important element of the distributed memory applications. Therefore, the storage hardware and its file system at software side should also be promising in providing maximum possible performance. However, its details and issues are discussed in later section. In short, the performance measurement or time-complexity

of an MPI application must be analyzed from 3 aspects: 1) IO, 2) computations and 3) communication.

MPI applications can also be executed on single compute node as much as multiple MPI processes can be run in parallel on a single node. However, it may limit the performance of the application. In this context, the MPI One-Sided communication and shared memory interface are useful tools to further explore the options for efficiently executing the application. The `MPI_Put()` and `MPI_Get()` APIs are efficient MPI methods to send/receive data efficiently among the processes on one shared memory node. They can directly access the other process local memory to read/write without creating extra buffer which is a requirement in case of `MPI_Send()/MPI_Recev()` APIs. So, this summarizes the basic idea of working with distributed memory programming model.

2.2.2 Shared Memory Model

The shared memory programming model is referred to as the parallel computing form of HPC that normally restricts to one compute node within a cluster, which can be seen from the work of Gibbons et al. [42]. Figure 2.4 depicts this model. The single node is nowadays usually a multi-core or many-cores machine attached to common shared memory i.e., RAM. However, there are other multiple levels or a hierarchy of memory i.e., registers, L1, L2 and L3 caches, which may be shared among CPU cores. The closest one is register memory followed by L1 cache whereas, farthest ones are Random Access Memory (RAM) and hard disk storage device. According to the memory hierarchy the CPU gets data in most efficiently and quickest time from the closest memory i.e., register and then L1 cache. Whereas CPU gets data in slowest time from the farthest devices i.e. RAM and then hard disk storage memory.

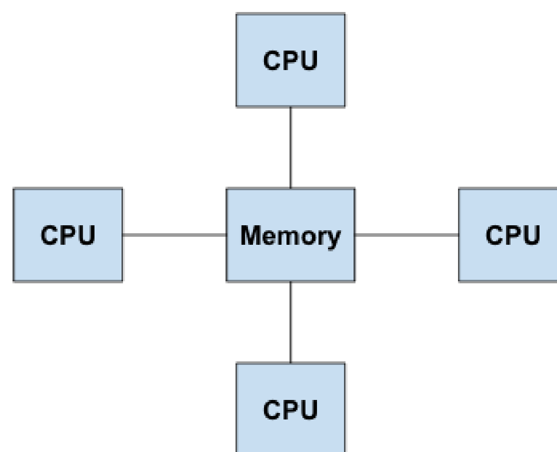


Figure 2.4: The Shared Memory Model Overview.

In this memory model, the problem size or data is split and distributed at the single machine level, among the number of threads running on single or multiple CPU cores. It is normally referred to multi-threading. However, multiple processes on the same machine can also be created to distribute the problem and solve it using shared memory constructs and inter-process communications. As the process creation in parallel requires separate space allocation in memory, it becomes an expensive operation. Whereas the thread creation does not require separate space rather its memory is allocated in the same process space. It is the reason that in the shared memory model the multi-threading is

always preferred over multi-processing, to execute the independent tasks in parallel within an application.

OpenMP is a widely used shared memory framework for C/C++ in the HPC community, which is built on top of the Unix POSIX threads (PThread) library [30, 43]. The OpenMP syntax and semantics directives, and other APIs, enable users to create task or loop parallelism based applications. The simplest form of OpenMP construct is `#pragma omp parallel{...}` which implicitly follows the fork-join flow for task parallelism for thread creation and execution. In addition, it runs the work sharing engine at the back-end for load balancing among running threads via loop parallelism construct `#pragma omp for`. Threads creation in a program space, share and access the same global memory. This is a reason for no explicit need for sending/receiving data to/from threads as the case is in distributed memory model.

Considering the same example of computing the sum of array elements using threads, their respective data chunks can be specified by defining starting and ending boundary indices. The chunks can be distributed among threads with respect to their unique identity number. Figure 2.5 represents the example working in the shared memory programming environment. The array of 12 integer elements is split into 4 chunks for 4 threads (`T0 - T3`). The starting (`s`) and ending (`e`) boundaries defining logic for threads is also mentioned in the diagram. Each thread afterwards computes and update its local sum in the same global variable. Thus, it can generate a data race condition leading to incorrect result.

In order to avoid a data race condition, a line of code can be defined as a critical section by specifying `#pragma omp critical`. This implicitly synchronizes the access of the section such that only one thread can enter it at an instance. There are many other ways to synchronize a particular code section or just over memory and update operation in OpenMP framework. Once all threads are executed synchronously over sum variable, it computes the correct result value for this example.

There are also other libraries and different programming language platforms for shared memory or multi-threading models i.e., Intel Thread Building Blocks (TBB), Java - Threads, Python Threads, etc [44, 39, 45]. In general, shared memory programming or multi-threading optimize application performance. Since there is no communication of data through a network among processes or threads, overall time-complexity or performance has only two components: IO of data and computations. In this model the maximum performance through thread parallelism is when the number of executing threads are equal to number existing physical CPU cores on board. Otherwise, the performance either stays the same or starts deteriorating if there is more than one thread on one CPU core due to concurrent context switching and time slicing. This summarizes the basic idea of parallel shared memory programming environment on a single compute node.

2.2.3 Hybrid Memory Model

In the Hybrid (Distribute-Shared) programming model, the data or tasks are broken down from compute node to CPU core or thread level within the cluster, as elaborated in the work of Pham et al. [46]. Since modern super-computing clusters come with many CPU-cores compute nodes, an application can exhibit both distributed and shared memory programming at the same time. Thus, it increases the complexity of code design logic of such parallel application which is a typical MPI-OpenMP program. However, it can further increase the application parallelism since a task distributed at a particular

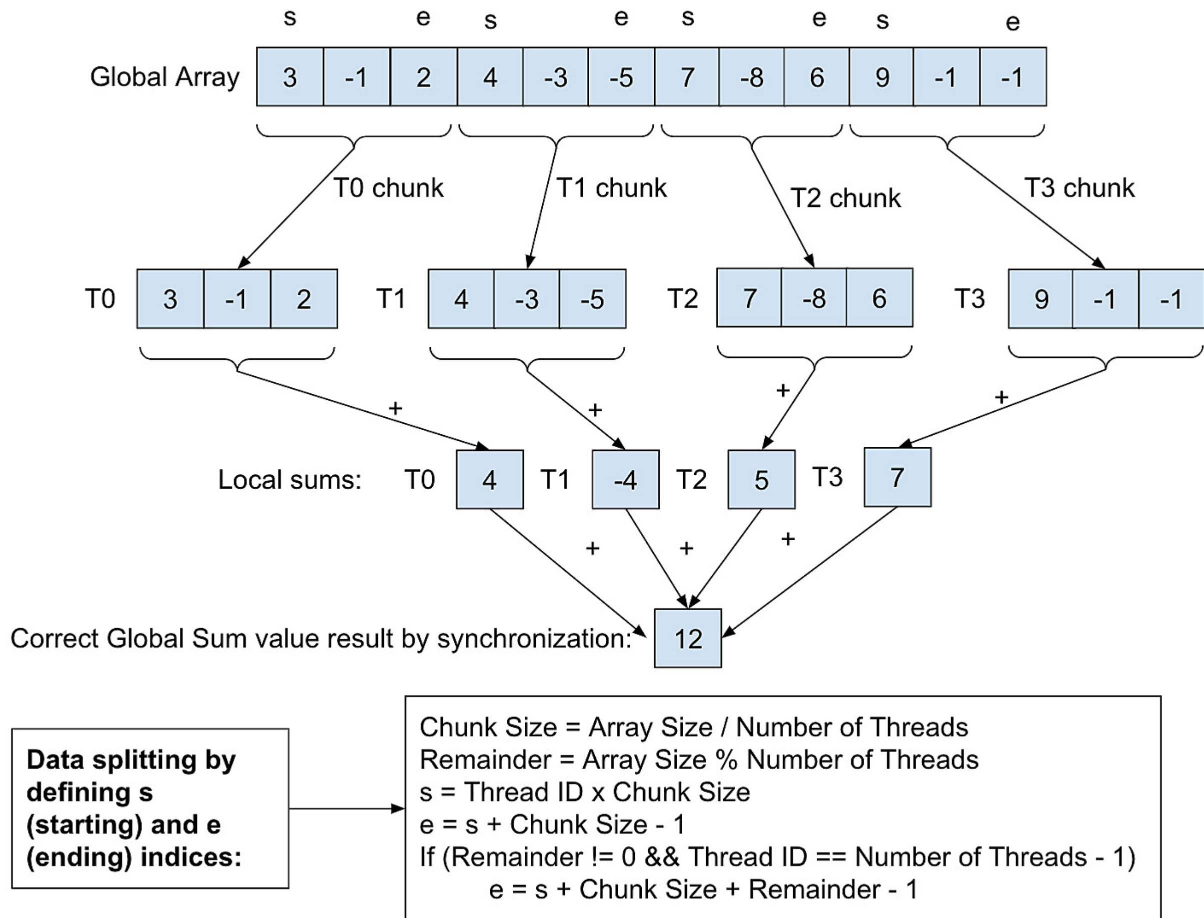


Figure 2.5: OpenMP Example - Sum of integers in array.

compute node is further split or distributed among CPU cores/threads for parallel execution. Figure 2.6 depicts that hybrid model which is a cluster of shared memory model compute nodes connected over the fast network as a distributed memory model.

For designing a hybrid memory model from a programming perspective, the very basic level of code structure of such software should be understood first. By considering the MPI-OpenMP program for instance, the MPI and OpenMP APIs and semantics would be working and carefully integrated together in a single code to execute the parallel tasks. These program generally starts with calling the `MPI_Init()` API to initialize the MPI based distributed memory environment. The number of MPI nodes and MPI processes per node within a communicator group are normally specified at command line, which are then normally accessed by `MPI_Comm_size()` in the program, and `MPI_Comm_rank()` returns the rank or process ID. Then the code logic for distributing the tasks among MPI processes is usually written before moving towards the OpenMP directives. Once the tasks are distributed among MPI processes at code level then the OpenMP directives and APIs can be applied within the `#pragma omp parallel{...}` section. In the scope of this section, the tasks are further split and distributed among CPU threads and executed in parallel.

Consider the same example of computing sum of array elements using this hybrid memory programming model. Figure 2.7 explains the flow of tasks of the problem from distribution from processes nodes to parallel execution at CPU threads. The initial global array is created at P0 MPI rank or head node. Note that the data splitting among

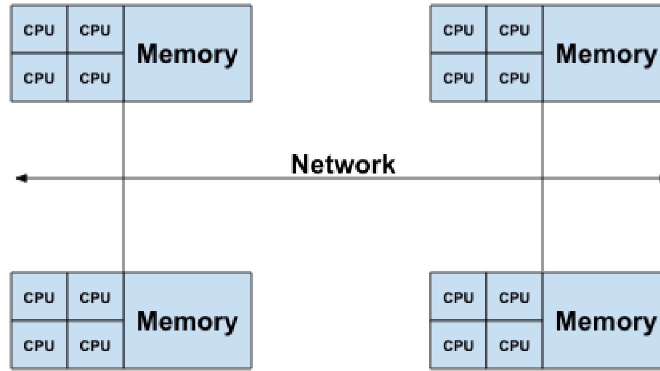


Figure 2.6: The Hybrid Memory Model Overview.

processes is similar to the mechanism for threads, as described at the bottom of the diagrams in the Figures 2.5 and 2.7. The differences are the number of processes and process ID or MPI rank involved in setting the starting and ending indices for each MPI process. The other difference is the particular chunks of the array are explicitly being sent to the specific MPI processes via `MPI_Send()/MPI_Recv()` APIs except P0. Whereas, for threads at each process level, the array data is split only by defining starting and ending indices. It is sent or received among threads as they shared a process or CPU memory.

Once the array is split and distributed among the MPI processes P0, P1 and P3 it is further divided among each process threads T0 and T1. Each thread then computes its sum of array which is further reduced to total sum at process level via OpenMP or any other shared programming API. Then finally all the processes sums are reduced to final global sum at P0 head node process via the `MPI_Reduce()` function call. This whole procedure depicted a very simple example for elaborating the working of a hybrid or distributed-shared memory parallel programming model. The problem size in this example is small in terms of array data, number of processes and number of threads to explain the overall working of hybrid memory model. Increasing the problem size in all means can show how the performance scales in terms of execution time and parallelism. According to the nature of problem, it can be analysed what number of processes and threads ensure maximum possible performance. Increasing the number of processes and threads after a certain threshold may not add up to further performance gains and can lead to deterioration instead of enhancement.

The other possible way to improve or further maximise the hybrid model performance is by using GPU programming at shared memory node level. This is since modern HPC clusters are GPU enabled as they come with dedicated high resolution performance graphics cards. A very big chunk of data can be offloaded to the large number of GPU cores and threads to solve the problem, provided that the tasks can be executed independently. This sums up the explanation of working with the hybrid memory programming model. The next section discusses in detail about the GPU programming model.

2.2.4 GPU Programming Model

The GPU programming was introduced with the invention of dedicated computing units specially designed for processing high-resolution image data [47]. Previously, the native CPU technology was not capable for smooth execution of basic operating system applications and heavy graphical processing simultaneously. Therefore, the separate Graphical

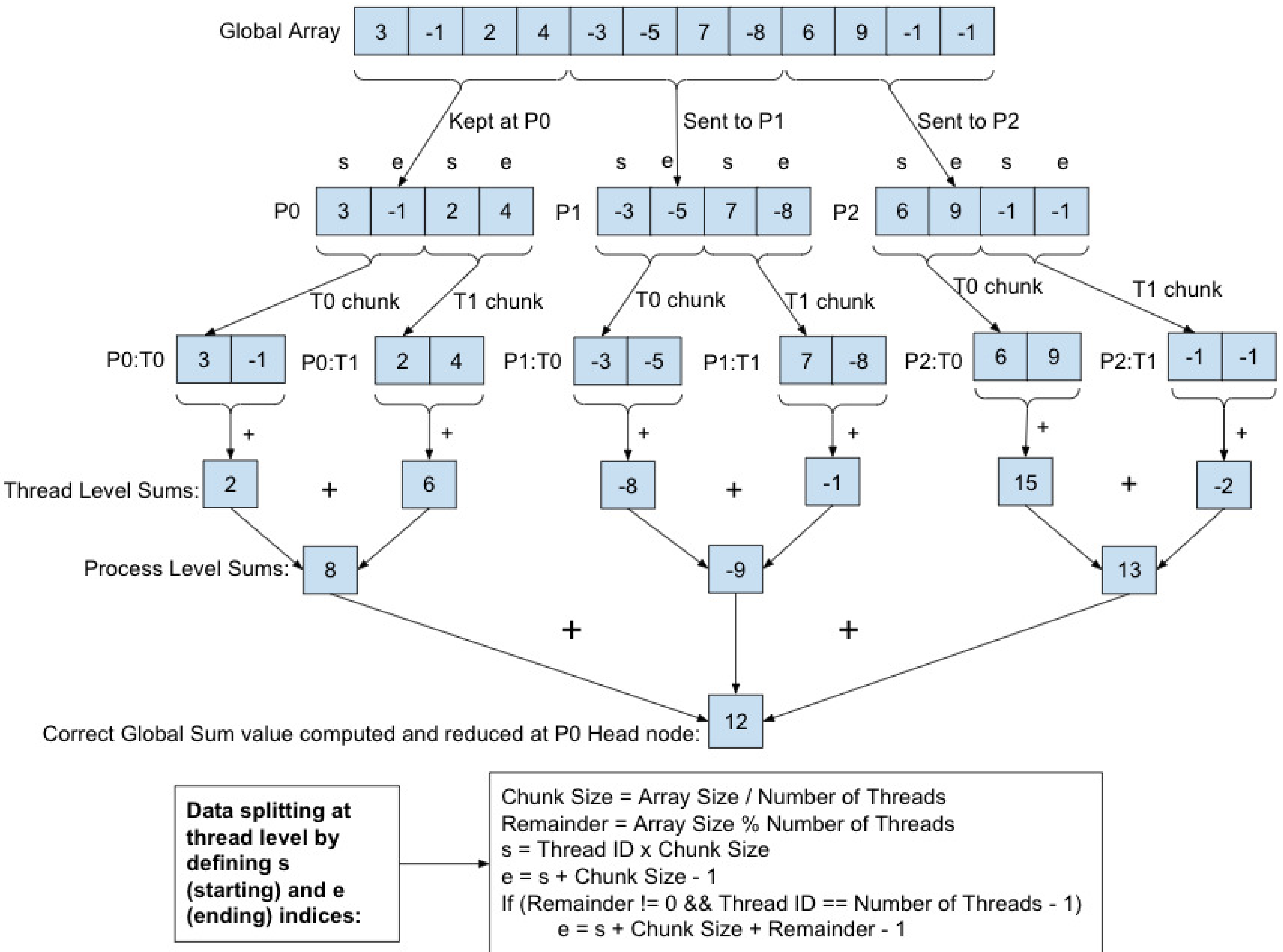


Figure 2.7: Hybrid Model Example - Sum of integers in array.

Processing Units (GPUs) were introduced on board to execute heavy resolution or large sized graphical imagery data tasks separately from the CPUs, i.e NVIDIA, AMD Radeon, etc [48, 31, 49, 50]. In the current era, the image pixels resolution is increasing to be rendered, and becoming computationally complex with the passage of time. Therefore, GPUs are becoming more equipped and advance to process the image data efficiently. However, the use of GPUs is not restricted to image processing or gaming applications but also used for simulations, Artificial Intelligence, Machine/Deep Learning (ML/DL) and other computing techniques.

Figure 2.8 shows the very basic overview of GPU inside and its communication to host CPU via Peripheral Component Interconnect Express (PCIe) bus on the motherboard. The single GPU device can contain numerous grid blocks sharing their common global memory. Each grid block can then contain numerous threads blocks where numerous threads can execute in parallel without pre-emption. This feature of GPUs makes the image or graphics complex problems easy and efficient from the perspective of processing their large sized data. The advancement of GPU cards also relies on these properties by extending them in greater numbers to scale the performance in processing huge graphics data.

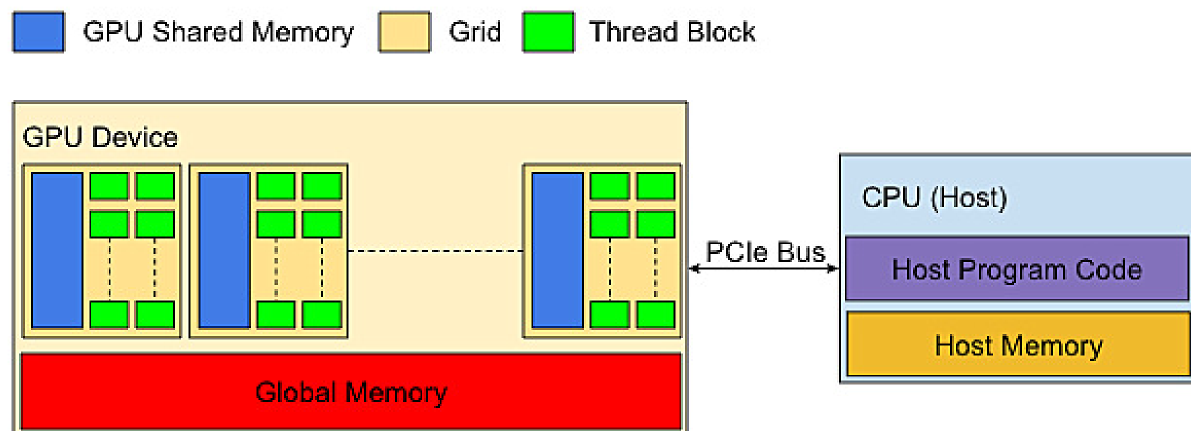


Figure 2.8: The GPU Memory Model Overview.

One GPU card can reside on a single shared memory node of an HPC cluster. Since the modern HPC clusters have many-cores nodes, GPU cards usually ranges from 2 to 4 on one shared memory node. For example, the KAY cluster used in this research has 2x NVIDIA Tesla V100 16GB PCIe (Volta architecture) GPUs on each node of partition of 16 nodes [51]. This has been done to further extend the working and performance of image processing tasks over ample amount of graphics bytes data. This means the GPU programming model can work with both shared memory and distributed memory in hybrid fashion.

In the GPU programming structure, the data first needs to be offloaded from CPU memory to GPU device memory to process it by the GPU threads. Once the result is computed by the GPU it is offloaded back to CPU memory. Considering the NVIDIA's GPU cards platform, it provides CUDA semantics and APIs to perform GPU programming tasks [31]. CUDA works and integrates with C/C++, also supports Java and Python. The `cudamemcpy()` API is used to transfer the data from CPU host memory to GPU device memory after allocation on both ends via `malloc()` and `cudaMalloc()` calls. Then a typical CUDA C/C++ program can contain a user or programmer defined GPU kernel

function that contains the logic of processing data by GPU threads i.e. `__global__ void GPU_kernel(int arg1,float *arg2,...){...}`. This GPU kernel function normally called after memory data transfer in `main()` function, as in the following example:

```
GPU_kernel<<<ThreadBlocksPerGrid,ThreadsPerBlock>>>(arg1, arg2,...);
```

After the complete execution of the GPU kernel function the result data still resides in the GPU memory unless it is transferred back to CPU host memory. For this data transfer, the same `cudamemcpy()` API is used to copy the result data from device to host memory. The typical operations performed at the GPU side are related to complex matrix problems like matrix multiplication, which is frequently executed during the ML training process as well. Therefore, it can involve large data transfers that hinders the performance of a GPU based program. In addition to this the CUDA aware MPI platform enables users to write applications exploiting GPUs power across the number of compute nodes within a HPC cluster [41]. This is conditioned to NVIDIA's GPU cards have enabled GPUDirect RDMA [52]. In this case data can be transferred from GPU to GPU on different machines via distributed memory by directly passing device memory pointers in `MPI_Send()` and `MPI_Recv()` calls.

Considering the example of two extremely large matrices addition which results in a large matrix since, it performs corresponding index-wise addition of all elements. In this scenario, the initial transfer of two source matrices data from CPU to GPU memory and the last transfer of resultant matrix data from GPU to CPU memory would be a major overhead for an application performance. This is despite the efficient execution of the main matrix addition performed by a number of parallel GPU threads.

Because of the issue of large memory transfers, there are some memory optimization techniques developed in the CUDA library. The CUDA Unified Memory is one approach to overcome these performance issues to some extent [53, 54, 55]. The concept is to use one allocated memory at both CPU and GPU sides. The corresponding API for this purpose is `cudaMallocManaged()`. Once the memory is allocated using this function call it can be accessed and modified by both CPU and GPU without any extra memory transfers as previously. Before the CPU accesses same memory after GPU kernel function call, the memory may not be updated yet.

To synchronize the memory access, `cudeDeviceSynchronize()` is useful API to be executed before CPU access, which is mostly recommended by experts. At the back-end during memory access, the CPU or GPU page caches might suffer from page faults upon the unavailability of updated data at the time of requirement. It is also suggested from a development point of view to initialize the memory at the GPU side by defining and calling an extra GPU kernel function to avoid the initial GPU page fault. The other useful approach is prefetching the initialized data into GPU device memory by using the `cudaMemPrefetchAsync()` API call. This usually brings GPU page faults to 0.

While addressing memory optimization, it should be noted that initially, all the required large data for processing usually resides in file on disk. Therefore, it cannot be ignored to discuss techniques for GPU directly accessing the file data on disk. Normally the data must be read from disk into CPU memory before it is brought or prefetched into GPU memory. So, there is always an IO overhead at the GPU side. Since there are some ways and techniques to make data readily available for GPU but eventually, they all passes through CPU memory first which always leaves an overhead.

The NVIDIA's GPUDirect Storage utility can provide the direct path between disk and GPU via PCIe to overcome the IO bottleneck [56]. The `cuFile()` APIs support to

carry out direct GPU based file IO. However, this utility has some system limitations and is not completely functional yet, it is still under development or improvement. The one limitation is that it is only compatible with SSDs NVMe technology, normally resides in Tier-1 storage. It does not work with the Tier-2 storage HDDs. Since SSDs are way faster than HDDs, the data needs to be brought into a Tier-1 storage layer from Tier-2, in order to be acceptable to GPUDirect Storage `cuFile()` functionality.

Apart from these issues, GPUs provide powerful computational performance features. For the same reason, ML/DL problems are often offloaded to GPUs. Since the most expensive and time consuming component is the training of models, executing large and complex matrix operations, GPU plays an important role to train the model way faster as compared to CPU. To keep the GPU based model training simple for ML/DL users, the popular frameworks are PyTorch and Tensorflow on Python platform. These platforms provide API support for utilizing GPUs for ML, by keeping the complex CUDA memory transfer details hidden from the users [22, 57]. These developments have further accelerated the DL based computer vision, image processing and many other problem areas apart from the games and simulations.

For the case study, specifically the significant importance of NVIDIA's GPUs can be imagined as they are playing a vital role for developing a vehicle aimed for 2030s manned mission to Mars planned by NASA [58]. Before the mission can launch, the critical challenge is the safe landing of heavy weight loads on the planet by a manned vehicle which can weigh at least around tens of metric tons. The solution is high-resolution simulations using retropropulsion. This involves complex fluid dynamics for vehicle's tiny turbulence to larger airflow meters away with different landing conditions and other parameters. This type of simulation yields around 100 terabytes of output data. Hundreds of thousands of these simulations will be run to test a range of possible conditions a vehicle could encounter when descending towards the Martian surface. By exploiting the processing power of 3,312 NVIDIA V100 GPUs (each 640 Tensor Cores) using GPUDirect Storage technology for high-bandwidth data transfers, six simulations can be executed at once with NASA's FUN3D computational fluid dynamics software, a work by Bartels [59]. Otherwise, it would take six months on conventional computing platform. This will enable engine designing team to visualize different engine design choices and finding the right adjustments of parameters to finalize design via retropropulsion.

In short, by analyzing the complexities in different problem studies the emergence of GPUs technology into the modern HPC clusters has been significantly proven in scaling the computationally intensive applications performance.

2.3 HPC Parallel Storage

2.3.1 Parallel IO

The IO portion of an application has always been a concern of researchers and scientists as it leads to performance degradation when not taken carefully [36, 37]. If application is consuming large chunk of time in reading the data from disk to memory for processing, then, meanwhile CPU cycles are not running the computations. As a result of this, the CPU cycles are wasted when CPU remains in an idle state, waiting for reading of data in memory, to complete. Afterwards, when the computations of the core tasks are finished, and application is spending another significant time on writing data back to disk then it is wasting additional CPU cycles. This means application is also IO bound alongside

being CPU (or less CPU) bound.

In the parallel programming memory models discussed earlier, the compute tasks are assigned to processes or threads. Similarly, the IO tasks can be assigned to processes or threads to read or write data to disks, in parallel. This parallel reading or writing of data by processes or threads, is normally referred to as parallel IO [60]. Each process or thread is responsible for reading or writing its own defined chunk of data to file on disk as demonstrated in Figure 2.9. This basic technique help reducing IO time as compared to one process or thread is performing read/write operation. Since this research evolves around MPI-IO APIs therefore read/write is normally done by just MPI processes [13].

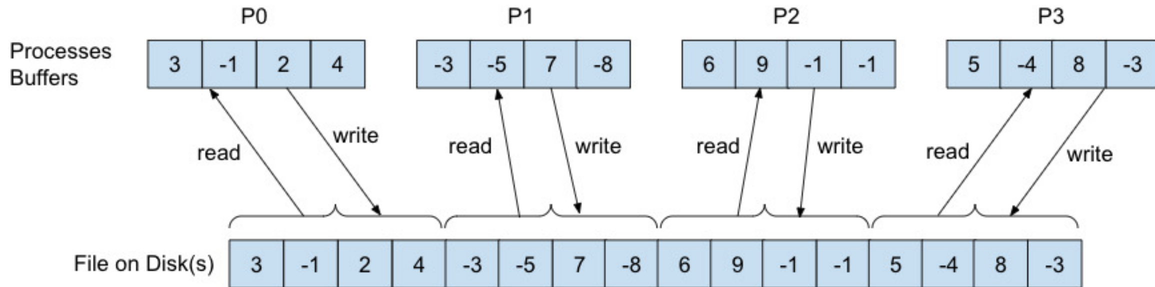


Figure 2.9: Parallel IO.

The parallel application running IO on Kay cluster ([51]) goes through its specific IO stack mentioned in Figure 2.10. It can be understood that an application would be written on top of MPI-IO APIs by means of using them. Then the MPI-IO calls are written on top of Unix based POSIX-IO APIs [61]. When the IO request reaches at file system level to read/write data to/from disks then, it first interfaces with an underlying employed PFS which is LFS in this case. Since LFS is responsible for giving access to processes for reading or writing files bytes across multiple disks, it is at the last level of the IO stack, for managing files on physical disks. This is the overall flow for handling parallel IO requests from the view of IO stack.

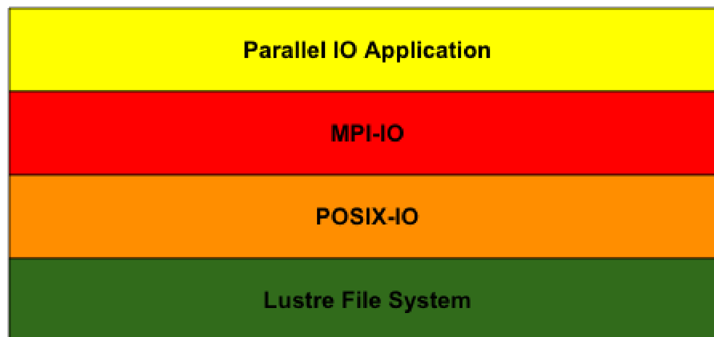


Figure 2.10: View of IO Stack in HPC cluster for handling parallel IO requests.

The MPI-IO APIs are a set of collective and non-collective functionalities for reading/writing data to a common file on disks by multiple MPI processes. Note that at the application side logic, the splitting of file chunks among MPI processes would be same as previously shown by examples in Figures 2.3, 2.5 and 2.7 by threads or processes. Under the hood, in non-collective IO the MPI processes access file chunks independently and read/write contiguously the stream of data bytes. It assumes that the file chunks on disk

for each process are placed contiguously as an ideal case for non-collective IO thus, it works exactly as in Figure 2.9.

On the contrary, in collective IO the MPI processes access file chunks independently but read/write data to file non-contiguously and collectively. This operation is especially useful for file chunks placed on disks non-contiguously for each MPI process. So, at the backend processes might copy data chunks to each other's buffers in order to contain the process requested data as a collective operation. This can be executed using three different types of techniques at the MPI-IO backend: 1) Data sieving, 2) Aggregation and 3) Two phase IO.

Figure 2.11 shows the data sieving techniques at each process end to read or write non-contiguous data to the disks [62, 63]. In case of data sieving within a read operation, the process requests its chunk portion which is non-contiguously placed over a single or multiple disks. Before requested portion reaches to process buffer, the whole chunk of non-contiguous data first fetched into the temporary buffer as the contiguous chunks. Then all the non-contiguous chunks are joined together in the process buffer. The data sieving in read operation, goes exactly same for all other MPI processes trying to access their respective chunks from the disks.

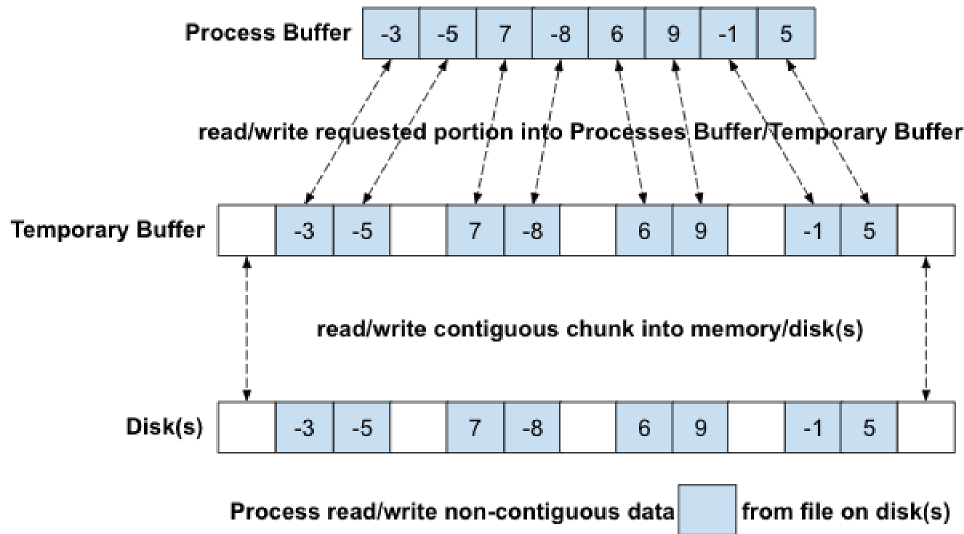


Figure 2.11: Data Sieving in parallel read or write operation.

For the case of data sieving write operation, the process buffer chunk is first written into temporary buffer by splitting into non-contiguous chunks. Then temporary buffer contents are contiguously written onto disks. The white boxes presented in Figure 2.11 in temporary buffer and disks can be the data chunks for other processes. In addition to this, data sieving reduces the number of operations by combining small accesses into a single large request for both read and write operations to improve the latency.

In aggregation technique for collective IO the subset of processes moves the data among processes buffers for reading or writing file chunks on disks as explained in [64]. In read operation, the data chunks for all processes are first moved to their buffers regardless of related data to processes or not. Then the subset of MPI processes called as aggregators move data to the respective right MPI processes buffers including themselves. The Figure 2.12 depicts collective read aggregation technique. This is transparent to process or user as there is no temporary buffer step involved as in previous case of data sieving.

Distinguishably, in write operation, the aggregator processes take their own data and

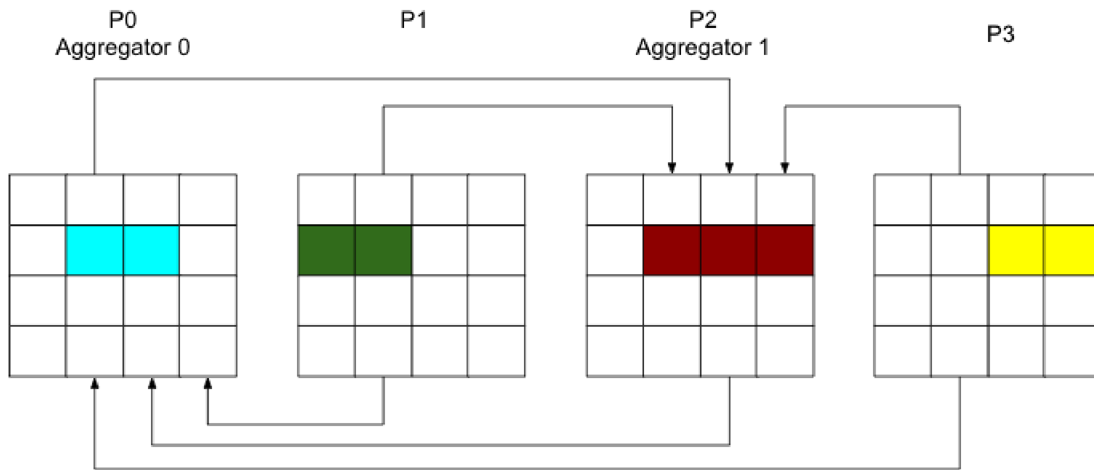


Figure 2.12: Aggregation READ.

other non-aggregator processes data into their buffers and copy the chunks into disks. This technique is depicted in Figure 2.13. It is visible from this figure that it might become possible the good number of chunks for each process end up on the same disk placed contiguously due to the underlying PFS file managing striping policy per unit MB in round-robin fashion, explained later. This can improve read operation in the future if the processes request same data from the disks.

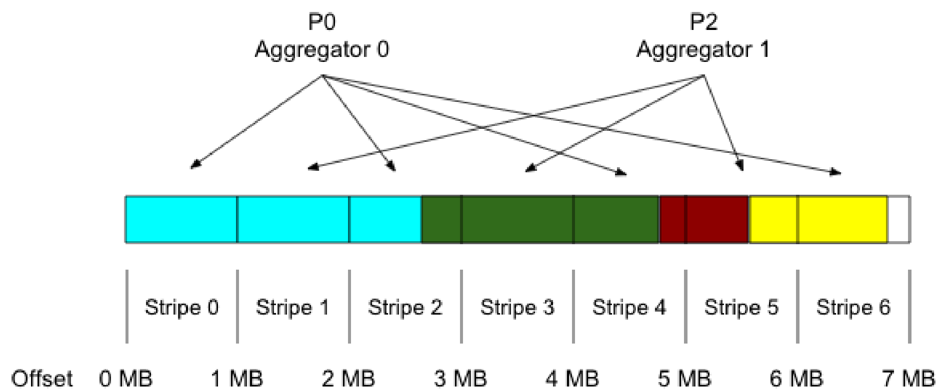


Figure 2.13: Aggregation WRITE.

Since it is known that file data normally is striped or distributed across multiple disks therefore, each process chunk data is also distributed among the parallel disks. In the Two phase collective IO, at the backend, as portrayed in Figure 2.14, each process read small parts of all processes designated chunks into their buffers in first phase [65]. In second phase, those small parts of chunks move to their related processes buffers. For writing data to disks, this flow of two phases will be just opposite in direction as the LFS distribute the chunks stripes across all the disks, such that it would end up in non-contiguous fashion. Considering this scenario, it can be imagined that the IO bandwidth is relying on number of parameters. These parameters can be number of MPI processes, MPI processes per compute node, the file access pattern, and how the file is distributed across parallel storage devices.

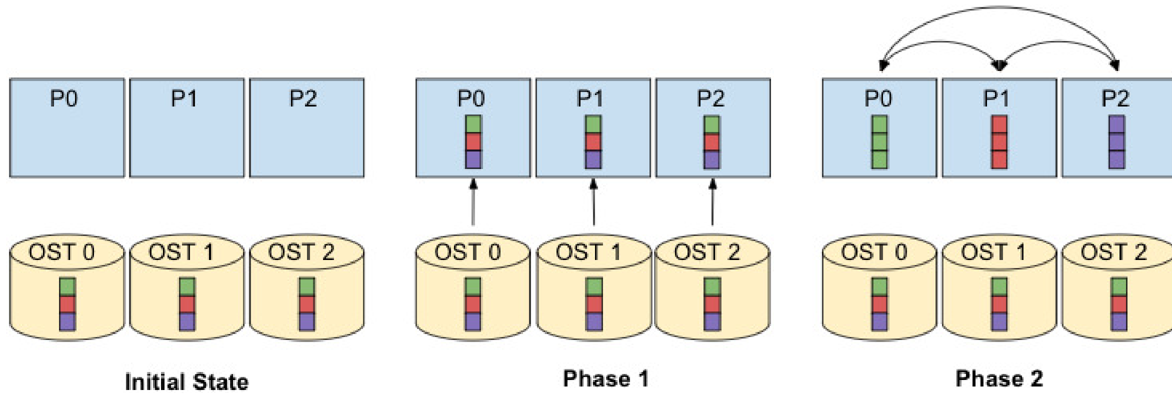


Figure 2.14: Two Phase Collective IO.

2.3.2 Parallel File System

The parallel file systems (PFSs) are usually one of the main software components that manage parallel disks in the HPC clusters storage architecture. The parallel networked storage disks are a common feature of modern HPC clusters for an efficient parallel IO therefore, PFS becomes essential component to manage distributed files and their parts stored on multiple disks. They are responsible for providing pattern to a file to be stored across multiple parallel disks. The storage pattern eventually impacts the file access pattern therefore, it is also a factor in setting the magnitude of parallel IO bandwidth. This feature is common in Lustre File System (LFS used in this research), General Parallel File System (GPFS) and Parallel Virtual File System (PVFS) which are popular in HPC large scale computing [8, 34, 32].

A typical parallel or distributed file system enables hosts to access separate or common files on shared storage resources over the network. The client side usually does not have the direct access to the block storage being managed by the server side. However, the client nodes can access file or data on storage indirectly by a remote access protocol over the network. This restricts the access of client side to the file system depending on access list and capabilities of multiple servers and clients. It is unlike the clustered filesystem storage giving equal access to all nodes over the block storage file system. The other main advantage of distributed file system is providing improved bandwidth via transparent replication and distribution of files data. Additionally, it also provides fault tolerance such that system continues to work without any data loss in case a limited number of storage nodes of a file system are offline.

In this research, the LFS has been used as underlying distributed PFS for parallel storage management within the HPC cluster Kay as presented by IO Stack in Figure 2.10 [8, 51]. The Lustre is short abbreviation for Linux Cluster since it is specifically designed for Linux based large-scale cluster computing and have been used on a majority of the Top500 supercomputers. It is generally known and used for high performance data IO processing scaling to petabytes over thousands of nodes, and somehow, easy to configure and deploy.

The LFS storage architecture has three functional units: 1) Meta Data Servers (MDSs), 2) Object Storage Servers (OSSs) and 3) Clients accessing the file data, as shown in Figure 2.15. A single or multiple MDSs can contain single or multiple Meta Data Targets (MDTs) - the disks for LFS to keep namespace information about file names, directory paths, access permissions and file distribution pattern. MDTs single

disk dedicated file system is mapped to the client-side nodes to give them access to files and paths, and inform whether an object makes a file or not. Then single or multiple OSSs can contain single or multiple Object Storage Targets (OSTs) - the disks containing actual file data stored on them. An OST employs an object-based file system which enables read/write operations. LFS capacity is defined by the number of OSTs and their usable bytes space. Then finally single or multiple clients remotely access the file data from OSTs over the cluster network.

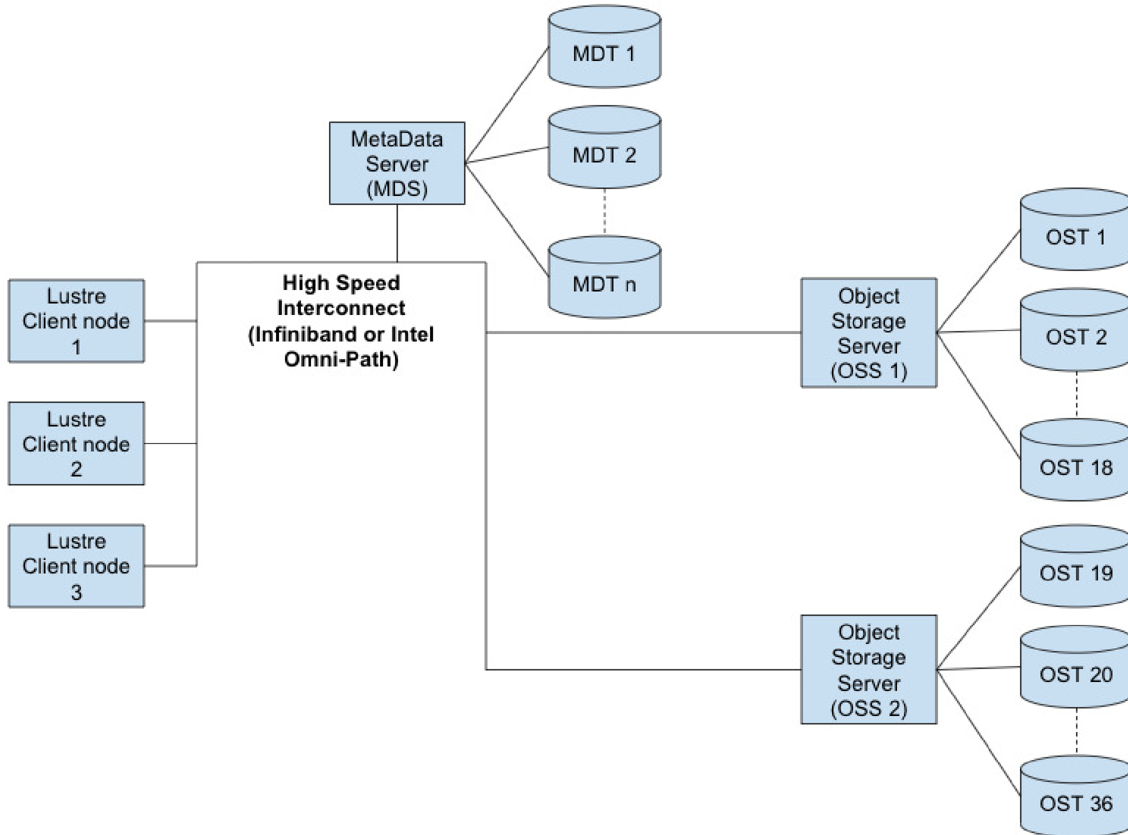


Figure 2.15: Lustre Architecture.

The clients are represented by unified namespace for the file system and its files data, allowing current and coherent read/write access. If a client request access for a file path, it is first lookup on the MDS from its MDTs. Then either a new file is created, or the distribution pattern of the existing file is returned to the client-end. The file locking on OST let client execute more than one read and write operations with synchronized concurrent access. However, client cannot directly modify file on OSTs rather than by OSSs dedicated tasks as per access requests. This ensures scalability, security, and reliability to avoid the file system gets corrupted by unsynchronized clients read/write requests.

The linux command to list all the available MDTs and OSTs is `lfs df -h`. The Figure 2.16 shows the example of the listing command that output 4 MDTs and 36 OSTs available in Kay cluster used in this research. The total bytes capacity of disks used and available are also shown. Additionally, the directory on which the LFS is mounted on shown as well. The command to get file layout pattern on the disk is `lfs getstripe somefile.txt` where `somefile.txt` is any file path whose layout is intended to be displayed. Provided that the file is already created with a layout pattern. The

Figure 2.17 shows the example output of the layout information of the file distribution pattern. The `lmm_stripe_count` value is 4 means file is distributed along 4 OSTs and `lmm_stripe_size` value is 1048576 means the file chunks are separated and spreaded across the OSTs by 1MB of data slicing. Whereas `lmm_pattern` is `raid0`, simply means file is placed on multiple disks by disk striping. It should be noted that this file and pattern is generated by the command `lfs setstripe -c 4 and -S 1M somefile.txt` where `-c 4` is number of disks are 4 and `-S 1M` is data slicing unit is 1MB. So, if this file was not created already then new empty file is created so, in future bytes will be filled on storage according to the pattern specified in example.

```
-bash-4.2$ lfs df -h
UUID                       bytes        Used   Available Use% Mounted on
ichec-MDT0000_UUID        515.0G       28.2G      479.8G   6% /ichec/work[MDT:0]
ichec-MDT0001_UUID        515.0G       37.9G      470.2G   8% /ichec/work[MDT:1]
ichec-MDT0002_UUID        515.0G       26.9G      481.1G   6% /ichec/work[MDT:2]
ichec-MDT0003_UUID        515.0G       36.5G      471.5G   8% /ichec/work[MDT:3]
ichec-OST0000_UUID         28.7T       23.6T        4.8T  84% /ichec/work[OST:0]
ichec-OST0001_UUID         28.7T       23.5T        4.9T  83% /ichec/work[OST:1]
ichec-OST0002_UUID         28.7T       23.0T        5.4T  82% /ichec/work[OST:2]
ichec-OST0003_UUID         28.7T       23.5T        4.9T  83% /ichec/work[OST:3]
ichec-OST0004_UUID         28.7T       22.4T        6.1T  79% /ichec/work[OST:4]
ichec-OST0005_UUID         28.7T       22.7T        5.7T  80% /ichec/work[OST:5]
ichec-OST0006_UUID         28.7T       23.2T        5.2T  82% /ichec/work[OST:6]
ichec-OST0007_UUID         28.7T       23.3T        5.1T  82% /ichec/work[OST:7]
ichec-OST0008_UUID         28.7T       22.9T        5.6T  81% /ichec/work[OST:8]
ichec-OST0009_UUID         28.7T       23.3T        5.2T  82% /ichec/work[OST:9]
ichec-OST000a_UUID         28.7T       22.1T        6.3T  78% /ichec/work[OST:10]
ichec-OST000b_UUID         28.7T       22.9T        5.5T  81% /ichec/work[OST:11]
ichec-OST000c_UUID         28.7T       23.2T        5.3T  82% /ichec/work[OST:12]
ichec-OST000d_UUID         28.7T       23.6T        4.9T  83% /ichec/work[OST:13]
ichec-OST000e_UUID         28.7T       23.3T        5.2T  82% /ichec/work[OST:14]
ichec-OST000f_UUID         28.7T       22.7T        5.7T  80% /ichec/work[OST:15]
ichec-OST0010_UUID         28.7T       23.1T        5.3T  82% /ichec/work[OST:16]
ichec-OST0011_UUID         28.7T       23.6T        4.8T  84% /ichec/work[OST:17]
ichec-OST0012_UUID         28.7T       23.3T        5.1T  82% /ichec/work[OST:18]
ichec-OST0013_UUID         28.7T       23.4T        5.0T  83% /ichec/work[OST:19]
ichec-OST0014_UUID         28.7T       23.3T        5.1T  83% /ichec/work[OST:20]
ichec-OST0015_UUID         28.7T       23.2T        5.2T  82% /ichec/work[OST:21]
ichec-OST0016_UUID         28.7T       23.0T        5.4T  81% /ichec/work[OST:22]
ichec-OST0017_UUID         28.7T       22.4T        6.0T  79% /ichec/work[OST:23]
ichec-OST0018_UUID         28.7T       23.0T        5.5T  81% /ichec/work[OST:24]
ichec-OST0019_UUID         28.7T       23.1T        5.3T  82% /ichec/work[OST:25]
ichec-OST001a_UUID         28.7T       23.0T        5.5T  81% /ichec/work[OST:26]
ichec-OST001b_UUID         28.7T       23.1T        5.4T  82% /ichec/work[OST:27]
ichec-OST001c_UUID         28.7T       23.6T        4.9T  83% /ichec/work[OST:28]
ichec-OST001d_UUID         28.7T       22.8T        5.7T  81% /ichec/work[OST:29]
ichec-OST001e_UUID         28.7T       23.3T        5.1T  82% /ichec/work[OST:30]
ichec-OST001f_UUID         28.7T       23.1T        5.3T  82% /ichec/work[OST:31]
ichec-OST0020_UUID         28.7T       23.6T        4.8T  84% /ichec/work[OST:32]
ichec-OST0021_UUID         28.7T       23.1T        5.4T  82% /ichec/work[OST:33]
ichec-OST0022_UUID         28.7T       23.2T        5.3T  82% /ichec/work[OST:34]
ichec-OST0023_UUID         28.7T       22.6T        5.8T  80% /ichec/work[OST:35]
```

Figure 2.16: Lustre command to list available MDTs and OSTs.

Coming towards the main feature of LFS is the file data storage pattern that projects IO bandwidth with the given number of processes running IO over multiple OSTs disks. This storage pattern is defined by two parameters as already seen, namely: 1) stripe count - the number of parallel disks on which file is intended to be placed by distribution

```
-bash-4.2$ lfs getstripe somefile.txt
somefile.txt
lmm_stripe_count:  4
lmm_stripe_size:   1048576
lmm_pattern:       raid0
lmm_layout_gen:    0
lmm_stripe_offset: 25
        obdidx           objid          objid           group
            25        32637069      0x1f2008d      0x980000400
            27        33529762      0x1ff9fa2      0x2c0000401
             7        32972225      0x1f71dc1      0x680000400
            10        50436225      0x3019881      0x240000401
```

Figure 2.17: Lustre command to get File Distribution Pattern.

and 2) stripe size - the size of file chunk parts that are intended to be distributed across the disks in round-robin fashion. The Figure 2.18 depicts file distributed across OSTs disks. An example could be the 4 MPI processes reading/writing file of size 16 MB from 4 OSTs as a stripe count where data slicing unit 1MB as its stripe size. It should be noted that there can be a contention of all running parallel MPI processes at each single disk access since their related chunks are striped on every disk. As mentioned earlier, OSTs are employed with file locking mechanism to avoid unsynchronized concurrent access of processes by means of data race condition on file stored, especially for write operation. This leads to IO bandwidth performance degradation therefore, there have been much research related to this problem to overcome this issue, discussed in the next section.

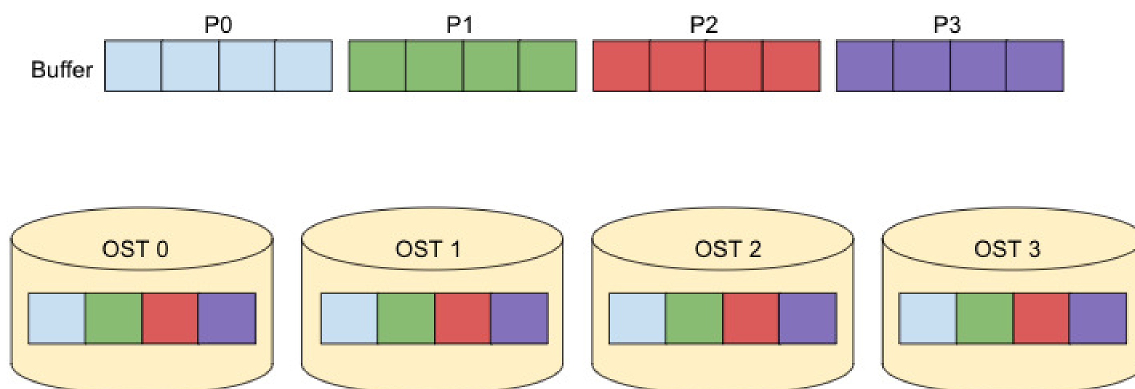


Figure 2.18: File Striping.

The Figure 2.19 depicts different striping where requested stripe count is 4 and stripe is 2MB. Therefore, due to round-robin data slicing only first two OSTs will be occupied and last two are unused presented with white boxes.

Similarly, in Figure 2.20 the last two OSTs are again left unused since stripe size was 8MB with 4 stripe count requested. Therefore, first two processes buffers got fixed in OST0 and last two processes buffers fitted in OST1 which completed the 16MB of file size.

Unlikely, the Figure 2.21 represent a different scenario where only stripe count for 2 OSTs requested with stripe size of 4mb. Therefore, due to same distribution strategy by LFS, the alternate processes buffers got fixed on the disks.

It should be noted that in Figures 2.18 to 2.21 each used disk contains at least two processes buffers or file chunk parts. This can cause contention among the running

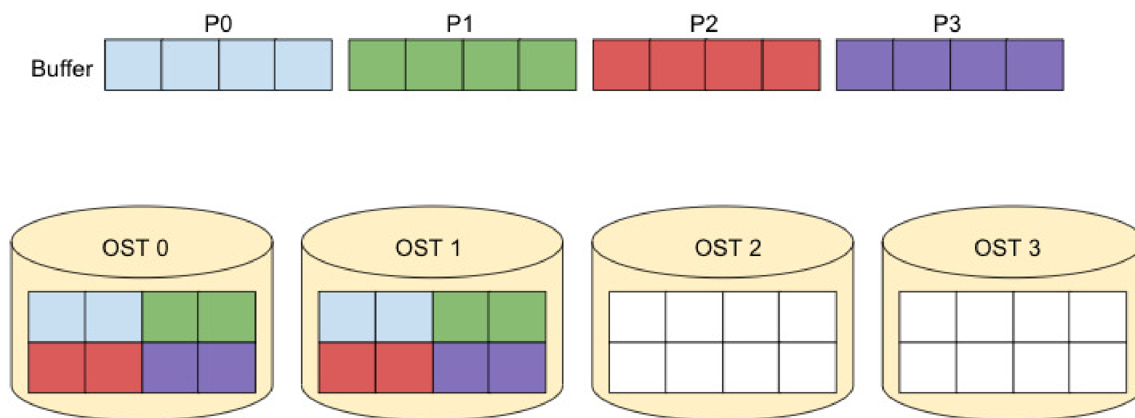


Figure 2.19: File Striping with stripe count 4 and stripe size 2MB.

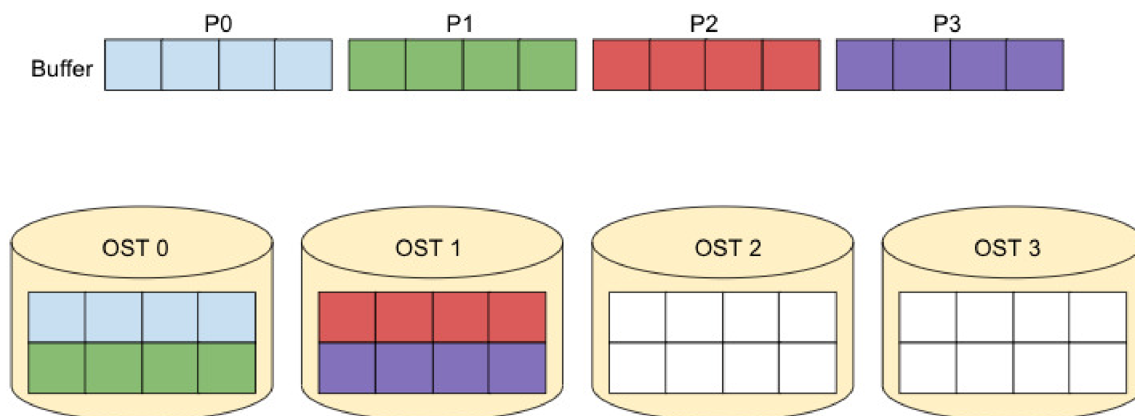


Figure 2.20: File Striping with stripe count 4 and stripe size 8MB.

parallel MPI processes at each single disk access since the chunks are striped on every OST. As mentioned earlier, OSTs are employed with file locking mechanism to avoid unsynchronized concurrent access of processes by means of data race condition to write file, and even for reading operation on file. This slows down the IO process and leads to bandwidth performance degradation therefore, there have been much research related to this problem to overcome this issue, discussed in the next section. This sums up the basic information about the Parallel Lustre (Linux Cluster) File System.

2.3.3 Previous Related Work

2.3.3.1 Research Work around Data-aligning strategies

Since LFS employs file locking mechanism that bares concurrent access of multiple processes at a time therefore, it greatly damages the IO bandwidth performance. It is because only one MPI process can have access to a file on disk at an instance. Despite the consideration for a moment that there is no file locking employed even then the disk header can perform only one read at a time. This one hardware limitation of HDDs cannot be ignored when designing software solutions for efficient parallel IO.

The solution to the file locking problem lies in data-aligning methodology adopted by number of researches in [9, 10, 11, 12], to further support in their own approaches. The strategy here is to set stripe count equal to number of MPI processes and stripe size equal to file chunk size. As the lustre stores bytes in round-robin fashion therefore, when

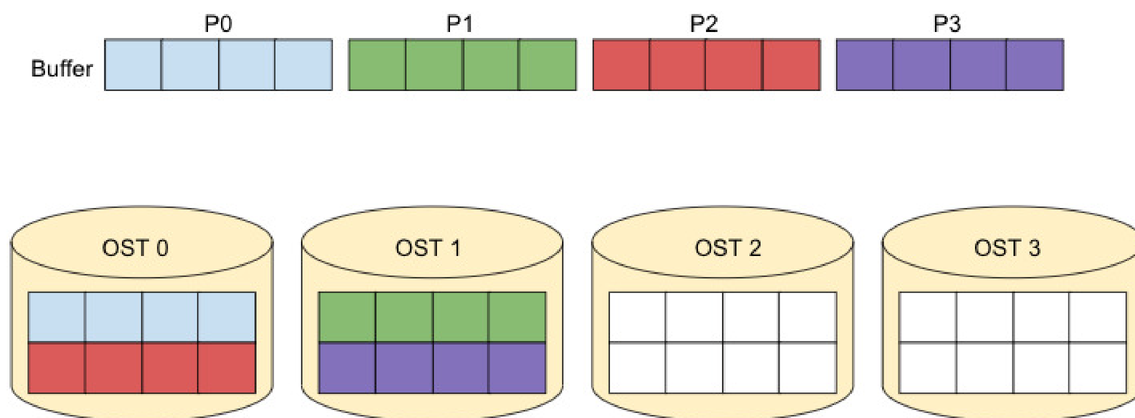


Figure 2.21: File Striping with stripe count 2 and stripe size 4MB.

stripe size equal to chunk size it will store each chunk on separate disks such each process would access separate disk for its corresponding file chunk.

In [12], Yu et al. introduced the Hierarchy Striping technique for file IO access across the Lustre File System. This technique creates the groups or pairs of OSTs for each process to access its subfiles on them. Each process will be accessing their respective group of OSTs for IO with free of file locking overhead and in the end all subfiles are joined back. This technique reduces the File Locking overhead in the Lustre File System and significantly improves the performance of collective IO.

Y-Lib: A User Level Library presented in [11] by Dickens et al., highlights the overheads in IO bandwidth performance on Lustre File System using MPI's Collective Two-phase IO strategy, in terms of data access patterns MPI aggregator processes-to-OSTs. First overhead is File locking by Lustre such that only one process can acquire a lock to access a file strip on an OST at a time that means there can be no concurrent access by two or more processes. The second overhead exploits the first one as the data for each process is being distributed across all the OSTs therefore, it becomes an all-to-all data access pattern as all processes would have to access all the OSTs to get their file blocks or chunks. To overcome this problem Y-lib: a user level library is designed on top of the MPI. It redistributes the data into one-to-one OST access fashion such that file view is set for each process to access each OST where their desired file blocks are being placed. This is generally known as data-aligning as depicted in Figure 2.22. When Two-phase IO request will be called now each process will have its independent but a lock-free concurrent access to each OST to pick up their respective file blocks. This significantly improves the Two-phase IO bandwidth performance with the factor of ten.

In paper by Liao [10], the technique is discussed to align the starting and ending offsets of partitions of file domains for each MPI process within the stripe sizes of parallel file systems to avoid file locking on two file domains being accessed from the same stripe server. This technique contributes to 30 times more of write bandwidth on IBM GPFS and Lustre File System.

The research conducted by Li et al. [9], presents another efficient way of writing output matrix (from matrix-multiplication in Cannon's Algorithms) to file is being discussed to over the extent-based locks of Lustre file system by aligning processes file partitions to stripe sizes of OSTs, through one-to-one pattern and stripe-continuous pattern. This improved the write bandwidth throughput significantly.

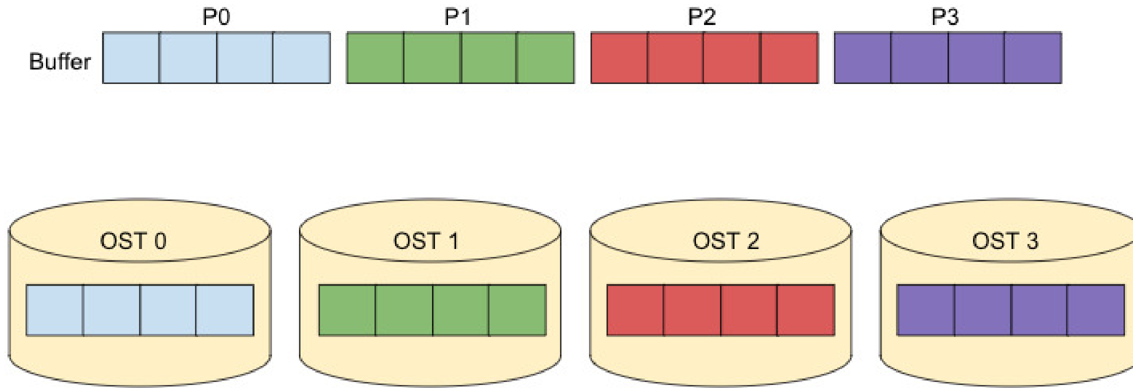


Figure 2.22: Data aligning at Lustre File System level.

2.3.3.2 Techniques Implementations at MPI-IO or LFS Level

There are also other techniques proposed in previous research which are not data-aligning oriented. However, they regard to file pattern reorganization, replication, and other MPI-IO or LFS level implementations. For example, work presented in [66] by Tran, addresses the key issues in access patterns of file of specific parallel MPI applications which affects the MPI atomicity because of file overlapping chunks at the splitted Storage backends employing Lustre system. Due to Lustre's exclusive access of processes, it badly affects the concurrency of application and thus IO bandwidth performance. The approach to resolve this moves around three key principles which are 1) Dedicated API at the level of the storage back-end (a library integrated with ROMIO), 2) Data Stripping and 3) Versioning as a key to enhance data access under concurrency (by using shadowing technique). It achieved an aggregated throughput ranging from 3.5 times to 10 times higher in several experimental setups, including highly standardized MPI benchmarks specifically designed to measure the performance of MPI-IO for non-contiguous overlapped writes that need to obey MPI-atomicity semantics.

The work by Byna et al. [67], presents the parallel IO prefetching technique via MPI file caching and IO signatures. Prefetching is an effective latency hiding optimization. However, traditional prefetching strategies limit themselves to simple strided patterns and avoid complex prediction strategies so that a prefetching decision can be made swiftly. Therefore, a combination of post-execution analysis with runtime analysis to reduce the overhead of predicting future IO reads, was introduced. This paper expands the classification of IO accesses and introduced a new IO signature notation. This notation can be used in reducing the size of trace file, in making decisions about prefetching, and in selecting prefetching strategies. The signature is predetermined and is attached to the underlying application for future executions. In this prefetching strategy, a separate thread is being used to prefetch at runtime. It was implemented in the MPI-IO implementation (ROMIO) and results shown significant performance benefits.

The work demonstrated in [68] by He et al., proposed the file pattern aware reorganization of file blocks mapping at the PFS and the storage level. The diverse access patterns of data-intensive applications cause physical file organization in parallel file systems not matching with the logical data accesses. Therefore, small, and non-contiguous data accesses can lead to large overhead in network transmission and storage transactions. In this paper, a pattern-aware file reorganization approach is proposed to take advantage of application IO characteristics by leveraging the file remapping layer in the existing

MPI-IO library. The proposed approach maintains a data block remapping mechanism between user data accesses and file servers, to turn small and non-contiguous requests into large and contiguous ones. Experimental results demonstrate that this approach shows improvements of up to two orders of magnitudes in reading and up to one order of magnitude in writing. This could be somehow like the research by Yu et al. in [12]. This is a bit closer to the different application specific IO techniques, as they can be specific to a particular file pattern at large scale, presented by Poyraz et al. in [69].

The research by Qian et al. [70], the paper addresses the key issue of IO congestion control in the Lustre File System. As the scale of clients to access file IO increases it induces the higher congestion over the OSSs and OSTs. An IO dynamic RCC (Request Concurrency Credits) based congestion control mechanism is being used for the scalable Lustre File System to overcome this problem. When servers are under light load; on the other hand, it can throttle the clients' IO and limit the number of IO requests queued on the server to control the IO latency and avoid congestive collapse, when the server is overloaded. This improves the IO throughput for both servers under light and heavy load of IO accesses from clients or processes.

The research by Tsujita et al. [71], presents the multithreaded Two-Phase IO technique. It covered the gap for the performance in the collective two-phase IO, which showed the 60% improvement. The technique covered this space was multi-threaded pipelining based Two-Phase IO. One main-thread manages the hole check and data exchange for IO request and the other IO-thread performs the read and write operations. Both are synchronized in a pipelined manner that there are no overlapping requests or data on collective buffers and user buffers during the IO processing.

The work by Hammond et al. [72], implements BigMPI Library on top of native MPI Library to address the problem of large count arises due to the INT_MAX value limitation which is around 2GB. When programmer wants to send/receive or read/write the larger bytes of data in one MPI call it limitizes it to value of INT_MAX that means more MPI calls will be needed to communicate rest of the data. As the MPI call is an expensive operation in terms of execution time therefore, it can degrade the overall performance but sending the whole large chunk in one call makes a lot of difference. So, in BigMPI library the large byte count MPI data types can be created by using its functions written over the native MPI functions and using the same functionality the functions to send/receive or read/write operations are also modified in a way that a user can pass more than 2GB of data as maximum possible.

The research done by Han et al. [73] presents another unique approach at LFS level. The LFS has been evaluated from the aspect of RPC (Remote Procedure Call) threads running on MDSs (Meta Data Servers), OSSs (Object Storage Servers) and Lustre Clients to communicate with each other upon the IO request. These threads normally match up with the number of CPU cores in those servers with certain ratio. Changing the number of threads of OSSs over different number of Lustre Cilent requests, can lead upto maximum of 114% improvement in IO throughput performance.

The technique presented in [74] by He et al., work around reorganizing of file domain blocks in heterogeneous storage, which can be somehow like work done in [68]. The paper describes the new collective IO technique implemented in MPI-IO library enabling the cross-platform systems in an HPC cluster to read or write data over the HDD or SSD storage devices. This technique has been implemented in ROMIO which reorganizes the File Domain Blocks in the Aggregator processes (leaving the first aggregator) buffers such that at one cycle of data storage process either the blocks are being placed in

HDDs in parallel or in SSDs in parallel. This significantly reduces the total number of cycles to process IO requests and outperformed the previous ROMIO's Collective IO and Concurrency Awareness IO techniques.

Then work presented by Zhang et al. [75], proposed new scheme IR+ which does the profiling run over the file to detect its interfered segments by MPI programs. It replicates it over the different SSDs so the programs can access them on independent disks which reduces the excessive disk seek heads to access those segments. This ultimately improves the disk IO for the future. This was another work done in relation to heterogeneous storage hardware in HPC clusters.

2.3.3.3 Related Work in LFS Performance Evaluation

There have been research studies regarding just LFS performance Evaluation. For example, work presented by Jackson et al. [76], addresses the key issues of IO in most of the HPC systems. This was from the aspects of both software libraries and underlying hardware infrastructures employed by LFS and IBM's GPFS. The performance evaluation was supported by benchmarking on various factors like multiple processes, file striping and IO libraries: MPI-IO, HDF5 and NetCDF [7, 77, 78].

Wang et al. [79], explains and shows the performance results related to IO bandwidth over the Infiniband based networked HPC cluster and employing the Lustre Parallel File System over it. Where, key factors are client count, stripe count, stripe size, etc.

Similarly, Jian et al. [80], analyze the different factors affecting the IO performance of Lustre File System. Those key factors are underlying network hardware, number of processors requesting IO in single application or client and stripe count values.

Then the research by Inacio et al. [81], a statistical analysis of the IO performance on PFSs was conducted. This paper also presented comparisons of IO performances over different factors. The key factors were the experimental environments setup, IO strategies with changing stripe counts and operations (read/write).

In the past research studies, the work has been carried out for performance evaluation of LFS by means of creating a mathematical model approach to predict IO performance. For example, Zhao et al. [82], conducted the in-depth survey on LFS. The different factors have been highlighted during the performance evaluation. Then prediction model based on Grey Theory was applied to them, leading to mathematical relation among those factors. This prediction model could obtain better prediction precision and could be further applied to performance evaluation of other parallel file systems. The similar approach was also carried out in [83] by Zhao et al., where the experiments based on the performance of IO bandwidth over Lustre File System followed by a mathematical based prediction model. This was designed to predict the IO bandwidth from 17% to 28%. The model involved various factors affecting the IO bandwidth such as OSS threads for OSTs, number of OSTs, read-write request size, etc. However, there have been more research conducted in later years, which were used to predict and optimize IO performance based on the various ML techniques, discussed in the next chapter.

2.4 Summary

In this chapter, the domain of HPC has been explained from the aspects of different memory models computing or programming and the usual parallel IO processing techniques followed within the supercomputing clusters. It first introduces the overall environment

of HPC clusters and the scenario of working parallel IO. Afterwards, it has been discussed that users can program distributed memory applications to work over multiple compute nodes within cluster nodes by splitting and distributing the data, also known as divide-and-conquer via standard MPI library. Then similar approach can be implemented by shared memory application to work over only one compute node by splitting and distributing the tasks among CPU cores or threads via the OpenMP standard library. By combining these two approaches users can program distributed-shared or hybrid memory application by first splitting and distributing the tasks among compute nodes and then further among each CPU cores or threads via combined MPI-OpenMP standards. Additionally, it has been discussed that modern HPC clusters come with the dedicated GPU cards on the number of compute nodes. Therefore, users can also program to execute tasks by splitting and distributing them among GPU threads on either single or multiple compute nodes via CUDA or MPI-CUDA standard libraries in case of NVIDIA's graphic cards.

The other half of the chapter discusses HPC storage infrastructure within the clusters involving parallel IO processing and the parallel LFS to support it in a performance efficient way. It first discusses the basic parallel IO technique via multiple MPI processes on single or multiple nodes, reading or writing their independent buffers on disks, following KAY cluster specific IO stack. It is a non-collective IO approach for accessing contiguous data. However, then various collective IO approaches have been discussed since the pattern of buffers data at processes end usually do not match with pattern on multiple disks at the back-end. This makes it collective IO approach efficient to some extent when accessing non-contiguous data on parallel disks.

Since the parallel storage is managed by a particular PFS, in this research it was LFS. It has been discussed that LFS applies a particular file striping to store its data on disks, based on stripe count of OSTs disks and stripe size data slicing unit. This results in data of processes being non-contiguously stored on the number of disks. Therefore, it creates issues of contention for reading/writing data among MPI processes running in parallel because OSSs applies file locking on OSTs to synchronize concurrent file access. This causes IO bandwidth performance degradation. Later in the thesis, different strategies have been discussed from past research with perspective of machine learning to improve parallel IO performance. This sums up the discussion about the HPC and cluster storage.

Chapter 3

Seismic Data & ExSeisDat

3.1 Introduction

Seismic data is image data from underneath the earth's surface. It is normally used and analysed by geophysicists working in various fields i.e., earthquake data studies, mineral exploration, reservoir management, Oil/Gas industry, etc. Instruments like seismometers are used to collect the sub-surface data, normally through a wave reflection mechanism [84]. It depends on the type of work of what is required to be located under the earth surface at certain depths like minerals, rocks, water, or oil. Once the seismic data is collected by the instruments then it can be visualized on computer systems or other graphical display devices for analysis. The importance of seismic data can be seen from the study of earthquakes jeopardising human lives, to its usage in Oil/Gas industry [2, 3].

The specific usage of seismic data in Oil/Gas industry is the concern of this research when comes to its large IO processing in the HPC clusters. The seismic data collected to locate oil for drilling, scales to petabytes in size thus, making it time consuming for computer and software systems to process it. The IO operations of such large sized seismic data significantly lead to performance issues and a need to develop HPC applications to process it along with other essential functionalities. This is where the ExSeisDat library comes in serving the purpose of efficiently processing seismic data as a solution within HPC clusters [1].

The ExSeisDat was explicitly developed to efficiently process seismic data. The library was built using the C/C++ standard MPI library [7, 13]. A user can write distributed parallel application that interfaces with ExSeisDat APIs instead of directly calling MPI functionalities. Considering the IO Stack of an ExSeisDat based application in HPC environment, the Figure 3.1 shows the ExSeisDat layer between application level and MPI level layers. This is the IO stack used in Kay cluster for executing the parallel application using ExSeisDat for seismic data operations. Additionally, it is used for further optimizing its IO processing as part of this research.

This chapter further discusses the seismic data file format, the working detail of the ExSeisDat library and presents the performance results.s

3.2 Seismic Data Format in SEG-Y File

The standard format used for seismic data within computer systems to read/write is SEG-Y file format [4]. The seismic data contains traces data of wave reflections exhibiting

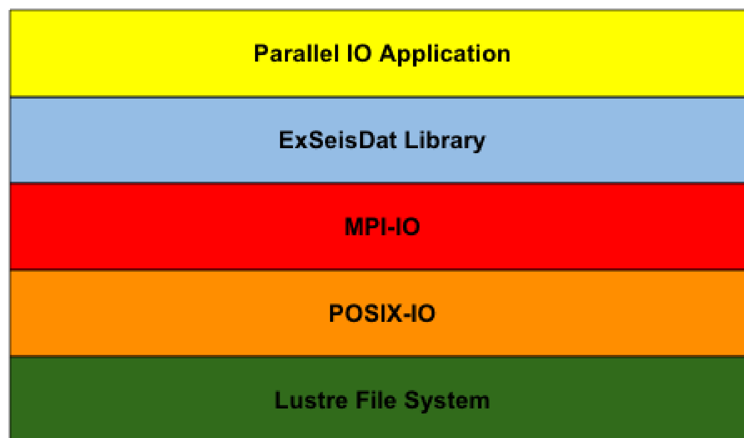


Figure 3.1: IO Stack involving ExSeisDat.

depths underneath earth surface, in numerical form, which can further be processed or visualized on computer or display devices. However, the traces in SEG-Y file are not arranged in sequence as they are recorded. The Figure 3.2 shows the pattern of arrangement of trace data in SEG-Y format.

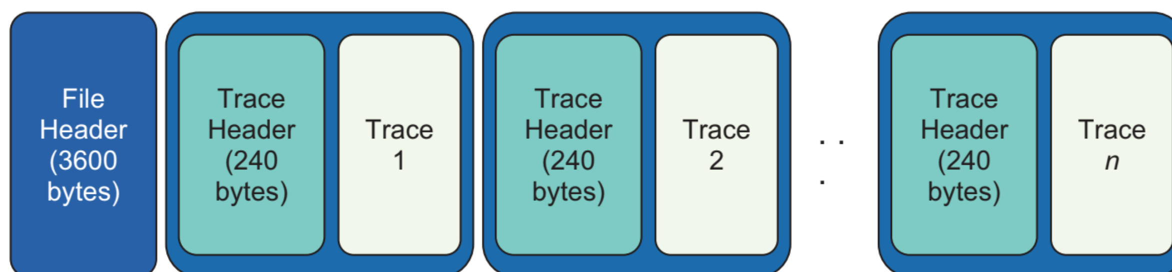


Figure 3.2: Structure of SEG-Y file format. Reprinted from "ExSeisDat: A set of parallel I/O and workflow libraries for petroleum seismology" by Fisher, M.A., Conbhuí, P.Ó., Brion, C.Ó., Acquaviva, J.T., Delaney, S., O'brien, G.S., Dagg, S., Coomer, J. and Short, R., 2018, *Oil & Gas Science and Technology–Revue d'IFP Energies nouvelles, 73*, p.74 [1].

A SEG-Y file start with a File Header of initial 3600 bytes then each trace data is place alternatively separated by the trace header of 240 bytes. This File Header is a breakdown of 3200 bytes Textual File Header and 400 bytes Binary File Header. The values against all headers and traces were earlier stored in Big-Endian number format, means most significant and byte and bit would be closer to beginning of file. However, in revision 2 of SEG-Y format, the Little-Endian and byte-swapped ordering format is used specially in regard to IO performance. This provides independence from the number format to which file is written as the byte ordering of a SEG-Y file would not be affected if data is stored on a mainframe's tape or a PC's disk. Additionally, the values stored are treated as signed values or 2's complement values regardless of their sizes whether they are 2-bytes, 4-bytes, or 8-bytes long. Since, Textual File Headers values are just texts or strings so, their byte ordering would always be same at the time of reading/writing.

As all the headers have fixed byte length however, a trace data length can be varied or fixed. A trace data length or size depends on the number of samples collected per trace. The info regarding trace data length fixed or varied, resides in the Binary File Header.

If a fixed length trace flag is set in the Binary File Header, it means all traces data are of the same length otherwise they are varied within a file. Upon a fixed length trace flag is set, the other useful information in the Binary File Header is about the sampling of trace data. For example, if a six seconds' data sampled at a two-millisecond interval then total number of samples per trace would be 300 within a SEG-Y file. It is also recommended for the people collecting seismic data to keep the bytes size consistent in each trace record. This makes equal number of samples are existing in all traces. This uniformity keeps easy read/write access in terms of IO performance. Otherwise, traces with varying sizes would lead to complex random access of file on disk. The reason is the starting indices of traces following the first trace would not be known without fetching the whole file from disk into memory. This can degrade the IO performance and eventually other seismic data workflows processing.

The other essential data for each trace is the source and receiver locations in form of coordinates. It is a basic requirement to process and interpret seismic data. The seismic trace coordinates can be either as geographic coordinates or grid coordinates. The source-x and source-y represents the source coordinates however, the receiver coordinates are not directly presented in SEG-Y format, rather they are computed by the under-surface depth from the source coordinates. The form of coordinates is determined by the coordinate reference system (CRS) specified in the Binary File Header and Trace Headers [85]. If the coordinates having units in seconds of arc, decimal degrees, or Degrees-Minutes-Seconds (DMS) format, then source-x (or X coordinate) values would represent longitude and the source-y (or Y coordinate) values would represent latitude. In case the coordinates are in the form of Greenwich Meridian grid projected system then a positive value in meters would depict east or north of the equator otherwise, a negative value would depict south or west [86, 87]. However, this can still produce ambiguity if the coordinates format is unknown or uncommon in one complete SEG-Y dataset. It can be avoided by a single CRS defined in the Binary File Header and the Extended Textual Headers for all coordinates in an individual SEG-Y file data. Furthermore, the units of coordinates should be common throughout all the traces. The common properties of SEG-Y data would make processing easier, less complicated thus, time efficient one. Whereas randomness and irregularities in SEG-Y data would make it more complex to process, even in designing logic for large workflows processing routines. Therefore, efficient SEG-Y data IO and its other workflows processing for extremely large datasets are required.

These are some very brief and basic details about the arrangement and complexity of seismic data in SEG-Y format. The complete detail about each value and properties are mentioned by Hagelund et al. [4]. In general, it can be seen that the SEG-Y data naturally has its own unique complexity once it is collected when transferred to magnetic tapes or disks on computers and can largely scale to petabytes in size. This has inspired researchers to develop a powerful high performance tool or application for efficient processing of seismic data. The next section describes ExSeisDat as a HPC application specifically created to address the need of efficient large scale SEG-Y data processing for the Oil/Gas industry.

3.3 ExSeisDat

3.3.1 Motivation

The resolution of seismic data analysis and studies have overall increased immensely in the past decades thus, the size of such data conveniently scales up to terabytes and beyond [88]. The high expansion of seismic data became inevitable since, its high resolution must give crystal clear insights of hydrocarbons from underneath the earth surface. Thereby, giving precise locations to extract petroleum and other related products [89, 90].

As the largely dense seismic datasets have enhanced their interpretation therefore, they also have yielded IO and memory overheads. For such problems HPC parallel IO has become an essential necessity however, it is complex due to large volume and the changing access patterns in multiple separate files. This means particularly for random access pattern such as in the case of SEG-Y format. In the last chapter, it has been discussed that there is random access and arrangement of file distribution on disks due to LFS (or any PFS or even serial file system) management. This can produce bottlenecks or contention among processes operating on multiple parallel storage hardware, which leads to IO bandwidth degradation. Caching can have limited positive effects since, an entire large file reaching terabytes cannot be prefetched or read into the RAM. For writing on disks, shared file access among numerous parallel processes becomes more inefficient since, it must deal with the PFS file locking at runtime. As the SEG-Y traces are placed alternately, and therefore are separated by their corresponding trace headers shown in Figure 3.2 therefore, parallel writes of such patterns become less efficient.

Programmers are obliged to deal with SEG-Y format directly despite, the availability of other conversion tools to visualize it in a convenient way. All those tools are serial and can take good a significant amount of time to process. Furthermore, the SEG-Y format is too complex to modify being a legacy code which is an industrial constraint. Therefore, the issue of SEG-Y access pattern could only be addressed at a library level within an IO Stack of the HPC cluster. A software layer that could provide abstraction to geologists or geophysicists from the complex working nature of underlying PFS however, putting the right code logic to give the efficient file access at the application level.

The Extreme-Scale Seismic Data (ExSeisDat) library served this purpose by addressing the overheads related to seismic data IO processing within the Oil/Gas industry via advanced parallel approaches combined with the IO storage hardware and software technologies. ExSeisDat was designed and developed in partnership with Tullow Oil plc and DataDirect Networks (DDN) to be both scalable and user friendly. ExSeisDat includes two open-source libraries: ExSeisPIOL, a low-level seismic parallel IO library, and ExSeisFlow, a high level parallel seismic workflow library. Both are discussed in next section.

3.3.2 Basic Design Concept

As stated earlier in the previous chapter that parallel IO enables multiple processes or compute nodes to execute read/write operations simultaneously within a system or cluster. The parallel IO of a single file in a cluster normally supported by a PFS as it stores file data in blocks across the multiple storage devices. This leverage multiple processes to read/write from multiple disks instead of being contended around one disk. However, LFS hinders the parallel IO performance to protect data by locking file portion on each OST at an instance. Furthermore, the file layout on multiple disks is hidden from

the application-level view. The MPI-IO collective or non-collective calls access pattern normally do not match with the file layout across the OSTs thus, causing a further IO and overall program performance degradation. Considering this scenario, the handling of parallel IO and workflows of a complex file format such as a SEG-Y in this case was quite a challenge which was addressed by ExSeisDat library.

ExSeisDat is the set of SEG-Y IO and workflow libraries implemented in an object-oriented paradigm on top of the standard C/C++ MPI APIs, to make parallel IO and file system optimization simpler alongside the other seismic data flows and computations. Ultimately, it is to facilitate the developer productivity and application performance. It is open source and portable, available at `https://github.com/ICHEC/ExSeisDat` under the LGPLv3 license.

The aims of ExSeisDat were to remove the bottlenecks and overheads in IO from the perspective of both development time and runtime. The libraries bridge the gap between MPI-IO and LFS different access patterns and hide the specific details of the SEG-Y file format. However, they still provide some openness for customizing the file and trace headers. The libraries also contain a language binding interface in C language platform for future development usability.

A key purpose of ExSeisDat when handling seismic SEG-Y data is supporting software and hardware to optimize one process reading or writing to only one OST disk as much as possible. This has been done to keep data-aligned to achieve maximum possible bandwidth. Additionally, the data access among processes, is also coordinated and communicated where possible. As this cannot be always a case therefore, ExSeisDat benefits from DDN's Infinite Memory Engine (IME) based hardware. IME middleware layer can convert random IO access patterns to contiguous IO access patterns. This can reorder the IO accesses to the PFS and optimizes the parallel IO throughput from ExSeisDat to IME to file system.

ExSeisDat is based upon its core libraries: ExSeisPIOL and ExSeisFlow for explicitly handling file IO and seismic workflows, respectively. Additionally, it contains, supplementary APIs as the wrappers on basic MPI calls for handling communication among processes and error logging. Whereas the majority of complex MPI calls are already contained within the two core libraries. The communication based supplementary API supports retrieving the number of MPI processes running and the rank of the current MPI process. The other example is the barrier wrapper function call which is implemented based on the `MPI_Barrier()` call to force some processes to wait at an instance in execution. Meanwhile, the other processes perform any tasks and then eventually call the barrier function. When each process comes to the instance of calling barrier function then all the processes become free to execute rest of the code. This provides synchronization among processes.

3.3.3 ExSeisPIOL

ExSeisPIOL is parallel IO library of ExSeisDat for efficiently extracting the traces and other related information from SEG-Y files distributed across the parallel storage devices. Despite the abstraction of the API, it is simple to use, it still requires three parameters related to IO from the application user, which are: memory limits, data selection and data decomposition throughout the parallel processes. Thus, leveraging the MPI-IO layer for efficiently accessing the PFS in multi-processing.

The further breakdown of the IO stack within the ExSeisPIOL design, consists of

multiple component layers responsible for communication between storage and MPI processes effectively. All components have downward dependency within the layers from top to bottom. However, on the same time it has flexibility to avoid or remove that dependency for code compilation and execution. These components are divided into three groups: the data layer, the object layer, and the API layer.

The data layer is at the lowest level to utilize MPI-IO function calls and manage data access on the file system by a process contiguously reading data block regardless of overlapping. To avoid data overlapping among processes, the number of data blocks of a specified by length can be read or written onto disks. In this way the data blocks can be separated by the predetermined offset value based on length for each process. Additionally, every process can write varying number of data blocks.

Then comes the object layer to specify and prepare data which the data layer can read or write. Its purpose is to transform the file and trace headers, and trace data into a block representation. As the object layer is separated from API level layers, file access pattern optimizations are performed independently from the packing operations to fill objects values. Furthermore, this object layer performs mapping of the file and headers attributes to their byte locations into memory thus, enabling the application programmer to use parameters in function calls by their names rather than by exact file paths. This layer is ultimately useful regarding addressing the particular description of SEG-Y file access.

Finally, the PIOL API layer that end-users will be directly calling for IO before seismic workflows computations, is specifically designed to read/write file and trace headers, and traces data, in a simplifying way. To support the memory management in this process, these values are read or written separately, allowing only the most required values. As mentioned before, the parameters are read and written by directly calling their names i.e., xSrc (x-source coordinate), ySrc (y-source coordinate), etc. ExSeisPIOL enable end-users to customize header data values. Furthermore, ExSeisPIOL enables them to create new parameters overloading for the SEG-Y format standard. In this way, the reference to standard SEG-Y parameters during programming code, is not required while referring to their customized parameters.

3.3.4 ExSeisFlow

ExSeisFlow takes advantages of the ExSeisPIOL to initiate and run seismic workflows in parallel once, the SEG-Y file data is read into memory and when the data needs to be written back to disks after the seismic operations. The memory limits, data selection and decomposition are implicitly handled. The separated headers and values are read or written internally. In the case of ExSeisPIOL, all fields are supposed to be specified explicitly, as stated earlier. This allows application end developers to put their full focus on only seismic data operations without worrying about the IO related conditions and essential necessities.

ExSeisFlow functionality aims at the pre-stack preprocessing operations which are determined as either trace, gather or survey wide. These categories point to a specific communication type. Trace based computations need communication within single trace data or headers data. Gather based operations need communication among all trace data and header data in a single SEG-Y file. Similarly, survey based operations also need communication among all trace data and header data during survey. The examples of trace computations are trace filtering and muting. Then examples of gather based com-

putations include executing gathers and transformations from radon to angle. Similarly, survey operations examples involve 4D binning and file sorting by x-source coordinates thus, a single workflow can be composed multiple function calls against such operations.

In a single workflow of ExSeisFlow, every MPI process is internally initialized, and parallel IO is executed after completion. Each operation is queued during the application execution. Upon the completion of the application, its destructor is called implicitly, and any required data is read into memory to perform remaining operations.

As the ExSeisFlow is developed to execute large scale datasets efficiently therefore, the internal memory usage is optimized by the library itself. Additionally, the advantage of queuing operations is that the data required is already known before performing any parallel IO operation. Therefore, only required data is read into memory while operations are being performed.

Since an operation has its own unique specifications in relation to its modification and data type dependencies, the modification level and communication type is set at the time of operation submission to the queue. This setting describes the data access type: header or trace, and the data modification can be one of the following options:

- traces added
- traces deleted
- trace values modified
- trace lengths modified
- header values modified
- traces reordered

Similarly, the modification dependencies can be one of the following:

- number of traces
- order of traces
- value of traces
- value of trace header

Additionally, the operation communication level can be one of the following:

- trace
- single gather
- survey

These categories settings enable ExSeisFlow to determine when a trace value or its metadata can be changed.

The operations in ExSeisFlow or SEG-Y workflows are executed according to the hierarchy as per the modification dependencies followed by the communication type settings. It should be noted that operations having modification dependencies are executed after those operations modifying that dependency. According to the communication hierarchy,

Table 3.1: ExSeisDat Benchmarking Settings.

Cluster	DDN.
File system	LFS with (IME mounted/unmounted).
Stripe count	40 OSTs.
Stripe size	1 MiB.
1 OST space	14 TiB.
File sizes	100 GiB, 500 GiB, 1 TiB, 2 TiB, 10 TiB, and 20 TiB.
Compute nodes	4.
Cores per node	16.

the trace wide type operations are executed prior to the gather wide type operations, followed by the survey wide type operations.

The traces and headers data frequently used by multiple operations within a workflow, are always cached to avoid redundant reads from main memory or disks. If any header or trace value is not modified frequently by operations in sequence of workflow, then it is saved on disk to reduce the data in memory space. Once all the data is modified by all the operations and is written on disk, then the remaining unmodified header or trace data is read into memory and subsequently written to output file on disk.

3.3.5 Overview of ExSeisDat Benchmarking Experimental Setup

The performance evaluation of any parallel IO operations or library requires benchmarking that depends on the hardware and software system specification on which the parallel application testing is carried out. These specifications include the underlying PFS type, number of disks (OSTs in case of LFS), striping unit for a file, and NHI interconnect speed. The specifications can vary among different HPC clusters and their file systems for parallel storage thus, significantly impacts the IO runtime of an application [91]. Similarly, the impact of modifications at the parallel IO library level, can be observed from the benchmarking results. Furthermore, the impact of increasing the number of processes reading and writing on the overall parallel IO runtime performance, is also evident in the benchmarking results.

The ExSeisDat Library was tested via benchmarking on DDN's cluster running LFS with IME mounted/unmounted. The system details are specified in Table 3.1. Each OST in this cluster had 14 TiB of space storage for the file system consisted of 40 OSTs as a stripe count value. Each file had been distributed by a unit size of 1 MiB as a stripe size value. In addition to this, the MPI-IO's LFS access was optimized. The profiling on LFS and LFS+IME stacks was conducted on 4 compute nodes where each node had 16 cores thus, one process on each core was executed. The benchmarked SEG-Y file sizes were 100 GiB, 500 GiB, 1 TiB, 2 TiB, 10 TiB, and 20 TiB. Since, the SEG-Y file sizes were varying therefore, the number of traces were kept the same to focus the benchmarks on the parallel IO runtime, instead of the computational runtime of the file sorting code logic. However, the samples per trace were varied within a trace data in each file size.

The parallel IO profiling was analyzed via the Darshan profiler tool [92]. This tool can track the number of MPI, and POSIX based calls within the application code to give the clear picture of the IO operations and their access patterns. The Darshan tool library covers MPI API and some other user space libraries APIs, by using `LD_PRELOAD` as the environmental variable which substitutes the IO calls with its own instrumentation.

It should be noted that Darshan makes no modifications to the application source code while instrumentation for profiling. Furthermore, it has been observed that the overheads introduced by Darshan instrumentation, are significantly less in case of the small file sizes (i.e., less than a GB), and they continue to reduce for the larger file sizes.

The ExSeisFlow's file sorting utility has been used to benchmark ExSeisDat library which consumed 32-cores on the cluster. Its sorting algorithm sorts traces from an input SEG-Y file by the trace header data i.e., x-source coordinate. Each process reads the minimum required data from each trace header within a file, in a parallel-distributed way. This indicates the memory required per process for a complete sorting is `O (number of traces / number of processes)`. This sorting algorithm is actually a variation of the parallel quick-sort [93]. In this utility, initially, every process sorts its own header data and keeps the track of original position of each header in the SEG-Y file. Afterwards, the processes send the lower half of their sorted values to the corresponding lowest ranked process. Consequently, the current process receives data from the corresponding highest ranked process. Eventually when the data is sorted, then each process sends the upper half of the sorted values to its corresponding highest ranked process. The procedure repeats unless the values are not modified at each process end, for the algorithm's single iteration. The worst case is that each process yields computational complexity about `O(N + (N/P) log(N/P))`, and communication complexity of `O(P)`, where `N` represents the number of traces in the SEG-Y file and `P` is the number of MPI processes.

The sorting algorithm utility from ExSeisFlow was ideal for the ExSeisDat's libraries benchmarking as it consists of running multiple IO access patterns. In the code logic, the individual traces are read contiguously whereas, trace and file headers are non-contiguous reads and writes, with the given stripe size. These variations of read and write patterns in algorithm shows the true potential of the ExSeisDat library.

3.3.6 ExSeisDat Benchmarking Results

The ExSeisPIOL was tested for underlying parallel IO, to show its ideal usage with the pre-existing seismic computations having complicated IO by porting the Kirchhoff migration utility, previously used by [94]. The results presented for ExSeisPIOL were rough estimations rather than direct representations of actual development. The code logic contained around 63 thousand lines of code with 44% IO related instructions. However, the porting of Kirchhoff migration decreased the IO related code instructions by 25%, with 16% decrease in overall code logic. The ExSeisPIOL calls replaced the previous IO calls. The ExSeisPIOL calls were including all the MPI-IO function calls alongside the type conversions and trace header data scaling. It should be noted that approximately 36% of code developed was related to IO optimizations and handling, which was specifically tough to code as per the developers reporting. In numerical figures, it approximately took 450 commits for parallel IO development and optimization whereas, it approximately took 30 commits for porting Kirchhoff migration to ExSeisPIOL. These ExSeisPIOL/Kirchhoff runtimes were within around 10% of the actual Kirchhoff migration code which was explicitly hand optimized by the expert(s).

The LFS supported IME shown notable performance increase in comparison to only LFS, as shown in Figure 3.3. It was evident from there that LFS-backed IME increased the read data performance by 2.8 times whereas, write performance was increased by 86 times. This resulted in 27% decrease in overall application runtime by LFS leveraging DDN's IME hardware. It should be noted that the computations portion of the sorting

algorithm utility was not considered in this speedup or runtime analysis as it was solely related to IO portion and its performance comparison among two file systems. The other noticeable fact was that separating IO and computations portions in terms of fraction could be exaggerating where both portions are partially overlapping. On the other hand, all file sizes kept the almost stable throughput rate (IO bandwidth) for reading from and writing to disks, as shown in Figure 3.4. It can be observed that the LFS based system has faster reads as compared to its writes. On the contrary, LFS supported IME system has shown faster writes than its reads.

The application IO time for reading and writing averaged around 73% of the total runtime on LFS based system, as shown in Figure 3.5. Specifically, the writing to disk consumed 70% of the total runtime. Whereas, on the LFS-backed IME system, the IO consumption was 8% or less throughout all the file sizes. Conclusively, the LFS based system application was IO bound and the LFS supported IME system application was CPU bound.

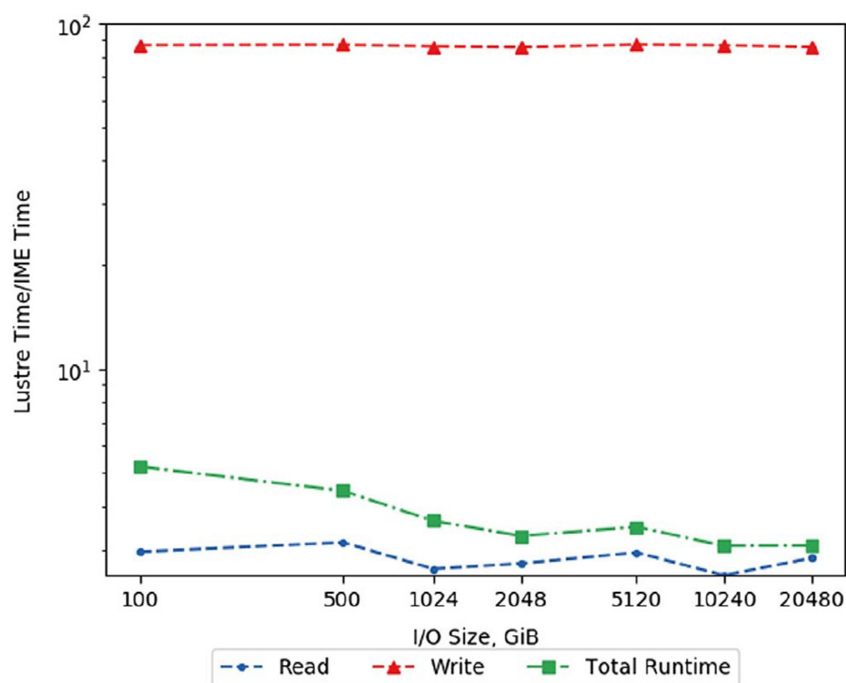


Figure 3.3: LFS runtime and LFS+IME runtime. Reprinted from "ExSeisDat: A set of parallel I/O and workflow libraries for petroleum seismology" by Fisher, M.A., Conbhuí, P.Ó., Brion, C.Ó., Acquaviva, J.T., Delaney, S., O'brien, G.S., Dagg, S., Coomer, J. and Short, R., 2018, *Oil & Gas Science and Technology–Revue d'IFP Energies nouvelles, 73*, p.74 [1].

3.4 Summary

Overall, the ExSeisDat library had yielded a highly significant performance results thus, convincible to industries and geophysicists to use it. The IO or workflows libraries were open to port any pre-existing seismic data analyzing codes into the ExSeisDat system. The Kirchhoff migration utility was ported into ExSeisDat library for benchmarking and performance evaluation. Furthermore, ExSeisDat could give notable benefits regarding

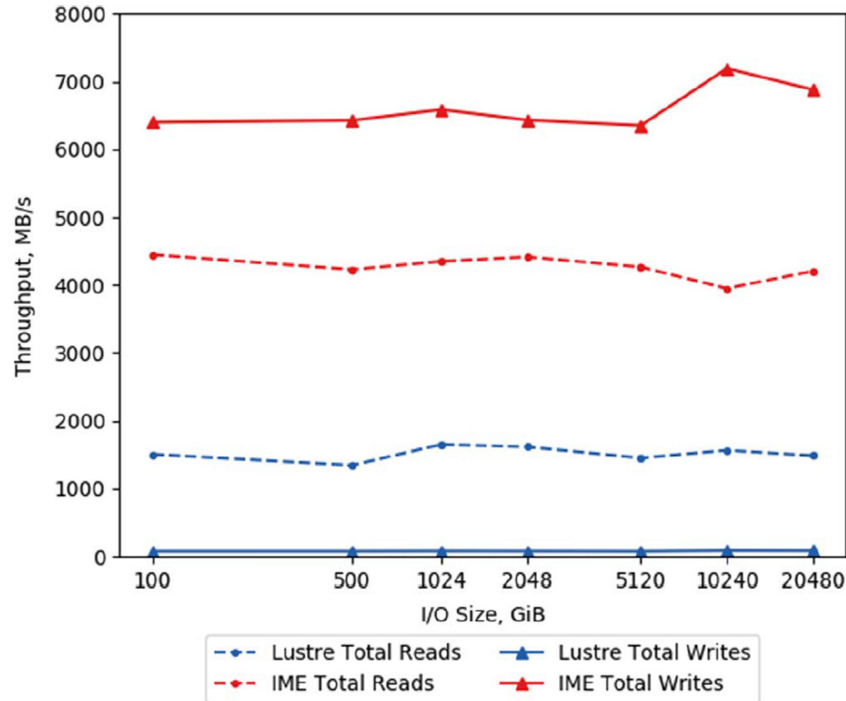


Figure 3.4: Throughput comparison between LFS and IME reads/writes. Reprinted from "ExSeisDat: A set of parallel I/O and workflow libraries for petroleum seismology" by Fisher, M.A., Conbhuí, P.Ó., Brion, C.Ó., Acquaviva, J.T., Delaney, S., O'brien, G.S., Dagg, S., Coomer, J. and Short, R., 2018, *Oil & Gas Science and Technology–Revue d'IFP Energies nouvelles, 73*, p.74 [1].

maintainability, portability and matchable execution time with experts' hand optimized code. The porting of seismic data based utilities and applications into the ExSeisDat could expectedly open the opportunities to introduce various functionalities within the library. Mainly, the actual advantage of ExSeisDat usage is its simplicity to use by geoscience programmers as it provides good level of abstraction and keeping the main code compact, providing highly significant performance at the same time. The approach was to comfort users by hiding the low-level details in relation to storage hardware and software design architecture [95].

The results mentioned earlier via benchmarking shown the ExSeisDat is convenient regarding parallel IO and computations of large sized SEG-Y files from 100 GiB to 20 TiB, with remarkable performance. One of the aims of ExSeisDat was to keep the code short for running common seismic data operations alongside being the high performance application program. The reason was the seismic data files other than the SEG-Y file format could be clearer and well organized therefore, simplicity in code and performance at the same time was kept at the level of easily introducing new seismic formats into ExSeisDat library. Since there were no previous implementations of parallel SEG-Y IO libraries or programs therefore, the comparison among the ExSeisDat and them was not possible.

As the ExSeisDat library in general, do not need the extra tuning at the hardware side therefore, it would expectedly enhance the parallel IO performance further. However, there is always a lot of room for tuning and optimizations from the software side. Considering the case for ExSeisDat managing IO on a cloud-based cluster, it would need extra study some or many steps forward to determine the parallel IO optimization strate-

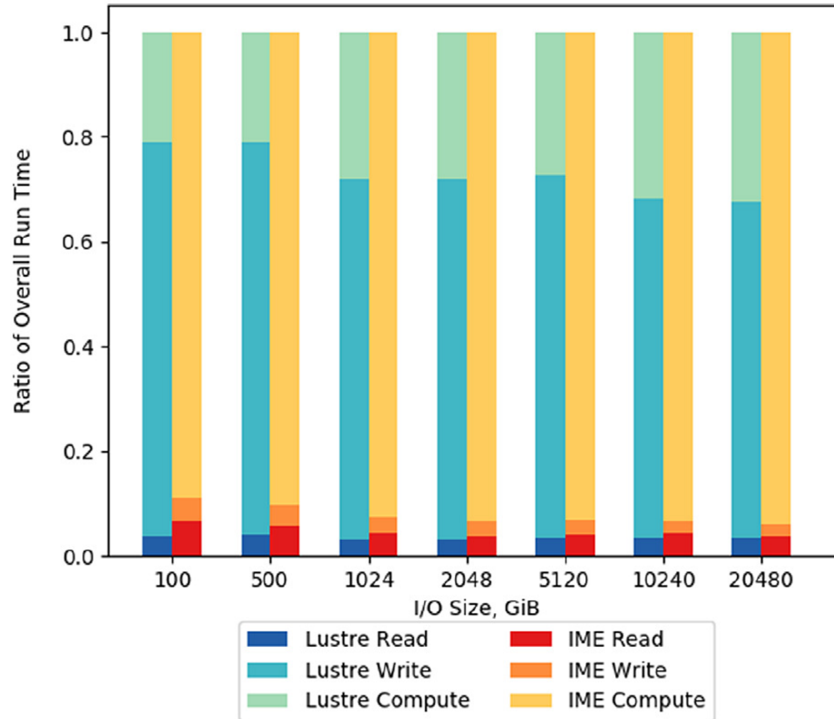


Figure 3.5: Fraction of runtime spent on IO and SEG-Y computations for LFS and LFS-supported IME. Reprinted from "ExSeisDat: A set of parallel I/O and workflow libraries for petroleum seismology" by Fisher, M.A., Conbhuí, P.Ó., Brion, C.Ó., Acquaviva, J.T., Delaney, S., O'brien, G.S., Dagg, S., Coomer, J. and Short, R., 2018, *Oil & Gas Science and Technology–Revue d'IFP Energies nouvelles, 73*, p.74 [1].

gies and techniques for different HPC or cloud cluster systems equipped with the specific networking infrastructure features. Despite this, the performance results stated earlier, also mentioned in [1] should be considered if jobs need to be run on a single compute node with many CPU-cores to read/write from a local disk.

The LFS being employed on the DDN's IME storage system further increased the ExSeisDat IO performance. Specifically, when IME increased the non-contiguous writes throughput, which is a common access pattern in most of the SEG-Y operations in the ExSeisDat. The IME file system played a major role in this performance increase by converting non-contiguous writes into contiguous ones on the disks. These writes were further combined to become largely contiguous for LFS being independent from the program execution. The speedup was expected as less than 3 times however, it was greater due to IME's SSD hardware reading 2.7 to 3 times faster than the HDD hardware. A transformation of sorting functionality from IO bound to compute bound from LFS to LFS+IME system open the opportunities of further computational optimizations i.e., by modification of sorting function. This can make a significant positive effect although, it would affect a bit the overall IO bound application performance which is a usual scenario with the extremely large sized files. It was also worth noting that the 86 times speedup in case of LFS+IME writes was encouraging and favourable for new applications using burst buffer technology to seismic files, in future.

By summarizing the advantages and results for ExSeisDat, it reduced the extra overhead of development by hiding the low-level details of data pattern from the end users by means of abstraction. This was done in parallel with keeping the performance high.

This was presented by the results on the latest architecture of storage hardware. Combining the efficient IO middlewares with high level libraries was seemed to be promising portable approach for further performance gain in HPC application programming rather than extremely hand tuned programming, as presented via ExSeisDat and LFS+IME system.

Chapter 4

Machine Learning

4.1 Introduction

Machine Learning (ML) is the process of enabling a computer system to improve by learning from past experiences and reflect on decision making based on predictions [96]. ML has been rapidly advancing overtime among the interdisciplinary research and development industries [97, 98, 99]. From the technical point of view, ML lies where computer science, artificial intelligence, data science and statistics overlap with each other.

ML is based on a mathematical model, an equation which is tuned and improved over time with different inputs. This usually involves many computations for the model to train [100]. There are already various learning techniques that are data intensive [101, 102, 19, 20]. Therefore, recent research studies have been developing new learning theory and algorithms. This is either with using online or offline available data to reduce the computational cost for some problem areas [103, 104, 105, 106]. Despite the ML techniques being data intensive, they are still being applied to a number of daily real-life problems. This includes technology, health care, commerce, education, manufacturing, policy making, business financial modelling and so on.

To generate a ML model using any learning algorithm, a number of steps must be followed in sequence [107]. The very initial step is to identify the problem that needs to be addressed through prediction or forecasting based on specific input parameters and values. In the ML terminology, the input values are normally called as input feature parameters and predicted estimated values are normally referred to as output feature parameters. Both input and output feature parameters can be single or multiple depending on the nature of the problem. The multiple input feature parameters and single output feature parameter, are common in many cases and scenarios i.e., prediction of house price in a future year depending on area, age, location, etc.

First, the input and output feature parameters are decided according to the given problem statement. Then next step is to prepare the data that can feed into the initial stage ML model equation. The Figure 4.1 shows the basic and common steps to complete and finalize the ML process from data preparation to model testing.

The generation of a dataset is primary component of ML process incase data is not readily available, as depicted in Figure 4.1 [108, 109, 110]. Dataset generation is an important step as the ML model learns and updates itself from data as its experience and becomes trained enough to make future predictions on unseen data. In some problem scenarios, datasets are readily available online or already within an organization however, in some cases datasets must be generated explicitly. Furthermore, despite the datasets are

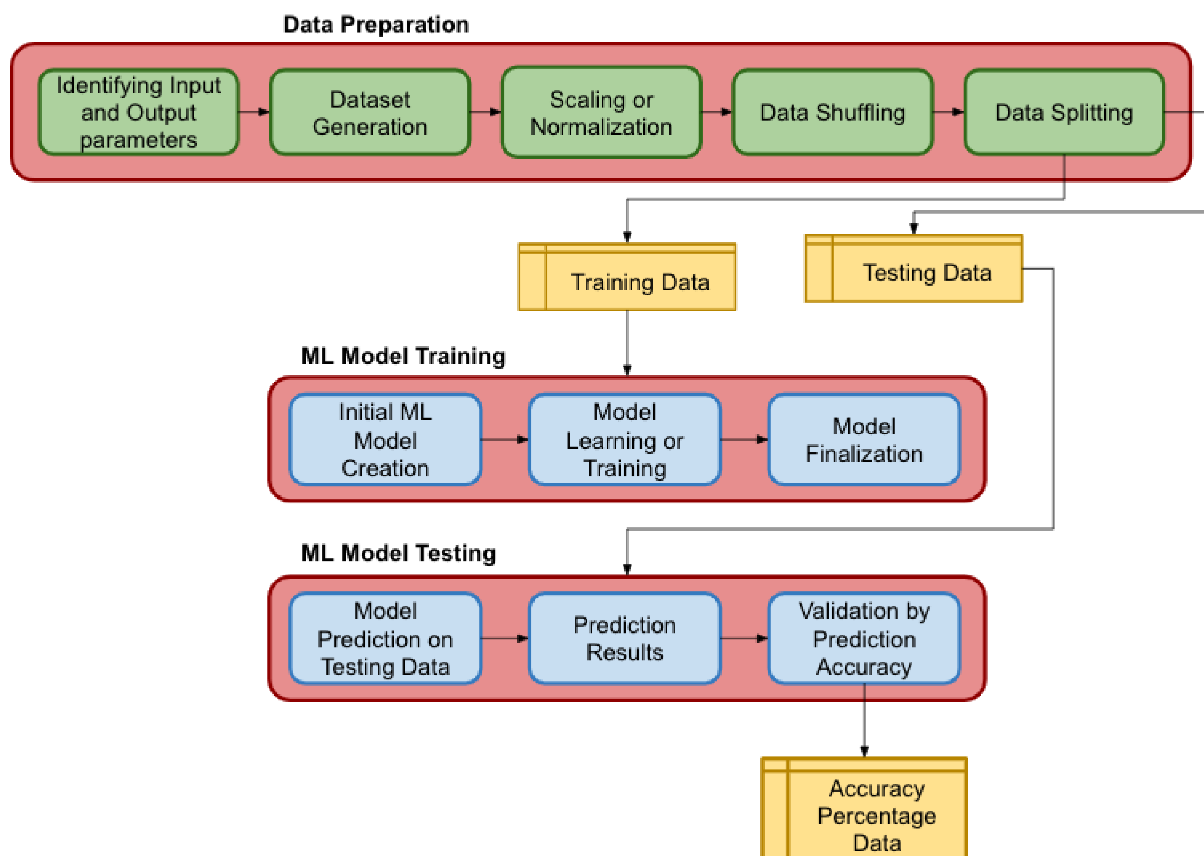


Figure 4.1: Basic Machine Learning Process steps.

generated or in hand already, they would still need additional pre-processing step. This is required to make the dataset values numerically acceptable for mathematical ML model to perform computations on them correctly. The data preparation or pre-processing step is specifically essential to the problems that fall under the supervised learning category as explained later in this chapter.

A most common issue during data preparation is that the fields in a dataset, are not homogeneous [111]. This means the input and output feature attributes have different data types that represent a single row or record of a dataset. In addition to this, sometimes data fields have the same data type values but with different scales. A mathematical model equation requires input and output fields parameters data on a common scale. Therefore, all values in a dataset requires scaling or normalization as next important step, explained later in this chapter. Similarly, the unification of dataset fields into decimal format type makes the processing convenient by a model. As result of this, a model learning is correctly improved according to a specific learning algorithm being used.

The data is usually shuffled prior to normalization or scaling, and then split into two sets: a training set and testing set. The purpose of shuffling the data in a dataset is to reduce the variance or standard deviation among classes and targets. This helps to ensure that a model is well generalized and accurate for predictions against the unseen data in the future. An apparent example would be if the data is sorted by the classes/targets against the parameters' fields within a dataset. In this case, the random shuffling of data provides a generalized representation of the initial distribution of data values in the training and testing sets.

The main step of the ML process is model training [101, 102, 19]. Since the training set can be easily split from a dataset after shuffling, subsequently, it is passed on to a model with its initial state according to a learning algorithm. This is where model training or learning begins. During this process, each data tuple from a training set is passed to the model equation. In each iteration, the model predicts a value which is compared to the original output feature parameter value in terms of computing the error with a specific loss function.

Based on the error loss value, model's weights or variables states are updated to process the next data tuple from a training set in the next iteration. This is normally referred to as gradient descent method which is widely used throughout different learning algorithms in various forms and software tools [112]. It eventually reduces the error and improves the prediction accuracy after several iterations depending on the problem scenario. The gradient here represents the error loss between predicted and original output value. Therefore, if the gradient approaches zero, this means prediction error is reducing. This normally improves the accuracy of a model during training and eventually the predictions over the unseen data in the testing set.

Once the model is finalized, then the testing dataset is passed onto the trained ML model. The model predicts output against each tuple of the testing dataset and subsequently, completes the prediction results. The results are typically validated against the actual output values via computing the prediction accuracy value in terms of percentage. The prediction accuracy shows how well the model is generalized and trained on unseen data. If the percentage value is low, then it means the model is not well generalized which can be a cause of being under-fit or over-fit [113]. However, a high prediction accuracy on the testing set is an indication of a generalized model and is considered as best-fit.

The fit of the model can be analyzed by plotting error loss value with respect to the training and testing set output values against a certain number of iterations [114]. In each iteration, execute the model over the training and testing sets, then compute their respective error loss values. If over the iterations, both the error loss values are approaching zero, and converge to each other and remain constant, it means model is trained, and can predict accurate output values.

Where both the error loss values on training and testing sets diverge or have a significant gap, then it means the model is over-fit. Being over-fit means the model is only accurate for the training set values but not for unseen data. Therefore, it is not generalized and trained well. In case the error loss values over training set are not approaching zero, or increasing, this means a model is under-fit, which is the worst-case scenario. There are several techniques explored in later sections which are useful to avoid under-fitting and over-fitting of a ML model.

This summarizes the overview of the ML process steps. The later sections discuss additional details about the data preparation, regression and classification based predictions, difference between supervised and unsupervised learning, artificial neural networks (ANNs), and then recent research done in the area of optimizing HPC IO using different ML approaches and execution environment levels.

4.2 Data Preparation

The data preparation is one of the preliminary and critical step of a ML process [108, 109, 110]. A dataset provides the basis of experience to a model to learn the pattern and

Table 4.1: Example - Estimation of House Prices.

Input Parameters			Output Parameter
area (ft^2)	location	year	price (€)
2722.51	location A	2020	100,000
2722.51	location A	2021	150,000
2722.51	location B	2020	200,000
2722.51	location B	2021	250,000
5445.01	location A	2020	200,000
5445.01	location A	2021	250,000
5445.01	location B	2020	300,000
5445.01	location B	2021	350,000

predict the output on unseen inputs. Since the data is a source of experience to gain by a model, it is essential to pre-process the data properly according to the needs of a problem. The steps to prepare and pre-process the data are: 1) identifying input and output parameters, 2) data generation, 3) data scaling or normalization, 4) data shuffling and 5) data splitting, as mentioned in Figure 4.1. These steps are further explained in this section.

4.2.1 Identification of Input and Output Feature Parameters

In data preparation, the first step is to identify the input and output feature parameters according to the ML problem [115]. The input feature parameters describe what type of input values will be passed to the model. Whereas, the output feature parameters describe the type of data which will be output by the model as the predicted values. Consider an example of a real time problem to predict the house price based on area, location, and year. In this scenario, the `area` of house, `location` of the house and the `year`, are input to a model which processes these values and gives a `price` of house as a predicted output. Therefore, the `area`, `location` and `year` become the identified input feature parameters. Whereas, the `price` becomes output feature parameter as the predicted value. This is illustrated by example in Table 4.1.

Since the number of input and output feature parameters can be varied from case to case, they can be minimum to one or as maximum as possible. Consider the time series problem of forecasting weather of a region on every next minute or hour. In this scenario, the input parameter is `time` and output parameter is the `temperature` of that region. This type of problems are normally addressed as time series forecasting methods based on ML [116, 117, 118]. This is illustrated by example in Table 4.2.

In contrast, consider the case of predicting CPU time and IO time of a parallel application, based on the number of threads, file size, memory usage. In this scenario, the application `threads`, `file size` and `memory usage`, are input to the model for computing the estimate (CPU time) and the `IO time` as the outputs. Therefore, the `threads`, `file size` and `memory usage` are identified as the input feature parameters. Whereas, `CPU time` and `IO time` are identified as the output feature parameters for prediction. This is illustrated by example in Table 4.3.

In general, the ratio of input to output parameters can be 1-to-1, many-to-1, or many-to-many, depending on the nature of the problem. The identification or selection of the input and output parameters is critical as it defines the problem more transparently.

Table 4.2: Example - Forecasting Temperature on Hourly basis.

Input Parameters	Output Parameter
time (hour)	temperature (°C)
01:00	15
02:00	16
03:00	17
04:00	18
05:00	17
06:00	14
07:00	16

Table 4.3: Example - Estimation of CPU time and IO time.

Input Parameters			Output Parameters	
threads	file size (GiBs)	memory usage (MiBs)	CPU time (s)	IO time (s)
4	2	256	3	4
4	2	512	1.5	3
4	4	256	2	3.5
4	4	512	1	3
8	2	256	1.5	2.5
8	2	512	0.75	2
8	4	256	1	2
8	4	512	0.5	1.5

However, the number of parameters in some problems, can be reduced in the later stages of ML if those parameters are not contributing much to determine the output of the system. This can be done via statistical or data analytical techniques.

4.2.2 Data generation

After the identification of input and output parameters, the next critical step is to generate datasets based on those parameters [119]. Since it is an essential step prior to model training, the data should depict an actual or true representation of the problem. This is because the accuracy of prediction on the future unseen data, relies on a true representation of the initial dataset. Otherwise, fake, or false data can lead to highly inaccurate predictions for future. Some problems require a time consuming training step on very large datasets therefore, it is necessary that training is being done on the data as accurate as possible. After a long training step, if predictions are not precise due to inaccurate datasets, then it can cost time and money depending on the situation.

In some problem scenarios, the data is readily available online, which can be tweaked and adjusted according to the parameters identified. The data may be available across different institutions and organizations for collection. After collection, data can be adjusted according to the requirements i.e., clinical data, government data etc. However, in other scenarios, the data is not available from any source. In such scenarios, the data needs to be gathered via surveys among the individuals or by running the different experimentations according to the domain of the problem. In this case, the outer factors should be reported such as the country where survey is conducted or environment and

equipment details for carrying out experiments generating the data.

Consider the example of runtime prediction of an OpenMP application based on the CPU cores, frequency, threads, cache memory, or any additional related parameters. In this scenario, the number of benchmarks are required to be executed in order to get the initial runtime data against each possible combination of these parameters values. A machine which runs these benchmarks experiments, has its own system specifications in terms of software and hardware, which can be different from another machine environment. For example, 8-cores 3.6 GHz Intel Xeon Gold 5315Y Processor with 12 MB cache memory, Linux Operating System. Therefore, these specifications should be reported as the same benchmarks can produce different runtime results on a different machine.

On the other hand, the experiments related to physics, chemical reactions, or other sciences, can be affected by having the different environment setups. Therefore, they should also be reported. In the later stages of ML, when the training process is executed on a system, its machine or framework specifications should be mentioned. Specially, if the GPUs are involved, which are very common nowadays to accelerate the training step, their specifications should be reported as well. The reporting of every single bit of information related to experiments is always required. This for the reproducibility of the same results in future by different researchers if they are working or extending the same research study.

It is recommended that the data generated should be saved in a file on the disks, in a way that it is clearly readable as represented by examples from Table 4.1 to 4.3. Consequently, it can be easily fetched afterwards into the memory for later use, being in the matrix form. Since the learning algorithms usually involve matrix operations, a generated dataset easily fetchable into the memory, makes the training process convenient in later stage. For this purpose, storing the data in comma-separated values (csv) or extensible markup language (xml) formats, is a common practice in ML community. They can be easily fetched into the matrix memory due to availability of powerful software tools and libraries within the programming languages like Python and MATLAB.

By keeping all these details in view, data generation or already available datasets is a crucial step towards ML. This is because the training process of the model is immensely depending on it. If the data is valid a model will be valid for future unseen data. Otherwise, if the data is invalid or false, then a model will be invalid for true unseen data.

4.2.3 Data Scaling

Scaling of the generated dataset is another important step prior to the training process [111]. It is common that the datasets having values of different input and output parameters at different scales. If one parameter has value 10 the other may have 1000 or 10000 or may be lower as 0.0002. For example, in Table 4.3, the values of all parameters have same scale except `memory usage` due to values range between 256 and 512, inclusively. If the values in datasets are not scaled to same level, then it probably causes model to under-fit as a result of training process.

Sometimes, there can be discrete categorized non-numerical values. For example, in Table 4.1, the `location` parameter has two discrete non-numerical values as 'location A'and 'location B'. In this case, the training cannot work at all. It probably does not execute and throw exception due to being non-numerical values in the dataset, depending on which programming framework is used to generate the ML models. This is because

semantically, it is not possible to perform an arithmetic operation on different types of values at computer hardware architecture level.

If the meaning of result of arithmetic operation on two different data types, is explicitly defined at software functionality like an operator overloading, the result is neither vague nor an error. For example, an operator overloading for summation of two string values can be defined as the concatenation of two strings. However, this still cannot benefit the training process at all because it completely relies on complex matrix decimal floating-point operations. Therefore, all input and output parameters values are supposed to be in the same scale.

A common solution to this problem is scaling or normalizing the values of all input and output parameters between 0 and 1. This means all values to become decimal values between 0 and 1 regardless of being originally discrete integers values having different scales. The two most commonly used scaling methods are: 1) `MaxAbsScaler` and 2) `MinMaxScaler` [111]. They are also called as normalization methods and either of them can be used to scale the values. Following are the equations that can be applied for scaling to each column or parameter of the dataset:

$$(MaxAbsScaler)\ x' = \frac{x}{x_{max}} \tag{4.1}$$

$$(MinMaxScaler)\ x' = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{4.2}$$

where x is the original (unscaled) value, x' is the new scaled value, x_{min} is the minimum value in a particular column against a specific column in the dataset, and similarly, x_{max} is the maximum value.

For categorical values, they can be explicitly set between 0 and 1, inclusively or not. For example, as there are only two categories for `location` {'location A','location B'} in the Table 4.1, they can be set as {0.0,1.0}. If there are more than two category values in the parameter column then, they can be set between 0 and 1 with equal difference from the previous corresponding category. For example, if there are three distinct categories existing in a column: {'A','B','C'} then, they can be set as: {0.0,0.5,1.0}. Similarly, if there are four categories: {'A','B','C','D'} then, they either can be set as: {0.25,0.50,0.75,1.00} or: {0.00,0.25,0.50,0.75}. Alternatively, these categories can be first set numerically in order: {1,2,3,4}, and then `MaxAbsScaler` can be applied which results in: {0.25,0.50,0.75,1.00}. If MinMaxScaler is applied then it results in {0.00,0.33,0.67,1.00}. Therefore, either of the set of values can be used as scaled values.

This summarizes the method to scale the data in same range, which is favourable for a smooth training process of the model.

4.2.4 Data Shuffling

The shuffling of data also plays an important role to avoid the incorrect model training [120]. It is possible that the datasets have rows placed in a sorted order with respect to any column index. In this case, when a dataset is split into a training and testing set in the next stage, the model trained on the dataset may not be a complete representation of the original dataset. This normally leads to an over-fitting of the model which shows the high prediction accuracy on training set and the low prediction accuracy on testing set during validation of the model. To avoid this situation, data is usually shuffled to give the overall representation of the dataset to the training of the model.

If programming in Python, the `pandas` module give extensive APIs for data manipulation, including sampling functionality to shuffle the data [121]. By using the `sample()` function of the module, the data can be shuffled mainly with respect to either number of rows or the fraction of rows of the dataset.

For example, consider the house prices data in the Table 4.1 to be saved in a CSV format file "house-prices.csv" on disk. First, a data frame is needed to be created by loading all the rows from a file. This must be done prior to calling any data manipulation functionality. Therefore, first two instructions can be as follows:

```
import pandas
data_frame = pandas.read_csv("house-prices.csv"),
```

where `import` keyword is used to import all objects and functions of specified module `pandas`. Additionally, the function `read_csv()` is used to load the data rows from the CSV file into data frame memory specified as `data_frame`. Then another data frame can be made where the new shuffled data can be placed. Subsequently, the next instruction would be as follows:

```
data_shuffled = data_frame.sample(n=8),
```

where `sample(n=8)` shuffles all the data in `data_frame` of `n` rows which is 8 since being the total rows of house prices data. Then moves it into another frame which is specified as `data_shuffled`. This is a flexibility of `sample()` functionality that any number of data rows specified by its argument `n`, can be shuffled out of the original dataset and moved into another data frame.

Similarly, instead of specifying rows to be shuffled, the fraction of dataset rows can be specified for shuffling. Since, all the rows are supposed to be shuffled, the alternative instruction can be as follows:

```
data_shuffled = data_frame.sample(frac=1),
```

where fraction is specified as 1 in `frac=1`. This instruction also shuffles all the data in the data frame. The Table 4.4 and 4.5 depict the two different sample outputs of the shuffled house prices data from two different instructions. The data is shuffled, which is evident from the unsorted row numbers specified by first column `row#`. However, repeated execution of same instruction can give different output. It should be noted that the data scaling and shuffling can be done alternatively in sequence as it does not affect the results. This summarizes the data shuffling procedure.

Table 4.4: Sample 1 of shuffled House Prices data by rows.

row#	area (ft^2)	location	year	price (€)
4	5445.01	location A	2020	200000
3	2722.51	location B	2021	250000
5	5445.01	location A	2021	250000
2	2722.51	location B	2020	200000
6	5445.01	location B	2020	300000
1	2722.51	location A	2021	150000
0	2722.51	location A	2020	100000
7	5445.01	location B	2021	350000

Table 4.5: Sample 2 of shuffled House Prices data by fraction.

row#	area (ft^2)	location	year	price (€)
4	5445.01	location A	2020	200000
0	2722.51	location A	2020	100000
1	2722.51	location A	2021	150000
6	5445.01	location B	2020	300000
3	2722.51	location B	2021	250000
7	5445.01	location B	2021	350000
2	2722.51	location B	2020	200000
5	5445.01	location A	2021	250000

4.2.5 Data Splitting

The next step is to select the training and testing rate of the model over the main dataset [122]. This can be done usually by splitting the data in two different sets with different percentage of the original dataset. At this stage, it is usually assumed that the data is already shuffled such that when it is split, both the training and testing sets have overall representation of the original dataset. The only concern here is the selection of percentage rate for partitioning the sizes of training and testing sets, to split them from the main dataset.

There is no general rule for selecting the training and testing rate since every problem has its own number input and output parameters and its dataset. However, if according to the number of parameters a dataset is covering most of the values combinations and sufficiently large, then 70-30% or 80-20% of training/testing ratios can be used. Since the training must be fed with the data as large as possible, the 70% or 80% of the dataset usually contains the overall representation of the data. In some scenarios, the ratios can be 50-50% where the number of data rows are extremely greater and are expected to have the overall representation in half of the dataset.

In the scenarios where the size of the dataset is too small, then cross-validation is usually recommended and applied to train the model [123]. It is normally referred to as `k-folds` cross-validation. In this procedure, the training set is selected from 70% to 90% of the dataset. Then within the training set, the data rows are split into `k` number of foldings for validation as illustrated in Figure 4.2. For each `k` folding for validation the remaining data rows are used for training. Consequently, the model is trained and validated by `k` number of times from the scratch. The model giving maximum accuracy or minimum error on validation of folding, can be chosen for testing on the remaining 10% to 30% of the dataset. It is also possible that a model giving less accuracy on a validation folding data rows, gives more accuracy on the testing set as compared to other models. Therefore, the best one according to these cases can be selected for further predictions on future unseen data.

It should be noted that the data splitting, training, validation, and testing are executing back and forth during the whole process of cross-validation. However, in the scenarios where cross-validation is not required, the training and testing is performed in sequence once the dataset is split by the required ratios. This summarizes the overview of data splitting.

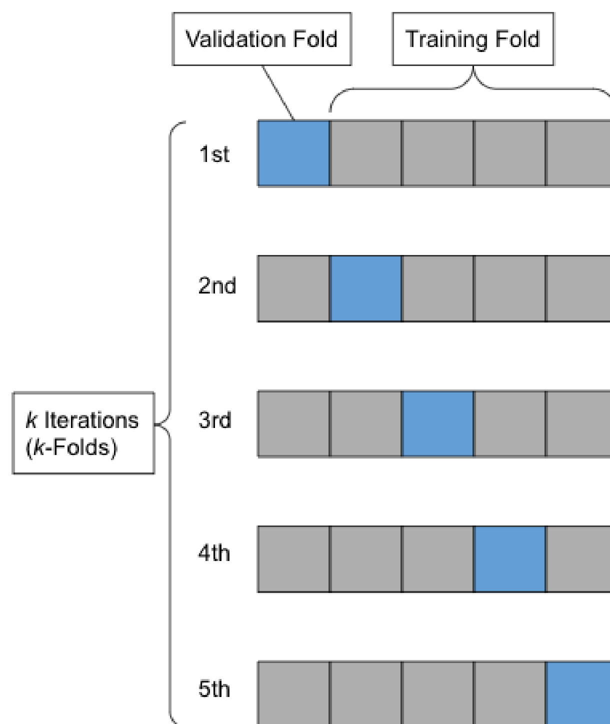


Figure 4.2: k-Folds Cross-Validation.

4.3 Supervised vs. Unsupervised Learning

In this section, two different types of ML approaches: supervised and unsupervised learning, are discussed. The training through both approaches is different than each other along with the requirements and conditions of the training dataset. Since both approaches have different weaknesses and strengths, the problem addressed by both learning models are normally different [124]. When a company intends to deploy a ML model, the selection of a specific learning algorithm is carried out. This selection is usually by analyzing the available data and a problem that must be addressed. In this case, the main differences between supervised vs unsupervised ML techniques should be understood that which approach can be correctly utilized to solve a problem.

4.3.1 Supervised Learning

The supervised ML processes evolve around training on the data which has labelled input and output features, as the few examples stated in earlier sections [125]. Since the labelling of data is important, it is mostly done by a person preparing the data during the dataset generation phase before training the model. When a model learns the relationship between the input and output labelled data, then it can be used to predict or classify the output on the given unseen future input data.

This process is called supervised learning because it requires human interference to convert the available raw data into the labelled data so, it can be processed properly. In other words, a human supervision is required over data to ensure the meaning of the data. The data should be labelled accordingly to exhibit its meaning. For example, `area`, `location` and `year` are input values to predict `price` of a house as an output value, as stated earlier. Additionally, learning on the labelled data can be compute-intensive if the

training dataset is large.

The supervised learning is usually used for; classification of file formatting types that represent words, documents, or images, and predicting the regression based output by learning the changing patterns in the training dataset. The examples for supervised learning based classification can include (but not limited to) the following problems:

- spam email detection as the firewall's utility.
- identification of objects within an image of a specific file format.
- face and speech recognition systems.
- classifying text or numbers written in different writing styles.
- Human sentiment analysis from the written text messages.

To address these problems, they first need to identify which type of classification problem it points to. Such classification problems can be any of three possible: binary classification, multiple class classification and multiple label classification. The multiple label classification can employ binary and multiple class classification on different multiple output features.

In binary classification problem, a model can predict output label feature with only two classes. For example, an email can be either “spam” or “not spam”. The typical approaches to binary classification based ML models are: Logistic Regression, Decision Trees, Naïve Bayes, etc [102, 126, 127].

On the other hand, the multiple class classification model can predict output label feature with multiple classes identified for a problem. An example can be the recognition of a digit written in any type of style. The typical ML approaches to such problems can be: Random Forest, k-Nearest Neighbours, Naïve Bayes, etc [128, 129, 127].

With respect to multiple label classification, a model predicts classes on multiple output label features. For example, classifying multiple objects within an image. The approaches to this type of problems, are commonly addressed by the ML techniques such as: Multiple Label Gradient Boosting, Multiple Label Random Forests, Multiple Label Logistic Regressions, or different classification approaches can be used on different labels [130, 131, 132].

Supervised learning based regression models are used to predict the outcomes of the input data, in the form of numeric values. The example problems for regression based on supervised learning can be (but not limited to) as follows:

- forecasting trading outcomes or stocks, which is a vital part of ML in finance.
- predicting the success rate of organizations for campaigns in relation to marketing.
- forecasting house prices in future.
- weather predictions in regard to forecast the temperature, wind speed and atmosphere in the next hours or days.
- predictions of changing trends of health in a specific region.

The typical regression techniques in supervised learning to address these kinds of problems are: Linear Regression and Decision Tree Regression. Linear Regression is one of the famous regression techniques as stated and explained earlier, which is used to predict the desired output from a single or multiple input feature variables or parameters. The model is trained via gradient descent and then can be used for prediction. Another example can be the prediction of salary based on gender and age.

Then Decision Tree Regression technique usually models the tree structure where branches are incremented by time during training [133]. Since it is a popular approach in the domain of supervised learning, it is normally used for both classification and regression. The dataset in this approach is spilt into multiple incremental subsets to learn the relation among multiple independent variables. This summarizes the basis of supervised learning.

4.3.2 Unsupervised Learning

Unsupervised learning is a group of ML techniques which train models on unlabelled or raw training data [134]. It is normally used for identifying trends and patterns of raw data, or more specifically, it is used for clustering the similar data into a number of specific groups. It is even used for understanding datasets gathered from an early exploration of experiments.

It can be noticed from the name that unsupervised learning requires no human interaction to pre-process or organize the data in a labelled dataset form. The actual purpose of unsupervised learning is to process huge arrays of raw or unlabelled data. This is opposite to the case of supervised learning. However, a small human interference is still required to set the model's hyper-parameters i.e., learning rate, the number of cluster points, etc. Since a limited human interference is needed, a clearer explanation is required in understanding the decisions of the model's output on the given input. Therefore, it is a well-suited technique for answering questions to unseen relationships or trends within datasets.

A large chunk of the data around the world is raw and unlabelled data. Therefore, unsupervised learning becomes a powerful tool when it comes to gaining insights to the data. This relates to the grouping of similar featured datasets, understanding the trends and patterns within a dataset. Whereas, in contrast, supervised learning can require extra memory and be more computationally intensive since it requires pre-processing and labelling of the data. Unsupervised learning is normally used for clustering datasets based on similarities in segmentation of data and some features. Additionally, it is used for understanding relations between various data points and analyzing initial datasets. The following example problems are usually addressed by the unsupervised learning:

- partitioning clusters of customers and audience working in marketing related environment.
- performing the initial data analysis on newly discovered raw datasets, for understanding the groups among different data points.
- detection of anomalies and outliers outside the clustered data.

The common unsupervised learning techniques to address these kinds of problems are K-means clustering and Gaussian Mixture Models. The K-means clustering is the famous and commonly used approach to cluster data into different groups [135, 136, 137].

`K` denotes the count of groups or clusters, which can be set by the programmers. The clusters in this approach, are determined from the centre-to-centre of each data group. This means if the cluster count is high then there is a high granularity of groups, and low cluster count means the low granularity of groups. This approach can recognize both exclusive and overlapping clusters. The exclusive clusters mean that a data point can exist in only one cluster at the moment. Whereas the overlapping clusters mean that data point can exist in more than on clusters at an instance.

In the Gaussian Mixture Models approach, the clusters are determined by the probability of data points belonging to a certain group [138, 139, 140]. This is called as probabilistic clustering approach. This method uses probability to map data points into a group or cluster, which contrasts with K-means clustering which determines the clusters by the distance between the centres of the groups.

The other class of examples in unsupervised learning, are based on association rules. Association is the identification of the relations among different features, to analyze how one data point feature is related with the other features. This is to map the relations among the different data points. The famous approach to form the association rules is a method called the Apriori algorithm. This algorithm identifies different trends within a database based on the frequency of trends [141]. The parallel processing based variants of the Apriori algorithm also exist for large scale data [142, 143]. The following problem examples require such method to form association rules:

- Recommendations of products or services to users, according to the habits of purchasing or buying.
- Recommendations of songs, movies or TV-series, based on the users interests i.e. Netflix, etc.
- Understanding the interests and habits of e-commerce customers, in regards to informing them about the marketing campaigns or e-commerce.

These problems can be addressed by the unsupervised learning algorithms stated earlier. This summarizes the overview of unsupervised learning.

4.4 Model Training

In this section, the two basic types of ML techniques are discussed for model training that corresponds to regression and classification problems, respectively [101, 102]. The techniques elaborated are supervised learning algorithms since they operate on specific input and output parameters. For regression problems, the Linear Regression, and for classification problems, the Logistic Regression, are explained as the supervised ML techniques. This is regarding the sequence to train the model which can be then tested on unseen data in later. The example problems illustrated in this section are the ones with multiple input feature parameters and a single output feature parameter.

4.4.1 Linear Regression (`for Regression problems`)

In regression problems, a numerical value is expected as the predicted output value based on the hypothesis of the model on its input parameters. Consider a case, where `n` input features exist with 1 output feature having `m` rows of training dataset. The Linear

Regression is a very basic ML technique to train the model and predict the output for future data [101].

The model hypothesis equation for Linear Regression is as follows:

$$h(\theta, x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n, \tag{4.3}$$

where x denotes the input parameter, and θ is the initial coefficient variable that determines the accumulated predicted output value of the model hypothesis, as represented by $h(\theta, x)$.

The summation of all the terms from 0 to n in the hypothesis equation, estimate the prediction value. It should be noted that the actual parameters are from x_1 to x_n, and x_0 is intuitively added with value 1 as the θ_0 is a bias value of the model. The values of θ_0 to θ_n update throughout the training iterations. Additionally, this equation of hypothesis can be presented in the form of vectors or matrices. Consider θ and x as the following vectors:

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix} \ and\ x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \tag{4.4}$$

then in the matrix-vector form the equation is as follows:

$$h(\theta, x) = \theta^T x = \begin{bmatrix} \theta_0 & \theta_1 & \theta_2 & ... & \theta_n \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \tag{4.5}$$

where θ^T is the transpose of the vector θ, and hence, the new hypothesis matrix equation $\theta^T x$ is equivalent to the stated equation 4.1.

The main task of training is to update the values of θ from the iterations in sequence. When, after the iterations, final values of θ are multiplied with corresponding input values of x, their accumulated sum should give the accurate prediction output value. The most popular approach to update θ or weights in different ML techniques is the gradient descent method [112, 144]. The following equation represents the gradient descent approach to update the values of θ as part of training of the model:

$$\theta_j \Leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, ..., \theta_n),\ for\ j \in 0, 1, 2, ..., n, \tag{4.6}$$

where α is the learning rate of the model and $J(\theta_0, ..., \theta_n)$ is the error cost function. This equation should be executed with repeated iterations for every weight coefficient θ_j that corresponds to x_j input feature parameter, till the value of θ_j is converged. Probably, in the last few iterations the values of each θ_j would be closely similar. This means θ values are converging to the global minimum point of cost function curve where its gradient is zero. Therefore, the iterations are supposed to be stopped at this point, and the model is now trained enough. These iterations that update the values of θ through gradient descent, actually train the model.

A gradient descent equation updates the values of θ in each iteration by subtracting the product of learning rate α and partial derivative of the cost function $J(\theta)$. Since these values are used in the gradient descent equation, the equation of the cost function and its partial derivative can be analyzed intuitively. The cost function is as follows:

$$J(\theta_0, \dots, \theta_n) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)^2, \tag{4.7}$$

where n and m are number of input features and training examples, respectively, as stated earlier. Then i is the ith training example therefore, x_i is a set of input features in an ith training example. Furthermore, y_i is its corresponding actual output value, and $h_\theta(x_i)$ is the corresponding predicted hypothesis value. Since the partial derivative of the cost function is required in the gradient descent based training, intuitively and mathematically using calculus, the equation of its partial derivative is as follows:

$$\frac{\partial}{\partial \theta_j} J(\theta_0, \dots, \theta_n) = \frac{1}{m} \sum_{i=1}^{m} (h(\theta, x_i) - y_i) x_{ij}, \tag{4.8}$$

where x_{ij} is the jth input feature within a set of ith input values in a training example x_i. Remaining details are same as stated earlier. It can be noticed that the cost function is the computational error cost against all input features and corresponding θ variables associated with the problem model. Since the function has $n + 1$ dimensions of θ, it can be represented as $n + 1$ dimensional linear or non-linear curve.

The partial derivative regarding specific values of θ gives the gradient of the cost function curve. This is done by computing difference between hypothesis value with x_i input feature values and an actual output value y_i in a training set example, with respect to specific input feature x_{ij}.

The key point is to descend or decrease the gradient towards 0 which is a global minimum point for all values of θ in each iteration of this training process. This is done when in each iteration the value of partial derivative multiplied with the learning rate α is subtracted from the current values of θ. As the iterations run, the values of θ are supposed to be getting closer to global minimum point. It may not reach at exact global point. However, if the values of θ are not changing much in the last few iterations that means the values are converged at point closer to global minimum gradient. Hence, a model can be considered as trained and the iterations can be stopped.

The speed of this training process via gradient descent can be altered by changing the value of α since it directly represents the learning rate of the model, as stated earlier. Generally, by increasing the value of α, the system can reach the global minimum point faster than by decreasing the value of α. This means it can take a smaller number of iterations to train the model as compared to training with the lesser value of α. However, the non-realistic increase in the value of α can lead the system to towards divergence from global minimum and in θ values, after being converged. For example, the previous value of α was 0.001 and the current learning rate is set to 100. This may further increase the time of training or iterations, as compared to smaller value of α. Therefore, the value of the learning rate α should be realistically increased to reduce the training time. For instance, α can be increased to 0.002, 0.004 or similar values, from 0.001.

The alternate way to minimize the cost function $J(\theta)$ value to its minimum optimal value, is the Normal Equation [145]. This updates the values of θ variables to the global minimum point of cost function. Since the gradient descent method can take a large

number of iterations as they are not fixed, the Normal Equation finds the optimal value of θ in a fixed number of iterations. This is with respect to m training examples and n input features, involving Matrix-Multiplication operation as the equation follows as:

$$\theta = (X^T X)^{-1} X^T y, \tag{4.9}$$

where X is matrix representing n input features values as columns in m training examples as rows therefore, X is $m \times n$ matrix, more accurately it is $X_{m\times n}$ matrix. Then, X^T is the transpose matrix of X so, it is $X^T_{n\times m}$ matrix. The term $(X^T X)^{-1}$ represents matrix-multiplication of X^T and X, and its inverse so this, has the complexity of $O(n \times m \times n)$. Finally, y is a vector of actual output feature values corresponding to their input feature values in X from m training examples.

The benefits of the Normal Equation are as follows:

1. no data scaling is required,
2. no selection of the learning rate α is required, and
3. no iterations are needed as in case of gradient descent.

However, there are some disadvantages as well, which are as follows:

1. it can be slower if the training set m and input features n are large,
2. it has greater time-complexity than gradient descent, due to matrix-multiplication, and
3. the resultant matrix of $X^T X$ can be non-invertible if there are redundant or closely related input features, or if the number of input features are greater than the number of training examples m (i.e. if $m \geq n$).

The non-invertibility can be simply avoided by removing extra redundant features and adding more training examples in a training set such that m becomes greater than n. This summarizes the use of Linear Regression based ML technique for training the model, which is used in previous research works [146, 147, 148, 149].

4.4.2 Logistic Regression (`for Classification problems`)

A categorical class is used to represent a specific group of data. For example, data regarding the human race can be specified by different groups of categories: Asian or African or European. Similarly, gender can be specified as male or female. This means an information like race or gender can be classified by the category it lies in.

In classification problems, a categorical class value is expected as the predicted output value based on the hypothesis of the model on its input parameters. Consider the case of binary class classification as an output `0` or `1` based on `n` parameters with `m` training examples. The very basic ML approach is the Logistic Regression to train the model and predict class of output for future input data [102, 144].

The model hypothesis equation for Logistic Regression is as follows:

$$h(\theta, x) = g(\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n) \tag{4.10}$$

$$or\ h(\theta, x) = g(\theta^T x)\ by\ \theta\ and\ x\ as\ matrices, \tag{4.11}$$

where $g(\theta^T x)$ is an additional sigmoid function over $\theta^T x$ computation [150]. The sigmoid function convert an any arbitrary regression value into a decimal number between the range of $(0.0, 1.0)$. This is a probability value well suited for classification of the output value. It is represented by the following equation:

$$g(\theta^T x) = \frac{1}{1 + e^{-(\theta^T x)}}, \tag{4.12}$$

where e is the exponential constant with already determined value 2.718. Considering this equation, the three distinct probabilities from the hypothesis in range $(0.0, 1.0)$ are as follows:

$$h(\theta, x) = g(\theta^T x) = 1.0 \ if \ \theta^T x \to \infty, e^{-\infty} \to 0, \tag{4.13}$$

$$h(\theta, x) = g(\theta^T x) = 0.5 \ if \ \theta^T x = 0, e^{0} = 1, \tag{4.14}$$

$$h(\theta, x) = g(\theta^T x) = 0.0 \ if \ \theta^T x \to -\infty, e^{\infty} \to \infty, \tag{4.15}$$

where it means that intuitively for discrete value output $(0, 1)$, if hypothesis $h(\theta, x)$ is greater than or equal to 0.5, the output is 1, otherwise 0. Implicitly, it also leads to the conclusion that if the value of $\theta^T x$ is greater than or equal to 0 then $h(\theta, x)$ or $g(\theta^T x)$ is greater than or equal to 0.5, and the output is 1. Otherwise, if $\theta^T x$ is less than 0 then $h(\theta, x) org(\theta^T x)$ is less than 0.5, and the output is 0, as presented in the following equations:

$$\theta^T x \geq 0 \Rightarrow h(\theta, x) \ or \ g(\theta^T x) \geq 0.5 \Rightarrow y' = 1, \tag{4.16}$$

$$\theta^T x < 0 \Rightarrow h(\theta, x) \ or \ g(\theta^T x) < 0.5 \Rightarrow y' = 0, \tag{4.17}$$

A probability from the hypothesis function determines the class output against the given input feature values. By example, if the $h(\theta, x) = 0.8$ then the class output is 1 because its probability is greater than 0.5, as a general rule. This means the probability of class output is not 0, is 0.8, in case of binary class classification therefore, probability of class output being 0, is 0.2. The sum of probabilities for both classes is equal to 1. The hypothesis equation can be presented in the form of probability equation as follows:

$$h(\theta, x) = P(y' = 1) = 1 - P(y' = 0), \tag{4.18}$$

$$and \ 1.0 = P(y' = 1) + P(y' = 0), \tag{4.19}$$

where P is probability over y' predicted output as 1 or 0. However, the probability equation is different for the multi class classification case.

In Linear or Non-linear Regression modelling the hypothesis is the straight or curved line giving the predicted output values over different input features. In contrast, the Logistic Regression line or curve of the model hypothesis is the decision boundary to classify the output values. Consider a model with two input features and one extra feature with value always 1. Assume $x = (x_0 = 1, x_1, x_2)$, and $\theta = (4, 3, 2)$. Now, for output $y' = 1$, the value of $\theta^T x$ should be greater than or equal to 0. This leads to the following equations:

$$4 + 3x_1 + 2x_2 \geq 0, \tag{4.20}$$

$$3x_1 + 2x_2 \geq -4, \tag{4.21}$$

$$x_2 \geq -\frac{3x_1}{2} - 4. \tag{4.22}$$

By imagining a graph relation between x_1 on horizontal axis and x_2 on vertical axis, this equation represents a straight line as decision boundary with a gradient $-3/2$, x_1 intercept at $-4/3$ and x_2 intercept at -4. Since the lefthand side of the equation is greater than or equal to righthand side, the region above this decision boundary line presents the output y' classified as 1. Under this line, the output y' is classified as 0. It should be noted that in different problem scenarios the decision boundary line does not has to be necessarily a linear straight line. It can be any type of non-linear curve separating different classes.

Moving towards the training part of Logistic Regression model, it is like the Linear Regression since the gradient descent method is general in both techniques. However, the hypothesis and error cost function are different but ultimately the partial derivative becomes closely similar as in Linear Regression case. Hence, the Logistic Regression model can be trained using the general gradient descent method. The following equation presents the cost function for logistic:

$$costfunc(h(\theta, x), y) = -y \ln(h(\theta, x)) - (1 - y) \ln(1 - h(\theta, x)), \tag{4.23}$$

where $\ln$ is natural logarithm function, whose base is the exponent constant e, and can be presented as $\log_e$. The $h(\theta, x)$ is hypothesis of model to output predicted class as stated earlier, and y is actual output class value against a training example. Since this equation has two terms determining the result, it should be noted that if y is 1 then the second term $(1 - y) \ln(1 - h(\theta, x))$ is 0, otherwise, if y is 0 then the first term $-y \ln(h(\theta, x))$ is 0. In both cases the net result cannot be affected. The equation of this cost function is for only one training example. By re-writing it for m training examples and $n + 1$ coefficient variables $(\theta_0, \dots, \theta_n)$, this equation is as follows:

$$J(\theta_0, \dots, \theta_n) = -\frac{1}{m} \sum_{i=1}^{m} (y_i \ln(h(\theta, x_i)) + (1 - y_i) \ln(1 - h(\theta, x_i))), \tag{4.24}$$

where $J(\theta_0, \dots, \theta_n)$ represents cost function just like previously in case of Linear Regression, m denotes the total number of training examples, n is number of input features, and i denotes the ith number of an example in the training set. Hence, x_i and y_i represents input feature values and output feature values set in an ith examples of the training set. Despite the cost function of Logistic Regression is different from the Linear Regression, its partial derivative with respect to different θ values, is the same, by using calculus techniques. Therefore, the resultant equation for training by gradient descent is also the same as follows:

$$\theta_j \Leftarrow \theta_j - \frac{\alpha}{m} \sum_{i=1}^{m} (h(\theta, x_i) - y_i) x_{ij} \ for\ j \in 0, 1, 2, \dots, n, \tag{4.25}$$

Similarly, this process must be repeated unless all the values of θ are converged closer to global minimum cost. Afterwards, the trained model hypothesis with these optimal θ values can be used to classify the predicted output against the given input features values. This equation can be converted into vectorized, or matrix form as follows:

$$\theta \Leftarrow \theta - \frac{\alpha}{m} X^T (h(\theta, X) - y), \tag{4.26}$$

$$where\ h(\theta, X) = g(X\theta). \tag{4.27}$$

In these equations, X denotes matrix containing m rows of training examples with $n + 1$ columns of input features values therefore, X^T is its transpose matrix. Then y is m rows

of vector containing actual output feature classified values corresponding to each example row of matrix X. Furthermore, θ is $n+1$ rows of vector to be multiplied with X matrix to complete the hypothesis and predict the class output. The application of the logistic regression is used in couple of previous research works [151, 152, 153, 154].

4.4.3 Multi-class Classification

In the multi-class classification problem, the multiple or more than two label categories exist as classes to classify the output of the hypothesis over given input [155, 144]. Therefore, rather than having two discrete output $y = (0, 1)$ classes, there are now $k+1$ discrete output classes to classify on a given input such that $y = (0, 1, 2, ..., k)$.

The main approach is to divide the $k+1$ problems into $k+1$ binary classification problems. Each problem predicts the probability of y belongs to one of the class from $k+1$ classes $(0, 1, 2, ..., k)$. Then one predicting the maximum probability of a class from $k+1$ classes is the classified output of the given input. Intuitively, this process can be represented by the following equations:

$$
\begin{aligned}
y &\in 0, 1, 2 ..., k, \\
h_0(\theta, x) &= P(y = 0), \qquad (4.28) \\
h_1(\theta, x) &= P(y = 1), \\
h_2(\theta, x) &= P(y = 2), \\
&... \\
h_k(\theta, x) &= P(y = k),
\end{aligned}
$$

$$Predicted\ Output = max_{c=0}^{k}(h_c(\theta, x)), \qquad (4.29)$$

where c is a counter for a discrete output class from 0 to k. In this mechanism, each hypothesis is selecting one class while considering all the remaining classes as a single different class, in terms of probability, by using Logistic Regression. When all the hypotheses' probabilities are computed for all the classes, then the one giving the maximum predicted probability of a class is the chosen class output.

In order to train the model for multi-class classification problem, a Logistic Regression binary classifier $h(\theta, x)$ should be trained. This should be executed for each class to predict the probability $P(y = c)$ of that class, as mentioned in the last section. It should be noted that since the Logistic Regression model must be trained for all the $k+1$ classes, the values of θ parameters will be different for each c class hypothesis. Therefore, the values of θ parameters are supposed to be maintained carefully and separately for each class c, from the perspective of programming code logic. Once the model is trained for all the classes then an unseen input can be given to the model to predict the probability of each class. The output class is the one having maximum probability for a given input where it most probably lies. This is worked out by using the equations 4.28 and 4.29. This summarizes the use and application of multi-class classification which exists in many previous research studies [156, 157, 158, 159].

4.4.4 Regularization - a solution to Overfitting

The problem of overfitting is common when training a model. In general, overfitting is spotted when a trained model gives a high prediction accuracy on the training dataset and low prediction accuracy on the testing dataset. This problem can occur on both regression

and classification problem, which means it can be the case for both Linear Regression and Logistic Regression models. There are mainly two choices to overcome this issue. One is to reduce the number of features by manually selecting the features to keep or use the model selection algorithm via cross-validation as stated earlier [123]. The second choice is to use the regularization method which implicitly means to reduce the magnitude θ parameters values, without reducing the number of original input features [160]. The regularization method is useful even when the problem contains many redundant or barely useful input features.

By adding the regularization parameter lambda λ in the cost function, it determines the inflation over the costs of θ values. By including the extra summation function over θ values and λ parameter, within the cost function, can decrease the overfitting in the model. The following two cost function equations can be analyzed for Linear and Logistic Regression, respectively:

$$J_{Linear}(\theta_0, \dots, \theta_n) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right], \tag{4.30}$$

$$J_{Logistic}(\theta_0, \dots, \theta_n) = -\frac{1}{m}\sum_{i=1}^{m}\left[(y_i \ln(h(\theta, x_i)) + (1 - y_i)\ln(1 - h(\theta, x_i)))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2. \tag{4.31}$$

The second summation term in both of the equations explicitly omits the bias parameter θ_0 since it starts from $j = 1$ instead of $j = 0$. It should be noted that if the selected value of λ is too large, it may cause underfitting instead, where accuracy is low with both training and testing datasets. On the other hand, if the value of λ is too small or 0 then it may again show the overfitting of the model. Therefore, the selection of value for λ should be carefully realistic keeping these points in consideration, to get the best results.

The partial derivative for both models are changed consequently however, in general, its equation and the gradient descent equation is the same for both Linear and Logistic Regression, which is as follows:

$$\theta_j \Leftarrow \theta_j - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}(h(\theta, x_i) - y_i)x_{ij} + \frac{\lambda}{m}\theta_j\right] \; for\ j \in 1, 2, \dots, n. \tag{4.32}$$

It should be noted that this regularized gradient descent equation is for values of θ parameters from θ_1 to θ_n. The gradient descent equation for θ_0 is the same as previously stated in earlier sections.

Specifically, for Linear Regression, the Normal Equation to get θ minimized cost, can also be regularized, as follows:

$$\theta = (X^T X + \lambda L)^{-1} X^T y, \tag{4.33}$$

where λ is any numeric or decimal value and L is a matrix of dimensions $(n+1) \times (n+1)$ with 0 at top left location and everywhere else except the diagonal positions are filled with value 1. The regularization is used to overcome the previously stated issue of the non-invertibility of term $X^T X$ in the Normal Equation when m is less than n. Since the term λL is added, this makes the whole term invertible. This summarizes the basic techniques to generate and train the model for regression and classification problems via Linear Regression and Logistic Regression, respectively, and to avoid overfitting by regularization.

4.5 Artificial Neural Network

Artificial Neural Network (ANN) is a specific ML/DL technique to address the predictions or forecasting problems in real time [19, 20]. The ANN technique or algorithm has the same purpose as some ML approaches discussed earlier, which is to identify the relation within a dataset, between input and output data. Therefore, this is another supervised learning algorithm to determine the relation between input and output features by forming a ML model for future predictions.

The way of forming the relation within a dataset using ANN, replicates the way a human brain functions. Therefore, since a neural network is a connected network of neurons passing information, the neurons can be either artificial or organic, depending on the context. In the context of computer science and artificial intelligence, the neurons are artificial that can be represented by computer memory locations. This is a reason that a neural network in the context of computer science or artificial intelligence, is referred to as an Artificial Neural Network most of the times, with its abbreviation as ANN.

Figure 4.3 shows the basic structures of ANNs that can be trained as ML models to identify the mathematical relation between input and output features within a dataset. The first one on the top-left is the Single Layer Perceptron (SLP) which is only useful for learning the linear patterns within a dataset. It has only two layers: the input layer and the output layer. Usually, it can contain one neuron memory in the output layer to estimate output on the given values to the input layer neurons. The neurons in input layer can be more than two.

In general, for any problem dataset, the input layer is meant to contain neurons memory cells to hold the values of several input features. Whereas, the output layer is meant to contain a memory to maintain predicted output value of the output feature. However, in some scenarios there can be more than one output values to predict. Hence, the output layer can contain more than one neurons.

The rest of the three structures are the basic examples of the Multi-Layer Perceptron (MLP), which is useful for learning both linear or non-linear patterns within a dataset [161]. An ANN is referred to as MLP or DL neural network if it contains one or multiple hidden layers between the input and output layers. It should be noted that the hidden layers or their neurons memory cells are treated as a linear function unless the non-linear activation functions are applied to them [162]. This is to learn the arbitrary or non-linear pattern within a dataset.

The top-right structure in Figure 4.3 is the most basic MLP structure with only one hidden layer. Whereas, the remaining two are more deep MLPs or neural networks since more hidden layers are added, specifically to be more precise learning the non-linearity in the dataset patterns. The last one at the bottom-right is with the two output neurons in the output layer, which is to predict two output features values. Considering the case of the MNIST dataset for recognizing handwritten digits using Convolutional Neural Network (CNN) [163]. It requires ten output neurons to give the probabilities of all ten digits, from which the one having maximum probability is chosen as the classified and predicted handwritten digit against an input image.

The training of an ANN model relies on the two main parts across the learning process iterations: 1) forward propagation pass and 2) backward propagation pass. A forward propagation pass is to estimate or predict the output from the input to output layers. Whereas, a backward propagation pass from the output to input layers, is to update the weights between each pair of neurons, after computing the loss between predicted and

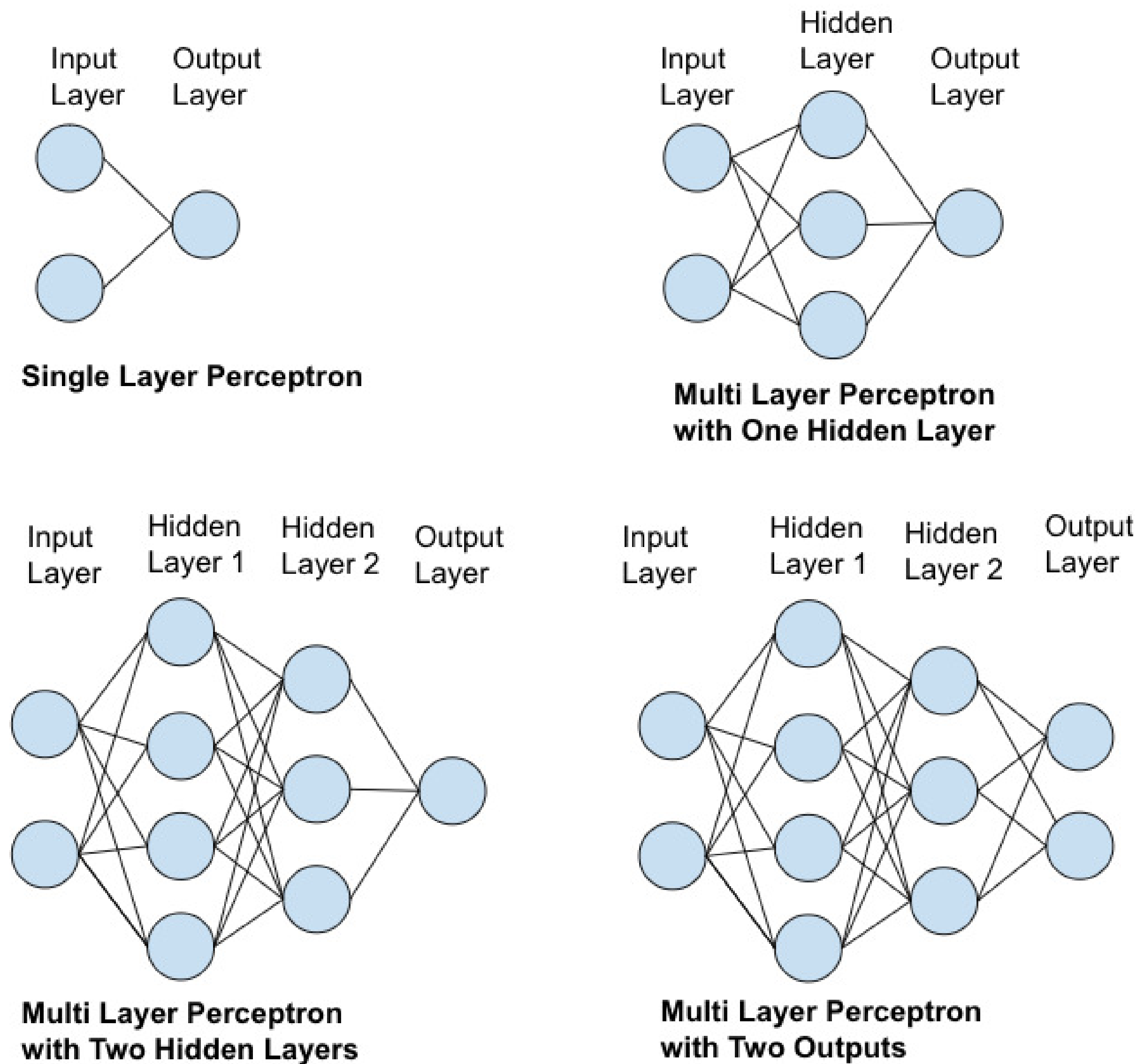


Figure 4.3: Basic Structures of ANNs.

actual output values. It should be noted that the lines connecting each pair of neurons in Figure 4.3, represents the weights of a neural network model.

The backward propagation pass uses the gradient descent approach to update the weights that minimize the cost value closer to global minimum. This is carried out by subtracting the partial derivative of cost function from the previous values of weights. This process should be executed backward throughout the layers, from the output to input layer, which also includes the hidden layers and their connecting weights.

By the end of training an ANN model, it has updated weights throughout the layers. This gives a cost function value near or equal to its global minimum point where its gradient approaches to zero. This is similar to the earlier discussion on linear and logistics regression models training since the gradient descent method is used in numerous ML algorithms for learning process.

4.5.1 Forward Propagation Pass

A forward propagation pass is the sequence of instructions containing matrix or vector operations step-by-step from input layers to hidden layers to output layer [164, 144]. The execution of a complete forward propagation pass gives the predicted output value at the end. Since it is a first part during the training iterations of an ANN, it is frequently used for predictions training set, and on unseen data once a model is trained.

Consider an ANN which is a MLP with 2 neurons in the input layer, 4 neurons in the hidden layer-1, 3 neurons in the hidden layer-2 and 2 neurons in the output layer, as depicted in the bottom-right structure of Figure 4.3. These layers can be represented by the following vectors in the CPU memory:

$$I_{2\times1} = \textit{an Input Layer vector of } 2 \textit{ rows}, 1 \textit{ column to keep } 2 \textit{ given input values}, \quad (4.34)$$
$$H1_{4\times1} = \textit{a Hidden Layer} - 1 \textit{ vector of } 4 \textit{ rows}, 1 \textit{ column to keep } 4 \textit{ neurons values}, \quad (4.35)$$
$$H2_{3\times1} = \textit{a Hidden Layer} - 2 \textit{ vector of } 3 \textit{ rows}, 1 \textit{ column to keep } 3 \textit{ neurons values}, \quad (4.36)$$
$$O_{2\times1} = \textit{an Output Layer vector of } 2 \textit{ rows}, 1 \textit{ column to keep } 2 \textit{ predicted outputs}. \quad (4.37)$$

Similarly, 3 weights matrices are required to contain weights values from the input layer to hidden layer-1, then hidden layer-1 to hidden layer 2, and then finally from hidden layer-2 to output layer. These matrices presented as following:

$$W1_{4\times2} = \textit{a matrix of weights connecting neurons from } I_{2\times1} \textit{ to } H1_{4\times1}, \quad (4.38)$$
$$W2_{3\times4} = \textit{a matrix of weights connecting neurons from } H1_{4\times1} \textit{ to } H2_{3\times1}, \quad (4.39)$$
$$W3_{2\times3} = \textit{a matrix of weights connecting neurons from } H2_{3\times1} \textit{ to } O_{2\times1}. \quad (4.40)$$

In each layer-to-layer, all the left-side layer neurons are connected with all the right-side layer neurons, representing a complete connected sub-graph. Therefore, each weight matrix is created to keep all possible weights values from layer-to-layer. Hence, $W1_{4\times2}$, $W2_{3\times4}$, and $W3_{2\times3}$ weight matrices have 8, 12 and 6 possible memory locations in total, respectively.

Since this system has all required layers vectors and weights matrices, the matrix-vector multiplication equations can be executed in the following sequence to complete one forward propagation pass to predict the outputs:

$$H1_{4\times1} \Leftarrow g(W1_{4\times2} \times I_{2\times1}), \quad (4.41)$$
$$H2_{3\times1} \Leftarrow g(W2_{3\times4} \times H1_{4\times1}), \quad (4.42)$$
$$O_{2\times1} \Leftarrow g(W3_{2\times3} \times H2_{3\times1}). \quad (4.43)$$

The first equation 4.41 executes to compute hidden layer-1 neurons values in vector $H1_{4\times1}$ by multiplying weight matrix $W1_{4\times2}$ with input layer vector $I_{2\times1}$. Then, the second equation 4.42 executes to compute hidden layer-2 neurons values in vector $H2_{3\times1}$ by multiplying weight matrix $W2_{3\times4}$ with the hidden layer-1 vector $H1_{4\times1}$. Finally, the last equation 4.43 executes to compute the outputs as the predictions in vector $O_{2\times1}$ by multiplying weight matrix $W3_{2\times3}$ with the hidden layer-2 vector $H2_{3\times1}$.

It should be noted that the sigmoid function $g()$ is applied on each element of a vector output from matrix-vector multiplication in each steps from eq. 4.41 to 4.43. Since there are two output neurons in the output layer, this means there are 2 output feature parameter values that are predicted by a one forward propagation pass. These equations are predicting the output against one input example.

Consider, there are m rows of training examples with n input features columns in matrix X and o number of output features in matrix Y. Then these steps can be rewritten by the following form of matrix operations:

$$H1_{4\times m} \Leftarrow g(W1_{4\times n} \times X^{T}_{n\times m}), \tag{4.44}$$

$$H2_{3\times m} \Leftarrow g(W2_{3\times 4} \times H1_{4\times m}), \tag{4.45}$$

$$O_{m\times o} \Leftarrow g((W3_{o\times 3} \times H2_{3\times m})^{T}). \tag{4.46}$$

The difference between matrix Y and O then can be computed to measure the loss between actual and predicted output. This loss is used to update the weights in later stages of the backward propagation pass, discussed in next section.

It should be noted that by adding the extra number of hidden layers in an ANN model, the steps of a Forward Propagation Pass will also increase. In general, if there are N hidden layers then there will be $N+1$ steps of a single pass to predict the model output. Furthermore, the amount of memory required will be increased as well since more weight matrices will be required for layer-to-layer neurons connecting weights values.

This means a forward propagation pass can be compute-intensive and memory-intensive at the same time for larger problems. This is a reason that the training of an ANN model of large and complex structure with huge amount of training data, is often offloaded to GPUs to accelerate learning.

4.5.2 Backward Propagation Pass

A backward propagation pass is the second main part of an ANN learning iteration. It is the critical step as it requires to update the weights in the weight matrices with respect to every layer in an ANN except the input layer [165, 144]. A backward propagation pass involves a series of steps in one iteration to update the weights values connecting the neurons from one layer to another.

A forward propagation pass starts from the leftmost side in the structure and executes till to the rightmost side. However, in the current context, the weights start to update from the rightmost side of the structure and run till the leftmost side, the reason it is referred to as backward propagation pass.

Consider the same ANN as used in the previous example of a forward propagation pass. Since hidden and output layers neurons are computed sequentially in a forward propagation pass for one training example, the error for each of these layers must be computed first. The error computation starts from the output layer, which can be described by following equation:

$$E^{\{4\}}_{2\times 1} = O_{2\times 1} - Y_{2\times 1}, \tag{4.47}$$

where $E^{\{4\}}$ is a vector matrix to contain error of (Output) Layer-4 neurons values, $O_{2\times 1}$ is predicted output vector matrix as a result of ANNs Forward Pass, and $Y_{2\times 1}$ is an actual output vector matrix against one training example input features set. This equation simply computes the error by taking the element-wise difference between values of predicted output vector $O_{2\times 1}$ and actual output vector $Y_{2\times 1}$.

Then next step is to compute the error in the hidden layers neurons values. The error of hidden neurons is dependent or based on the error computed for the (Output) Layer-4.

The error can be expressed in vector-matrix form of equations as following:

$$E^{\{3\}}_{3\times1} = (W3_{3\times 3})^T E^{\{4\}}_{2\times1} .* g'(H2_{3\times1}), \tag{4.48}$$

$$E^{\{2\}}_{4\times1} = (W2_{3\times4})^T E^{\{3\}}_{3\times1} .* g'(H1_{4\times1}), \tag{4.49}$$

where $E^{\{3\}}_{3\times1}$ and $E^{\{2\}}_{4\times1}$ are the error vector matrices for hidden layer-2 (Layer-3) and hidden layer-1 (Layer-2), respectively. Both error vectors are computed by multiplying the transpose of the weight matrix of a current layer with the error vector of right side layer. Then element wise multiplication with the derivative of sigmoid of current layer's neurons vector, is performed. The derivative of sigmoid function is represented by $g'()$ which can be expressed by the following equation by applying mathematical calculus transformation steps:

$$g'(H) = H. * (1 - H), \tag{4.50}$$

where H can be considered as a vector for hidden layer neurons values. It should be noted that $E^{\{1\}}_{2\times1}$ is not computed since it represents the error in the input values which is logically inappropriate.

The next main part is to compute the partial derivate of the errors with respect to weights values so, that it can be subtracted from the weights to minimize the error towards global minimum point. The partial derivative of error with respect to a weight for a layer, can be expressed by the following equation:

$$\frac{\partial}{\partial W(l)} J(W) = E^{\{l+1\}}(H^{\{l\}})^T, \tag{4.51}$$

where $J(W)$ represents error cost function and W denotes a weight matrix with respect to layer l. $E^{\{l+1\}}$ is error vector of next layer from the current layer and $(H^{\{l\}})^T$ is transpose of the vector represent current layer's neurons values. The result of multiplication of these terms in the equation, is matrix of partial derivatives of error corresponding to weight values in $W(l)$ matrix.

The row-column formation of the resultant matrix of partial derivatives should be the same as of weight matrix $W(l)$ in current layer l. Consequently, in this scenario, the partial derivates of error cost with respect to weights matrices of Layer-3 (hidden layer-2), Layer-2 (hidden layer-1) and Layer-1 (input layer) are computed by the following equations:

$$\frac{\partial}{\partial W3} J(W) = PW3_{2\times3} = E^{\{4\}}_{2\times1}(H^{\{3\}}_{3\times1})^T, \tag{4.52}$$

$$\frac{\partial}{\partial W2} J(W) = PW2_{3\times4} = E^{\{3\}}_{3\times1}(H^{\{2\}}_{4\times1})^T, \tag{4.53}$$

$$\frac{\partial}{\partial W1} J(W) = PW1_{4\times2} = E^{\{2\}}_{4\times1}(I_{2\times1})^T, \tag{4.54}$$

Once the partial derivatives: $PW3_{2\times3}$, $PW2_{3\times4}$ and $PW1_{4\times2}$, are computed, the next step is to multiply them with a learning rate value and subtract them element wise from the values in corresponding weight matrices. This is presented by the following equations:

$$W3_{2\times3} \Leftarrow W3_{2\times3} - \alpha PW3_{2\times3}, \tag{4.55}$$

$$W2_{3\times4} \Leftarrow W2_{3\times4} - \alpha PW2_{3\times4}, \tag{4.56}$$

$$W1_{4\times2} \Leftarrow W1_{4\times2} - \alpha PW1_{4\times2}, \tag{4.57}$$

where α is the learning rate. These equations depict the gradient descent method which has been previously discussed in Linear and Logistic Regression methods. Therefore, as stated earlier, the basic functionality to update the weight parameters values is to take the partial derivatives with respect to specific layers weight matrices. Subsequently, the partial derivatives should be subtracted element wise from the current weight matrices, to minimize the error cost towards global minimum point.

Additionally, the partial derivatives of error cost with all weight matrices should be executed simultaneously first. Afterwards, the weight matrices should be updated simultaneously. This is important to avoid logical errors in training since all the errors and then partial derivatives should be computed with the initial or previous values matrices in an iteration. This completes one backward propagation pass with one training example. When this process of forward and backward propagation is executed in a number of iterations for all training examples, all the weights should then be near the global minimum error.

The ANNs training can take an ample amount of time to give accurate predictions, especially when learning a complex non-linear pattern, as it can require a lot of training data. In practice, once there is a sufficient training data matrix, it can be reused in further numerous iterations over the training examples. This keeps updating the weights of the ANN model until it approaches the global minimum error and gives precise estimations. However, in this case the partial derivatives are supposed to be set to 0 before the training examples iterations, and then they should be accumulated throughout the iterations.

In addition to this, the extra equation for regularization can be added for some weight decay to avoid the overfitting of the model. Once the iterations of all training examples are complete, then the weights are updated by subtracting the corresponding accumulated and regularized partial derivatives from the previous values. The accumulation of partial derivatives during the training example iterations, can be further generalized in matrix forms and can be expressed by the following sequence of equations:

$$E^{\{L\}} = O - Y, \tag{4.58}$$

$$PW(l) = 0, \tag{4.59}$$

$$\textit{For training examples } 1 \textit{ to } m \ \{$$

$$E^{\{l\}} = (W(l))^T E^{\{l+1\}} .* H^{\{l\}} .* (1 - H^{\{l\}}) \textit{ for } (l = L-1, L-2, L-3, \dots, 2), \tag{4.60}$$

$$PW(l) \Leftarrow PW(l) + E^{\{l+1\}} (H^{\{l\}})^T \textit{ for } (l = 1, 2, 3, \dots, L-1)\}, \tag{4.61}$$

$$D(l) = \frac{1}{m}(PW(l) + \lambda W(l)), \tag{4.62}$$

$$W(l) \Leftarrow W(l) - \alpha D(l) \textit{ for } (l = 1, 2, 3, \dots, L-1), \tag{4.63}$$

where L is the total number of layers in an ANN model, l denotes current layer and $D(l)$ represents regularized accumulated partial derivative of error with respect to weight matrix of layer l. Additionally, m denotes number of training examples and λ is the regularization parameter or weight decay value in the context of ANNs. This means the partial derivatives with respect to each layer's weight matrices are accumulated, excluding the last output layer.

The equations 4.58-4.59 and 4.62-4.63, are part of the outer iterations which run till some maximum limit. Before equation 4.58-4.59, the Forward Propagation Pass is executed to predict the output values in vector O which is then subject to differences with the actual output values from training set in vector Y. The equation 4.58 can be

changed to a different error loss function i.e., MSE, MAE, etc. This summarizes the overview of ANNs structure and the ML process.

4.6 Model Testing

In this section, the model testing is discussed in terms of different error metrics and methods. Since regression and classification represent two different sets of learning algorithms, the methods to test their models prediction accuracy are different. The model testing is normally conducted on the testing dataset. The regression models' prediction accuracy is usually tested with commonly used statistical error metrics. The error metrics are such as: Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), Mean Absolute Percentage Error (MAPE) and R-Squared Score. Whereas, the methods to check classification models' prediction accuracy are different. These methods are such as: Classification Accuracy, Confusion Matrix, Precision and Recall, F1 Score, Sensitivity and Specificity, and ROC curve and AUC. This section follows the details of methods to test the regression and classification models.

4.6.1 Regression Model Testing

To test a regression model, a simplest error metric is MAE. This takes the average of absolute error differences between the number of actual and predicted values [166]. The mean average can range from 0 to infinity (∞). If mean is closer to 0 value, the better the model is. This is a basic technique to get the insight of error in a regression model. It can be described by the following equation:

$$MAE = \frac{1}{n}\sum_{i=0}^{n-1}|a_i - p_i|, \tag{4.64}$$

where n is the number of actual and predicted values in a set, a_i is the ith actual value in a set and p_i is the ith predicted value in a set.

Since MAE is the natural average error measure, its value shows how much error can be expected in future predictions or forecast. However, in this case the relative error of two models cannot be compared since one model may have a big MAE due to one or few big errors. The other model may have a big error due to multiple small errors, which can only be analyzed by human eye if a dataset is sufficiently small.

The improved form of MAE is RMSE [167]. It is the square root of the average of squared error difference between actual and predicted values. It can also range from 0 to infinity (∞). Since it is greater than the magnitude of MAE, its lower value means the model is better in precision. The RMSE or just MSE (mean squared error) are normally used as the error metric for regression model. RMSE can be expressed by the following equation:

$$RMSE = \sqrt{\sum_{i=0}^{n-1}\frac{(a_i - p_i)^2}{n}}. \tag{4.65}$$

The RMSE measures how the residuals of error are wide and spread. In some sense it can be considered as the standard deviation of the uninterpretable variance. As the squared terms in RMSE gives relatively value which may covers the greater errors or outliers therefore, reducing the RMSE value reduces the greater error, to optimize the model.

A prediction accuracy of a regression model can also be measured in the form of percentage value. For this purpose, the metric MAPE is required to be computed [168]. The MAPE is the average of percentage of error difference between actual and predicted values. This can also range from 0 to ∞, and the lower percentage error means higher prediction accuracy in terms of percentage. It can be expressed by the following equation:

$$MAPE = \frac{1}{n}\sum_{i=0}^{n-1}\left|\frac{a_i - p_i}{a_i}\right| \times 100\%. \tag{4.66}$$

Since it is expressed in terms of percentage, it is normally and easily understood by most of the people, specially, the non-technical ones. Hence, the prediction accuracy can be simply calculated by subtracting a MAPE value from 100%. This prediction accuracy metric $(100\% - MAPE)$ has been also used in recent and latest research work by Madhumitha et al., Nilmini et al. and Marcus et al. in [169, 170, 171].

It should also be noted from the equation that if actual values are too small or close to 0 then each error percentage is extremely large and MAPE can be biased. Therefore, MAPE is mostly discouraged by the experts if there are many small data observations.

Another useful metric is R-Squared Score [172]. This score value is the determination coefficient. The coefficient represents the proportion in variance of the output feature values. The score value or determination coefficient can be expressed by the following equation:

$$R^2 = 1 - \frac{\sum_{i=0}^{i=n-1}(a_i - p_i)^2}{\sum_{i=0}^{i=n-1}(a_i - \bar{a})^2}, \tag{4.67}$$

where $\bar{a}$ is the mean of the actual output values. This metric value normally ranges from 0 to 1 however, it can be negative in some cases. If the score value is negative, this means a model is not following the pattern in dataset at all, hence, a model is under-fit.

A positive score value close to 0 indicates that the model is not improving in making accurate predictions over the mean model. Whereas, a score value near to 1 indicates that the model is making predictions accurately. Therefore, normally, a higher R-Squared Score means a better the model is.

4.6.2 Classification Model Testing

In case of classification models, the simplest metric to test the model is the classification accuracy [173]. It simply shows the number or percentage of predictions which are correct. Straightforwardly, the higher number of correct predictions means highly accurate model. It can be described in the following form of equation:

$$CA = \frac{Total\ Correct\ Predictions}{Total\ Predictions}. \tag{4.68}$$

In some cases, this metric may not be enough as it may not be giving the actual insights. For example, consider a case of binary classification model predicting all the outputs as "spam email". This can be due to the reason that the dataset contains 95% of the cases with "spam email" as the actual output values and remaining only 5% of the cases with "not spam email" outputs.

In this case, a model is hardly doing any computations to predict the output because of the class distribution in the dataset is not balanced. Therefore, it has caused a high bias in the model although it is giving 95% accuracy. This model can probably fail on the

unseen dataset with the balanced class distribution or the dataset with high frequency of the other class as the output. Consequently, it requires other metrics or methods to properly analyze the classification model with more insight detail.

To deal with such nature of problems regarding the classification accuracy, the confusion matrix is normally a better approach rather than a metric. Confusion matrix provides more clear insights of the predictions. It is a prerequisite to other metrics regarding classification, such as precision and recall.

A confusion matrix indicates true or false predictions for each output class [174]. Generally, if there are two classes for an output as in binary classification then a confusion matrix has a size of 2×2. In case of a multi-class classification with n classes for an output then a confusion matrix has size of $n \times n$, which is first general rule for a confusion matrix.

There are mainly 4 entities which represents the confusion matrix regarding grouping the predictions: 1) True Positive, 2) False Positive, 3) True Negative, and 4) False Negative. The True Positive case occurs where the actual class is positive and predicted class is also positive. The False Positive case occurs where the actual class is negative, but the predicted class is positive. The False Negative case occurs where the actual class is positive, but the predicted class is negative. Lastly, the True Negative case occurs where the actual class is negative, and the predicted class is also negative. The desired outcome is that the actual class and predicted class should be the same.

Once the confusion matrix is established then the Precision and Recall metric can be considered. This takes a step further from the classification accuracy, allowing deeper evaluation of a model [175]. The emphasis of Precision is to see the number of True Positive predictions out of the total positive predictions, and can be presented by the following equation:

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}. \tag{4.69}$$

Then the emphasis of Recall is to measure that how the model is performing at predicting the positive class correctly, means True Positives out of the total actual positives, and it can be expressed by the following equation:

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}. \tag{4.70}$$

This is a direct indicator that would tell how many actual positives are the predicted positives actually. It should be noted that both the Precision and Recall cannot be maximized at the same time. Either Precision or Recall can be maximized, which depends on the nature of the problem. For instance, in spam email detection system, maximized Precision is desired since the actual spam email should be detected as spam, because detecting a normal email as spam is a False Positive which is not desired. In the other case of tumour detection, maximizing Recall is required as the maximum possibility of detecting a tumour is required.

Another metric which can combine both Precision and Recall in one equation and value, is F1 Score [176]. The F1 Score is the average Precision-Recall product. It can be expressed by the following equation:

$$F1\ Score = 2 \times \frac{Precision \times Recall}{Precision + Recall}. \tag{4.71}$$

It is a reliable metric, especially if a dataset contains an unbalanced distribution of both False Positives and False Negatives. Normally, the best F1 Score is considered as 1 and worst is considered as 0.

The further step to summarize the model performance is the ROC (Receiver Operating Characteristics) curve [177]. It combines confusion matrices at different threshold values. The graph of ROC curve normally presents Y-axis with Sensitivity values which are different True Positive rates (TPR) or Recall, and X-axis with False Positive rates (FPR) which are the opposite of Specificity [178]. Specificity is used to focus on the negative class therefore, it is based on the amount of the cases where actual negative output class is also predicted as negative class. In other words, it is True Negative rate (TNR). The Specificity (TNR) and FPR can be expressed by the following equations:

$$TNR = Specificity = \frac{TN}{TN + FP}, \tag{4.72}$$

$$FPR = (1 - Specificity) = 1 - TNR = 1 - \frac{TN}{TN + FP} = \frac{FP}{TN + FP}. \tag{4.73}$$

Plotting these values on their axes, the curve can give the summary of the model performance. However, setting threshold to 0 results in TPR and FPR values equal to 1, and threshold to 1, results in TPR and FPR to 0. Therefore, this becomes an unwise option to set threshold value to 0 or 1. While aiming to increase the TPR and keeping the FPR as low as possible, the FPR also increases when TPR get increased. This detects a number of False Positives that can be tolerated.

This issue can be overcome by measuring the Area Under the curve (AUC) of ROC [179]. Since it is the area between (0,0) and (1,1) co-ordinates of FPR and TPR axes, it can be computed by the integration method in calculus. The AUC value gives the aggregated model performance at all threshold values. As the perfect classifier has the value 1 therefore, an area value closer to 1 indicates the better classification model.

To summarize this, metric used to measure the performance or accuracy of the classification model, depends on the requirements of the problem being addressed by any ML approach. Therefore, the requirements should be defined clearly to ease the process of metric selection for measuring the ML model performance.

4.7 Recent ML Research for Optimizing HPC IO

ML has been used in the range of studies, to analyze, predict and improve the parallel IO performance in the HPC systems. Some of them have also ANNs for the same purpose.

Xu et al. has demonstrated that the HPC IO is affected by factors like CPU frequency, number of IO threads, and the IO scheduler [14]. The IO behaviour is predicted and determined based on these factors using interpolation and extrapolation techniques by developing a model using a data analytic framework over large-scale experiments. The performance of the methods is being evaluated by measuring prediction accuracy at previously unseen system configurations. The methodology for optimizing system configurations uses the estimated variability map based on Bayesian Treed Gaussian Process and some other regression methods. This gives new insights into existing statistical methods for the practice of HPC variability management in terms of parameters selection.

The work presented by Bez et al., is the adaptive method to schedule parallel IO request for any application on any HPC system by tuning the parameters depending

on time window of current workload [15]. The adaptive method is formulated using reinforcement learning of the scheduling algorithm. It achieves 88% of precision to select parameters on runtime after observing the access pattern (contiguous or non-contiguous) for few minutes. Consequently, the system is able to optimize its IO performance for rest of its life, as claimed.

The approach presented by Bağbaba, is the random forest regression modelling, used as the ML technique to predict the IO bandwidth for only collective write operation in MPI-IO Library [17]. In this paper, the accuracy of prediction is very high. It varies from 82% to 99% approximately depending upon maximum depth setting. However, the training and testing datasets are very small in comparison to the size of the benchmarks datasets generated in this thesis. The accuracy can be lower if more variation is added in datasets. The remaining differences are the format of bench-marking file, the parameters and the processor or cluster hardware.

The research by Behzad et al., the IO optimization is proposed for HDF5 file format parallel applications on various HPC platforms with LFS and GPFS [16]. The parallel IO is optimized via auto-tuning, supported by IO modelling of Lustre's IOR (Interleaved or Random) benchmarks and other IO benchmarks, through nonlinear regression models prediction. The tuning achieves significant increase in IO bandwidth for different applications on the HPC platforms, by means of prediction and selection of new parameters values. To some extent, Madireddy et al. also provides the parallel IO predictive modelling of LFS IOR benchmarks by developing a Gaussian process regression ML model [18].

The system `PRIONN` developed by Wyatt et al., predicts the runtime and IO resource usage of an IO aware scheduled job on HPC cluster [24]. This is carried out using Artificial Neural Networks (ANNs) as its deep learning model, particularly Convolutional Neural Networks (1D-CNN and 2D-CNN) [19]. The novelty of this tool is the image representation of job scripts as the input and output is the automatically detected important features and patterns from sets of characters from the input. This output is further passed down to ANN models to predict the runtime. The whole strategy achieved over 75% mean and 98% median accuracy for runtime and IO predictions on 300,000 job scripts on real HPC system.

The research presented by Schmidt et al., predicts the file access times on a Lustre file system from the client end [21]. File access times are measured in various test series and are then used to develop different models to predict them. The evaluation shows that these models which utilize ANNs, give 30% less average prediction error in comparison to linear models. The distribution of file access times was evaluated with respect to file accesses using identical parameters. The typical access times usually differ by orders of magnitude and depend upon an alternative IO path.

The research conducted by Zhao et al., involves the in-depth survey on Lustre File System [82]. The different factors have been highlighted during the performance evaluation and the prediction model using Grey Theory. The Grey Theory in general, refers to the Grey systems [180]. The systems that lack information regarding its data pattern or behaviour. They are turned into regular series by establishing a non-function based forecasting methods instead of time series and regressive methods. This eventually led to a mathematical relation among those factors affecting IO in Lustre File System. This prediction model can obtain better prediction precision and could be further applied to performance evaluation of other parallel file systems.

The other paper presented by Zhao et al., experiments the performance of IO band-

width over Lustre File System [83]. The purpose of this experiment is to form a mathematical based prediction model, resulted in prediction of the IO bandwidth from 17% to 28%. This model involved various factors affecting the IO bandwidth such as OSS threads for OSTs, number of OSTs, read-write request size, etc.

This summarizes the related work in the area of improving HPC IO performance over different levels of execution and environments, with the use powerful ML algorithms and techniques.

4.8 Summary

In this chapter, the ML process and different techniques have been discussed in detail. It first discusses the different stages of data preparation, which is a primary requirement before starting the learning process of the models. This includes, identification of input and output feature parameters, which can explicitly define what should be input to the model and what should be predicted as output.

Furthermore, the dataset generation has been discussed which is based on true and valid data and reporting of every minor software/hardware platform and environment detail. Then the importance of scaling or normalization of the data has been discussed in order the support the training process. Afterwards, the techniques and options to shuffle and split the data have been discussed to select the partitions of the dataset that can be used for training and testing or validation of the model.

In relation to model training, two main ML techniques have been discussed for two main categories of the problems: regression and classification. For regression problems, the Linear Regression approach has been discussed with its mathematical detail. Similarly, the Logistic Regression approach has been discussed for classification problems, including both binary and multi class classification.

Later in this chapter, the further two different categories of ML problems and their solutions have been discussed in terms of supervised and unsupervised learning. The problem addressed by both techniques have been differentiated and their solutions in terms different ML algorithms have been discussed in a brief detail.

Finally, the Artificial Neural Network technique have been discussed as part of model training, which is also used in this thesis. This has been explained with its two main functionality components for the learning process: forward propagation pass and backward propagation pass.

Afterwards, the different model testing metrics and techniques have been discussed. This has been explained from the aspects of regression and classification problems, separately. For regression models, the metrics such as MAE, RMSE, MAPE, R-Squared Score, have been briefly explained.

For classification models, the Classification Accuracy has been explained, which is prerequisite to Confusion Matrix, Precision and Recall. Afterwards, Precision and Recall have been explained to be used for F1 Score, Sensitivity and Specificity, ROC curve and Area Under the curve (AUC).

Finally, recent studies have been discussed regarding optimizing HPC-IO performance at different execution levels and the computing environments or platforms, based on the IO performance prediction based on the various ML approaches.

Chapter 5

Applying Neural Networks to predict HPC-IO bandwidth over seismic data on Lustre File System for ExSeisDat

5.1 Introduction

Seismic data is one of the most critical factors for geophysicists to study and understand the earth structure beneath its surface or seabed. Aside from being of critical importance in understanding the globe and when earthquakes and tremors might jeopardise human life [2], the study of Seismic data is also a critical factor for the Oil and Gas industry [3]. The SEG-Y File format is the standard across the globe for the encapsulation and processing of the seismic data [4]. The SEG-Y File format shown in Figure 3.2 is typically very complex with alternative arrangements of traces, preceded by their corresponding headers. Its is also quite common for its file size to reach petabytes in scale. This dramatically increases demands on high performance based IO processing of seismic data across the research and Oil/Gas production industry. This is where the High Performance Computing (HPC) or super-computing clusters play a significant role.

The Extreme-Scale Seismic Data (ExSeisDat) Library is developed to process the SEG-Y files efficiently by further using its Parallel-IO Library (PIOL) and Workflow Library on the HPC clusters [1]. It is based on the C++ language platform and the standard Message Passing Interface (MPI) Library, a parallel-distributed memory framework, which provides a large set of Application Interfaces (APIs) to exploit the potential of parallelism within the clusters [7]. Commonly, the parallel MPI-IO struggles in overcoming the performance degradation of a program because as it relies on certain parameters to project the IO bandwidth. This is also the case with respect to MPI-IO when applied to ExSeisDat data and the processing of SEG-Y files. These important parameters are related to a number of components employed on clusters. For example, the number of MPI processes running on compute nodes, the Parallel File System (PFS) managing multiple storage objects which is known as the Lustre File System (LFS) ([8]) in case of this thesis, and also the file properties, access patterns, etc.

Previous research has shown that IO performance prediction based on the different parameters settings can result in significant benefits [14, 15, 16, 17, 18]. Despite the key differences between this and existing research with respect to parameters and ML

techniques, the prediction of SEG-Y file IO performance beforehand is itself innovative. It can be immensely beneficial for tuning the related configuration parameter settings to overcome the poor IO performance during the execution of MPI application. In [21], the Artificial Neural Networks (ANN [20, 19]) based approach was used as a ML technique to estimate the IO performance based on the file access times and patterns and showed 30% less prediction error in comparison to linear models. This provided the motivation to apply ANNs to this problem, arising from its forecasting capabilities to the key input parameters, and predict the IO bandwidth before execution of the actual IO operation within application or program. The high prediction accuracy of Deep Learning (DL) ANNs has also been proven on different frameworks and applications as mentioned in [23]. The PyTorch is one of those DL frameworks that have been used in the research, for developing and training the ANN models [22].

The critical idea is to use the predicted bandwidth as a means of optimizing the IO performance by tuning the parameters. The tuning of parameters can be those settings from the evaluations of benchmarks execution results, given later in this chapter. In addition to this, some can be the combinations suggested in previous research that can increase the IO bandwidth performance on the LFS based cluster in [9, 10, 11, 12]. These existing studies suggest that by setting the number of MPI processes to the number of parallel hard disks, and chunk size (being read or written by each MPI process) to stripe size (file striping unit over hard disks in round robin fashion) can significantly increase the IO bandwidth performance. There are another series of approaches that are outside the scope of this research. These include manipulating Remote Procedure Call (RPC) thread counts, controlling the Object Storage Targets (OSTs or hard disks) and using the single MPI calls to read or write large chunk sized *MPI_Datatype* [73, 72].

Our research focuses primarily on the very basic and useful parameter settings that can be considered for the prediction of bandwidth values. These settings can enhance IO throughput, as shown with the support of parallel HiPlot utility's graph plots ([25]), from separately executed MPI-IO, SEG-Y File IO and Sorting benchmarks from ExSeisDat. These parallel plots of IO bandwidth values against different parameters settings is another novelty of this research. In addition to this, the other primary focus of this research approach is the prediction of IO bandwidth prior to its runtime, and therefore potentially yielding significant performance outcomes in terms of accuracy.

The prediction results show that the ANN based models are sufficiently generalized in the research, to predict the IO bandwidth behaviour. This chapter is structured as follows; Related Work in section 5.2, Design and Implementation in 5.3, Experimental Result Analysis in 5.4, Discussion in 5.5 and Conclusion in 5.6.

5.2 Related Work

Previous research has examined where IO behaviour was machine-observed using certain ML techniques, parameters and environment, subsequently prediction of IO was used as a tool to overcome poor performance in IO bandwidth. These research studies motivated to approach the prediction of IO performance for ExSeisDat over SEG-Y format files distributed over LFS based networked storage in HPC clusters.

Xu et al. in [14], it has been demonstrated that HPC IO is affected by factors like CPU frequency, number of IO threads, and the IO scheduler. The IO behaviour is predicted and determined on the basis of these factors using interpolation and extrapolation

techniques by developing a model using a data analytic framework over large-scale experiments. The performance of the methods is being evaluated by measuring prediction accuracy at previously unseen system configurations. Then the methodology for optimizing system configurations uses the estimated variability map based on Bayesian Treed Gaussian Process and some other regression methods. This yield new insights into existing statistical methods for the practice of HPC variability management in terms of parameters selection.

The work presented by Bez et al. in [15], is the adaptive method to schedule parallel IO request for any application on any HPC system by tuning the parameters depending on time window of current workload. The adaptive method is formulated using reinforcement learning of the scheduling algorithm. It achieves 88% of precision to select parameters on runtime after observing the access pattern (contiguous or non-contiguous) for few minutes. Consequently, the system will be able to optimize its IO performance for rest of its life, as claimed.

The approach presented by Bağbaba in [17], is the random forest regression modelling, used as the ML technique to predict the IO bandwidth for only collective write operation in MPI-IO Library. The accuracy of prediction is very high in this paper. It varies from 82% to 99% approximately depending upon maximum depth setting, but the training and testing datasets are very small in comparison to the size of the benchmarks datasets used in this thesis. The accuracy can be lower if more variation is added in datasets. The remaining differences are the format of bench-marking file, the parameters and the processor or cluster hardware.

The research by Behzad et al. in [16], the IO optimization is proposed for HDF5 file format parallel applications on various HPC platforms with LFS and GPFS. The parallel IO is optimized via auto-tuning, supported by IO modelling of Lustre's IOR (Interleaved or Random) benchmarks and other IO benchmarks, through nonlinear regression models prediction. The tuning achieves significant increase in IO bandwidth for different applications on the HPC platforms, by means of prediction and selection of new parameters values. To some extent, Madireddy et al. in [18] also provides the parallel IO predictive modelling of LFS IOR benchmarks by developing a Gaussian process regression ML model.

The research presented by Schmidt et al. in [21], predicts the file access times on a Lustre file system from the client end. File access times are measured in various test series and are then used to develop different models to predict them. The evaluation shows that these models which utilize ANNs, give 30% less average prediction error in comparison to linear models. The distribution of file access times was evaluated with respect to file accesses using identical parameters. The typical access times usually differ by orders of magnitude and depend upon an alternative IO path.

In addition to this, few other researches are also explored in the context of prediction based optimization for MPI applications through ML and auto-tuning parameters, but they all lack consideration of the IO side [181, 182].

In contrast to these existing studies, the approach of this research focuses on predictive modelling of IO bandwidth, using ANNs, over seismic data in the form of SEG-Y File format as the primary usecase for this research.

5.3 Design and Implementation

5.3.1 Research Methodology

This research is conducted in number of steps which are: 1) Identification of key configuration parameters, 2) Generating list of configurations values sets, 3) Development of Benchmarks over key parameters, 4) Execution of Benchmarks over key parameters, 5) Collection of runtime performance data as a training set for ML, 6) Training of ANN model over collected training set, 7) Prediction of IO performance over test set (20% of collected data) and 8) Accuracy evaluation for predicted results. Figure 5.1 shows the main components of the research model.

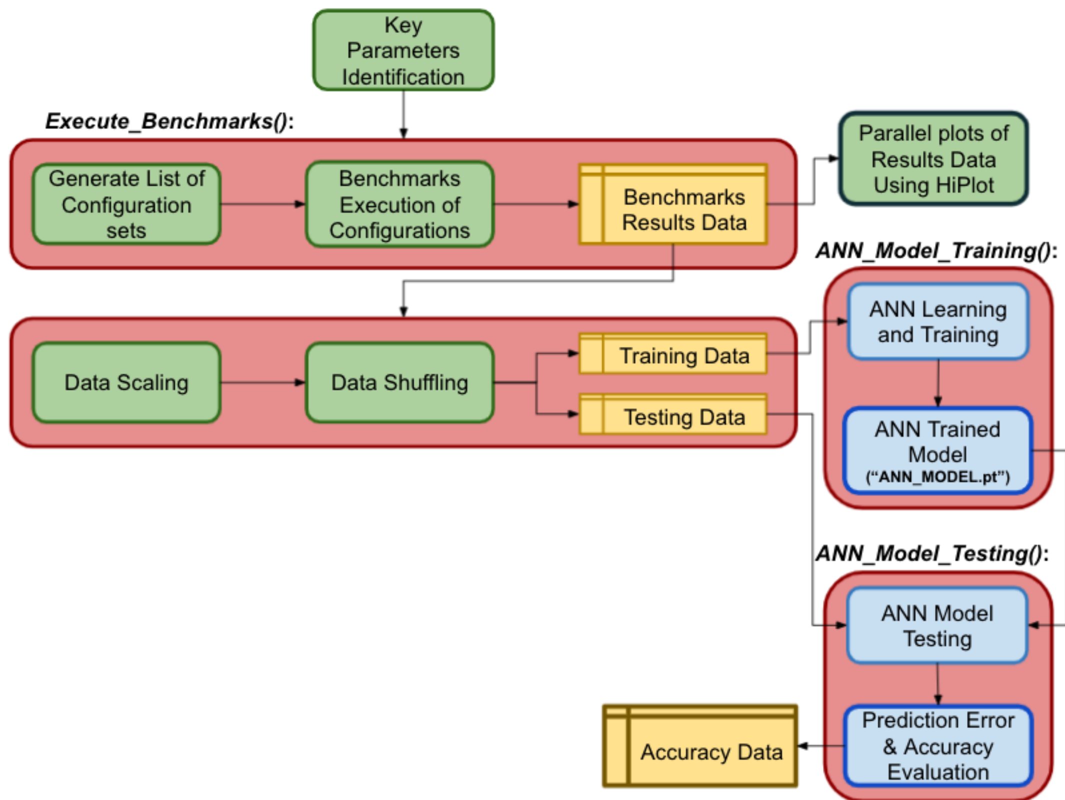


Figure 5.1: Research Flow

5.3.2 Key Parameters Identified

In this research, three types of benchmarks from ExSeisDat are executed: 1) basic MPI-IO benchmarks to read and write stream of bytes in file, 2) SEG-Y File IO benchmarks to read and write seismic trace data and 3) SEG-Y file sorting benchmarks to sort seismic trace data from different unsorted orders. It should be noted that some parameters are overlapping for all benchmarks and some are different due to the nature of their workings which are explained in later sections.

Table 5.1 shows the complete list of key parameters identified and the values used to conduct each of the three benchmarks. The most right column of this table has been merged to show that both (SEG-Y IO and Sorting) benchmarks have the same parameters except some of their different possible values which are explained later.

Table 5.1: Configuration Parameters and their values.

Parameters	Benchmarks	
	MPI-IO	SEG-Y IO & Sorting
number of MPI nodes	2,4,8,16	2,4,8,16
MPI processes per node	1,2,4,8	1,2,4,8
stripe count	2,4,8,16	2,4,8,16
stripe size (MBs)	1,256,512,1024,2048	1,256,512,1024,2048
file size (GBs)	1,2,4,8,16,32	-
chunk size (GBs) per process	0.25,0.5,1,2	-
io operation	read/write	read/write
file access pattern	collective/non-collective	contiguous/random
number of traces	-	512,1024,2048,4096
samples per trace	-	256,512,1024,2048
unsorted order	-	uniform/reverse/random

5.3.3 Development of Benchmarks

This Section explains working of each benchmark type, implementation and formulation of results.

5.3.3.1 Working of MPI-IO Benchmarks

MPI-IO benchmarks are the simplest, such that MPI processes on each node would perform READ or WRITE operations on their respective chunk sizes in a single file size striped across number of Lustre disks (stripe count) with a certain stripe size. The MPI read or write calls would be either *collective* (where each process also accesses neighbouring processes memory space) or *non-collective* (where each process accesses only their own memory space) to complete the parallel IO. Considering all the possible values mentioned in Table 5.1 from MPI-IO benchmarks, the total number of possible configurations of parameters or the total number of these benchmark executions would be 30720 (15360 for each READ and WRITE operations).

5.3.3.2 Working of SEG-Y File-IO Benchmarks

SEG-Y File-IO Benchmarks are different than basic MPI-IO benchmarks. In this case, they are reading or writing (modifying) a combination of traces and samples per trace within an already generated SEG-Y file with only *uniform* order. MPI processes in these benchmarks would be either read or write using *contiguous* or *random* access patterns as can be seen from SEG-Y File-IO Benchmark parameters stated in Table 5.1. As the SEG-Y file is Lustre striped over a number of OSTs, therefore the number of traces and samples per trace are distributed over different disks with a certain stripe size. In this case, the total possible configuration of parameters or executions of this benchmark type are 20480 (10240 for each READ and WRITE operations).

5.3.3.3 Working of SEG-Y file sorting Benchmarks

Sorting Benchmarks are more complex than the SEG-Y File-IO benchmarks. In these benchmarks, first a SEG-Y file would be generated with any of the unsorted orders: 1)

uniform, means SEG-Y file is sorted in ascending order with respect to source-x coordinate from the trace header value [4], 2) *uniform_reverse*, means trace data is arranged in descending order with respect to source-x or 3) *random*, means trace data is arranged in arbitrary manner, as mentioned in Table 5.1. Then MPI processes in each sorting benchmark would either read or write with only contiguous access pattern the number of traces and samples per trace in SEG-Y file, Lustre striped. Therefore, in this scenario, the total possible configuration of parameters or executions of this benchmark type are 30720 (15360 for each READ and WRITE operations).

5.3.3.4 Code Structure Generating Benchmarks Data

The Listing 5.1, represents the overview of the method to collect IO bandwidth data against different parameters and values. The parameters and values from Table 5.1 are passed as arguments to a function definition `Execute_Benchmarks()`, for executing all possible the benchmarks on line 1 in Listing 5.1. Initially, they are being passed to a function `GenerateConfigsValuesList()` on line 2. This function is responsible to generate a complete proper possible lists of configuration settings according to hierarchy of nested loops for each parameter in Figure 5.2. The configurations lists have been generated with respect to each benchmark type (Basic MPI-IO, SEG-Y File-IO and SEG-Y file sorting). The last nested loop completes one set of configuration values which is appended to the list that eventually builds a complete list of all possible configurations for each benchmark type. These lists for each benchmark type are being fetched in a sequence on line 3 with outer-loop. In line 4 with inner-loop a unique set of a configuration setting (`config`) is being fetched from a current benchmark type configurations list (`configs_list`). This one set of configuration setting object represents one execution benchmark for a file i.e. [`MPI_nodes = 16, processes_per_node = 8, stripe_count = 8, stripe_size = 256, io_operation = WRITE, access_pattern = RANDOM, traces = 4096, samples_per_trace = 1024,...`] for one SEG-Y WRITE operation. Apart from configuration settings values in `config` object, it also contains all other necessary information related to currently executing benchmark i.e. benchmark type, input/output file names, the target Darshan profiling file path and name, etc.

For each benchmark execution, the pre-benchmark settings are applied for the Lustre File System and Darshan utility on line 11 and 15, respectively. Darshan is a HPC IO characterization tool, designed to capture IO bandwidth performance [91]. As discussed previously, benchmarking is completed using Lustre File System disks therefore, before each execution of the benchmarks with certain configurations, the Lustre Striping on target file is executed to distribute the file parts on OSTs. This is being done to test the performance of a particular benchmark configuration with respect to its file striping.

On line 18, an input file is always being generated before the execution of actual IO benchmark except the MPI-IO WRITE benchmark. The IO bandwidth value is recorded by the Darshan utility, once the benchmark is executed over a set of configuration parameters, on line 21, by reading and parsing the generated Darshan file. After the execution of a benchmark the file is being deleted in order to remove it from cache, as a post-benchmark setting on line 25.

5.3.3.5 Formation of results

The results of each benchmark along with their configuration parameters are appended in the YAML file on line 22 of Listing 5.1. Therefore, each `(config, io_bandwidth)`

Listing 5.1: Overview of Code Executing IO Benchmarks and Collecting Bandwidth Data.

```
1  def Execute_Benchmarks(Parameters_And_Values_List):
2  MPI_IO_Configs, SEGY_IO_Configs, SEGY_Sorting_Configs =
       GenerateConfigsValuesList(Parameters_And_Values_List)
3  for each configs_list in [MPI_IO_Configs,SEGY_IO_Configs,
4  SEGY_Sorting_Configs]
5      for each config in configs_list:
6
7          # Here each config is a unique set of values from each
8          # parameter mentioned in Table 1 with respect to the
9          # Benchmark type.
10
11         # First applying pre-benchmark settings of the config.
12         Set_Lustre_File_Striping(config['file_name'],
13                         config['stripe_count'],
14                         config['stripe_size'])
15
16         Set_Darshan_File_Name_And_Path(config['darshan_file'])
17
18         # Executing the IO benchmark with this config values.
19         Execute_Benchmark_With_This_Config(config)
20
21         # Appending results in output file.
22         io_bandwidth = ReadDarshanFile(config['darshan_file'])
23         Append_Output_in_YAML_Format_File(config,io_bandwidth)
24
25         # Remove benchmarked file as a post-benchmarking step.
26         Delete_Benchmarked_File(config['file_name'])
```

object written to file is a dictionary object with its `(key, value)` pairs to represent each execution.

5.3.4 Development of ML based ANN models

In this section, the approach is outlined that is used to learn and predict the performance behaviour arising from the benchmark results and types outlined previously.

For this purpose, the ANNs have been used to learn the changing performance behaviour over the various configurations on each benchmark's result. The Listing 5.2 shows the very basic algorithmic view of the ML process under the definition of function `ANN_Model_Training()`. The formation of ANN's layers plays a key role for the Machine Learning approach as shown from the line 5 to 14 of Listing 5.2. The ANN for each benchmark comprises of a 7 node input layer which represents the input parameters, then the output layer has one node to represent a single output as bandwidth value for each input of configuration parameters values. The number of hidden layers chosen are 2, the number of nodes in the first hidden layer are 256, and the number of nodes in the second hidden layer are 128, as described in Table 5.2. These values represent the Neural Network architectures used in this research, for training the models of each benchmark type, running independently for read and write operations.

In Table 5.3, the hyper-parameters applied to the models during the training are described, which eventually supports in improving the accuracy on the test set. The Dropout percentage rate (`nn.Dropout()`) and Rectified Linear Unit activation function

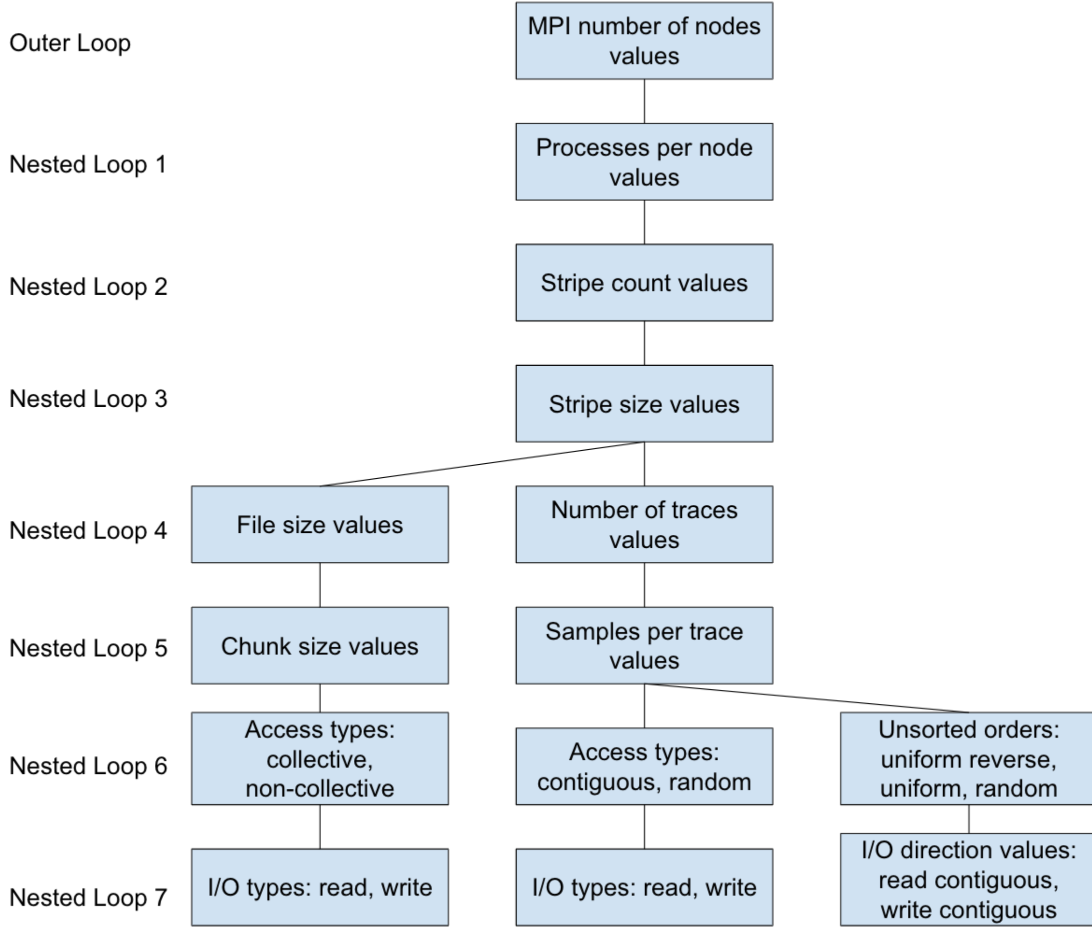


Figure 5.2: Hierarchy of parameter values nested loops generating all possible configurations to benchmark.

Table 5.2: ANN's Description Table.

input layer	hidden layer 1	hidden layer 2	output layer
7	256	128	1

(`nn.ReLU()`) are applied on ANN's nodes and layers during training on line 9, 11 and 13 [183]. These are the example values to demonstrate training for one model otherwise, they are different in actual as mentioned in 4th column from the left side in Table 5.3. Similarly, the weight decay value on line 19 is also different for each model, as mentioned in 5th or most right column. The weight decays are applied as means of regularization to avoid over-fitting as much as possible. The learning rate (`lr`) on line 18, is same for all models, as mentioned in 3rd column.

The main loop shown on line 23 of Listing 5.2, is responsible for training the ANN models. The training set is 80% of the complete data for each benchmark type operation. It assumes all the data is scaled between 0 and 1 (in addition to random sampling) on line 20 and 21, and uses the following `MaxAbsScaler` formula:

$$x' = \frac{x}{x_{max}} \tag{5.1}$$

where x' is the new scaled value, x is the original value and x_{max} is the maximum absolute value of any configuration parameter given in Table 5.1. For example, as the number of

Table 5.3: ANN's Hyper-parameters.

Benchmark	IO	Learning rate	Dropout	Weight Decay
MPI-IO	READ	0.002	0.0	$1e^{-5}$
	WRITE		0.05	$1e^{-5}$
SEG-Y IO	READ		0.0	0.0
	WRITE		0.05	0.0
SEG-Y Sort	READ		0.05	$1e^{-5}$
	WRITE		0.0	0.0

Listing 5.2: ANN based ML Process over Benchmarks Results.

```
1  import torch.nn as nn
2  def ANN_Model_training():
3
4  # Setting the number of nodes in ANN layers.
5  input_layer, hidden_layer_1, hidden_layer_2, output_layer =
6      7, 256, 128, 1
7
8  # Initializing ANN model with certain dropout ratio on nodes.
9  model = nn.Sequential(nn.Dropout(p=0.00),nn.ReLU(),
10     nn.Linear(input_layer,hidden_layer_1),
11     nn.Dropout(p=0.05),nn.ReLU(),
12     nn.Linear(hidden_layer_1,hidden_layer_2),
13     nn.Dropout(p=0.05),nn.ReLU(),
14     nn.Linear(hidden_layer_2,output_layer))
15
16 criterion = torch.nn.MSELoss()
17
18 optimizer = torch.optim.Adam(model.parameters(), lr=0.002,
19                              weight_decay=1e-5)
20 X = Input_Training_Set # Scaled between 0 and 1.
21 y = Output_Training_Set # bandwidths scaled between 0 and 1.
22 # Training Loop...
23 for epoch in range(MAXIMUM_LIMIT):
24     # Forward Propagation to predict bandwidth.
25     Predictions = model(X)
26     # Compute loss using MSE.
27     loss = criterion(y, Predictions)
28     # Zero the gradients.
29     optimizer.zero_grad()
30     # Perform a backward pass via Back Propagation.
31     loss.backward()
32     # Update the parameters weights
33     optimizer.step()
34     model.train()
35
36 # Save the trained ANN model
37 torch.save(model,"ANN_MODEL.pt")
```

MPI nodes have possible original values $(2, 4, 8, 16)$ therefore, the possible scaled values will be $(0.125, 0.25, 0.5, 1.0)$. The *MaxAbsScaler* is applied on both the input feature parameters (the possible configuration values) and the output feature parameter which is the IO bandwidth value collected on each configuration benchmark execution using Darshan [91]. The input features with two categorised values are explicitly set to 0 and 1 as part of scaling. For example, the original possible values for IO access pattern $(non - collective, collective)$ will be scaled to $(0.0, 1.0)$. If categorised values are three as in case of parameter `unsorted order` in SEG-Y Sorting benchmarks, they are set to $(0.0, 0.5, 1.0)$ against $(uniform, reverse, random)$ values settings.

The Mean Square Error (MSE) value is computed at line 27 as the loss criterion set on line 16, to compute the loss in predicted bandwidth value of ANN's feed forward propagation on line 25, in each iteration. The MSE values are computed using the following equation:

$$MSE = \frac{\sum_{i=0}^{n}(y_i - r_i)^2}{n} \tag{5.2}$$

where y_i is the predicted value of the model and r_i is the real value of the ith test data from n number of test samples from benchmarks results. After computing the loss, the optimizer is being executed on line 29, to zero the gradients by applying the gradient descent method `Adam`, provided by PyTorch ANN's API. After optimizing gradients, the loss is propagated backwards in ANN on line 31, followed by updating the parameters weights on line 33, accordingly. Then on line 34 `model.train()` finally updates itself with updated parameters weights.

Once the ANN models were trained enough, they were saved in their separate respective `ANN_MODEL.pt` files (on line 37). Those trained models were then applied to their respective test sets (20% of the complete data) of their benchmark results, with the intention of predicting IO bandwidths on unseen data. The Section 5.4 shows the overall results of executed benchmarks and the prediction accuracy of the trained ANN based ML models.

5.3.5 Prediction Accuracy Evaluation of ANN Models

The prediction accuracy evaluation of ANN models is being conducted over the testing set of benchmarks results, the 20% of the complete data. This evaluation has been completed using the testing scheme mentioned under the later section of Prediction Results Analysis. The Listing 5.3 represents the overview of the code logic defined under `ANN_Model_Testing()` against the testing scheme. On line 5, the saved trained ANN model is being loaded from the file path `ANN_MODEL.pt`. Then its respective testing set is loaded in `X` and `y` containing scaled input feature values and output feature (IO bandwidth) values respectively, on line 8 and 9. These values are scaled between 0 and 1 using `MaxAbsScaler` formula as mentioned earlier and the ANN models are trained on scaled values of their training sets. The value `n` on line 10 holds the total number of records or rows of the testing set.

Once the required model and its respective test set is loaded then model is being applied on test input feature configuration parameters values on line 13. The new predicted values are being stored in `y_predictions`. Then on line 16 and 17 the previous and predicted bandwidth values in `y` and `y_predictions` respectively, are re-scaled to their actual values by re-ordering Equation 1 as following:

$$x = x' \times x_{max} \tag{5.3}$$

Listing 5.3: Testing ANN models over Benchmarks Results Test set.

```
1  import torch
2  def ANN_Model_testing():
3
4  # Load the saved trained ANN model
5  torch.load(model,"ANN_MODEL.pt")
6
7  # Load Testing Set ...
8  X = Input_Testing_Set # Scaled between 0 and 1.
9  y = Output_Testing_Set # bandwidths scaled between 0 and 1.
10 n = len(y) # Number of Rows in Testing Set.
11
12 # Testing ...
13 y_predictions = model(X)
14
15 # Rescale the bandwidth output and prediction values.
16 y = y * max_io_bandwidth_value
17 y_predictions = y_predictions * max_io_bandwidth_value
18
19 # Mean Absolute Error from Equation 4.
20 MAE = Sum(y - y_predictions) / n
21 # Mean Square Error from Equation 2.
22 MSE = Sum((y - y_predictions)**2) / n
23 # Mean Absolute Percentage Error from Equation 5.
24 MAPE = Sum((y - y_predictions) / y * 100.0) / n
25 # Accuracy from Equation 6.
26 Accuracy = 100.0 - MAPE
27
28 # Breakdown Accuracy values into groups of number of
29 # configurations representing different percentage ranges.
30 Breakdown_Accuracy_values(y, y_predictions)
```

where x represents initial or actual IO bandwidth, x' represents scaled bandwidth value and x_{max} represents the maximum IO bandwidth value recorded from the execution of a complete set of a particular benchmark type in Listing 5.3. After re-scaling actual bandwidth values the MAE, MSE, MAPE and Accuracy values are computed using the Equations 4, 2, 5 and 6 respectively, from line 20 to 26. Then finally mean accuracy values are further broken down into groups of different configuration sets within a test set of a benchmark types. All these errors and accuracy values are presented in Tables 5.4 to 5.8.

5.4 Experimental Result Analysis

The Benchmarks have been executed on up-to 16 ICHEC's KAY Cluster nodes [51] with each 2x 20-core 2.4 GHz Intel Xeon Gold 6148 (Skylake) processors employed on up-to 36 Lustre storage disks. The ML models have been trained and tested on KAY's GPU node equipped with NVIDIA Tesla V100 16GB PCIe (Volta architecture) using PyTorch Tensors framework [184, 22].

5.4.1 Benchmarks Results

The results graph are shown in Figure 5.3 as scatter plots for both READ and WRITE operations for all benchmarks executed with same hierarchical order of nested loops as mentioned in Figure 5.2. This HPC Cluster and its Lustre disks are not standalone and therefore data interference from other users is to be expected. Therefore, these benchmarks were executed over different periods of time repeatedly to ensure consistency. The x-axis on each graph represents the configuration number and y-axis represents the IO bandwidth value in Mega Bytes per second (MB/s).

The configurations shown in Figure 5.3, indicate how alternative configuration numbers shown on the x-axis, an associated bandwidth value (y-axis) can be established. The y-axis represents performance for a particular set of unique parameters which are not shown in the graphs. For this reason, it is not clear which parameter settings are causing certain IO bandwidth performance values.

Therefore, all the following benchmark results were visualised in the parallel plot using HiPlot utility [25]. In particular, the high bandwidth configuration settings are highlighted.

The use of parallel plots in this context is novel and has not been utilised by any previous that have been reviewed in this area. The evaluations explain the overall pattern of IO behaviour, while also highlighting the specific configurations that exhibit high bandwidth scenarios.

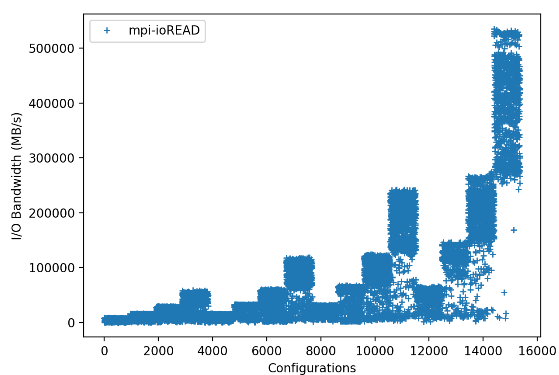
5.4.1.1 MPI-IO Benchmarks Evaluations

As mentioned before the possible combinations for basic MPI-IO benchmark configurations are 30,720 (15,360 for READ and 15360 for WRITE separately). In relation to the READ bandwidth results, the benchmarks are executed with a common nested order given in Figure 5.2 therefore, the outer loop executed for number of MPI nodes values. This means the first main groups of configurations are 4, corresponding to 4 values of number of MPI nodes in 5.1, in ascending order. These 4 groups of MPI nodes values can be seen in Figure 5.3a. Each group is comprised of 4 steps of 960 configurations. These 4 steps in each MPI node value group corresponds to the MPI processes per node from Table 5.1 in ascending order, as the next Nested Loop 1 is executed according to Figure 5.2. This means the steps in Figure 5.3a are an indication that the overall IO bandwidth increases as the MPI processes per node values also increase in each group of MPI nodes. As MPI nodes increase, the size of the steps also increases, which means the IO bandwidth is significantly affected by the number of MPI nodes.

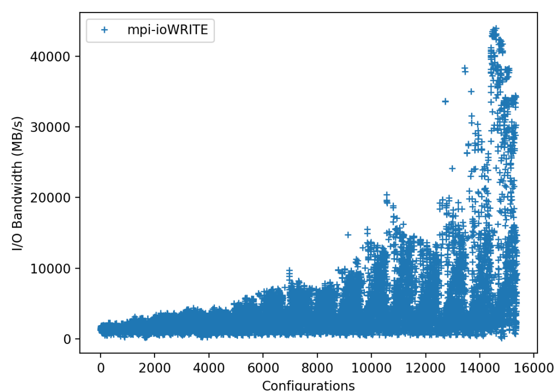
In WRITE bandwidth results the steps cannot be seen (in Figure 5.3b) but the bandwidth is increasing as the overall number of MPI nodes and processes per node increase. The IO (both READ and WRITE) performance is less impacted by the lustre striping parameter values as compared to the overall number of processes.

The data-aligning strategy suggested in [9, 10, 11, 12] does not guarantee sufficient bandwidths where $MPI_nodes < MPI_processes_per_node$. This means that more than one MPI process is running on each MPI node, which keeps the total number of MPI processes the same as in a case of $MPI_nodes >= MPI_processes_per_node$. The total number of MPI processes are equal to stripe count. Bandwidth can be worst affected with the collective IO as compared to non-collective IO.

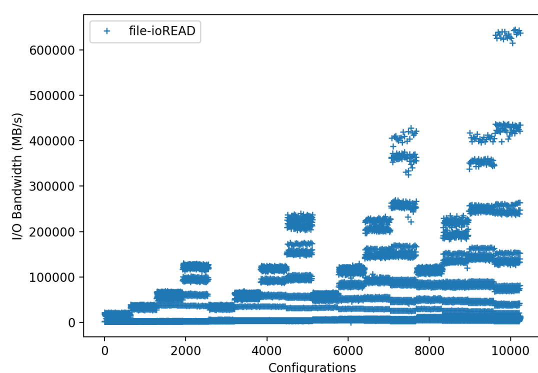
Figures 5.4a, 5.4b present an example scenario where 2 MPI nodes and 8 MPI processes per node (total 16 MPI processes) are reading and writing file data, respectively.



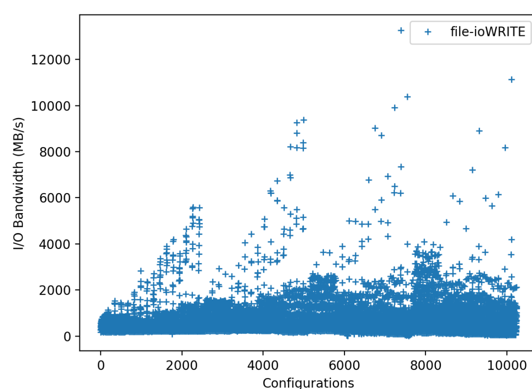
(a) MPI-IO READ Results



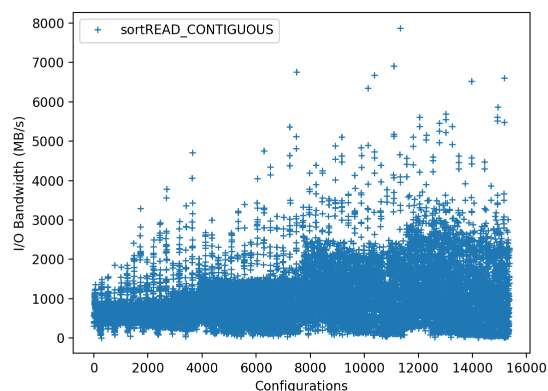
(b) MPI-IO WRITE Results



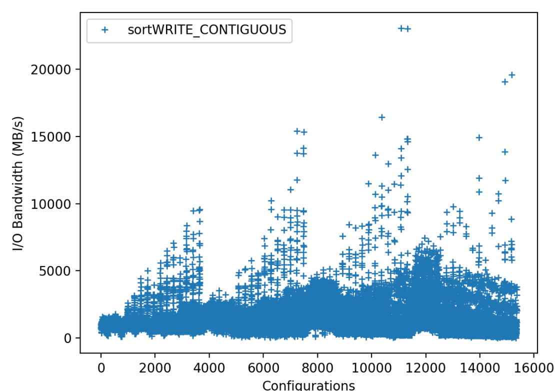
(c) SEG-Y File-IO READ Results



(d) SEG-Y File-IO WRITE Results



(e) SEG-Y File Sort READ Results

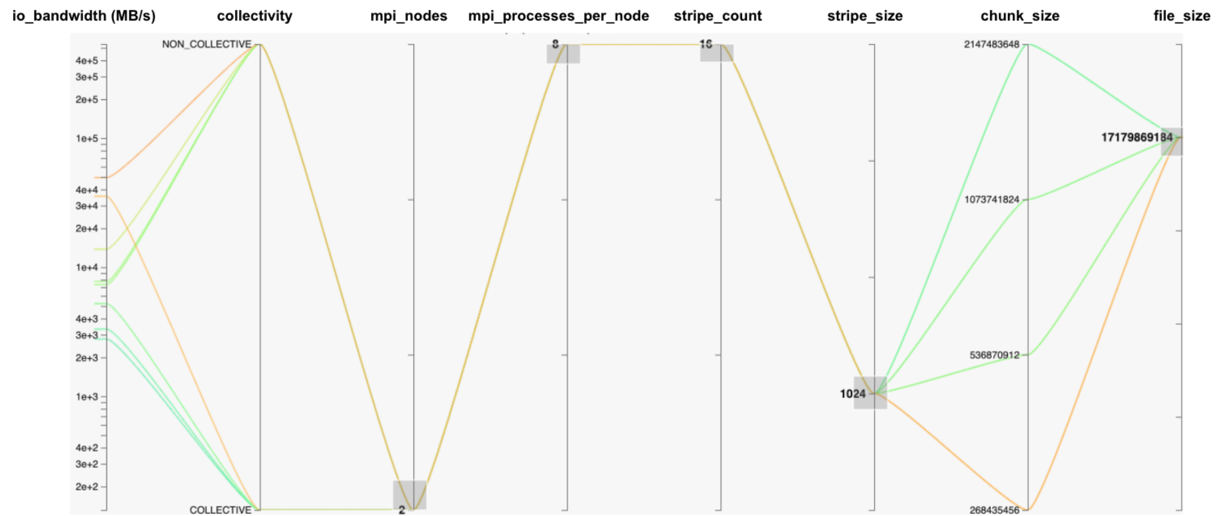


(f) SEG-Y File Sort WRITE Results

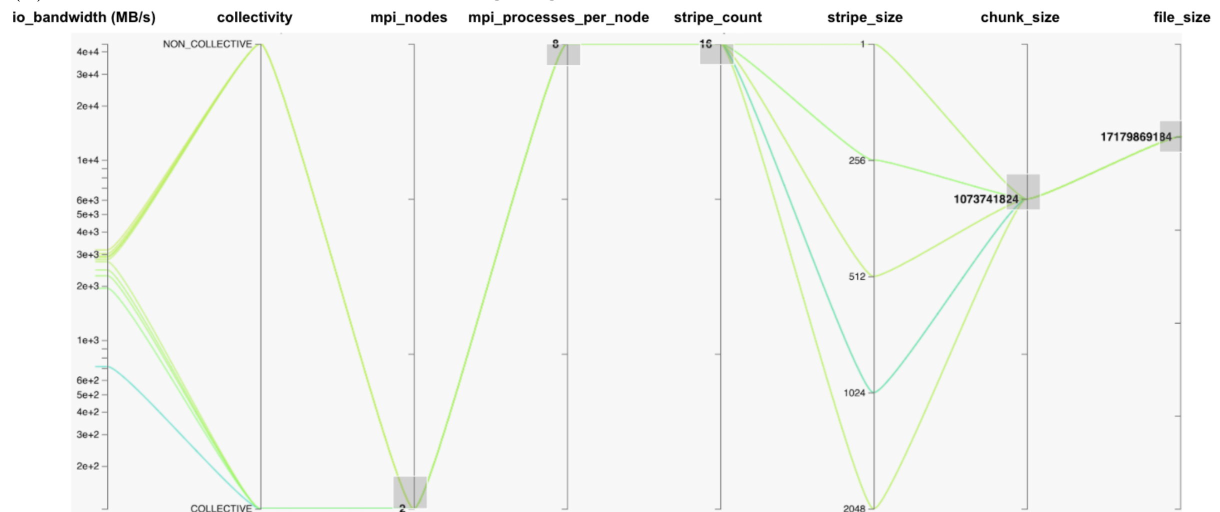
Figure 5.3: Benchmarks Results Graphs.

The low or decreased bandwidth values can be observed for both READ and WRITE operations when 16 MPI processes are aligned with 16 lustre disks (OSTs stripe count value) and stripe size is aligned with chunk size, completing the data-aligning strategy.

These figures also suggest the settings to overcome this problem where stripe size is not aligned with chunk size, in both READ and WRITE operations scenarios. For READ operation in Figure 5.4a lowering the chunk size per MPI processes improves the



(a) Low READ bandwidth of data-aligning with low MPI nodes.



(b) Low WRITE bandwidth of data-aligning with low MPI nodes.

Figure 5.4: Data-aligning strategy showing Low IO bandwidths. **Note:-** stripe_size is represented in the units of Mega Bytes (MBs), chunk_size and file_size are represented in the units of Bytes.

bandwidth. The minimum chunk size gives the highest bandwidth for both collective and non-collective READ operations.

Changing stripe size in this scenario is not considered because it has no benefit at runtime. In this case, the file is already distributed and striped across the disks with certain striping unit (size). The different bandwidth values for different stripe sizes also exists, provided that the file is already generated with the same stripe size.

For WRITE operations in Figure 5.4b, changing or lowering the stripe size doubles the bandwidth from the data-aligned parameters in this case. This jump in bandwidth may not always be the case as the disks are not standalone, due to the continuous interference from other users IO processes on the HPC cluster. However, the IO bandwidth still can be increased with some difference, possibly from the lowest stripe size value (1 MB), as shown in this scenario. The other chunk size values are not shown but they can also make small differences in increasing the WRTIE operation bandwidth.

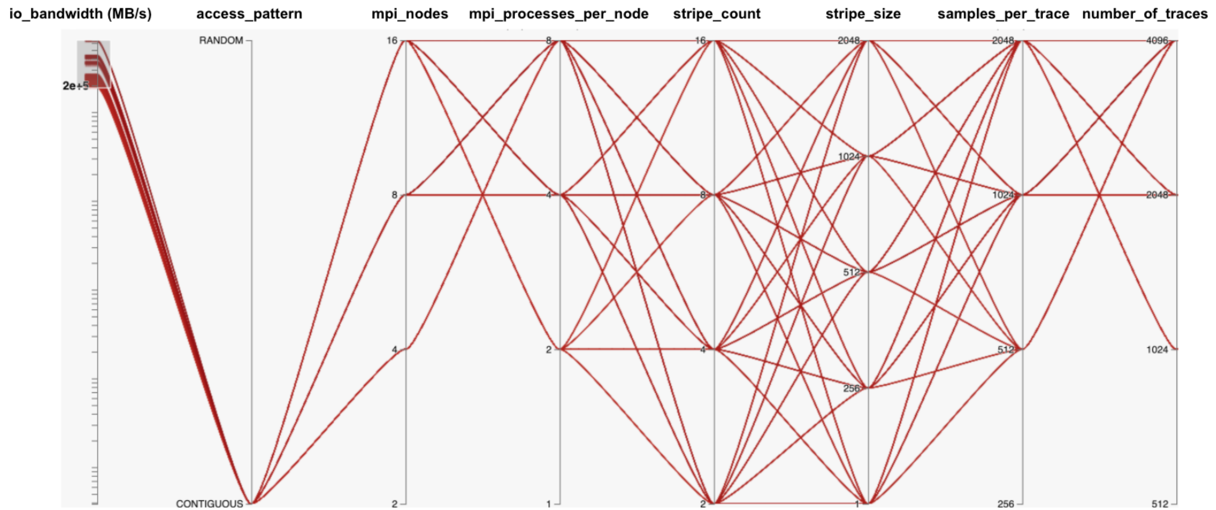
5.4.1.2 SEG-Y File IO Benchmarks Evaluations

For SEG-Y Files benchmarks, READ operations graph (in Figure 5.3c) somehow shows a similar pattern as in simple MPI-IO READ operations in Figure 5.3a. Considering the same nested loop patterns in Figure 5.2, for each number of MPI node values, the bandwidths steps increase as the value of MPI processes per node increases. As the value of MPI nodes increases the size of the bandwidth step also increases.

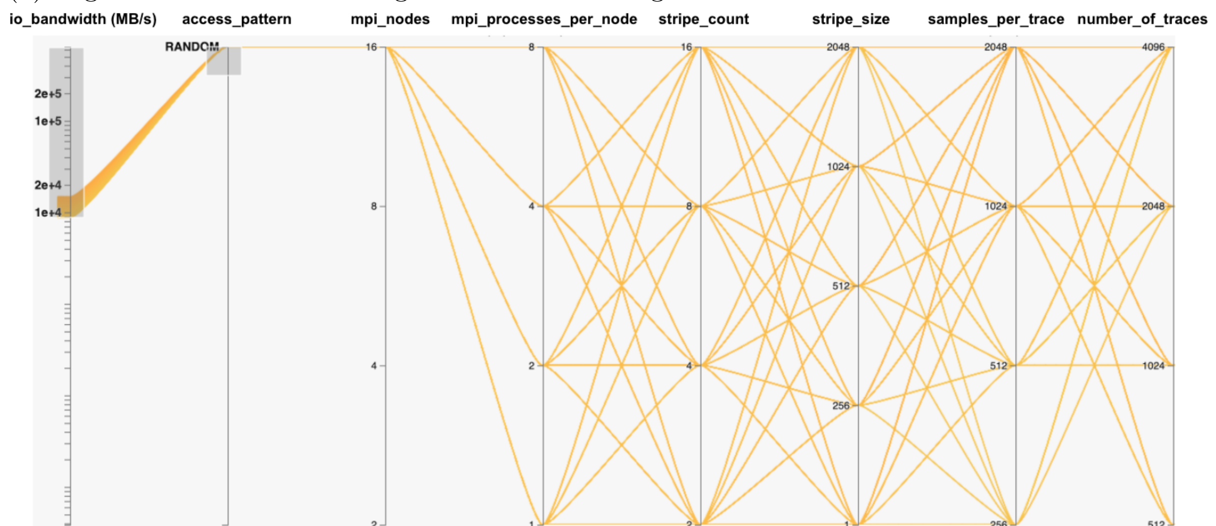
In the case of when the threshold is set at 200000 (MB/s), fewer combinations are observed above this value. When visualized in HiPlot, these configurations have higher MPI process values, higher samples per trace, and higher number of traces values for contiguous READ. This is visible in Figure 5.5a. If random READ is to be visualised, then Figure 5.5b shows the highest MPI nodes value is required for most of the cases. The SEG-Y File WRITE operations graph in Figure 5.3d are different from basic MPI-IO WRITE operations in Figure 5.3b. The reason for this, is the parameters involved in SEG-Y File WRITE are different from the basic MPI-IO WRITE operation.

Keeping the SEG-Y File structure (Figure 3.2) in view, each MPI process would be interested in writing specific parts of the SEG-Y File. In particular, those parts are trace data which follows trace headers of 240 bytes. Normally a chunk size is not known to a process which can be aligned with stripe size. The other factor is the access pattern which can be contiguous or random. But as the striping occurs when writing a file on a parallel file system (LFS), the striping parameters still have some impact on bandwidth.

In Figure 5.3d, the rising peaks can be seen indicating that for each MPI nodes value group, but overall there are numerous low bandwidth configurations, and fewer high bandwidth configurations. It is easier to identify those configuration settings with high or rising bandwidth values. The low bandwidth scenarios occur from value 4000 (MB/s) and below whereas higher ones occur at greater values. Figure 5.6a shows those parameter values and their combinations. It can be seen that 1 MB stripe size (regardless of stripe count), higher samples per trace, number of traces and overall number of MPI processes using contiguous pattern gives higher SEG-Y File WRITE bandwidth values. If random pattern is to be visualised for high bandwidth configuration settings then Figure 5.6b shows the 1 MB stripe size with highest samples per trace, number of traces and MPI nodes values. The number of traces and samples per trace values ultimately define the size of the SEG-Y File. This also means the large SEG-Y File is more likely to give improved WRITE bandwidth as compared to smaller file sizes.



(a) High read bandwidth configurations with contiguous access.



(b) High read bandwidth configurations with random access.

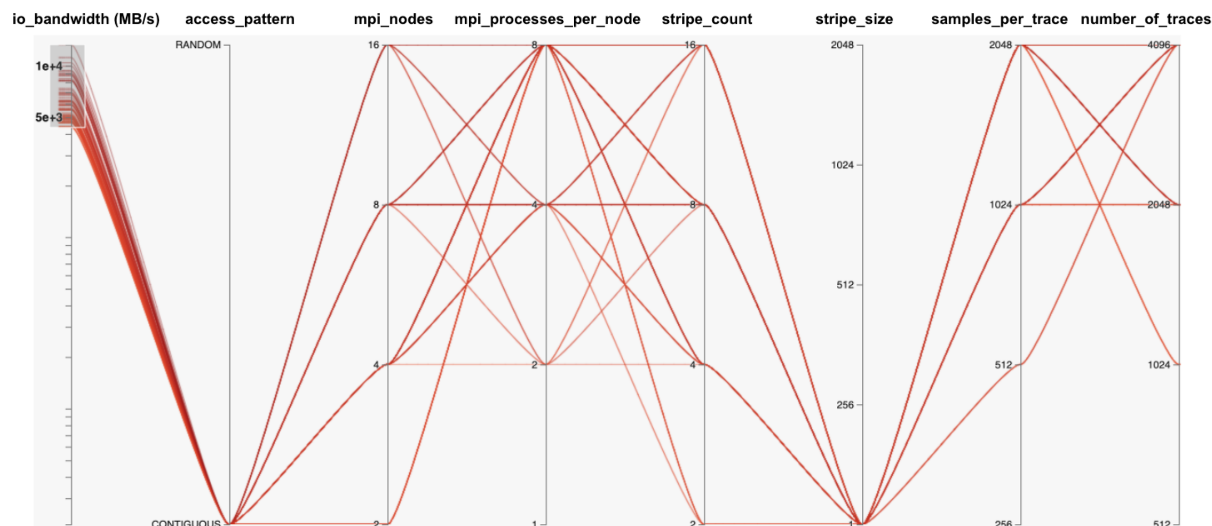
Figure 5.5: High bandwidth configurations settings for SEG-Y File READ operations. **Note:-** stripe_size is represented in the units of Mega Bytes (MBs).

5.4.1.3 SEG-Y file sorting Benchmarks Evaluations

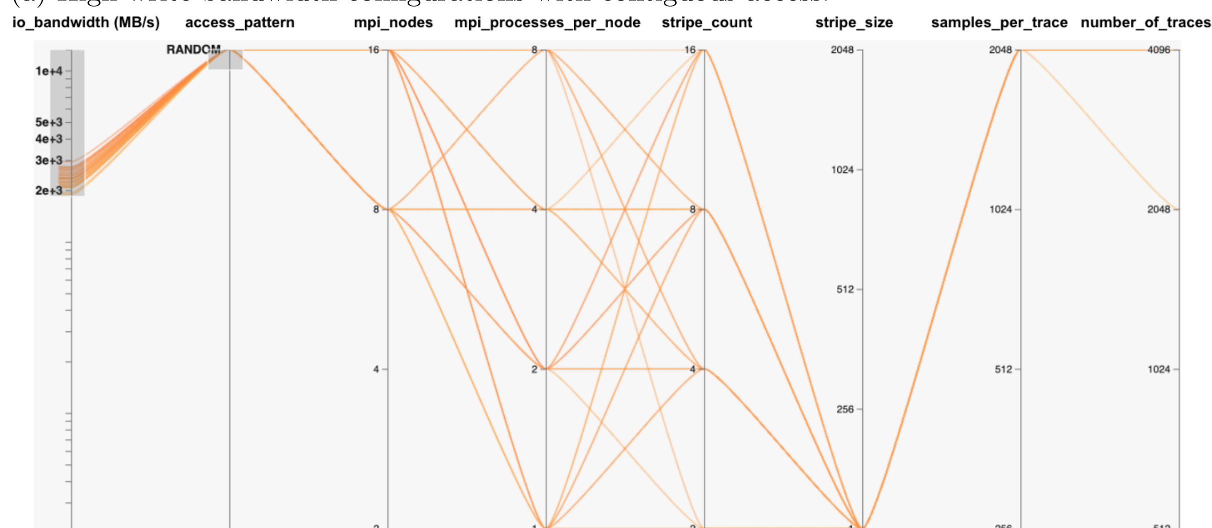
We now examine benchmarks relating to SEG-Y file sorting. These benchmarks read or write a file contiguously that was previously generated using any of these unsorted orders: uniform (ascending order), random and uniform reverse (descending order) with respect to x-source of trace header value in the file.

It is important to then determine which combinations of configuration parameters values give good or high IO bandwidth values for both reading and writing a file contiguously with different unsorted orders. Both results in Figures 5.3e, 5.3f show numerous low bandwidth plots and fewer high bandwidth plots. It is again important to identify the combinations of configurations settings with high bandwidths after certain threshold values, as in the case of SEG-Y File-IO benchmarks findings.

First, considering the READ operation results shown in Figure 5.3e, the highest bandwidth plots occur from 4000 (MB/s), which represents the upper half of the graph. Setting those values in the HiPlot utility gives an insight of all possible combinations in



(a) High write bandwidth configurations with contiguous access.



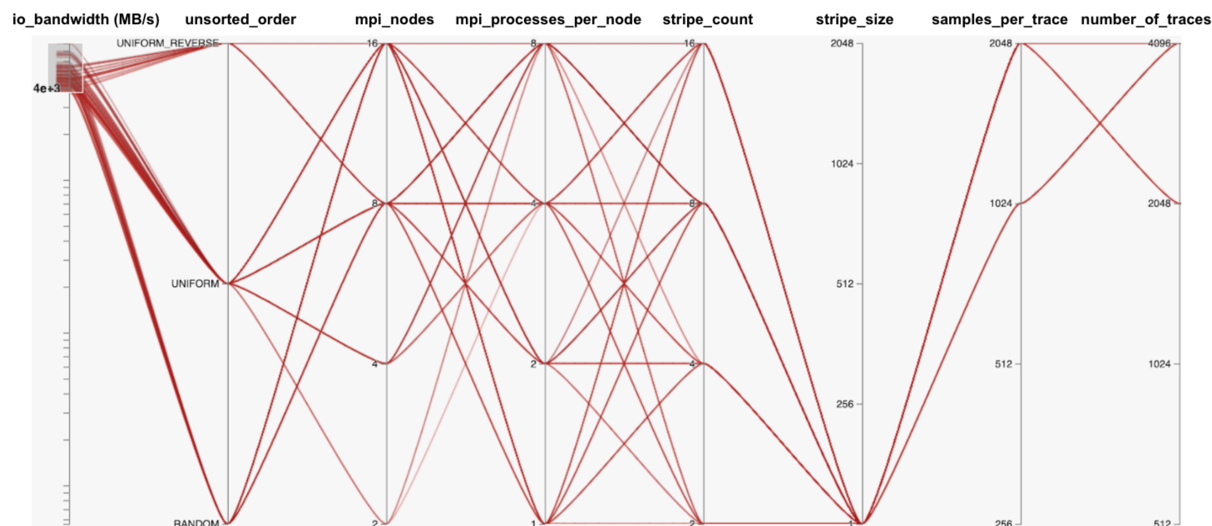
(b) High write bandwidth configurations with random access.

Figure 5.6: High bandwidth configurations settings for SEG-Y File WRITE operations. **Note:-** stripe_size is represented in the units of Mega Bytes (MBs).

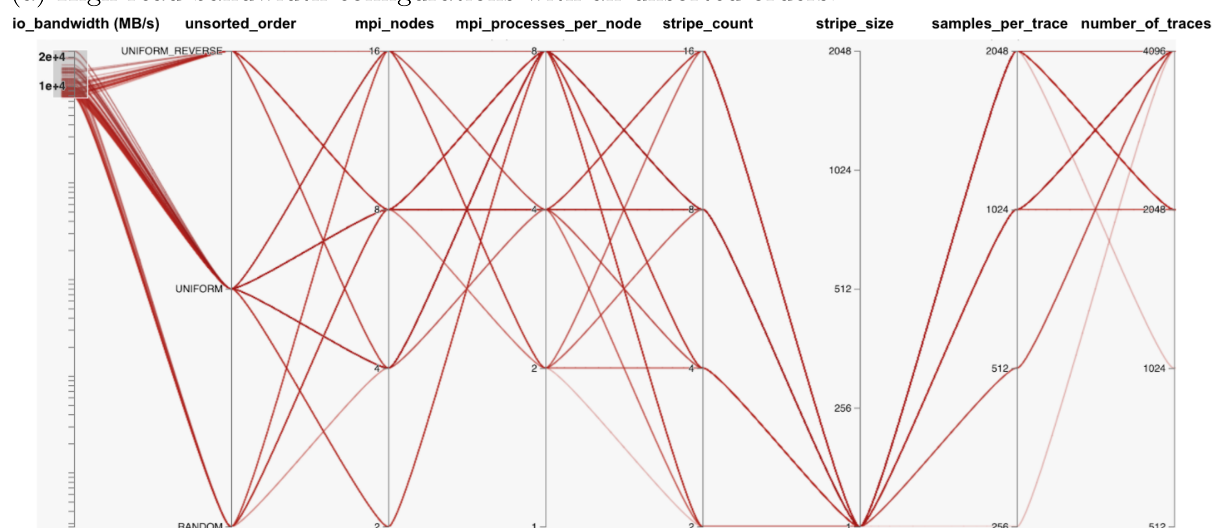
Figure 5.7a. The settings shown are for each unsorted order, have a high total number of MPI processes, and regardless of stripe count, the stripe size should be set to 1MB, and the number of traces with samples per trace should be as high as possible. These configuration settings are shown to yield high READ bandwidths.

With respect to the WRITE operation results in Figure 5.3f, the highest bandwidths plots occur from some values greater than 10000 (MB/s) and above. This is again the upper half of the graph as it was in READ operation results mentioned previously. A visualization of these bandwidth values and combinations in HiPlot is shown in Figure 5.7b. The settings shown are for each unsorted order, have a high total number of MPI processes, and regardless of stripe count, the stripe size should be set to 1MB, and the number of traces with samples per trace should be as high as possible. These configuration settings are shown to yield high WRITE bandwidths. It is worth noting that the approach identified between READ and WRITE operation results was identical.

The data shown in this section has demonstrated that there are no continuously in-



(a) High read bandwidth configurations with all unsorted orders.



(b) High write bandwidth configurations will all unsorted orders.

Figure 5.7: High bandwidth configurations settings for SEG-Y file sorting. **Note:-** stripe_size is represented in the units of Mega Bytes (MBs).

creasing or continuously decreasing patterns in the data that can be assumed. Instead, the changing patterns of IO bandwidth over alternative configurations of parameter values, are observed.

It has been shown that most configurations result in low bandwidth values while only a small number result in more optimal higher bandwidth values. Therefore, it demonstrates the need for an approach that will identify good combinations of settings, which can achieve better overall performance. The challenge is to identify these combinations of settings prior to the execution of any IO operations begin.

For this reason, this research has proposed to apply Neural Networks as a means of making certain predictions about the optimal combinations of settings. Neural Networks are an appropriate approach to allow the system to learn these changing patterns over different features or parameters which effect the IO bandwidth performance so significantly.

Table 5.4: Errors during Training and Testing.

Benchmark	IO	MSE (Training)	MSE (Testing)	MAE (MB/s)
MPI-IO	READ	6.09e-4	6.66e-4	7530.36
	WRITE	4.05e-3	5.04e-3	1244.14
SEG-Y IO	READ	1.04e-6	2.07e-5	1111.05
	WRITE	8.60e-5	1.65e-4	82.34
SEG-Y Sort	READ	7.28e-4	9.99e-4	167.82
	WRITE	1.98e-4	3.59e-4	249.45

5.4.2 Prediction Results Analysis

The ANN models for all the benchmarks have been applied to their respective test sets as means of prediction. The MPI-IO and SEG-Y Sorting benchmarks have 3072 number of configurations tested for each of their READ and WRITE operations. Whereas the SEG-Y IO benchmarks have 2048 tested configurations for each READ and WRITE operation.

5.4.2.1 Testing Scheme

The prediction results of the ANN models on their related test sets are determined via different accuracy metrics, namely: MSE (Mean Squared Error) values, MAPE (Mean Absolute Percentage Error), percentage accuracy and MAE (Mean Absolute error) [185, 186, 187, 79]. This was done to check the accuracy of bandwidth predictions that how fit is the model as compare to bandwidth pattern generated by benchmarks results in Figure 5.3. The prediction percentage accuracy has also been used as a metric for measuring the precision of ML models in couple of previous research works [23, 17, 21]. The Table 5.4, shows the computed MSE values during training and testing, and MAE in IO bandwidth prediction on test set, for each benchmarks READ and WRITE operations models. The MAE in 5th column of this table, MAPE and percentage accuracy in Table 5.5, are computed using the following equations:

$$MAE = \frac{\sum_{i=0}^{n}(y_i - r_i)}{n} \tag{5.4}$$

$$MAPE = \frac{1}{n}\sum_{i=0}^{n}\left|\frac{(y_i - r_i)}{y_i}\right| \times 100.0\% \tag{5.5}$$

$$Accuracy = 100.0 - MAPE \tag{5.6}$$

where y_i is the predicted value of the model and r_i is the real value of the ith test data from n number of samples in test set from benchmarks results data.

In Table 5.4, the MSE values presented after the final testing shown in 4th column are closer to zero and slightly greater than the MSE of training in the 3rd column. This indicates that the trained ANN models for each benchmark type are not under-fitted and very less over-fitted according to [113], which is further supported by Figure 5.8. The MAE values in 5th column represents the mean difference of predicted IO bandwidth values from the actual IO bandwidth values of benchmarks results. These MAE values in predictions are very small in comparison to the range or scale of actual IO bandwidth values given on the y-axis of benchmarks graphs in Figure 5.3. As evident from the table,

Table 5.5: Accuracy of applied ANN models.

Benchmark	IO	MAPE	Accuracy (%)	Accuracy without -ve values (%)
MPI-IO	READ	37.5	62.5	83.5
	WRITE	27.9	72.1	76.3
SEG-Y IO	READ	3.5	96.5	96.5
	WRITE	11.9	88.1	90.4
SEG-Y Sort	READ	23.0	77.0	84.8
	WRITE	20.0	80.0	85.2

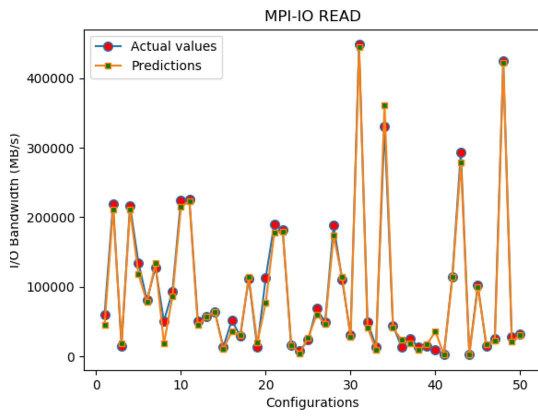
the MPI-IO READ's ANN model gives the highest MAE value despite the fact it mostly follows the pattern similar to actual bandwidth values as shown in Figure 5.8a.

5.4.2.2 Testing Results Analysis

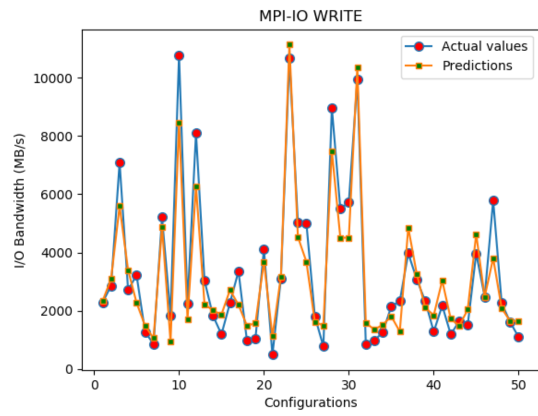
The Figure 5.8 presents 50 randomly selected configurations from the test set with their actual and predicted IO bandwidth values for each model of a benchmark type. It demonstrates that the models can predict the IO bandwidth values according to the pattern or behaviour exhibited by actual values from the benchmarks results in Figure 5.3, for future unseen configuration data.

The accuracy ranges from 62.5% to 96.5% with respect to the generated models, as shown in Table 5.5 - 4th column. This results from the MAPE values ranging from 3.5% to 37.5%, in the 3rd column. There are few cases while making predictions on test set, where predicted bandwidth is larger than the actual bandwidth value, results in negative percentage accuracy values. This affects the overall accuracy of a model but still prediction follows the right pattern in estimating bandwidth, as shown in Figure 5.8. Excluding these cases results in increased accuracy values as mentioned in 5th column. It is exception to the case of SEG-Y IO READ ANN model with the highest accuracy among all the models, which has no case of negative accuracy values when predicting IO bandwidth on its test set configurations.

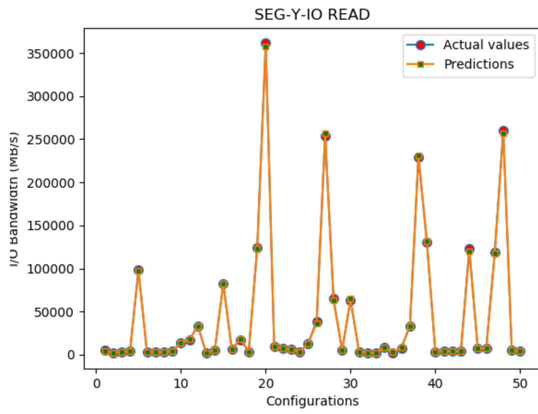
The mean accuracy values are further breakdown in different number of configuration cases of test set in Tables 5.6, 5.7 and 5.8. The 1st and 3rd columns of these tables represents the 11 different groups of number of configurations from the test set. Each group has its own mean accuracy percentage. For example, in Table 6, a group of 484 READ Test configurations have their IO bandwidth predicted with 85.7% Mean Accuracy. This is exception to the case of SEG-Y IO READ tests have few groups as almost all test configurations covered and resulted in the high accuracy values groups . The 2nd and 4th column represents their respective mean accuracy values. This shows that most of the predictions on test set cases tend towards the high percentage accuracy values. There are few with very low or negative mean accuracy values. The Table 5.6 shows ≈7% and ≈3% of the READ and WRITE test configurations predicted with negative mean accuracy values, respectively, shown in the second last row. Similarly, the Table 5.7 shows 0% and ≈1%, and Table 5.8 shows ≈4% and ≈3%, of their READ and WRITE test configurations, respectively, being predicted with negative mean accuracy values. Dividing the sum of all Mean Accuracy values by total tests, for each benchmark type, results in overall accuracy of a model as represented in the 4th column of Table 5.5. Excluding the negative accuracy values and the corresponding test cases from them, results in overall accuracy without negative values, as represented in in the 5th column of Table 5.5.



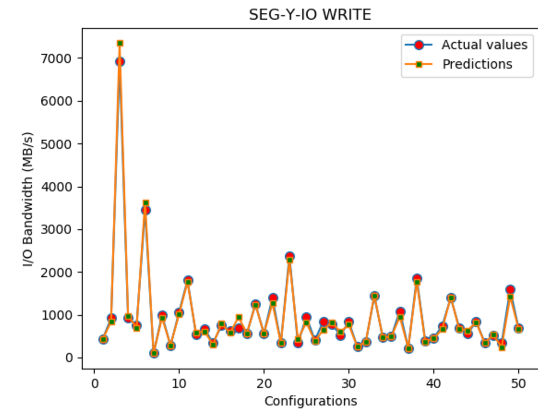
(a) MPI-IO READ Predictions.



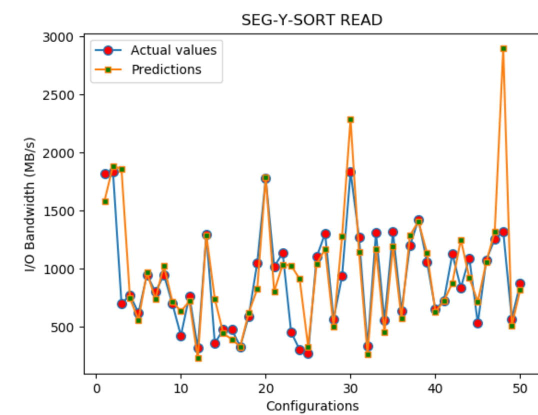
(b) MPI-IO WRITE Predictions.



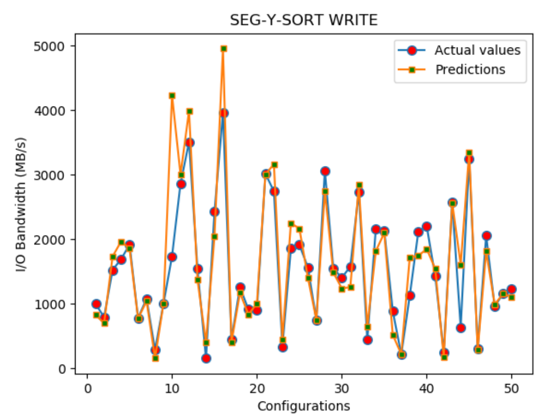
(c) SEG-Y READ Predictions.



(d) SEG-Y WRITE Predictions.



(e) SEG-Y Sort READ Predictions.



(f) SEG-Y Sort WRITE Predictions.

Figure 5.8: Predictions Results using Trained ANN Models.

Table 5.6: Accuracy breakdown for MPI-IO benchmarks prediction models.

READ Tests	Mean Accuracy (%)	WRITE Tests	Mean Accuracy (%)
1580	96.0	794	95.1
484	85.7	726	85.1
273	75.5	612	75.2
192	64.9	358	65.5
99	54.8	216	55.4
74	45.3	109	45.5
54	36.3	69	35.5
39	25.5	50	25.4
28	15.0	33	15.0
36	5.6	22	5.3
213 (≈7%)	-218.2	83 (≈3%)	-79.4
Total=3072		Total=3072	

Table 5.7: Accuracy breakdown for SEG-Y IO benchmarks prediction models.

READ Tests	Mean Accuracy (%)	WRITE Tests	Mean Accuracy (%)
1907	97.4	1371	95.5
118	86.7	444	86.1
18	76.6	120	76.0
4	67.0	48	65.1
1	44.9	21	54.4
-	-	9	46.0
-	-	3	34.5
-	-	2	21.9
-	-	4	13.3
-	-	3	5.1
- (0%)	-	23 (≈1%)	-122.8
Total=2048		Total=2048	

An important consideration is that the trained ANN models are following almost the same trend when predicting the IO bandwidth values on different sets of configurations, as compared to actual benchmarks. It depicts that if one configuration has greater IO bandwidth than the other configuration then, the same behaviour is expected by the prediction model.

Since the trained ANN models are simulating the pattern of changing IO behaviour, their usefulness is to compare the bandwidth values among the different configurations. This feature of these ML models can be used to detect and tune the best configuration settings for any given scenario, prior to the execution of IO processing in the MPI application. This is hugely significant, in the ongoing challenge of more efficiently processing seismic data through SEG-Y Files.

5.5 Discussion

In the previous results section, a series of experiments have been presented. These show that certain configuration parameters are involved in setting high or low IO bandwidths

Table 5.8: Accuracy breakdown for SEG-Y Sorting benchmarks prediction models.

READ Tests	Mean Accuracy (%)	WRITE Tests	Mean Accuracy (%)
1338	95.2	1431	95.1
919	85.6	896	85.8
381	75.9	323	75.9
132	66.1	131	65.8
69	55.6	60	55.7
31	44.4	42	44.4
34	35.1	30	36.0
24	24.7	17	23.7
21	14.5	22	15.1
12	5.4	20	6.3
111 (≈4%)	-129.1	100 (≈3%)	-75.6
Total=3072		Total=3072	

i.e. the MPI processes, lustre striping parameters, size of trace data in SEG-Y file, etc. This information assisted in identifying the exact configuration parameters for the basic MPI-IO file operations and SEG-Y File IO/sorting operations in ExSeisDat.

As per related work, the common practice to improve HPC-IO in different scenarios is the prediction of IO bandwidth over the configuration parameters which can be tuned beforehand to give the improved bandwidth afterwards. It is noted that, perceiving the changing behaviour of IO bandwidth performance is difficult when multiple configurations are involved. The best approach determined for prediction was a ML approach based on ANNs, due to their high accuracy [23, 19, 21].

Once the key parameters were identified, the benchmarks have been executed on the generated list of all possible configurations, to provide training and testing data to the ML models. As the SEG-Y File IO and Sorting are common operations in ExSeisDat apart from basic MPI-IO File operations, the benchmarks are categorised into three types; 1) MPI-IO benchmarks for basic file read and write operations, 2) SEG-Y File IO benchmarks and 3) SEG-Y file sorting benchmarks. The findings on each benchmark, explains the IO behaviour. This is shown by the graphs in Figure 5.3 and highlights the high bandwidth configurations on parallel plots using HiPlot Utility in Figures 5.4, 5.5, 5.6 and 5.7, which has not previously been presented by any state-of-the-art research.

In the case of the remaining configurations with low bandwidths, as shown in section of benchmarks results in Figure 5.3. These were greater in number, and therefore it is difficult to determine the IO pattern that each configuration contributes to increased or decreased performance. This results in the development of the ML ANN models for each benchmark type to predict the IO bandwidth over different configurations. The prediction models are trained and tested through different metrics such as MSE, MAE and Accuracy using MAPE, values. The models learned the trend of changing IO behaviour which is reflected in the prediction results graphs shown in Figure 5.8. As has been shown, the prediction models can predict the IO bandwidth according to the learned trend for future unseen configurations. Therefore, these are hugely beneficial in tuning the configuration parameters for basic MPI-IO operations and SEG-Y operations in ExSeisDat.

Having a well-trained model, the execution path or flow can be completed from prediction to tuning parameters in application before IO operations. The steps of execution can be: 1) Get the current set of configuration values, 2) predict its IO bandwidth 2)

Check different parameters settings and compare their prediction bandwidths values with current predicted value, 3) Choose the settings predicting maximum IO bandwidth value and 4) Tune the configuration parameters with chosen settings. Therefore, it is possible that current or existing parameter settings predict maximum bandwidth out of all other possible settings being checked, and parameters may not be tuned to different values. In relation to tuning parameters on runtime execution of MPI program, it should be noted that not all the parameters are tunable except some of them. For example, in SEG-Y WRITE operation the access pattern, stripe count and stripe size values can be tuned but MPI processes and processes per node cannot be changed as the MPI application is already in execution, and number of traces and samples per trace values cannot be changed being the requirement of a user. Therefore, a set of tunable parameters values should be chosen, predicting higher or maximum IO bandwidth with the given non-tunable parameters values. This also means tuning parameters logic will be different with respect to type of operation. Despite this, eventually the tuning parameters based on the ML prediction can be gainful in a longer term due to forecasting bandwidth beforehand and adapting the optimized configurations.

5.6 Conclusion

In this research chapter, it has been examined that how MPI-IO performance varies in ExSeisDat over SEG-Y files on LFS [1, 4, 8]. It first demonstrates the need for a flexible and efficient optimisation approach through a series of experimental benchmarks. This is similar to much existing research which examines how IO performance can prove challenging in these scenarios [14, 15, 16, 17, 18, 21]. There are two key contributions outlined in this chapter. Firstly, the provision of parallel plots that give clear and succinct insights into the high bandwidth performance of the system across various configuration parameters. Secondly, the application of the ANN based ML approach to the prediction of IO bandwidth using the previously identified benchmarks. The prediction accuracy ranges from 62.5% to 96.5% throughout the trained ANN models for those benchmarks. Previous studies in this area which have been reviewed earlier in this chapter, indicate the limitations of previous approaches to each of these contributions. The results show that these contributions have a significant benefit to practitioners in this field, by contributing major improvements on overall bandwidth prediction, which in itself has many other knock-on benefits with respect to parameter tuning. Thereby, improving the overall efficient completion of tasks for seismic data processing.

Chapter 6

Artificial Neural Networks based Predictions towards the Auto-tuning and Optimization of Parallel IO Bandwidth in HPC System

6.1 Introduction

This research examines the use of Artificial Neural Networks in optimisation of parallel High Performance Computing (HPC) Input / Output. HPC or Super-computing machines are normally clusters of many-core CPU nodes and storage disks. These are inter-connected via fast LAN cables (e.g., Infiniband, Intel OmniPath, etc.) as the medium of communication at hardware infrastructure level [5, 6]. The Figure 2.1 shows the typical overview of a HPC cluster hardware structure. Parallelization has been highly effective in exploiting the power of a cluster via Message Passing Interface (MPI) standard software library written on C/C++ platform [7]. MPI is a distributed memory parallel programming framework that comes with a broad set of functionalities to utilize a cluster's parallelism potential. When an application using MPI is built, the MPI-IO part within the program is often leftover to consider for optimization thus, causes the poor IO throughput, resulting in the overall performance issues. Since, the IO side needs to be manually tuned by different configurations for performance optimization, this creates an extra overhead at the user end to perform this task. It becomes even more challenging for the user to select particular settings since not knowing the IO performance outcome. To overcome this challenge, in this research, we have proposed the technique to predict the IO bandwidth performance and auto-tune the related configuration parameters before an IO operation. This extra software layer between user and MPI-IO operation takes away the burden of manually tuning the configuration settings. Additionally, the powerful Artificial Neural Network (ANN) based ML prediction model supports in selecting new value settings with highly expected improvement in parallel IO performance.

Despite MPI being an efficient parallel computing framework, the parallel MPI-IO side of the HPC cluster suffers from application performance degradation. The first reason is the slow pace of development in the advancement of IO storage processing hardware in contrast to computing hardware. The second reason is, in HPC systems parallel IO depends on particular parameters to determine its bandwidth from the software side. These critical parameters are related to different components which support parallel IO

on the clusters. Parallel IO and storage are normally managed by a specific Parallel File System (PFS) in the clusters at a low level. In this research case, the PFS is Lustre File System (LFS) [8]. The common factors which effect IO bandwidth performance are; the number of MPI processes, the number of parallel discs, the file access patterns and other properties. These factors and IO optimizations are not generally taken into account by the parallel application programmers. This results in poor IO and consequently overall application performance degradation. The parallel IO side is often neglected and left solely to manual tuning by researchers to devise techniques for the improvement of parallel IO at the software side, as being done in the past [9, 10, 11, 12]. These approaches suggest the different strategies around data-alignment based configuration settings that improve IO but do not cover all the scenarios as explained in the previous chapter. In this research, the approach is required to optimize IO for the maximum possible scenarios within MPI-IO applications. To achieve this, the proposed technique is to auto-tune parameter settings from current values that estimate maximum possible bandwidth values, before any IO execution.

The need of HPC and Machine Learning (ML), is crucial to areas such as Internet of Things (IoT) based smart city environment, that critically relies on the efficient data IO, and has shown recent advancement [188, 189, 190, 191, 192]. In this research, the ML technique is applied to HPC MPI-IO operations, for the bandwidth performance optimization. It is the extension of the previous work completed in chapter 5, in regard to MPI-IO operations bandwidth prediction only, on the related parameters settings. In extending this previous work, our innovation lies in auto-tuning those parameters on the basis of MPI-IO bandwidth prediction to the optimized values. Previous research studies show significant benefits of ML based IO performance prediction and auto-tuning over different parameters settings at different environment experimental setups [14, 15, 16, 17, 18]. This has provided the motivation to apply similar approach to improve the MPI-IO application bandwidth performance, in this research scenario. Therefore, the prediction models are generated through the ML process, and used in this research to support the auto-tuning approach. The work presented in [19, 20, 21, 22, 23], explains the basic working of ANNs as ML technique, application to improve file access times in HPC, and shows the significant prediction accuracy of using ANNs, specially when models constructed in PyTorch framework. This provided us further motivation to use ANNs based ML technique with PyTorch API. In result of this, the generated ANN models in this research, have proven significantly beneficial for optimizing the MPI-IO performance through auto-tuning parameter settings against the default configuration test cases.

Initially, the MPI-IO READ/WRITE benchmarks are re-generated and re-executed for bandwidth profiling over the related parameters with possible settings, as mentioned in chapter 5. This was to ensure that the similar IO bandwidth patterns are replicated from varying the configuration settings. Then the separate ANN models for READ-/WRITE operations, are re-generated, having the precise similar prediction accuracy to the previous work, as stated in the earlier chapter of the thesis. Subsequently, the auto-tuning strategy is designed based on those ANNs, which is a critical first contribution of this research chapter. This research presents a ANN ML based approach to select and auto-tune the parameters settings to get optimized bandwidth. The results have shown notable improvements in bandwidth through a detailed statistical analysis which represents the second key contribution contained in this research. Finally, since having significant IO bandwidth performance outcomes, the most beneficial configurations have been noted and highlighted in detail as the third key contribution.

The results presented later in this research chapter, demonstrate that the proposed auto-tuning strategy using ANN models contributes to a significant gain in IO performance. The remaining structure of this chapter follows with the Related Literature in section 6.2, Research and Design Strategy in 6.3, Performance Evaluation in 6.4, Summary of the Work in 6.5 and Conclusion in 6.6.

6.2 Background and Related Research

Earlier research studies have been explored, involving the use of ML based predictive IO modelling, and how best to address poor IO performance issues. This is primarily achieved by tuning related parameters on certain cluster environments. These research studies have been a motivation towards auto-tuning of the related parameters in this research scenario, predicting the maximum possible IO bandwidth performance for files striped across parallel networked Lustre disks in HPC clusters.

In earlier chapter, it was shown that the parameters relating to MPI, LFS and file properties can cause increases or decreases in IO bandwidth. Therefore, ANN models were created to predict IO bandwidth performance for the MPI-IO READ and WRITE operations scenarios. The models' prediction accuracy values were 62.5% and 72.1% for READ and WRITE operations, respectively. The parallel HiPlots have also been used to visualize bandwidth changes with different parameters settings values [25].

The work by Xu et al. in [14], the study demonstrates that parameters like the IO scheduler, number of IO threads and CPU frequency affects HPC-IO performance. The IO behaviour is predicted and determined upon these factors as parameters through extrapolation and interpolation techniques. This was completed by large-scale experiments using a data analytics framework. Their performance evaluations were conducted by prediction accuracy calculations with the unseen testing system configurations. Afterwards, the system used a Bayesian Treed Gaussian Process variability map with different regression techniques. This supported parameter selection through HPC variability management and insights of current statistical methods.

The method demonstrated by Bez et al. in [15], allows for the adaptive scheduling of parallel IO requests within a HPC system running application. This was achieved by tuning the time window based parameters of the current executing workload. This adaptive technique is implemented through reinforcement learning by the scheduler. It achieved 88% precision in parameter selection on runtime after access pattern observation and classification using neural networks by the system for few minutes. Subsequently, its IO performance would be optimized by the system for its remaining lifetime, as declared in the study.

The work presented by Bağbaba in [17], examines bandwidth predictive modelling for MPI WRITE Collective operations via random forest regression. The prediction accuracy values are extremely high, ranging around 82-99%, which depends on depth setting maximum value. The datasets for training and testing were relatively small, which would be require more examination. Having greater variation in the data, the accuracy values could be significantly lowered.

In the research presented by Behzad et al. in [16], the IO performance optimization of parallel applications was proposed for HDF5 format files. It was tested on a range of varied HPC clusters employing LFS and General Parallel File System (GPFS). The auto-tuning played a key role on the basis of predictive IO modelling. The predictive models

were trained with Lustre IOR and other benchmarks data using a nonlinear regression technique. The IO performance notably increased with new parameters selection via auto-tuning. Comparatively, the work by Madireddy et al. in [18] also demonstrated predictive IO modelling for LFS IOR benchmarks but using a Gaussian process regression (GPR).

The work carried out by Schmidt et al. in [21], performs prediction for time of file accesses on LFS storage disks. The file access time for a series of tests are recorded to be used for developing prediction models. The evaluation shows the generated ANNs yield 30% less error in average prediction as compared to linear predictive modelling. The file access times distribution and its evaluation were conducted with regard to similar parameters used to access files. It also discovered that the typical file access times usually differed by a degree of magnitude, depending on the different IO paths.

Additionally, some other research studies were inspected in the area of MPI application optimization using ML prediction and auto-tuning parameters, however, the IO side was mostly ignored [181, 182].

6.3 Experimental Design and Implementation

This research has been conducted by applying a series of steps in sequence. 1) identification of the key parameters 2) the generation of benchmark data 3) generating 2 ANN based prediction models for each READ and WRITE operation, 4) designing and applying the auto-tuning strategy, 5) analysing IO bandwidth improvements statistics and 6) identifying the common configuration settings selected by auto-tuning the system model. The Figure 6.1 explains the main components that summarise the sequence of steps involved in our proposed approach with the experimental setup, further elaborated and explained in Listings 6.1 to 6.5.

6.3.1 Identification of Key and Tunable Parameters

Initially, the key configuration parameters must be identified with fixed possible value settings. This is an essential step prior to the execution of READ/WRITE bytes of data from the file on disk in the series of benchmarks. Each benchmark execution should be completed in order to profile bandwidth, as an output against each READ/WRITE operation with a specific set of configuration values settings. This bandwidth profiling data is then split and used as training and testing sets separately. The data is split in a 80:20 ratio of the total benchmarks results for these training and testing sets. This approach then allows training and testing the ANN models. It should be noted that configuration parameters are either tunable or non-tunable, when considering them for auto-tuning in later.

The Table 6.1 contains the identified key tunable and non-tunable parameters against their corresponding values settings specified for benchmarks execution. The total number of value settings available to benchmark is shown to total 30720, which gives an indication of the scale of configurations for the machine learning process. However, the `IO operation` specified in this table is not essentially a configuration parameter. This is to just specify that these configurations are executed for both READ and WRITE operations separately. They are marked as "Not Applicable" from the tunability point of view. Arising from this, the total is 15,360 possible configurations $(30,720/2)$ for each operation type (READ & WRITE). Thus, this is an intuitive process for the generation

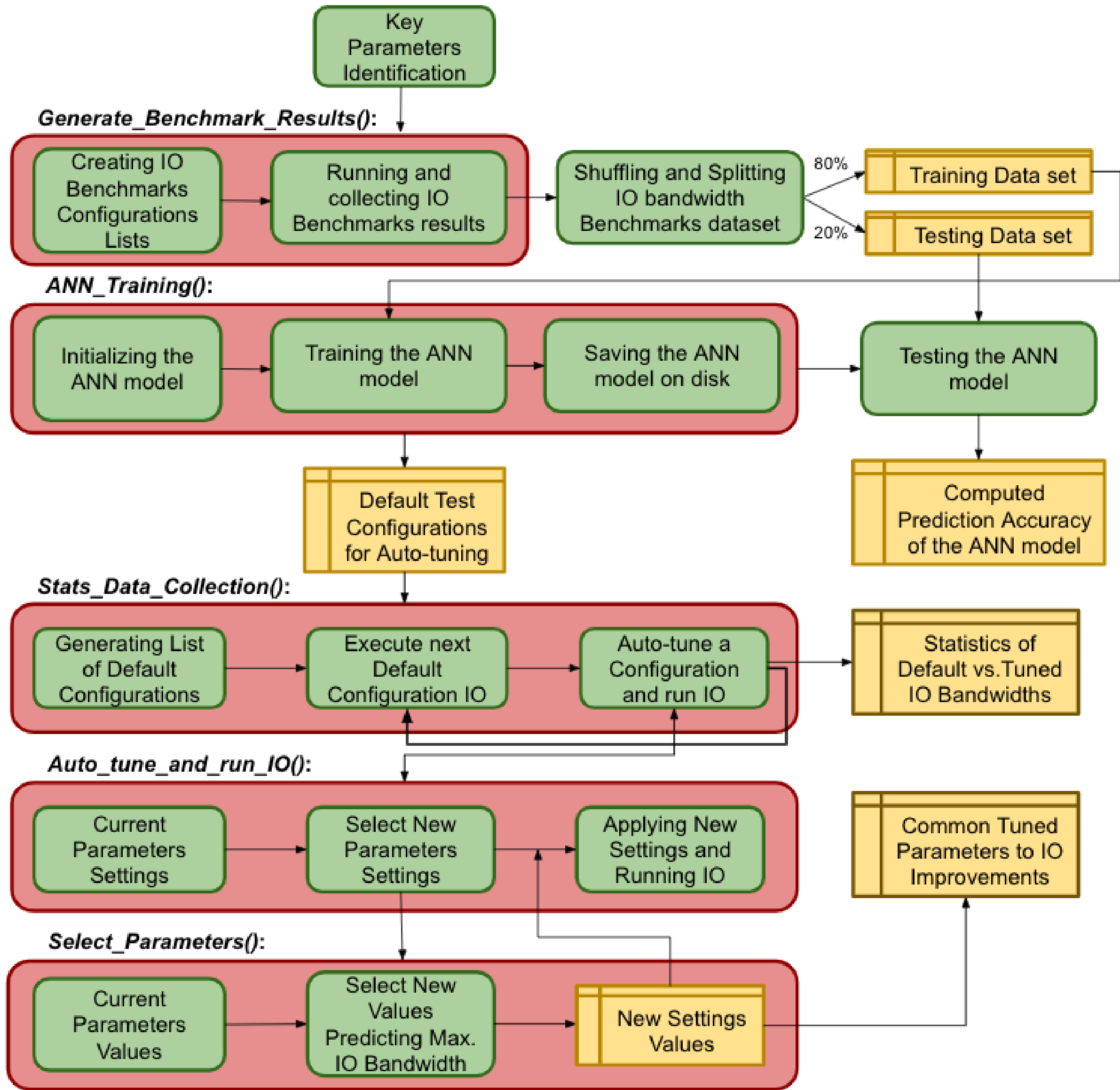


Figure 6.1: Research Methodology of the Experimental Setup to train the ANN ML models and auto-tune the configuration parameters settings based on the IO bandwidth predictions.

of datasets relating to both the READ and WRITE bandwidth patterns. This can then be modelled by two separate ANN models for each operation type.

To summarise, a single configuration setting row in a list of dataset can be represented as follows:

$$['number_of_MPI_nodes = 16','MPI_processes_on_each_node = 1',$$
$$'lustre_stripe_count = 4','lustre_stripe_size = 1MiB',$$
$$'file_size = 32GiB','chunk_size = 0.5GiB','file_access_pattern = collective'],$$

where it runs for a READ and WRITE operation separately. Similarly, all configuration settings in that list are fetched and benchmarked iteratively and separately for both IO operations (READ/WRITE).

Table 6.1: Tunable/Non-tunable Parameters with possible values settings.

Configuration Parameters	Values Settings	READ/WRITE Tunability
Number of MPI nodes	2,4,8,16	No/No
MPI processes on each node	1,2,4,8	No/No
Lustre stripe count	2,4,8,16	No/Yes
Lustre stripe size (MiBs)	1,256,512,1024,2048	No/Yes
File size (GiBs)	1,2,4,8,16,32	No/No
Chunk size (GiBs) per process	0.25,0.5,1,2	Yes/Yes
IO operation	read,write	Not Applicable
File access pattern	collective,non-collective	Yes/Yes

6.3.2 Generating MPI-IO Benchmarks Results Data

In this section, the procedure is described for generating MPI-IO benchmarks in order to then use this as a training and testing set for the ANN models.

The Listing 6.1 shows the method `Generate_Benchmark_Results()` used to generate IO bandwidth profiling data by means of benchmarking against each configuration setting. It takes `Configuration_Settings` as an argument, which has the list of value settings for each parameter. The first step (Line 2) is to generate a complete list of all possible configuration settings to be benchmarked. Each row of (MPI_IO_Configs_List) have one set of configuration settings, as stated by the example mentioned in Section 6.3.1 previously.

During the execution of MPI-IO benchmarks, the MPI processes read/write the corresponding file chunks. Prior to execution, the file is striped and generated over `stripe count` of LFS employed disks with the specified value of `stripe size` in the case of READ operations. This is stated from Line 11 to 16. For WRITE operations these Lustre settings are applied to an empty file beforehand.

When the file is written on disks it is distributed on the same `stripe count` of disks with the `stripe size` value. A single MPI read/write execution can be `collective` or `non-collective`. In the `collective` operation each process accesses other running parallel processes address space, therefore making a non-contiguous access. Whereas, in `non-collective` operation each MPI process can have access to its own address space only and keeps the access contiguous.

Before every benchmark execution the pre-benchmarks settings are applied in regard to LFS striping as mentioned earlier, and the Darshan utility. Darshan is the tool for characterization of HPC IO for capturing bandwidth during execution [91], as the setting applied on Line 18.

At this point, pre-benchmark settings are applied as a means of measuring the bandwidth performance of a specific configuration in regard to specific file striping. Once the execution of a configuration set completes on Line 21, the IO bandwidth performance gets captured by Darshan and parsed to retrieve on Line 24. The benchmarked bandwidth value for each IO execution with its configuration parameters values settings are added at the end of file in the YAML dictionary style format on Line 25.

Afterwards, the post-benchmark setting is applied to that benchmarked file by deleting it to free the cache, on the last Line 28. This completes a single benchmark iteration over a configuration set of parameter settings for each READ and WRITE operation.

The following Figure 6.2 shows the benchmark results which depict the bandwidth

Listing 6.1: Overview of Code Executing IO Benchmarks and Collecting Bandwidth Data.

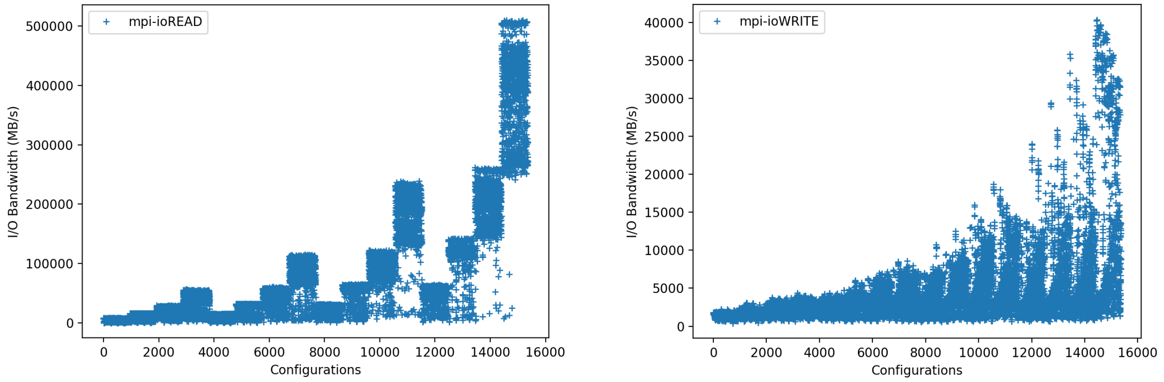
```
1  def Generate_Benchmark_Results(Configuration_Settings):
2  MPI_IO_Configs_List = Generate_List(Configuration_Settings)
3  for each config_row in MPI_IO_Configs_List:
4
5      # A config_row represents a single unique configuration
6      # setting as stated by example in section 6.3.2 in the second
7      # paragraph. However, it additionally contains other
8      # attributes i.e. target file name, darshan file path, etc.
9
10     # Applying pre-benchmarking steps before a config execution.
11     Set_File_Striping(config_row['file_name'],
12                       config_row['lustre_stripe_count'],
13                       config_row['lustre_stripe_size'])
14
15     if(config_row['io_operation'] == "READ"):
16         Generate_File_to_be_read(config_row['file_name'])
17
18     Set_Darshan_File_Path(config_row['darshan_file'])
19
20     # Benchmarking IO of the current config_row values.
21     Execute_Benchmark(config_row)
22
23     # Append the bandwidth value in YAML file after it is parsed.
24     bandwidth_value = ParseDarshanFile(config_row['darshan_file'])
25     Append_Output_in_YAML_File(config_row, bandwidth_value)
26
27     # The last (post-benchmark) step is to delete the target file.
28     Delete_Benchmarked_Target_File(config_row['file_name'])
```

patterns for all configuration settings available. In general, the bandwidth values increase with respect to the increasing number of MPI nodes and processes per node.

6.3.3 Creation and Development of ANN models

In this section, the training and development of ANN models is presented, that are used for bandwidth predictions. These ANNs have two hidden layers with 256 and 128 nodes respectively, as outlined in Table 6.2. The input layer and output layer has 7 and 1 nodes against the 7 input feature parameters from Table 6.1 and 1 output feature as the predicted IO bandwidth value in all models. The Table 6.3 shows the mappings of 7 input layer nodes to the 7 configuration parameters. The 1 output layer node maps to the IO bandwidth value. The `IO operation` parameter from Table 6.1 is excluded because the models are created separately for READ and WRITE operations. Therefore, there are two distinct ANNs models, ANN_{READ} and ANN_{WRITE}. Table 6.4 represents the values for hyper-parameters being used during the ANNs training, to support the accuracy during validation.

The ANN models are developed and trained using PyTorch ([22]) through the pseudo-code `ANN_Training()` in Listing 6.2. The hyper-parameters values are applied according to Table 6.4. Prior to the ANNs training, the MPI-IO benchmarks results data is scaled using `MaxAbsScaler` ([111]). Afterwards, data is shuffled and split between training and testing sets. For each ANN model 80% of benchmarks results are used for training



(a) READ Bandwidths. (b) WRITE Bandwidths.

Figure 6.2: MPI-IO Benchmarks Results.

purposes and 20% for testing.

Initially, Line 5 to 14 initialized the ANN model with random weights, where 0.05% dropout is applied on the hidden layers in the case of training ANN_{WRITE}, and 0.00% for ANN_{READ}. Additionally, the Rectified Linear Unit activation function (`nn.ReLU()`) is also applied from layer to layer in order to get the expected decimal value at the output [183]. The loss function is defined as the `MSELoss()` on Line 16 to compute the loss between actual and predicted value during training, which is a standard Mean Squared Error (MSE) [185]. The other hyper-parameters are; learning rate 0.002, and weight decay `1e−5` are applied on Line 18. This relates to using (Adam) as the gradient descent optimizer. Then `X` and `y` are set as the input and output features from the training set on Line 20 and 21.

Subsequently, the Line 23 to 34 control the main loop which trains the ANN model until the specific number of iterations defined as `MAX_Limit` is reached. During loop iterations, the bandwidth value is predicted on Line 25. Then, Line 27 computes the loss by MSE value. The `Adam` optimizer zeros the gradients on Line 29 before the back propagation can occur. Once complete, loss is propagated backward on Line 31 and the optimizer updates the weights which are eventually updated by the model calling its `train` function on Line 34.

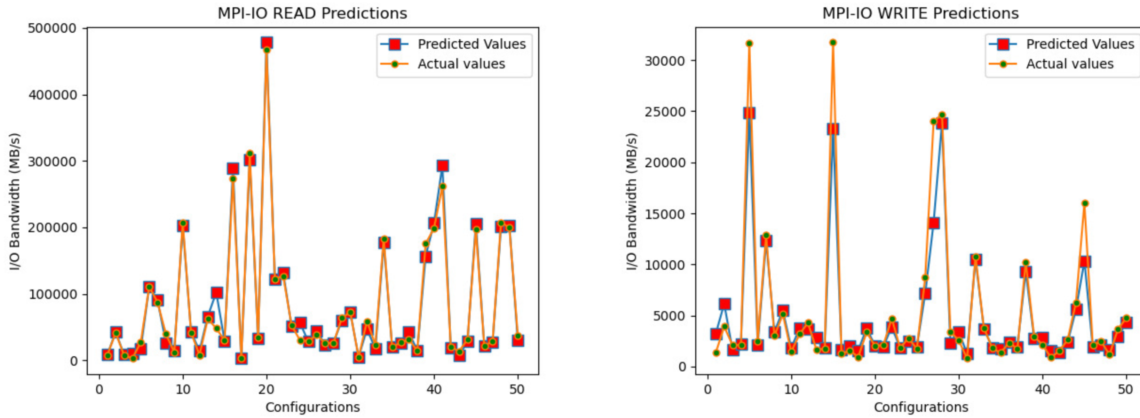
Once the model is trained it has been saved in `".pt"` file on Line 37. It is then loaded back to memory at the time of prediction to support the auto-tuning process later.

The ANN models run on the test set, and the predicted values are re-scaled to their original values to compute accuracy with the profiled benchmark bandwidth values. The prediction accuracy denoted by $P.A$ for both ANN models, is defined by the following equations:

$$
\begin{aligned}
Let\ X &= testing\ set, \\
let\ y &= actual\ bandwidth\ values\ against\ X, \\
r &= model(X)\ gives\ predicted\ values\ against\ X, \\
P.A &= 100.0 - \frac{1}{n}\sum_{i=0}^{n}\left|\frac{(y_i - r_i)}{y_i}\right| \times 100.0\%,
\end{aligned}
$$

where y and r are actual and predicted n number of total bandwidth values, respectively, and i is the ith row of X, y and r thus, y_i is the ith actual bandwidth value and r_i is the ith predicted bandwidth value computed by running $model()$ on X_i the ith configuration parameters values set.

The prediction accuracy of ANN_{READ} yields $\approx 63\%$, and for ANN_{WRITE} yields $\approx 79\%$. The Figure 6.3 shows the model predictions on a randomised 50 configuration settings for each READ and WRITE operations. It is evident from these figures that both ANN models are well fitted for future predictions on unseen configuration data. Later on, these models play a vital role in auto-tuning the parameters required for overall bandwidth performance gain.



(a) READ Predictions. (b) WRITE Predictions.

Figure 6.3: MPI-IO Bandwidth Predictions.

Table 6.2: ANNs Description table to Model READ/WRITE bandwidth behaviour.

Input nodes	hidden layer-1 nodes	hidden layer-2 nodes	Output nodes
7	256	128	1

Table 6.3: ANNs Input and Output Nodes Mapping to Configuration and Output Parameters for READ and WRITE Bandwidth Predictions.

Input Nodes	Configuration Parameters
Node - 1	Number of MPI nodes
Node - 2	MPI processes per node
Node - 3	Lustre Stripe Count
Node - 4	Lustre Stripe Size
Node - 5	File Size
Node - 6	Chunk Size
Node - 7	File Access Pattern
Output Node	**Output Parameter**
Node - 1	IO Bandwidth

Table 6.4: Hyper parameters applied to ANNs.

ANNs IO modelling types	Dropout(%)	Weight decay	Learning rate
READ	0.00	$1e^{-5}$	$2e^{-3}$
WRITE	0.05		

Listing 6.2: ANN based ML Process over Benchmarks Results.

```
1  import torch.nn as nn
2  def ANN_Training():
3
4  # Setting the number of nodes in ANN layers.
5  Input_Layer, Hidden_Layer_1, Hidden_Layer_2, Output_Layer =
6       7, 256, 128, 1
7
8  # Initializing ANN model with certain dropout ratio on nodes.
9  Model = nn.Sequential(nn.Dropout(p=0.00),nn.ReLU(),
10      nn.Linear(Input_Layer,Hidden_Layer_1),
11      nn.Dropout(p=0.05),nn.ReLU(),
12      nn.Linear(Hidden_Layer_1,Hidden_Layer_2),
13      nn.Dropout(p=0.05),nn.ReLU(),
14      nn.Linear(Hidden_Layer_2,Output_Layer))
15
16 Criterion = torch.nn.MSELoss()
17
18 Optimizer = torch.optim.Adam(Model.parameters(), lr=0.002,
19                                weight_decay=1e-5)
20 X = Input_Features_Set # Scaled between 0 and 1.
21 y = Output_Feature_Set # bandwidths scaled between 0 and 1.
22 # Training Loop...
23 for Epoch in range(MAX_LIMIT):
24      # Forward Propagation to predict bandwidth.
25      y_predicted = Model(X)
26      # Compute loss using MSE.
27      Loss = Criterion(y, y_predicted)
28      # Zero the gradients.
29      Optimizer.zero_grad()
30      # Perform a backward pass via Back Propagation.
31      Loss.backward()
32      # Update the parameters weights
33      Optimizer.step()
34      Model.train()
35
36 # Save the trained ANN model
37 torch.save(Model,"ANN_MODEL.pt")
```

6.3.4 Complete Auto-tuning Design Applied to Test Cases

In this section, the complete Auto-tuning process is outlined, and the procedure to collect the statistical evaluation data relating to performance. The ANNs which are described earlier, can be used for auto-tuning the configuration parameters.

It should be noted that there are a number of tunable parameters that relate to IO operations (READ or WRITE), and many remaining which are non-tunable parameters, as mentioned in Table 6.1 3rd column.

The tunable parameters are marked `"Yes"`, and non-tunable parameters are marked `"No"` with respect to READ/WRITE operation. The common non-tunable parameters in both IO operations are `Number of MPI nodes`, `MPI processes on each node` and `File size`. The reason is MPI nodes and processes cannot be altered once the application is initialized and the file size also cannot be changed being the user requirement. In case of READ operation the tunable parameters are `Chunk size` and `File access pattern` values. The additional two non-tunable parameters are `Lustre stripe count` and `Lustre stripe size`. As the file will be already distributed over a specific number of Lustre disks to be READ therefore, re-striping on runtime will not affect the file distribution pattern on disks. In a situation where the file is first written to disk, which is then subsequently read, this is not optimal behaviour as it will add unnecessary IO overhead. Whereas, for WRITE operations the tunable parameters are `Lustre stripe count`, `Lustre stripe size`, `Chunk size` and `File access pattern`. All these configurations can affect WRITE bandwidth at the runtime.

Considering the configuration parameters tunability, the summarized steps to complete the execution flow are: 1) retrieve the existing applied set of parameters settings, 2) predict the IO bandwidths on current and all other possible settings of tunable parameters specified in Listing 6.3 with the given non-tunable parameter settings, 3) Compare all the predicted bandwidth values with each other. 4) Select the maximum predicted IO bandwidth settings and 5) apply the selected settings to configure the tunable parameters. These steps are further elaborated in Listing 6.3 and 6.4.

6.3.4.1 Parameters Selection

In this section, a procedure is presented on how to select new parameters settings that can predict maximum bandwidth.

The function `Select_Parameters()` in Listing 6.3 returns the new tunable configuration settings. It takes the arguments for the given current settings, IO operation (READ or WRITE) and the file path to saved ANN model ("*.pt"). The Line 1 imports the `torch` package to load the saved and trained ANN model in PyTorch on Line 5. The Line 9 to 11 loads the current settings in variable `X`, then sets the `max_bandwidth` to predicted value using `model()` on current settings `X`, and `max_settings` hold the current settings. Lines 14 to 17 represent the possible value settings for tunable parameters for both IO operations, also specified in Table 6.1.

The mechanism to select new configuration settings is to check every possible combination of tunable parameters values in nested loops on Line 20 to 23. The `stripe_counts` and `stripe_sizes` are checked when the current IO operation is WRITE, from Line 26 to 28. By predicting on Line 29 and comparing the new bandwidth value with current `max_bandwidth` value gives the maximum possible bandwidth and its new configuration settings in `max_settings` as on Line 30 to 32. If the current IO operation is READ then the last two nested loops for checking `stripe_count` and `stripe_size` values are not

Listing 6.3: Parameters values selection based on maximum IO bandwidth prediction.

```
1  import torch
2  def Select_Parameters(current_settings,io,model_path):
3
4      # Load ANN Model according to IO and hidden layers.
5      model = torch.load(model_path)
6
7      # Current_settings includes tunable and non-tunable
8      # parameters with values currently configured.
9      X = current_settings
10     max_bandwidth = model(X)
11     max_settings = current_settings
12
13     # Possible values range specified in Table 1.
14     chunk_sizes = [1.00,2.00,4.00,8.00] # in GiBs
15     access_types = ['non-collective','collective']
16     stripe_counts = [2,4,8,16]
17     stripe_sizes = [1,256,512,1024,2048] # in MiBs
18
19     # Loops to check all possible tunable settings.
20     for chunk_size in chunk_sizes:
21         for access_type in access_types:
22             for stripe_count in stripe_counts:
23                 for stripe_size in stripe_sizes:
24                     X['chunk_size'] = chunk_size
25                     X['access_type'] = access_type
26                     if io == "WRITE":
27                         X['stripe_count'] = stripe_count
28                         X['stripe_size'] = stripe_size
29                     new_bandwidth = model(X)
30                     if new_bandwidth > max_bandwidth:
31                         max_bandwidth = new_bandwidth
32                         max_settings = X
33                     if io == "READ": break
34                 if io == "READ": break
35     return max_settings
```

required. In that case, the `break` statements are applied on Line 33 and 34, respectively, to terminate the loops for READ operation.

Finally, the new configurations predicting maximum possible IO bandwidth will be returned on Line 35. These new settings return to code shown in Listing 6.4. According to steps in Listing 6.3 it is possible that parameters may not be tuned to new values. This can be due to the existing configurations predicting the maximum possible IO bandwidth, by checking other value settings in range.

6.3.4.2 Auto-tuning IO operations

In this section the auto-tuning procedure on the MPI side is presented, based on the new parameter values selection. After the selection of these new parameter value settings, they are used to tune the configurations before executing parallel IO.

The Listing 6.4 elaborates the overview of how the auto-tuning task is carried out at MPI application level in the function defined as `Auto_tune_and_run_IO()`. It takes the current settings, intended IO operation, target file path or name:`f_name` and ANN

Listing 6.4: Auto-tuning at Application Level.

```
1  #include <string>
2  using namespace std;
3  void Auto_tune_and_run_IO(char *current_settings[], string io,
4                            string f_name, string model_path){
5    // Initializing MPI space ...
6    MPI_Init();
7
8    // Get current MPI rank.
9    int rank; MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10
11   // Selecting new parameters values at rank 0 ...
12   if(rank == 0)
13     max_settings = Select_Parameters(current_settings,io,
14                                      model_path);
15   // All processes wait and tune parameters ...
16   MPI_Barrier(MPI_COMM_WORLD);
17   MPI_Bcast(&max_settings,1,new_max_type,0,MPI_COMM_WORLD);
18
19   access_pattern = max_settings.access_pattern;
20   chunk_size = max_settings.chunk_size;
21   // New striping only run by rank 0 for WRITE file intention.
22   if(io == WRITE && rank == 0){
23     Remove_Previous_File(f_name);
24     Apply_New_Lustre_Striping(f_name,max_settings.stripe_count
25                               ,max_settings.stripe_size);
26   }
27   MPI_Barrier(MPI_COMM_WORLD);
28   // Rest of the program continues here by
29   // executing MPI-IO APIs on file using tuned settings ...
30 }
```

model file path as arguments on Line 3. Line 5 initializes the MPI space and environment using `MPI_Init()`. Subsequently, Line 9 returns the current process ID or `rank`. The current settings, IO operation and model file path are passed to the method `Select_Parameters()` on Line 13, and defined in Listing 6.3. If the current process ID or rank is 0 only then `Select_Parameters()` function executes, as checked on Line 12. This returns the new parameters values in `max_settings` predicting maximum possible bandwidth with the given non-tunable parameters.

Since the `max_settings` are not returned, all other processes are waiting on Line 16. This is enforced by `MPI_Barrier()` method which is explained in [13]. Once the new values are returned to the rank 0 process, it will send the `max_settings` object message to other MPI processes or ranks. Those ranks receive the message in their respective local `max_settings` object. This is done on Line 17 using the MPI broadcasting method `MPI_Bcast()`, also explained in [13].

At this point, all processes will have received their `max_settings`. Therefore, each process updates its configurations with a new set of value settings (Lines 19 to 20). The `access_pattern` and `chunk_size` are required by all processes to read or write the file with same access pattern and number of bytes. While the `stripe_count` and `stripe_size` are updated for all processes, they are only required at the rank 0 process when IO operation is WRITE. This is essential, for avoiding a race condition on the same file when applying a new file striping order (Line 22 to 24). In the case of the Rank 0 process, this removes the previous stripe settings by removing the existing empty file on Line 23 via `Remove_Previous_File()` method. Then the new stripe settings are applied on the file (Line 24) using the `Apply_New_Lustre_Striping()` method.

It should be noted here, that all processes except rank 0, wait again on Line 27 until the rank 0 process is applying the new Lustre stripe settings on the file. This creates a new empty file on the same path with new stripe settings. When the WRITE operation executes afterwards by all processes the file will be written with new settings applied to it. On completion, the bandwidth will be recorded using Darshan, as mentioned earlier. It is also shown in Listing 6.5 for auto-tuning the default configuration test cases, using this method defined in Listing 6.4.

6.3.4.3 Statistical Data Collection upon Auto-tuning Test Cases

In this section the experimental setup is presented for auto-tuning the test cases and collecting the key statistical values for the system bandwidth performance. The working functionalities in Listings 6.3 and 6.4 perform auto-tuning on default configuration test cases, as mentioned in Table 6.5. These default test cases with parameters values settings make a total of 1458 configuration settings to be tested. The tunability of parameters is exactly the same as in Table 6.1.

The purpose of auto-tuning a number of default test cases is to analyze the improvement in IO bandwidth in terms of percentage, through the use of ANN models. The bandwidth values have been collected using the Darshan characterization tool, upon auto-tuning the test cases. Listing 6.5 presents the methodology to collect all this necessary statistical data for performance evaluation presented later in this research.

It should be noted that while the ANN models are trained on smaller file size and chunk size values, in the evaluation approach, the larger file and chunk size values have been used for test cases. These are shown in the default configuration test cases to auto-tune, in Table 6.5.

Listing 6.5: Statistical Data Collection upon Auto-tuning.

```
1  import numpy
2  import statistics
3  def Stats_Data_Collection(Default_Configurations,io,model_path):
4
5      X = Generate_List(Default_Configurations)
6      n = len(X)
7      Repetitions = 3
8      #Loop to Auto-tune test cases to gather performance data
9      for i in range(n):
10         default_bandwidth = list()
11         tuned_bandwidth = list()
12         for j in range(Repetitions):
13             # Executing Default Configurations.
14             f_name, d_name = Lustre_and_Darshan_Settings(io,
15                             X[i]['stripe_count'],X[i]['stripe_size']))
16             if (io == READ): Generate_File_to_be_read(f_name)
17             Execute_Default_Configuration(X[i],io,f_name)
18             default_bandwidth.append(ParseDarshanFile(d_name))
19             Remove_Target_Files(f_name,d_name)
20
21             # Executing Auto-tuned Configurations.
22             f_name, d_name = Lustre_and_Darshan_Settings(io,
23                             X[i]['stripe_count'],X[i]['stripe_size']))
24             if (io == READ): Generate_File_to_be_read(f_name)
25             Auto_tune_and_run_IO(X[i],io,f_name,model_path)
26             tuned_bandwidth.append(ParseDarshanFile(d_name))
27             Remove_Target_Files(f_name,d_name)
28
29         old_bandwidths.append(statistics.mean(default_bandwidth)
30         new_bandwidths.append(statistics.mean(tuned_bandwidth)
31         if (new_bandwidth[i] > old_bandwidth): c_im += 1
32         else: c_di += 1
33
34     average_old_bandwidth = statistics.mean(old_bandwidths)
35     average_new_bandwidth = statistics.mean(new_bandwidths)
36
37     overall_percentage_improvement = (average_new_bandwidth -
38         average_old_bandwidth) / average_old_bandwidth * 100.0
39
40     per_imp_cases = c_im / n * 100.0
41     per_dis_cases = c_di / n * 100.0
42     max_tuned_bandwidth = max(new_bandwidths)
43     max_default_bandwidth = max(old_bandwidths)
44     min_tuned_bandwidth = min(new_bandwidths)
45     min_default_bandwidth = min(old_bandwidths)
46     median_tuned_bandwidth = statistics.median(new_bandwidths)
47     median_default_bandwidth = statistics.median(old_bandwidths)
48     stdev_tuned_bandwidth = numpy.stdev(new_bandwidths)
49     stdev_default_bandwidth = numpy.stdev(old_bandwidths)
50     var_tuned_bandwidth = stdev_imp**2
51     var_default_bandwidth = stdev_dis**2
```

Table 6.5: Default Configurations Test Cases for Auto-tuning.

Configuration Parameters	Default Values Settings
Number of MPI nodes	16,8,4
MPI processes on each node	16,8,4
Lustre stripe count	16,8,4
Lustre stripe size (MiBs)	512,1024,2048
File size (GiBs)	50,75,100
Chunk size (GiBs) per process	2,4,8
IO operation	read,write
File access pattern	collective,non-collective

Multiplying the number of value settings results in 1458 possible combinations of the default configuration settings in one complete list. The description of steps to collect statistics of IO performance improvement data, starts with the function definition of `Stats_Data_Collection()`. This method provides 7 arguments: 1) `Default_Configurations`, the lists of all default parameters values, 2) `io`, READ/WRITE operation and 3) `model_path`, the path of ANN model (`".pt"`) file to use. This is followed by first generating a complete list of possible default configuration settings in `X` on Line 5, such that each element of X is a set of different parameters with values as stated earlier by example in section 6.3.1. Then Line 6 computes the length or size of that list in memory `n`. On Line 7 the `Repetitions` count is set to the value 3, which controls the iterations of the nested loop on Line 12. This determines the repeated execution of the default configurations and the auto-tuned configurations respectively.

Line 9 initiates the main loop to process default and auto-tuned configuration settings against each set of parameters in X. Lines 10 and 11 create the `default_bandwidth` and `tuned_bandwidth` lists. The purpose of these lists is to save the default settings bandwidths and the auto-tuned settings bandwidth after their execution. The bandwidth values of the default and auto-tuned configuration settings are saved in their respective lists three times from Line 14 to 27. The reason for three repetitions was to mitigate against individual variations across bandwidth values due to any reason, and therefore provide an average. After this inner loop, both bandwidths are averaged and appended in `old_bandwidths` and `new_bandwidths` lists, respectively on Line 29 and 30. The bandwidth improvement or dis-improvement count is also maintained in `c_im` and `c_di` on Line 31 and 32, respectively.

Once the main loop is finished, all the default and auto-tuned bandwidths are averaged on Line 34 and 35. The overall bandwidth improvement in percentages is calculated (Lines 37 to 38). Then all remaining statistics are calculated (Lines 40 to 51). These include percentages of improved and dis-improved test cases, maximum, minimum, median, standard deviation and variance in both default and auto-tuned bandwidth values for comparison. A comprehensive study of performance is explained in the next section using the data gathered in these steps.

6.4 Experimental Results and Evaluation

This section outlines the runtime complexity analysis of using ANNs predictions for parameter selection, the improvement results in READ/WRITE bandwidth, achieved by

auto-tuning the default configuration test cases in Table 6.5, and the common configuration settings applied by the system model during the auto-tuning process.

The Auto-tuning of the test cases is conducted on 16 compute nodes of the Intel Xeon Gold 6148 Skylake processors cluster (KAY) and its 16 LFS employed disks as IO object storage targets [51]. The ANN models have been supported by training and testing through Tensors construct of PyTorch, on NVIDIA Tesla V100 GPU cards [184, 22].

6.4.1 Runtime Complexity Analysis of ANNs Predictions

Since this procedure of parameters value selection involves a brute force to check all combinations, its runtime cost should be analysed. Line 29 of Listing 6.3, runs the ANN feed forward propagation pass to predict the value using `model(X)`. In this case, there are three steps involving: 1) input layer to hidden layer 1, 2) hidden layer 1 to hidden layer 2 and finally 3) hidden layer 2 to output layer. The Table 6.7 presents all the steps involved, alongside the number of computations against multiplications (Mul.) and additions (Add.).

In the first step, there are roughly 3584 computations of multiplications and additions. Similarly, in second step there are approximately 65536 computations. This is the most expensive phase in this feed forward propagation process in both computation and memory consumption terms. This is due to the reason that the hidden layers and their weights matrices have the greatest number of nodes and memory allocation, respectively.

Finally, in the third and last step, it performs 256 computations. By summing these computations, the 69376 runs in total are determined, for a single feed forward propagation pass to predict a value. In case of a READ operation the forward propagation pass runs 8 (4×2) times making a total execution of 555,008 instructions to select parameters predicting maximum bandwidth. Whereas, in case of a WRITE operation the forward propagation pass runs 160 ($4\times2\times4\times5$) times which makes a total of 111,001,60 computations to select parameters. This is according to the code logic of Listing 6.3.

In order to analyze how fast these millions of computations can be processed, this relies on memory resources usage during the feed forward propagation. The Table 6.6 shows the matrices and the memory space used during a forward propagation pass. These matrices were part of the ANN model which is created with a default data type `float32`. This represents a 32-bit or 4-bytes floating point number. There are 7 matrices used in a feed forward propagation process as can be noticed.

The total memory required by the ANN model is 140320 bytes or almost 137 KiB. If the model is created using a 64-bit or 8 bytes double precision floating point number then this value will be doubled to almost 274 KiB. However, with either of the decimal datatypes or sizes used, these matrices are easily cacheable in the CPU RAM cache memory. Therefore, running the millions of computations as stated earlier, approximately takes a negligible execution time in the unit seconds of $10e^{-4}$ including the loading time of the required libraries. However, for a first time execution, the program loading can take around 30 seconds. Afterwards, it reduces to less than a second, as tested on a KAY's compute node [51].

For the sake of simplicity and ease, the coded parameters selection logic is a separate python script to execute using PyTorch module, which interfaces with the caller MPI based C++ program and applies the new configuration parameters settings and run the READ or WRITE operation, as stated in Listing 6.4. Otherwise, the trained ANN model could be coded in the same MPI based program as well by using its C++ version

Table 6.6: Matrices Description and Memory Usage During ANNs Feed Forward Propagation Pass.

Matrix	Description	Memory Usage (bytes)
$I_{7\times1}$	7 Input features Layer against configurations	$7 \times 1 \times 4 = 28$
$H_{256\times1}$	256 hidden layer 1 nodes values	$256 \times 1 \times 4 = 1024$
$W_{256\times7}$	Contain weights for $I_{7\times1}$ to $H_{256\times1}$ nodes	$256 \times 7 \times 4 = 7168$
$H_{128\times1}$	128 hidden layer 2 nodes values	$128 \times 1 \times 4 = 512$
$W_{128\times256}$	Contain weights for $H_{256\times1}$ to $H_{128\times1}$ nodes	$128 \times 256 \times 4 = 131072$
$O_{1\times1}$	1 output value against predicted IO bandwidth	$1 \times 1 \times 4 = 4$
$W_{1\times128}$	Contain weights for $H_{128\times1}$ to $O_{1\times1}$ nodes	$1 \times 128 \times 4 = 512$
	Total Memory used is:	140320 bytes ≈ **137 KiB**

Table 6.7: ANNs Feed Forward Propagation Pass to predict bandwidth.

Step#	Equation	No. of Computations per step pass
1	$H_{256\times1} = W_{256\times7} \times I_{7\times1}$	$(7Mul. + 7Add.) \times 256 = 3584$
2	$H_{128\times1} = W_{128\times256} \times H_{256\times1}$	$(256Mul. + 256Add.) \times 128 = 65536$
3	$O_{1\times1} = W_{1\times128} \times H_{128\times1}$	$(128Mul. + 128Add.) \times 1 = 256$
	Total computations are ≈	**69376** per ANNs Forward Propagation Pass

of PyTorch library which is more complex than the python scripting however, it could be even faster by avoiding the extra program loading during the already executing program.

6.4.2 READ Auto-tuning and Common Configurations Analysis

In this section, a detailed discussion is provided for the Auto-tuning performance with respect to READ operations.

Table 6.8 shows the READ improvements after execution the procedure represented in Listing 6.5. The first observation, is that the number of test cases improved during the testing process. These are 1206 indicating a 82.7% of the total cases (1458). On further examination, it has been observed that the bandwidth values also show significant improvements. The mean bandwidth value of the tuned settings is 62630.5 MiB/s, which is significantly 1.65× times greater than the mean bandwidth value 37798.1 MiB/s of the default settings. This is a clear indication of how significant the tuning approach can be with respect to the critical metric of bandwidth. The overall READ bandwidth improvement is shown to be approximately 65.7% which is a significant optimized performance gain. The remaining statistics are presented with maximum, minimum, median and standard deviation of bandwidth values for default configurations against the tuned configurations.

Furthermore, the Figure 6.4 shows the graph of default versus tuned bandwidths throughout all 1458 test configuration settings. It can be seen that the auto-tuned bandwidths demonstrate far better values than the default bandwidths in many cases.

The ANN predictions have leveraged the auto-tuning process and demonstrate these significant READ bandwidth performance optimization. It is also worth noting the common auto-tuned parameters values selection throughout the test cases. The Table 6.9 shows the performance of three common tuned parameters values versus the default con-

Table 6.8: READ Auto-tuning Improvement Results.

Metrics	Numerical Figures
Improvements	1206/1458
Improvements(%)	82.7%
Mean Bandwidth of Default Configurations (MiB/s)	37798.1
Mean Bandwidth of Tuned Configurations (MiB/s)	62630.5
Max. Default Bandwidth (MiB/s)	712334.2
Max. Tuned Bandwidth (MiB/s)	697943.6
Min. Default Bandwidth (MiB/s)	121.8
Min. Tuned Bandwidth (MiB/s)	307.9
Median Default Bandwidth (MiB/s)	14020.5
Median Tuned Bandwidth (MiB/s)	17731.1
Default Standard Deviation (MiB/s)	85578.9
Tuned Standard Deviation (MiB/s)	114305.7
Default Variance	7323745984.2
Tuned Variance	13065796947.6
Overall Bandwidth Improvement(%)	65.7%

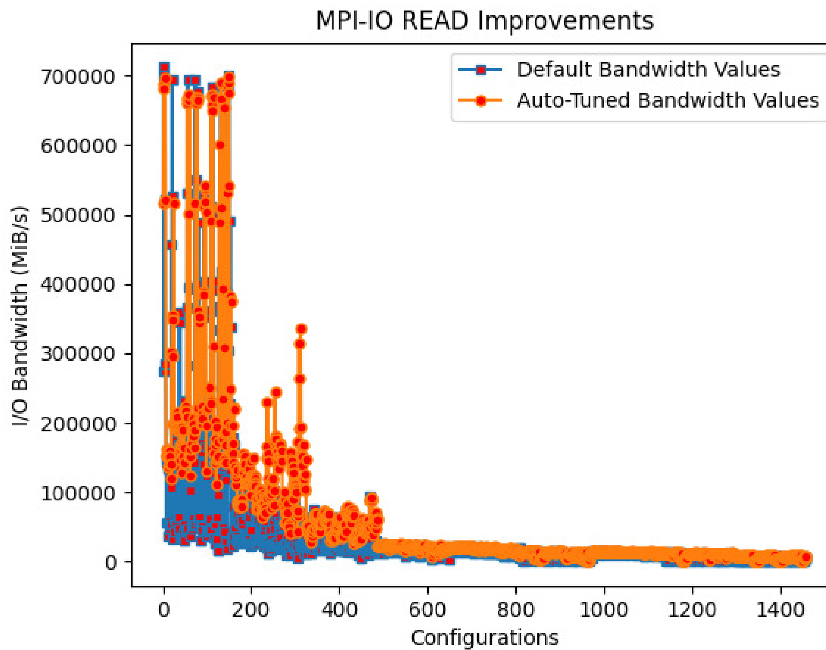


Figure 6.4: READ Improvements.

figuration settings test cases. This performance is stated with respect to the relative improvements and dis-improvements. When examining the tuned configurations, it is noted that the first uses a chunk size of 1 GiB and a `non-collective` file access pattern.

This setting has been selected in the majority of test cases, 1332 times. This resulted in 1156 occurrences of improvement while there were 176 occurrences of dis-improvement. Therefore probabilistically, this configuration setting has a 86.8% chance to improve bandwidth while indicating a 13.2% chance of dis-improving.

The second frequently tuned setting involved a chunk size of 2 GiB with the non-

Table 6.9: Common READ Auto-Tuned Configurations. Note: - **c_s** denotes chunk size for each MPI process and **f_a_p** denotes file access pattern.

Tuned Configurations	Improvements	Dis-improvements	Total selections
[c_s:1GiB, f_a_p:non-collective]	1156 (86.8%)	176 (13.2%)	1332
[c_s:2GiB, f_a_p:non-collective]	48 (42.1%)	66 (57.9%)	114
[c_s:1GiB, f_a_p:collective]	2 (16.7%)	10 (83.3%)	12
Total test cases =	1206 (82.7%)	252 (17.3%)	1458

collective file access pattern. This setting was selected 114 times in which it yielded 48 improvements and 66 dis-improvements. In this case, the model prediction resulted in a 42% chance of improvement and a 58% chance of dis-improvement.

The third and last setting selected for auto-tuning was 1 GiB of chunk size with the collective file access pattern. This was selected the least number of times, showing 2 improvements and 10 dis-improvements. This has indicates a chance of 16.7% for bandwidth improvement. Therefore, it is the most unfavourable setting to apply in auto-tuning.

We note that, the non-collective file access pattern was a common factor in the improvements recorded. The non-collective access setting has been used mostly and proven successful with two different chunk size values throughout the test cases. This yielded close to 83% chance of improving bandwidth according to the Table 6.9. Therefore, according to the findings, non-collective access is the most favourable pattern for READ operations with a view to maximising bandwidth.

6.4.3 WRITE Auto-tuning and Common Configurations Analysis

This section presents a series of results related to MPI WRITE operation optimization. It is evident from Table 6.10 that majority of test cases improved when auto-tuning is applied. The number of overall improvements observed were 1353 which is 92.8% of the total number (1458) of the default configurations test cases. The remaining data shows the mean bandwidth throughput values of the default and tuned configurations. It can be seen that the mean bandwidth of the tuned configuration cases is 9798.1 MiB/s which is 1.83× times greater than the mean bandwidth of the default configurations (5346.9 MiB/s). These numerical figures indicate an overall bandwidth performance of 83.2%. This shows the WRITE bandwidth performance throughput has been significantly optimized with the devised ANN prediction based auto-tuning strategy.

We note that the maximum tuned bandwidth is greater than the maximum default bandwidth. This significant increase can be seen in the minimum and median bandwidth values from the default to tuned settings. The minimum value increased by almost 49× and the median value by 2.2× due to auto-tuning. The standard deviations remain comparable between both default and auto-tuned results.

To support the observations, the data in Figure 6.5 has been observed, which shows the default versus tuned bandwidths. It is noted that the auto-tuned bandwidths show consistently higher values than the default bandwidths in the majority of the cases. Therefore, it is safe to conclude the significant performance gain in WRITE operation bandwidth optimized by auto-tuning based on the ANNs predictions.

The configuration settings which are most commonly selected by the auto-tuning

Table 6.10: WRITE Auto-tuning Improvement Results.

Metrics	Numerical Figures
Improvements	1353/1458
Improvements(%)	92.8%
Mean Bandwidth of Default Configurations (MiB/s)	5346.9
Mean Bandwidth of Tuned Configurations (MiB/s)	9798.1
Max. Default Bandwidth (MiB/s)	18944.6
Max. Tuned Bandwidth (MiB/s)	19563.5
Min. Default Bandwidth (MiB/s)	70.9
Min. Tuned Bandwidth (MiB/s)	3488.0
Median Default Bandwidth (MiB/s)	4718.1
Median Tuned Bandwidth (MiB/s)	10446.1
Default Standard Deviation (MiB/s)	3192.4
Tuned Standard Deviation (MiB/s)	3350.4
Default Variance	10191711.7
Tuned Variance	11225469.3
Overall Bandwidth Improvement(%)	83.2%

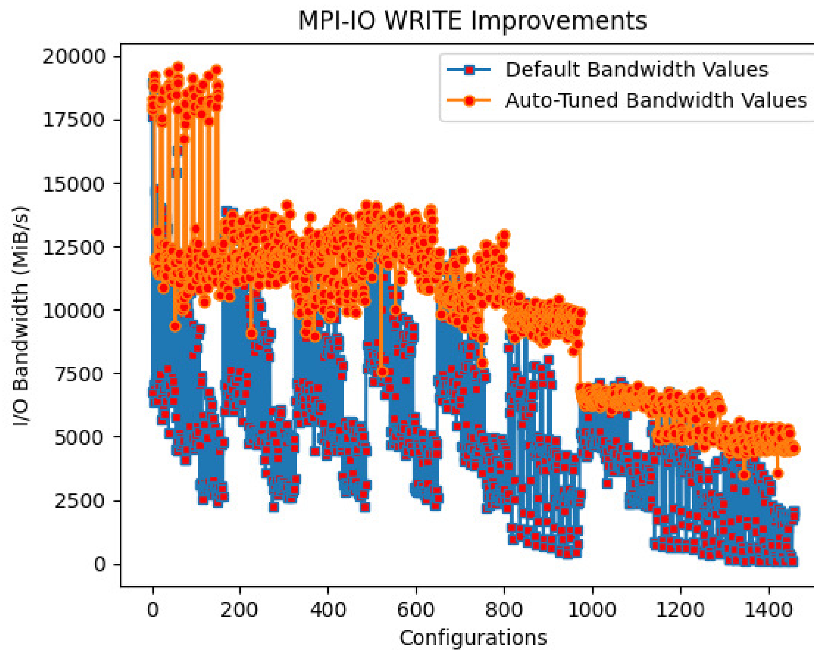


Figure 6.5: WRITE Improvements.

process are shown for WRITE operations in Table 6.11. This Table shows the 5 common settings applied during the auto-tuning of default configuration test cases. For auto-tuning WRITE operations there are 4 tunable paramaters: stripe count, stripe size, chunk size and file access pattern. The most common configuration setting was selected 864 times which shown first in the table. This contains stripe count of value 16, stripe size of 2048 MiB and the non-collective or contiguous file access pattern. This setting resulted in 794 improvements and 70 dis-improvements which are both the largest when compared to rest of the settings. Probabilistically, the chance of improving bandwidth

Table 6.11: Common WRITE Auto-Tuned Configurations. Note: - **s_c** denotes lustre stripe count, **s_s** denotes lustre stripe size, **c_s** denotes chunk size for each MPI process and **f_a_p** denotes file access pattern.

Tuned Configurations	Improvements	Dis-improvements	Total selections
[s_c:16, s_s:2048MiB, c_s:1GiB, f_a_p:non-collective]	794 (91.9%)	70 (8.1%)	864
[s_c:16, s_s:512MiB, c_s:8GiB, f_a_p:non-collective]	103 (95.4%)	5 (4.6%)	108
[s_c:16, s_s:2048MiB, c_s:2GiB, f_a_p:non-collective]	145 (89.5%)	17 (10.5%)	162
[s_c:16, s_s:1MiB, c_s:1GiB, f_a_p:non-collective]	214 (99%)	2 (1%)	216
[s_c:16, s_s:1024MiB, c_s:1GiB, f_a_p:non-collective]	97 (89.8%)	11 (10.2%)	108
Total test cases =	1353 (92.8%)	105 (7.2%)	1458

through using this setting is 91.9%, with a 8.1% chance of dis-improvement.

The second, third and fifth configurations in Table 6.11 show improvements of 103, 145 and 97 respectively. Due to these settings being less frequently selected by the system arising from auto-tuning, they show comparatively small observations of dis-improvements of 5, 17 and 11. However, in terms of probability percentage, the second configuration has a 95.4% chance of improving bandwidth with a 4.6% chance of dis-improving.

This shows better more promising results than the first configuration with respect to probability. Whereas, the third and fifth settings have ≈90% chance to improve bandwidth, with a ≈10% chance to dis-improve it. These marginally less optimal than than the first configuration.

The fourth configuration has 214 improvements and only 2 dis-improvements. Therefore, probabilistically, it has a 99% chance of improving bandwidth with only a 1% chance of dis-improving it. Interestingly, this configuration has a common feature with results that were recently published in this area. The selected stripe size of 1 MiB, was identified and suggested as a means of increasing the WRITE bandwidth from the parallel-plots, as explained in the previous chapter of the thesis.

Another factor which should be noted is that the stripe count value of 16 and non-collective file access pattern are common across all these selected configurations. The stripe count value of 16 was the maximum in the range specified in Listing 6.3, for the selection. Therefore, it can be inferred to keep the maximum stripe count available. This means bandwidth is expected to increase by increasing number of stripe count value. Since the file access pattern is set to non-collective in all these settings, it is best for maximising overall bandwidth.

From studying these settings, they or similar ones can be applied to maximise bandwidth throughput in the event of the prediction model not being available for parameter selection.

6.5 Summary of the Work

The results presented in this research indicate strong performance gains arising from using ANN models to auto-tune HPC MPI-IO configurations. These improvements have been demonstrated with respect to bandwidth performance primarily.

The tuning process is focused on a number of key parameters that directly effect IO bandwidth. Since the optimization has to be performed at runtime, the tunable param-

eters can be modified i.e., the chunk size per MPI process, Lustre striping parameters, etc. Therefore, the certain tunable parameters and settings are identified for READ and WRITE operations separately.

The optimization of HPC IO using ML predictions has been a common approach in recent studies [14, 15, 16, 17, 193, 18, 24, 21, 82, 83]. The parameters before IO execution can be modified or tuned to improve IO bandwidth. It is noted that the predictive IO modelling with multiple configurations is a primary requirement of the existing problem. The most convincing technique for predictive IO modelling is ANN due to their high forecasting accuracy [23, 19, 21].

In previous chapter, the key parameters were identified and executed as benchmarks. However, in the research outlined in this chapter, the research has been further extended to identify the tunable and non-tunable parameters for MPI-IO operations. The previous benchmarks were re-executed on the generated list of all possible configurations as mentioned in Table 6.1. This was done to regenerate the ANNs through a ML process, as mentioned in Table 6.2. This process was conducted with respect to their specific hyper-parameters in Table 6.4, for both READ and WRITE operations. The mappings of the input layer nodes to configuration parameters are also described in Table 6.3. The models predict bandwidth precisely and match with the pattern of the changing configurations. The benchmarks and prediction results are presented in Figure **??** and 6.3.

Subsequently, the auto-tuning approach was applied on the default configuration test settings, as mentioned in Table 6.5, with respect to READ and WRITE operations.

We observed the auto-tuning of configuration parameters based on ANN predictions, and these have shown significant optimized IO bandwidth performance results. The READ operation has shown 65.7% improvement in bandwidth while having 83% of the test cases improved. Whereas, the WRITE operation has shown 83% improvement in the overall IO bandwidth while having almost 93% of the test cases improved.

Subsequently, the common configuration settings are discussed as a result of auto-tuning for both READ and WRITE operations. It was observed that the non-collective or contiguous file access pattern was selected in the majority of the READ test cases. Furthermore, the stripe count was set to a maximum available value 16 in all the WRITE test cases, in order to have maximum possible bandwidth value.

These configurations can be considered with variations of other parameters values for tuning MPI IO operations, in the event that a ANN model is not available, to optimise significant IO bandwidth performance.

6.6 Conclusion

This thesis chapter has analyzed MPI-IO bandwidth performance improvements by auto-tuning configuration parameters via ANNs predictions. It has explained the requirement of an efficient and adaptable optimization technique by extending the work from previous chapter. This aligns with already established research which examines the challenges of HPC IO performance through ML prediction [14, 15, 16, 17, 193, 18, 24, 21, 82, 83].

There are three main contributions outlined in this research chapter. First is the auto-tuning strategy based on the ANNs MPI-IO bandwidth prediction on various configuration parameters. Second is the statistical analysis of the overall auto-tuned cases with respect to bandwidth performance. Third is the identified common tunable parameters settings, selected and used by ANN based system model.

From the results presented in the research, it has shown an overall READ bandwidth improvement of 65.7% with almost 83% test cases improved. Whereas, overall WRITE bandwidth improvement yielded upto 83% with almost 93% test cases improved.

Furthermore, from the configurations identified in this research, it is shown that there are clear benefits to firstly using the non-collective file access pattern in both READ-/WRITE cases. Secondly, the benefits are also noted for using the maximum lustre stripe count value in WRITE operations. Throughout the results, these observations were noted as being prominent in the most optimal settings.

In the event that a ML optimisation approach is not available, the identified configuration policies could be easily applied to benefit the bandwidth utilisation challenges noted in previous research. Furthermore, the approach and results outlined in this research show the considerable advantages to professionals in this area. It has demonstrated substantial improvements on HPC MPI-IO bandwidth performance, thereby enhancing the overall efficiency of large sized data processing task completion.

Chapter 7

Seismic Data IO and Sorting Optimization in HPC through ANNs Prediction based Auto-tuning for ExSeisDat

7.1 Introduction

The Oil and Gas industry extensively relies on the Seismic data as a critical factor for processing it to understand and visualize the structure beneath the surface of earth and the seabed [3]. The seismic data is normally stored and encapsulated in SEG-Y format files which is a global standard [4]. The Figure 3.2 shows a complex structure layout of SEG-Y format to store the seismic data in file. It contains trace headers of 240 bytes each and actual traces data on alternate positions in the file. Usually the SEG-Y or seismic data scales to petabytes when written in file on disks. Therefore, it compellingly increase the need of high performance computing (HPC) or super-computing systems to perform large scale IO processing, across the Oil and Gas production industry as well as the research industry.

The modern HPC clusters are normally well equipped to perform exascale operations and significantly time efficient. They usually contain high number of computing nodes and parallel storage disks connected via fast network hardware infrastructure [5, 6]. The Figure 2.1 shows a very basic structure of HPC system. At software side a programming paradigm is also required to exploit the clusters potential by writing and executing parallel applications, which, in this case is the standard Message Passing Interface (MPI) library to serve the purpose [7]. The library also contains the application programming interface (API) for carrying out parallel IO processing tasks by communicating with multiple storage disks. These disks are normally controlled and managed by a parallel file system (PFS) protocols. In this research, the Lustre file system (LFS) is the PFS running on the targeted machine disks [8].

To process the SEG-Y files data the Extreme-Scale Seismic Data (ExSeisDat) library is already developed based on the existing MPI-IO APIs [1, 13]. The ExSeisDat contains its own parallel IO library, namely PIOL, and Workflow library to perform seismic data related functionalities. In spite of this parallel processing library, it faces degradation in program execution performance. The reason is underlying running multiple MPI processes get restricted by LFS protocols, to access a disk at a same instance in parallel. This

normally occurs as data-aligning is not applied and considered by users [9, 10, 11, 12]. Although, it does not guarantee maximum possible bandwidth performance, as explained in chapter 5. It is also tricky to apply data-aligning to SEG-Y format file as traces data is placed on alternative positions rather than being placed consecutively, and consequently not considered in ExSeisDat functionality. This means the projection of parallel IO bandwidth performance of ExSeisDat to process seismic data critically relies on certain configuration parameters related to MPI, LFS and pattern to access data from SEG-Y file. For example, number of running MPI processes, lustre stripe count of parallel disks, random or contiguous access pattern, etc. Therefore, it leaves the user with the choice to find and tune the suitable parameters settings that can improve the bandwidth performance from the currently existing value settings.

As the aim is to keep the user of ExSeisDat free from the overhead of manually tuning the settings of related parameters therefore, the auto-tuning approach for SEG-Y operations is proposed based on maximum IO bandwidth prediction value. This research chapter further extends the work elaborated in chapter 5, regarding the bandwidth prediction of SEG-Y IO and file sorting operations. The first key contribution in this research is the auto-tuning design strategy for SEG-Y IO and file sorting operations on the basis of related parameters bandwidth prediction to optimize performance. The recent studies have shown the notable advantages of predictive IO bandwidth or performance modelling through Machine Learning (ML), for auto-tuning parameters in different HPC problem scenarios [14, 15, 16, 17, 18]. Consequently, the Artificial Neural Networks (ANNs) ML technique is particularly chosen for predictive modelling [19, 20], on the collected SEG-Y IO and Sorting benchmarks execution profiling data. The research studies from [21, 22, 23] are the source of motivation to implement and execute ANNs ML process. This has been carried out using Python PyTorch package which has significant precision in predicting outputs. Once the ANNs are trained, they are validated and applied into auto-tuning design over the default configuration test settings.

We have tested the range of ANNs with varying number of nodes in the hidden layers to observe its impact on prediction accuracies of ANN models, which is the second key contribution. The runtime cost analysis of ANNs prediction is also presented, as it is a core component in new parameters values selection.

Additionally, the statistical analysis of resulted bandwidth values from default and auto-tuned test settings executions, is a third key contribution. This has also been conducted on the same range of ANNs with varying hidden layers nodes, as stated earlier. The purpose of this analysis is to see the most suitable ANN for auto-tuning of SEG-Y IO and file sorting operations.

Our research mainly emphasizes on the designing of auto-tuning strategy over SEG-Y IO and file sorting configurations by bandwidth prediction of ANNs. Furthermore, it focuses on the statistical analysis on varying overall bandwidth performance improvements, over the range of ANNs.

As evident from results and analysis, the auto-tuning design based on ANNs contribute to a notable increase in SEG-Y IO and file sorting bandwidth performance. This research chapter is structured as follows; Related Work in section 2, Design and Implementation in 3, Experimental Result Analysis in 4, Discussion in 5 and Conclusion in 6.

7.2 Related Work

The ML based predictive modelling has been an important element in the previous and recent studies, for handling IO performance degradation. The performance prediction is evaluated in the area of HPC clusters before further steps such as auto-tuning the specific configuration parameters. Those researches have been a great source of motivation to adapt prediction based auto-tuning strategy. This is for optimizing the SEG-Y IO and file sorting performance across LFS employed disks in the super-computing cluster.

The work presented chapter 5 of the thesis, predicts the IO bandwidth performance on SEG-Y IO and file sorting parameters using ANNs and shows the prediction accuracy. However, those ANNs had only one type of setting in the hidden layers. The setting was 256 nodes in first hidden layer (h1) and 128 nodes in second hidden layer (h2). The SEG-Y IO READ and WRITE prediction accuracies were 96.5% and 88.1%, respectively. Whereas, the SEG-Y file sorting READ and WRITE predictions yielded to 77% and 80% in accuracy, respectively.

The study demonstrated by Xu et al. in [14], presents the parameters such as the number of IO threads, CPU frequency and the IO scheduler impacts the HPC-IO performance throughput. Using these parameters, the IO pattern behaviour is determined via predictive extrapolation and interpolation modelling approaches. It was achieved by a data analytic framework which supported the exa-scale modelling experiments. The performance evaluation was based on computing prediction accuracies on the unseen configurations of testing system. Subsequently, the map between Bayesian Treed Gaussian Process variability and the varying regression techniques, was used to optimize the system configurations. This leveraged the parameters selection by statistical methods insights and the HPC variability management.

The research by Bez et al. in [15], handles the parallel IO requests by adaptively scheduling them based on the tuning of time window of executing workload within the HPC system. The aim is achieved by constructing the adaptive scheduler through reinforcement learning. As per performance evaluation, the 88% precision is achieved for runtime parameters selection once the access pattern is observed and classified. This classification is determined by deep neural networks in the initial few minutes by the system. Afterwards, the system optimize and improves its IO performance for rest of the lifetime, as endorsed by the literature. The key aspect of this study is being more dynamic as compare to other technique, as it involves no training step.

In the research by Bağbaba in [17], the random forest regression has been used as the ML technique for predictive bandwidth modelling against the collective MPI-WRITE operation. The accuracy of the predictions has been notably high, ranges in approximately 82-99%. This was also dependent on maximum depth setting. As per the literature, the training and testing datasets have been significantly small sized as compared to the datasets used in this thesis. By adding further variation in the data, the prediction accuracy could be low by increasing data volume.

The work done by Behzad et al. in [16], deals with the HDF5 format data files by means of optimizing the parallel IO performance. The testing was carried on various HPC systems running the LFS and General Parallel File System (GPFS) as the parallel file systems (PFSs). In this study, the auto-tuning had a vital role, based on the IO predictions. The predictive IO modelling was done using the data from LFS-IOR and some different benchmarks by nonlinear regression ML algorithm. This caused notable gain in parallel IO performance by auto-tuning the newly selected parameters. In comparison,

the work in [18] by Madireddy et al., demonstrates the IO predictive modelling using LFS-IOR benchmarks as well. However, for the training purpose, the ML approach has been used is Gaussian process regression.

The approach demonstrated by Schmidt et al. in [21], predicts file access time on storage disks managed by LFS. The series of benchmarks were run to profile file access times, so they could be used in training ML models. The ANNs were used as the ML technique. The ANNs generated and produced around 30% less the average prediction error in comparison to the linear ML modelling techniques. The evaluation of distributed file access times was carried out in regard to similar parameters that used to access the file. The study also revealed that normally, the file access times are determined by the degree of magnitude in regard to particular IO paths.

Furthermore, the other researches explored regarding MPI application performance optimization via ML predictive modelling and then auto-tuning parameters. However, they do not consider the part of IO processing within a parallel program [181, 182].

Contrasting with existing recent research, the work of this thesis stresses upon optimizing HPC IO performance for SEG-Y READ/WRITE and file sorting through auto-tuning parameters, which is based on the predictive IO bandwidth modelling over the range of ANNs. This provides different bandwidth performance statistics over variation in hidden layers nodes.

7.3 Design and Implementation

7.3.1 Research Methodology

This work has been carried out with the common sequence of steps for both SEG-Y IO and file sorting operations separately. The steps comprises of the: 1) Identification of the key tunable and non-tunable configuration parameters, 2) Re-execution of SEG-Y IO and file sorting benchmarks to generate profiling data, 3) Training/Testing of 6 ANNs for each SEG-Y READ/WRITE and file sorting READ/WRITE operation (total 24 ANNs), 4) Applying design strategy for auto-tuning on basis of generated ANNs and 5) Prediction accuracy values and statistical analysis of bandwidth performance results against all ANNs. Figure 7.1 explains the main components of the flow of this research.

7.3.2 Key Parameters Identified

The SEG-Y IO and file sorting benchmarks are executed after identification of key configuration parameters, to read/write and sorting traces data in file, respectively. The benchmarks profiling datasets comprises of both training and testing sets for ANN models learning and prediction, respectively. The configuration parameters have also been used as the input features to ANN models.

Table 7.1 and 7.2 show the complete separate list of the related key parameters for both SEG-Y IO and file sorting operations and their corresponding values settings. A complete and separate list for each operation type is generated prior to benchmarking, which contains all possible configuration settings according to the tables. A single configuration

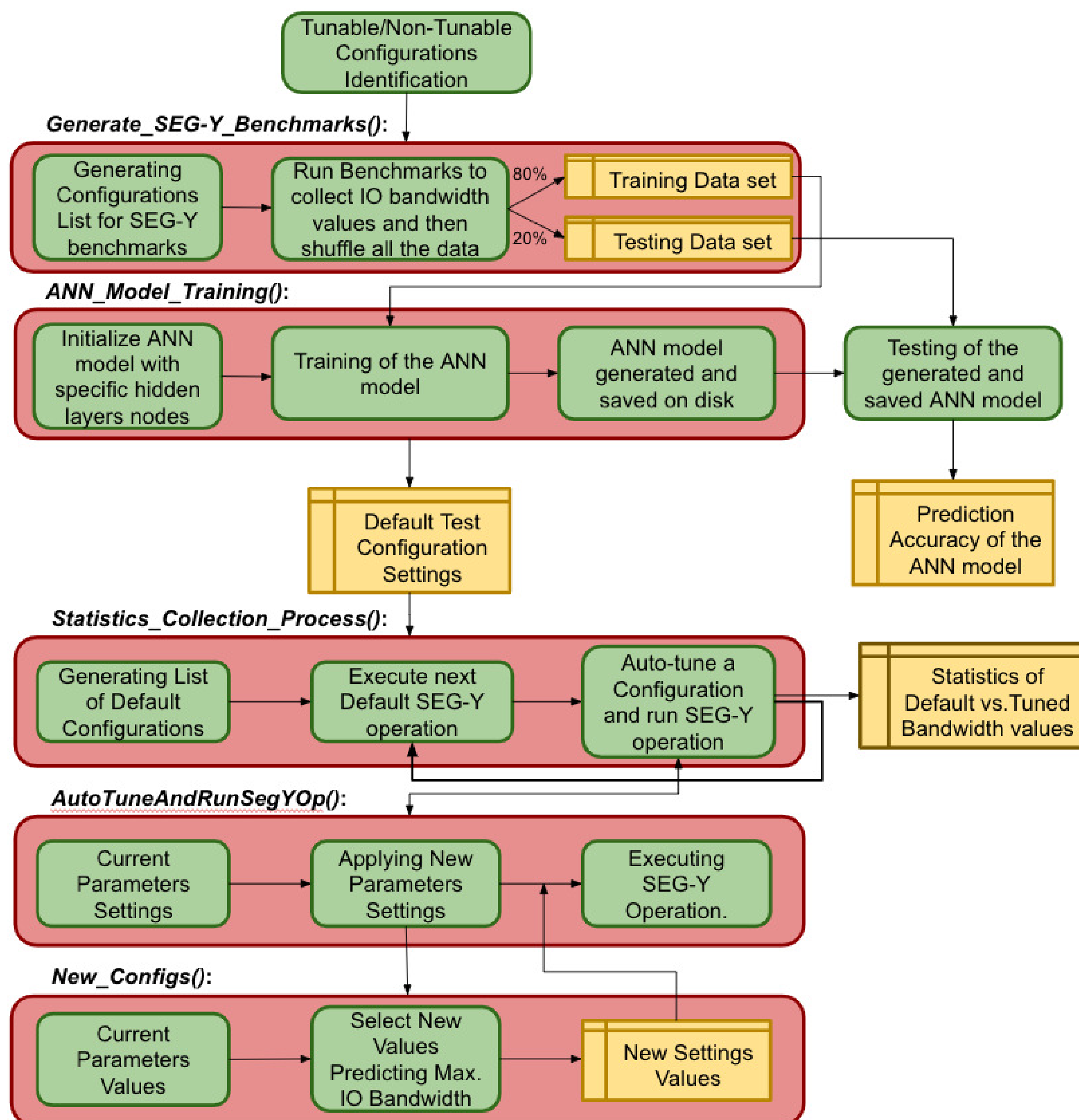


Figure 7.1: Research Flow for Auto-tuning SEG-Y operations.

setting in SEG-Y IO list can be presented as follows:-

$$\begin{aligned} \mathit{config_SEG\text{-}Y\text{-}IO} = [&\mathit{MPI\ nodes} = 8, \mathit{processes\ per\ node} = 4,\\ &\mathit{stripe\ count} = 4, \mathit{stripe\ size} = 1MiB,\\ &\mathit{number\ of\ traces} = 512, \mathit{samples\ per\ trace} = 256,\\ &\mathit{file\ access\ pattern} = \mathit{random}] \end{aligned}$$

Similarly, a single configuration setting in a list for SEG-Y file sorting can be as follows:-

$$\begin{aligned} \mathit{config_SEG\text{-}Y\text{-}SORT} = [&\mathit{MPI\ nodes} = 8, \mathit{processes\ per\ node} = 4,\\ &\mathit{stripe\ count} = 4, \mathit{stripe\ size} = 1MiB,\\ &\mathit{number\ of\ traces} = 512, \mathit{samples\ per\ trace} = 256,\\ &\mathit{unsorted\ order} = \mathit{reverse}] \end{aligned}$$

These type of configuration settings in the lists are used to execute both SEG-Y READ and WRITE benchmarks, as well as the sorting benchmarks. The 3rd column in both Tables 7.1 and 7.2, tells if a particular parameter is tunable or not, which is further explained in later.

Table 7.1: SEG-Y IO Benchmarks Configuration Parameters and their values.

Parameters	Values Settings	READ/WRITE Tunability
number of MPI nodes	2,4,8,16	No/No
MPI processes per node	1,2,4,8	No/No
stripe count	2,4,8,16	No/Yes
stripe size (MiBs)	1,256,512,1024,2048	No/Yes
file access pattern	contiguous/random	Yes/Yes
number of traces	512,1024,2048,4096	No/No
samples per trace	256,512,1024,2048	No/No
IO operation	READ/WRITE	Not Applicable

Table 7.2: SEG-Y file sorting Benchmarks Configuration Parameters and their values.

Parameters	Values Settings	READ/WRITE Tunability
number of MPI nodes	2,4,8,16	No/No
MPI processes per node	1,2,4,8	No/No
stripe count	2,4,8,16	Yes/Yes
stripe size (MBs)	1,256,512,1024,2048	Yes/Yes
unsorted order	uniform/random/reverse	No/No
number of traces	512,1024,2048,4096	No/No
samples per trace	256,512,1024,2048	No/No
IO operation	contiguous READ/WRITE	Not Applicable

7.3.3 Generating SEG-Y IO and file sorting Benchmarks Results Data

The procedure to execute SEG-Y IO and file sorting benchmarks, and collect the bandwidth profiling data, is specified in Listing 7.1, defined as `Generate_SEG-Y_Benchmarks()`. This method takes the input as all the parameter settings values specified in Table 7.1 and 7.2. It first generate two separate lists of all possible combinations SEG-Y IO configurations and SEG-Y file sorting configurations, respectively, on Line 2 to 3. Here each combination in either list is a set of single configuration settings values as stated in earlier section 7.3.2. So, Line 4 to 6 are two nested loops applied to iterate both lists and their configuration sets.

In a loop iteration for each configuration setting, the pre-benchmark settings are applied prior to the execution of main benchmark execution, which are: LFS stripe settings, file generation and Darshan settings. From Line 13 to 17, the LFS stripe settings are applied on an empy file, accordingly, which is then generated to be read/write or sorted with the specific access pattern or sorting order, respectively. Subsequently, on Line 19, the Darshan setting is applied which just sets the path for darshan file to be generated after the main benchmark execution. This is to keep the IO bandwidth profiling data.

It should be noted that the Darshan is a HPC IO characterization tool for parallel IO bandwidth profiling [91]. The LFS and Darshan pre-benchmark settings are also being used during auto-tuning process for test cases, discussed in later section.

For SEG-Y IO operations, the SEG-Y file is generated in `uniform` order with the LFS stripe settings. The file is striped across number of Lustre Object Storage Targets (OSTs) as a `stripe count` value, before benchmark execution. This distribution of this SEG-Y file over the parallel OST disks is according to certain `stripe size` unit value in a round-robin fashion [8].

The SEG-Y IO benchmarks read or write traces data that makes the amount of `number of traces` and `samples per trace` within a file. The MPI processes read or write their respective parts of trace data in either `contiguous` or `random` access pattern on Line 22. However, the file should first be generated with a `uniform` order on Line 19. The uniform order is the ascending order of trace data according to trace header source-x coordinate [1, 4]. According to possible value settings mention in Table 7.1, the total maximum parameters configurations makes total of 20480 benchmarks executions where 10240 are for each READ/WRITE executions.

The SEG-Y file sorting benchmarks are a bit complex in nature than the basic IO executions. An unsorted input file is given to MPI processes to sort it in ascending order with respect to source-x trace header coordinate in an output file. Here, in both cases of `contiguous READ` and `contiguous WRITE` an output file is ultimately going to be written on disks either non-contiguously or contiguously. When file is read contiguously then it is written on disk non-contiguously in sorted order, which is the case of `contiguous READ`. In case of `contiguous WRITE` the file is read non-contiguously and written to disk contiguously in sorted order.

Initially, it requires an input SEG-Y file to be created with in regards to corresponding `unsorted orders` values mentioned in Table 7.2. The `uniform` order means trace data is sorted in ascending order with respect to source-x coordinate from the trace header value as mentioned earlier [4]. In differentiation, the `reverse` order is the descending order of trace data and `random` order is any arbitrary order of trace data.

During the creation of an input SEG-Y file with any of the unsorted orders as stated, the file is distributed across the OSTs according to Lustre striping parameters as the pre-benchmark LFS settings, from Line 13 to 17. This is similar to the case of SEG-Y IO benchmarks with respect to `stripe count` and `stripe size` file distribution settings. Now during the execution of a benchmark on Line 22, in case of `contiguous READ` the MPI processes will read contiguously their chunk of unsorted traces in their local memory. Then the traces are reordered using a mapping from unsorted input local trace index to the sorted output global trace index with respect to source-x coordinate. Subsequently, this sorted index map is used to write non-contiguously the reordered or sorted trace data into the output SEG-Y file. For the case of `contiguous WRITE`, the trace indices are already sorted using the unsorted local to sorted global index mapping. Therefore, each MPI process read trace from file non-contiguously using that sorted index mapping of trace indices. Once the traces are reordered or sorted by each process then they are written contiguously into the SEG-Y output file. In both scenarios the output file is also Lustre striped across OSTs using the same `stripe count` and `stripe size` values used to generate and distribute the input SEG-Y file. This happens when a configuration settings benchmark is being executed and an output file is produced.

To summarize, the trace data in an input and output SEG-Y files is generated with respect to specified `unsorted order`, `number of traces`, `samples per trace` and Lus-

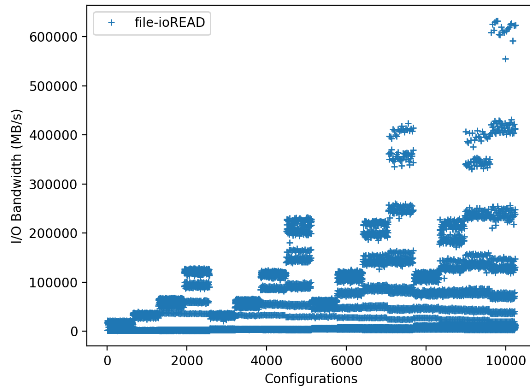
Listing 7.1: Overview of SEG-Y IO and file sorting Benchmarks Execution and Bandwidth Profiling.

```
1  def Generate_SEG−Y_Benchmarks(Parameter_Settings):
2  SEG−Y_IO_Configs_List, SEG−Y_Sorting_Configs_List =
3      Generate_Lists(Parameter_Settings)
4  for each Configs_List in [SEG−Y_IO_Configs_List,
5      SEG−Y_Sorting_Configs_List]:
6      for each config_set in Configs_List:
7          # A config_set is a single configuration parameter
8          # setting as mentioned with example in section 7.3.2.
9          # Furthermore, each set also hold the other required
10         # key/values e.g. file name, darshan file path, etc.
11
12         # Pre−benchmarking applied each config set.
13         Set_Lustre_Striping(config_set['file_name'],
14                     config_set['stripe_count'],
15                     config_set['stripe_size'])
16
17         Make_File(config_set['file_name'])
18
19         Fix_Darshan_File_Path(config_set['darshan_file'])
20
21         # SEG−Y IO/Sorting benchmarking of this config_set.
22         Execute_SEG−Y_IO_or_Sort_Benchmark(config_set)
23
24         # Append parsed bandwidth and config_set in YAML file.
25         bandwidth = Read_Bandwidth(config_set['darshan_file'])
26         Append_Output_in_Yaml_File(config_set,bandwidth_value)
27
28         # Applying post−benchmark as deleting the target file.
29         Delete_Benchmark_Target_File(config_set['file_name'])
```

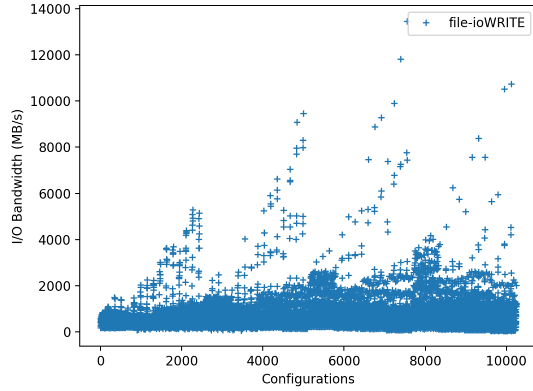
tre stripe values settings. By considering all the possible value settings mentioned in Table 7.2, the total maximum parameters configurations or benchmarks to run are 30720 where 15360 are for each `contiguous READ/WRITE` executions.

The initial LFS settings are applied to target files to test the IO or sorting bandwidth performance against a specific configuration settings benchmark. This is in regard to its distribution of file on parallel disks (OSTs). The parallel IO bandwidth performance during each benchmark execution is measured by the Darshan software tool. The benchmarked file is deleted after execution completion to release the cache, being a post-benchmark setting on Line 29.

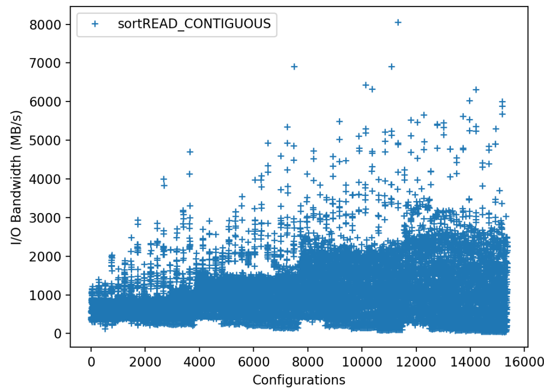
The parallel IO bandwidth profiling results are appended to the end of YAML file after being parsed from generated Darshan file on Line 25 and 26. This is against each benchmark parameters configuration settings execution. These results are kept in form of dictionary style object for each execution benchmark. The Figure 7.2 shows the IO bandwidth profiling from all executed benchmarks results of SEG-Y IO and sorting operations.



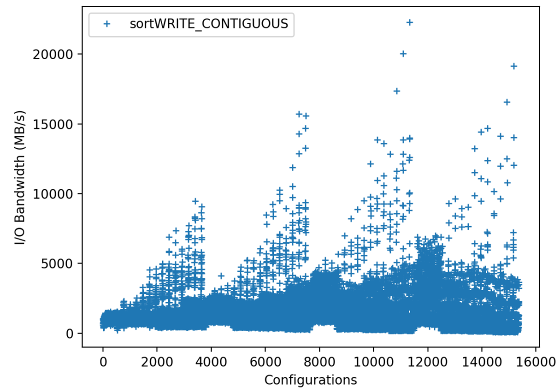
(a) SEG-Y File-IO READ Results



(b) SEG-Y File-IO WRITE Results



(c) SEG-Y File Sort READ Results



(d) SEG-Y File Sort WRITE Results

Figure 7.2: SEG-Y Benchmarks Results.

7.3.4 Predictive IO bandwidth Modelling with ANNs

As mentioned earlier, to achieve goals of this research, a ML process for predictive IO modelling has been used to estimate bandwidth value, which is specifically the ANN technique. This is to support auto-tuning process in next step. The training and testing sets for the ML procedure are 80% and 20% of the benchmarks results, respectively. This is for each SEG-Y operation type, as mentioned in Table 7.3. The recent related work mentioned in [17] has closely comparable problem with the SEG-Y operations, addressed in this thesis. The portion of some feature parameters is same, and train test rate is almost 80% and 20%. Therefore, the similar ratio is used to split benchmarks results into training and testing datasets.

The ANNs generated in this research according to their structure description in Table 7.4 and 7.5, and their corresponding hyper-parameters values in Table 7.6. As this is being done for both SEG-Y IO and file sorting READ/WRITE operations therefore, total trained and tested ANN models are 24. The ANNs input layer nodes map to input configuration parameters, as described in Table 7.5, which are essential to model training and testing.

The training of all ANN models for both SEG-Y IO and sorting operations, has been carried out using the method `ANN_Model_Training()` specified in Listing 7.2. It takes arguments as: `h1` - specifies number of nodes in 1st hidden layer, `h2` - specifies number of

Table 7.3: Training and Testing Sets Partitions from Benchmarks Results.

Operation Type	Total Benchmarks	Training Set	Testing Set
SEG-Y IO READ	10240	8192	2048
SEG-Y IO WRITE	10240	8192	2048
SEG-Y Sort READ Contiguous	15360	12288	3072
SEG-Y Sort WRITE Contiguous	15360	12288	3072

nodes in 2nd hidden layer, `dp` - specifies dropout ratio for hidden layers during training and `wd` - specifies weight decay layers nodes, as mentioned in Line 2. On Line 5 it set the values representing number of nodes in all the layers of the ANN. This is to initialize the ANN model on Line 9 to 14, with dropout ratios and activation function `ReLU()`. The `ReLU()` is rectified linear unit activation function which gives the anticipated value as output from layer to layer, on the nodes [183].

Afterwards, on Line 16 the criteria to compute loss between actual and predicted value, has been defined, which is the Mean Squared Error (MSE) [185]. Subsequently on Line 18, the optimizer is defined by `Adam()` which is a gradient descent method to reach the global minimum point. When gradient is close zero or global minimum, a model is able to make precisely accurate predictions. The `Adam()` requires the learning rate and weight decay values as responsible for the speed of converging gradient to global minimum. Therefore, 0.002 learning rate for all SEG-Y operations modelling and their specific weight decay values mentioned in Table 7.6, are applied to train the model in an adequate time.

The Line 20 and 21 sets the `X` and `y` matrices containing training set configuration settings and their corresponding actual profiled IO bandwidth values, respectively. These settings and values are scaled between 0 and 1 using `MaxAbsScaler` [111]. Then Line 23 to 34 runs the main training or learning loop iterations till the `MAX_limit` which is set with the accordance of hyper-parameters. This determines how fast a model can converge to global minimum gradient in what maximum iterations. In each iteration, on Line 25 model predicts and save the bandwidth values against all configuration settings in `X` via Feed Forward Propagation pass in `y_predictions`. Then, on Line 27, loss is being computed between actual and predicted bandwidth values using the already defined criterion function as MSE. Afterwards, on Line 29, optimizer Adam zero the gradient and loss value is propagated backward on Line 31. Subsequently, Line 33 and 34 runs optimizer step function and model train function, respectively, to update the weights among nodes from layer to layer.

When iterations are completed, the ANN model is trained therefore, it is then saved in `"ANN_Model.pt"`, as mentioned on last Line 37 of this method. Since all the 24 models have different settings and purpose, therefore, their file names are different from each other in actual to avoid any clash. The prediction evaluation results for all the ANN models have been presented and discussed later in the section of Experimental Result Analysis section.

7.3.5 Full Design Strategy for Auto-tuning Parameters

The generated and saved ANNs are the basis to auto-tuning parameters during a running SEG-Y application prior to IO or the sorting operations. Therefore, it is important to note that only a proportion of configuration parameters are tunable. This depends on the

Table 7.4: ANNs Structure Configuration.

Layers	ANN-1	ANN-2	ANN-3	ANN-4	ANN-5	ANN-6
Input	7	7	7	7	7	7
H1	256	128	64	32	16	8
H2	128	64	32	16	8	4
Output	1	1	1	1	1	1

Table 7.5: Mapping of ANNs Input/Output nodes to Configuration Parameters for SEG-Y IO and File Sort Bandwidth Modelling.

input layer Nodes	Configuration Parameters
Node-1	number of MPI nodes
Node-2	MPI processes per node
Node-3	stripe count
Node-4	stripe size
Node-5	number of traces
Node-6	samples per trace
Node-7	file access pattern (for IO) or unsorted order (for sorting)
output layer Node	**Output Parameter**
Node-1	IO bandwidth value

Table 7.6: Hyper-parameters for SEG-Y IO and file sorting prediction models.

Operation Type	Weight Decay	Learning rate	Dropout
SEG-Y IO READ	0.0	2×10^{-3}	0.0
SEG-Y IO WRITE	0.0	2×10^{-3}	0.05
SEG-Y Sort `contiguous` READ	10^{-5}	2×10^{-3}	0.05
SEG-Y Sort `contiguous` WRITE	0.0	2×10^{-3}	0.0

Listing 7.2: ANN based ML Process over Benchmarks Results.

```
1  import torch.nn as nn
2  def ANN_Model_Training(h1,h2,dp,wd):
3
4  # Setting the ANN layers to specific number of nodes
5  Input_layer, Hidden_layer_1, Hidden_layer_2, Output_layer =
6      7, h1, h2, 1
7
8  # Applying dropout ratios and activation function to ANN nodes
9  ANN_Model = nn.Sequential(nn.Dropout(p=0.00),nn.ReLU(),
10     nn.Linear(Input_layer,Hidden_layer_1),
11     nn.Dropout(p=dp),nn.ReLU(),
12     nn.Linear(Hidden_layer_1,Hidden_layer_2),
13     nn.Dropout(p=dp),nn.ReLU(),
14     nn.Linear(Hidden_layer_2,Output_layer))
15
16 Model_Criterion = torch.nn.MSELoss()
17
18 ANN_Optimizer = torch.optim.Adam(Model.parameters(), lr=0.002,
19                                  weight_decay=wd)
20 X = Input_training_set # Scaled between 0 and 1.
21 y = Output_training_set # bandwidths scaled between 0 and 1.
22 # Learning loop iterations...
23 for Iteration in range(MAX_limit):
24     # Forward Propagation for predicting IO bandwidth values.
25     y_predictions = ANN_Model(X)
26     # Computing loss via model criterion MSE.
27     Loss = Model_Criterion(y, y_predictions)
28     # Zeroing the gradients.
29     ANN_Optimizer.zero_grad()
30     # Performing Back Propagation pass using the Loss.
31     Loss.backward()
32     # Updating the weights from layer to layer.
33     ANN_Optimizer.step()
34     ANN_Model.train()
35
36 # Saving the trained ANN model in .pt file.
37 torch.save(ANN_Model,"ANN_Model.pt")
```

SEG-Y operation type and its given non-tunable parameters values. However, the SEG-Y file sorting has only two same tunable parameters for both contiguous READ/WRITE operations. This is because both operations are ultimately writing a sorted file on the disks.

The 3rd column in Table 7.1 and 7.2, `Yes` means that parameter is tunable and `No` means non-tunable, depending on READ/WRITE operation. The common non-tunable parameters are `number of MPI nodes`, `MPI processes per node`, `number of traces` and `samples per trace`. The MPI nodes and processes cannot be changed once the SEG-Y MPI application is started, and the formation of trace data cannot be tuned as well being a user requirement. For SEG-Y IO READ case, the only option is to switch between the `file access patter` which is contiguous read or random read. As far as SEG-Y WRITE operation is concerned, the `stripe count`, `stripe size` and `file access pattern` settings, can be tuned. In case of SEG-Y READ operation, the file is already striped across networked parallel Lustre OSTs (disks) with specific `stripe count` and `stripe size`. Therefore, changing these values at runtime cannot change the file striping arrangement on the disks. Whereas, writing a file with stripe values to read is an additional overhead in this scenario. For the case of SEG-Y WRITE operation, the changed or tuned stripe values can impact the bandwidth during execution.

The SEG-Y file sorting by contiguous READ/WRITE operations have only two tunable parameters which are `stripe count` and `stripe size` values. In SEG-Y Sorting, the `unsorted order` is also a non-tunable parameter in addition to previous common non-tunable parameters. By changing its value also does not impact in bandwidth as this was just used to generate a SEG-Y file with a particular unsorted order before actual benchmark execution. The stripe values in contiguous READ/WRITE operations can be tuned because both write a sorted file eventually. Therefore, both operations bandwidth is expected to be impacted at runtime.

By consideration of the tunabilities of parameters, the following steps for flow of execution are: 1) retrieve the existing configurations, 2) compute the bandwidth predictions on the provided configurations and other tunable/non-tunable possible value settings, 3) select the configurations having maximum performance of IO bandwidth predicted and 4) Change the tunable parameters settings with selected configurations. This procedure is exhibited in detail by Listing 7.3 and 7.4.

7.3.5.1 New Configuration Settings by SEG-Y Operations Models

The method `New_Configs()` in Listing 7.3 returns the set of newly suggested values of tunable configuration parameters against the existing settings in `current_configs` argument on Line 2. The new configurations predicting maximum bandwidth from the different combination of values are returned in `max_configs` at the end of this method. It also takes third argument for ANN model ".pt" file in `model_path` specific to second argument against SEG-Y operation type in `op_type`. According to SEG-Y IO and Sorting operations stated earlier, the possible string values specified for `op_type` are: "SEGYRead", "SEGYWRITE", "SortRead" and "SortWrite", from programming point of view. The PyTorch (`torch`) package is imported on Line 1 for loading the specific generated ANN model on Line 5. The Line 8 to 10 get the existing settings in `X`, predict the current bandwidth value as maximum in `max_bandwidth_value`. This is predicted by calling `seg_y_model(X)`, and `max_configs` holds the current settings as maximum bandwidth settings, initially. Afterwards, list of tunable and possible value settings are initialized

from Line 13 to 15. These lists are initialized as `file_access_types`, `stripe_counts` and `stripe_sizes` as per values mentioned in Table 7.7 and 7.8.

The procedure to configure new parameter values, checks all the possible configurations of tunable settings with the given non-tunable parameters values. The one combination of values giving the highest possible predicted bandwidth, is the chosen parameters setting. This is elaborated from Line 18 and 33. The procedure covers all the 4 types SEG-Y operations as stated earlier. The lists of `file_access_types`, `stripe_counts` and `stripe_sizes` are not applicable on all SEG-Y operations types except "SEGY-Write". The "SEGYRead" operation only requires `file_access_types` to check and set the highest possible bandwidth setting. Both SEG-Y file sorting operations "SortRead" and "SortWrite" only require Lustre settings `stripe_counts` and `stripe_sizes` to check and set maximum bandwidth settings. In order to achieve this, the nested loops are run for checking all possible settings. However, the condition statements are set in place from Line 21 to 23 to only set the values in iteration according to a specific type of SEG-Y operation about to execute. Subsequently onward, from Line 31 to 33 have conditions to break loop for particular settings which are not applicable to currently executing SEG-Y operation.

In loop, once the values are set in `X` according to SEG-Y operation type then `new_bandwidth_value` is computed by calling `segy_y_model(X)` on Line 27. Then Line 28 to 30 checks if the `new_bandwidth_value` is greater than current `max_bandwidth_value` then update the `max_bandwidth_value` with the `new_bandwidth_value` and `max_configs` with currently checked settings in `X`. By the end of loop the `max_configs` contains the new configuration settings of tunable and unchanged values of non-tunable parameters, predicting maximum IO bandwidth value. These new maximum bandwidth settings are returned on Line 34, to the application running that particular SEG-Y operation whose method is represented in Listing 7.4. If for some reason the new configurations are exactly as they were initially, it means the current configuration settings predict highest possible IO bandwidth value from other combinations of settings.

7.3.5.2 Auto-tuning SEG-Y Operations with New Configuration Settings

This section gives the overview of auto-tuning within the application before executing a SEG-Y operation. The Listing 7.4 presents the method `AutoTuneAndRunSegYOp()` which auto-tune settings on the basis of newly chosen settings returned from method in Listing 7.3 `New_Configs()`. It takes arguments as current parameters settings in `current_configs`, SEG-Y operation type in `op`, target file path in `t_file` and ANN model ".pt" file path in `model_path`.

Initially, the ExSeisDat Parallel IO Library (PIOL) has been included, followed by acquiring its namespace on Line 1 and 2, respectively. Then in the scope of this method, first, the ExSeisDat environment is initialized on Line 6. Secondly, the current MPI process number is retrieved in variable `rank` on Line 9. Then on Line 12 and 13, it checks if current `rank` is 0, it means the current process is a head MPI process. Subsequently, it calls the `New_Configs()` method defined in Listing 7.3. This is to get the new configuration settings predicting maximum bandwidth. These settings are returned in `max_configs` memory of `struct` type `max_configs_type`. Meanwhile, all other MPI processes ranks are waiting until rank-0 is finished getting new settings, on Line 16 via `MPI_Barrier()` method. Afterwards on Line 17, rank-0 process send the new settings to all other ranks. All those ranks receive the new settings in their corresponding ob-

Listing 7.3: New Configuration Parameters Settings based on predicted maximum bandwidth.

```
1  import torch
2  def New_Configs(current_configs,op_type,model_path):
3
4      # Loading SEG-Y Model w.r.t IO/Sorting and h1-h2 nodes.
5      seg_y_model = torch.load(model_path)
6
7      # current_configs = tunable:non-tunable values.
8      X = current_configs # existing settings.
9      max_bandwidth_value = seg_y_model(X)
10     max_configs = current_configs
11
12     # Lists of value settings to check prediction.
13     file_access_types = ['contiguous','random']
14     stripe_counts = [2,4,8,16]
15     stripe_sizes = [1,256,512,1024,2048] # in MBs
16
17     # Setting configurations predicting maximum bandwidth.
18     for file_access_type in file_access_types:
19         for stripe_count in stripe_counts:
20             for stripe_size in stripe_sizes:
21                 if op_type == "SEGYWrite" or "SEGYRead":
22                     X['file_access_type'] = file_access_type
23                 if op_type == "SEGYWrite" or "SortRead" or
24                    "SortWrite":
25                     X['stripe_count'] = stripe_count
26                     X['stripe_size'] = stripe_size
27                 new_bandwidth_value = seg_y_model(X)
28                 if new_bandwidth_value > max_bandwidth_value:
29                     max_bandwidth_value = new_bandwidth_value
30                     max_configs = X
31                 if op_type == "SEGYRead": break
32             if op_type == "SEGYRead": break
33         if op_type == "SortRead" or "SortWrite": break
34     return max_configs
```

Listing 7.4: Auto-tuning before SEG-Y Operations Execution.

```
1  #include <ExSeisDat/PIOL.hh>
2  using namespace exseis::PIOL;
3  void AutoTuneAndRunSegYOp(char *current_configs[], string op,
4                              string t_file, string model_path){
5    // Loading ExSeisDat Environment on MPI and IO Library ...
6    auto piol = ExSeis::New();
7
8    // Current MPI rank from ExSeisDat's PIOL Wrapper functions.
9    size_t rank = piol->getRank();
10
11   // Get New Configuration parameters settings on rank-0 ...
12   if(rank == 0)
13     max_configs = New_Configs(current_configs, op, model_path);
14
15   // All ranks wait and get new config settings for tuning.
16   MPI_Barrier(MPI_COMM_WORLD);
17   MPI_Bcast(&max_configs, 1, max_configs_type, 0, MPI_COMM_WORLD);
18
19   if (op == SEGYWrite || op == SEGYRead)
20     file_access_pattern = max_configs.file_access_pattern;
21   // New stripe settings are only run by rank 0 to write file.
22   if(rank == 0 && (op == SEGYWrite || SortWrite || SortRead)){
23     Remove_Previous_Target_File(t_file);
24     Apply_New_Stripe_Settings(t_file, max_configs.stripe_count,
25                               max_configs.stripe_size);
26   }
27   MPI_Barrier(MPI_COMM_WORLD);
28   // Remaining section of the application continue ...
29   // Running SEG-Y operations by new tuned value settings ...
30 }
```

jects of `max_configs`. This is executed by calling `MPI_Bcast()` method. The methods `MPI_Barrier()` and `MPI_Bcast()` are mentioned and explained in [13].

Once all the ranks or processes are updated with new configuration settings then, it checks for the currently required SEG-Y operation type apply tuning accordingly. On Line 19 and 20, it checks if the current operation is one of the SEG-Y IO operations "SEGYWrite" or "SEGYRead". Then all processes update their `file_access_pattern` in order to READ or WRITE SEG-Y file according to the access pattern set. Subsequently, on Line 22 to 24, it checks if current rank is 0 and operation type is either "SEGYWrite" or one of the file sorting operations: "SortWrite" or "SortRead". Then it removes the previous target file path specified in `t_file` and apply new lustre settings specified in `max_configs.stripe_count` and `max_configs.stripe_size` on an empty target file. Meanwhile again, all other processes wait on Line 27 until rank-0 process is finished applying new Lustre stripe settings. Afterwards, all processes runs the remaining section of application to complete the task of particular SEG-Y operation type with new tuned configuration settings. This is how the goal of auto-tuning parameters prior to IO execution on runtime, has been achieved.

On the completion of one execution of a SEG-Y operation through the application method in Listing 7.4, the new IO bandwidth value is captured via Darshan profiling as stated earlier and elaborated by method in Listing 7.5, explained in the next section.

7.3.5.3 Statistical Data Process upon Executing Auto-tuned SEG-Y Operations

A statistical analysis for performance evaluation requires experimental code setup. This involves computations of specific metrics which show the real picture of the approach's performance. In this scenario, the default configuration settings and the auto-tuned settings are executed to make comparison between their IO bandwidths. This is eventually conducted by computing the statistical metrics and the overall bandwidth percentage improvement. The Tables 7.7 and 7.8 present the default configuration settings test cases to execute and auto-tune for SEG-Y IO and file sorting, respectively. The collected IO bandwidths against these default settings and their corresponding auto-tuned settings leveraged the statistical analysis.

In this section, an experimental Python code setup is presented, to run auto-tuning benchmarks on the test cases stated above for all SEG-Y operations. This procedure is elaborated in Listing 7.5 method `Statistics_Collection_Process()`. The Line 1 to 3 imports the required Python packages as `torch` (PyTorch), `numpy` and `statistics`. The arguments of this method are: `X` - a list of all possible test configuration settings specific SEG-Y operation from Table 7.7 or 7.8, `N` - the total number of test cases in a test set, `op` - a type of a specific SEG-Y operation as stated earlier, `H1` - number of nodes in hidden layer 1 of an ANN, `H2` - number of nodes in hidden layer 2 of an ANN and `model_path` - ANN model file path against `H1` and `H2` nodes configuration.

The method begins with main loop on Line 5 which runs till `N` default test settings. It should be noted for SEG-Y IO operations the value of `N` is 1458 for each READ and WRITE execution. Whereas, it will be 2187 for SEG-Y file sorting operations for each contiguous READ and WRITE execution. The value of `N` is computed by multiplying numbers of value settings against each parameter specified in Table 7.7 or 7.8, excluding the `IO operation` parameter.

On Line 6 and 7, the two lists; `Def_bandwidth` and `Tuned_bandwidth` are initialized to contain bandwidth values against a single default configuration and its auto-tuned setting execution, respectively. Then Line 8 has variable `repititions`, which specifies the number of times the default and auto-tuned settings would be executing. Subsequently, their bandwidth values would be saved in their corresponding lists stated earlier. Since the repetitions of a configuration settings can run any number of times, therefore, the nested inner loop is applied on Line 9. So, a default and auto-tuned settings would run repeatedly according to repetitions specified on Line 8, which is 2 in this case. This has been done to ensure the bandwidth corresponding to the settings are approximately same when they are repeated and averaged later on. The reason is sometimes bandwidth can be low at an instance due interference on OSTs by other user programs.

In the inner loop after Line 9, first the IO or sorting operation is executed with default setting from Line 11 to 16. The Line 11 runs required settings for LFS and Darshan, which set their target file names and paths for an operation in variables `t_file` and `d_file`, respectively. The naming of files is according to a specific SEG-Y operation - `op`, number of nodes in hidden layers - `H1` and `H2`, and the Lustre stripe setting values of an ith configuration - `X`. Then Line 13 generates file according to current configuration LFS stripe settings. Subsequently, the IO or sorting operation with default configuration settings is executed on Line 14. The bandwidth value of the current configuration setting, is profiled in the darshan file path set in `d_file` which is then parsed and appended in `Def_bandwidth` on Line 15. The target darshan file and the operation file with LFS

settings, are deleted to free the cache.

Once an operation with default setting is executed, then it will be re-executed by auto-tuning that default setting from Line 19 to 24. The steps to execute IO or sorting operations by auto-tuning the configurations, are similar to executing with default settings. However, the first difference is on Line 22, which calls `AutoTuneAndRunSegYOp` method described in Listing 7.4, to auto-tune configurations and execute the SEG-Y operation. The second difference is on Line 23 where the bandwidth of auto-tuned SEG-Y IO or Sorting operation is appended in `Tuned_bandwidth` list.

After the completion of the inner iterations of a configuration setting, their default and the tuned bandwidths are averaged and appended in the main lists `Old_bandwidths` and `New_bandwidths` on Line 26 and 27, respectively. These lists would hold the default and tuned bandwidth values against `N` configuration settings test cases. On Line 28, it checks for current configuration that if its tuned or new bandwidth is greater than its default or old bandwidth then improvement count `C_IM` is incremented.

The completion of outer main loop is followed by computing the mean default and tuned bandwidths by averaging the values of `Old_bandwidths` and `New_bandwidths` lists, respectively, on Line 30 and 31. Then the overall percentage improvement in bandwidth is computed on Line 32. The percentage of test cases with improved bandwidths is computed in `per_IMP_cases` on Line 34. Afterwards, from Line 35 to 44, the remaining statistics are computed, which include maximum, minimum, median, standard deviation and variance values against both default and Tuned bandwidths, separately.

The whole procedure explained above using Listing 7.3, 7.4 and 7.5 completes the auto-tuning design for optimizing SEG-Y operations in HPC cluster, based on the ANNs predictions. The statistical analysis of auto-tuning results, is presented in the next section of Experimental Results Analysis.

Table 7.7: Default SEG-Y IO Configurations Test Cases to Auto-tune.

Parameters	Default Settings
Number of MPI nodes	16,8,4
MPI processes on each node	16,8,4
Stripe count	16,8,4
Stripe size (MiBs)	512,1024,2048
Number of traces	4096,8192,16384
Samples per trace	4096,8192,16384
IO operation	read,write
File access pattern	contiguous,random

7.4 Experimental Results Analysis

The SEG-Y IO and file sorting test cases for auto-tuning have been executed from 4 to 16 nodes of KAY cluster of ICHEC, for performance optimization analysis. Each node consists of 2x 20-core 2.4 GHz Intel Xeon Gold 6148 (Skylake) processors [51]. The Lustre OSTs (disks) are utilized from 2 to 16 in range. The ML processes for ANN models have been carried out using one of the GPU node of KAY having NVIDIA Tesla V100 16GB PCIe (Volta architecture) card, via Tensors memory construct in PyTorch [184, 22]. The

Listing 7.5: Statistics Collection Process upon Auto-tuning SEG-Y Operations.

```
1  import torch
2  import numpy
3  import statistics
4  def Statistics_Collection_Process(X,N,op,H1,H2,model_path):
5      for i in range(N):
6          Def_bandwidth = list()
7          Tuned_bandwidth = list()
8          repetitions = 2
9          for j in range(repetitions):
10             # Execute SEG-Y IO or Sorting with Default Setting
11             t_file , d_file = Darshan_and_Lustre_Settings(op,H1
12                 ,H2,X[i]['stripe_count'],X[i]['stripe_size']))
13             Make_File(t_file)
14             Execute_default_setting(X[i],op,t_file)
15             Def_bandwidth.append(parseDarshanFile(d_file))
16             Remove_target_files(t_file ,d_file)
17
18             # Auto-tune settings and run SEG-Y IO or Sorting
19             t_file , d_file = Darshan_and_Lustre_Settings(op,H1
20                 ,H2,X[i]['stripe_count'],X[i]['stripe_size']))
21             Make_File(t_file)
22             AutoTuneAndRunSegYOp(X[i],op,t_file ,model_path)
23             Tuned_bandwidth.append(parseDarshanFile(d_file))
24             Remove_target_files(t_file ,d_file)
25
26         Old_bandwidths.append(statistics.mean(Def_bandwidth)
27         New_bandwidths.append(statistics.mean(Tuned_bandwidth)
28         if (New_bandwidths[i] > Old_bandwidths[i]): C_IM += 1
29
30     mean_def_bandwidth = statistics.mean(Old_bandwidths)
31     mean_tuned_bandwidth = statistics.mean(New_bandwidths)
32     Percentage_improvement = (mean_tuned_bandwidth -
33             mean_def_bandwidth) / mean_def_bandwith * 100.0
34     per_IMP_cases = C_IM / N * 100.0
35     Max_tuned_bandwidth = max(New_bandwidths)
36     Max_def_bandwidth = max(Old_bandwidths)
37     Min_tuned_bandwidth = min(New_bandwidths)
38     Min_def_bandwidth = min(Old_bandwidths)
39     Median_tuned_bandwidth = statistics.median(New_bandwidths)
40     Median_def_bandwidth = statistics.median(Old_bandwidths)
41     Stdev_tuned_bandwidth = numpy.stdev(New_bandwidths)
42     Stdev_def_bandwidth = numpy.stdev(Old_bandwidths)
43     Var_tuned_bandwidth = Stdev_tuned_bandwidth**2
44     Var_default_bandwidth = Stdev_def_bandwidth**2
```

Table 7.8: Default SEG-Y Sorting Configurations Test Cases to Auto-tune.

Parameters	Default Settings
Number of MPI nodes	16,8,4
MPI processes on each node	16,8,4
Stripe count	16,8,4
Stripe size (MiBs)	512,1024,2048
Number of traces	4096,8192,16384
Samples per trace	4096,8192,16384
IO operation	contiguous read/write
Unsorted order	uniform,random,reverse

results are mainly comprises of two parts: ANNs Performance Analysis and Auto-tuning Results Analysis

7.4.1 ANNs Performance Analysis

Being the prediction accuracy as one of the metric used to compare the performance of ML models in previous couple of studies [17, 21, 23]. Therefore, it is used in this research and how it is affected by variation in hidden layers configuration of nodes. In this section, the impact of changing hidden layers nodes is shown on bandwidth prediction accuracy of ANN models for all SEG-Y operations.

For testing the models, the IO bandwidth data from 20% of benchmarks results as the testing set, are re-scaled to original values. Each model is loaded back in memory at an instance to predict and re-scale the bandwidth values. Then, the prediction accuracy has been measured for all models, using the following equations:

$$Let\ X = testing\ set\ which\ is\ the\ 20\%\ of\ complete\ benchmarks\ data\ set.$$
$$Let\ y = actual\ bandwidth\ values\ set\ against\ X\ configuration\ settings.$$
$$r = model(X)\ gives\ all\ the\ predicted\ values.$$
$$Prediction\ Accuracy = 100.0 - \frac{1}{n}\sum_{i=0}^{n}\left|\frac{(y_i - r_i)}{y_i}\right| \times 100.0\%,$$

where y and r are actual and predicted n number of total bandwidth values, respectively, and i is the ith row of X, y and r. Therefore, y_i is the ith actual bandwidth value and r_i is the ith predicted bandwidth value. This is computed by running $model()$ on X_i the ith configuration parameters values set.

7.4.1.1 SEG-Y IO ANNs Prediction Evaluation

The Figure 7.3 shows the trend of prediction accuracy changes with increasing the number of nodes in the ANNs hidden layers, denoted as (h1,h2). For SEG-Y READ operation predictions, the accuracy is as low as closer to 39.5% with minimum nodes configuration (8,4). The accuracy takes a significant notable increase upto 86.5% when the nodes are just doubled to (16,8). Afterwards, it is insignificantly increased and then almost constant around ≈95% till the nodes reached to (256,128) nodes configuration. In short, it is safe to conclude hidden nodes equal to or above (32,16) nodes should be ideal as far as just the

predictions are concerned. In later, model selection can be more specific after analyzing auto-tuning results.

For SEG-Y WRITE operation predictions, the accuracy starts from 65.5% as a minimum value for (8,4) nodes ANN. It makes ≈3% increase accuracy to reach around 68.2% when nodes configuration is doubled as in case of (16,8) nodes. Switching to (32,16) nodes, it gives notable increase to 83.9% of accuracy. Afterwards, it is gradually increased from ≈87% to 89% as the nodes configuration is changed from (64,32) to (256,128). By analyzing the line gradient, it can be expected that accuracy becomes constant after switching to greater nodes configuration than (256,128). Currently, in this scenario the most suitable ANN for SEG-Y WRITE predictions is one with (256,128) nodes, as it is yielded to maximum accuracy value. However, after analyzing the auto-tuning results it can be more clear to choose the most suitable ANN for auto-tuning and optimizing SEG-Y WRITE operations.

The SEG-Y READ bandwidth prediction accuracy ranges from 39.5% to 96% approximately whereas, for WRITE operation, it ranges from 65.5% to 89.1%, as mentioned in Table 7.9. The Figures 7.4, 7.5 depict the bandwidth predictions against actual values of all the 12 ANN models for SEG-Y IO READ/WRITE operations. Each graph in the figures, represents a specific benchmark and operation type, and hidden layers nodes configuration used in an ANN model. It can be noticed that predictions of ANNs with fewer hidden layer nodes are less likely to follow the actual changing bandwidth pattern as compared to those ANNs with higher hidden layer nodes. The reason is decreased prediction accuracy with fewer hidden layer nodes.

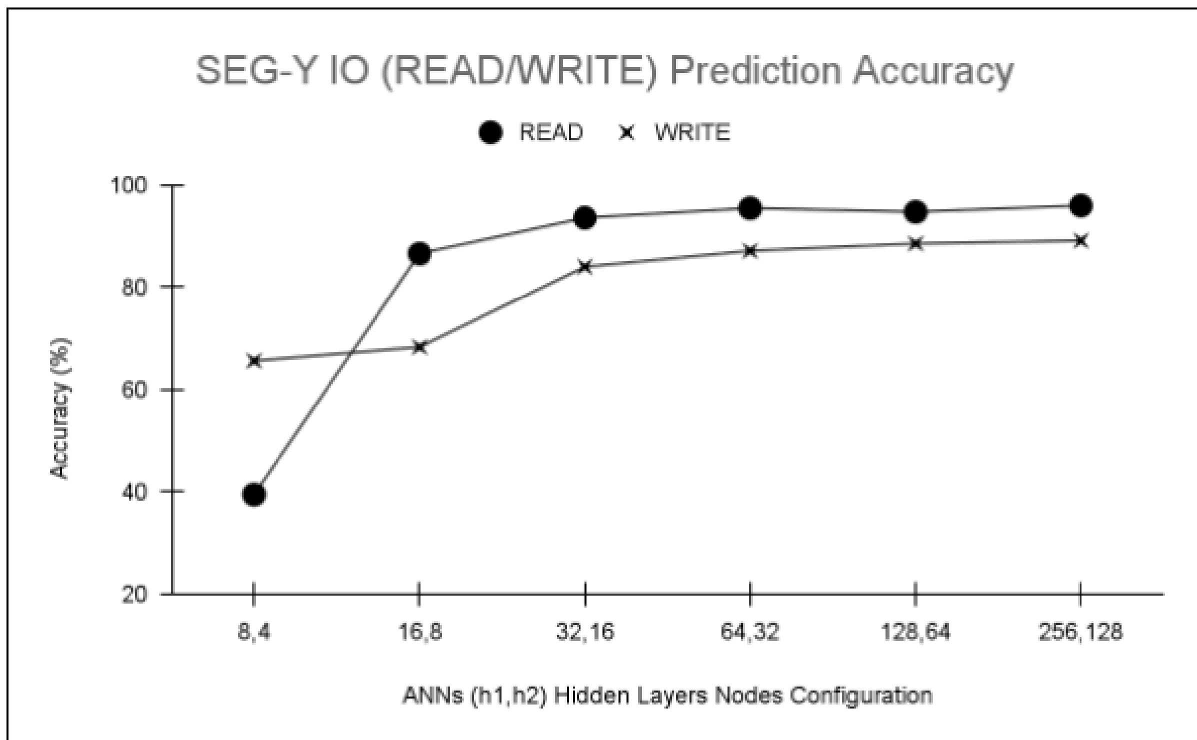


Figure 7.3: SEG-Y IO ANNs Prediction Accuracy.

Table 7.9: SEG-Y IO Prediction Accuracy values.

SEG-Y IO Operation	ANNs					
	8,4	16,8	32,16	64,32	128,64	256,128
READ	39.5	86.5	93.5	95.4	94.7	96.0
WRITE	65.5	68.2	83.9	87.1	88.4	89.1

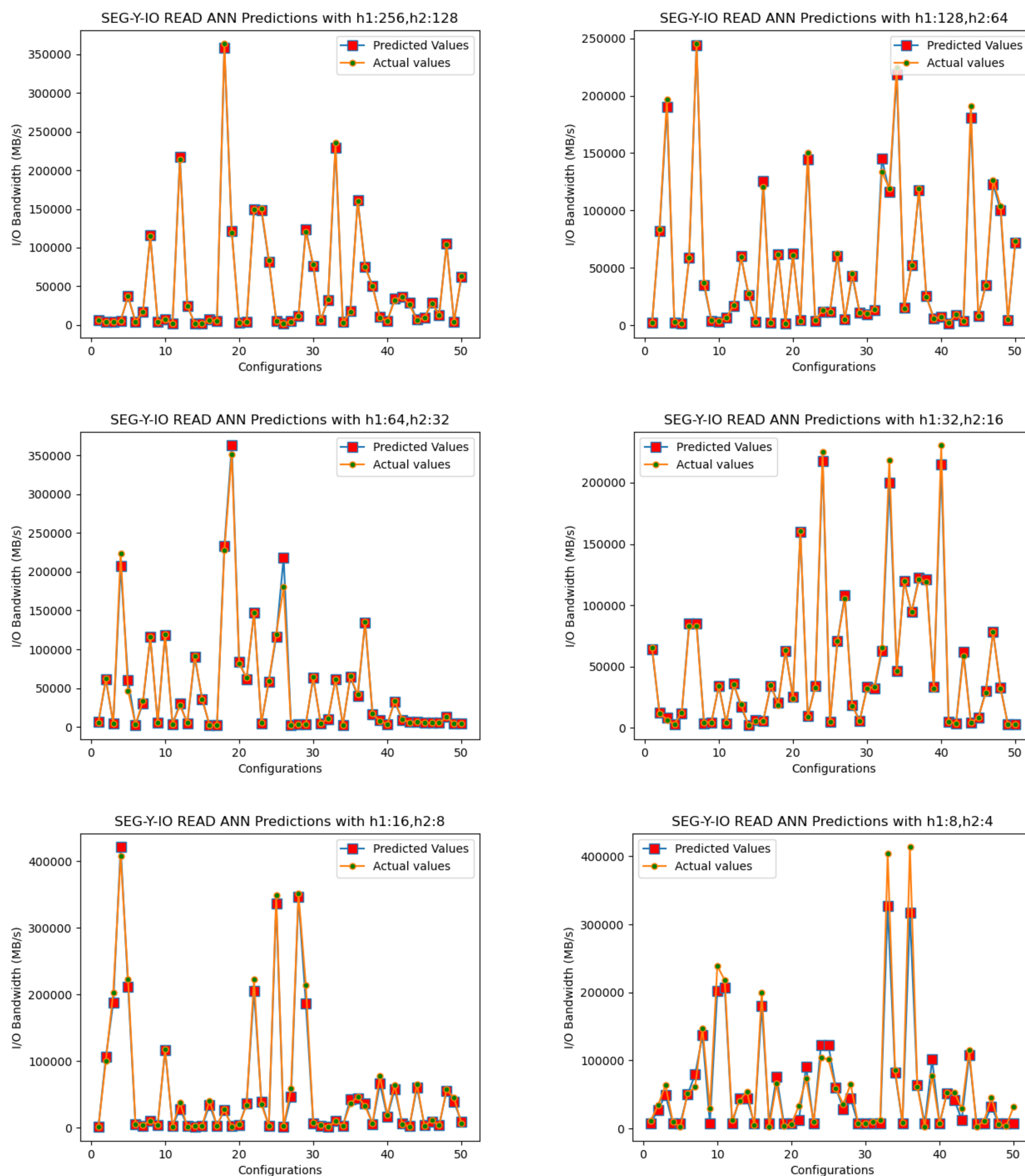


Figure 7.4: SEG-Y IO READ Predictions.

7.4.1.2 SEG-Y file sorting ANNs Prediction Evaluation

The Figure 7.6 shows the trend of SEG-Y file sorting bandwidth prediction accuracy which changes with increasing the number of hidden layers nodes. For SEG-Y file sorting

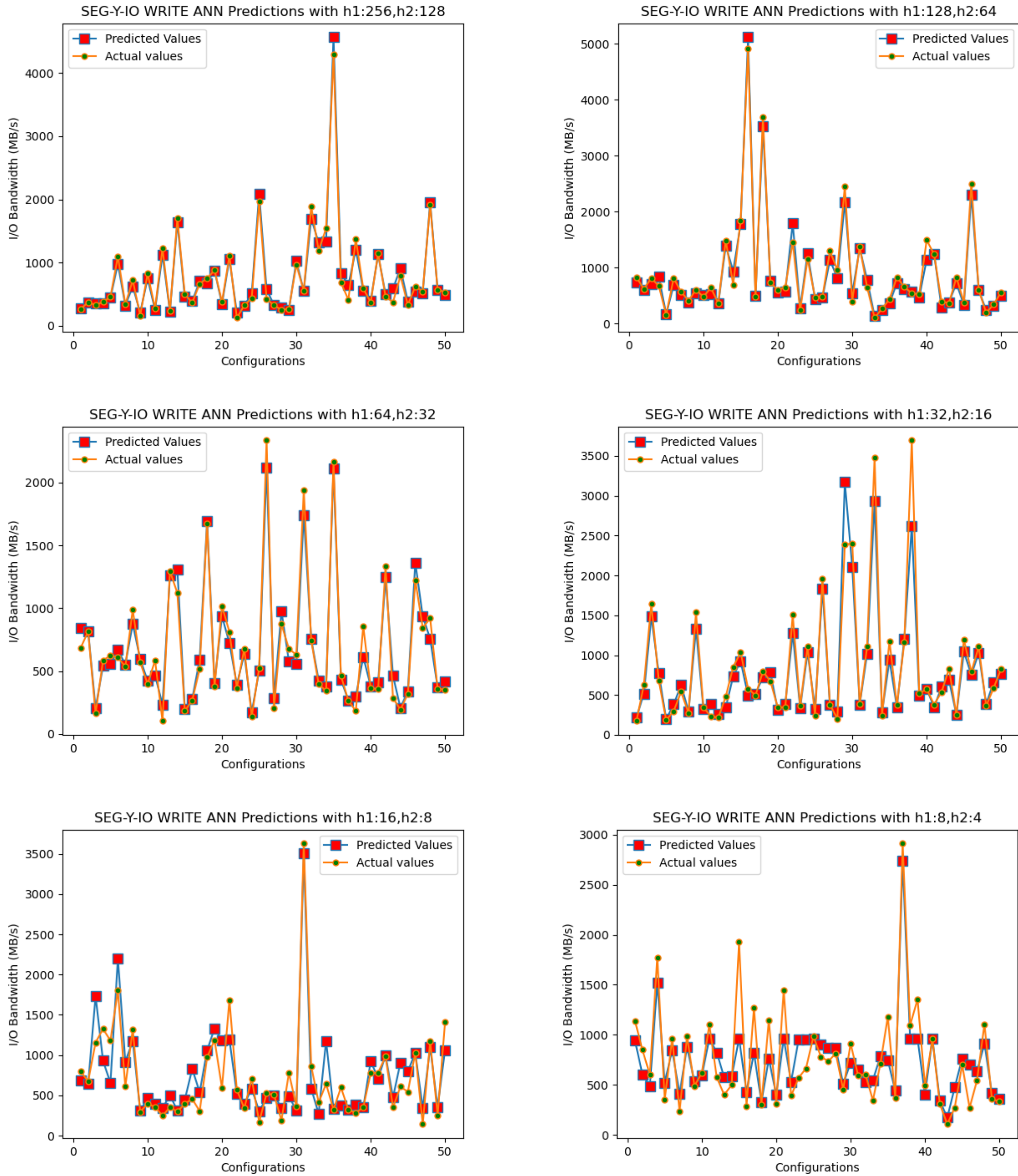


Figure 7.5: SEG-Y IO WRITE Predictions.

with `contiguous` READ operation predictions, where the WRITE operation is `non-contiguous`, it starts with ≈75% with the least nodes configuration (8,4). Then it takes slight increase of 3% to reach almost 78% accuracy when nodes double to (16,8). Afterwards, a slight notable increase to ≈87% and 90% when switch to (32,16) and (64,32) nodes, respectively. Onward, it keeps almost constant accuracy around 90% till (256,128) nodes configuration. It is conveniently visible that any ANN model with equal to or above (64,32) nodes should be ideal as far as just the predictions are concerned. However, model selection can be more specific after analyzing auto-tuning results.

For SEG-Y sorting by `contiguous` WRITE operation predictions for file sorting, where READ unsorted file is `non-contiguous`, the accuracy starts from around 50% as a

minimum value for (8,4) nodes ANN. It makes a significant notable increase in accuracy till (64,32) nodes from almost 70 to 90%. Afterwards, it remain around 90% accuracy value till the last (256,128) configuration of hidden layers nodes. Again, it can be clearly seen that ANN model with equal to or above (64,32) nodes are ideal from only concern of predictions. The specific model selection would be possible after the analysis of auto-tuning SEG-Y file sorting operations.

The SEG-Y sorting by `contiguous` READ operation prediction accuracy ranges from 75.3% to 90.6% whereas, for `contiguous` WRITE operation, it ranges from 50% to 92.2%, as mentioned in Table 7.10. The Figures 7.7 and 7.8 depict the bandwidth predictions against actual values of all the 12 ANN models for SEG-Y file sorting operations. Each graph in figures, represents a specific benchmark and operation type, and hidden layers nodes configuration used in an ANN model. It can be noticed again that predictions of ANNs with fewer hidden layer nodes are less likely following the actual changing bandwidth pattern as compared to the ANNs with higher hidden layer nodes. The reason is again decreased prediction accuracy with fewer hidden layer nodes.

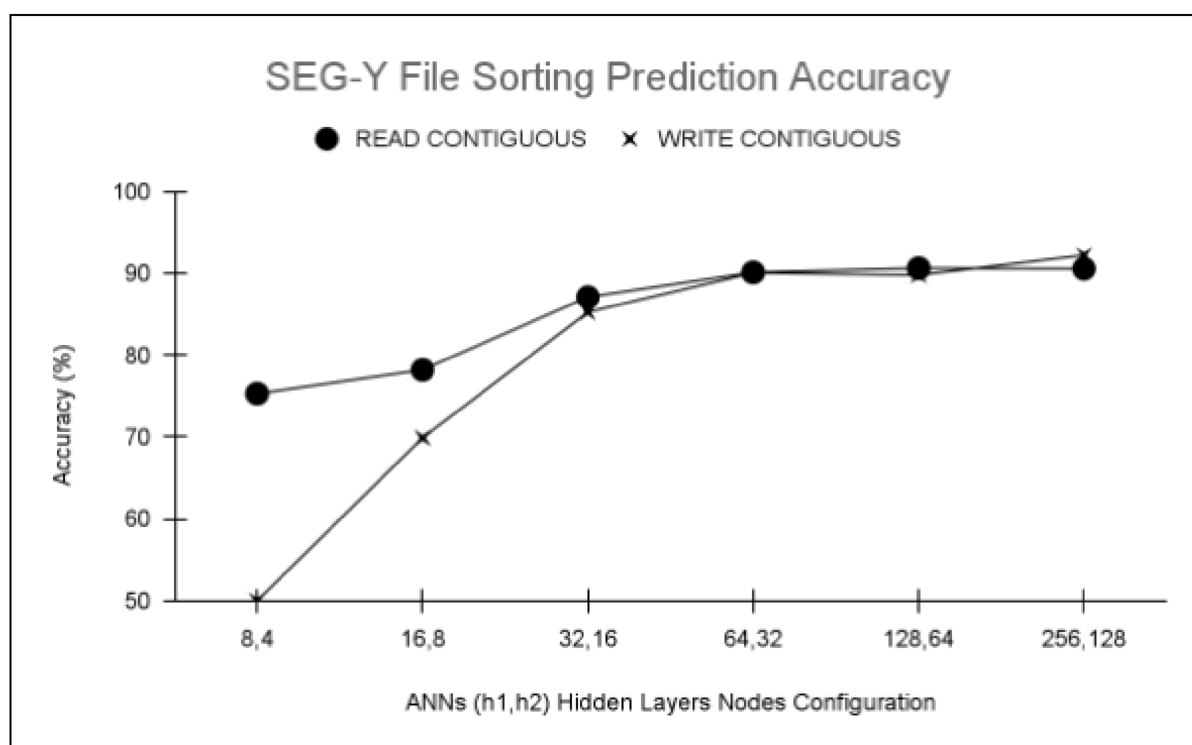


Figure 7.6: SEG-Y file sorting ANNs Prediction Accuracy.

Table 7.10: SEG-Y file sorting Prediction Accuracy values.

SEG-Y Sort Operation	ANNs					
	8,4	16,8	32,16	64,32	128,64	256,128
`contiguous` READ	75.3	78.2	87.1	90.1	90.6	90.5
`contiguous` WRITE	50.0	69.9	85.3	90.1	89.8	92.2

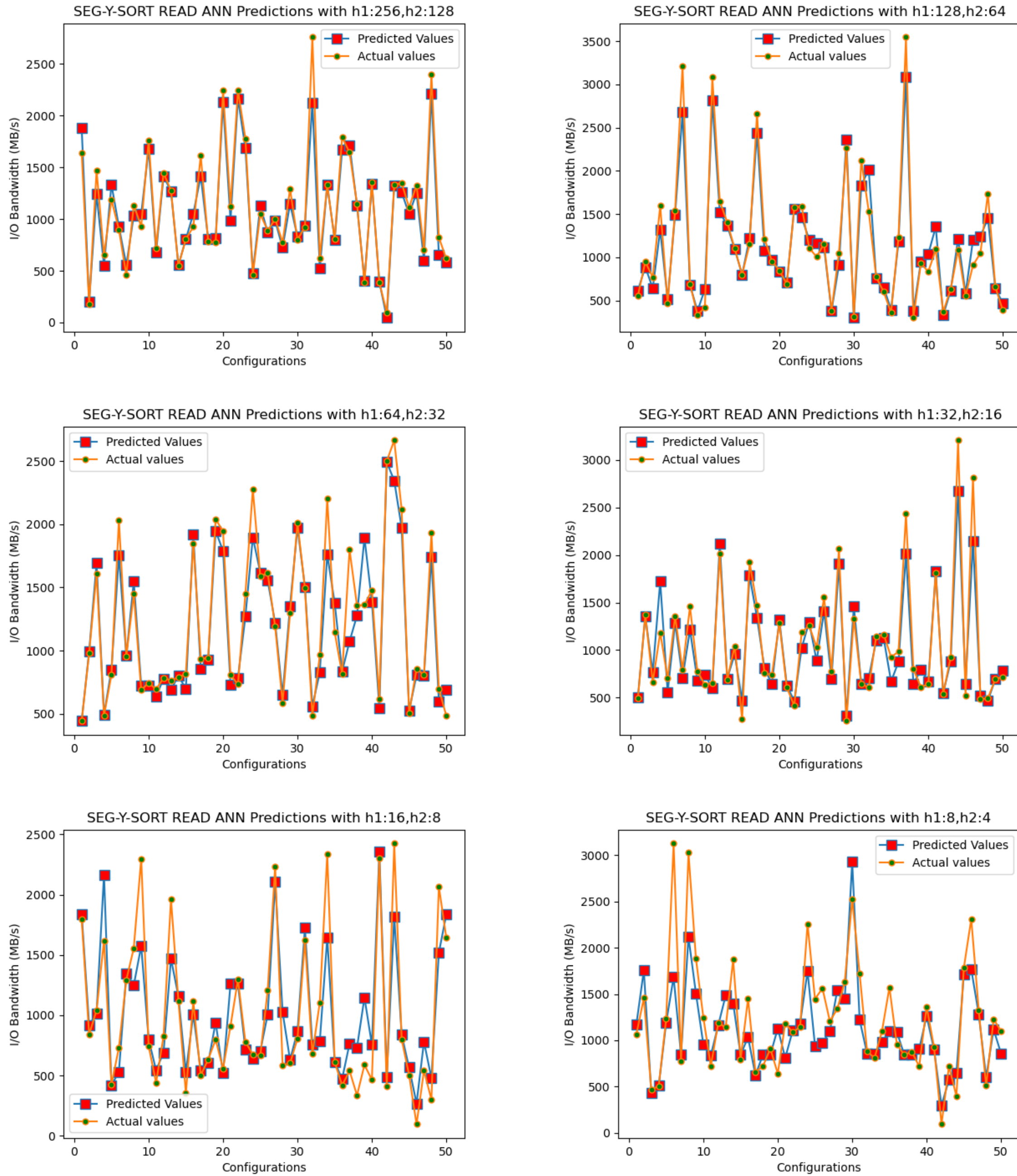


Figure 7.7: SEG-Y Sorting Contiguous READ Predictions.

7.4.1.3 Runtime Cost Analysis for New Configurations selection

As the method in Listing 7.3, runs brute force to check all possible combinations to select new configuration settings for auto-tuning therefore, its runtime cost is also analysed. The Line 29 in Listing 7.3, runs the ANN feed forward propagation pass to predict the value using `model(X)`, over several iterations depending on an operation type. In case of ANNs used in this research, there are three sub-passes involved in a single forward propagation pass: 1) input layer to hidden layer 1, 2) hidden layer 1 to hidden layer 2 and finally 3) hidden layer 2 to output layer. By considering the ANN with maximum hidden layers nodes (256 and 128), the Table 7.12 presents the working of these three sub-passes. The details in the table also contain the number of computations having multiplications

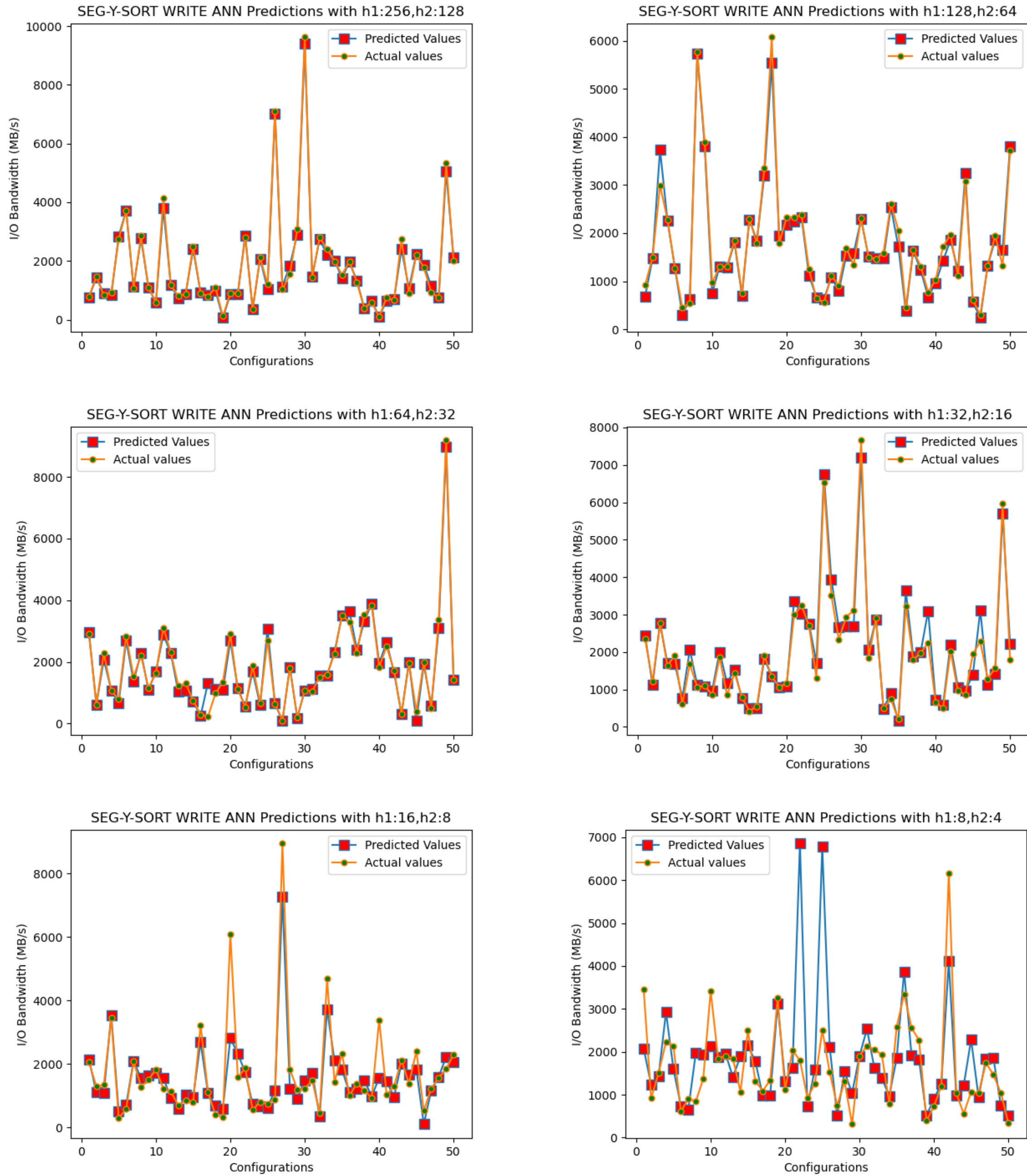


Figure 7.8: SEG-Y Sorting Contiguous WRITE Predictions.

(Mul.) and additions (Add.) involved during a whole single pass. The first sub-pass runs the ≈3584 computations of multiplications and additions overall. Similarly, the second sub-pass runs ≈65536 computations. This is the most expensive sub-pass by memory and CPU consumption wise. The reason is the hidden layers and their weights matrices have the most number of nodes and memory allocation in a single pass, respectively. Finally, the third sub-pass runs 128 multiplications and 128 additions which makes 256 computations in total. By adding all the computations of the sub-passes it makes total of ≈69376 computations in a single ANN's feed forward propagation pass to predict a bandwidth value.

Keeping the computations in view, the new parameters values selection for SEG-Y

Table 7.11: Matrices Detail and Memory Space in a Feed Forward Propagation Pass of an ANN.

Matrix	Description	Memory Usage (bytes)
$I_{7\times1}$	7 configuration values to input layer nodes	$7 \times 1 \times 4 = 28$
$H1_{256\times1}$	256 hidden layer-1 nodes values	$256 \times 1 \times 4 = 1024$
$W1_{256\times7}$	Weights matrix between I and $H1$ layers	$256 \times 7 \times 4 = 7168$
$H2_{128\times1}$	128 hidden layer-2 nodes values	$128 \times 1 \times 4 = 512$
$W2_{128\times256}$	Weights matrix between $H1$ and $H2$ layers	$128 \times 256 \times 4 = 131072$
$O_{1\times1}$	1 output value as predicted IO bandwidth	$1 \times 1 \times 4 = 4$
$W3_{1\times128}$	Weights matrix between $H2$ and O layers	$1 \times 128 \times 4 = 512$
	Memory space used in total is:	140320 bytes ≈ **137 KiB**

Table 7.12: Sub-passes of Feed Forward Propagation Pass for a bandwidth prediction.

Sub-pass#	Equation	Number of Computations
1	$H1_{256\times1} = W1_{256\times7} \times I_{7\times1}$	$(7Mul. + 7Add.) \times 256 = 3584$
2	$H2_{128\times1} = W2_{128\times256} \times H1_{256\times1}$	$(256Mul. + 256Add.) \times 128 = 65536$
3	$O_{1\times1} = W3_{1\times128} \times H2_{128\times1}$	$(128Mul. + 128Add.) \times 1 = 256$
	Total computations are ≈	**69376** per ANNs pass

READ operation, runs a forward propagation pass 2 times. This makes ≈138,752 of the total executing instructions. Whereas, the new parameters selection for SEG-Y WRITE operation, runs 40 (2×4×5) times the execution of feed forward propagation pass. This makes ≈2,775,040 computations in total. In case of SEG-Y file sorting with `contiguous` READ/WRITE, the new parameters values selection for both operations, runs a forward propagation pass 20 (4×5) times. This makes a total of ≈1,387,520 computations.

To analyze how quick the million of computations can CPU run, depends on the usage of memory. The Table 7.11 describes the layout of matrices and their memory usage in a forward propagation pass. The matrices for an ANN model are created with default `float32` data type, which means each location of a matrix contains 32-bit or 4-bytes of floating point number. The total matrices used are 7 in a single pass which can be seen from the Table 7.11. By adding all the bytes of the 7 matrices, it makes ≈140320 bytes or ≈137 KiB of total memory required by an ANN model. If a model would be generated with 64-bit or 8 bytes of double precision decimal memory, then the required memory would be doubled to ≈274 KiB. In either case, this size of memory can be easily fit into the cache RAM. Since the matrices are conveniently cacheable, the execution of millions of instructions takes negligible execution time by $\approx 10e^{-4}$ seconds. This runtime tested several times on a KAY's [51]compute node, includes the loading time of the required libraries. However, the first time program loading can take around 30 seconds. Later on, it runs by less than a second as just mentioned.

In order to keep simplicity and convenience, the code logic for parameters selection in Listing 7.3 is written in python script to execute using PyTorch module. This interfaces with the caller MPI program in C++ to apply the new configuration settings and run the SEG-Y operation, stated in Listing 7.4. Alternatively, the parameters selection could be embedded in a single MPI program with using C++ version of PyTorch library. This can be complex than a python script however, it could be faster due to removal of the first time program loading step.

7.4.2 Auto-tuning Results Analysis

In this section, the series of auto-tuning results of improvements are presented in terms of bandwidth values with the statistical analysis against all the ANNs used. As stated earlier, to choose a specific ANN for a SEG-Y operation optimization, the auto-tuning results must be analyzed, using all the ANNs separately. The Table 7.13 presents the formal definitions of all the statistical metrics used in the experiment, and later being referred in the tables.

The results tables presented in this section are for SEG-Y READ, WRITE and their combined IO. Similarly, the tables are also presented for SEG-Y file sorting by `contiguous` READ, `contiguous` WRITE and their combined sorting bandwidth results. The statistics of combined IO and sorting results are computed by merging the lists of bandwidth values results of their corresponding READ and WRITE operations. This has been carried out against all ANNs used by programming the code logic. However, the combined prediction accuracy READ/WRITE results has been computed by the following equation:

$$Combined\ Accuracy = \frac{A_r + A_w}{2},$$

where A_r and A_w are prediction accuracy percentage values of READ and WRITE ANN models, respectively.

It should be noted that the metrics presenting bandwidth values are in the unit of MiB/s and the multiple of 10^x represented as "ex" in the tables. For example, if the value is 2.54 written in a cell and its corresponding row has a metric represented in the left most column with (e5 MiB/s), this means the value is 2.54×10^5 MiB/s or 2.54e5 MiB/s, which are equivalent to each other.

7.4.2.1 SEG-Y READ Auto-tuning Results

In this section, the auto-tuning results of improvements are presented for SEG-Y READ operation test cases, as mentioned in the Table 7.14. According to the results, the auto-tuning design has resulted in majority of improvements in the test cases. The number of improved test cases ranges from 65.2% to 87.2%. In case of most of the ANNs (out of 6), the improved test cases are above 81%. The overall bandwidth improvement ranges in 35.3 % to 97.3%. The worst case of ANN is with (16,8) hidden layers nodes configuration where the lowest improved test cases and overall bandwidth improvement can be noticed. This is regardless to its significant prediction accuracy. By excluding this case, majority of ANNs show bandwidth improvement above 95% whereas, rest of them has 84.6% and 87.8%. It can also be seen that the overall statistics values of tuned mean, maximum, minimum and median bandwidth values are also significantly greater than their default values. However, the standard deviation and variance values are somehow comparable with most of the ANNs. The auto-tuning of SEG-Y READ operation has significantly optimized the bandwidth performance, is evident from these results. This is further supported by Figure 7.9, which shows the graph plots of auto-tuned bandwidth values against the default values, for each hidden layers (h1,h2) nodes configuration of the ANN models.

It is visible from the Table 7.14 that the improvements do not consistently increase with increasing the number of hidden layers nodes or prediction accuracy. However, the most suitable ANN for the SEG-Y READ operation scenario is apparently the one with (64,32) hidden layers nodes. This ANN has shown maximum test cases improved by

87.2% and optimized bandwidth performance with 97.3%. Therefore, specifically this hidden layers nodes configuration can be chosen to auto-tune and execute SEG-Y READ operations by the system.

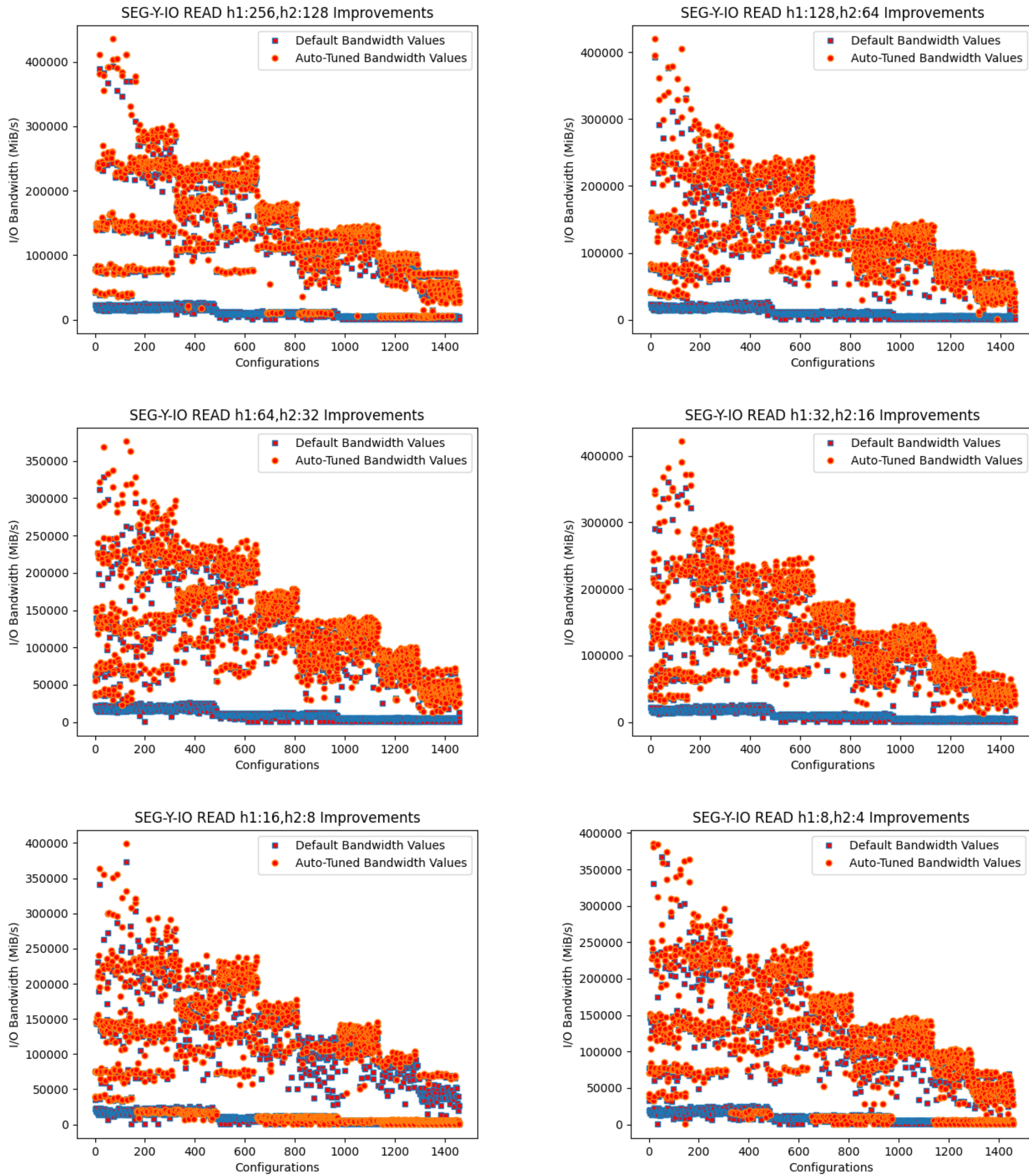


Figure 7.9: SEG-Y IO READ Improvements.

7.4.2.2 SEG-Y WRITE Auto-tuning Results

In this section, the auto-tuning results of improvements are presented for SEG-Y WRITE operation test cases, as mentioned in the Table 7.15. According to the results, the auto-tuning design has resulted in majority of improvements in the test cases. The number of improved test cases ranges from 86.1% to 96.7%. In case of most of the ANNs (out of

6), the improved test cases are above 90%. The overall bandwidth improvement ranges in 475.5% to 602.6%, which indicates very significant optimization for SEG-Y WRITE operation. The worst case of ANN is with (128,64) hidden layers nodes configuration where the lowest improved test cases and overall bandwidth improvement can be noticed. This is regardless to its significant second highest prediction accuracy in all ANNs configruation. By excluding this case, majority of ANNs show bandwidth improvement above 580% whereas, rest of them has 556.0% and 558.2%. It can also be seen that the overall statistics of tuned mean, maximum, median, standard deviation and variance in bandwidth values are also significantly greater than their default values. However, the minimum bandwidth values are somehow comparable with most of the ANNs. The auto-tuning of SEG-Y WRITE operation has shown very significant increase in bandwidth performance by optimization, is evident from these results. This is further supported by Figure 7.10, which shows the graph plots of auto-tuned bandwidth values against the default values, for each hidden layers (h1,h2) nodes configuration used in ANN models.

It is visible from the Table 7.15 that the improvements do not consistently increase with increasing the number of hidden layers nodes or prediction accuracy. However, the most suitable ANN for SEG-Y WRITE operation scenario is apparently the one with (256,128) hidden layers nodes. This ANN has maximum nodes in this scenario. It has shown maximum test cases improved by 96.7% and optimized bandwidth performance with 602.6%. Therefore, specifically this hidden layers nodes configuration can be chosen to auto-tune and execute SEG-Y WRITE operations by the system. The second ideal case can be an ANN with (8,4) hidden layers nodes. It shows closely comparable bandwidth improvement by 602.4%, despite having lowest prediction accuracy and hidden nodes.

7.4.2.3 Combined SEG-Y IO Auto-tuning Results

In this section, the combined auto-tuning results of improvements are presented for SEG-Y IO operation test cases. In the HPC cluster systems, the applications are use to run both READ and WRITE operations in sequence or parallel. Therefore, it is worth noting the combined READ/WRITE improvements, as mentioned in the Table 7.16. According to these results, first of all, the combined prediction accuracy has been increased by increasing the hidden layers nodes in the ANN model. This ranges from 52.5% to 92.5%. The improved test cases do not consistently increase but yield percentage from 79.7% to 91.7%, where (256,128) nodes show maximum improvements. In most of ANNs, the improved test cases are above 85%. However, the overall bandwidth improvement ranges from 50.9% to 108.8%, which is significant, and (64,32) nodes have shown maximum percentage of improvement. The worst case of ANN is with (16,8) hidden layers nodes configuration where the lowest improved test cases and overall bandwidth improvement can be noticed. This is regardless to its significant combined prediction accuracy. By excluding this case, all other ANNs have shown bandwidth improvement above 99.0%. It can also be seen that the overall statistics of tuned mean, maximum, median, standard deviation and variance in bandwidth values are also significantly greater than their default values. However, the minimum bandwidth values are somehow comparable in most of the ANNs. The combined auto-tuning results indicate that the SEG-Y IO operations has been significantly optimized in bandwidth performance, is clearly evident.

To choose the most suitable ANN for the series of SEG-Y IO READ/WRITE operations in the HPC cluster, the one with (64,32) hidden layers nodes, is apparently the best. This is because it has shown the maximum overall SEG-Y IO bandwidth improve-

ment with 108.8%. Therefore, specifically this hidden layers nodes configuration can be chosen to auto-tune and execute the series of SEG-Y IO operations in an application by the system. The second ideal case can be an ANN with (32,16) hidden layers nodes, as it shows closely comparable bandwidth improvement of 108.7%, where the percentage of improved cases is a bit greater.

7.4.2.4 SEG-Y Sorting via `contiguous` READ Auto-tuning Results

In this section, the auto-tuning results of improvements are presented for SEG-Y file sorting via `contiguous` READ operation test cases, as mentioned in the Table 7.17. According to the results, the auto-tuning design has resulted in majority of improvements in the test cases. The number of improved test cases ranges from 74.0% to 91.7%. In case of most of the ANNs (out of 6), the improved test cases are above 76%. The overall bandwidth improvement ranges in 130.9% to 283.9%, which indicates very significant optimization in this SEG-Y sorting operation. The worst case of ANN is with (8,4) hidden layers nodes configuration where the lowest improved test cases and overall bandwidth improvement can be noticed. By excluding this case, majority of ANNs show bandwidth improvement above 158% whereas, rest of them has 141.6% and 156.6%. It can also be seen that the overall statistics of tuned mean, maximum, median, standard deviation and variance in bandwidth values are also significantly greater than their default values. However, the minimum bandwidth values are somehow lesser in all the ANNs cases. The auto-tuning of SEG-Y sorting with `contiguous` READ operation has shown very significant increase in bandwidth performance by optimization, is evident from these results. This is further supported by Figure 7.11, which shows the graph plots of auto-tuned bandwidth values against the default values, for each hidden layers (h1,h2) nodes configuration used in ANN models.

It is visible from the Table 7.17 that the improvements do not consistently increase with increasing the number of hidden layers nodes or prediction accuracy. However, the most suitable ANN for SEG-Y WRITE operation scenario is apparently the one with maximum hidden layers nodes (256,128). This ANN has shown maximum test cases improved by 91.7% and optimized bandwidth performance with 283.9%. Therefore, specifically this hidden layers nodes configuration can be chosen to auto-tune and execute SEG-Y sorting via `contiguous` READ operations by the system.

7.4.2.5 SEG-Y file sorting via `contiguous` WRITE Auto-tuning Results

In this section, the auto-tuning results of improvements are presented for SEG-Y file sorting with `contiguous` WRITE operation test cases, as mentioned in the Table 7.18. According to the results, the auto-tuning design has resulted in majority of improvements in the test cases. The number of improved test cases ranges from 67.2% to 92.2%. In case of most of the ANNs (out of 6), the improved test cases are above 81%. The overall bandwidth improvement ranges in 76.3% to 221.8%, which indicates very significant optimization for SEG-Y sorting with `contiguous` WRITE operation. Surprisingly, a model with lowest hidden layers nodes of (8,4) has shown maximum improved test cases about 92.2 %, as compare to other models. The worst case of ANN is with (64,32) hidden layers nodes configuration where the lowest improved test cases and overall bandwidth improvement can be noticed. This is regardless to its significant prediction accuracy. By excluding this case, majority of ANNs show bandwidth improvement above 213%. It can

also be seen that the overall statistics of tuned mean, maximum, median, standard deviation and variance in bandwidth values are also significantly greater than their default values. However, the minimum bandwidth values are somehow comparable. The auto-tuning of SEG-Y file sorting via `contiguous` WRITE operation has shown very significant increase in bandwidth performance by optimization, is evident from these results. This is further supported by Figure 7.12, which shows the graph plots of auto-tuned bandwidth values against the default values, for each hidden layers (h1,h2) nodes configuration used in ANN models.

It is visible from the Table 7.18 that the improvements do not consistently increase with increasing the number of hidden layers nodes or prediction accuracy. However, the most suitable ANN for SEG-Y sorting with `contiguous` WRITE operation scenario is apparently the one with maximum (256,128) hidden layers nodes. This ANN has shown second highest test cases improved by 91.4% but maximum optimized bandwidth performance of 221.8%. Therefore, specifically this hidden layers nodes configuration can be chosen to auto-tune and execute SEG-Y file sorting via `contiguous` WRITE operations by the system.

7.4.2.6 Combined SEG-Y file sorting Auto-tuning Results

In this section, the combined auto-tuning results of improvements are presented for SEG-Y file sorting operations test cases. The user of ExSeisDat library can run series of SEG-Y file sorting with both `contiguous` READ and `contiguous` WRITE in sequence or parallel, on HPC cluster system. Therefore, it is again worth noting the combined file sorting improvements, as mentioned in the Table 7.19. According to these results, first of all, the combined prediction accuracy has been increased by increasing the hidden layers nodes in the ANN model. This ranges from 62.7% to 91.4%. The improved test cases do not consistently increase but yield percentage from 71.9% to 91.6% where (256,128) nodes show maximum improvements. In most of ANNs, the improved test cases are above 81%. However, the overall bandwidth improvement ranges from 98.4% to 237.4%, which is very significant, and (256,128) nodes have shown maximum bandwidth improvement. The worst case of ANN is with (64,32) hidden layers nodes configuration where the lowest improved test cases and overall bandwidth improvement can be noticed. This is regardless to its significant combined prediction accuracy. By excluding this case, all other ANNs have shown bandwidth improvement above 196.0%. It can also be seen that the overall statistics of tuned mean, maximum, median, standard deviation and variance in bandwidth values are also significantly greater than their default values. However, the minimum bandwidth values are somehow lesser in all the ANNs cases. The combined auto-tuning results indicate that the SEG-Y file sorting operations has been significantly optimized in bandwidth performance, is clearly evident.

To choose the most suitable ANN for the series of SEG-Y file sorting operations in the HPC cluster, the one with maximum (256,128) hidden layers nodes, is apparently the best. This is because it has shown the maximum overall SEG-Y IO bandwidth improvement of 237.4% and test cases improved by 91.6%. Therefore, specifically this hidden layers nodes configuration can be chosen to auto-tune and execute the series of SEG-Y IO operations in an application by the system.

7.5 Discussion

Summarizing the work done in this research, the results presented in the section earlier, are the prediction accuracy evaluation to the auto-tuning analysis. This has been carried out for all individual SEG-Y IO and file sorting types of operations in ExSeisDat on HPC cluster, subsequently, leading to combined statistics. The numerics show the impact of variation in ANNs on both prediction accuracy and auto-tuning results. This leverages to choose a specific ANNs hidden layers nodes configuration for having the maximum possible bandwidth performance of SEG-Y data processing. The auto-tuning involves certain parameters responsible for predicting the bandwidth performance. Nevertheless, the tunable parameters can be only tempered at the runtime, depending on the type of SEG-Y data operation executing i.e. the file access pattern, file striping configurations, etc. Consequently, the certain tunable parameters are identified for a type of SEG-Y operation and value settings separately.

As per past research, the performance prediction and auto-tuning over various parameters have been the key elements in research to improve IO [14, 15, 16, 17]. Since, some parameters can be tuned prior to IO execution, the predictive modelling for SEG-Y data processing has been one of the crucial requirement in this scenario, towards auto-tuning design. Consequently, the best ML approach towards predictive modelling have been ANNs for its significant forecasting capability as per couple of researches [23, 19, 21, 24].

Previously in chapter 5 of the thesis, the main parameters in regards to MPI-IO, LFS and SEG-Y file format were identified to generate benchmarks bandwidth profiling data. This data was later used for the predictive modelling of ANNs with (256,128) hidden layers nodes. However, this research has conducted re-execution of benchmarks profiling on the identified tunable and non-tunable configurations in Table 7.1 and 7.2. Subsequently, the ANNs mentioned in Table 7.4 have been trained with various less hidden layers node configurations. This is to predict bandwidth for all the SEG-Y IO and file sorting operations, using all these ANNs.

The ANNs have different prediction accuracy values for SEG-Y bandwidth data, depending on the ANNs hidden layers nodes (h1,h2) configuration. Then the constructed design of auto-tuning strategy, using the ANNs predictions, is applied on the default test case settings of the SEG-Y operations, as mentioned in Table 7.7 and 7.8. This has been done to get the statistical data of bandwidth performance over default settings and the corresponding auto-tuned settings, using all the ANNs. Subsequently, to observe the the impact of (h1,h2) configuration on the improvements in bandwidth performance. The steps to collect the statistical data is comprised of three main functionalities, namely: `New_Configs()`, `AutoTuneAndRunSegYOp()` and `Statistics_Collection_Process()`. They are indicated in the Figure 7.1 as part of the flow of this research, and also elaborated by Listings 7.3, 7.4 and 7.5.

In the experiment results, first the impact of (h1,h2) nodes configuration on prediction accuracy is discussed for both SEG-Y IO and file sorting operations ANNs, as shown in Figure 7.3 and 7.6. It can be examine that prediction accuracy notably increases from (8,4) to (32,16) nodes configuration. Afterwards, it is either very gradually increasing or almost constant till (256,128) nodes configuration. This is generally the pattern for both SEG-Y operations ANNs predictive modelling. Afterwards, it is followed by the runtime cost analysis of the bandwidth predicition by ANN, which is computed by the feed forward propagation pass. It is observed that the parameters selection using `New_Configs()` method, mostly takes negligible time in the unit of 10^{-4} seconds.

Onward, the improvements in the overall bandwidth performance are analyzed, which occurred during auto-tuning process of all SEG-Y operations. The SEG-Y READ, WRITE and combined IO auto-tuning yield to maximum 97.3%, 602.6% and 108.8% of the overall bandwidth performance improvement, respectively. Similarly, SEG-Y file sorting via `contiguous` READ, `contiguous` WRITE, and their combined sorting results, yield to maximum 283.9%, 221.8% and 237.8% of improvements in the overall bandwidth performance. In addition to this, it is also discussed that which ANN can be appropriate from case to case in regard to SEG-Y operation type. In general, it has been demonstrated in this research, the advantage of adaptability over particular configuration settings at the runtime on the basis of ML predictions.

7.6 Conclusion

This research examined the improvements of auto-tuning over SEG-Y IO and file sorting operations for ExSeisDat, based on the IO bandwidth predictions by ANNs. It has been demonstrated that the adaptable and efficient IO optimization technique as the key requirement, that extends the work presented in chapter 5. The approach presented aligns with recent research works emphasizing on the challenges in area of overcoming poor IO performance in HPC. The recent works are mostly carried out via ML prediction over certain parameters, subsequently auto-tuning them in different scenarios [14, 15, 16, 17, 193, 18, 24, 21, 82, 83]. Consequently, the indicated three key contributions in this research are: 1) the SEG-Y IO and file sorting operations auto-tuning design logic based on the ANNs bandwidth performance prediction, 2) the impact on prediction accuracy by changing hidden layers nodes configuration, and 3) statistical analysis of default and auto-tuned configuration test settings, over specified ANNs. By witnessing the results, the maximum overall bandwidth improved by 97.3%, 602.6% and 108.8% for SEG-Y READ, WRITE and combined, respectively. For SEG-Y file sorting by contiguous READ, WRITE and combined sorting results, the overall bandwidth performance improved by 283.9%, 221.8% and 237.8%, respectively. As part of this research, it has been demonstrated that the overall advantage of adaptive technique for SEG-Y operations optimization based on the ANNs prediction. As evident from the results, these contributions have meaningful benefits in terms of SEG-Y (Seismic) Data IO and Sorting optimization. Thus, paving the way for the efficient IO throughput for an ExSeisDat library based application at runtime within an HPC cluster.

However, this work can be further extended by applying reinforcement learning over the ANNs and auto-tuning design, which is not the part of this research. This can enable the optimization of the SEG-Y operations at runtime without requiring the prior large time consuming steps of the benchmarks profiling and ML model training process. Furthermore, the different performance measures can be analyzed with varying hidden layers nodes configuration of the ANNs.

Table 7.13: Description of Statistical Metrics used.

Metric	Explanation
Accuracy(%)	Prediction accuracy of an ANN model with specific hidden layers nodes (h1,h2), in terms of percentage.
Improvements	Number of improved test cases when auto-tuning is applied to default test settings.
Improvements(%)	Percentage of improved test cases when auto-tuning is applied to default test settings.
Default Mean	Mean bandwidth value of default configuration test settings execution.
Tuned Mean	Mean bandwidth value of auto-tuned configuration settings execution.
Default Max.	Maximum bandwidth value captured during the execution of default configuration test settings.
Tuned Max.	Maximum bandwidth value captured during the execution of auto-tuned configuration test settings.
Default Min.	Minimum bandwidth value captured during the execution of default configuration test settings.
Tuned Min.	Minimum bandwidth value captured during the execution of auto-tuned configuration test settings.
Default Med.	Median bandwidth value in all the default test cases.
Tuned Med.	Median bandwidth value in all the auto-tuned test cases.
Default Std. dev.	Standard Deviation of bandwidth values from the mean value, in the default test cases.
Tuned Std. dev.	Standard Deviation of bandwidth values from the mean value, in the auto-tuned test cases.
Default Variance	Variance of bandwidth values in the default test cases.
Tuned Variance	Variance of bandwidth values in the auto-tuned test cases.
Bandwidth Improvement(%)	Overall percentage bandwidth improvement in Tuned Mean bandwidth value from the Default Mean bandwidth value.

Table 7.14: SEG-Y READ Improvements.

Metrics	ANNs					
	8,4	16,8	32,16	64,32	128,64	256,128
Accuracy(%)	39.5	86.5	93.5	95.4	94.7	95.9
Improvements	1194.0	950.0	1255.0	1271.0	1229.0	1265.0
Improvements(%)	81.9	65.2	86.1	87.2	84.3	86.8
Default Mean (e4 MiB/s)	6.55	6.51	6.70	6.58	6.67	7.20
Tuned Mean (e5 MiB/s)	1.21	0.88	1.31	1.30	1.30	1.35
Default Max. (e5 MiB/s)	3.67	3.74	3.61	3.28	3.92	3.94
Tuned Max. (e5 MiB/s)	3.85	3.99	4.22	3.76	4.20	4.35
Default Min. (e2 MiB/s)	1.26	2.86	2.30	2.28	2.54	2.81
Tuned Min. (e2 MiB/s)	2.07	3.17	136.0	131.0	7.90	40.1
Default Med. (e4 MiB/s)	2.47	2.42	2.54	2.49	2.58	2.67
Tuned Med. (e5 MiB/s)	1.19	0.75	1.25	1.22	1.24	1.34
Default Std. dev. (e4 MiB/s)	7.03	6.91	7.26	7.04	7.10	7.70
Tuned Std. dev. (e4 MiB/s)	7.37	8.39	6.74	6.60	6.56	7.67
Default Variance (e9)	4.94	4.78	5.27	4.96	5.04	5.93
Tuned Variance (e9)	5.43	7.04	4.54	4.35	4.31	5.88
Bandwidth Improvement(%)	84.6	35.3	96.0	97.3	95.2	87.8

Table 7.15: SEG-Y WRITE Improvements.

Metrics	ANNs					
	8,4	16,8	32,16	64,32	128,64	256,128
Accuracy(%)	65.6	68.3	84.0	87.1	88.5	89.1
Improvements	1371.0	1373.0	1353.0	1298.0	1256.0	1410.0
Improvements(%)	94.0	94.2	92.8	89.0	86.1	96.7
Default Mean (e3 MiB/s)	1.92	1.92	1.89	1.70	1.71	2.22
Tuned Mean (e4 MiB/s)	1.35	1.31	1.24	1.12	0.98	1.56
Default Max. (e3 MiB/s)	5.76	6.02	5.71	5.76	6.10	7.04
Tuned Max. (e4 MiB/s)	3.58	3.55	3.59	3.63	3.55	3.71
Default Min. (e2 MiB/s)	2.30	2.66	3.00	0.88	1.79	3.77
Tuned Min. (e2 MiB/s)	1.06	1.92	0.72	1.31	0.97	5.72
Default Med. (e3 MiB/s)	1.33	1.32	1.31	1.18	1.18	1.46
Tuned Med. (e4 MiB/s)	1.15	1.17	1.11	0.92	0.75	1.55
Default Std. dev. (e3 MiB/s)	1.28	1.30	1.27	1.21	1.24	1.52
Tuned Std. dev. (e3 MiB/s)	9.27	9.07	8.63	9.09	9.22	9.91
Default Variance (e6)	1.65	1.70	1.62	1.45	1.54	2.32
Tuned Variance (e7)	8.60	8.22	7.44	8.26	8.51	9.82
Bandwidth Improvement(%)	602.4	581.3	558.2	556.0	475.5	602.6

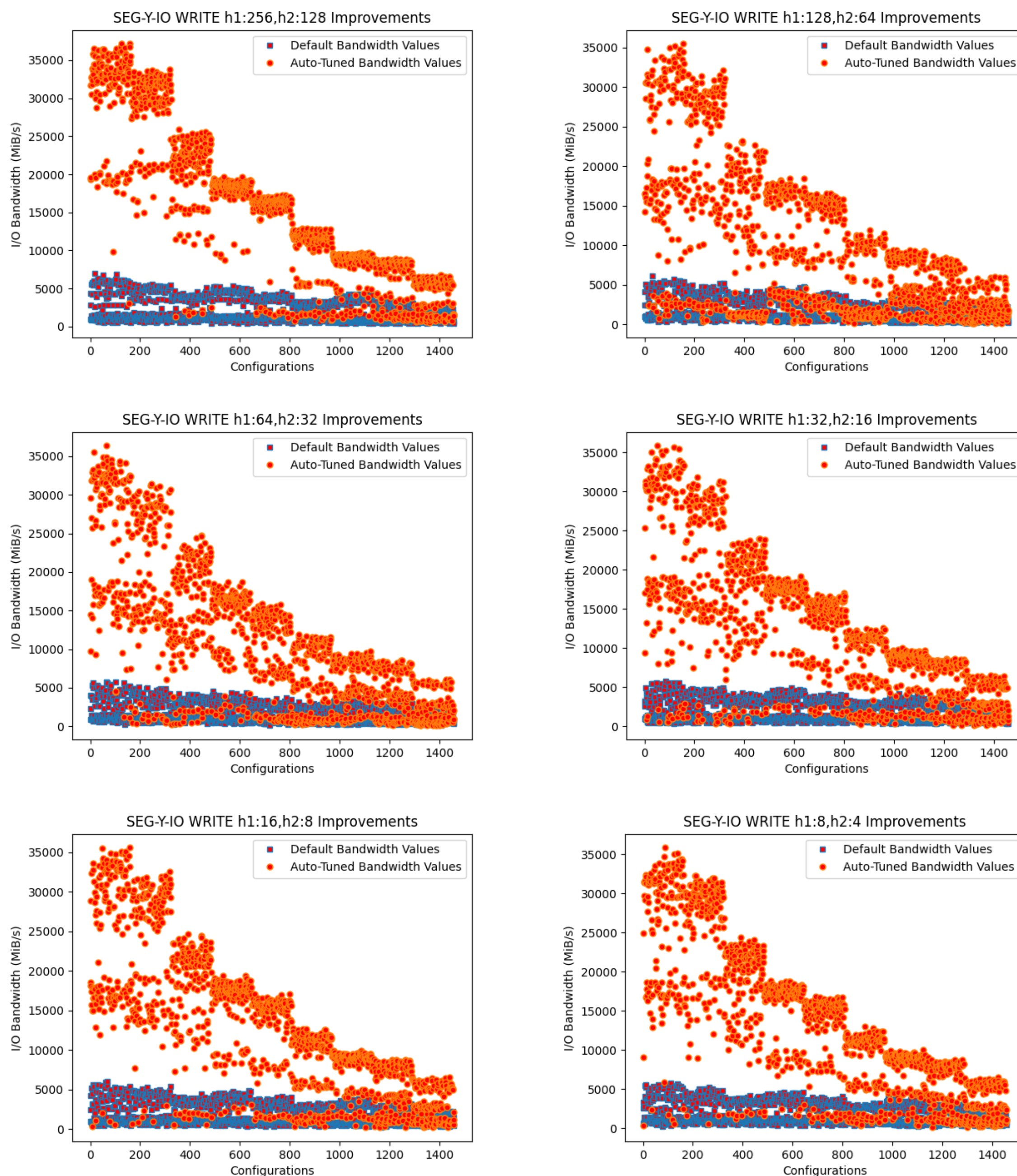


Figure 7.10: SEG-Y IO WRITE Improvements.

Table 7.16: Combined SEG-Y IO Improvements.

Metrics	ANNs					
	8,4	16,8	32,16	64,32	128,64	256,128
Accuracy(%)	52.5	77.4	88.7	91.2	91.6	92.5
Improvements	2565.0	2323.0	2608.0	2569.0	2485.0	2675.0
Improvements(%)	88.0	79.7	89.4	88.1	85.2	91.7
Default Mean (e4 MiB/s)	3.37	3.35	3.44	3.37	3.42	3.71
Tuned Mean (e4 MiB/s)	6.72	5.06	7.18	7.05	7.00	7.54
Default Max. (e5 MiB/s)	3.67	3.74	3.61	3.28	3.92	3.94
Tuned Max. (e5 MiB/s)	3.85	3.99	4.22	3.76	4.20	4.35
Default Min. (e2 MiB/s)	1.26	2.66	2.30	0.88	1.79	2.81
Tuned Min. (e2 MiB/s)	1.06	1.92	0.72	1.31	0.97	5.72
Default Med. (e3 MiB/s)	4.24	4.30	4.19	3.96	4.11	4.80
Tuned Med. (e4 MiB/s)	2.76	1.57	3.22	3.22	3.19	3.23
Default Std. dev. (e4 MiB/s)	5.90	5.82	6.08	5.92	5.98	6.47
Tuned Std. dev. (e4 MiB/s)	7.51	7.05	7.64	7.57	7.63	8.10
Default Variance (e9)	3.48	3.39	3.70	3.51	3.58	4.18
Tuned Variance (e9)	5.64	4.96	5.84	5.74	5.81	6.56
Bandwidth Improvement(%)	99.3	50.9	108.7	108.8	104.7	103.2

Table 7.17: SEG-Y file sorting with `contiguous` READ Improvements.

Metrics	ANNs					
	8,4	16,8	32,16	64,32	128,64	256,128
Accuracy(%)	75.3	78.2	87.1	90.1	90.6	90.5
Improvements	1618.0	1712.0	1783.0	1677.0	1667.0	2005.0
Improvements(%)	74.0	78.3	81.5	76.7	76.2	91.7
Default Mean (e3 MiB/s)	1.02	1.02	1.01	0.99	0.97	1.22
Tuned Mean (e3 MiB/s)	2.36	2.63	2.60	2.75	2.35	4.69
Default Max. (e3 MiB/s)	4.47	4.35	4.37	3.96	4.25	4.81
Tuned Max. (e4 MiB/s)	2.00	1.95	2.17	1.97	1.99	2.28
Default Min. (e2 MiB/s)	1.17	2.25	2.23	1.21	1.33	3.01
Tuned Min. (e1 MiB/s)	9.21	5.55	7.00	7.22	5.54	5.40
Default Med. (e2 MiB/s)	7.30	7.40	7.30	7.29	7.22	8.08
Tuned Med. (e3 MiB/s)	1.20	1.46	1.46	1.51	1.31	3.31
Default Std. dev.(e2 MiB/s)	7.17	6.73	6.88	6.56	6.56	8.48
Tuned Std. dev.(e3 MiB/s)	2.61	2.70	2.65	2.88	2.52	4.19
Default Variance (e5)	5.14	4.52	4.73	4.31	4.30	7.19
Tuned Variance (e6)	6.81	7.30	7.04	8.28	6.35	17.6
Bandwidth Improvement(%)	130.9	158.2	156.6	176.8	141.6	283.9

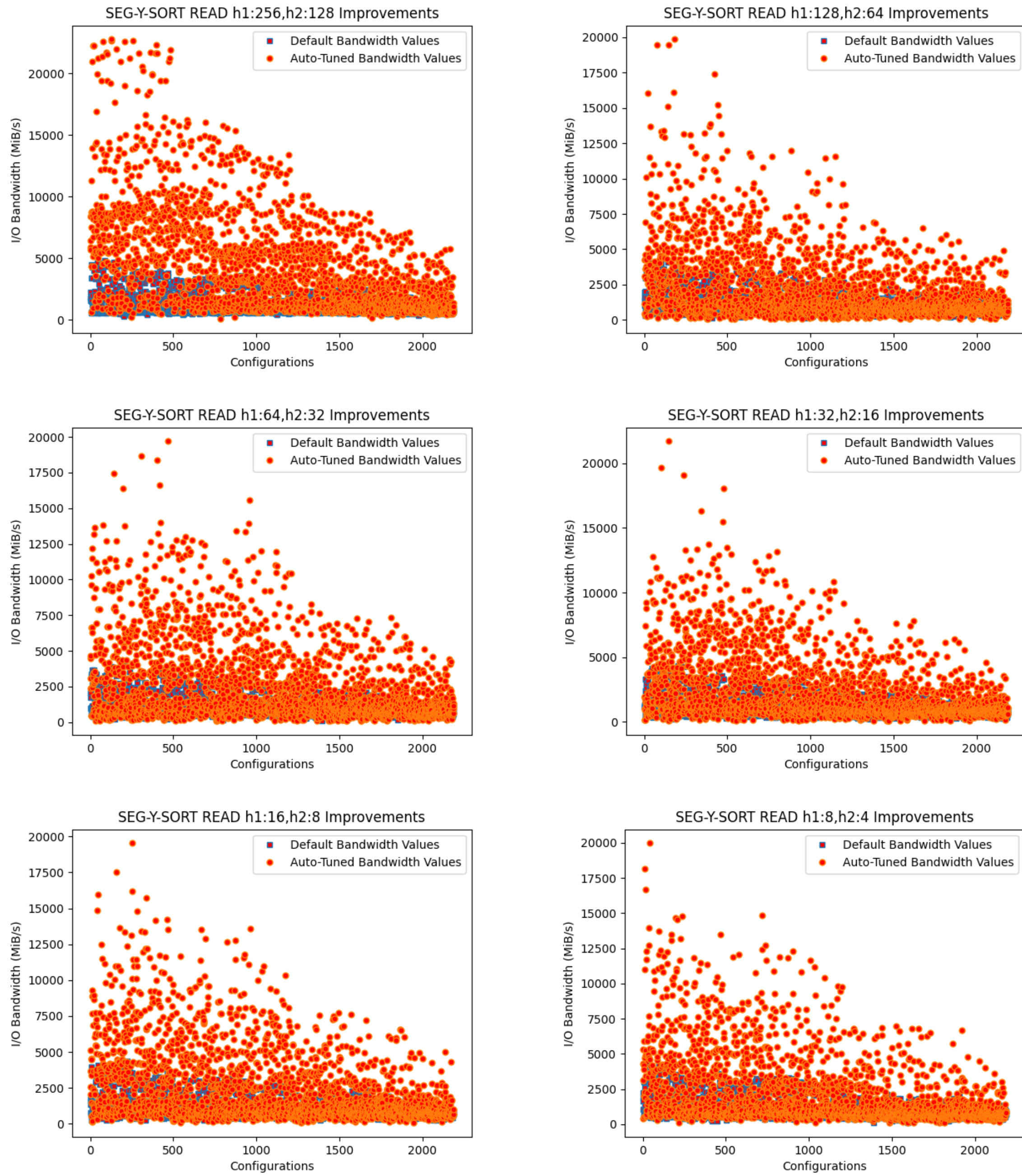


Figure 7.11: SEG-Y file sorting Improvements via `contiguous` READ.

Table 7.18: SEG-Y file sorting with `contiguous` WRITE Improvements.

Metrics	ANNs					
	8,4	16,8	32,16	64,32	128,64	256,128
Accuracy(%)	50.0	69.9	85.3	90.1	89.8	92.2
Improvements	2016.0	1909.0	1790.0	1469.0	1892.0	2000.0
Improvements(%)	92.2	87.3	81.8	67.2	86.5	91.4
Default Mean (e3 MiB/s)	3.46	3.42	3.18	3.53	3.14	3.65
Tuned Mean (e4 MiB/s)	1.10	1.08	1.02	0.62	0.98	1.17
Default Max. (e3 MiB/s)	8.92	9.08	8.74	9.55	9.15	9.59
Tuned Max. (e4 MiB/s)	4.18	4.22	4.11	4.12	4.14	4.28
Default Min. (e2 MiB/s)	4.51	4.71	2.74	6.50	4.73	5.83
Tuned Min. (e2 MiB/s)	4.15	2.40	2.57	6.05	2.50	4.25
Default Med. (e3 MiB/s)	3.06	3.03	2.73	3.06	2.60	3.37
Tuned Med. (e3 MiB/s)	9.57	9.57	8.39	3.72	8.43	10.1
Default Std. dev. (e3 MiB/s)	1.90	1.87	1.89	2.00	1.87	2.00
Tuned Std. dev. (e3 MiB/s)	8.36	8.57	8.94	6.69	8.22	9.05
Default Variance (e6)	3.59	3.51	3.57	3.98	3.48	4.01
Tuned Variance (e7)	6.99	7.35	7.99	4.48	6.75	8.18
Bandwidth Improvement(%)	216.8	217.3	219.7	76.3	213.7	221.8

Table 7.19: Combined SEG-Y file sorting Improvements.

Metrics	ANNs					
	8,4	16,8	32,16	64,32	128,64	256,128
Accuracy(%)	62.7	74.1	86.2	90.1	90.2	91.4
Improvements	3634.0	3621.0	3573.0	3146.0	3559.0	4005.0
Improvements(%)	83.1	82.8	81.7	71.9	81.4	91.6
Default Mean (e3 MiB/s)	2.24	2.22	2.10	2.26	2.06	2.43
Tuned Mean (e3 MiB/s)	6.66	6.74	6.38	4.49	6.10	8.21
Default Max. (e3 MiB/s)	8.92	9.08	8.74	9.55	9.15	9.59
Tuned Max. (e4 MiB/s)	4.18	4.22	4.11	4.12	4.14	4.28
Default Min. (e2 MiB/s)	1.17	2.25	2.23	1.21	1.33	3.01
Tuned Min. (e1 MiB/s)	9.21	5.55	7.00	7.22	5.54	5.40
Default Med. (e3 MiB/s)	1.49	1.49	1.39	1.49	1.36	1.65
Tuned Med. (e3 MiB/s)	3.62	3.83	3.11	2.43	3.10	5.86
Default Std. dev. (e3 MiB/s)	1.88	1.85	1.79	1.95	1.77	1.96
Tuned Std. dev. (e3 MiB/s)	7.54	7.57	7.60	5.44	7.14	7.88
Default Variance (e6)	3.54	3.42	3.20	3.82	3.13	3.84
Tuned Variance (e7)	5.68	5.73	5.78	2.96	5.10	6.21
Bandwidth Improvement(%)	197.2	203.7	204.5	98.4	196.6	237.4

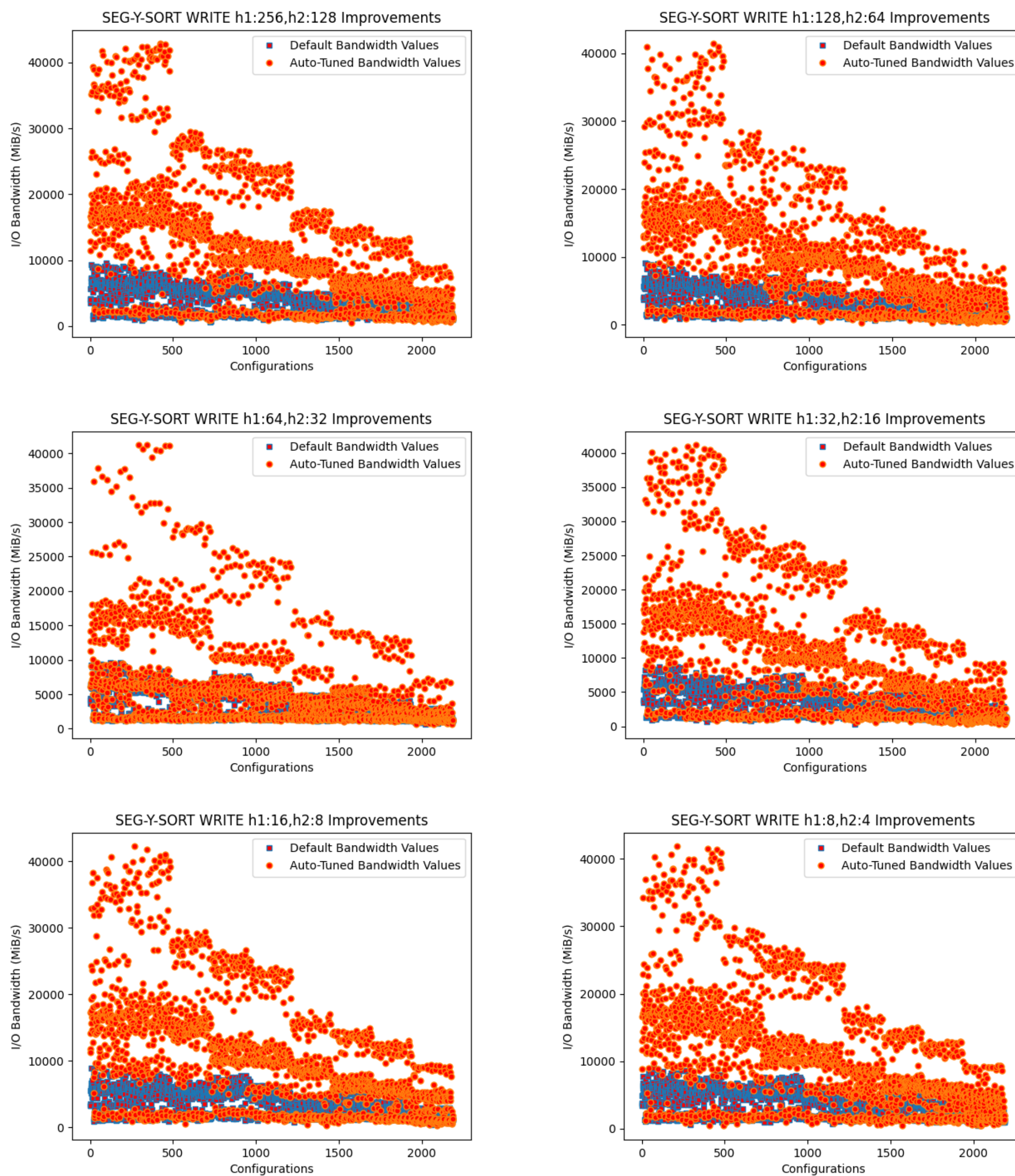


Figure 7.12: SEG-Y file sorting Improvements via `contiguous` WRITE.

Chapter 8

Summary of Thesis

In this research, the machine learning approach based on ANNs modelling is used to overcome poor parallel IO performance in the High Performance Computing cluster. Mainly, the parallel IO data bandwidth is optimized by auto-tuning the parallel lustre file system parameters with the given set of the Standard MPI IO parameters [8, 13, 7]. This is then extended to the application of seismic data processing with SEG-Y format which is the ExSeisDat library [4, 1].

Initially, background research is reviewed to motivate the problem and identify the research questions. For this purpose, the related research has been studied from three perspectives. The first is the background research reviewed in chapter 2 of the thesis, is in relation to the area of high performance computing. This extends the discussion from different parallel programming memory models to the working of parallel IO and its bandwidth performance issues that arise in super-computing clusters.

Since parallel IO mainly depends on message passing interface-IO standard and lustre file system, the related research examined points to their specific configuration parameters that determine the IO performance [9, 10, 11, 12]. This adds up few more parameters when parallel seismic (SEG-Y) data IO processing comes in the form of an ExSeisDat user application runs on cluster [4]. This is reviewed and discussed in chapter 3.

Afterwards, chapter 4 reviews background research from third perspective of machine learning techniques. The research reviewed, leads to the motivation of how to approach the solution of overcoming IO performance degradation, with the use of machine learning as a state-of-the-art [96, 14, 15, 16, 17, 18, 21].

By reviewing the related background research, the three research questions have been formed and addressed in sequence. The first research question RQ1 is: What are the configurations that determine high bandwidth for parallel IO and seismic data operations? What is the most suitable technique to accurately predict the changing bandwidth pattern in basic parallel IO, and ExSeisDat's parallel seismic data IO and sorting operations?

The RQ1 is addressed in chapter 5. It is demonstrated that the identified configuration parameters related to message passing interface-IO and the lustre file system influence the parallel IO bandwidth performance, with the use parallel graph plots [25]. These parallel plots have not been utilized by any previous state-of-the-art work in relation to describing the IO performance behaviour over several parameters. The seismic data parameters have been also taken into consideration for their IO and file sorting operations performance.

The basic parallel IO and ExSeisDat's seismic data benchmarking results with respect to configuration parameters, have supported in formation of those parallel graph plots.

The same benchmarking results have been used to perform predictive modelling of IO bandwidth performance behaviour for basic parallel IO, seismic data IO and file sorting operations. The predictive modelling is executed by artificial neural networks based machine learning technique. The prediction models are generated against each identified read/write parallel IO and seismic data operation. This technique has been most suitable to accurately predict the IO bandwidth against the identified configuration parameters. As per observation, the prediction accuracy of the models, range in 62.5-96.5% across all the cases.

Since having the accurate bandwidth prediction models, the next question arises is that how to optimize the IO bandwidth performance on the basis of predictions of each operation model. The second research RQ2 targets the basic parallel IO operations based on the message passing interface-IO, which is: what is the best approach to optimize the basic parallel IO operations performance?

The chapter 6 addresses the RQ2, by extending the work of chapter 5, in the context of basic parallel IO operations bandwidth performance. It is demonstrated that an IO operation performance can be optimized by auto-tuning the related configuration parameters on the basis of that operation model predictions, as being a novel approach. The design of experimental setup starts from re-execution of the parallel IO benchmarks, and regeneration of the prediction models for read/write operations. The prediction accuracy is increased to some extent, which is 63% for read bandwidth predictions and ≈79% for write bandwidth predictions.

Afterwards, the auto-tuning implementation design is mainly discussed. It implements a brute-force approach to check the predicted bandwidth on all specified possible value settings of the tunable parameters with the given non-tunable settings. The tunable settings with maximum bandwidth prediction are selected as the new settings which are then applied to file which is, as intended, distributed on the multiple parallel lustre storage disks according to read/write operation.

The IO bandwidth value is profiled with the Darshan utility when an MPI-IO operation is executed with new configuration settings by all processes in parallel on file. Upon running the designed auto-tuning strategy on number of default settings, it is observed that the parallel IO bandwidth performance is significantly improved. The overall percentage improvement in bandwidth, ranges from 65.7-83.2% in read/write cases.

Furthermore, it is observed that the 3 configuration settings are mostly chosen in case of read operation test cases, by auto-tuning design based on predictions. Those 3 settings yielded ≈83% of improvements in the total test cases. Whereas, for write operation test cases, 5 configuration settings are mostly chosen by the auto-tuning design. Those 5 settings yielded ≈93% of improvements in the total test cases.

It is evident from the overall percentage improvement in IO bandwidth and the number or improvements in total test cases that are presented in this thesis, have significant benefits for the application performance. Therefore, it is further extended to the application of seismic data using ExSeisDat library, to address the third and last research question RQ3 of the thesis: what approach can be used to optimize the ExSeisDat's seismic data IO and sorting operations performance?

The chapter 7 addresses the RQ3, as the third major contribution of this thesis. The auto-tuning design logic is applied to according to seismic data parameters and its operations test cases. It is discussed that the benchmarks for seismic data IO and file sorting operations are re-executed and the prediction models are also regenerated that are outlined in chapter 5.

In this work, there are 6 neural network models analyzed for each of the four seismic data operations: SEG-Y read/write and file sorting via contiguous read/write. The artificial neural networks prediction models are varied over a range of different hidden layers configurations. The purpose of this is to observe the impact of different models nodes configuration on prediction accuracy and auto-tuning results for each operation.

First the prediction accuracy is observed on all the models for each operation. It is mostly seen that the prediction accuracy is notably increased initially, then gradually increased or almost remained constant afterwards, across all the models. This is a common case in all SEG-Y operations prediction models. Secondly, using all the models, the maximum improvement in bandwidth performances noted are ranging from 97.3% to 602.6% throughout all the auto-tuning results.

In addition to this, the percentage of total improvements in the total cases range in 65.2-96.7% across all the models and auto-tuning test cases.

Summarizing the thesis, the novel approach to auto-tune or optimize the parallel HPC IO and its applications, using machine learning based on artificial neural networks, has been significantly proven beneficial indeed, as demonstrated with a number of usecases.

Chapter 9

Conclusion & Future work

In this thesis, the main purpose was to optimize seismic data IO and sorting operations which are performed by an ExSeisDat based application, within a HPC cluster environment. The seismic data is specified by the standard SEG-Y format, which ExSeisDat API utilizes in its library implementation. Since ExSeisDat is built on the standard MPI-IO library, the basic parallel MPI-IO file operations have also been considered for performance optimization. Consequently, the three identified operations were basic MPI-IO, SEG-Y IO and SEG-Y file sorting that were optimized using the AI/ML approach.

Since the storage architectures have been advanced recently in the form of SSDs or NVMe, the HPC community is gradually updating their clusters with this new hardware. However, the current HPC systems have not fully adopted the NVMe/SSDs as their sole multiple storage and parallel file systems are still based on the magnetic HDDs. Instead, a storage-tiers approach is normally being used to store the frequently accessed data in SSDs and infrequent data in HDDs. The one reason can be the cost and capacity of SSDs. Whereas the other main reason can be a vulnerability to failures as recently experimentally demonstrated by Lu et al. in [194].

As the storage architectures are not rapidly evolving, the applications and IO optimization strategies still work around the existing hardware and the parallel file systems. Many recent research studies are mentioned in the thesis that optimize IO using machine learning followed by tuning different parameters based on the predictions. In the same context, a work by Julius et al. is also worth mentioning that overcomes IO overhead by data-aware compression in HPC storage using AI [195]. All these applications from the research studies operate on the existing storage HDDs and their parallel file systems in clusters. In addition to this, the work demonstrated by Sriniket et al. in [196], compares the performances of famous file formats: HDF5, netCDF4 and Zarr, which are normally used by scientific applications in HPC clusters. However, these file formats were tested on SSDs with sequential reads, writes and other basic operations. This involved no parallel file system and parallel IO operations so far during the testing.

Considering this, there are not any potential limitations of the thesis due to changing storage architectures. If in the future, the parallel file systems within HPC clusters fully adopt the SSDs then their API have to be responsible for taking care of the low-level driver's integration. This means a user is able to use different stripe count and stripe size configuration just as the way now or before.

However, since the thesis work is based only on lustre file system, there might be some changes required at the software code level for using different parallel file systems. Further modifications will also be required for correctly porting the MPI-IO prediction

models and auto-tuning to different scientific applications. This is only possible if the data structure and parameters used by applications are adjustable and can be mapped to the basic parallel MPI-IO operations. Otherwise, if there is a legacy code system in storing the data as in case of SEG-Y file format then the separate READ/WRITE prediction models and auto-tuning strategy will be required for IO optimization.

Ultimately, this research still poses a potential future work. Extending this thesis, the machine learning and auto-tuning can be transformed into reinforcement learning. This includes the porting of prediction models and auto-tuning code at the MPI-C/C++ code level to avoid switching and packages loading overheads between different platforms. This can further improve the system performance for optimizing MPI-IO and ExSeisDat's SEG-Y operations. In addition to this, several scientific applications using different file formats can be tested under this IO optimization solution.

Bibliography

[1] Meghan A Fisher, Pádraig Ó Conbhuí, Cathal Ó Brion, Jean-Thomas Acquaviva, Seán Delaney, Gareth S O'brien, Steven Dagg, James Coomer, and Ruairi Short. Exseisdat: A set of parallel i/o and workflow libraries for petroleum seismology. *Oil & Gas Science and Technology–Revue d'IFP Energies nouvelles*, 73:74, 2018.

[2] Iris Bödvarsdóttir and Ask Elklit. Psychological reactions in icelandic earthquake survivors. *Scandinavian Journal of Psychology*, 45(1):3–13, 2004.

[3] Öz Yilmaz. *Seismic data analysis: Processing, inversion, and interpretation of seismic data*. Society of exploration geophysicists, 2001.

[4] Rune Hagelund and Stewart A Levin. Seg-y_r2. 0: Seg-y revision 2.0 data exchange format, 2017.

[5] Gregory F Pfister. An introduction to the infiniband architecture. *High performance mass storage and parallel I/O*, 42:617–632, 2001.

[6] Mark S Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D Underwood, and Robert C Zak. Intel® omni-path architecture: Enabling scalable, high performance fabrics. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 1–9. IEEE, 2015.

[7] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[8] Petros Koutoupis. The lustre distributed filesystem. *Linux Journal*, 2011(210):3, 2011.

[9] Yunchun Li and Hongda Li. Optimization of parallel i/o for cannon's algorithm based on lustre. In *2012 11th International Symposium on Distributed Computing and Applications to Business, Engineering & Science*, pages 31–35. IEEE, 2012.

[10] Wei-keng Liao. Design and evaluation of mpi file domain partitioning methods under extent-based file locking protocol. *IEEE Transactions on Parallel and Distributed Systems*, 22(2):260–272, 2010.

[11] Phillip M Dickens and Jeremy Logan. Y-lib: a user level library to increase the performance of mpi-io in a lustre file system environment. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 31–38. ACM, 2009.

[12] Weikuan Yu, Jeffrey Vetter, R Shane Canon, and Song Jiang. Exploiting lustre file joining for effective collective io. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid'07)*, pages 267–274. IEEE, 2007.

[13] MPI: A Message-Passing Interface Standard Version 3.1, 2015. Accessed: 2019-11-07.

[14] Li Xu, Thomas Lux, Tyler Chang, Bo Li, Yili Hong, Layne Watson, Ali Butt, Danfeng Yao, and Kirk Cameron. Prediction of high-performance computing input/output variability and its application to optimization for system configurations. *Quality Engineering*, 33(2):318–334, 2021.

[15] Jean Luca Bez, Francieli Zanon Boito, Ramon Nou, Alberto Miranda, Toni Cortes, and Philippe OA Navaux. Adaptive request scheduling for the i/o forwarding layer using reinforcement learning. *Future Generation Computer Systems*, 112:1156–1169, 2020.

[16] Babak Behzad, Surendra Byna, and Marc Snir. Optimizing i/o performance of hpc applications with autotuning. *ACM Transactions on Parallel Computing (TOPC)*, 5(4):1–27, 2019.

[17] Ayşe Bağbaba. Improving collective i/o performance with machine learning supported auto-tuning. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 814–821. IEEE, 2020.

[18] Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert Latham, Robert Ross, Shane Snyder, and Stefan M Wild. Machine learning based parallel i/o predictive modeling: A case study on lustre file systems. In *International Conference on High Performance Computing*, pages 184–204. Springer, 2018.

[19] John J Hopfield. Artificial neural networks. *IEEE Circuits and Devices Magazine*, 4(5):3–10, 1988.

[20] Martin T. Hagan, Howard B. Demuth, and Mark Beale. *Neural Network Design*. PWS Publishing Co., USA, 1997.

[21] Jan F Schmidt and Julian M Kunkel. Predicting i/o performance in hpc using artificial neural networks. *Supercomputing Frontiers and Innovations*, 3(3):19–33, 2016.

[22] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[23] Radwa Elshawi, Abdul Wahab, Ahmed Barnawi, and Sherif Sakr. Dlbench: a comprehensive experimental evaluation of deep learning frameworks. *Cluster Computing*, pages 1–22, 2021.

[24] Michael R Wyatt II, Stephen Herbein, Todd Gamblin, Adam Moody, Dong H Ahn, and Michela Taufer. Prionn: Predicting runtime and io using neural networks. In *Proceedings of the 47th International Conference on Parallel Processing*, page 46. ACM, 2018.

[25] Daniel Haziza and Jérémy Rapin. Gs: Hiplot-high dimensional interactive plotting, 2020.

[26] Abdul Jabbar Saeed Tipu, Padraig Ó Conbhuí, and Enda Howley. Applying neural networks to predict hpc-i/o bandwidth over seismic data on lustre file system for exseisdat. *Cluster Computing*, pages 1–22, 2021.

[27] Abdul Jabbar Saeed Tipu, Pádraig Ó Conbhuí, and Enda Howley. Artificial neural networks based predictions towards the auto-tuning and optimization of parallel io bandwidth in hpc system. *Cluster Computing*, pages 1–20, 2022.

[28] Abdul Jabbar Saeed Tipu, Pádraig Ó Conbhuí, and Enda Howley. Seismic data io and sorting optimization in hpc through anns prediction based auto-tuning for exseisdat. *Neural Computing and Applications*, pages 1–34, 2022.

[29] Kevin Dowd and Charles Severance. High performance computing. 2010.

[30] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[31] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020.

[32] Walter B Ligon III and Robert B Ross. An overview of the parallel virtual file system. In *Proceedings of the 1999 Extreme Linux Workshop*. Citeseer, 1999.

[33] Philip H Carns, Walter B Ligon III, Robert B Ross, and Rajeev Thakur. {PVFS}: A parallel file system for linux clusters. In *4th Annual Linux Showcase & Conference ({ALS} 2000)*, 2000.

[34] Terry Jones, Alice Koniges, and Robert Kim Yates. Performance of the ibm general parallel file system. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pages 673–681. IEEE, 2000.

[35] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.

[36] Lisandro D Dalcin, Rodrigo R Paz, Pablo A Kler, and Alejandro Cosimo. Parallel distributed computing using python. *Advances in Water Resources*, 34(9):1124–1139, 2011.

[37] Julian Martin Kunkel, Eugen Betke, Matt Bryson, Philip Carns, Rosemary Francis, Wolfgang Frings, Roland Laifer, and Sandra Mendez. Tools for analyzing parallel i/o. In *International Conference on High Performance Computing*, pages 49–70. Springer, 2018.

[38] David W Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20(4):657–673, 1994.

[39] Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. mpijava: An object-oriented java interface to mpi. In *International Parallel Processing Symposium*, pages 748–762. Springer, 1999.

[40] Lisandro Dalcin and Yao-Lung Leo Fang. mpi4py: Status update after 12 years of development. *Computing in Science & Engineering*, 2021.

[41] Ammar Ahmad Awan, Jereon Bédorf, Ching-Hsiang Chu, Hari Subramoni, and Dhabaleswar K Panda. Scalable distributed dnn training using tensorflow and cuda-aware mpi: Characterization, designs, and performance evaluation. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 498–507. IEEE, 2019.

[42] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. Can shared-memory model serve as a bridging model for parallel computation? In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 72–83, 1997.

[43] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads programming.* O'Reilly & Associates, Inc., 1996.

[44] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.

[45] Roger Eggen and Maurice Eggen. Thread and process efficiency in python. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 32–36. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2019.

[46] Buu Q Pham and Mark S Gordon. Hybrid distributed/shared memory model for the ri-mp2 method in the fragment molecular orbital framework. *Journal of Chemical Theory and Computation*, 15(10):5252–5258, 2019.

[47] Bratin Saha, Xiaocheng Zhou, Hu Chen, Ying Gao, Shoumeng Yan, Mohan Rajagopalan, Jesse Fang, Peinan Zhang, Ronny Ronen, and Avi Mendelson. Programming model for a heterogeneous x86 platform. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 431–440, 2009.

[48] Peter N Glaskowsky. Nvidia's fermi: the first complete gpu computing architecture. *White paper*, 18, 2009.

[49] Mike Mantor. Amd radeon™ hd 7970 with graphics core next (gcn) architecture. In *2012 IEEE Hot Chips 24 Symposium (HCS)*, pages 1–35. IEEE, 2012.

[50] Seong Hwan Kim. White paper— enabling smart software defined networks.

[51] Kay. `https://www.ichec.ie/about/infrastructure/kay`.

[52] Ching-Hsiang Chu, Xiaoyi Lu, Ammar A Awan, Hari Subramoni, Bracy Elton, and Dhabaleswar K Panda. Exploiting hardware multicast and gpudirect rdma for efficient broadcast. *IEEE Transactions on Parallel and Distributed Systems*, 30(3):575–588, 2018.

[53] Dan Negrut, Radu Serban, Ang Li, and Andrew Seidl. Unified memory in cuda 6.0. a brief overview of related data access and transfer issues. *SBEL, Madison, WI, USA, Tech. Rep. TR-2014-09*, 2014.

[54] Steven Wei Der Chien, Ivy Bo Peng, and Stefano Markidis. Performance evaluation of advanced features in CUDA unified memory. *CoRR*, abs/1910.09598, 2019.

[55] Raphael Landaverde, Tiansheng Zhang, Ayse K Coskun, and Martin Herbordt. An investigation of unified memory access performance in cuda. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2014.

[56] Adam Thompson and C Newburn. Gpudirect storage: A direct path between storage and gpu memory. *NVIDIA Developer Whitepapers*, 8, 2019.

[57] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[58] Isha Salian. How nasa is helping humans reach the red planet, using gpus and 3d visualization, 2019.

[59] Robert E Bartels. Development, verification and use of gust modeling in the nasa computational fluid dynamics code fun3d. Technical report, 2012.

[60] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snirt, Bernard Traversat, and Parkson Wong. Overview of the mpi-io parallel i/o interface. In *Input/Output in Parallel and Distributed Computer Systems*, pages 127–146. Springer, 1996.

[61] Brent Welch. Posix i/o high performance computing extensions, December 2005.

[62] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective i/o in romio. In *Proceedings. Frontiers' 99. Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE, 1999.

[63] Yin Lu, Yong Chen, Prathamesh Amritkar, Rajeev Thakur, and Yu Zhuang. A new data sieving approach for high performance i/o. In *Future Information Technology, Application, and Service*, pages 111–121. Springer, 2012.

[64] Javier Garcia Blas, Florin Isaila, David E Singh, and Jesus Carretero. View-based collective i/o for mpi-io. In *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 409–416. IEEE, 2008.

[65] Yong Chen, Xian-He Sun, Rajeev Thakur, Huaiming Song, and Hui Jin. Improving parallel i/o performance with data layout awareness. In *2010 IEEE International Conference on Cluster Computing*, pages 302–311. IEEE, 2010.

[66] Viet-Trung Tran. Towards a storage backend optimized for atomic mpi-i/o for parallel scientific applications. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 2057–2060. IEEE, 2011.

[67] Surendra Byna, Yong Chen, Xian-He Sun, Rajeev Thakur, and William Gropp. Parallel i/o prefetching using mpi file caching and i/o signatures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 44. IEEE Press, 2008.

[68] Jun He, Huaiming Song, Xian-He Sun, Yanlong Yin, and Rajeev Thakur. Pattern-aware file reorganization in mpi-io. In *Proceedings of the sixth workshop on Parallel Data Storage*, pages 43–48, 2011.

[69] Efecan Poyraz, Heming Xu, and Yifeng Cui. Application-specific i/o optimizations on petascale supercomputers. *Procedia Computer Science*, 29:910–923, 2014.

[70] Yingjin Qian, Ruihai Yi, Yimo Du, Nong Xiao, and Shiyao Jin. Dynamic i/o congestion control in scalable lustre file system. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5. IEEE, 2013.

[71] Yuichi Tsujita, Kazumi Yoshinaga, Atsushi Hori, Mikiko Sato, Mitaro Namiki, and Yutaka Ishikawa. Multithreaded two-phase i/o: Improving collective mpi-io performance on a lustre file system. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 232–235. IEEE, 2014.

[72] Jeff R Hammond, Andreas Schäfer, and Rob Latham. To int_max... and beyond!: exploring large-count support in mpi. In *Proceedings of the 2014 Workshop on Exascale MPI*, pages 1–8. IEEE Press, 2014.

[73] Jaehyun Han, Deoksang Kim, and Hyeonsang Eom. Improving the performance of lustre file system in hpc environments. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 84–89. IEEE, 2016.

[74] Shuibing He, Yang Wang, Xian-He Sun, Chuanhe Huang, and Chengzhong Xu. Heterogeneity-aware collective i/o for parallel i/o systems with hybrid hdd/ssd servers. *IEEE Transactions on Computers*, 66(6):1091–1098, 2016.

[75] Xuechen Zhang, Song Jiang, Alseny Diallo, and Lei Wang. Ir+: Removing parallel i/o interference of mpi programs via data replication over heterogeneous storage devices. *Parallel Computing*, 76:91–105, 2018.

[76] Adrian Jackson, Fiona Reid, Joachim Hein, Alejandro Soba, and Xavier Saez. High performance i/o. In *2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 349–356. IEEE, 2011.

[77] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 36–47, 2011.

[78] Russ Rew and Glenn Davis. Netcdf: an interface for scientific data access. *IEEE computer graphics and applications*, 10(4):76–82, 1990.

[79] Yuan Wang, Yongquan Lu, Chu Qiu, Pengdong Gao, and Jintao Wang. Performance evaluation of a infiniband-based lustre parallel file system. *Procedia Environmental Sciences*, 11:316–321, 2011.

[80] Sun Jian, Li Zhan-huai, and Zhang Xiao. The performance optimization of lustre file system. In *2012 7th International Conference on Computer Science & Education (ICCSE)*, pages 214–217. IEEE, 2012.

[81] Eduardo C Inacio, Pedro A Barbetta, and Mario AR Dantas. A statistical analysis of the performance variability of read/write operations on parallel file systems. *Procedia Computer Science*, 108:2393–2397, 2017.

[82] Tiezhu Zhao and Jinlong Hu. Performance evaluation of parallel file system based on lustre and grey theory. In *2010 Ninth International Conference on Grid and Cloud Computing*, pages 118–123. IEEE, 2010.

[83] Tiezhu Zhao, Verdi March, Shoubin Dong, and Simon See. Evaluation of a performance model of lustre file system. In *2010 Fifth Annual ChinaGrid Conference*, pages 191–196. IEEE, 2010.

[84] Philippe Jousset, Thomas Reinsch, Trond Ryberg, Hanna Blanck, Andy Clarke, Rufat Aghayev, Gylfi P Hersir, Jan Henninges, Michael Weber, and Charlotte M Krawczyk. Dynamic strain determination using fibre-optic cables allows imaging of seismological and structural features. *Nature communications*, 9(1):1–11, 2018.

[85] Alden Partridge Colvocoresses. *A unified plane coordinate reference system*. The Ohio State University, 1965.

[86] Baker K.B. and Wing S. A new magnetic coordinate system for conjugate studies at high latitudes. *Journal of Geophysical Research: Space Physics*, 94(A7):9139–9143, 1989.

[87] Sanjeev K Srivastava. Techniques for developing resources to understand geographic and projected coordinate systems. *Journal of Spatial Science*, 59(1):167–176, 2014.

[88] Giuseppe Stanghellini and Gabriela Carrara. Segy-change: The swiss army knife for the seg-y files. *SoftwareX*, 6:42–47, 2017.

[89] Steinar Hustoft, Jürgen Mienert, Stefan Bünz, and Hervé Nouzé. High-resolution 3d-seismic data indicate focussed fluid migration pathways above polygonal fault systems of the mid-norwegian margin. *Marine Geology*, 245(1-4):89–106, 2007.

[90] Fan-Chi Lin, Dunzhu Li, Robert W Clayton, and Dan Hollis. High-resolution 3d shallow crustal structure in long beach, california: Application of ambient noise tomography on a dense seismic array. *Geophysics*, 78(4):Q45–Q56, 2013.

[91] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. Understanding and improving computational science storage access through continuous characterization. *ACM Transactions on Storage (TOS)*, 7(3):1–26, 2011.

[92] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 24/7 characterization of petascale i/o workloads. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE, 2009.

[93] Russ Miller and Laurence Boxer. *Algorithms sequential & parallel: A unified approach*. Cengage Learning, 2012.

[94] Gareth O'Brien, Sean Delaney, Michael Igoe, and John Doherty. Kirchhoff migration with a time-shift extended imaging condition: A synthetic example. In *SEG Technical Program Expanded Abstracts 2017*, pages 4671–4675. Society of Exploration Geophysicists, 2017.

[95] Corentin Roussel, Kai Keller, Mohamed Gaalich, Leonardo Bautista Gomez, and Julien Bigot. Pdi, an approach to decouple i/o concerns from high-performance simulation codes, 2017.

[96] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[97] Kathryn D Betts and Kyle R Jaep. The dawn of fully automated contract drafting: Machine learning breathes new life into a decades-old promise. *Duke L. & Tech. Rev.*, 15:216, 2016.

[98] Stephen M Schueller, Adrian Aguilera, and David C Mohr. Ecological momentary interventions for depression and anxiety. *Depression and anxiety*, 34(6):540–545, 2017.

[99] Ahmad Mansour, Ahmed Hassan, Wessam M Hussein, and Ehab Said. Automated vehicle detection in satellite images using deep learning. In *International Conference on Aerospace Sciences and Aviation Technology*, volume 18, pages 1–8. The Military Technical College, 2019.

[100] Stephen Marsland. *Machine learning: an algorithmic perspective.* Chapman and Hall/CRC, 2011.

[101] Dastan Maulud and Adnan M Abdulazeez. A review on linear regression comprehensive in machine learning. *Journal of Applied Science and Technology Trends*, 1(4):140–147, 2020.

[102] Raymond E Wright. Logistic regression. 1995.

[103] Sauptik Dhar, Junyao Guo, Jiayi Liu, Samarth Tripathi, Unmesh Kurup, and Mohak Shah. A survey of on-device machine learning: An algorithms and learning theory perspective. *ACM Transactions on Internet of Things*, 2(3):1–49, 2021.

[104] Francis Bach and Guillaume Obozinski. Sparse methods for machine learning theory and algorithms. *ECML/PKDD Tutorial*, 2010.

[105] Fen Xia, Tie-Yan Liu, Jue Wang, Wensheng Zhang, and Hang Li. Listwise approach to learning to rank: theory and algorithm. In *Proceedings of the 25th international conference on Machine learning*, pages 1192–1199, 2008.

[106] Ran Gilad-Bachrach, Amir Navot, and Naftali Tishby. Margin based feature selection-theory and algorithms. In *Proceedings of the twenty-first international conference on Machine learning*, page 43, 2004.

[107] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, et al. Api design for machine learning software: experiences from the scikit-learn project. *arXiv preprint arXiv:1309.0238*, 2013.

[108] Jason Brownlee. Data preparation for machine learning, 2022.

[109] Jason Brownlee. *Data preparation for machine learning: data cleaning, feature selection, and data transforms in Python.* Machine Learning Mastery, 2020.

[110] Zahraa Said Abdallah, Lan Du, and Geoffrey I Webb. Data preparation., 2017.

[111] Ivan Miguel Pires, Faisal Hussain, Nuno M Garcia, Petre Lameski, and Eftim Zdravevski. Homogeneous data normalization and deep learning: A case study in human activity classification. *Future Internet*, 12(11):194, 2020.

[112] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[113] Yoshua Bengio, Aaron Courville, and Ian J Goodfellow. Deep learning: adaptive computation and machine learning. *Bengio. A. Courville*, 2016.

[114] Prithwiraj Choudhury, Ryan T Allen, and Michael G Endres. Machine learning for pattern discovery in management research. *Strategic Management Journal*, 42(1):30–57, 2021.

[115] Luguang Wang, Fei Long, Wei Liao, and Hong Liu. Prediction of anaerobic digestion performance and identification of critical operational parameters using machine learning algorithms. *Bioresource technology*, 298:122495, 2020.

[116] Chris Chatfield. *Time-series forecasting.* Chapman and Hall/CRC, 2000.

[117] Jan G De Gooijer and Rob J Hyndman. 25 years of time series forecasting. *International journal of forecasting*, 22(3):443–473, 2006.

[118] Bryan Lim and Stefan Zohren. Time-series forecasting with deep learning: a survey. *Philosophical Transactions of the Royal Society A*, 379(2194):20200209, 2021.

[119] Hari Prasanna Das, Ryan Tran, Japjot Singh, Xiangyu Yue, Geoffrey Tison, Alberto Sangiovanni-Vincentelli, and Costas J Spanos. Conditional synthetic data generation for robust machine learning applications with limited pandemic data. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 11792–11800, 2022.

[120] Kangwook Lee, Maximilian Lam, Ramtin Pedarsani, Dimitris Papailiopoulos, and Kannan Ramchandran. Speeding up distributed machine learning using codes. *IEEE Transactions on Information Theory*, 64(3):1514–1529, 2017.

[121] Wes McKinney et al. pandas: a foundational python library for data analysis and statistics. *Python for high performance and scientific computing*, 14(9):1–9, 2011.

[122] V. Roshan Joseph. Optimal ratio for data splitting. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 2022.

[123] Daniel Berrar. Cross-validation., 2019.

[124] Seldon Technologies. Supervised vs unsupervised learning explained, September 2022.

[125] Pádraig Cunningham, Matthieu Cord, and Sarah Jane Delany. Supervised learning. In *Machine learning techniques for multimedia*, pages 21–49. Springer, 2008.

[126] Sotiris B Kotsiantis. Decision trees: a recent overview. *Artificial Intelligence Review*, 39(4):261–283, 2013.

[127] Kevin P. Murphy. Naive bayes classifiers. *University of British Columbia*, 18(60):1–8, 2006.

[128] Archana Chaudhary, Savita Kolhe, and Raj Kamal. An improved random forest classifier for multi-class classification. *Information Processing in Agriculture*, 3(4):215–222, 2016.

[129] Padraig Cunningham and Sarah Jane Delany. K-nearest neighbour classifiers-a tutorial. *ACM Computing Surveys (CSUR)*, 54(6):1–25, 2021.

[130] Michael Rapp, Eneldo Loza Mencía, Johannes Fürnkranz, Vu-Linh Nguyen, and Eyke Hüllermeier. Learning gradient boosted multi-label classification rules. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 124–140. Springer, 2020.

[131] Xin Wu, Yuchen Gao, and Dian Jiao. Multi-label classification based on random forest algorithm for non-intrusive load monitoring system. *Processes*, 7(6):337, 2019.

[132] Weiwei Cheng and Eyke Hüllermeier. Combining instance-based learning and logistic regression for multilabel classification. *Machine Learning*, 76(2):211–225, 2009.

[133] Marcin Czajkowski and Marek Kretowski. The role of decision tree representation in regression problems–an evolutionary perspective. *Applied soft computing*, 48:458–475, 2016.

[134] Zoubin Ghahramani. Unsupervised learning. In *Summer school on machine learning*, pages 72–112. Springer, 2003.

[135] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)*, 28(1):100–108, 1979.

[136] Khaled Alsabti, Sanjay Ranka, and Vineet Singh. An efficient k-means clustering algorithm. 1997.

[137] Kristina P Sinaga and Miin-Shen Yang. Unsupervised k-means clustering algorithm. *IEEE access*, 8:80716–80727, 2020.

[138] Douglas A Reynolds. Gaussian mixture models. *Encyclopedia of biometrics*, 741(659-663), 2009.

[139] Cinzia Viroli and Geoffrey J McLachlan. Deep gaussian mixture models. *Statistics and Computing*, 29(1):43–51, 2019.

[140] Jakob J Verbeek, Nikos Vlassis, and Ben Kröse. Efficient greedy learning of gaussian mixture models. *Neural computation*, 15(2):469–485, 2003.

[141] Markus Hegland. The apriori algorithm–a tutorial. *Mathematics and computation in imaging science and information processing*, pages 209–262, 2007.

[142] Ning Li, Li Zeng, Qing He, and Zhongzhi Shi. Parallel implementation of apriori algorithm based on mapreduce. In *2012 13th ACIS international conference on software engineering, artificial intelligence, networking and parallel/distributed computing*, pages 236–241. IEEE, 2012.

[143] Cristian Aflori and Mitica Craus. Grid implementation of the apriori algorithm. *Advances in engineering software*, 38(5):295–300, 2007.

[144] Andrew Ng. Machine learning, 2018.

[145] George AF Seber and Alan J Lee. *Linear regression analysis*. John Wiley & Sons, 2012.

[146] Joshua O Ighalo, Adewale George Adeniyi, and Goncalo Marques. Application of linear regression algorithm and stochastic gradient descent in a machine-learning environment for predicting biomass higher heating value. *Biofuels, Bioproducts and Biorefining*, 14(6):1286–1295, 2020.

[147] Lars Kutzbach, Jodi Schneider, Torsten Sachs, Giebels M., Nykänen H., Shurpali N.J., Martikainen P.J., Alm J., and Wilmking M. Co 2 flux determination by closed-chamber methods can be seriously biased by inappropriate application of linear regression. *Biogeosciences*, 4(6):1005–1025, 2007.

[148] Poliak E.I., Shim M.K., Kim G.S., and Choo W.Y. Application of linear regression analysis in accuracy assessment of rolling force calculations. *Metals and Materials*, 4(5):1047–1056, 1998.

[149] H. Passing and W. Bablok. A new biometrical procedure for testing the equality of measurements from two different analytical methods. application of linear regression procedures for method comparison studies in clinical chemistry, part i. 1983.

[150] Xinyou Yin, JAN Goudriaan, Egbert A Lantinga, JAN Vos, and Huub J Spiertz. A flexible sigmoid function of determinate growth. *Annals of botany*, 91(3):361–371, 2003.

[151] Lian Niu. A review of the application of logistic regression in educational research: Common issues, implications, and suggestions. *Educational Review*, 72(1):41–67, 2020.

[152] Wang Q.Q., Yu S.C., Xiao Qi, Hu Y.H., Zheng W.J., Shi J.X., and Yao H.Y. Overview of logistic regression model analysis and application. *Zhonghua yu fang yi xue za zhi [Chinese journal of preventive medicine]*, 53(9):955–960, 2019.

[153] Paul D Allison. *Logistic regression using SAS: Theory and application*. SAS institute, 2012.

[154] SARO Lee. Application of logistic regression model and its validation for landslide susceptibility mapping using gis and remote sensing data. *International Journal of remote sensing*, 26(7):1477–1491, 2005.

[155] Guang-Bin Huang, Hongming Zhou, Xiaojian Ding, and Rui Zhang. Extreme learning machine for regression and multiclass classification. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 42(2):513–529, 2011.

[156] Murat Koklu and Ilker Ali Ozkan. Multiclass classification of dry beans using computer vision and machine learning techniques. *Computers and Electronics in Agriculture*, 174:105507, 2020.

[157] Fatih Gürcan. Multi-class classification of turkish texts with machine learning algorithms. In *2018 2nd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, pages 1–5. IEEE, 2018.

[158] Panca V. and Rustam Zuherman. Application of machine learning on brain cancer multiclass classification. In *AIP Conference Proceedings*, volume 1862, page 030133. AIP Publishing LLC, 2017.

[159] Johannes Stallkamp, Marc Schlipsing, Jan Salmen, and Christian Igel. The german traffic sign recognition benchmark: a multi-class classification competition. In *The 2011 international joint conference on neural networks*, pages 1453–1460. IEEE, 2011.

[160] Xue Ying. An overview of overfitting and its solutions. In *Journal of physics: Conference series*, volume 1168, page 022022. IOP Publishing, 2019.

[161] Charu C Aggarwal. An introduction to neural networks. In *Neural Networks and Deep Learning*, pages 1–52. Springer, 2018.

[162] Chigozie Enyinna Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: comparison of trends in practice and research for deep learning. In *2nd International Conference on Computational Sciences and Technology*, pages 124–133, 2021.

[163] Lukas Schott, Jonas Rauber, Matthias Bethge, and Wieland Brendel. Towards the first adversarially robust neural network model on mnist. In *International Conference on Learning Representations*, 2018.

[164] Alexander Shekhovtsov and Boris Flach. Feed-forward propagation in probabilistic neural networks with categorical and max layers. In *International conference on learning representations*, 2018.

[165] Henseler J. Back propagation. *Artificial neural networks*, pages 37–66, 1995.

[166] Cort J Willmott and Kenji Matsuura. Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance. *Climate research*, 30(1):79–82, 2005.

[167] Tianfeng Chai and Roland R Draxler. Root mean square error (rmse) or mean absolute error (mae)?–arguments against avoiding rmse in the literature. *Geoscientific model development*, 7(3):1247–1250, 2014.

[168] Arnaud De Myttenaere, Boris Golden, Bénédicte Le Grand, and Fabrice Rossi. Mean absolute percentage error for regression models. *Neurocomputing*, 192:38–48, 2016.

[169] Madhumitha Ravichandran, Guanyu Su, Chi Wang, Jee Hyun Seong, Artyom Kossolapov, Bren Phillips, Md Mahamudur Rahman, and Matteo Bucci. Decrypting the boiling crisis through data-driven exploration of high-resolution infrared thermometry measurements. *Applied Physics Letters*, 118(25):253903, 2021.

[170] Nilmini Wickramasinghe, Nalika Ulapane, Tuan Anh Nguyen, Amir Andargoli, Chinedu Ossai, Nadeem Shuakat, and John Zelcer. Towards discovering digital twins of dementia patients: Matching the phases of cognitive decline. *Alzheimer's & Dementia*, 18:e066336, 2022.

[171] Marcus Cheetham, Catia Cepeda, Hugo Gamboa, Christoph Hoelscher, and Seyedabolfazl Valizadeh. Automated detection of decision-making style, based on users' online mouse pointer activity. In *16th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC 2023)*, 2023.

[172] Cameron A. Colin and Windmeijer Frank A.G. R-squared measures for count data regression models with applications to health-care utilization. *Journal of Business & Economic Statistics*, 14(2):209–220, 1996.

[173] Bob L Sturm. Classification accuracy is not enough. *Journal of Intelligent Information Systems*, 41(3):371–406, 2013.

[174] James T Townsend. Theoretical analysis of an alphabetic confusion matrix. *Perception & Psychophysics*, 9(1):40–50, 1971.

[175] Michael Buckland and Fredric Gey. The relationship between recall and precision. *Journal of the American society for information science*, 45(1):12–19, 1994.

[176] Zachary C Lipton, Charles Elkan, and Balakrishnan Naryanaswamy. Thresholding classifiers to maximize f1 score. *stat*, 1050:14, 2014.

[177] Luzia Gonçalves, Ana Subtil, M Rosário Oliveira, and Patricia de Zea Bermudez. Roc curve estimation: An overview. *REVSTAT-Statistical journal*, 12(1):1–20, 2014.

[178] John J Dziak, Donna L Coffman, Stephanie T Lanza, Runze Li, and Lars S Jermiin. Sensitivity and specificity of information criteria. *Briefings in bioinformatics*, 21(2):553–565, 2020.

[179] Andrew P Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern recognition*, 30(7):1145–1159, 1997.

[180] Deng Julong. Introduction to grey system theory. *The Journal of grey system*, 1(1):1–24, 1989.

[181] Zheng W., Fang J., Juan C., Wu F., Pan X., Wang H., Sun X., Yuan Y., Xie M., Huang C., Tang T., and Wang Z. Auto-tuning mpi collective operations on large-scale parallel systems. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 670–677, 2019.

[182] Álvaro Brandón Hernández, María S Perez, Smrati Gupta, and Victor Muntés-Mulero. Using machine learning to optimize parallelism in big data applications. *Future Generation Computer Systems*, 86:1076–1092, 2018.

[183] Abien Fred Agarap. Deep learning using rectified linear units (relu), 03 2018.

[184] Nikhil Ketkar. Introduction to pytorch. In *Deep learning with python*, pages 195–208. Springer, 2017.

[185] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[186] Max Kuhn, Kjell Johnson, et al. *Applied predictive modeling*, volume 26. Springer, 2013.

[187] Ummul Khair, Hasanul Fahmi, Sarudin Al Hakim, and Robbi Rahim. Forecasting error calculation with mean absolute deviation and mean absolute percentage error. In *Journal of Physics: Conference Series*, volume 930, page 012002. IOP Publishing, 2017.

[188] Didier El Baz. Iot and the need for high performance computing. In *2014 International Conference on Identification, Information and Knowledge in the Internet of Things*, pages 1–6. IEEE, 2014.

[189] Muhammad Shafiq, Zhihong Tian, Yanbin Sun, Xiaojiang Du, and Mohsen Guizani. Selection of effective machine learning algorithm and bot-iot attacks traffic identification for internet of things in smart city. *Future Generation Computer Systems*, 107:433–442, 2020.

[190] Jing Qiu, Zhihong Tian, Chunlai Du, Qi Zuo, Shen Su, and Binxing Fang. A survey on access control in the age of internet of things. *IEEE Internet of Things Journal*, 7(6):4682–4696, 2020.

[191] Muhammad Shafiq, Zhihong Tian, Ali Kashif Bashir, Xiaojiang Du, and Mohsen Guizani. Corrauc: a malicious bot-iot traffic detection method in iot network using machine-learning techniques. *IEEE Internet of Things Journal*, 8(5):3242–3254, 2020.

[192] Muhammad Shafiq, Zhihong Tian, Ali Kashif Bashir, Xiaojiang Du, and Mohsen Guizani. Iot malicious traffic identification using wrapper-based feature selection mechanisms. *Computers & Security*, 94:101863, 2020.

[193] Eugen Betke and Julian Kunkel. Footprinting parallel i/o–machine learning to classify application's i/o behavior. In *International Conference on High Performance Computing*, pages 214–226. Springer, 2019.

[194] Ruiming Lu, Erci Xu, Yiming Zhang, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Minglu Li, and Jiesheng Wu. {NVMe}{SSD} failures in the field: the {Fail-Stop} and the {Fail-Slow}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1005–1020, 2022.

[195] Julius Plehn, Anna Fuchs, Michael Kuhn, Jakob Lüttgau, and Thomas Ludwig. Data-aware compression for hpc using machine learning. *ACM SIGOPS Operating Systems Review*, 56(1):62–69, 2022.

[196] Sriniket Ambatipudi and Suren Byna. A comparison of hdf5, zarr, and netcdf4 in performing common i/o operations. *arXiv preprint arXiv:2207.09503*, 2022.

Appendices

Appendix A - 3 columns version of Table 6.8.

Table 1 - READ Auto-tuning Improvement Results.

Metrics	Default Figures	Tuned Figures
Mean Bandwidth (MiB/s)	37798.1	62630.5
Max. Bandwidth (MiB/s)	712334.2	697943.6
Min. Bandwidth (MiB/s)	121.8	307.9
Median Bandwidth (MiB/s)	14020.5	17731.1
Standard Deviation (MiB/s)	85578.9	114305.7
Variance	7323745984.2	13065796947.6
	Overall Bandwidth Improvement(%)	65.7%
	Improvements	1206/1458
	Improvements(%)	82.7%

Appendix B - 3 columns version of Table 6.10.

Table 2 - WRITE Auto-tuning Improvement Results.

Metrics	Default Figures	Tuned Figures
Mean Bandwidth (MiB/s)	5346.9	9798.1
Max. Bandwidth (MiB/s)	18944.6	19563.5
Min. Bandwidth (MiB/s)	70.9	3488.0
Median Bandwidth (MiB/s)	4718.1	10446.1
Standard Deviation (MiB/s)	3192.4	3350.4
Variance	10191711.7	11225469.3
	Overall Bandwidth Improvement(%)	83.2%
	Improvements	1353/1458
	Improvements(%)	92.8%