© 2015 Huong Vu Thanh Luu

OPTIMIZING I/O PERFORMANCE FOR HIGH PERFORMANCE COMPUTING APPLICATIONS: FROM AUTO-TUNING TO A FEEDBACK-DRIVEN APPROACH

BY

HUONG VU THANH LUU

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Doctoral Committee:

Professor Marianne Winslett, Chair
Professor William Gropp, Director of Research
Professor Marc Snir
Dr Robert Ross, Argonne National Laboratory

ABSTRACT

The 2014 TOP500 supercomputer list includes over 40 deployed petascale systems, and the high performance computing (HPC) community is working toward developing the first exaflop system by 2023. Scientific applications on such large-scale computers often read and write a lot of data. With such rapid growth in computing power and data intensity, I/O continues to be a challenging factor in determining the overall performance of HPC applications.

We address the problem of optimizing I/O performance for HPC applications by firstly examining the I/O behavior of thousands of supercomputing applications. We analyzed the high-level I/O logs of over a million jobs representing a combined total of six years of I/O behavior across three leading high-performance computing platforms. Our analysis provides *a broad portrait of the state of HPC I/O usage*. We proposed *a simple and effective analysis and visualization procedure* to help scientists who do not have I/O expertise to quickly locate the bottlenecks and inefficiencies in their I/O approach. We proposed *several filtering criteria* for system administrators to find application candidates that are consuming system I/O resources inefficiently. Overall, our analysis techniques can help both application users and platform administrators improve I/O performance and I/O system utilization. In the second part, we develop a framework that can *hide the complexity of the I/O stack* from scientists without penalizing performance. This framework will allow application developers to issue I/O calls without modification and rely on an intelligent runtime system to transparently determine and execute an I/O strategy that takes all the levels of the I/O stack into account. Lastly, we develop *a multi-level tracing framework* that provides a much more detailed feedback for application's I/O runtime behavior. These details are needed for in-depth application's performance analysis and tuning.

To my family, my husband and my bundle of Joyce, for their love and support.

ACKNOWLEDGMENTS

"No duty is more urgent than that of returning thanks." - James Allen

I would like to express my deepest gratitude to all the guidance, support and encouragement from so many people who have helped me through this journey. I could not have done it without you.

First and foremost, I would like to thank my advisors, Professor Marianne Winslett and Professor William Gropp. Your immense knowledge, brilliant vision and research enthusiasm are the motivation for me to continue to strive as a researcher. Professor Marianne Winslett have been a tremendous advisor for me, providing me with so many opportunitites to learn and perform research, helping me with my writing and supporting me financially. I am very fortunate to have her as my advisor and mentor whom I respect tremendously both professionally and personally. I am thankful to Professor Gropp who gave me the freedom to explore on my own, and at the same time the guidance to shape my research direction. He taught me how to question thoughts and express ideas. I am grateful to him for holding me to a high research standard and teaching me how to do research.

I would like to thank my committee members, Professor Marc Snir and Dr Robert Ross for their insightful comments, suggestions and valuable feedbacks at different stages of my research. I am thankful for the opportunities to work with many amazing researchers, Prabat, Suren Byna, Ruth Aydt, Quincey Koziol, Robert Latham in diverse exciting projects. I would also like to thank all my colleagues in department of computer science. I benefited from their assistance and their comments during the research meetings, their discussions and advices in various topics even with my personal struggles. I especially enjoyed the friendship with Elena Caraba and the collaboration with Babak Behzad. Thank you for sharing this journey with me.

My time at UIUC was made enjoyable because of the Vietnamese Student Community that has become a big part of my life and made me feel like home.

I thank them for providing the much needed encouragement, entertainment and helping me since my first day at UIUC or times when I learnt to balance PhD life with being a new parent. Anh chi Thang Cac, Vuong Le, anh chi Hoa Luyen, Hung Duong, Anh Nguyen; Hoi gia dinh: Anh chi Minh My, Phong Quyen, Thuy Trong, Ha Huy, Nhung Quang; cac em Lien Nguyen, Thao Hoang, ... There are too many of you to list here, but do know that I thank you and I appreciate everything you have done for me. I thank my BFFs Phuong Nguyen, Ha Tran, Anh Nguyen, Binh Han for helping me get through the difficult times, and for all the emotional support, comraderie, entertainment, and caring they provided.

Most importantly I am thankful for my extended family without whom there will be no me. Thank you, Mom and Dad and my dearest sister, for loving me, raising me, for continuously believing and supporting me. I want to make you proud and this dissertation is dedicated to you. My family-in-law have constantly encouraged me to reach my potentials and follow my dreams. I am grateful for all the sacrifices you have made to come to the US and help me taking care of Joyce while I pursued this PhD. Thank you for welcoming me into the family.

Finally, I want to thank the loves of my life: my dear husband and my bundle of Joyce. I know I am very lucky - "Lucky I am in love with my best friend, Lucky to have been where we have been". I could only come this far thanks to my loving, supportive, encouraging, and patient husband who helped me every step of the way. I am thankful for my little girl, Joyce to have come into my life. She has been the light of my life for the last three years and who has given me the extra strength and motivation to get things done. I love you more than anything. Daddy and I look forward to accompanying you exploring the world.

TABLE OF CONTENTS

LIST OF TABLES viii

LIST OF FIGURES ix

CHAPTER 1 INTRODUCTION 1
1.1 Motivation 1
1.2 Research Statements 3
1.3 Research Contributions 4
1.4 Thesis Organization 7

CHAPTER 2 A STUDY OF I/O BEHAVIOR ON PETASCALE SUPERCOMPUTERS 8
2.1 Related Work 9
2.2 Target Platforms and Datasets 10
2.3 Darshan: A Lightweight I/O Characterization Tool 17
2.4 Database Schema 19
2.5 Application-Specific Analysis 22
2.6 Platform-wide Analysis 25
2.7 Cross-platform Analysis 51
2.8 Conclusions 58

CHAPTER 3 TAMING PARALLEL I/O COMPLEXITY WITH AUTO-TUNING 61
3.1 Background and Related Work 62
3.2 Auto-tuning Framework 66
3.3 Experimental Setup 67
3.4 Results 71
3.5 Conclusions 76

CHAPTER 4 IN-DEPTH ANALYSIS OF APPLICATIONS' I/O ACTIVITIES WITH RECORDER 78
4.1 Related Work 79
4.2 Framework 82
4.3 Example Use Case 84
4.4 Conclusions 86

CHAPTER 5 OPEN ISSUES AND CONCLUSIONS 88

REFERENCES 92

APPENDIX A MYSQL AND R SCRIPTS 99
A.1 Section 2.2.4 scripts 99
A.2 Section 2.5 scripts 100
A.3 Section 2.6.1 scripts 100
A.4 Section 2.6.2 scripts 103
A.5 Section 2.6.3 scripts 106
A.6 Section 2.6.4 scripts 106
A.7 Section 2.6.5 scripts 109
A.8 Section 2.6.6 scripts 110
A.9 Section 2.7 scripts 111

LIST OF TABLES

2.1 Target platforms and their Darshan datasets 12
2.2 List of I/O functions captured by Darshan 18
2.3 Distribution of time on different file types of Earth1's randomly-selected job 24
2.4 Mira's 15 Applications with Biggest I/O Time 32
2.5 Edison's 15 Applications with Biggest I/O Time 33
2.6 Intrepid's 15 Applications with Biggest I/O Time 34
2.7 Percentage of jobs using a particular I/O paradigm on Edison, Intrepid and Mira 50

3.1 Details of various HPC systems used in this study 69
3.2 Weak scaling configuration for the three I/O benchmarks - Total amount of data 69
3.3 A list of the tunable parameters and their ranges used for experiments in this study. We show the minimum and maximum values for each parameter, with powers-of-two values in between. The last column shows the number of distinct values used for each parameter. 70
3.4 I/O rate and speedups of I/O benchmarks with tuned parameters over default parameters 75
3.5 Tuned parameters of all benchmarks on all the systems for 2048-core experiments 76

4.1 Recorder Framework Summary Table 82
4.2 VPIC-IO parameters and I/O bandwidth 85

LIST OF FIGURES

1.1 Our research contributions 7

2.1 Darshan Coverage on Intrepid 13
2.2 Darshan Coverage on Mira 13
2.3 Darshan Coverage on Edison 14
2.4 Cross-platform comparison of each job's number of processes . 15
2.5 Cross-platform comparison of each job's number of bytes read/written 15
2.6 Cross-platform comparison of each application's maximum number of processes 16
2.7 Cross-platform comparison of each application's maximum bytes transferred 16
2.8 Database schema 20
2.9 Breakdown of total run time for each Earth1 job 23
2.10 Earth1's I/O time and number of POSIX write-only global files (red line) 23
2.11 Correlation between Earth1's bytes and number of POSIX global files 24
2.12 Maximum I/O throughput of each application across all its jobs on a platform, and platform peak. 26
2.13 Number of jobs with a given I/O throughput and total number of bytes on Mira 27
2.14 Number of jobs with a given I/O throughput and total number of bytes on Intrepid 27
2.15 Number of jobs with a given I/O throughput and total number of bytes on Edison 28
2.16 Mira: Breakdown of each application's jobs, by bytes written in each job, and average and maximum I/O throughput of each application's jobs. 28
2.17 Intrepid: Breakdown of each application's jobs, by bytes written in each job, and average and maximum I/O throughput of each application's jobs. 29

2.18 Edison: Breakdown of each application's jobs, by bytes written in each job, and average and maximum I/O throughput of each application's jobs. 29
2.19 Cumulative percentage of platform I/O time consumed by jobs on Edison, Intrepid and Mira 31
2.20 I/O throughput of top 15 big-time applications on Mira . . . 36
2.21 I/O throughput of top 15 big-time applications on Intrepid . 36
2.22 I/O throughput of top 15 big-time applications on Edison . . 37
2.23 Average I/O throughput of Mira's (squares) and Edison's (circles) big-time applications' jobs, by job size 38
2.24 Job sizes for Mira's (left) and Edison's (right) big-time applications 38
2.25 Evolution of PDE1's I/O paradigms. Red dots show the number of processes of each job (right-hand y-axis). 40
2.26 Earth2 read hundreds of thousands of text files 41
2.27 Number of jobs using POSIX IO only (teal) and using MPI-IO directly or indirectly for at least one file (red) 42
2.28 I/O throughput for applications that use only POSIX-IO and those that directly or indirectly use MPI-IO for at least one file 43
2.29 Jobs with more metadata time than data transferring time: Breakdown of jobs by total data size 44
2.30 I/O throughput and I/O time breakdown for jobs that access over a million files 45
2.31 Turbulence1's 290 jobs on Intrepid 47
2.32 Climate1's 3578 runs on Intrepid, sorted by I/O throughput . 47
2.33 Physics7's 199 runs on Edison, sorted by I/O throughput . . . 49
2.34 Breakdown of I/O interface of jobs using global-file paradigm on Edison, Intrepid and Mira 51
2.35 Breakdown of I/O interface of jobs using subset paradigm on Edison, Intrepid and Mira 51
2.36 Breakdown of I/O interface of jobs using local paradigm on Edison, Intrepid and Mira 52
2.37 Choice of I/O paradigm of jobs with more than 10 GB/s throughput on Edison, Intrepid and Mira 52
2.38 Quartiles and outliers of total bytes of each job, for the ten applications that ran on all three platforms 53
2.39 Heat map of Earth1's jobs, broken down by data size and I/O throughput, on Intrepid (left) and Mira (right) 54
2.40 Breakdown of Earth1's total I/O time on Intrepid (left) and Mira (right) 55
2.41 I/O throughput, data size, and number of processes for each of Crossplat1's jobs on three platforms 56

2.42 I/O time and throughput (green dot), bytes accessed (black X) and number of local POSIX read/write files (red diamond) for each of Crossplat1's jobs on Mira. 56
2.43 I/O time and throughput (green dot), bytes accessed (black X) and number of local POSIX read/write files (red diamond) for each of Crossplat1's jobs on Edison. 57
2.44 Breakdown of Weather1's jobs by I/O throughput, number of processes, total data size, and platform 58
2.45 I/O time breakdown of Weather1 jobs on Intrepid 59
2.46 I/O time breakdown of Weather1 jobs on Edison. 59

3.1 The I/O software stack 62
3.2 Overall Architecture of the Auto-Tuning Framework 68
3.3 Throughput improvement for each I/O benchmark running on 128 cores 72
3.4 Throughput improvement for each I/O benchmark running on 2048 cores 73
3.5 Throughput improvement for each I/O benchmark running on 4096 cores 74

4.1 Dynamic instrumentation of the I/O stack by Recorder 83
4.2 Sample trace generated by Recorder 83
4.3 VPIC-IO pattern 85

5.1 Proposed architecture for the Intelligent Runtime System . . . 90

CHAPTER 1

INTRODUCTION

1.1 Motivation

The 2014 TOP500 supercomputer list includes over 40 deployed petascale systems, and the high-performance computing (HPC) community is working toward developing the first exaflop system by 2023. Scientific applications on such large-scale computers often read and write a lot of data. For example, a study of computational science requirements of applications running on computing systems at Oak Ridge National Laboratory in 2007 showed that one of the largest data-producing applications dumps a 160TB restart file every hour [1]. In another study of system workload of an IBM BG/P system at Argonne National Laboratory, an Earth science application reads nearly 3.5 PB of data during two months in 2010 [2]. With such rapid growth in computing power and data intensity, I/O continues to be a challenging factor in determining the overall performance of HPC applications.

We address the problem of optimizing I/O performance for HPC applications by firstly examining the I/O behavior of thousands of supercomputing applications to understand the current state of HPC I/O practices. We believe that analysis of the I/O behavior of applications can bring multifold benefits to application developers, application users and system administrators. By analyzing the runtime behavior of an individual job, we can identify its I/O bottlenecks and potential implementation inefficiencies and suggest improvements to its developers and users. By analyzing the I/O behavior of an application (i.e., the set of all its jobs), we can identify patterns in its behavior. By analyzing the I/O behavior of the workload of a platform , we can give the platform administrators insights into the usage of their storage systems and identify applications that consume I/O resources inefficiently, so that improvements to these applications may free up platform resources for

other applications. By analyzing the changes in I/O behavior when applications migrate to similar or radically different platforms, we can help scientists avoid unexpected performance degradation. I/O behavior analysis can even show us how the behavior of individual applications evolves over time. To accomplish all these purposes, *we need a systematic approach to application-specific, platform-wide, and cross-platform analysis of I/O behavior.*

After studying the I/O behavior of applications on largescale supercomputers, we continue our work in providing scientists with performance portability. Scientists want to write their application once and obtain reasonable performance across multiple systems - they want performance portability. However, obtaining good I/O performance for a broad range of applications on diverse HPC platforms is a major challenge, in part because of complex inter-dependencies between I/O middleware and hardware. The parallel file system and I/O middleware layers all offer optimization parameters that can, in theory, result in better I/O performance. In fact, various studies [3, 4, 5, 6] have shown that significant improvements are achievable when the optimization parameters at multiple layers are tuned appropriately. Unfortunately, the right combination of parameters is highly dependent on the application, HPC platform and problem size/concurrency. Scientific application developers do not have the time or expertise to take on the substantial burden of identifying good parameters for each problem configuration. They resort to using system defaults, a choice that frequently results in poor I/O performance. We expect this problem to be compounded on exascale class machines, which will likely have a deeper software stack with hierarchically arranged hardware resources.

Scientists should be able to achieve good I/O performance without becoming experts on the tunable parameters for every filesystem and I/O middleware layer they encounter. In order to use HPC machines and human resources effectively, it is imperative that we design systems that can hide the complexity of the I/O stack from scientific application developers without penalizing performance. *Our vision is to develop a system that will allow application developers to issue I/O calls without source code modification and rely on an intelligent runtime system to transparently determine and execute an I/O strategy that takes all the levels of the I/O stack into account.*

Automatic tuning and high-level I/O behavior analysis can greatly assist scientists who do not have much I/O expertise to improve their applications'

I/O performance. Nevertheless, an in-depth analysis of application I/O behavior is needed in many cases to expose the origin of the performance bottleneck. Platform-wide I/O workload study and cross-platform analysis often require a lightweight, portable and non-intrusive method of collecting runtime behavior feedback. In order to keep the low overhead, important details are sometime missed. An application with the need of in-depth analysis for its I/O approach will need a tracing tool that can provide a multi-level view of the I/O software stack.

In a hierarchical I/O software stack, the layers provide bridges between the data representations of adjacent levels and offer essential abstractions to users. The layers help hide complex implementation details and employ optimization techniques designed to improve performance. Unfortunately, since each layer is normally treated as a black box, optimizations are seldom coordinated across layers and the source of performance bottlenecks can be extremely difficult to determine. *A multi-level I/O tracing and trace data analysis tool* that presents a view of the function call flow through the entire I/O stack can expose cause and effect relationships across layers and make the origin of performance bottlenecks more apparent.

1.2 Research Statements

In the context of optimizing I/O performance for HPC applications, we specifically aim to answer the following research questions:

1. What is the current state of I/O practice on modern HPC platforms?
2. How can platform admins identify applications that consume I/O resources ineffectively?
3. What can we do to improve I/O performance for an application?
    - 3.1. Is auto-tuning effective for finding well-performing set of I/O optimization parameters?
    - 3.2. How can we help users locate the inefficiency in their I/O approach, without overwhelming details?
    - 3.3. How can detailed feedbacks of applications I/O activities help in identifying performance bottlenecks?

1.3 Research Contributions

We address the research problems stated above using a number of approaches, ranging from studying high-level application's I/O behavior, autotuning framework to multi-level tracing for in-depth analysis.

Firstly, we examine the I/O behavior of thousands of supercomputing applications in the wild, by analyzing the feedbacks of runtime behavior of over a million jobs representing a combined total of six years of I/O behavior across three leading high-performance computing platforms. We mined these feedbacks to analyze the I/O behavior of applications across all their runs on a platform; to analyze the evolution of an application's I/O behavior across time, and as the application moves across platforms; and to analyze the I/O behavior of a platform's entire workload. Our analysis techniques can help both application and platform administrators improve I/O performance and I/O system utilization, by quickly identifying underperforming jobs/applications and offering early intervention to save valuable system resources.

Our contributions includes:

1. The logs provide a broad and often surprising portrait of the state of HPC I/O usage on three modern platforms. For example, among Darshan-instrumented jobs:

   (a) Every widely used I/O paradigm (including file per process, global shared file, or subsetting I/O) is represented in the set of best-performing and worst-performing applications, in terms of aggregate I/O throughput. Thus use of a particular paradigm does not in itself guarantee good or bad performance.

   (b) Roughly a third of jobs have aggregate average I/O throughput no more than that of a single contemporary USB flash memory thumb drive (256 MB/s [7]). Three-quarters of applications never exceed the throughput of four thumb drives in any of their jobs. Over a third of jobs spent more time in I/O metadata functions than in transfer of actual data.

   (c) Roughly half of applications have low throughput because none of their jobs access more than 1GB of data, so that file startup costs cannot be amortized across much data transfer; or because they rely on text files instead of binary files. Even on the most

data-intensive platform we studied, half of applications wrote less than 10GB of data in 99% or more of their jobs. On one platform we studied, roughly one-fifth of applications rely exclusively on text files, which almost certainly guarantee poor performance at scale.

(d) Three-quarters of jobs use only POSIX to perform I/O. This does not condemn a job to poor I/O throughput, but it does suggest a need to investigate why higher-level parallel I/O libraries are not more widely used.

2. We discuss ways to address the above problems, including the use of an application as its own I/O benchmark for purposes of identifying and classifying underperforming I/O behavior, and an I/O boot camp for the users and developers of underperforming applications. The resulting performance improvements could raise the level of satisfaction of application users, application developers, and platform administrators. Two subtle points:

   (a) The I/O performance of an application may satisfy its users but not necessarily the platform administrators, and vice versa. Thus analysis of I/O logs must address the needs of both populations.

   (b) 90% of a platform's I/O usage comes from less than 10% of its applications, but some of these applications do not have many large jobs. The greatest potential resource savings for platform administrators lies in identification and correction of an application's I/O issues before it becomes a top consumer of I/O time. Automated analysis can be particularly helpful here, as smaller jobs are less likely to attract expert human scrutiny.

3. We created a user-friendly, simple analysis and visualization procedure to identify the I/O bottleneck and the inefficiency in application's I/O approach without overwhelming details.

4. We proposed several I/O filters that platform administrators can use to identify potential candidates that are consuming I/O resources inefficiently and can benefit from further investigation.

Our work has resulted in a publication in The 24th International ACM Symposium on High-Performance Parallel and Distributed Computing, 2015.

Secondly, we develop a framework that can *hide the complexity of the I/O stack* from scientists without penalizing performance. This framework will allow application developers to issue I/O calls without modification and rely on an intelligent runtime system to transparently determine and execute an I/O strategy that takes all the levels of the I/O stack into account.

- We design and implement an auto-tuning framework that searches a large space of configurable parameters for multiple layers and transparently sets I/O parameters at runtime to identify optimization settings that perform well.

- We demonstrate performance portability across diverse HPC platforms: a Cray XE6 system at National Energy Research Scientific Computing Center (NERSC); an IBM BlueGene/P system at Argonne Leadership Computing Facility (ALCF) and a Dell PowerEdge C8220 cluster at Texas Advance Computing Center (TACC).

- We demonstrate the applicability of the framework to multiple scientific application benchmarks.

- We demonstrate I/O performance tuning at different scales (both concurrency and dataset size).

Our work has resulted in a publication in The 25th International Conference for High Performance Computing, Networking, Storage and Analysis, 2013 [8].

Lastly, we provide a much more detailed feedback for application's I/O runtime behavior. We develop a system that can provide users with a multi-level view and understanding of the I/O stack. This system could help application developers to identify potential bottlenecks and suggest optimization alternatives. It will also help I/O library developers to realize the inefficiency in their implementation and design.

- We design and implement a tracing tool, called Recorder, that captures I/O activities at multiple layers of the I/O stack, namely at HDF5, MPI-IO and POSIX I/O levels.

- We demonstrate its applicability in helping user to understand I/O activities of an HPC application and the I/O subsystem.

Our work has resulted in a publication in the 5th Workshop on Interfaces and Architectures for Scientific Data Storage (IASDS), 2013 [9].

Our list of contributions is depicted in Figure 1.1.

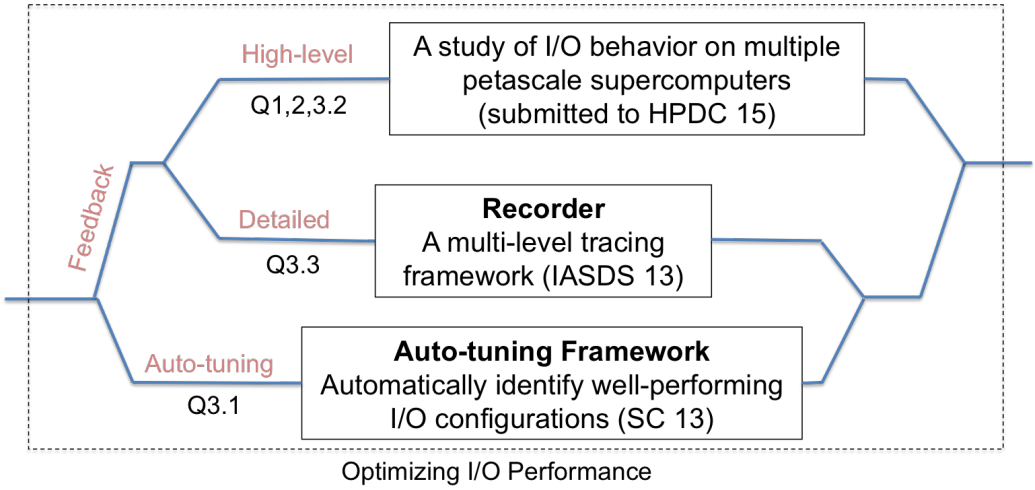


Figure 1.1: Our research contributions

1.4 Thesis Organization

The rest of the thesis is organized as follows: Chapter 2 describes our approach in systematically analyzing high-level I/O feedbacks for application-specific, platform-wide and cross-platform understanding of I/O behavior. Chapter 3 describes our auto-tuning approach for hiding the complexity of the I/O software stack and providing scientists with performance portability. Chapter 4 discusses our work in providing detailed feedback for in-depth I/O analysis. We present our conclusions and directions for future work in Chapter 5.

CHAPTER 2

A STUDY OF I/O BEHAVIOR ON PETASCALE SUPERCOMPUTERS

In this chapter, we examine the I/O behavior of thousands of supercomputing applications in the wild, by analyzing the high-level I/O logs of over a million jobs representing a combined total of six years of I/O behavior across three leading high-performance computing platforms. The logs are collected by Darshan, a lightweight I/O characterization tool with very low overhead, even at production scale. We mined these logs to analyze the I/O behavior of applications across all their runs on a platform; to analyze the evolution of an application's I/O behavior across time, and as the application moves across platforms; and to analyze the I/O behavior of a platform's entire workload. We presents a subset of the insights we gleaned by analyzing Darshan logs from three large-scale supercomputers: Intrepid and Mira at the Argonne Leadership Computing Facility (ALCF), and Edison at the National Energy Research Scientific Computing Center (NERSC). The logs span a substantial period of time 4 years on Intrepid, 18 months on Mira and 9 months on Edison and capture the I/O behavior in the wild of about 1 million jobs, representing thousands of scientific applications and roughly a third of the workload on these platforms. To the best of our knowledge, this is the first study that has been able to compare and contrast the I/O behavior and evolution of many different applications at production scale across platforms. Our analysis techniques can help application users, application developers and platform administrators improve I/O performance and I/O system utilization, by quickly identifying underperforming jobs/applications and offering early intervention to save valuable system resources.

Our work has resulted in a paper submission to The 24th International ACM Symposium on High-Performance Parallel and Distributed Computing, 2015.

The remainder of this chapter is organized as follows. We review related work in Section 2.1. Section 2.2 describes the target platforms and collected

data. Section 2.3 provides the background understanding about Darshan. Section 2.4 discusses how we import the logs into a MySQL database and use SQL scripts to analyze the data. Section 2.5, 2.6 and 2.7 present a three-dimensional analysis of the collected logs: application-specific, platform-wide, and cross-platform. Section 2.8 summarizes our findings and outlines future work. The MySQL and R scripts used to creat graphs in this Chapter are shown in the Appendix. Some of the graphs are created using Excel.

2.1 Related Work

For over 20 years, researchers have sought to understand HPC I/O workloads. As the size, composition, and complexity of platforms and their workloads grows continuously, the topic must be revisited in each generation of platforms (see, e.g., [10], [11] from the 1990s, [12], [13] from the 2000s, and [14], [15] from the 2010s). Many workload studies (e.g., [12], [13], [15] among more recent works) use popular scientific applications such as FLASH [16], GCRM [17], Nek5000 [18], CESM [19], and their associated benchmarks as representative of the entire I/O workload on the system. Such benchmarks are widely used to tune and refine I/O libraries and storage systems. Since these apps are widely used in their fields, any improvements made to them can benefit many users. As important as they are, however, these well-studied apps and benchmarks are not necessarily representative of the long tail of apps that constitute the majority of submitted jobs. By considering a platforms entire workload, we can gain additional insights into its I/O system usage. By considering multiple platforms and many apps, we can gain general insights into I/O performance portability.

I/O tracing is very helpful in capturing details of individual I/O functions and allowing in-depth analysis of application performance. Researchers have created many tools for generating application I/O traces, such as RIOT I/O [20], ScalaIOTrace [21], //TRACE [22], IPM [23], LANL-Trace [24], TraceFS [25], and Recorder [9]. After the traces have been generated, they can be used for application debugging, performance tuning, creating benchmark, system analysis or cross-platform studies. For example, the RIOT I/O tracing toolkit has been used to assess the performance of three I/O benchmarks on three platforms with GPFS and Lustre file systems. ScalaIOTrace, //TRACE and

Recorder traces can be replayed to create application-specific benchmarks. I/O tracing provides very detailed information about app executions, which can be extremely useful in improving I/O performance. However, collecting a lot of data increases a tool's overhead and thus limits the potential scale of its deployment. Such I/O tracing tools are ideal for investigating individual runs in full detail, but are too expensive to be used to find broader patterns at the scale of thousands of jobs and apps. Darshan's minimal collection of data allows us to observe a platform at workload scale, and to identify its jobs and applications that can most benefit from follow-up analyses with I/O tracing and other performance analysis tools.

Kim et al. [14] used storage instrumentation to characterize platform workloads. However, storage system instrumentation does not provide application-specific information for analysis. We rely on data captured for a general production workload, which can be used to characterize I/O behavior at both the application and workload levels.

Darshan logs have already been used for system-wide analysis. Carns et al. collected Darshan logs for two months of data on Intrepid [2] and four months on Hopper [26] to explore how the logs can be used to improve storage system utilization and identify candidate applications for additional I/O tuning. We extend this approach to cover three platforms over a much longer period of time. To the best of our knowledge, this is the first study that compares and contrasts application I/O behavior across platforms at full scale.

2.2 Target Platforms and Datasets

We begin this Section with a note on terminology.

In this chapter, a job or run is a particular execution of an application. Unless otherwise noted, we consider two jobs to belong to the same application if and only if their executables have the same name. We have considered an automated clustering of application's name based on string similarity between application names using metrics such as Levenshtein distance (i.e. edit distance) [27]. However, we cannot fully automate this process because sometime application names with smaller edit distance are completely unrelated while ones with larger edit distance belong to the same application. For example, among applications run on Mira, the edit distance between "vasp.bgq"

and "vida.exe" is 4. They are completely different applications: "vasp.bgq" is a Molecular Dynamics code while "vida.exe" is a Flow Simulation code. On the other hand, the edit distance between "AVBP_V7.0_beta.MIRA" and "AVBP_V7.0_beta2Ga_Colin_Med_i8pr_F12_9ed56db6d8.M" is 35 even though they are essentially the same application.

We consider someone who submits a job is a user; users may have to configure an application before they submit a job. Someone responsible for developing the source code of an application is its developer, or rather one of its developers. A widely used application may have a small set of developers and a much bigger set of users. A platform is a particular installation of a supercomputer. Someone responsible for configuring or administering a platform or for helping its users is an administrator of that platform.

Darshan is deployed and enabled by default for all users of ALCF and NERSC platforms as well as BlueWaters supercomputer at the National Center for Supercomputing Applications (NCSA). Edison users automatically see Darshan's I/O summary report on a web page for their completed job. But Darshan does not see every job running on a platform. Applications are not logged if they do not use MPI (e.g., PAMI codes), use non-default build scripts, or run legacy executables that are not already linked to Darshan. Further, an issue in the F90 MPI wrapper on Intrepid and Mira prevents Darshan from observing F90 codes (a fix has been requested from IBM). Users can also choose to disable Darshan, but do not normally do so.

Darshan's minimal collection of data (adding no more than a few seconds to job execution time for most apps [26],[28]) allows it to be enabled for all jobs by default. This allows us to observe a platform at workload scale and to identify its jobs and apps that can most benefit from follow-up analyses with I/O tracing and other performance analysis tools. As Darshan captures all runs of the apps it observes, we can see the patterns of I/O behavior at scale and across platforms, rather than only for selected jobs.

On average, Darshan logs on Intrepid, Mira and Edison cover roughly a third of jobs. The remainder of this chapter considers only those jobs and applications observed by Darshan, and uses the term workload to refer to the platform workload as observed by Darshan. We do not know whether Darshan's observations are typical of the I/O behavior of the unobserved part of the workload; but the observed fraction of the workload is still large enough to interest platform administrators in its own right. Table 2.1 describes

Platform	**Intrepid**	**Mira**	**Edison**
Architecture	BG/P	BG/Q	Cray XC30
Peak Flops	0.557PF	10PF	2.57PF
Memory	80 TB	768 TB	357 TB
Cores per node	4	16	24
Number of nodes	40K	48K	5,576
Number of cores	160K	768K	130K
Storage	6PB	24PB	7.56PB
Peak I/O	88GB/s	240GB/s	168GB/s
File system	GPFS	GPFS	Lustre
Period logged	Jan 2010 - Dec 2013	Apr 2013 - Oct 2014	Jan 2014 - Sep 2014
Jobs logged	239,304	137,311	703,647

Table 2.1: Target platforms and their Darshan datasets

Intrepid, Mira, Edison, and their Darshan logs.

2.2.1 Intrepid

Intrepid is an IBM BlueGene/P supercomputer at ALCF with 40,960 quad-core nodes, 557 TFlops peak performance, and 88 GB/s peak I/O throughput to its GPFS file system. Each set of 64 compute nodes has one of 640 dedicated I/O forwarding nodes (ION). Darshan was enabled from January 2010 until Intrepid was decommissioned in December 2013. In four years, Darshan captured 239K jobs representing over 1K applications, consuming 1405M core-hours, using up to 163K processes and moving as much as 218TB of data in one job.

2.2.2 Mira

Mira is Intrepid's successor at ALCF, an IBM BlueGene/Q supercomputer running GPFS. Mira has 48K 16-core nodes, a peak computing performance 20x faster than Intrepid, and peak I/O throughput 3x faster than Intrepid. Mira has 384 IONs, each serving 128 compute nodes. Mira entered production mode in April 2013, with Darshan enabled. The 137K jobs Darshan observed there consumed 1456M core-hours and used up to 1,048,576 processes, moving as much as 570 TB of data in a job.

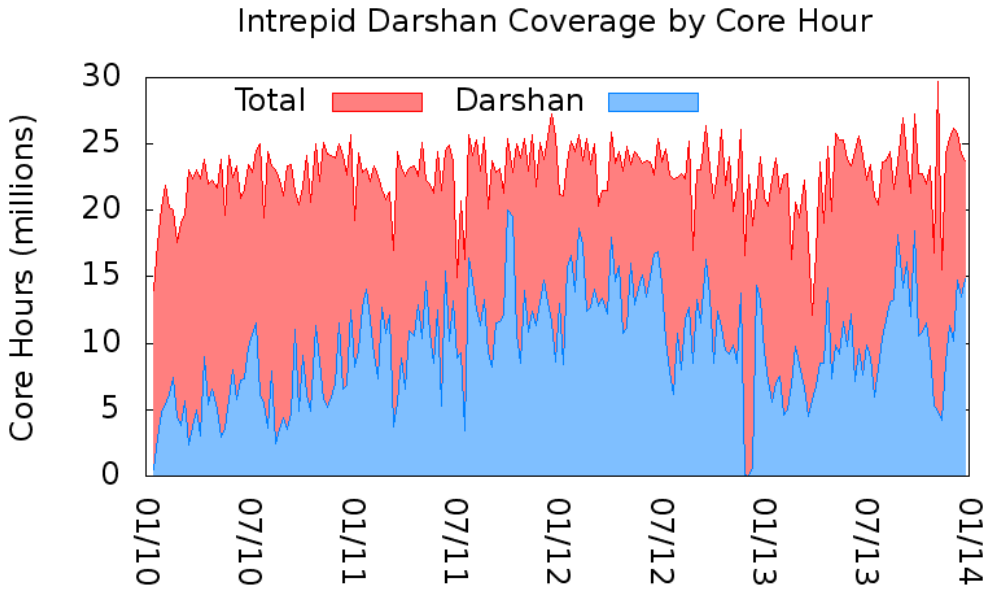


Figure 2.1: Darshan Coverage on Intrepid

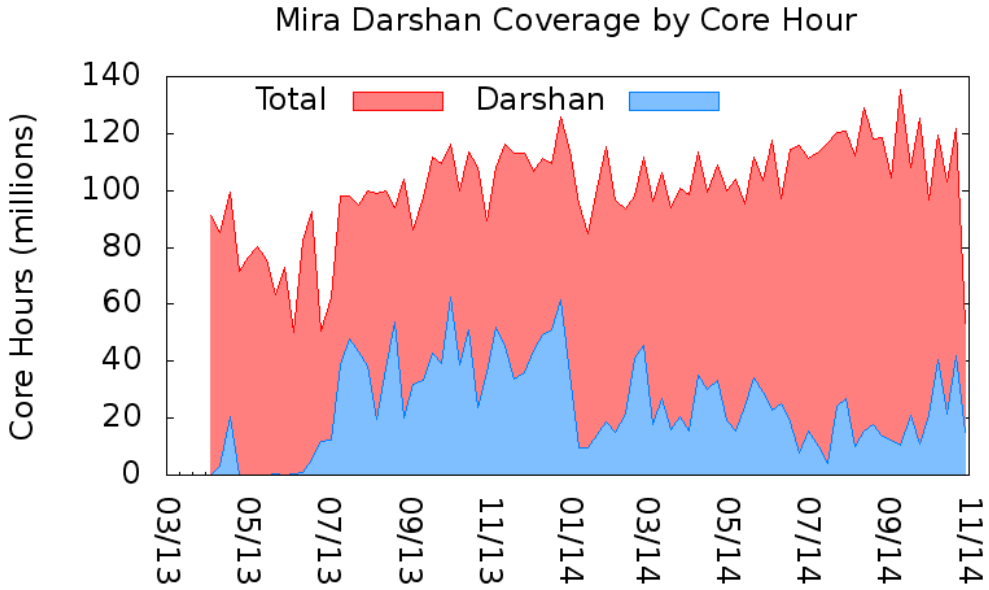


Figure 2.2: Darshan Coverage on Mira

2.2.3 Edison

Edison is the newest supercomputer at NERSC, a Cray XC-30 of size and performance roughly halfway between Intrepid and Mira. Edison has 5,576 24-core nodes and a peak I/O bandwidth of 168 GB/s to its Lustre file system.

We want to note that according to [29], the peak bandwidth is from a file system combined by three Lustre file systems. The peak bandwidth from each file system are 48 GB/s, 48 GB/s and 72 GB/s. Users might need to span their I/O on all three file systems to achieve advertised peak bandwidth (168 GB/s).

The Darshan logs for Edison include 703K jobs consuming 75M core-hours, using up to 131,072 processes and moving as much as 426 TB of data in one job. Darshan coverage on Edison is similar to the other platforms, ranging from 20% to 40%.

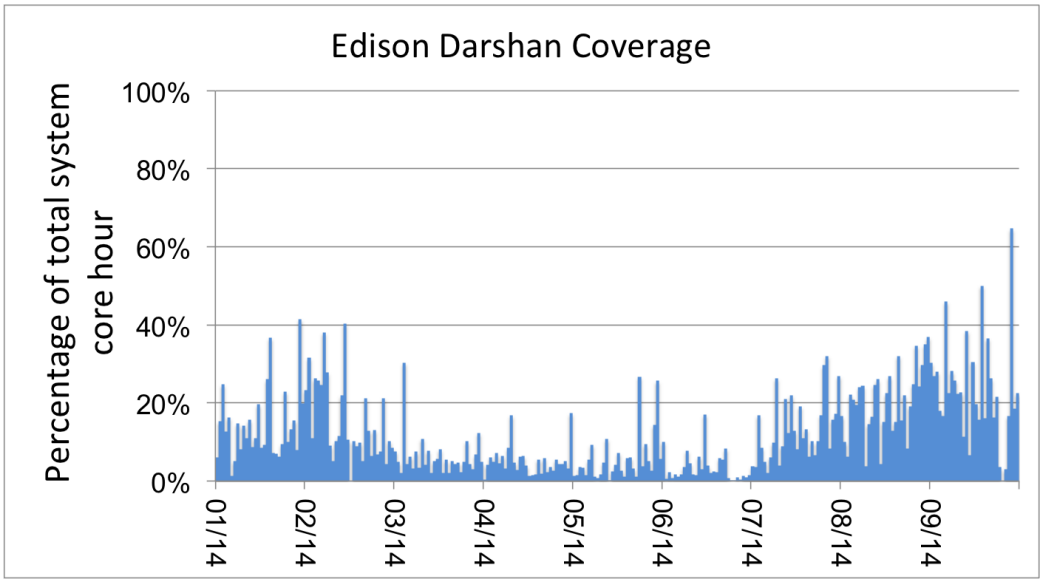


Figure 2.3: Darshan Coverage on Edison

2.2.4 Jobs size across platforms

For these three platforms, Figure 2.4 and Figure 2.5 compare the number of processes per job and the number of bytes that each read or wrote, showing quartiles and outliers in log scale. On all platforms, some jobs ran at full system scale and/or transferred over 100TB. However, most jobs transfer relatively little data, and use few processes compared to the available number of cores. On Edison, 75% of the jobs used less than 100 processes and/or transferred no more than 3GB of data. On Intrepid and Mira, 50% of jobs transferred less than 4GB of data and/or used no more than 2K processes.

Figure 2.6 and Figure 2.7 show that few applications ever run on more than 4K processors or transfer more than a few gigabytes of data.

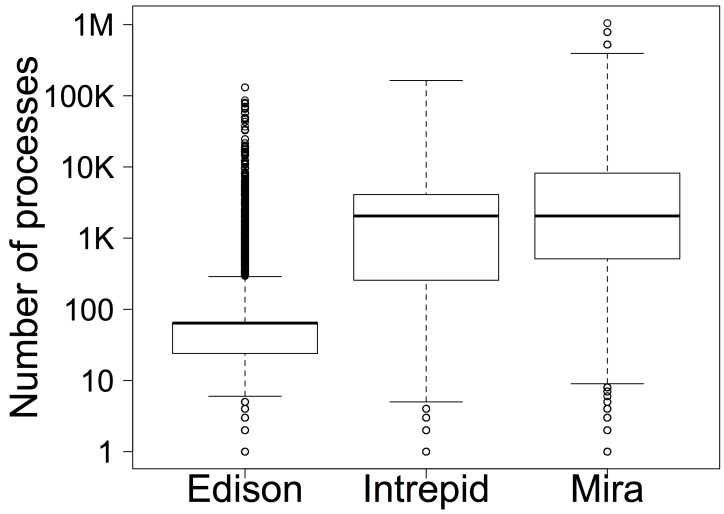


Figure 2.4: Cross-platform comparison of each job's number of processes

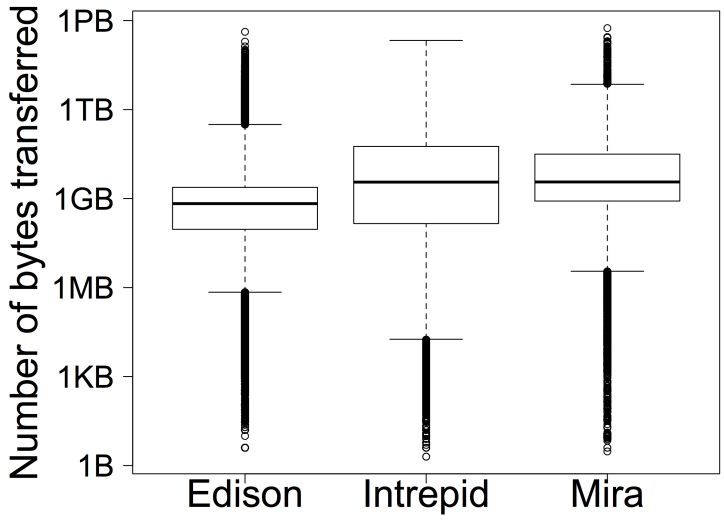


Figure 2.5: Cross-platform comparison of each job's number of bytes read/written

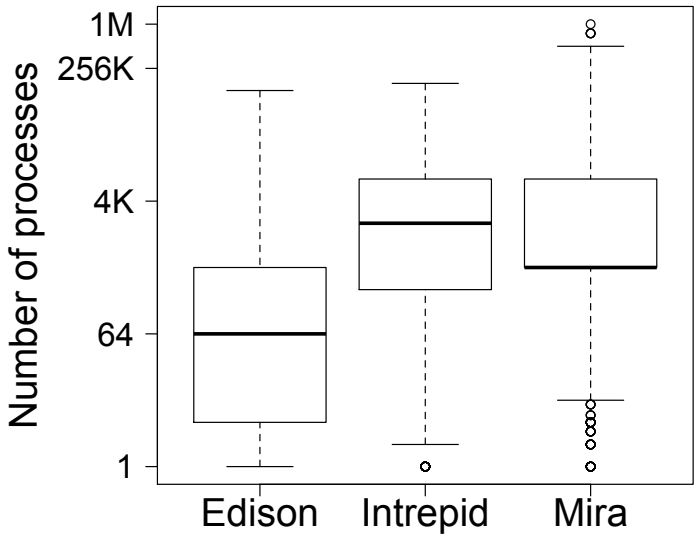


Figure 2.6: Cross-platform comparison of each application's maximum number of processes

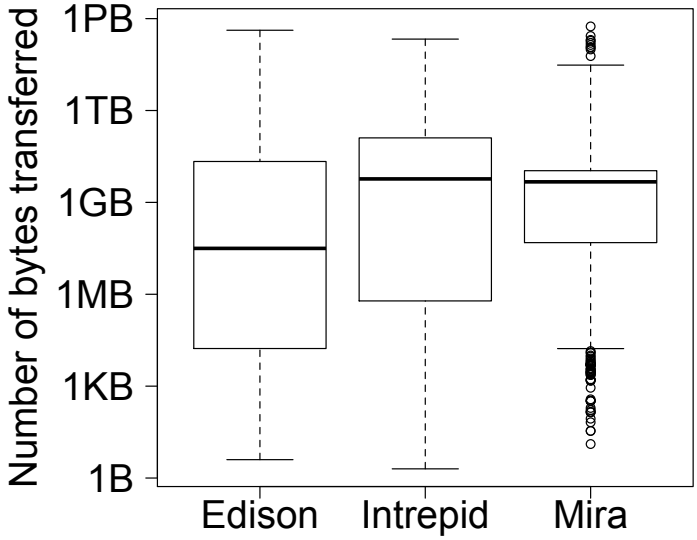


Figure 2.7: Cross-platform comparison of each application's maximum bytes transferred

2.3 Darshan: A Lightweight I/O Characterization Tool

Darshan is a lightweight, highly scalable I/O workload characterization library that can capture application-level I/O behavior with negligible overhead even at production scale. First released in 2009, Darshan is deployed at the large-scale HPC facilities at Argonne, Lawrence Berkeley, Lawrence Livermore, and Los Alamos National Laboratories, the National Center for Supercomputing Applications, and Australia's National Computational Infrastructure, among others. ANL, Blue Waters, and NERSC have automatically enabled Darshan for all users of their facilities, providing broad insight into production HPC workloads.

Darshan does not provide a complete chronological trace of application's I/O functions like traditional tracing or profiling tools. It instead instruments I/O functions at multiple levels and captures key access information for every file opened by an application. These access statistics are aggregated and compressed at the end of an application's executution to create a summary of the application's I/O behaviors. Example recorded characteristics include the amount of data read/written, the number of files opened, cumulative timers, access patterns, common access sizes, operation counters.

Darshan intercepts most of the key POSIX I/O and MPI-IO functions as well as a few HDF5/PnetCDF functions. However, Darshan does not track character-oriented functions such as getc and putc and their higher-level analogs scanf and printf, all intended for text data transfer. This omission was a deliberate decision by the Darshan developers, in light of the need for low overhead and the assumption that users would not and should not spend much time performing character-oriented I/O. The list of I/O functions captured by Darshan is listed in Table 2.2.

Darshan log files are stored in a compressed binary format which can then be parsed into text files using one of the Darshan command line utilities. An example of Darshan text log is listed in Listing 2.1. Information in a Darshan log can be categorized into two categories: per-job data and per-file data. Per-job data contains important information about the whole job such as number of processes, total runtime, total number of bytes transferred, etc. Darshan reports about 30 pieces of overall information about the job. Furthermore, Darshan also captures 162 additional counters for each file opened by a process of the job. Examples of per-file data are count of POSIX

Level	Category	Functions
POSIX	Metadata	creat, creat64, open, open64, close, lseek, lseek64, __fxstat, __fxstat64, __lxstat, __lxstat64,__xstat, __xstat64, mmap, mmap64, fopen, fopen64, fclose, fseek
POSIX	Data transfer	write, read, pread, pread64, pwrite, pwrite64, readv, writev, fread, fwrite, fsync , fdatasync, aio_read, aio_read64, aio_write, aio_write64, aio_return, aio_return64
MPI	Metadata	MPI_File_open, MPI_File_close, MPI_File_set_view
MPI	Data transfer	MPI_File_sync, MPI_File_read, MPI_File_read_at, MPI_File_read_at_all, MPI_File_read_all, MPI_File_read_shared, MPI_File_read_ordered, MPI_File_read_at_all_begin, MPI_File_read_all_begin, MPI_File_read_ordered_begin, MPI_File_iread_at, MPI_File_iread, MPI_File_iread_shared, MPI_File_write, MPI_File_write_at, MPI_File_write_at_all, MPI_File_write_all, MPI_File_write_shared, MPI_File_write_ordered, MPI_File_write_at_all_begin, MPI_File_write_all_begin, MPI_File_write_ordered_begin, MPI_File_iwrite_at, MPI_File_iwrite, MPI_File_iwrite_shared
HDF5/ PnetCDF	Metadata	H5Fclose, H5Fopen, H5Fcreate, ncmpi_create, ncmpi_open, ncmpi_close

Table 2.2: List of I/O functions captured by Darshan

read/write operations, total number of bytes read/written from the file, 4 most common access sizes to the file.

Listing 2.1: Example Darshan log

```
# exe: <executable_name> <args>
# uid: 56599
# jobid: 1637023
# start_time_asci: Thu Aug 28 23:20:42 2014
# end_time_asci: Thu Aug 28 23:38:57 2014
# nprocs: 8192
# run time: 1096
   ..
# performance
# total_bytes: 57383308
# I/O timing for unique files (seconds):
# unique files: slowest_rank_time: 1171.233706
.........
```

2.4 Database Schema

We have collected the Darshan logs of over a million jobs spanning a substantial period of time on three large-scale supercomputers. Manual analysis of the resulting massive number of logs can be an overwhelming task. Therefore, we load the logs into a relational database and use SQL scripts to automate the analysis.

As described in Section 2.3, Darshan log files are stored in a compressed binary format. We utilize a parsing tool, provided in Darshan command line utility collection, to parse the logs into text files. We then use our Python scripts to extract important fields out of the log and load them into the database.

Darshan log contains per-job data, i.e. information about the whole job, and per-file data, i.e. information about each file opened. Since the amount of per-file data correlates to the number of files (which could be millions files), storing per-file data does not scale well. In our current design, we only store per-job data which is divided into three tables: Job_header, Job_perf

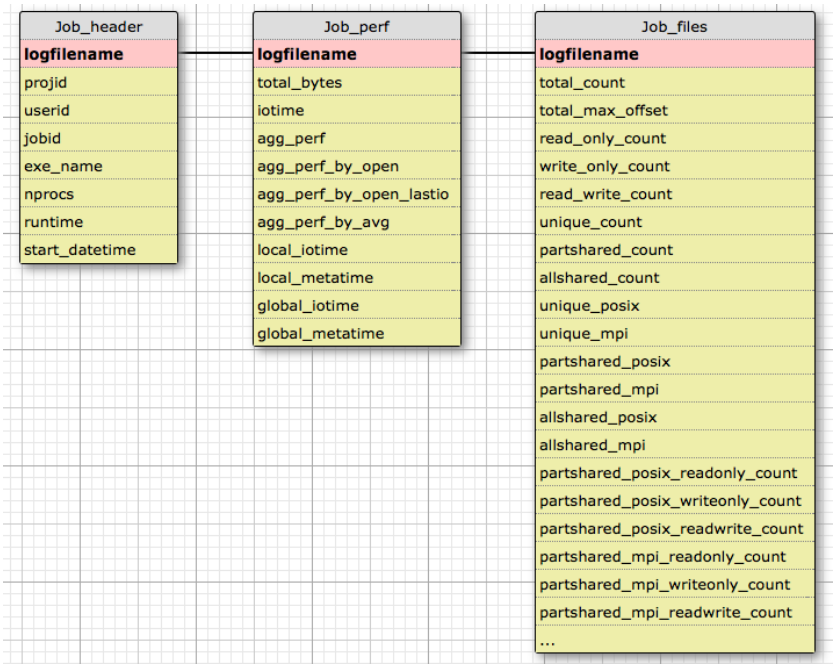


Figure 2.8: Database schema

and Job_files. Our database schema is depicted in Figure 2.8.

When Darshan log is generated, the log file name is created in the following format: USERID_EXENAME_JOBID_DATE_RANDOM#.darshan.gz The log file name is guaranteed to be unique for each job. Therefore, we choose to use it as the primary key in our tables.

For each process of the job, we consider the total time spent in Darshan-tracked POSIX IO or MPI-IO data and metadata function calls for all of the files the process opened. We set the job's I/O time to be the largest I/O time among all of its processes. We compute the job's (aggregate) I/O throughput as its total bytes moved in Darshan-tracked POSIX IO or MPI-IO calls, divided by its I/O time.

Application-level I/O throughput could be computed in other ways, e.g., sum/median/average across processes, but we find the slowest process's viewpoint best for comparing throughput across many jobs/apps. Usually computation does not resume until the slowest process has finished its I/O, due to an explicit barrier or the need to exchange data with neighbors. Thus from the application's point of view, our formula approximates its I/O throughput, and avoids misleading statistics when I/O loads are skewed across processes. We want to note that our definition of job's I/O time does not reflect the file

system's point of view.

Darshan categorizes all the I/O activities that it records into several categories such as:

- Global vs Non-global files: *Global* files are files accessed by all processes. Non-global files are files accessed by a proper subset of the job's processes. These files may be *local*, that is, accessed by a single process; or *subset*, i.e., accessed by multiple processes, such as under a subsetting I/O paradigm.
- Metadata vs Data transfer functions: Metadata functions include open, close, stat and seek functions. Data transfer functions include read, write and sync functions.

Based on these categories, we distribute an application I/O time into 4 sections:

- Global Metadata. All metadata functions for global files. This includes only metadata functions that are called by user and not those called by other functions.
- Non-global Metadata. Metadata functions for files that are not global. This includes only metadata functions that are called by user and not those called by other functions.
- Global Data. Data transfer functions for global files called by user.
- Non-global Data. Data transfer functions for nonglobal files called by user.

A file type can be categorized along three dimensions:

- Scope: Global, local or subset
- Interface: Does it use MPI-IO or POSIX I/O?
- Objective: Is it read only, write only or both read and write?

2.5 Application-Specific Analysis

In this section, we quickly review how to analyze a job's Darshan logs to identify its main I/O bottleneck and inefficiencies. These techniques can be used by application developers and users when Darshan data is delivered to them at the end of a run. We also show how our SQL scripts can be used to examine performance across all runs of an application. Platform administrators can use the same techniques to take a closer look at the applications that are their top users of I/O time (as identified by another set of scripts we wrote). The analysis proceeds as follows.

1. Identify where the job/application spends most of its I/O time, out of four possibilities: Global Metadata, Non-global Metadata, Global Data, Non-global Data.

2. Identify which file(s) consume most of that time. We categorize the files along three dimensions: Scope, Interface and Objective as mentioned in Section 2.4

3. Examine Darshan's performance counters for those files.

As a case study, consider the application that consumed the most I/O time on Mira, an Earth science code we'll call Earth1. Our scripts amalgamated its runs, and found that Earth1 ran about 18K times in 4400 wall-clock hours and consumed 36M core-hours. With Earth1's jobs ordered by their percentage of run time that is not I/O time (light blue), Figure 2.9 divides each job's remaining run time into the four categories listed above. As shown, Earth1 spends over half its time performing I/O, of which the majority is for global file metadata.

To begin Step 2, we examined a randomly-selected Earth1 run. This job used 8192 processes and opened 49193 files. The total wall-clock runtime was 1185s, among which 699s was spent in I/O. It transferred 653 GB of data. Among all the files it opened, 49158 was independently opened using POSIX for read-only and write-only purpose. The majority of total data transferred come from those independent files but that was not where most of the time was spent. This job had 35 global shared files, including 24 using MPI for write-only files, 5 using POSIX for read-only files, and 6 using POSIX for write-only files. As shown in Table 2.3, Earth1 spends most of its time with the 6 POSIX write-only global files.

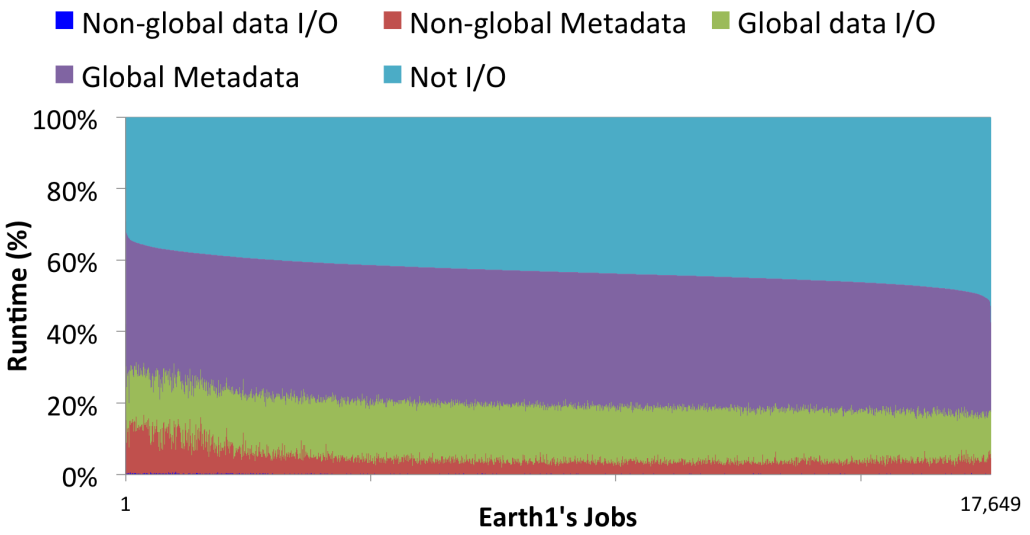


Figure 2.9: Breakdown of total run time for each Earth1 job

Returning to the set of all Earth1 jobs, Figure 2.10 shows how Earth1's I/O time relates to the number of POSIX write-only global files its jobs use, as computed by our scripts. Global data I/O time increases gracefully with the number of files, while global metadata time increases much faster even though the amount of total data transferred differs by a factor of 3 across runs with the same number of POSIX write-only global files as depicted in Figure 2.11. In other words, I/O throughput tracks the changes in file count.

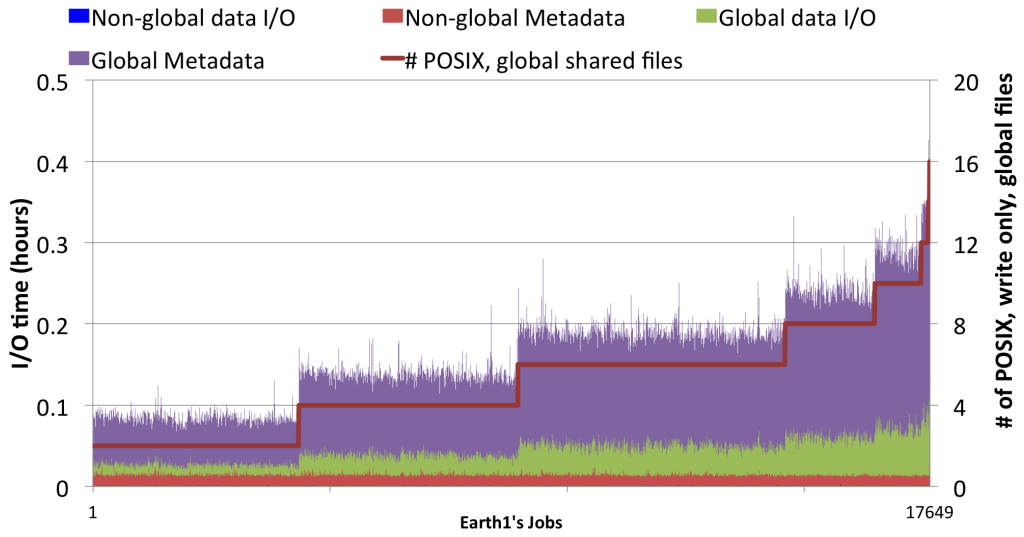


Figure 2.10: Earth1's I/O time and number of POSIX write-only global files (red line)

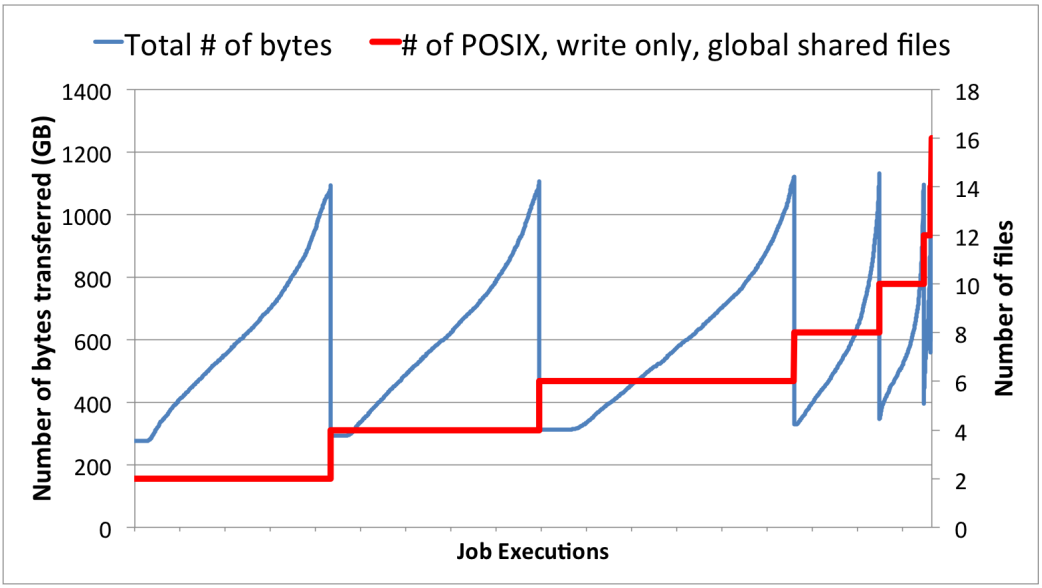


Figure 2.11: Correlation between Earth1's bytes and number of POSIX global files

# of files	File type	Total bytes	Time
49158	POSIX, independently, read only (49152 files) + write only (6 files)	619 GB	103s
35	Global shared	34 GB	
	24 MPI, write only		27s
	5 POSIX, read only		2s
	6 POSIX, write only		567s

Table 2.3: Distribution of time on different file types of Earth1's randomly-selected job

This result indicates that the application developers should take a closer look at the organization of those files.

An I/O expert would quickly notice that according to the per-file log files, each process writes the POSIX global files in relatively small pieces (less than 256KB) that don't align with file block boundaries, making I/O costs high. Common issues of this nature could be included in a checklist for users or automatically recognized.

Job- and application-specific analysis can be done immediately after a run or a series of runs and help the application developer or user quickly locate the main bottleneck, avoiding a long-lasting inefficient I/O implementation. Darshan's data is relatively high level, so developers/users may need to follow up with a tracing or debugging tool; but Darshan can give developers/users

an idea about where their I/O problems may lie. Nevertheless, Darshan captures all runs of the applications it observes, so we can see the patterns of I/O behavior at full scale, rather than only for selected jobs. Traditional tracing tools are ideal for investigating individual runs in full detail, but are too expensive to be used to find broader patterns at the scale of thousands of jobs and applications.

2.6 Platform-wide Analysis

When an application uses shared platform resources inefficiently, it may impact other applications' ability to perform useful scientific work. Platform administrators can use platform-wide analyses to assess job performance, identify large underperforming applications, and offer early intervention to save system resources. In this section, we assess the performance of I/O workloads on Edison, Intrepid, and Mira and propose several criteria for system administrators to quickly identify underperforming applications that consume lots of system resources.

2.6.1 Very low I/O performance is the norm for most applications on these platforms

Even though these supercomputers have very fast file systems with a peak I/O throughput of hundreds of gigabytes per second, few applications experience high I/O throughput.

For each application and platform, Figure 2.12 shows the maximum aggregate I/O throughput reported by Darshan, among all of the application's jobs on that platform. The horizontal lines show the platform's advertised peak I/O bandwidth. (When applications exceed the platform peak, the reason is that their data fits in the file system cache and that reads/writes do not have to access the disk before the function returns.) Aggregate throughput for three-quarters of applications never exceeds 1 GB/s, roughly 1% of average peak platform bandwidth. As noted earlier, most applications are relatively small; and no one should expect a job running on a few nodes to approach peak platform I/O bandwidth. For example, the Mira administrators told us that a 1K-node job cannot expect more than 20 GB/s I/O

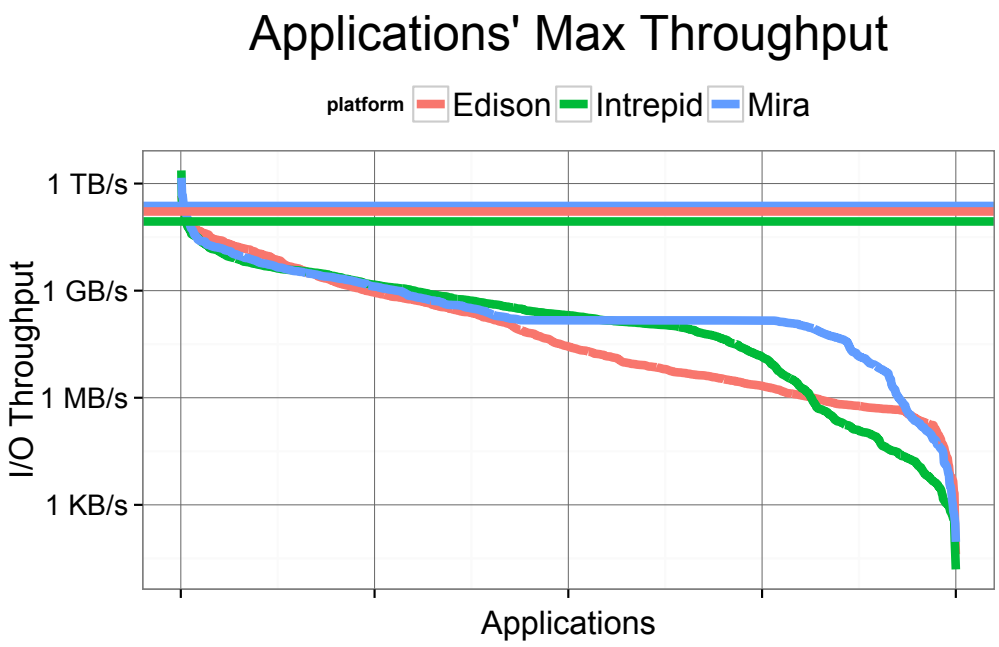


Figure 2.12: Maximum I/O throughput of each application across all its jobs on a platform, and platform peak.

throughput, less than 10% of the platform peak. Looking at the situation another way, however, three-quarters of applications never exceed the aggregate throughput of four modern USB thumb drives (writes average 239 MB/s and reads average 265 MB/s on the 64 GB Lexar P10 USB3.0 [7]).

In Figure 2.13, 2.14 and 2.15, each dot represents one or more jobs of a certain number of bytes and I/O throughput, with the dot color intensity indicating the number of jobs. The figures show that on all platforms, a job's I/O throughput increases roughly linearly with its data size. Jobs that write very little data will not have high I/O throughput, because the fixed costs for accessing a file cannot be amortized across significant data transfer. Still, more than 75% of jobs never achieve one percent of the systems' peak I/O bandwidth.

Each application on a platform is represented by one vertical bar in Figure 2.16, 2.17 and 2.18. Each bar represents all the jobs of that application; the colors of the bar give the breakdown of the total bytes written by each of the jobs. For example, a half-red, half-orange bar means that half the application's jobs accessed over 100 GB, and the other half accessed 10-100 GB (with perhaps a few smaller jobs not visible without magnification). The

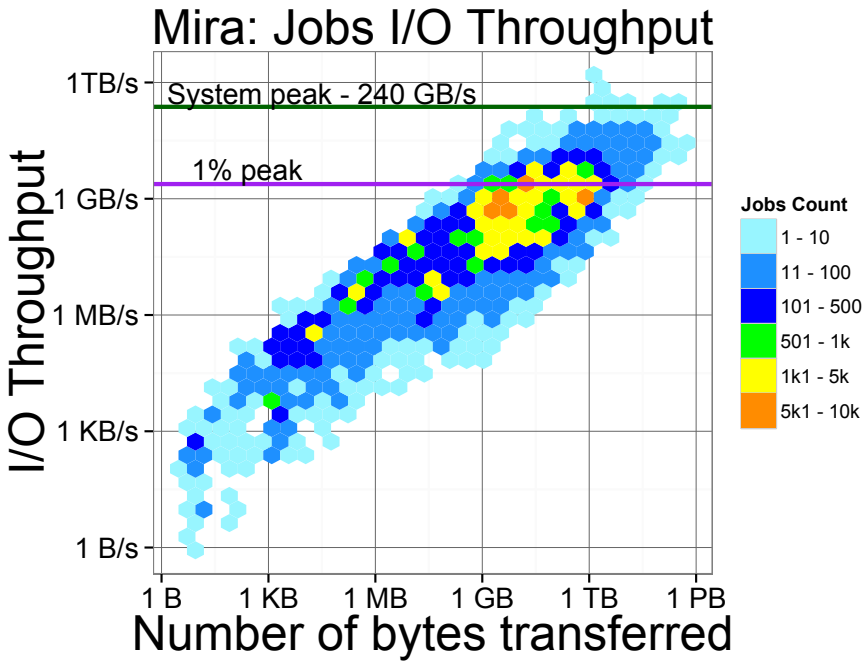


Figure 2.13: Number of jobs with a given I/O throughput and total number of bytes on Mira

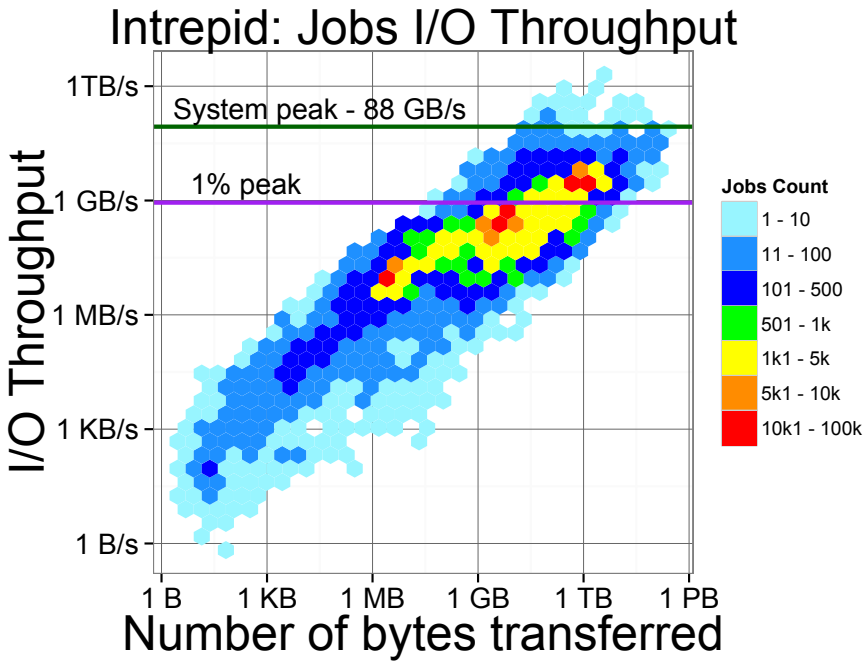


Figure 2.14: Number of jobs with a given I/O throughput and total number of bytes on Intrepid

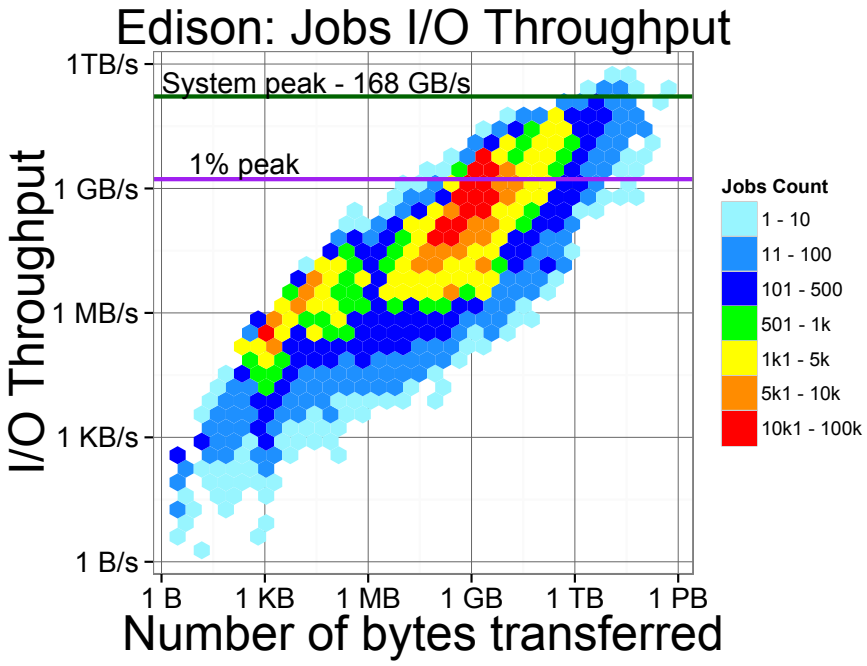


Figure 2.15: Number of jobs with a given I/O throughput and total number of bytes on Edison

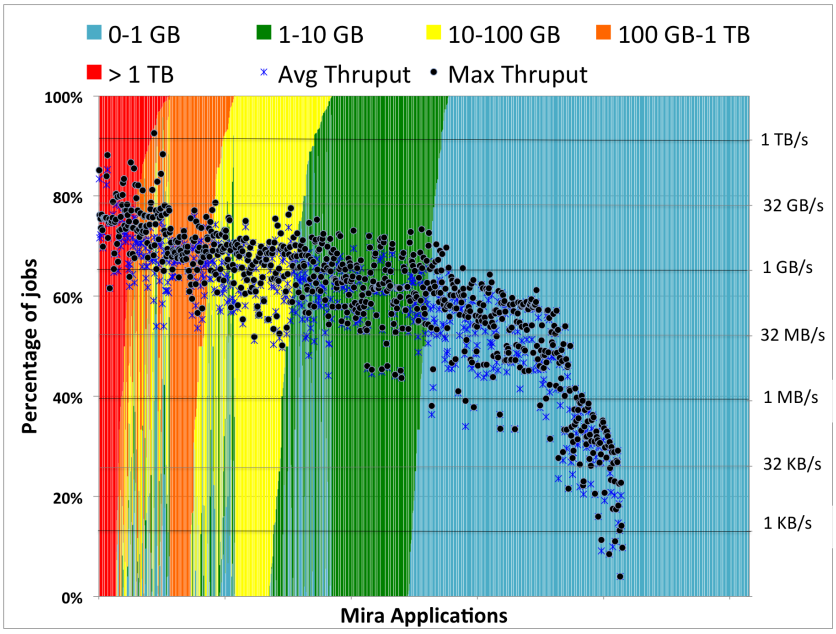


Figure 2.16: Mira: Breakdown of each application's jobs, by bytes written in each job, and average and maximum I/O throughput of each application's jobs.

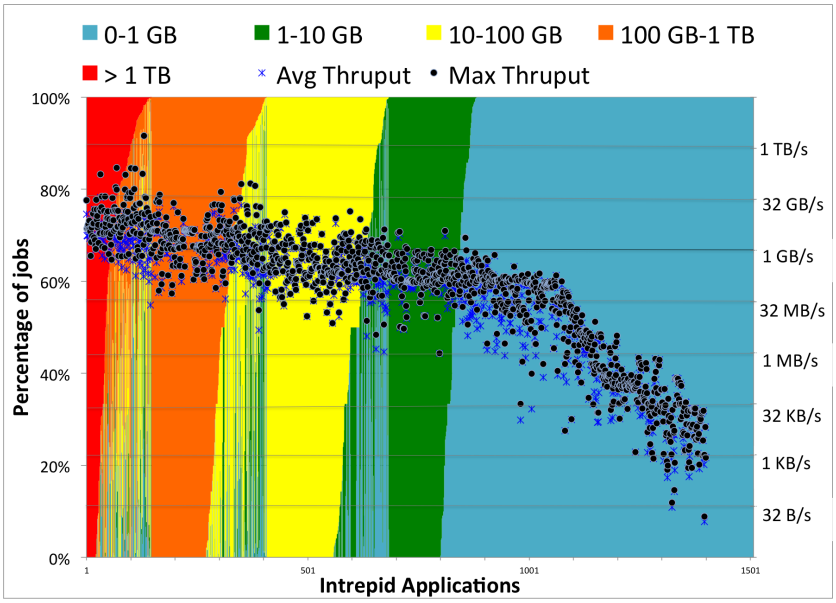


Figure 2.17: Intrepid: Breakdown of each application's jobs, by bytes written in each job, and average and maximum I/O throughput of each application's jobs.

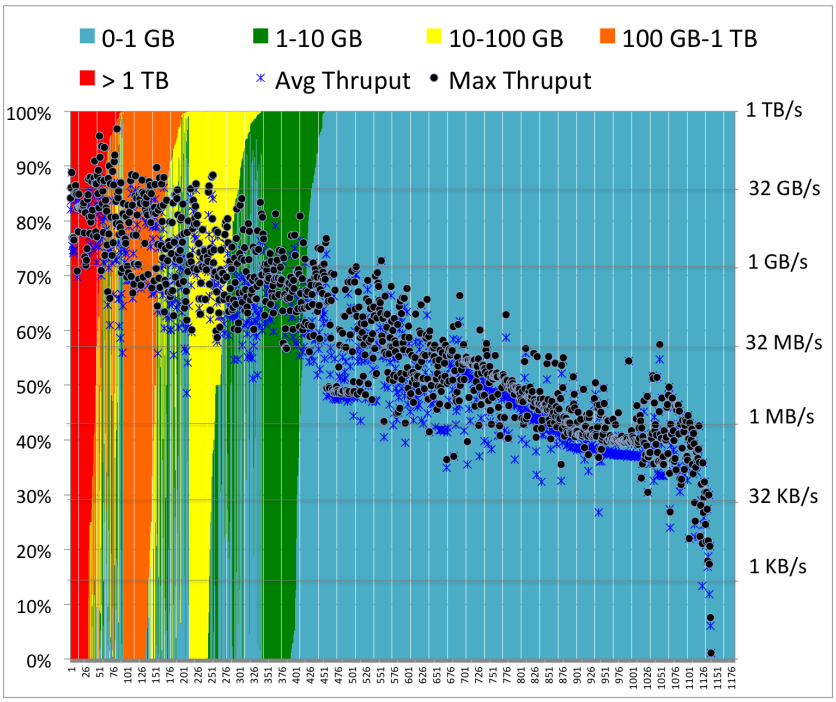


Figure 2.18: Edison: Breakdown of each application's jobs, by bytes written in each job, and average and maximum I/O throughput of each application's jobs.

maximum and average I/O throughput for each application is indicated by circles and crosses, respectively, using the log-scale right-hand axis. The applications are sorted in decreasing order of importance for the storage system, as measured by the total bytes transferred across all the jobs of the application. Figure 2.16, 2.17 and 2.18 show that roughly half of applications do not transfer more than 1 GB of data in their jobs.

Recall that Darshan does not track text-oriented data access functions, so applications that rely entirely on text files will register as having made metadata calls but transferring zero bytes, even if they access a lot of data and therefore are important to the storage system. Along with the applications that perform no I/O (e.g., a hello-world test), these text-only applications can be found at the far right-hand side of each graph, where there is a visible knee in the cloud of throughput dots. As the results indicate, 105 out of 1507, 201 out of 1032, and 42 out of 1183 applications open files but perform no binary I/O in any of their jobs on Intrepid, Mira, and Edison, respectively. Some of these applications are small by any measure, but others are not. For example, a third of the Mira text-only applications had an average job size of at least 1K processes, and a quarter of them averaged 16K or more processes per run. From our case-study analysis of the applications that use the most I/O time on each platform, we know that some applications that heavily rely on text files also access binary files, so the counts listed above understate the extent of the usage of text files. Since we do not know how many bytes of text each application accesses, Figure 2.16, 2.17 and 2.18 also understates the importance and impact on the storage system of text-based I/O. Since text-based I/O generally does not scale up well, we conclude that text-based I/O is a more widespread practice than previously thought and deserves further investigation.

I/O throughput for small jobs and applications does not matter, in the sense that users and developers will be happy as long as total I/O time is only a second or so per job. But small jobs may be test runs for large jobs from the same application, such as the many Mira jobs in Figure 2.13 that transfer a terabyte of data and spend 10-20 minutes in I/O (with the throughput of about 1 GB/s). Thus, small jobs may offer a chance to identify poor I/O practices before significant amounts of platform and user time have been wasted. Further, an application consisting entirely of relatively small jobs can still be a top user of I/O time on a platform. We consider these

two points in the following discussions, which focus on applications that are heavy users of I/O time.

2.6.2 Platform I/O resource usage is dominated by a small number of jobs and applications

On Edison, Intrepid, and Mira, the total I/O time consumed by all the jobs observed by Darshan is 5,920 hours, 13,052 hours, and 5,335 hours, respectively. With the jobs sorted by their total I/O time, Figure 2.19 shows the cumulative portion of platform I/O time that they use. On Edison, the top 10% of the jobs consume 90% of the I/O time. On Intrepid and Mira, the top 25% of jobs consume 90% of the I/O time. The curve is even steeper for applications : 90% of I/O time goes to under 4% of the applications on Intrepid, 3% on Mira, and 6% on Edison; each platform has approximately 1K - 1.5K applications. These results echo the findings of [2], in which a single application dominated I/O time usage in a two-month study.

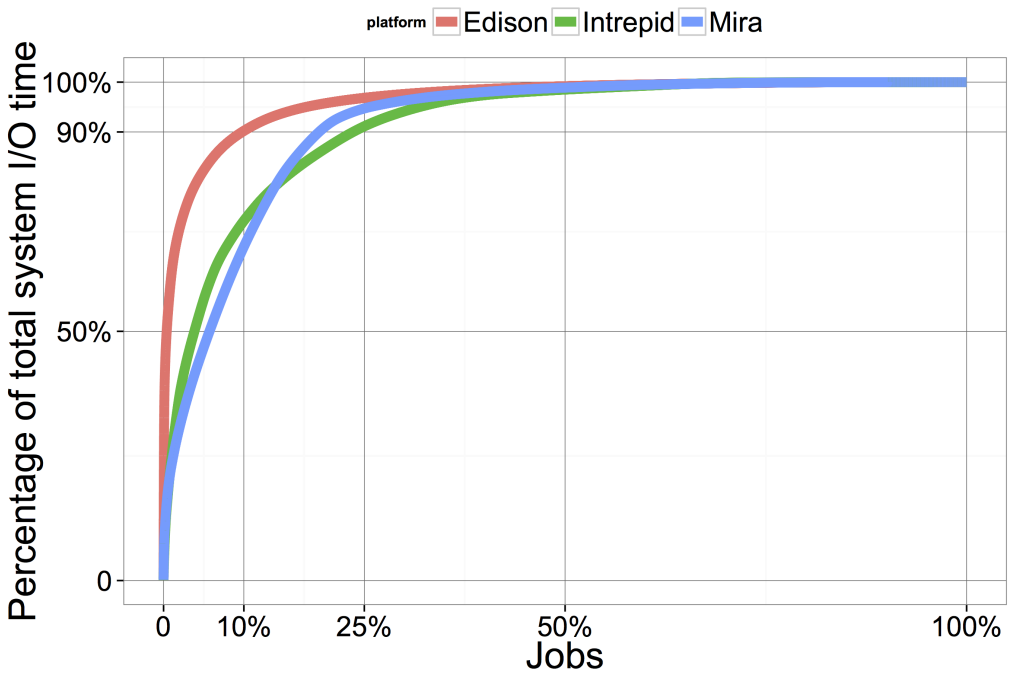


Figure 2.19: Cumulative percentage of platform I/O time consumed by jobs on Edison, Intrepid and Mira

Rank	Application	Total I/O time (h)	Total run time (h)	Number of jobs	Total bytes (TB)	Median I/O throughput (GB/s)	Runtime I/O %
1	**Earth1**	2,480	4,406	17,649	10,037	1.205	56%
2	**Materials1**	577	22,912	4,579	196	.103	3%
3	**Turbulence1**	428	4,121	972	153	.123	10%
4	**Physics1**	150	3,387	762	1,051	.475	4%
5	**Physics2**	133	6,262	1,966	1,115	.467	2%
6	Climate1	95	2,039	1,520	112	.291	5%
7	Molecular1	89	27,826	19,622	156	.571	0%
8	Turbulence2	83	671	335	251	.212	12%
9	Turbulence3	74	96	323	1,961	1.700	77%
10	Physics3	67	202	66	51	3.274	33%
11	Molecular2	67	1,686	2,480	34	.167	4%
12	PDE1	62	120	298	1	.098	52%
13	Plasma1	48	934	58	3,052	18.320	5%
14	Physics4	42	202	309	90	.186	21%
15	Aero1	41	61	151	359	2.505	67%

Table 2.4: Mira's 15 Applications with Biggest I/O Time

Rank	Application	Total I/O time (h)	Total run time (h)	Number of jobs	Total bytes (TB)	Median I/O throughput (GB/s)	Runtime I/O %
1	Materials2	1,109	3,397	847	60	.016	33%
2	Materials3	505	7,329	78,302	10,351	.475	7%
3	Physics5	395	2,698	2,171	6	.005	15%
4	Physics6	322	3,353	6,687	15	.010	10%
5	Materials4	263	8,252	1,231	17	.038	3%
6	Molecular3	249	7,392	2,194	51	.036	3%
7	**Materials1**	219	11,671	16,221	44	.109	3%
8	Materials5	215	21,439	34,213	27	.061	1%
9	Materials6	213	983	926	16	.070	22%
10	Chem1	145	18,909	5,412	4	.013	1%
11	Materials7	129	453	5,769	18	.039	29%
12	**Weather1**	110	686	299	1,189	.660	16%
13	Materials8	103	1,011	1,383	2,477	7.993	10%
14	Materials9	93	175	12,344	266	.860	53%
15	Plasma2	89	102	41	246	2.265	87%

Table 2.5: Edison's 15 Applications with Biggest I/O Time

Rank	Application	Total I/O time (h)	Total run time (h)	Number of jobs	Total bytes (TB)	Median I/O throughput (GB/s)	Run-time I/O %
1	Earth3	1973	2,457	6,363	3,658	.546	80%
2	Climate2	1,761	19,262	3,140	11,604	2.085	9%
3	**Weather1**	1,012	12,723	1,262	653	.160	8%
4	**Earth1**	898	1,738	13,256	13,737	4.521	52%
5	**Physics1**	869	13,687	2,778	2,394	.434	6%
6	Earth4	521	895	8,897	65	.040	58%
7	Physics7	505	6,277	7,643	5,070	2.908	8%
8	Earth5	455	714	4,288	131	.072	64%
9	Earth6	451	522	5,150	2,878	1.907	86%
10	Earth7	424	482	5,194	2,924	2.459	88%
11	Physics8	212	1,691	670	44	.226	13%
12	Molecular4	198	2,168	260	3,002	.087	9%
13	**Physics2**	152	9,294	5,543	600	.472	2%
14	**Turbulence1**	149	2,178	253	308	.479	7%
15	Physics9	137	1,170	916	809	.220	12%

Table 2.6: Intrepid's 15 Applications with Biggest I/O Time

Let us look at these applications more closely. Table 2.4, Table 2.5 and Table 2.6 show the biggest 15 applications on Mira, Edison and Intrepid, in terms of total I/O time summed across all their jobs. In the remainder of the chapter, we refer to these as the big-time apps. Materials1, Turbulent1, and Molecular1 each merge two apps with near-identical executable names. We consider apps to be the same across platforms if their executables names differ at most in version numbers. Applications whose names are in bold are on the big-time list for multiple platforms. Keep in mind that Darshan is not configured to observe data accesses using character-oriented I/O, so the I/O time for text-file-based applications is undercounted when picking out the big-time apps. The top 15 big-time apps account for 83% of I/O time on Mira, 70% on Edison, and 73% on Intrepid. The total data read/written across all their jobs varies from a high of 10 PB for Earth1 and Materials3 to a low of 1 TB for PDE1.

By focusing their attention on improving the performance of the big-time applications, platform administrators may be able to free up resources for others to use and improve the satisfaction of all users. This principle drives the attention given to important applications and their I/O benchmarks; and indeed, the I/O behavior of at least five of the applications in Table 2.4 and three in Table 2.5 is well studied and carefully tuned. However, applications with I/O bugs and with I/O paradigms that are suboptimal for their situation also appear in the tables. For example, as discussed elsewhere in this chapter, PDE1 used global text files with a large number of processes, and Earth1 was held back by relatively small POSIX writes to global files. Indeed, the applications in these tables are top in usage of I/O time, not top in terms of I/O throughput. Applications that are extremely successful in extracting I/O performance will not be listed in the tables unless their total data size is incredibly high.

In the tables, the percentage of run time that big-time applications devote to I/O rises from almost 0% for Molecular1 on Mira to 87% for Plasma2 on Edison. The developers and users of the applications at the lower end of this range are likely to be happy with their I/O throughput, even if the platform administrators are not. **Boosting the minimum aggregate throughput for all of the big-time apps to 1 GB/s (equivalent to four USB thumb drives) would save the platform administrators 42% of the total I/O time on Intrepid** (3758 hour out of 8920), **41% on Mira** (1803

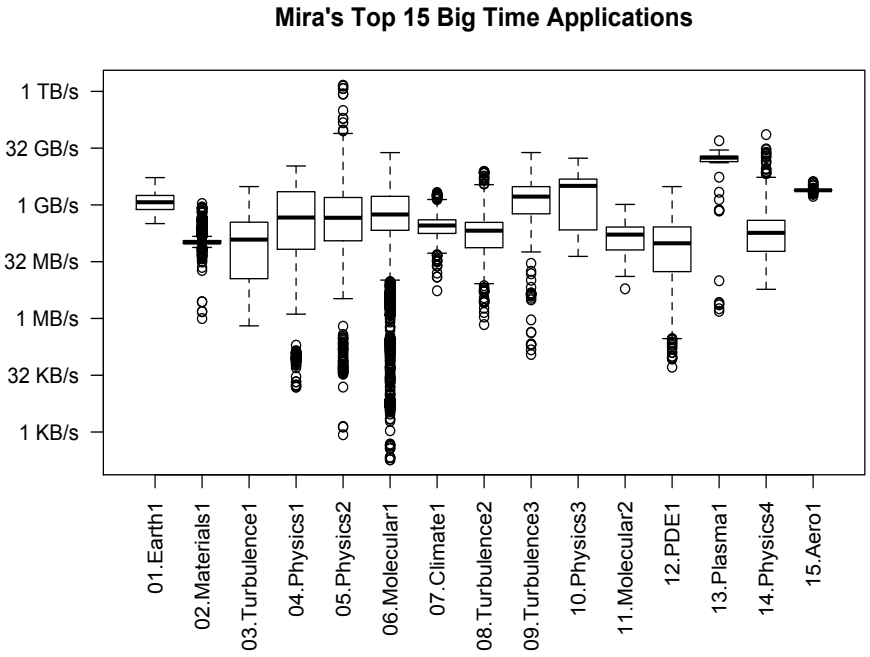


Figure 2.20: I/O throughput of top 15 big-time applications on Mira

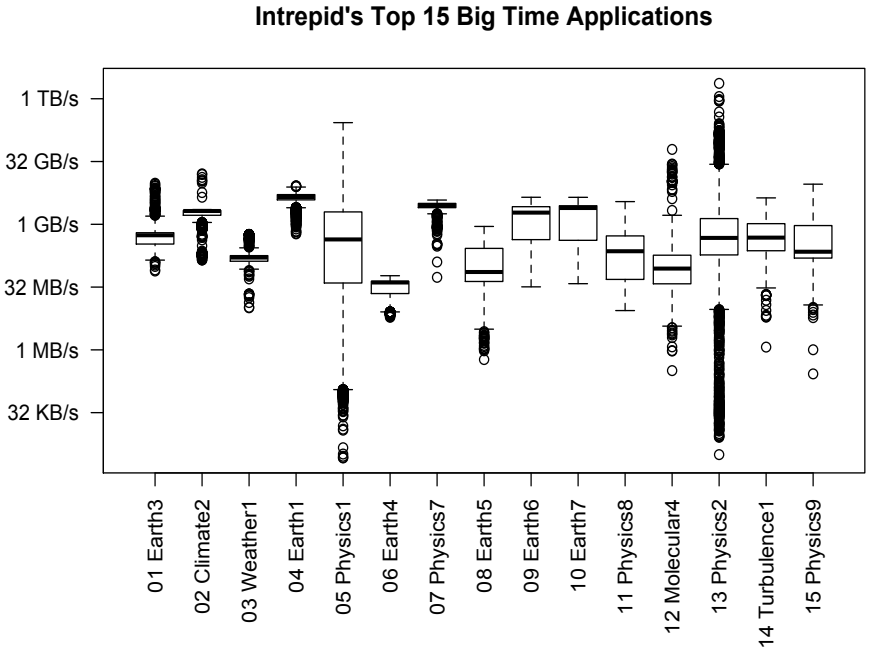


Figure 2.21: I/O throughput of top 15 big-time applications on Intrepid

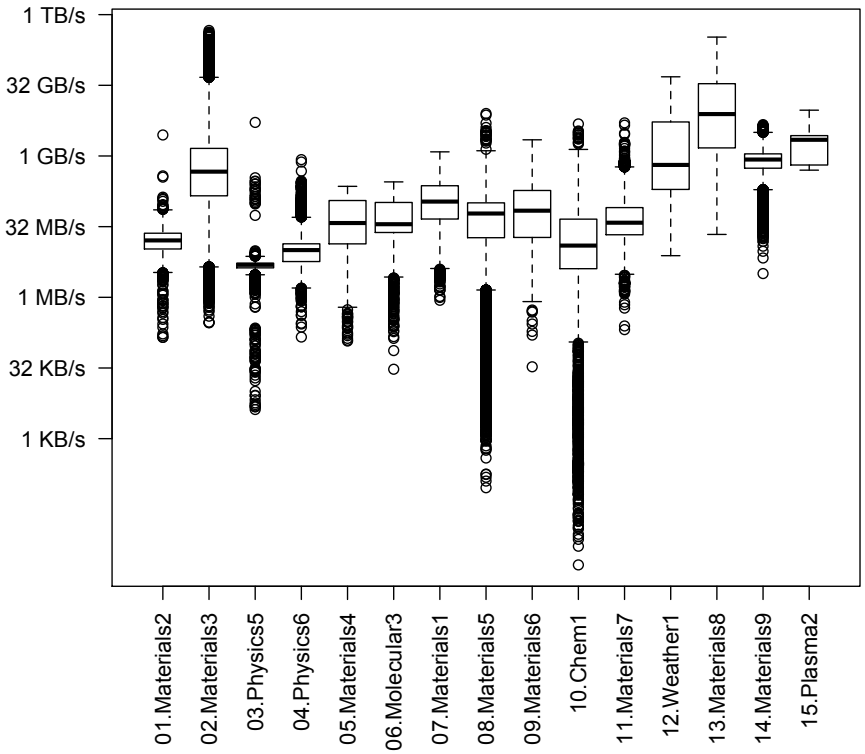


Figure 2.22: I/O throughput of top 15 big-time applications on Edison

hours out of 4435), **and 85% on Edison** (3542 hours out of 4158). The jobs running concurrently with big-time applications could also benefit from the increased availability of I/O resources.

According to the tables, less than a quarter of Edison's and Mira's big-time applications get more than 1 GB/s I/O throughput in their median job; only one gets more than 10 GB/s in its median job (Plasma1, 18 GB/s on Mira). Figure 2.20, 2.21 and 2.22 show the quartiles and outliers for the I/O throughput of the big-time applications' jobs on Mira and Edison. As was true for the set of all jobs, big-time applications' jobs get better I/O throughput when they have more data. Figure 2.23 shows this with a four-category breakdown of the big-time applications' performance, based on whether they have small data usage (read/write under 10 GB) and/or small numbers of processes (less than 2K). Figure 2.23 shows that most big-time applications' jobs with big data and processes get 1-16 GB/s of throughput

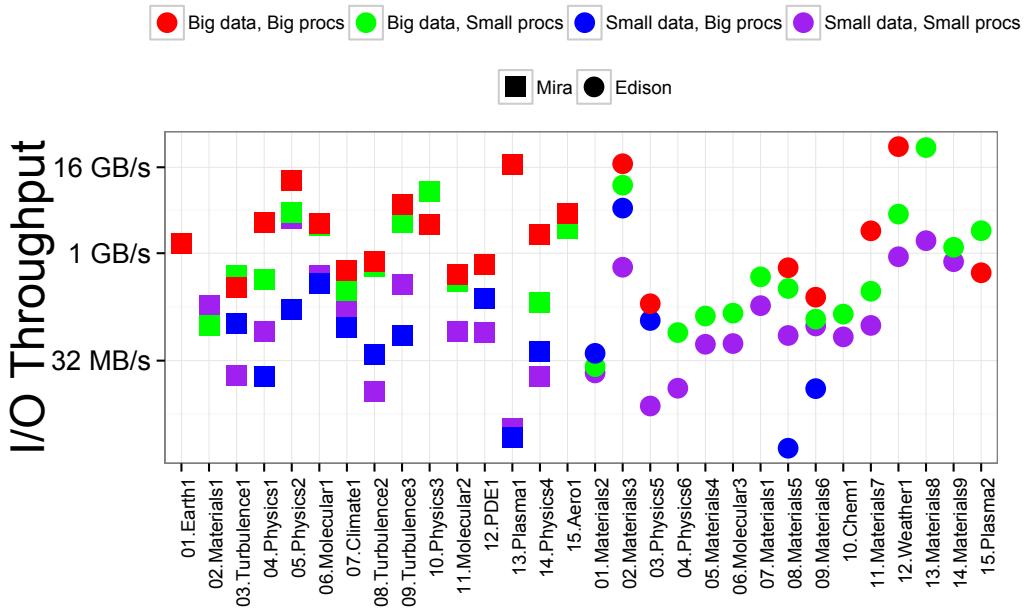


Figure 2.23: Average I/O throughput of Mira's (squares) and Edison's (circles) big-time applications' jobs, by job size

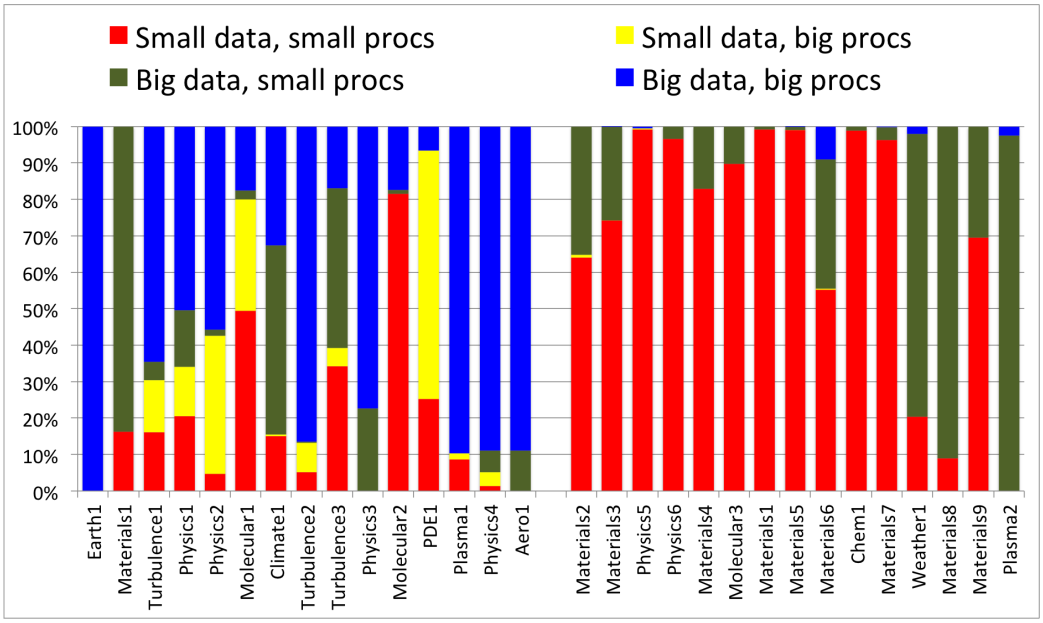


Figure 2.24: Job sizes for Mira's (left) and Edison's (right) big-time applications

on Mira. As we will see later, each platform has applications with much higher median-job I/O throughput than the big-time applications.

We believe that all interested stakeholders - application users, application developers and platform administrators can benefit from the results of observation and analysis of the I/O behavior of HPC applications. The most important applications to bring to attention are those that are bad for the user and bad for the system, because incentives are aligned for the platform administrators to want them to change and for the user to fix the problem themselves. Those applications can be identified, for example, as ones in top big-time applications with high I/O percentage.

2.6.3 Early intervention by platform administrators can identify applications with I/O problems, save I/O resources, and improve user satisfaction

Table 2.4, Table 2.5 and Table 2.6 show that most of the big-time applications on Mira and Edison ran over a thousand times, and all but three ran over a hundred times. Clearly, early intervention where needed could have saved a huge amount of system resources. As shown in Figure 2.24, almost all big-time applications have small jobs, especially on Edison, which is the newest platform; first runs where the problem occurs are the ideal point for identifying and addressing problems.

For example, PDE1 in Table 2.4 used over 13 million core-hours on Mira and spent 87% of its run time in I/O. When PDE1 ran at scale (64K-128K processes) in its original implementation, I/O activities consumed almost all of its run time. For example, one job with 512K processes took 7 hours and over 3.5 million core-hours. Figure 2.25 includes a stacked bar for each successive PDE1 job, breaking down its total run time; the 7-hour run is excluded because it is off the chart. The clump of blue bars in Figure 2.25 shows that in its early runs at scale, PDE1's I/O time was devoted to metadata functions; in fact, the data transfer time for most files was zero. This signals that the files are being read/written with functions not tracked by Darshan, namely, character-oriented functions for text files.

Conversations with PDE1's developers confirmed that the initial implementation used fprintf to write to the output text file accessed by all processes. After PDE1's developers attended the Mira performance boot camp and learned about MPI-IO, the developers created an MPI-IO-based im-

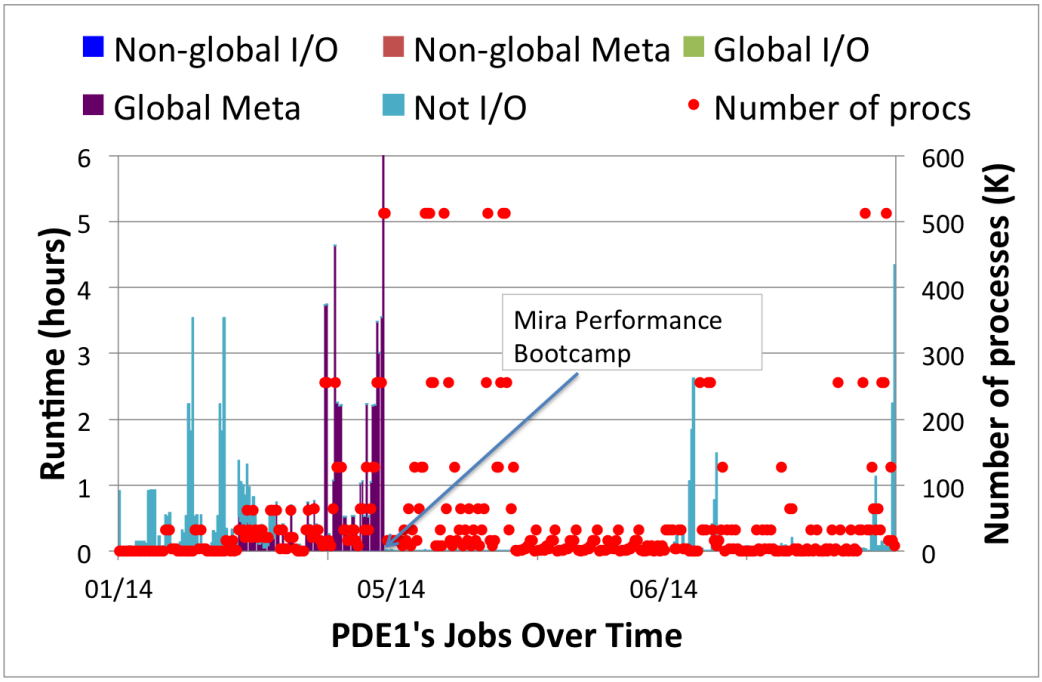


Figure 2.25: Evolution of PDE1's I/O paradigms. Red dots show the number of processes of each job (right-hand y-axis).

plementation that runs in 11 seconds with 512K processes. PDE1's developers would have benefited from automated analysis of the Darshan logs from its early small jobs in Figure 2.25. Without extending Darshan to track character-oriented I/O functions, a script can still find applications that make inappropriate use of text files, by searching the logs for instances of files with high metadata time and zero data access time.

The logs also show how application I/O behavior evolves over time. PDE1's earliest runs used few processes, so its I/O paradigm was inexpensive relative to computation. As the number of processes increased, I/O immediately began to dominate (purple bars). The developers' change to MPI-IO is marked by the disappearance of the purple bars.

As another example, consider Earth2, an Earth science code that ran for 60 hours wall time on Mira and consumed about 100K core hours. It read from hundreds of thousands to over a million files and spent the vast majority of its time in I/O, as shown in Figure 2.26. Its I/O time breakdown reveals the tell-tale pattern of text files: high metadata time and zero data access time. Later, its owners identified a bug that put their read operations inside an unrelated nested loop, rather than outside. This costly bug persisted for

a long time before it was noticed. The situation is another argument for automated early intervention.

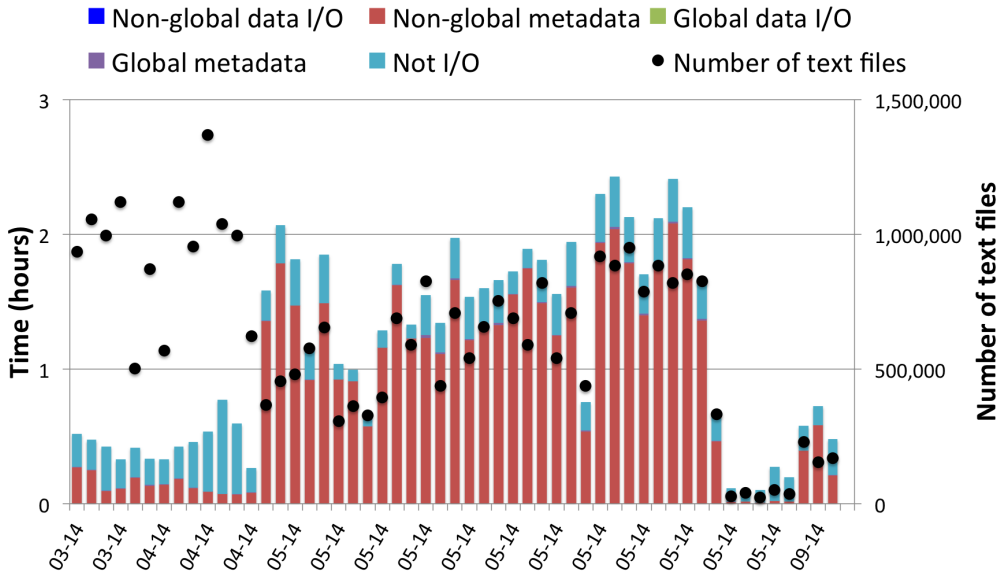


Figure 2.26: Earth2 read hundreds of thousands of text files

We suggest the following four criteria to help platform owners identify apps whose I/O behavior makes them candidates for further investigation. The criteria are not absolute indicators of I/O problems, but rather help to narrow down the number of applications to consider.

- Applications using a text file I/O approach, such as PDE1 and Earth2. A query for jobs that use only text files finds 2121 jobs from 59 apps on Edison, 5561 jobs from 237 apps on Mira and 4725 jobs from 171 apps on Intrepid.
- Applications with many files and high metadata costs. For example, a query for Mira jobs with over 100k files and metadata time that is more than one third of run time finds 111 jobs from 11 apps, including Physics4 (discussed in Section 2.6.6).
- Apps with little data but large I/O time. For example, on Edison, a query for jobs with under 4 GB of data that spend over 5 minutes in I/O finds 4020 jobs from 79 apps. One of the apps has more than 500 jobs that match this criterion.
- Big time apps, such as the Top 15 discussed earlier.

The filtering capability further emphasizes the importance of having a central database about system workload that will enable early intervention from platform owners to save system resources and improve system utilization.

2.6.4 POSIX I/O is far more widely used than parallel I/O libraries

The HPC community has worked hard to create a stack of parallel I/O libraries, including MPI-IO, HDF5, and NetCDF. But Figure 2.27 shows that users tend to stick with the POSIX I/O library (open, read, write). Nearly 95% of jobs visible to Darshan on Edison use POSIX exclusively. On Intrepid and Mira, the percentages are 80% and 50%, respectively. The remaining jobs use MPI-IO directly or use the libraries built atop MPI-IO (e.g., HDF5), for at least one of their files. MPI-IO is used most often among mid-sized jobs, in terms of their number of processes.

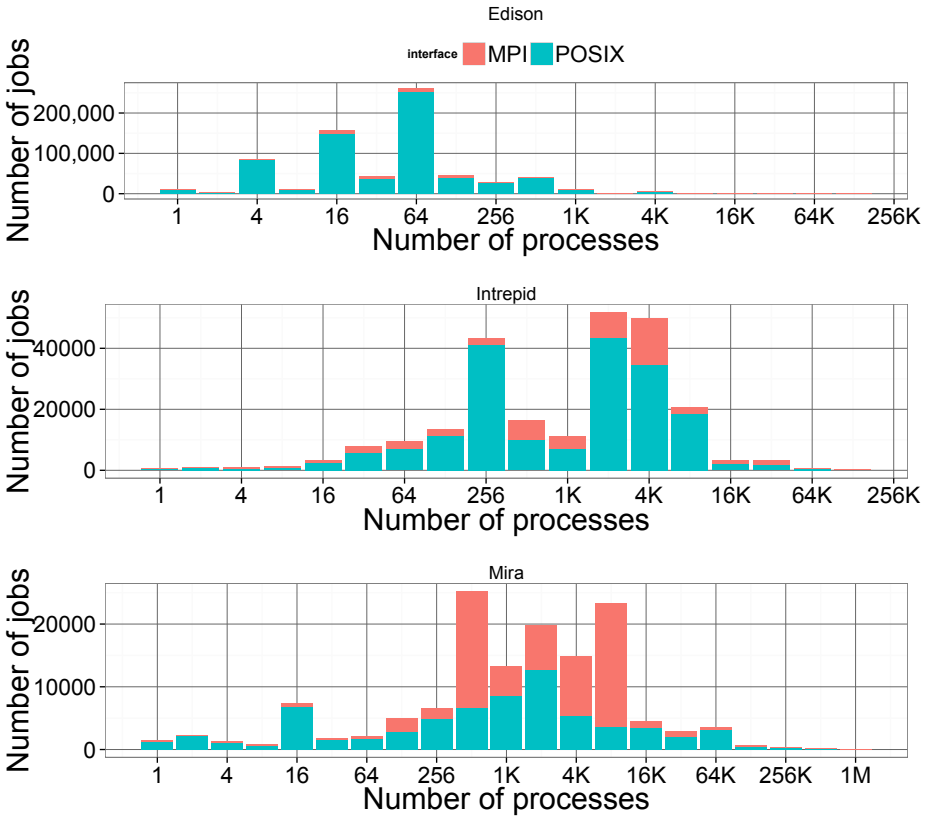


Figure 2.27: Number of jobs using POSIX IO only (teal) and using MPI-IO directly or indirectly for at least one file (red)

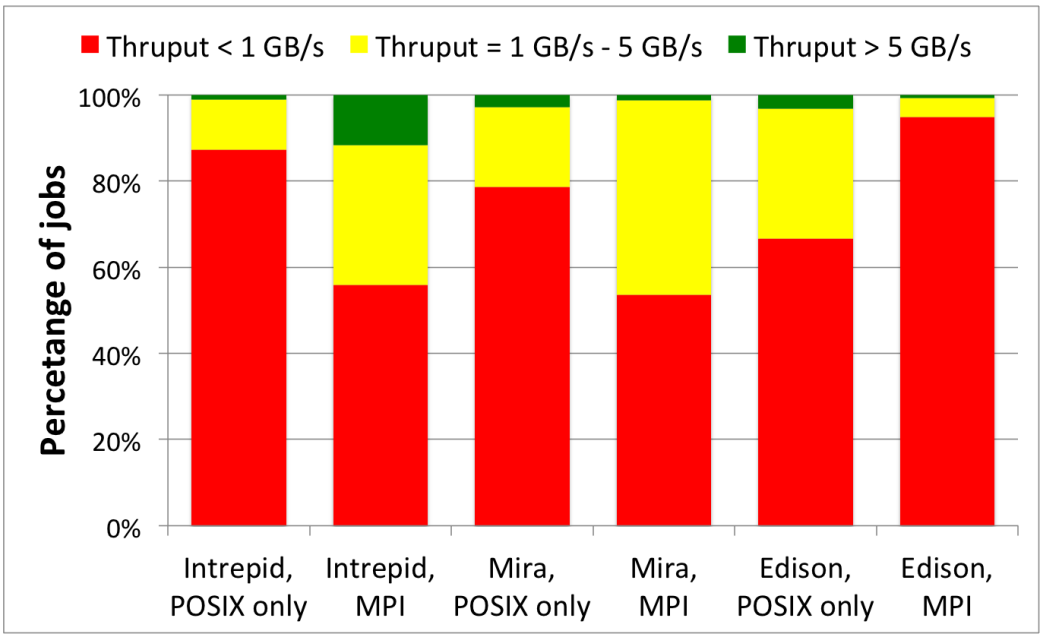


Figure 2.28: I/O throughput for applications that use only POSIX-IO and those that directly or indirectly use MPI-IO for at least one file

The POSIX-only approach does not necessarily mean low I/O performance; with careful implementation and tuning, POSIX applications can achieve high I/O bandwidth. However, using MPI-IO offers more chances for decent I/O performance. As shown in Figure 2.28, on Mira and Intrepid, about 45% of jobs that used the MPI-IO library achieve more than 1 GB/s of aggregate I/O throughput (four thumb drives), while less than 20% of the POSIX-only jobs reach 1 GB/s. On Edison, most applications that used MPI-IO do not do so efficiently, although some have excellent throughput. We return to this point in our cross-platform analysis.

Carns et al. [2] analyzed the usage of different I/O interfaces and found that most jobs with few processors use POSIX for I/O, whereas jobs with many processors use POSIX primarily for reads, if at all. MPI-IO prevailed among jobs with many processors and applications that wrote more data than they read. We found that, in addition, POSIX is popular among many-processor jobs. This result agrees with another study in [30].

2.6.5 Metadata costs often exceed data I/O costs

Metadata costs are a major factor in the I/O throughput of applications [26]. Averaging across the platforms in Figure 2.29, roughly 40% of jobs spend more time in metadata functions than in reading and writing data. We have already touched on a variety of reasons for this problem: the prevalence of small-data jobs and applications, which Figure 2.29 highlights; the hidden problem of overreliance on text files; and small data request sizes.

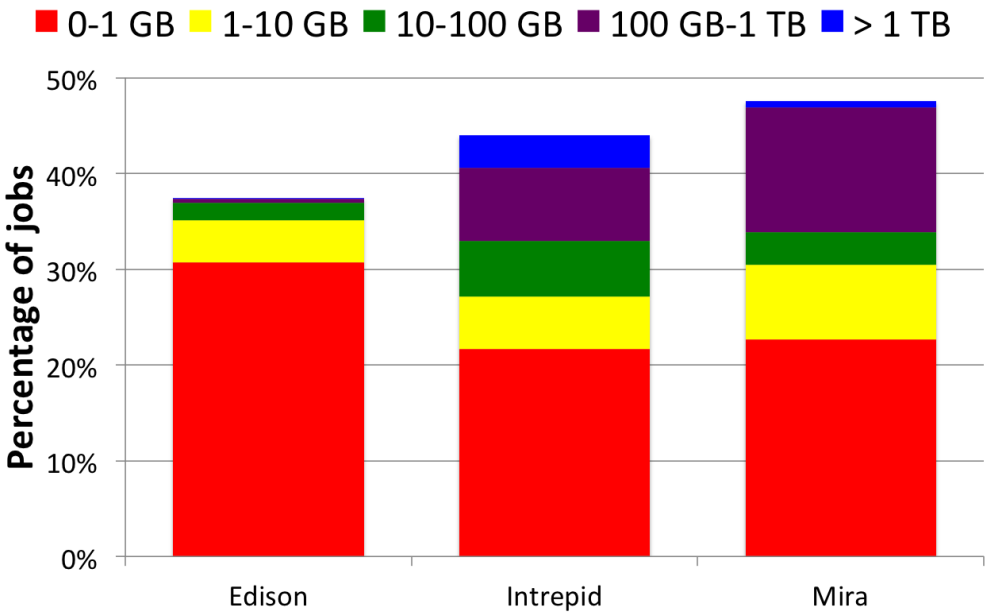


Figure 2.29: Jobs with more metadata time than data transferring time: Breakdown of jobs by total data size

2.6.6 No major I/O paradigm is always good or bad

Text files are the only I/O approach that almost guarantees poor performance at scale; we do not consider their overuse further in this section, and we omit from this section's figures the few applications that use this approach.

As mentioned earlier, nonglobal files can be broken down into local files (i.e., accessed by one process) and subset files (i.e., accessed by more than one process but not all processes). An application uses the subset paradigm because it makes sense for the scientific problem and computational method

being usedfor example, for adaptive mesh refinement or because the developers want to put a subset of the processes (e.g., one process per node) in charge of all I/O. We call the latter subsetting I/O. Subsetting I/O can reduce contention and keep down the number of files, but it requires care and tuning for a good implementation. Taken to the extreme, subsetting I/O turns into serial I/O; that is, one process does all the I/O, which never scales. In interpreting log data, one must take care to distinguish between these three kinds of subset files.

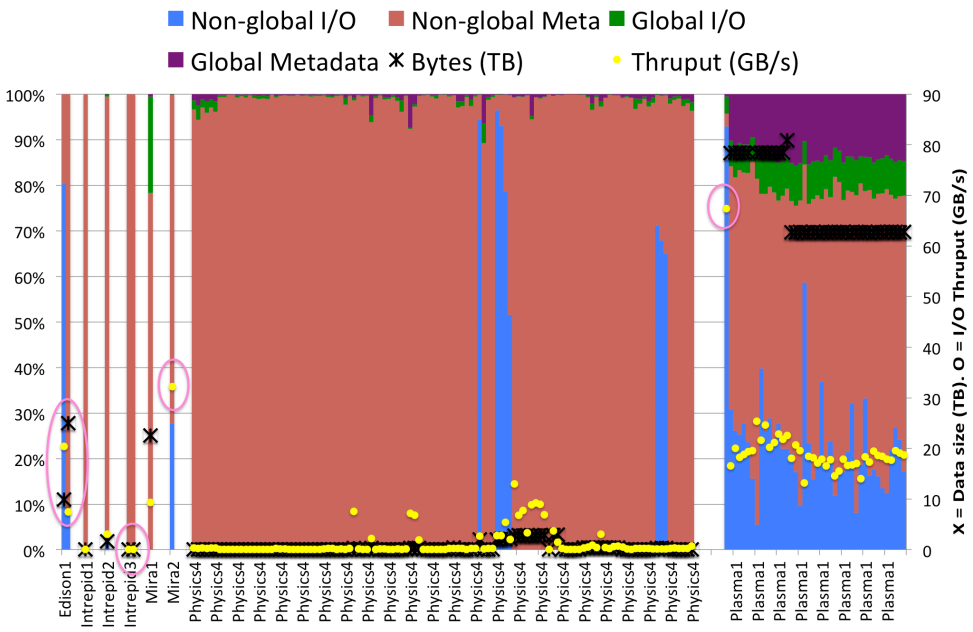


Figure 2.30: I/O throughput and I/O time breakdown for jobs that access over a million files

Local files, often called files-per-process, are easy for users to implement, with no coordination needed between processes. As the number of processes scales up, however, metadata costs can be high, and post-run data analysis and file management become painful. The use of global files, each accessed by all processes, can keep the job's input/result data tidy. Depending on the implementation, global files can have high metadata costs at scale, and contention can be an issue. Good implementations of this paradigm tend to require significant expertise; the resulting parallel I/O libraries have a learning curve for users.

Some of these categories can be broken down further. For example, a

sophisticated application might use subsetting I/O with files that are accessed by (and thus global to) all the processes allowed to perform I/O. And of course a single application can use multiple paradigms in different jobs or inside the same job. But the coarser breakdown suffices for our purposes. Each paradigmglobal, local, and subsethas its pros and cons, and each is found among jobs with the worst and best I/O throughputs. We note also that the local and global paradigms coincide for single-process runs.

Local-file paradigm If a job has enough data, it may be able to avoid the pitfall of excessive metadata costs at scale. An excellent example is the set of all jobs that access at least 1M files, grouped by application in Figure 2.30. Each job is represented by one vertical bar, subdivided into colors based on how it spends its I/O time. Each job is distinguishable in the figure by the yellow dot indicating its throughput and the black X indicating its data size, except for two applications with tightly packed jobs: Physics4 (116 jobs on Mira) and Plasma1 (1 subset-paradigm job on Intrepid and 41 local-file paradigm jobs on Mira). Figure 2.30 shows how fast the local file paradigm can be: Mira1 attains over 10 GB/s with 25 TB of data, and Plasma1 attains about 20 GB/s for its many jobs with 60-80 TB of data. Plasma1 on Mira shows that even when millions of local files are used and their metadata costs (red) exceed their transfer costs (blue), I/O throughput can reliably reach a level that would be the envy of most applications. But the I/O time of the vast majority of local-file jobs in the figure is almost totally dominated by metadata costs, resulting in extremely low throughput. The throughput closely tracks the data size, both of which use the same right-hand y-axis. Two of Plasma1's data points are off the chart: 174 TB and 100 TB data.

Subset paradigm In Figure 2.30, jobs that used the subset paradigm are indicated by pink circles around their yellow throughput dots. Two of the applications used the subset paradigm exclusively, and the figure shows that it can be very effective: Mira2 attained 30 GB/s and Edison1 had 10-20 GB/sfar better aggregate throughput than that of most applications. (Mira2's job has 165 TB of data, putting that data point off the chart.) But the third application, Plasma1, is the star, with over 60 GB/s in its lone subsetting job on Intrepid. The logs show that 1/8 of Plasma1's processes performed I/O in that job, and approximately 1/225 of Mira2's. Edison1

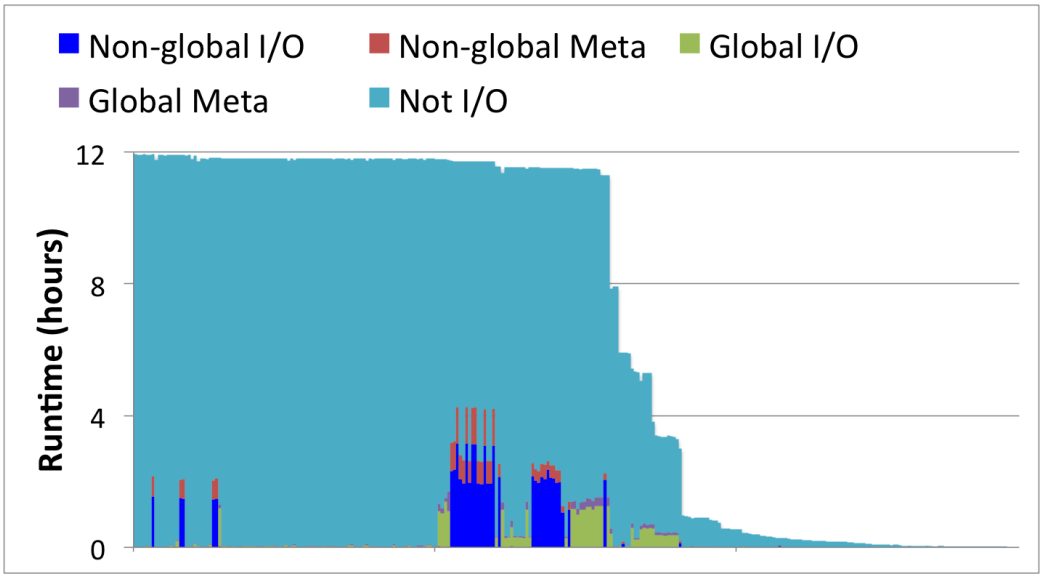


Figure 2.31: Turbulence1's 290 jobs on Intrepid

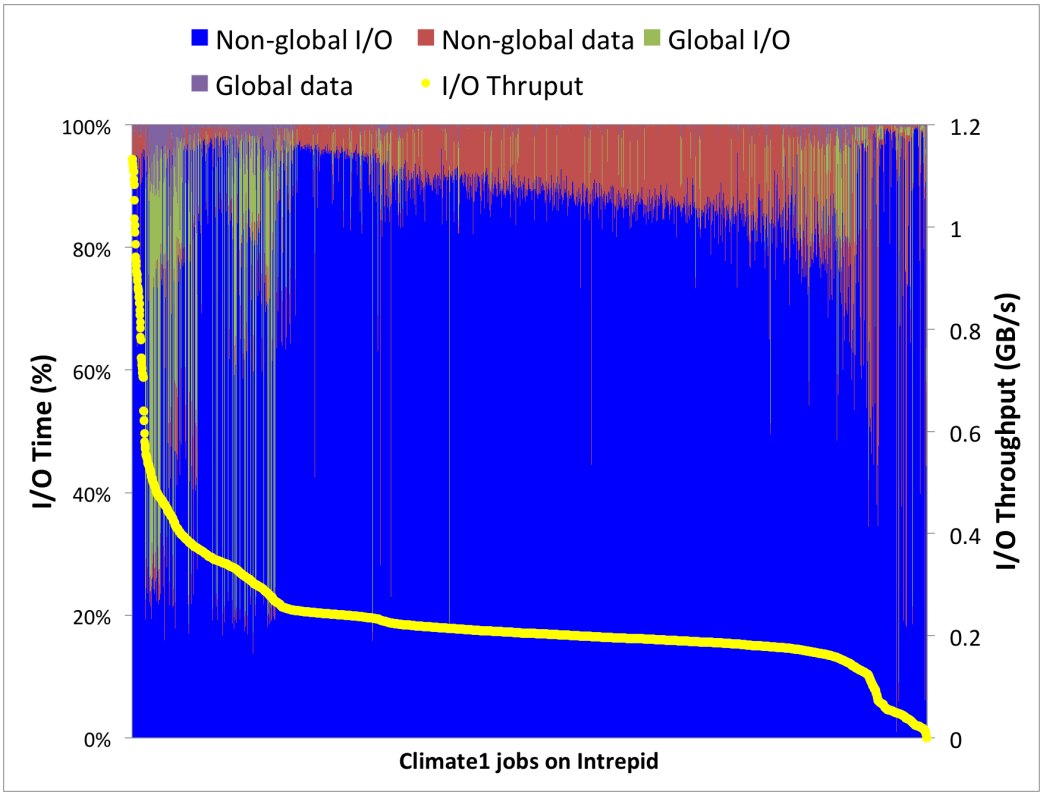


Figure 2.32: Climate1's 3578 runs on Intrepid, sorted by I/O throughput

is using subset files, but not I/O subsetting; recall that subsetting can also serve other purposes, such as I/O for adaptive mesh refinement.

Subsetting is not a panacea: Intrepid3 has poor I/O throughput, totally

dominated by metadata costs. However, Intrepid3 was not doing I/O subsetting, as three-quarters of its processes wrote to the same file. For a better example of ineffective I/O subsetting, consider Turbulence1, which ran on Intrepid and Mira and is among Mira's big-time applications; its I/O time there averages 10% of run time. Figure 2.31 shows Turbulence1's Intrepid jobs, sorted by non-I/O time; the dark blue blocks are jobs using POSIX IO with subsetting (ratio 1024 to 1), and the light green blocks are using MPI-IO with global files. No matter what paradigm is used, note how little impact the I/O time has on the total run time, which means that the users would have little motivation to try other I/O approaches. (One method to achieve this insensitivity is to dedicate processors to I/O, so computation can resume once the data to be output has been sent to the dedicated processes.) Examining a randomly selected job, however, we see that 90% of Turbulence1's I/O requests are of size 8 B, which may be very inefficient for the storage layers and is not something that platform administrators like to see.

Climate1 also offers I/O subsetting as an option for its users, along with interfaces to a variety of storage options. Figure 2.32 shows that users took advantage of these different options in its many jobs on Intrepid. Through other channels, we know that Climate1's developers worked very hard to tame metadata costs and reach its median job throughput on Mira, which Table 2 pegs at 0.3 GB/s, a bit faster than a thumb drive. But Climate1's throughput is still hampered by very small I/O request sizes. For example, in three randomly selected Mira and Intrepid jobs covering both primarily POSIX IO and primarily MPI-IO runs, over half of its I/O requests are of size 100 B. A randomly selected Intrepid job shows subsetting ratios ranging from 4:1 to 1000:1 during different parts of the job; each job subsets differently, with little visible impact on I/O throughput. With median job I/O time at just 5% of total run time on Mira, Climate1's developers has little incentive to refine its I/O approach further.

Global-file paradigm Global files did not guarantee good performance for Earth1, which made small-size POSIX IO requests, or Climate1, which made small requests whether working with MPI-IO or POSIX IO. But Figure 2.33 shows that global MPI-IO files have worked well for the jobs of the Physics7 application on Edison, shown sorted by throughput. The I/O throughput

of Physics7's median job is 7 GB/s, helped along by its tendency to access data in 1 MB requests, well aligned with storage block boundaries. Because Physics7 accesses a lot of data, the y-axis on the right is in log scale. Physics7 had 53 hours total run time, 2.4 hours total I/O time, 65 TB total data, and an I/O percentage of run time of 4.5%. If the application's jobs are sorted into submission order, one can clearly see that Physics7's users experimented twice with nonglobal files: once when they first arrived on Edison and then again after about a hundred jobs, always using a dozen or more processes. Both trials were quickly abandoned.

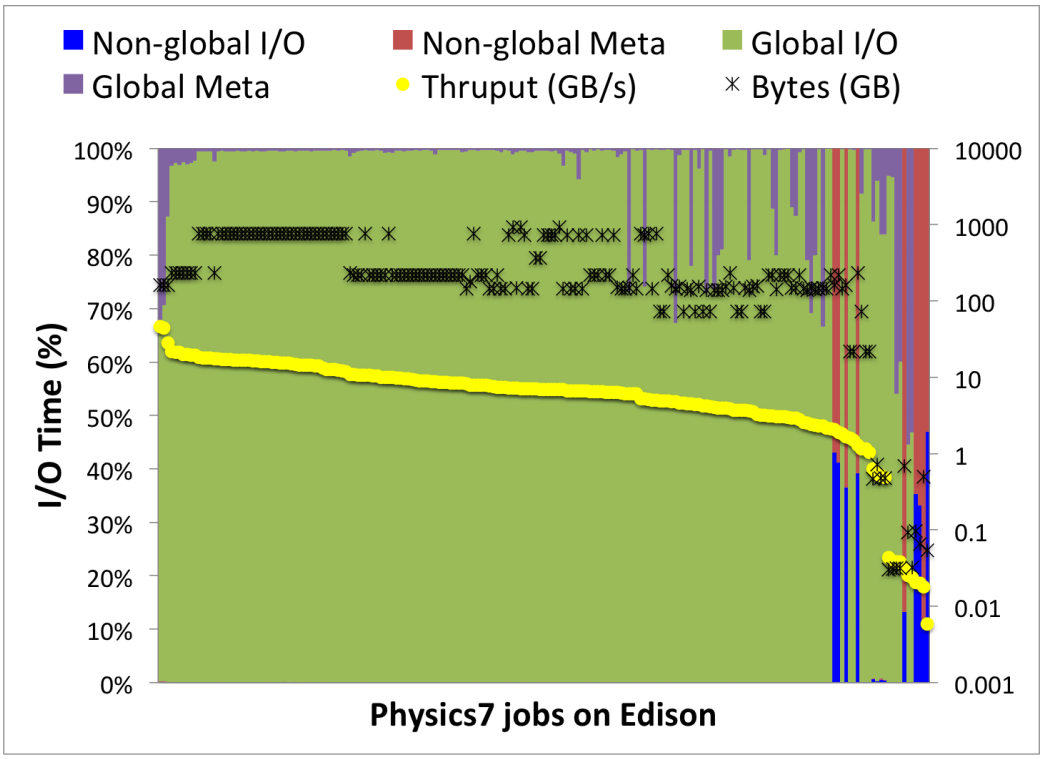


Figure 2.33: Physics7's 199 runs on Edison, sorted by I/O throughput

I/O paradigm usage The choices of I/O paradigm are not mutally exclusive. In an application, user can choose to use multiple I/O paradigms. As an example, Earth1, discussed in Section 2.5, uses more than 50K files in local-file paradigm together with a number of global shared files where it spends most of its I/O time. The use of a particular I/O paradigm does not mean that it is the main paradigm of the job.

We consider a job of using global-file or subset paradigm if it has at least 1 global shared file or 1 subset file. A job is considered as using local-file

Platform	Global-file	Subset	Local-file
Edison	46%	6%	58%
Intrepid	70%	16%	29%
Mira	83%	11%	19%

Table 2.7: Percentage of jobs using a particular I/O paradigm on Edison, Intrepid and Mira

paradigm if the number of local files is larger than number of processes and number of process is larger than 1. The percentage of jobs using a particular I/O paradigm on Edison, Intrepid and Mira is shown in Table 2.7. As shown, the percentage of jobs using global-file paradigm on Edison is much lower than that on Intrepid and Mira and it is lower than percentage of jobs using local-file paradigm. On Intrepid and Mira, global-file paradigm is much more popular than the local-file paradigm. The subset paradigm is not as widely used as the global-file and local-file paradigm in all three platforms.

We compute the percentage of jobs that use only MPI , only POSIX or use both MPI and POSIX for each I/O paradigm (both interfaces mean there are global files that use MPI and other use POSIX in the same job). The percentages are shown in Figure 2.34, 2.35, 2.36. Similar to the trends of using POSIX and MPI on these platform as discussed 2.6.4 , the percentage of jobs using POSIX for global or subset files on all platform is quite significant. The popularity of using POSIX to access shared files suggests that platform administrators might need to have early introduction of users to MPI-IO since MPI-IO can help to coordinate accesses between processes to shared files more efficiently. As expected, nearly all local-file paradigm jobs use POSIX I/O to access the data.

We check the choice of I/O paradigms of jobs with high I/O throughput on each platform (more than 10 GB/s throughput). The percentage of jobs using a particular paradigm is depicted in Figure 2.37. On Intrepid and Mira, global-file paradigm is more popular among high throughput jobs than other paradigms. On Edison, 95% of high throughput jobs use local-file paradigm. Global-file paradigm is also popular among high throughput jobs on Edison.

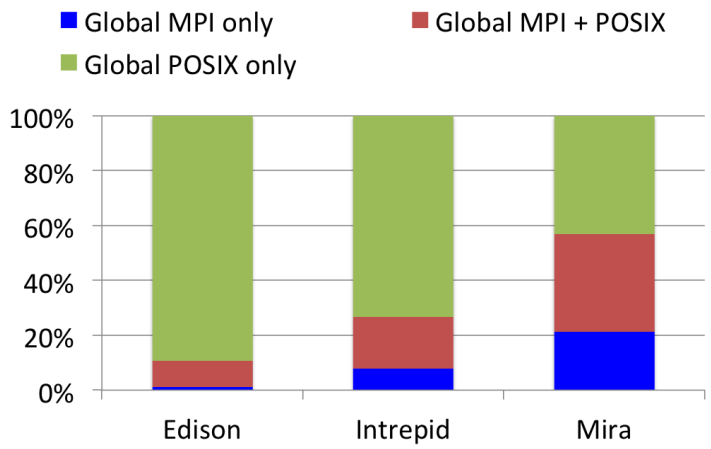


Figure 2.34: Breakdown of I/O interface of jobs using global-file paradigm on Edison, Intrepid and Mira

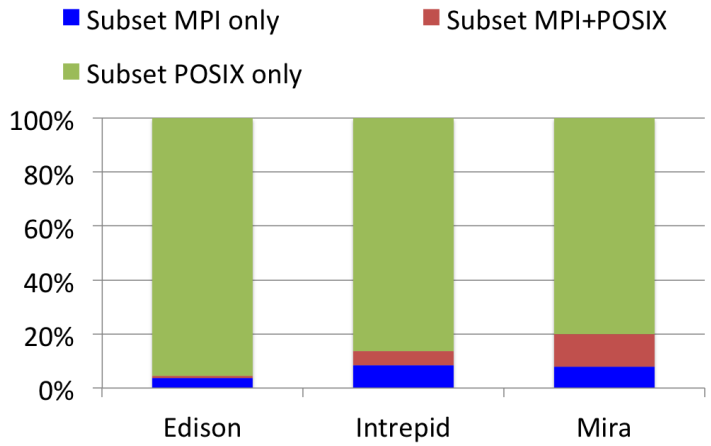


Figure 2.35: Breakdown of I/O interface of jobs using subset paradigm on Edison, Intrepid and Mira

2.7 Cross-platform Analysis

Supercomputer lifetimes are short; a new and faster platform is always on the way. But improved performance does not always come easily for users, as noted by Anantharaj et al.: The high development and maintenance effort required to tune [applications] to multiple platforms is considered a large burden, taking time and resources that might otherwise be spent on other aspects of the projects [31].

Migration to a new platform normally requires retuning of code for good performance, and I/O is no exception. Seemingly small details of the storage

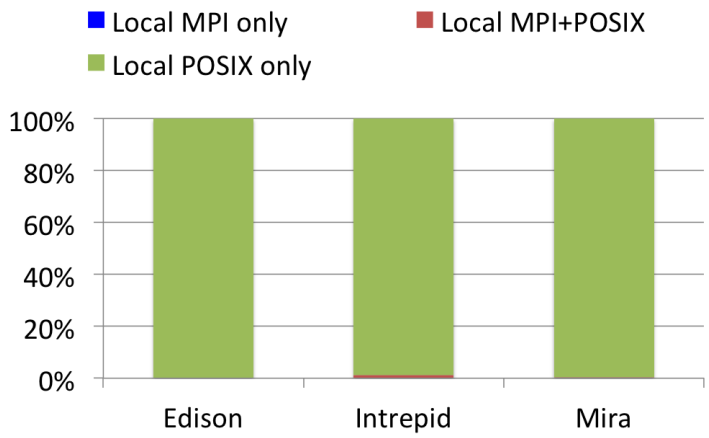


Figure 2.36: Breakdown of I/O interface of jobs using local paradigm on Edison, Intrepid and Mira

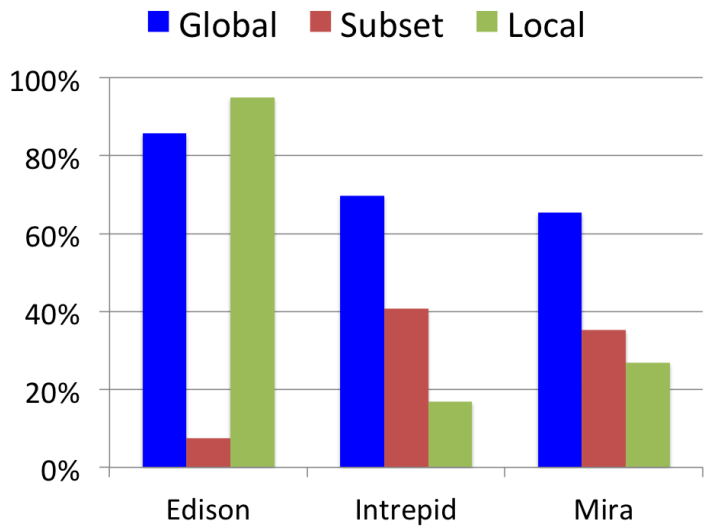


Figure 2.37: Choice of I/O paradigm of jobs with more than 10 GB/s throughput on Edison, Intrepid and Mira

system can have a huge impact on a particular application's throughput [32]. Further, the general trend toward packing more cores into each node tends to increase file access contention for processes in the same node. Thus an application running with the same number of processes on both platforms might see throughput fall even if the new storage system is similar to the old and has higher peak throughput. Case studies and I/O benchmarks have provided such insights in the past; Darshan now lets us examine the impact of migration at a larger scale. The lesson we learnt during our cross-platform

study is that to maintain current performance, application I/O may need retuning even when moving to a similar but faster platform.

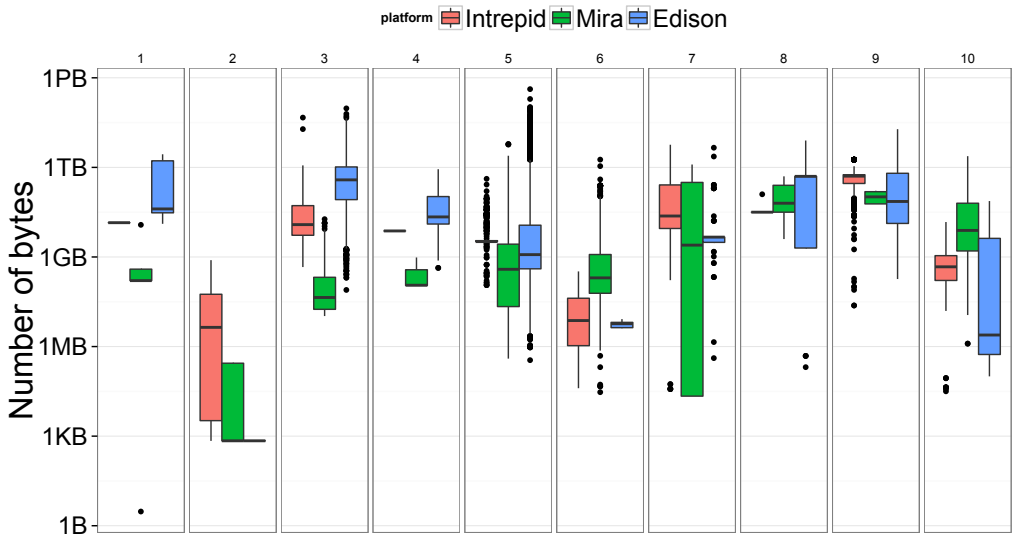


Figure 2.38: Quartiles and outliers of total bytes of each job, for the ten applications that ran on all three platforms

We identified each application that ran under an identical executable name on two or more of our platforms: 82 applications on both Intrepid and Mira, 39 on Mira and Edison, 27 on Intrepid and Edison, and 10 on all three platforms. For each of these applications, we determined the median aggregate I/O throughput of all its jobs on those platforms and compared that value across platforms. However, the median job of most cross-platform applications on most platforms has small total data, just as do most applications; Figure 2.38 illustrates this situation with a box plot of job data size for the ten applications that ran on all three platforms, with applications separated by vertical black bars. We have already observed that a job with low total data size will have well under 1 GB/s aggregate I/O throughput. Thus, the difference in median aggregate I/O throughput of jobs on different platforms is due primarily to differences in a jobs total data size. For a fair cross-platform comparison of these applications, we need to match job sizes across platforms. With over a hundred applications to match up, we could not do the matching by hand, so full analysis of that dataset must wait for future work.

Here, we present three cross-platform case studies.

2.7.1 Case study 1: Earth1

Earth1 is the number 1 big time app on Mira and number 4 on Intrepid. Figure 2.39 has a heat map of all jobs of Earth1 on Mira and Intrepid. More intense color indicates a greater number of jobs. The throughput of Earth1's median job drops from 4.5 GB/s on Intrepid to 1.2 GB/s on Mira. Data size also declines but remains large enough not to explain the drop.

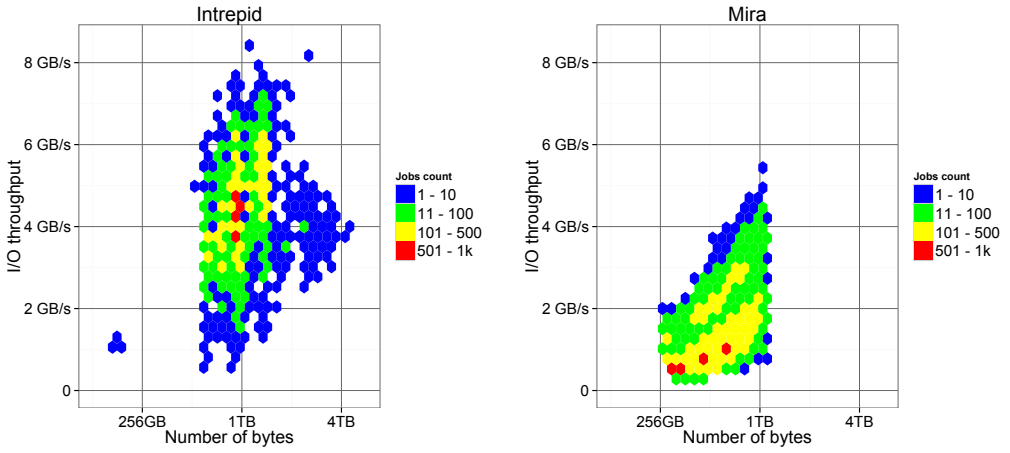


Figure 2.39: Heat map of Earth1's jobs, broken down by data size and I/O throughput, on Intrepid (left) and Mira (right)

The decreased I/O throughput means that Earth1 spends a larger portion of its run time in I/O. As shown in Figure 2.40, the main bottleneck is in metadata activity for global shared files. As discussed earlier, Earth1 uses POSIX to write to a number of global shared files. Earth1's jobs on Mira tend to use more processes, which also are packed more tightly into nodes than on Intrepid. With more processes and less total data, request sizes drop. The tighter packing, more processes issuing requests, and decreased size of requests all increase contention. Earth1's I/O implementation does not scale proportionately.

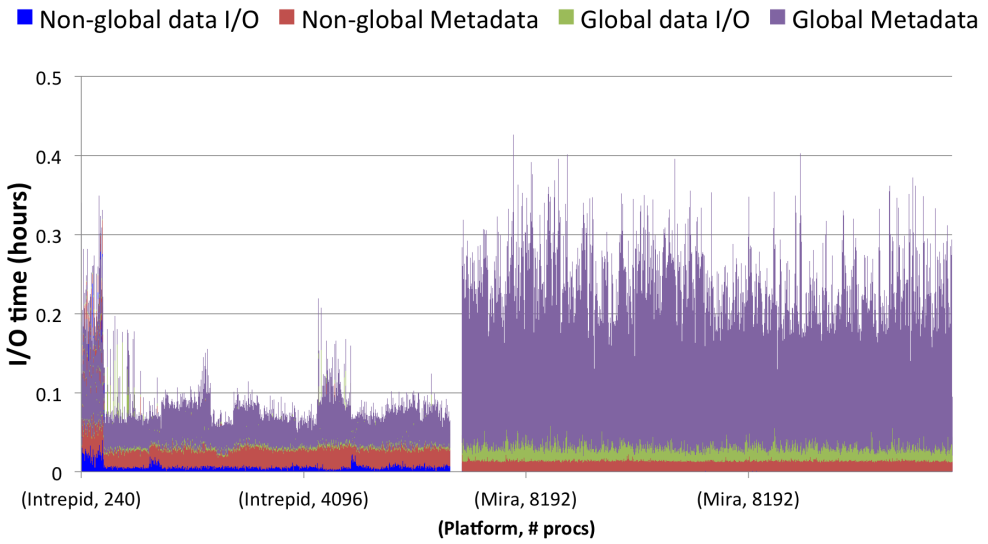


Figure 2.40: Breakdown of Earth1's total I/O time on Intrepid (left) and Mira (right)

2.7.2 Case study 2: Crossplat1

Now consider the rightmost application in Figure 2.38, a physics code that we will call Crossplat1. Figure 2.41 shows that in general, Crossplat1 scales well with increasing data on Mira and Edison, and with more processes on Edison. On Intrepid, Crossplat1 rarely exceeded 1 GB/s throughput.

We applied the three-step app-specific analysis procedure to Crossplat1 on Mira and Edison, and found that most I/O time was spent in non-global I/O of a number of local POSIX read/write files (#lPrwf for short). Figure 2.42 depicts this for Mira, with jobs sorted by #lPrwf. The log-scale right-hand y-axis is for the overlay variables: I/O throughput, total bytes and #lPrwf. For a fixed #lPrwf, I/O throughput increases nicely with data size. But when #lPrwf is 512 or more, metadata costs shoot up (tall red bars). This suggests that limiting #lPrwf may improve throughput for Crossplat1 on Mira.

The first runs on Edison use single-processor, with very little data; small jobs naturally have low throughput. After that, Crossplat1s behavior on Edison was similar to Mira as shown in Figure 2.43 except that #lPrwf did not exceed 256, so metadata costs remained modest in almost all jobs on Edison.

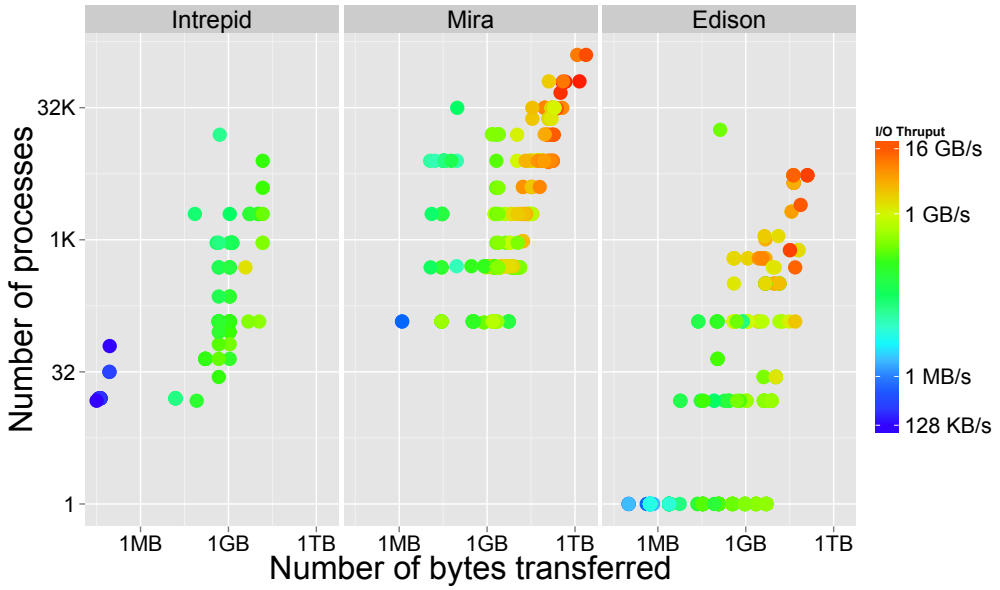


Figure 2.41: I/O throughput, data size, and number of processes for each of Crossplat1's jobs on three platforms

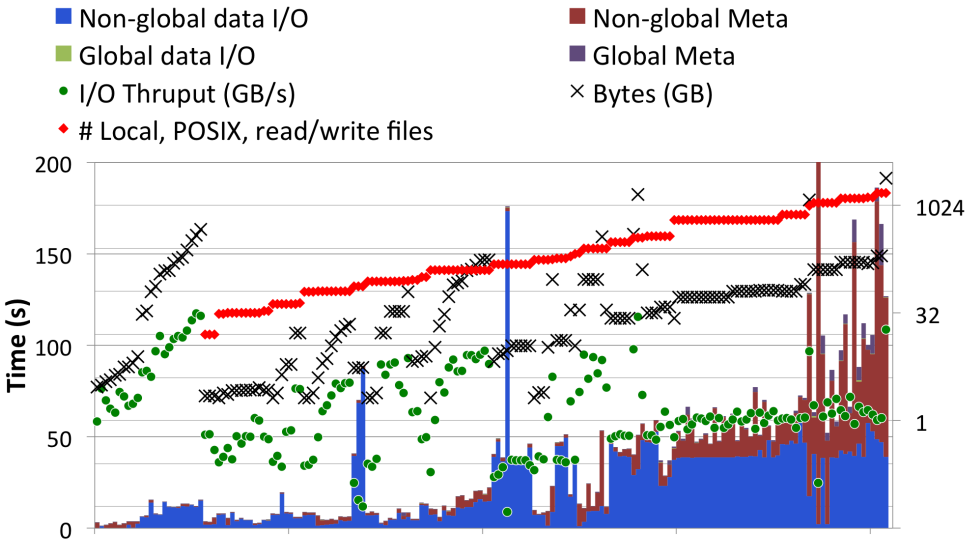


Figure 2.42: I/O time and throughput (green dot), bytes accessed (black X) and number of local POSIX read/write files (red diamond) for each of Crossplat1's jobs on Mira.

2.7.3 Case study 3: Weather1

As our third case study, consider Weather1, the ninth application in Figure 2.38 and a big-time application on Edison and Intrepid. Weather1 has

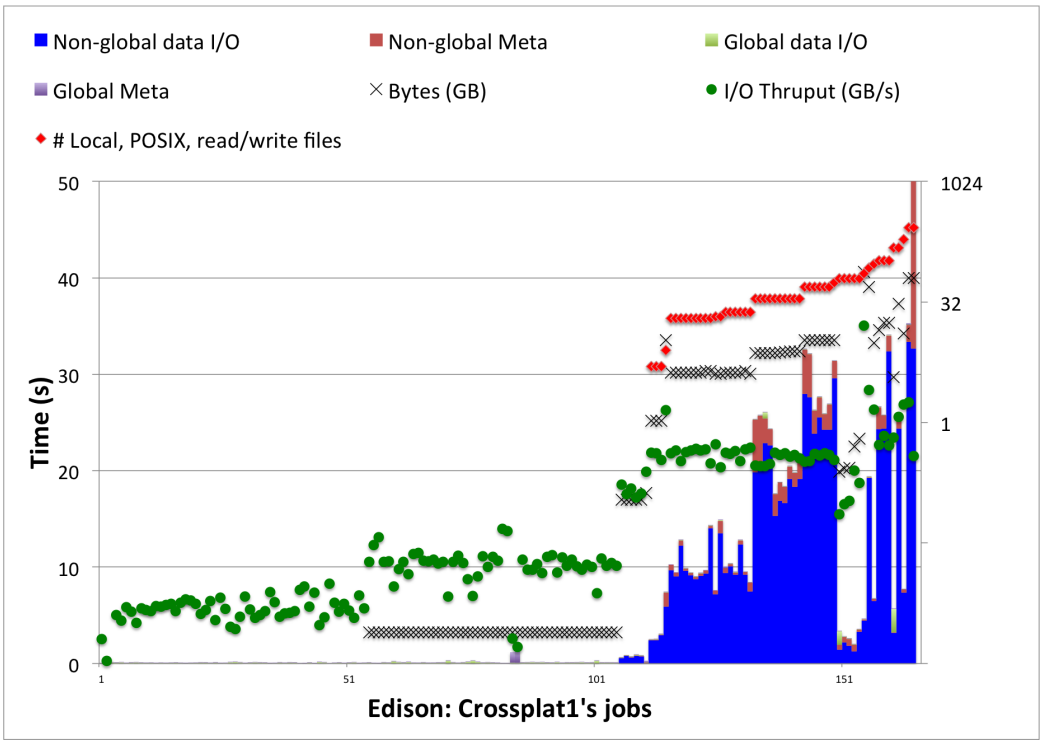


Figure 2.43: I/O time and throughput (green dot), bytes accessed (black X) and number of local POSIX read/write files (red diamond) for each of Crossplat1's jobs on Edison.

few Mira runs, and we do not consider them here. Figure 2.44 shows that Weather1's I/O throughput was consistently low on Intrepid, but as high as 48 GB/s on Edison. The scaling pattern is unclear.

In Figure 2.45, each Weather1 job on Intrepid is represented by a vertical bar whose colors give a breakdown of the jobs total I/O time (left-hand y-axis). The figure also shows each jobs I/O throughput (black X), number of processes (yellow dot) and data size (blue plus) on the log-scale right-hand y-axis. The jobs are sorted by data size. Different I/O paradigms were used by different users, visible in the figure as four distinct blocks of colors. Weather1 spent most of its I/O time in MPI global shared files and never reached 1 GB/s of throughput under any paradigm, even when accessing over 1 TB of data.

Weather1 fares better on Edison, where a third of the jobs exceed 1 GB/s throughput, as shown in Figure 2.46 with jobs sorted by I/O throughput (black X). The figure also shows each jobs data size (blue plus), number of POSIX global files (orange dot) and number of MPI global files (yellow

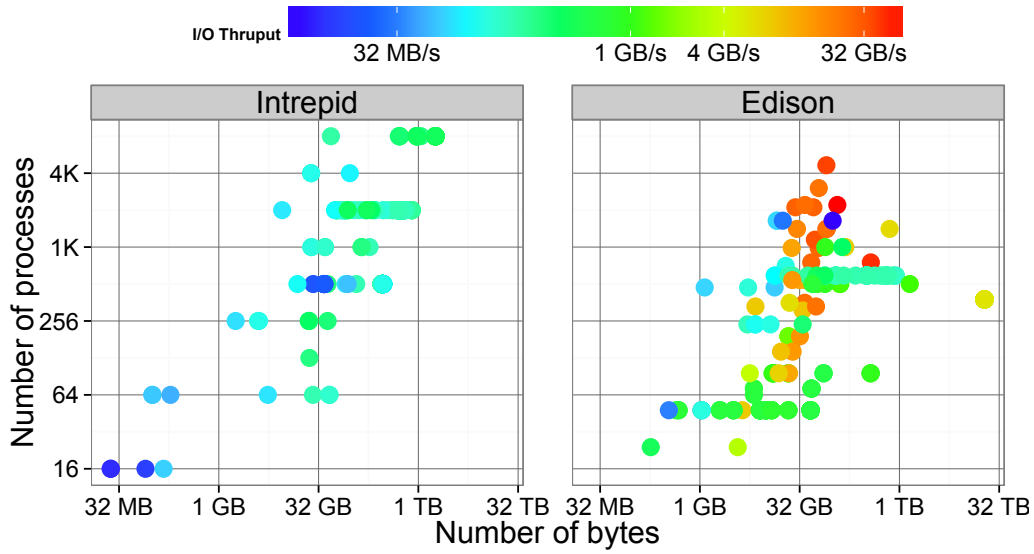


Figure 2.44: Breakdown of Weather1's jobs by I/O throughput, number of processes, total data size, and platform

dot) on the log-scale right-hand y-axis. Here, Weather1 jobs fall into three groups. The first group uses MPI-IO global shared files and has consistently low throughput (¡0.2 GB/s). The second group uses local files and more modest data sizes (always under 1 TB) and throughput closely tracks data size, reaching as high as 48 GB/s. The third group of jobs has extremely large data (over 10 TB), and uses POSIX global files; these jobs attain 3-6 GB/s. Darshan does not observe whether jobs tune Lustre parameter settings, but it is worth noting that these results are in line with others observations that the default settings on Lustre lead to low MPI-IO performance [33], and that the local file I/O paradigm tends to perform relatively well on Edison.

2.8 Conclusions

I/O is an increasingly important factor in the productivity of HPC applications. However, obtaining good I/O performance for a broad range of applications on diverse platforms is a major challenge.

Lightweight tools such as Darshan can augment traditional benchmarking and tracing tools, and provide an overall understanding of the I/O behavior of applications, workloads, and platforms. We used Darshan I/O logs to provide

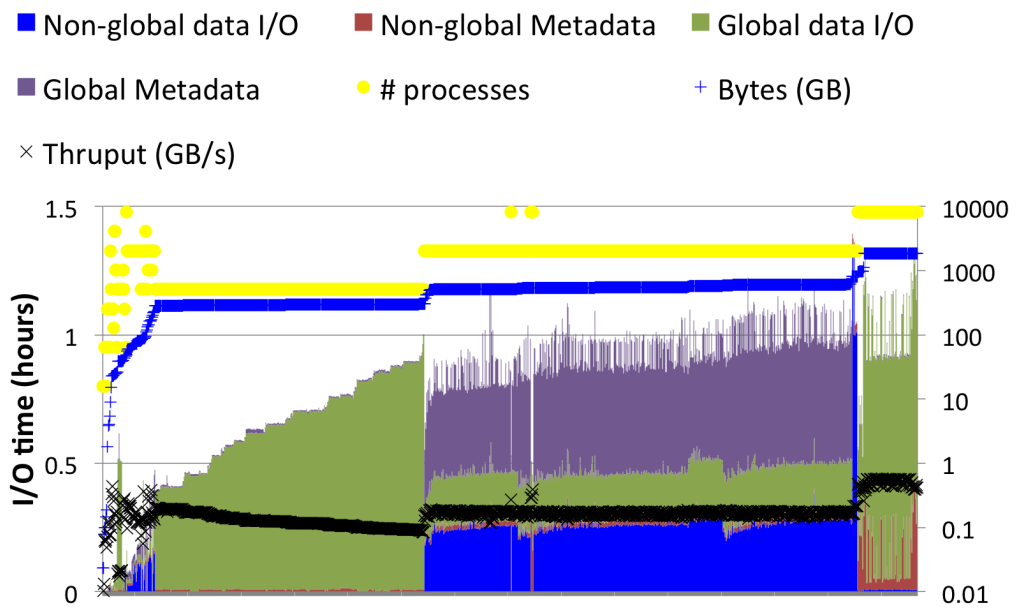


Figure 2.45: I/O time breakdown of Weather1 jobs on Intrepid

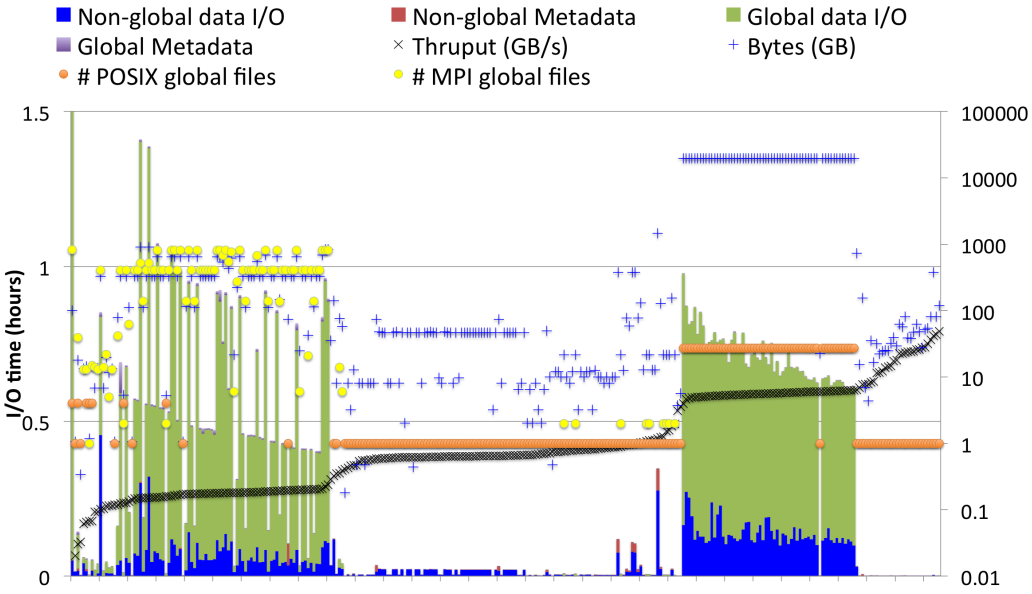


Figure 2.46: I/O time breakdown of Weather1 jobs on Edison.

a broad view of I/O behavior on three leading HPC platforms. Our results lead us to believe that while tremendous progress has been made in hardware and software research for HPC I/O, gaps remain in the adoption of best practices by scientific application developers. For instance, strategies such as usage of text files and raw, low-level POSIX I/O calls will be untenable on

future platforms; adoption of higher-level I/O libraries can help increase the longevity of codes on future generations of supercomputers. HPC I/O specialists need to ensure that application developers understand the tradeoffs in different ways of performing I/O, perhaps through I/O boot camps and tutorials offered in cooperation with platform administrators. Our results also lead us to believe that while much research effort is invested in extreme-scale testing and optimization, a large fraction of the HPC community has modest-scale metadata and data challenges; designers of HPC facilities must take these needs into account when designing and provisioning I/O resources. We believe that tools such as Darshan can give platform administrators critical insights into system utilization; early and proactive intervention into suboptimal I/O behavior can greatly enhance the utilization of a platform's existing HPC resources.

CHAPTER 3

TAMING PARALLEL I/O COMPLEXITY WITH AUTO-TUNING

In this chapter, we present an auto-tuning system for optimizing I/O performance and demonstrate its value across platforms, applications and at scale. Our techniques apply for applications that uses HDF5 as their high-level I/O library. This framework uses a genetic algorithm to search a large space of tunable parameters and to identify effective settings at all layers of the parallel I/O stack. The parameter settings are applied transparently by the auto-tuning system via dynamically intercepted HDF5 calls.

To validate our auto-tuning system, we applied it to three I/O benchmarks (VPIC, VORPAL, and GCRM) that replicate the I/O activity of their respective applications. We tested the system with different weak-scaling configurations (128, 2048, and 4096 CPU cores) that generate 30 GB to 1 TB of data, and executed these configurations on diverse HPC platforms (Cray XE6, IBM BG/P, and Dell Cluster). In all cases, the auto-tuning framework identified tunable parameters that substantially improved write performance over default system settings. We consistently demonstrate I/O write speedups between 2x and 100x for test configurations.

Our work has resulted in a publication in The 25th International Conference for High Performance Computing, Networking, Storage and Analysis, 2013 [8].

The remainder of this chapter is organized as follows. Section 3.1 presents the background on the I/O software stack and review related work in existing research literature. Section 3.2 presents our I/O auto-tuning system; Section 3.3 discusses the experimental setup used to evaluate benefits of the auto-tuning system across platforms, applications, and scale. Section 3.4 presents performance results from our tests and discusses the insights gained from the auto-tuning effort and current limitations. Finally, Section 3.5 offers concluding thoughts.

3.1 Background and Related Work

3.1.1 The I/O Software Stack

In order to achieve high I/O performance, HPC applications rely on a multilayer I/O software stack to access the data. Figure 3.1 depicts a typical I/O software stack found on many current HPC systems. All the layers in the I/O stack are designed for portable data abstractions and performance optimization. A high level I/O library translates application's data models into a structured file format such as HDF5 or NetCDF. The middleware layer (which typically is an MPI-IO implementation) takes care of organizing and optimizing accesses from many processes. At the bottom of the stack, the parallel file system such as GPFS, PVFS or Lustre provides accesses to files stored on the storage hardware. Application can choose to access its data using one or a combination of I/O libraries in the I/O stack. Ideally, the use of these software layers should reduce the amount of effort spent on optimizing I/O performance of scientific applications.

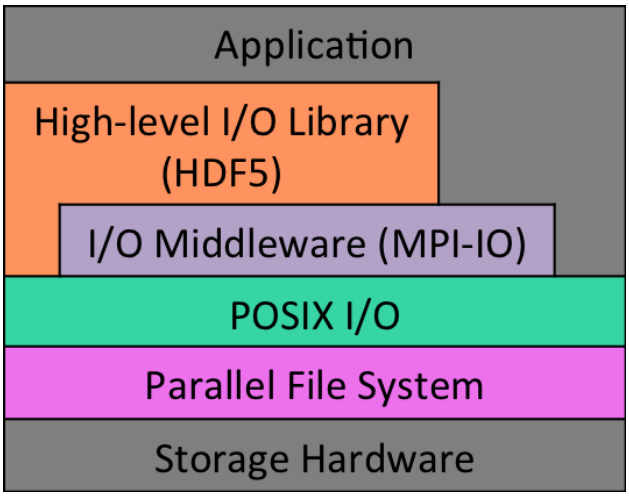


Figure 3.1: The I/O software stack

The advantage of having a hierarchical I/O stack is that these layers provide a bridge for the gap in data representation at adjacent levels and essential abstractions to users. They help to hide complex implementation details and apply optimization techniques to improve the overall performance. Each layer offers many optimization techniques, for example, chunking in HDF5, data sieving and collective buffering in ROMIO, (an MPI-IO implementa-

tion), and striping in the Lustre filesystem. Users can enable these optimization techniques by setting appropriate parameters or hints. However, isolated optimizations at separate layers are often not enough to achieve the full potential of the I/O subsystem. In order to do so, users need to combine these optimization techniques. Unfortunately, due to the complex interaction between layers of the I/O stack, determining the right combination of optimization parameters is a difficult task. Moreover, each combination of application, HPC platform and problem size requires a different configuration for optimized performance. Scientific application developers do not have the time or expertise to take on the substantial burden of manual exploration of the parameter space for each problem configuration. As a result, they often resort to using system defaults, a choice that frequently results in poor I/O performance. We expect this problem to be compounded on exascale class machines, which will likely have a deeper software stack with hierarchically arranged hardware resources.

3.1.2 Related Work

Auto-tuning in computer science is a prevalent technique for improving performance of computational kernels. There has been extensive research in developing optimized linear algebra libraries and matrix operation kernels using auto-tuning [34, 35, 36, 37, 38, 39, 40]. The search space in these efforts involves optimization of CPU cache and DRAM parameters along with code changes. All these auto-tuning techniques search various data structure and code transformations using performance models of processor architectures, computation kernels, and compilers. Our study focuses on auto-tuning the I/O subsystem for writing and reading data to a parallel file system, in contrast to tuning computational kernels. There are a few key challenges unique to the I/O auto-tuning problem. Each function evaluation for the I/O case takes on the order of minutes, as opposed to milli-seconds for computational kernels. Thus, an exhaustive search through the parameter space is infeasible and a heuristic based search approach is needed. I/O runs also face dynamic variability and system noise while linear algebra tuning assumes a clean and isolated single node system. The interaction between various I/O parameters and how they impact performance are not very well

understood, making interpreting tuned results a complex task.

We use genetic algorithms as a parameter space searching strategy. Heuristics and meta-heuristics have been studied extensively for combinatorial optimization problems as well as code optimization [41] and parameter optimization [42] problems similar to the one we address. Of the heuristic approaches, genetic algorithms seem to be particularly well suited for real parameter optimization problems, and a variety of literature exists detailing the efficacy of the approach [43, 44, 39]. A few recent studies have used genetic algorithms [45] and a combination of approximation algorithms with search space reduction techniques [46]. Both of these are again targeted to auto-tune compiler options for linear algebra kernels. We chose to implement a genetic algorithm to intelligently traverse the sample space for each test case; we found our approach produced well-performing configurations after a suitably small number of test runs.

Various optimization strategies have been proposed to tune parallel I/O performance for a specific application or an I/O kernel. However, they are not designed for automatic tuning of any given application and require manual selection of optimization strategies. Our auto-tuning framework is designed towards tuning an arbitrary parallel I/O application. Hence, we do not discuss the exhaustive list of research efforts. We focus on comparing our research with automatic performance tuning efforts.

There are a few research efforts to auto-tune and optimize resource provisioning and system design for storage systems [47, 48, 49]. In contrast, our study focuses on tuning the parallel I/O stack on top of a working storage system.

Auto-tuning of parallel I/O has not been studied at the same level as the tuning for computation kernels. The Panda project [50, 51] studied automatic performance optimization for collective I/O operations where all the processes used by an application to synchronize I/O operations such as reading and writing an array. The Panda project searched for disk layout and disk buffer size parameters using a combination of a rule-based strategy and randomized search-based algorithms. The rule-based strategy is used when the optimal settings are understood and simulated annealing is used otherwise. The simulated annealing problem is solved as a general minimization problem, where the I/O cost is minimized. The Panda project also used genetic algorithms to search for tuning parameters [52]. The optimization

approaches proposed in this project were applicable to the Panda I/O library, which existed before MPI-IO and HDF5. The Panda I/O library is not in use now and the optimization strategy was not designed for parallel file systems that are in current use.

Yu et al. [53] characterize, tune, and optimize parallel I/O performance on the Lustre file system of Jaguar, a Cray XT supercomputer at Oak Ridge National Laboratory (ORNL). The authors tuned data sieving buffer size, I/O aggregator buffer size, and the number of I/O aggregator processes. This study did not propose an auto-tuning framework but manually ran a selected set of codes several times with different parameters. Howison et al. [4] also perform manual tuning of various benchmarks that select parameters for HDF5 (chunk size), MPI-IO (collective buffer size and the number of aggregator nodes) and Lustre parameters (stripe size and stripe count) on the Hopper supercomputer at NERSC. These two studies prove that tuning parallel I/O parameters can achieve better performance. In our study we develop an auto-tuning framework that can select tuning parameters.

You et al. [54] proposed an auto-tuning framework for the Lustre file system on Cray XT5 systems at ORNL. They search for file system stripe count, stripe size, I/O transfer size, and the number of I/O processes. This study uses mathematical models based on queuing models. The auto-tuning framework first develops a model in a training phase that is close to the real system. The framework then searches for optimal parameters using search heuristics such as simulated annealing, genetic algorithms, etc. A mathematical model developed for different systems based on queuing theory can be farther from the real system and may produce inaccurate performance results. In contrast, our framework searches for parameters on real system using search heuristics. On the other hand, once the mathematical model is created and validated, it can help to guide the tuning process to be more effective and quicker.

A preliminary version of our auto-tuning framework appears in earlier work [55], where we primarily study the performance of our system at a small scale. In this work, we do a more thorough analysis of the system on diverse platforms, applications, and concurrencies, and conduct an in-depth analysis of resulting configurations.

3.2 Auto-tuning Framework

The main challenges in designing and implementing an I/O auto-tuning system are (1) selecting an effective set of tunable parameters at all layers of the stack, and (2) applying the parameters to applications or I/O benchmarks without modifying the source code. We tackle these challenges with the development of two components: *H5Evolve* and *H5Tuner*.

For selecting tunable parameters, a naïve strategy is to execute an application or a representative I/O kernel of the application using all possible combinations of tunable parameters for all layers of the I/O stack. This is an extremely time and resource consuming approach, as there are many thousands of combinations in a typical parameter space. A reasonable approach is to search the parameter space with a small number of tests. Towards this goal, we developed *H5Evolve* to search the I/O parameter space using a genetic algorithm.

A genetic algorithm (GA) is a meta-heuristic for approaching an optimization problem, particularly one that is ill-suited for traditional exact or approximation methods. A GA is meant to emulate the natural process of evolution, working with a "population" of potential solutions through successive "generations" (iterations) as they "reproduce" (intermingle portions between two members of the population) and are subject to "mutations" (random changes to portions of the solution). A GA is expected, although it cannot necessarily be shown, to converge to an optimal or near-optimal solution, as strong solutions beget stronger children, while the random mutations offer a sampling of the remainder of the space. The H5Evolve module, implementing the genetic algorithm, samples the parameter space by selecting a set of parameter combinations, running the application with this parameter set and measuring the performance of the run. Based on the measured performance, H5Evolve adjusts the combination of tunable parameters for further testing. As H5Evolve passes through multiple generations, better parameter combinations (i.e., sets of tuned parameters with high I/O performance) emerge.

An application can control tuning parameters for each layer of the I/O stack using hints set via API calls. For instance, HDF5 alignment parameters can be set using the `H5Pset_alignment()` function. MPI-IO hints can be set in a similar fashion for the collective I/O and file system striping parameters.

While changing the application source code is possible if the code is available, it is impractical when testing a sizable number of parameter combinations. *H5Tuner* solves this problem by dynamically intercepting HDF5 calls and injecting optimization parameters into parallel I/O calls at multiple layers of the stack without the need for source code modifications. H5Tuner is a transparent shared library that can be preloaded before the HDF5 library, prioritizing it over the original HDF5 function calls.

Figure 3.2 shows our auto-tuning system that uses both H5Tuner and H5Evolve for searching a parallel I/O parameter space. H5Evolve takes the I/O parameter space as input and for each experiment generates a configuration file in XML format. The parameter space contains possible values for I/O tuning parameters at each layer of the I/O stack and the configuration file contains the the parameter settings that will be used for a given run. H5Tuner reads the configuration file and dynamically links to HDF5 calls of an application or I/O benchmark. After running the executable, the parameter settings and I/O performance results are fed back to H5Evolve and influence the contents of the next configuration file. As H5Evolve tests various combinations of parameter settings, the auto-tuning system selects the best performing configuration for a specific I/O kernel or application.

3.3 Experimental Setup

We have evaluated the effectiveness of our auto-tuning framework on three HPC platforms using three I/O benchmarks at three different scales.

The HPC platforms include Hopper, a Cray XE6 system at National Energy Research Scientific Computing Center (NERSC); Intrepid, an IBM BlueGene/P (BG/P) system at Argonne Leadership Computing Facility (ALCF); and Stampede, a Dell PowerEdge C8220 cluster at Texas Advanced Computing Center (TACC). Table 3.1 lists details of these HPC systems; note that the number and type of I/O resources vary across these platforms. We also note that the I/O middleware stack is different on Intrepid from that on Hopper and Stampede. On Intrepid, the parallel file system is GPFS, while Hopper and Stampede use the Lustre file system.

The I/O benchmarks are the I/O kernel of the VPIC, VORPAL, and GCRM applications. These I/O kernels represent three distinct I/O write

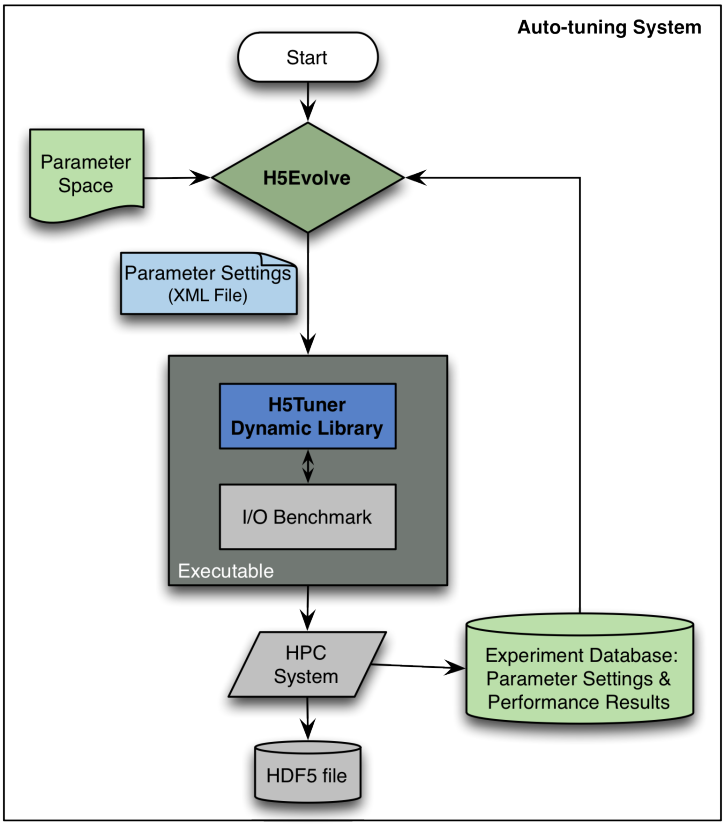


Figure 3.2: Overall Architecture of the Auto-Tuning Framework

motifs with different data sizes. The I/O motif of VPIC-IO is a 1D particle array of a given number of particles; each particle has eight variables. The kernel writes 8M particles per MPI process for all experiments reported here. VORPAL-IO uses H5Block to write non-uniform chunks of 3D data per processor. The kernel takes 3D block dimensions (x, y, and z) and the number of components as input. In our experiments, we used 3D blocks of 100x100x60 with different number of processors and the data is written for 20 time steps. I/O pattern of GCRM-IO corresponds to a semi-structured geodesic mesh, where the grid resolution and subdomain resolution are specified as input. In our tests we used varying grid resolutions at different concurrencies. By default, this benchmark uses 25 vertical levels and 1 iteration.

We designed a weak-scaling configuration to test the performance of the auto-tuning framework at three concurrencies, i.e., 128, 2048, and 4096 cores. The amount of data each core writes is constant for a given I/O kernel, i.e., the amount of data an I/O kernel writes increases proportional to the number of cores used. Table 3.2 shows the sizes of the datasets generated by the I/O

HPC System	Architecture	Node Hardware	File-system	Storage Hardware	Peak I/O BW
NERSC Hopper	Cray XE6	AMD Opteron processors, 24 cores per node, 32 GB memory	Lustre	156 OSTs, 26 OSSs	35 GB/s [56]
ALCF Intrepid	IBM BG/P	PowerPC 450 processors, 4 cores per node, 2 GB memory	GPFS	640 IO Nodes, 128 file servers	47 GB/s (write) [57]
TACC Stampede	Dell PowerEdge C8220	Xeon E5-2680 processors, 16 cores per node, 32GB memory	Lustre	160 OSTs, 58 OSSs	159 GB/s [58]

Table 3.1: Details of various HPC systems used in this study

benchmarks. The amount of data written by a kernel ranges from 32 GB (with 128 cores) to 1.1 TB (with 4096 cores).

I/O Benchmark	128 Cores	2048 Cores	4096 Cores
VPIC-IO	32 GB	512 GB	1.1 TB
VORPAL-IO	34 GB	549 GB	1.1 TB
GCRM-IO	40 GB	650 GB	1.3 TB

Table 3.2: Weak scaling configuration for the three I/O benchmarks - Total amount of data

3.3.1 Parameter Space

H5Evolve can take arbitrary values as input for a parameter space. However, evolution of the GA will require more generations to search a parameter space with many values. To shorten the search time, we selected a few meaningful parallel I/O parameters for all the layers of the I/O stack based on previous research efforts [4] and our experience [56]. We have chosen most of the parameter values to be powers-of-two except some parallel file system parameters. We set the largest parameter value for the Lustre stripe count to be equal to the maximum number of available object storage targets (OSTs), which is 156 on Hopper and 160 on Stampede. The GPFS parameters that

Parameter	Min	Max	Number of values
strp_fac	4	156/160	10
strp_unt / cb_buf_siz	1 MB	128 MB	8
cb_nodes	1	256	12
align(thresh, bndry)	(0,1)	(16KB, 32MB)	14
bglockless	True	False	2
IBM_largeblock_io	True	False	2
chunk_size	10 MB	2 GB	25

Table 3.3: A list of the tunable parameters and their ranges used for experiments in this study. We show the minimum and maximum values for each parameter, with powers-of-two values in between. The last column shows the number of distinct values used for each parameter.

we tuned are boolean. The process of curtailing parameter values to reasonable ranges based on knowledge of page sizes, min/max striping ranges and powers-of-two values can be done by one who is modestly familiar with the system. And this task needs to be performed only once on a per-system basis. Table 3.3 shows ranges of various parameter values. A user of our auto-tuning system can set the parameter space by modifying the parameter list in H5Evolve. Adding new parameters to search requires simple modifications to H5Tuner. The following is a list of parameters we used as part of the parameter space and their target platforms.

- Lustre (on Hopper and Stampede):
  - Stripe count (`strp_fac`) sets the number of OSTs over which a file is distributed.
  - Stripe size (`strp_unt`) sets the number of bytes written to an OST before cycling to the next OST.
- GPFS (on BG/P Intrepid):
  - Locking: Intrepid has a ROMIO (an MPI-IO implementation [59]) driver to avoid NFS-type file locking. This option is enabled by prefixing a file name with `bglockless:`.
  - Large blocks: ROMIO has a hint named `IBM_largeblock_io` which optimizes I/O with operations on large blocks.
- MPI-IO (on all three platforms):

  - The number of collective buffering nodes (`cb_nodes`) sets the number of aggregators for collective buffering. On Intrepid, the parameter to set the number of aggregators is `bgl_nodes_pset`.
  - The collective buffer size (`cb_buf_size`) is the size of the intermediate buffer on an aggregator for collective I/O. We set this value to be equal to the stripe size on Hopper and Stampede.

- HDF5 (on all three platforms):
  - Alignment (`align(thresh, bndry)`): HDF5 file access is faster if certain data elements are aligned in a specific manner. Alignment sets any file object with size more than a threshold value to an address that is a multiple of a boundary value.
  - Chunk size (`chunk_size`): In addition to contiguous datasets, where datasets are stored in single blocks in files, HDF5 supports chunked layout in which the data are stored in separate chunks. We used this parameter specifically for the GCRM-IO kernel.

3.4 Results

Out of the 27 experiments (3 I/O benchmarks `x` 3 concurrencies `x` 3 HPC platforms), we successfully completed 24 experiments. Due to computer resource allocation limitations on Stampede, we could not finish the three 4096-core experiments on that system. However, we expect the performance improvement trends in the remaining runs to be the same as the completed experiments.

3.4.1 Tuned I/O Performance Results

The plots in Figure 3.3, 3.4, 3.5 present the I/O rate improvement using tuned parameters that our auto-tuning system chose for the three I/O benchmarks at 128, 2048 and 4096 concurrencies. H5Evolve ran for 10 hours, 12 hours, and 24 hours for the three concurrencies to search through the parameter space of each experiment. In most cases, the GA evolved through 15 to 40 generations. We selected the tuned configuration that achieves the

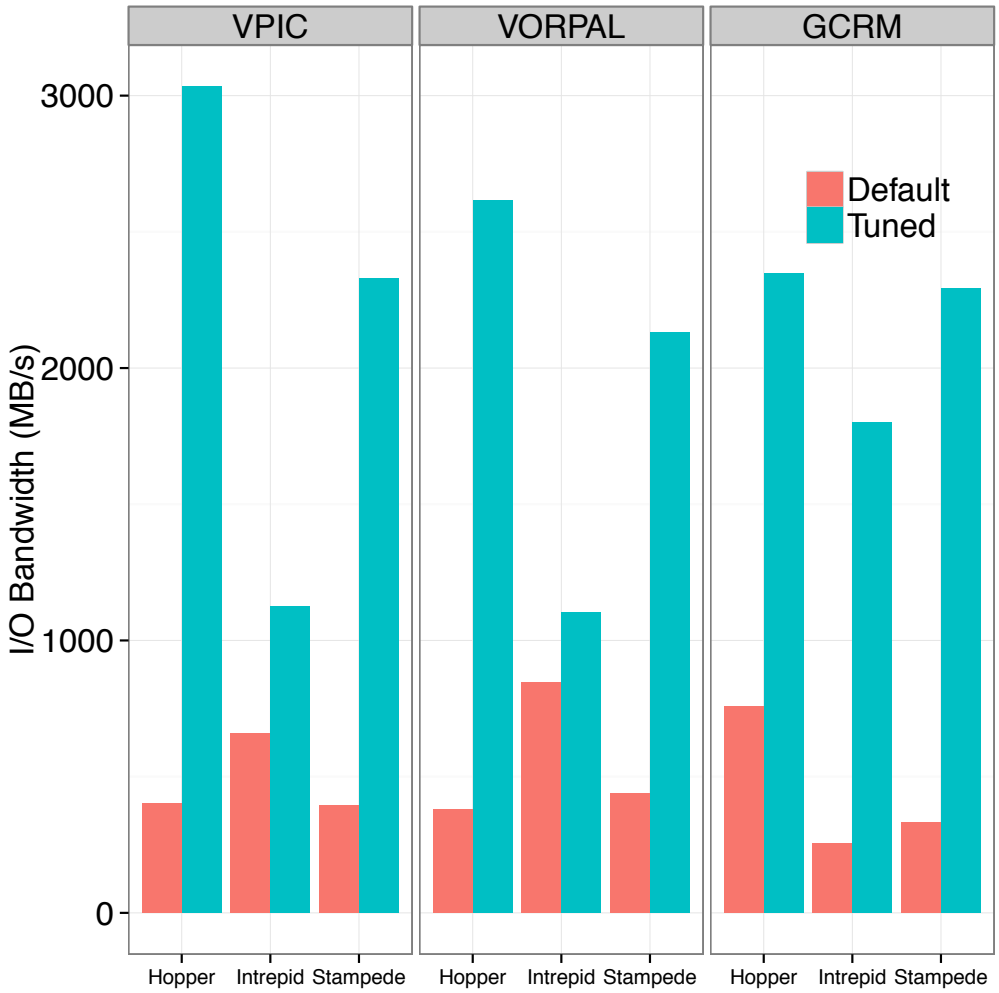


Figure 3.3: Throughput improvement for each I/O benchmark running on 128 cores

best I/O performance through the course of the GA evolution. Figure 3.3, 3.4 and 3.5 compare the tuned I/O rate with the default I/O rate for all applications on all HPC systems . We calculated the I/O rate as the ratio of the amount of data a benchmark writes into a HDF5 file at any given scale to the time taken to write the data. The time taken includes the overhead of opening, writing, and closing the HDF5 file. The overhead of HDF5 call interception by H5Tuner, which is included in the time taken, was negligibly small even at high core counts. The I/O rate on the y-axis is expressed in MB/s. Readers should note that the range of I/O rates shown in each of the three plots is different. The measured default I/O rate for a benchmark on a HPC platform is the average I/O rate we obtained after running the

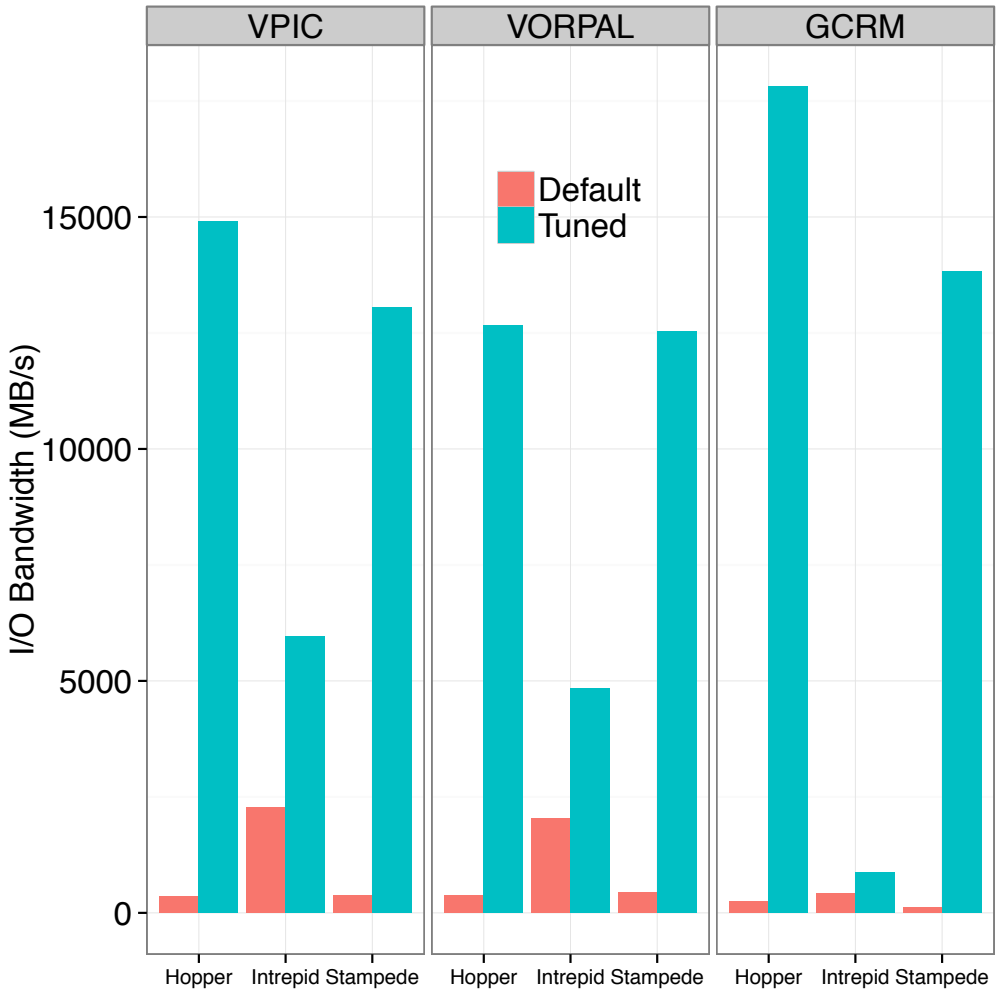


Figure 3.4: Throughput improvement for each I/O benchmark running on 2048 cores

benchmark three times. The default experiments correspond to the system default settings that a typical user of the HPC platform would encounter should he/she not have access to an auto-tuning framework.

Table 3.4 shows the raw I/O rate numbers (in MB/s) of the default and tuned experiments for all 24 experiments. We also show the speedup that the auto-tuned settings achieved over the default settings for each experiment. For all the benchmarks, platforms, and concurrencies, the speedup numbers are generally between 1.3X and 38X, with 48X, 50X, 70X, and 100X speedups in four cases. We note that the default I/O rates for the Intrepid platform are noticeably higher than those on Hopper and Stampede. Hence, the speedups on Hopper and Stampede with tuned parameters are much larger than those

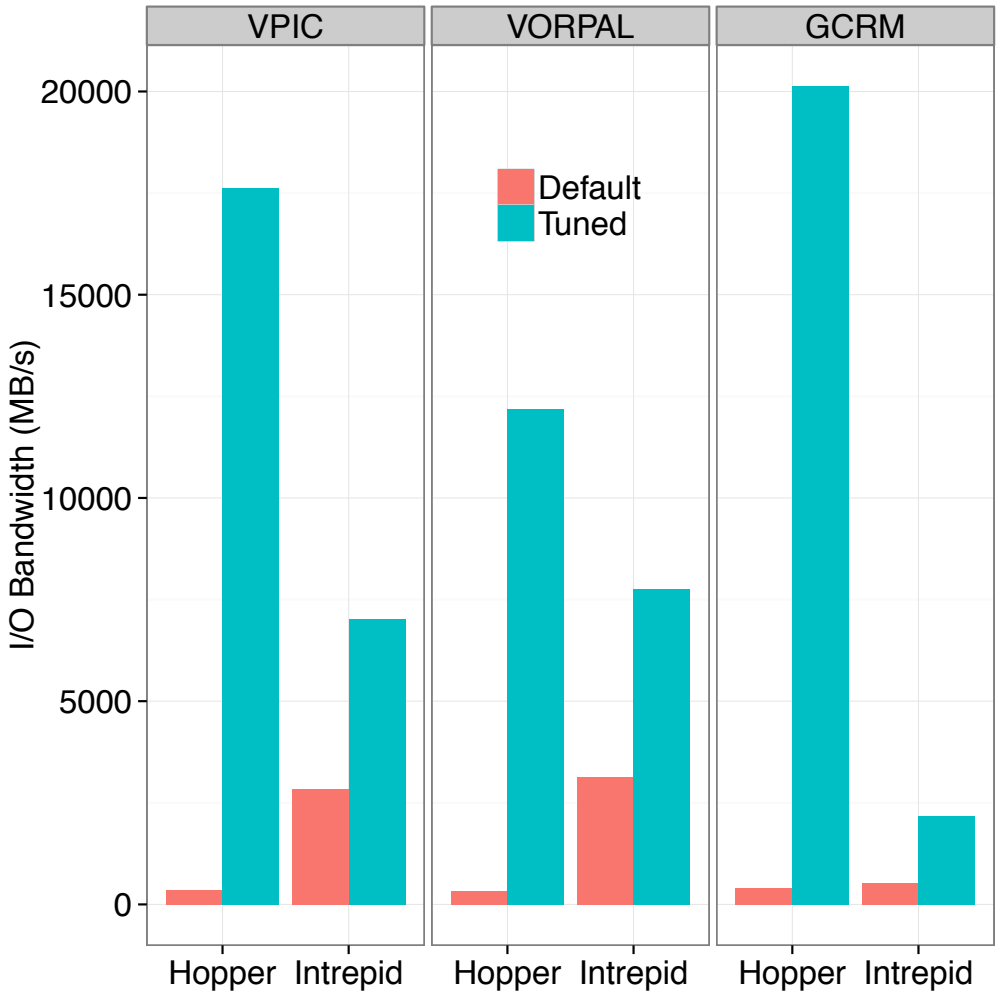


Figure 3.5: Throughput improvement for each I/O benchmark running on 4096 cores

on Intrepid.

3.4.2 Tuned Configurations

Table 3.5 shows the sets of tuned parameters for all benchmarks on all systems for the 2048-core experiments. We generally observed similar trends for the 128-core and 4096-core experiments. First, we note that the tuned parameters are different for all benchmarks and platforms. This highlights the strength of the auto-tuning framework: while I/O experts and sysadmins can probably recommend good settings for a few cases based on their experience, it is hard to encapsulate that knowledge and generalize it across

Application/ # Cores	Platform	Bandwidth (MB/s)								
		VPIC-IO			VORPAL-IO			GCRM-IO		
		Default	Tuned	Speedup	Default	Tuned	Speedup	Default	Tuned	Speedup
128	Hopper	400	3034	7.57	378	2614	6.90	757	2348	3.10
	Intrepid	659	1126	1.70	846	1102	1.30	255	1801	7.05
	Stampede	394	2328	5.90	439	2130	4.85	331	2291	6.90
2048	Hopper	365	14900	40.80	370	12669	34.16	240	17816	74.12
	Intrepid	2282	5964	2.61	2033	4842	2.38	414	870	2.10
	Stampede	380	13047	34.28	436	12542	28.70	128	13825	107.73
4096	Hopper	348	17620	50.60	320	12192	38.00	413	20136	48.67
	Intrepid	2841	7014	2.46	3131	7766	2.47	523	2177	4.16

Table 3.4: I/O rate and speedups of I/O benchmarks with tuned parameters over default parameters

multiple problem configurations.

VPIC-IO and VORPAL-IO on Hopper and Stampede have similar tuned parameters, i.e., `strp_fac`, `strp_unt`, `cb_nodes`, `cb_buf_size`, and `align`. On Intrepid, these two benchmarks include `bgl_nodes_pset`, `cb_buf_size`, `bglockless`, `IBM_largeblock_io`, and `align`. On all platforms, GCRM-IO achieved better performance with HDF5's chunking and alignment parameters, and Lustre parameters (stripe factor and stripe size) without the MPI-IO collective buffering parameters. We chose this parameter space for GCRM-IO as Howison et al. [4] demonstrated that HDF5 chunking provides a significant performance improvement for this I/O benchmark. Moreover, we show that the auto-tuning system is capable of searching a parameter space with multiple HDF5 tunable parameters. On Intrepid, GCRM-IO did not use GPFS tunable parameters because going through HDF5's MPI-POSIX driver avoids the MPI-IO layer, which is needed to set the GPFS parameters. Despite that, HDF5 tuning alone achieved 2X improvement.

We note some higher-level trends from Table 3.5. For the same concurrency and with the same benchmark, the tuned parameters are different on various platforms even with the same parallel file system. For example, although the VPIC-IO benchmark on Hopper and Stampede use the Lustre file system, their stripe settings to achieve the highest performance are different. The tuned parameters can be different on the same platform and concurrency for different benchmarks. For instance, the VPIC-IO and VORPAL-IO benchmarks obtain the highest I/O rates with different MPI-IO collective buffering settings and HDF5 alignment settings, whereas their Lustre settings are the

I/O Kernel	System	Tuned Parameters
VPIC-IO	Hopper	strp_fac=128, strp_unt=64MB, cb_nds=1024, cb_buf_size=64MB, align=(0,64K)
VPIC-IO	Intrepid	bgl_nodes_pset=512, cb_buf_size=128MB, bglockless=true, large-block_io=false, align=(8K, 1MB)
VPIC-IO	Stampede	strp_fac=128, strp_unt=8MB, cb_nds=512, cb_buf_size=8MB, align=(8K, 2MB)
VORPAL-IO	Hopper	strp_fac=128, strp_unt=16MB, cb_nds=1024, cb_buf_size=16MB, align=(1K,16K)
VORPAL-IO	Intrepid	bgl_nodes_pset=128, cb_buf_size=128MB, bglockless=true, large-block_io=true, align=(8K, 8MB)
VORPAL-IO	Stampede	strp_fac=160, strp_unt=2MB, cb_nds=512, cb_buf_size=2MB, align=(8K, 8MB)
GCRM-IO	Hopper	strp_fac=156, strp_unt=32MB, chunk_size=(1,26,327680)=32MB, align=(2K, 64KB)
GCRM-IO	Intrepid	chunk_size=(1,26,1048760)=1GB, align=(1MB, 4MB)
GCRM-IO	Stampede	strp_fac=160, strp_unt=32MB, chunk_size=(1,26,1048760)=1GB, align=(1MB, 4MB)

Table 3.5: Tuned parameters of all benchmarks on all the systems for 2048-core experiments

same. Similarly, the same benchmark at different concurrencies on the same platform has different tunable parameters. For example, at 128-cores (not shown in the table), VPIC-IO achieves tuned performance with 48 Lustre stripes and 32 MB stripe size, whereas at 2048-cores, VPIC-IO uses 128 stripes with 64 MB stripe size.

3.5 Conclusions

We have presented an auto-tuning framework for optimizing I/O performance of scientific applications. The framework is capable of transparently

optimizing all levels of the I/O stack, consisting of HDF5, MPI-IO, and Lustre/GPFS parameters, without requiring any modification of user code. We have successfully demonstrated the power of the framework by obtaining speedups between 2x and 100x across diverse HPC platforms, benchmarks, and concurrencies. Perhaps most importantly, we believe that the auto-tuning framework can provide a route to hiding the complexity of the I/O stack from application developers, thereby providing a truly performance portable I/O solution for scientific applications.

CHAPTER 4

IN-DEPTH ANALYSIS OF APPLICATIONS' I/O ACTIVITIES WITH RECORDER

In this chapter, we present our work in providing users with detailed runtime feedback to enable in-depth I/O analysis and tuning.

Understanding the I/O behavior of an HPC application is difficult task due to the complex interactions between multiple software components. As discussed in Section 3.1.1, the common I/O software stack found on many-current HPC systems consists of several layers. All layers in the I/O stack are designed to support portable data abstractions and performance optimizations. A high-level I/O library translates the applications data structures into a structured file format, such as HDF5[1] or NetCDF[3]. The middleware layer, which is typically an MPI-IO implementation, takes care of organizing and optimizing accesses from many processes. At the bottom, the parallel file system layer accesses to files stored on the storage hardware using bytes and blocks.

In a hierarchical I/O stack, the layers provide bridges between the data representations of adjacent levels and offer essential abstractions to users. The layers help hide complex implementation details and employ optimization techniques designed to improve performance. Unfortunately, since each layer is normally treated as a black box, optimizations are seldom coordinated across layers and the source of performance bottlenecks can be extremely difficult to determine. A multi- level I/O tracing and trace data analysis tool that presents a view of the function call flow through the entire I/O stack can expose cause and effect relationships across layers and make the origin of performance bottlenecks more apparent.

To the best of our knowledge, while there are several tracing facilities for the MPI-IO and POSIX I/O levels (will be discussed in Section 4.1), none of the currently available tracing tools work with higher-level I/O libraries such as HDF5. We believe that tracing I/O functions at higher levels in the stack is important because events closer to the application better reflect inherent

application characteristics and are more intuitive to analyze. In addition, insights into all levels of the I/O stack are necessary in order to get a full picture of interactions between layers and to identify sources of performance bottlenecks.

In this chapter, we argue that a multi-level I/O tracing and trace data analysis tool can help end users understand the behavior of their application and I/O subsystem, and can provide insights into the source of I/O performance bottlenecks. We have designed and developed a tracing tool, Recorder, that captures I/O function calls at the HDF5, MPI-IO and POSIX I/O levels along with key parameters. Recorder provides a multi-level view and helps users obtain an overall understanding of the I/O activities of the application. We demonstrate how Recorders trace output can be used to investigate I/O activity and identify performance inefficiencies in two I/O benchmarks running on a leading edge HPC platform. We also demonstrate the effectiveness of Recorder as an aid to identifying a performance bottleneck in HDF5s current implementation of metadata read. We believe that a multi-level I/O tracing framework can provide key insights to end users and I/O library developers working to improve I/O on HPC platforms.

Our work has resulted in a publication in the 5th Workshop on Interfaces and Architectures for Scientific Data Storage (IASDS), 2013 [9].

The remainder of this chapter is organized as follows: We review related work in Section 4.1 and describe our framework in Section 4.2. In Section 4.3, we evaluate the effectiveness of our framework. Finally, we summarize our current efforts in Section 4.4, discuss open issues, and outline future work.

4.1 Related Work

There are two main types of tools often used in I/O performance diagnosis and optimization, namely, profiling and tracing tools. A profiling tool normally provides performance statistics and summaries, while a tracing tool provides more details and records events based on time. There has been much work done in this area, such as Darshan [28], IOPin [60], RIOT [20], ScalaHTrace [61] and //TRACE [22]. One common theme among these tools is that they all work at multiple levels, namely the MPI-IO and POSIX I/O levels or the file system of the I/O stack. This demonstrates the benefit of

having a multi-level view of I/O activity that cuts through all the complexity of today's parallel I/O stacks.

The Darshan library is a profiling library that characterizes I/O workloads and provides insights into the I/O characteristics of an application with negligible overhead. Darshan captures MPI-IO routines using the PMPI interface and POSIX routines by inserting wrapper functions via the GNU linker options. Darshan characterizes the application by providing statistics and cumulative timing information. The advantage of this approach is that the collected data is small enough to be stored in memory. The information recorded by Darshan includes counters for MPI-IO and POSIX I/O operations, counters for MPI-IO datatypes and access patterns, and cumulative information about the number of bytes read or written or time spent in operations. Its lightweight design allows Darshan to be deployed full time for workload characterization of large systems and it has been enabled by default on IBM Blue Gene/P series at Argonne National Laboratory (ANL) [2] and a Cray XE6 system at National Energy Research Scientific Computing Center (NERSC) [26]. However, the compact nature of Darshan's profiling information means that other approaches are needed to understand the detailed behavior of an application or an I/O library implementation.

IOPin is another profiling tool that instruments the MPI library and PVFS file system. IOPin gathers information such as rank, mpi_call_id, pvfs_call_id, I/O type (read/write), and latency and stores them in a database. IOPin makes use of runtime binary instrumentation of the MPI library and the file system in order to incur low overhead and also avoid the need of recompilation of the program. One distinguishing feature of IOPin is its ability to gather and correlate function calls at the MPI-IO and file system levels.

The RIOT I/O toolkit, created at the University of Warwick, intercepts I/O function calls at the MPI-IO and POSIX I/O levels. For each function, it records timestamp, the size of data written and the file offset. The toolkit also includes a post-processing tool to create statistical and graphical report of the application's I/O activities. This work has further demonstrated the use of tracing facilities in performance analysis and tuning. We share the same vision with the authors and provide a framework that also captures the I/O activities of a high level I/O library. We believe that providing more details about the application's I/O behaviors will be beneficial to users in identifying potential performance bottlenecks (either of the application or

the I/O library implementation) as well as understanding and tuning the application.

Sigovan et al. [62] have developed a system that records both communication and I/O traces for performance analysis of large scale parallel I/O systems. They record MPI-IO and POSIX I/O function calls, I/O requests sent from I/O nodes to the storage cluster and the accesses on the storage hardware. They also present a novel visual analysis method to visualize the connections between components to identify problematic areas. They focus on the performance analysis of the I/O *system* while we concentrate on the analysis and optimization of an I/O *application*.

Earlier versions of the HDF5 library (1.4.x) included a tracing framework developed by the Pablo Research Group [63]. The instrumented HDF5 library places hooks into the HDF5 code that call Pablo's routines at the entrance and exit of intrumented HDF5 functions. Users need to modify their code to enable tracing and to specify the list of HDF5 functions to be traced. The Pablo Trace library is not supported anymore and no longer available.

//TRACE and ScalaHTrace both include tracing and a trace replaying engine. //TRACE's tracing engine discovers inter-node data dependencies. Because //TRACE mainly focuses on high replay accuracy, the time overhead is large due to identifying these data dependencies. ScalaHTrace captures both communication and I/O activities. Using a novel compression technique, ScalaHTrace has been able to keep the trace file of near constant size at different scales. Its replay engine uses a distributed approach to deterministically replay the traces without decompressing them. Both //TRACE and ScalaHTrace mainly focus on trace replaying rather than performance diagnosis and optimization. Therefore, the traces generated by them mainly contain timing information and not much more, making it hard to pin point the problematic areas and provide an explanation for performance degradation.

In [64], Konwinski et al. propose a taxonomy for cataloging features of I/O tracing frameworks based on their survey of three existing packages (LANL-TRACE [24], Tracefs [25], and //TRACE). We found the proposed taxonomy very useful in considering and describing the features of Recorder.

Feature	Recorder
Parallel file system compatibility	Yes
Ease of installation and use	Easy
Anonymization	Medium
Event types	System calls and library calls
Control of trace granularity	Yes
Replayable trace generation	Yes
Trace replay fidelity	N/A
Reveals dependencies	No
Intrusive vs. Passive	Passive
Analysis tools	No
Trace data format	Both binary and human readable
Accounts for time skew and drift	No
Elapsed time overhead	N/A

Table 4.1: Recorder Framework Summary Table

4.2 Framework

There are different ways to record I/O events, either via compile time wrappers for static executables or dynamic library preloading for dynamic executables. For statically linked executables, the tracing tool can use the -wrap functionality of the linker to create a compiler wrapper that includes all necessary link options and libraries, and link to the application at compile time. The tracing tool can also be built as a shared library to use with dynamically linked executables. As a shared library, the tracing library uses function interpositioning to prioritize itself over the execution in the library stack. When the application makes a function call, the tracing library will intercept the call, record information about the function, then call the standard version of the function to complete the operation that the application requested. We chose to build our tracing tool, Recorder, as a dynamic library so that it does not require source code modification or recompilation of the application.

The features of Recorder based on the taxonomy proposed in [64] are summarized in Table 4.1 and briefly described here. Recorder is designed to work with parallel file systems and does not require any modifications to application or I/O library source code. It provides a medium level of anonymization to protect sensitive data by, for example, recording only the size of a message and not its value. Recorder can capture I/O functions at

the HDF5, MPI I/O, and POSIX I/O layers, and users can specify the layers to be traced when Recorder is compiled, providing some control over trace granularity. The traces were designed with replay in mind, and a replay engine is planned. Recorder provides a passive method of tracing events through the use of dynamic library preloading. Development of analysis tools is future work, while direct inspection of the trace files is possible now as both binary and human readable formats are supported.

As mentioned above, we chose to implement Recorder as a dynamic library. Every processor creates a trace file once the program calls the `MPI_Init()` function. Figure 4.1 depicts the way Recorder works. At each layer, for each function Recorder collects a timestamp, function name and its parameters. A sample ouput of the trace is shown in Figure 4.2.

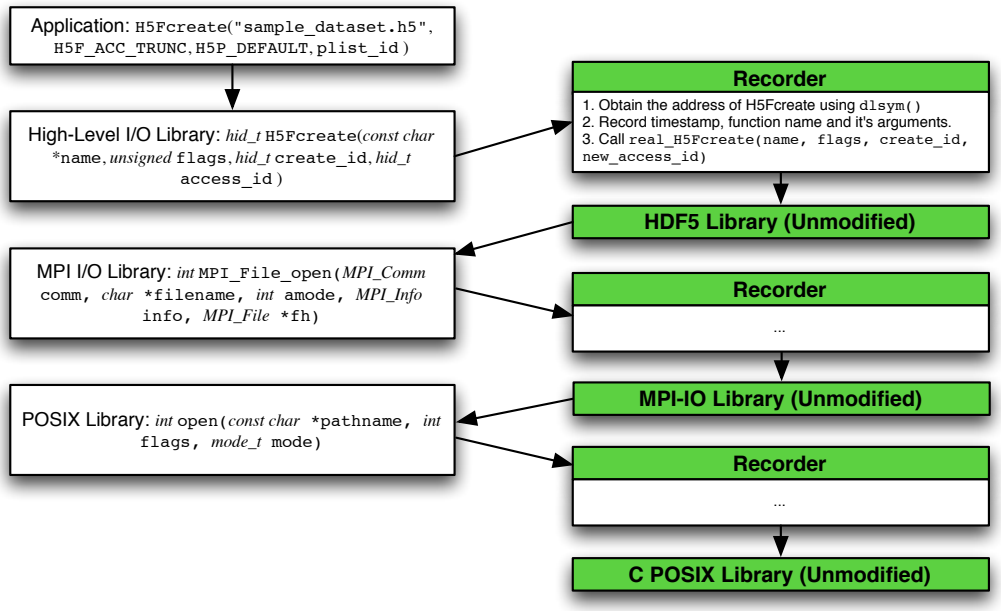


Figure 4.1: Dynamic instrumentation of the I/O stack by Recorder

```
1377833582.00000 H5Dwrite(83886080,50331691,0,0,167772179)
1377833582.00000 MPI_File_write_at(fh=16542184, offset=2144,
buf=*buf, count=1073741824, datatype=MPI_BYTE, status=894040288)
1377833582.00000 write(sample_dataset.h5, void *buf, 1073741824)
```

Figure 4.2: Sample trace generated by Recorder

4.3 Example Use Case

We demonstrate our framework's applicability by helping the user to understand the I/O activities of an HPC application and the I/O subsystem.

Application developers can use Recorder to understand the effect of using a hint. Hints are normally used by application developers to provide information such as I/O patterns, enabling or disabling an optimization. Using hints may allow an implementation to have better performance or control the way an optimization is performed. However, hints can be ignored by the underlying system implementations if they do not support or understand the hints. Therefore, an application developer might have false assumptions about the hints being used. For example, one of the most commonly used hints is `cb_nodes`, which sets the maximum number of aggregators to be used in the Collective Buffering optimization. However, on the BlueGene/P system, this hint is ignored and the `bgl_nodes_pset` hint is used instead. `bgl_nodes_pset` specifies how many aggregators to use per pset, where a pset is a collection of an ION and its compute nodes. On BlueGene/P, each pset consists of 1 dedicated I/O node (ION) and 64 compute nodes. Unfortunately, not every user of the BlueGene system is aware of this. By using Recorder, one can realize that changing the `cb_nodes` value does not have any effect on the application's I/O activities, or see the effect of `bgl_nodes_pset`.

We will demonstrate this use case using the VPIC-IO benchmark. Figure 4.3 depics VPIC-IO benchmark's I/O patterns. Each process writes 8M particles and each particle has eight variables. The size of each variable on each process is 32MB. VPIC-IO writes data for one time step only.

We ran the experiments using 128 processors on Intrepid and compared the results of Recorder's logs when we

1. use default settings.
2. set `cb_nodes` to 64.
3. use the `bgl_nodes_pset` hint.

Table 4.2 summarizes the value of the corresponding parameters in our experiments.

In the the default setting case, the Blue Gene/P system chose to use 8 aggregators per pset. This fact is reflected in the trace files. In aggregators' log

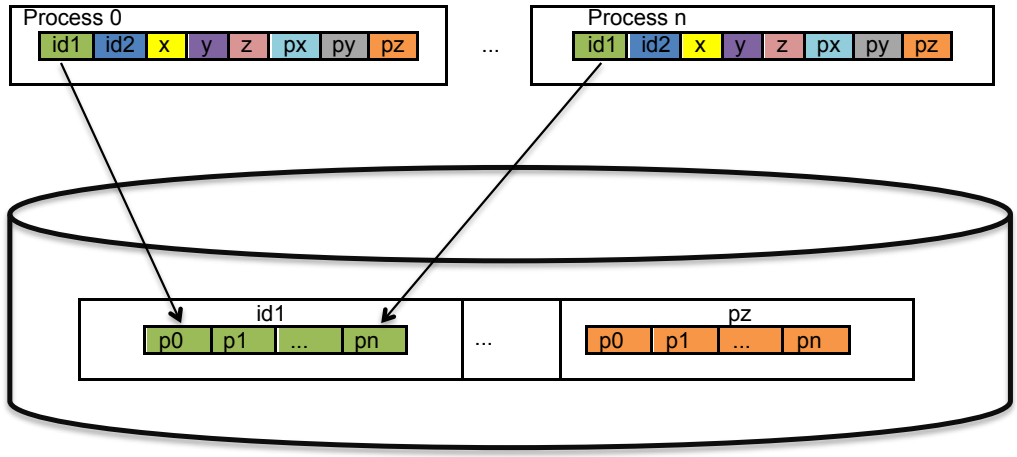


Figure 4.3: VPIC-IO pattern

Parameter	**Default Setting**	**cb_nodes Setting**	**bgl_nodes_pset Setting**
Collective buffering nodes	N/A	64	N/A
Collective buffer size	16 MB	16 MB	16 MB
`bgl_nodes_pset`	8	8	64
I/O bandwidth	**362 MB/s**	**360 MB/s**	**405 MB/s**

Table 4.2: VPIC-IO parameters and I/O bandwidth

files, the `write` function is called to write out the data. In non-aggregators' log files, there is no `write` function call. Log files of processor 0 and processor 8 are shown in Listing 4.1 and Listing 4.2. There are 16 `write` calls and each call writes out 16MB of data (which is the size of `cb_buffer_size`). We have a total of 16 calls because each process has 32MB of data and the aggregators collect data from 7 other processes and itself.

When the `cb_nodes` is set to 64, there is no change in the I/O activities captured in the log files created by Recorder. When the `bgl_nodes_pset` is set to use 64 aggregator per pset which means every process is an aggregator. Therefore, in this case, all the log files show that every process calls to `write` function to write its own data using 2 `write` calls, each with the buffer of size 16MB.

Listing 4.1: Default setting: log.0

```
..512.63856 H5Dwrite(83886080,..)
..512.63871 MPI_Allreduce(..)
..512.63900 MPI_Comm_rank(..)
..512.63904 MPI_File_set_view(fh=40052760, ..)
..512.63925 MPI_File_write_at_all (fh=40052760,.,count
   =268435456,..)
..512.69349 write(./dataset.h5part, buf=0xa2900010,
   16777216)
..513.59258 write(./dataset.h5part, buf=0xa2900010,
   16777216)
..514.29326 write(./dataset.h5part, buf=0xa2900010,
   16777216)
...
..523.16982 H5Dclose(83886080)
```

Listing 4.2: Default setting: log.8

```
..512.63901 H5Dwrite(83886080,..)
..512.63916 MPI_Allreduce(..)
..512.63954 MPI_Comm_rank(..)
..512.63958 MPI_File_set_view(fh=40052760, ..)
..512.63979 MPI_File_write_at_all(fh=40052760,.,count
   =268435456,..)
..523.17036 H5Dclose(83886080)
```

4.4 Conclusions

We have presented our first step towards a feedback-driven approach to optimize I/O performance by providing a multi-level I/O tracing framework, Recorder. Recorder is built as a dynamic library so it does not require any modification or recompilation of the application. We showed how Recorder can be very useful in providing an in-depth analysis of I/O activities of the application and the I/O subsystem. We believe that by providing users

with feedback and knowledge about I/O activities of their application, users can then apply the knowledge to improve their optimization effort by, for example, providing a better search space for the auto-tuning framework, which in turn will help to reduce the long runtime issue associated with genetic algorithms.

CHAPTER 5

OPEN ISSUES AND CONCLUSIONS

In this thesis, we address the problem of optimizing I/O performance for HPC applications using a number of approaches, ranging from Auto-tuning to analyzing applications' I/O feedbacks.

We address the research problem by analyzing the I/O behavior feedback of the application provided by lightweight, high-level I/O instrumentation tool like Darshan. Lightweight tools such as Darshan can augment traditional benchmarking and tracing tools, and provide an overall understanding of the I/O behavior of applications, workloads, and platforms. We used Darshan I/O logs to provide a broad view of I/O behavior on three leading HPC platforms. Our results lead us to believe that while tremendous progress has been made in hardware and software research for HPC I/O, gaps remain in the adoption of best practices by scientific application developers. For instance, strategies such as usage of text files and raw, low-level POSIX I/O calls will be untenable on future platforms; adoption of higher-level I/O libraries can help increase the longevity of codes on future generations of supercomputers. HPC I/O specialists need to ensure that application developers understand the tradeoffs in different ways of performing I/O, perhaps through I/O boot camps and tutorials offered in cooperation with platform administrators. Our results also lead us to believe that while much research effort is invested in extreme-scale testing and optimization, a large fraction of the HPC community has modest-scale metadata and data challenges; designers of HPC facilities must take these needs into account when designing and provisioning I/O resources. We believe that tools such as Darshan can give platform administrators critical insights into system utilization; early and proactive intervention into suboptimal I/O behavior can greatly enhance the utilization of a platform's existing HPC resources.

We have presented an auto-tuning framework for optimizing I/O performance of scientific applications. The framework is capable of transparently

optimizing all levels of the I/O stack, consisting of HDF5, MPI-IO, and Lustre/GPFS parameters, without requiring any modification of user code. We have successfully demonstrated the power of the framework by obtaining speedups between 2x and 100x across diverse HPC platforms, benchmarks, and concurrencies. Perhaps most importantly, we believe that the auto-tuning framework can provide a route to hiding the complexity of the I/O stack from application developers, thereby providing a truly performance portable I/O solution for scientific applications.

There are several open issues with this approach. In this research, we have focused on developing and testing the auto-tuning system on multiple platforms using different I/O *benchmarks*. We have not addressed the issue of how one can generalize the results from running benchmarks to arbitrary applications. We believe that I/O motifs or patterns are the key to this generalization problem. In the future, we will characterize and enumerate prototypical motifs and use the current auto-tuning framework to populate a database of good configurations for these motifs. We will then implement an intelligent runtime system, which will be capable of extracting I/O motifs from arbitrary applications and consulting the performance database to suggest an optimal I/O strategy. Figure 5.1 illustrates our proposed architecture for an intelligent runtime system that could address this challenge.

Another challenging issue is dealing with runtime noise and dynamic interference from other users, which is a fact of life in production HPC facilities. While our auto-tuning framework has presented compelling results, we are assuming that the user will encounter a runtime workload which is comparable to the one encountered during the auto-tuning process. We believe that measuring noise and interference during the tuning process and deriving models for projecting their effect at runtime will be key in tackling this hard problem.

The long runtime of the GA is a potential concern and it is worsened due to the nature of I/O benchmarking. Each evaluation's runtime is on the order of minutes and keeps increasing when the concurrency increases. For example, the average total running time of each execution of GCRM-IO at 4096 cores is 1500 seconds or 25 minutes. As a matter of fact, we have spent a total of 2.4 million computing hours on Intrepid to collect the experiment data. We need to address this issue to make the auto-tuning framework more practical.

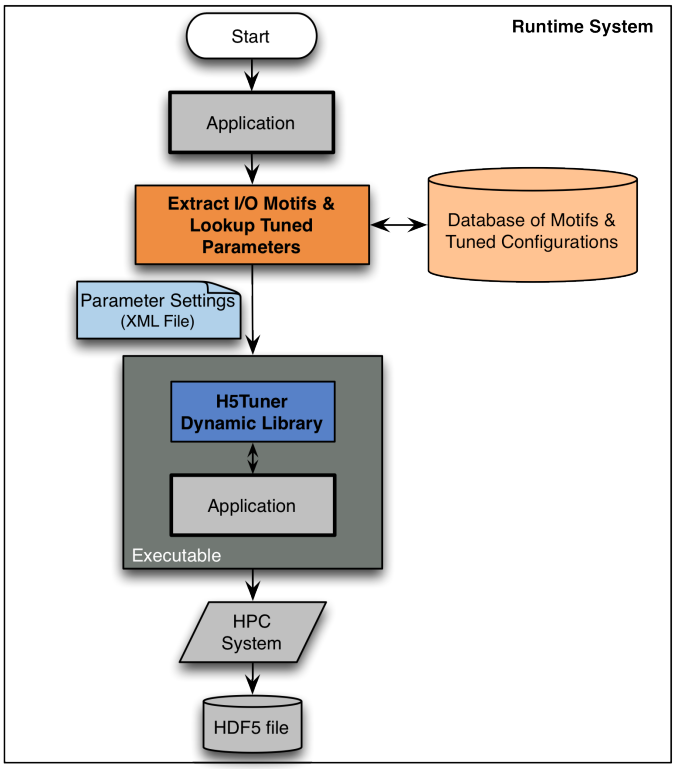


Figure 5.1: Proposed architecture for the Intelligent Runtime System

In the future, we plan to address the issue of generalizing the results from running a collection of representative benchmarks to arbitrary applications using I/O motifs. We are also looking into several approaches to further reduce the runtime of our framework, such as providing users with performance feedback from one run to incorporate users' expertise into the next run, or using a performance model to guide the tuning process.

Lastly, we develop a multi-level tracing framework that provides a much more detailed feedback for application's I/O runtime behavior. We showed how Recorder can be very useful in providing an in-depth analysis of I/O activities of the application and the I/O subsystem. We believe that by providing users with feedback and knowledge about I/O activities of their application, users can then apply the knowledge to improve their optimization effort.

There are several open issues of this framework that we have not addressed in this research. A potential addition that would be very useful to users is to create a trace analysis or visualization tool that can help to identify the I/O bottleneck or automatically draw useful conclusions from large scale runs. As

the scale increases, the generated traces capture thousands of events that will be too much for manual inspection. Users also need to have knowledge of the parallel I/O software stack to understand the traces and relate them back to the I/O activities of an application. A trace analysis/visualization tool will be useful even to users without the expertise in I/O to diagnose and optimize their application. The trace analysis tool needs to have heuristics to suggest ways to improve the performance, such as changing to an appropriate mode (independent or collective I/O), matching values of tunable parameters at different layers or even changing the I/O paradigm.

In order to support effective cross-level trace analysis, we need to correlate lower-level functions with the higher-level function where they originated. The correlation will be essential for multi-threaded applications and those with asynchronous I/O, because one cannot simply use the order of events in the trace file to infer which operation caused another under those circumstances.

Finally, we plan to create a parametric replay tool that can support replaying the function calls recorded in the trace files with varied I/O modes. After identifying the problematic area in the application, the user will want to fix the problem by adjusting the way I/O is performed. A parametric replay tool can help the user to try different I/O methods and translate back to a good optimization for the original application.

REFERENCES

[1] D. Kothe and R. Kendall, "Computational Science Requirements for Leadership Computing," https://www.olcf.ornl.gov/wp-content/uploads/2010/03/ORNL_TM-2007_44.pdf, July 2007.

[2] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and Improving Computational Science Storage Access Through Continuous Characterization," *Journal ACM Transactions on Storage*, vol. 7, no. 3, pp. 8:1–8:26, Oct. 2011.

[3] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp, "Parallel I/O prefetching using MPI file caching and I/O signatures," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08, 2008, pp. 44:1–44:12.

[4] M. Howison, Q. Koziol, D. Knaak, J. Mainzer, and J. Shalf, "Tuning HDF5 for Lustre File Systems," in *Proceedings of 2010 Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS10)*, Heraklion, Crete, Greece, Sep. 2010, lBNL-4803E.

[5] A. Uselton, M. Howison, N. J. Wright, D. Skinner, N. Keen, J. Shalf, K. L. Karavanic, and L. Oliker, "Parallel I/O performance: From events to ensembles," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 1–11, 2010.

[6] Y. Yin, S. Byna, H. Song, X.-H. Sun, and R. Thakur, "Boosting Application-Specific Parallel I/O Optimization Using IOSIG," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, ser. CCGRID '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 196–203.

[7] "Fastest USB Thumb Drive, 14 February 2014," http://www.maximumpc.com/fast_usb_thumb_drive_2014.

[8] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir, "Taming parallel i/o complexity with auto-tuning," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. ACM, 2013, pp. 68:1–68:12.

[9] H. Luu, B. Behzad, R. A. Aydt, and M. Winslett, "A multi-level approach for understanding I/O activity in HPC applications." in *5th Workshop on Interfaces and Architectures for Scientific Data Storage (IASDS)*. IEEE, 2013, pp. 1–5.

[10] B. K. Pasquale and G. C. Polyzos, "A Static Analysis of I/O Characteristics of Scientific Applications in a Production Workload," in *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '93. New York, NY, USA: ACM, 1993, pp. 388–397.

[11] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. Best, "File-Access Characteristics of Parallel Scientific Workloads," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 10, pp. 1075–1089, October 1996.

[12] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, D. D. E. Long, and T. T. McLarty, "File System Workload Analysis for Large Scale Scientific Computing Applications," in *Proceedings of the 21st IEEE/12th NASA GODDARD conference on Mass Storage System and Technologies*, 2004, pp. 139–152.

[13] S. Saini, D. Talcott, R. Thakur, P. A. Adamidis, R. Rabenseifner, and R. Ciotti, "Parallel I/O Performance Characterization of Columbia and NEC SX-8 Superclusters." in *IPDPS*. IEEE, 2007, pp. 1–10.

[14] Y. Kim, R. Gunasekaran, G. Shipman, D. Dillow, Z. Zhang, and B. Settlemyer, "Workload characterization of a leadership class storage cluster," in *Petascale Data Storage Workshop (PDSW), 2010 5th*, Nov 2010, pp. 1–5.

[15] S. Saini, J. Rappleye, J. Chang, D. Barker, P. Mehrotra, and R. Biswas, "I/O performance characterization of Lustre and NASA applications on Pleiades." in *HiPC*. IEEE, 2012, pp. 1–10.

[16] "FLASH," http://www.flash.uchicago.edu/site/.

[17] "GCRM," https://svn.pnl.gov/gcrm.

[18] "Nek5000," http://nek5000.mcs.anl.gov/index.php/Main_Page.

[19] "CESM," http://www2.cesm.ucar.edu/.

[20] S. A. Wright, S. D. Hammond, S. J. Pennycook, R. F. Bird, J. A. Herdman, I. Miller, A. Vadgama, A. H. Bhalerao, and S. A. Jarvis, "Parallel File System Analysis Through Application I/O Tracing." *Computer Journal*, vol. 56, no. 2, pp. 141–155, 2013.

[21] K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth, "Scalable I/O Tracing and Analysis," in *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, ser. PDSW '09. New York, NY, USA: ACM, 2009, pp. 26–31.

[22] M. P. Mesnier, M. Wachs, R. R. Sambasivan, J. Lpez, J. Hendricks, G. R. Ganger, and D. R. O'Hallaron, "//TRACE: Parallel Trace Replay with Approximate Causal Events." in *File Systems and Storage Technologies*. USENIX, 2007, pp. 153–167.

[23] N. J. Wright, W. Pfeiffer, and A. Snavely, "Characterizing parallel scaling of scientific applications using IPM," in *In Proc. of the 10th LCI International Conference on High-Performance Clustered Computing*, 2009.

[24] "LANL-Trace," http://institutes.lanl.gov/data/software/index.php#lanl-trace.

[25] A. Aranya, C. P. Wright, and E. Zadok, "Tracefs: A File System to Trace Them All," in *In Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004*. USENIX Association, 2004, pp. 129–143.

[26] P. Carns, Y. Yao, K. Harms, R. Latham, R. Ross, and K. Antypas, "Production I/O Characterization on the Cray XE6," in *Proceedings of the Cray User Group meeting 2013 (CUG 2013)*, May 2013.

[27] G. Navarro, "A guided tour to approximate string matching," *ACM Comput. Surv.*, vol. 33, no. 1, pp. 31–88, Mar. 2001.

[28] P. H. Carns, R. Latham, R. B. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 Characterization of Petascale I/O Workloads," in *Proceedings of the First Workshop on Interfaces and Abstractions for Scientific Data Storage*, New Orleans, LA, USA, 09/2009 2009.

[29] "Edison File Systems," https://www.nersc.gov/users/computational-systems/edison/file-storage-and-i-o/.

[30] H. Shan, K. Antypas, and J. Shalf, "Characterizing and predicting the i/o performance of hpc applications using a parameterized synthetic benchmark," in *In Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC)*, 2008.

[31] V. Anantharaj, F. Foertter, W. Joubert, and J. Wells, "Approaching exascale: Application requirements for OLCF leadership computing," https://www.olcf.ornl.gov/wp-content/uploads/2013/01/OLCF_Requirements_TM_2013_Final1.pdf, 2013.

[32] S. Langer, B. Still, D. Hinkel, B. Langdon, and E. Williams, "A pF3D case study of obtaining good I/O performance while running on over 100,000 processors," http://visitbugs.ornl.gov/attachments/43/Langer-pf3d-IO-study-Mar2011-final1.pdf.

[33] B. Behzad, S. Byna, S. M. Wild, Prabhat, and M. Snir, in *The 23th International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2014.

[34] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated Empirical Optimization of Software and the ATLAS Project," *Parallel Computing*, vol. 27, no. 1–2, pp. 3–35, 2001.

[35] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, vol. 3. IEEE, 1998, pp. 1381–1384.

[36] B. Jeff, A. Krste, C. Chee-Whye, and D. Jim, "Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology," in *Proceedings of the 11th international conference on Supercomputing*, ser. ICS '97, 1997, pp. 340–347.

[37] R. Vuduc, J. Demmel, and K. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Proceedings of SciDAC 2005, Journal of Physics: Conference Series*, 2005.

[38] S. Williams, K. Datta, J. Carter, L. Oliker, J. Shalf, K. A. Yelick, and D. Bailey, "PERI: Autotuning Memory Intensive Kernels for Multicore," in *Journal of Physics, SciDAC PI Conference: Conference Series: 123012001*, 2008.

[39] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, ser. SC '07, 2007, pp. 38:1–38:12.

[40] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08, 2008, pp. 4:1–4:12.

[41] K. Seymour, H. You, and J. Dongarra, "A comparison of search heuristics for empirical code optimization," in *Cluster Computing, 2008 IEEE International Conference on*, 29 2008-oct. 1 2008, pp. 421 –429.

[42] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman, "Heuristics for Scheduling Parameter Sweep Applications in Grid Environments," 2000.

[43] T. Bäck and H.-P. Schwefel, "An overview of evolutionary algorithms for parameter optimization," *Evol. Comput.*, vol. 1, no. 1, pp. 1–23, Mar. 1993.

[44] K. Deb, A. Anand, and D. Joshi, "A Computationally Efficient Evolutionary Algorithm for Real-Parameter Optimization," 2002.

[45] A. Tiwari and J. K. Hollingsworth, "Online adaptive code generation and tuning," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 879–892.

[46] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch, "A multi-objective auto-tuning framework for parallel codes," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 10:1–10:12.

[47] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes, "Minerva: An automated resource provisioning tool for large-scale storage systems," *ACM Trans. Comput. Syst.*, vol. 19, no. 4, pp. 483–518, Nov. 2001.

[48] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch, "Hippodrome: Running Circles Around Storage Administration," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, ser. FAST '02. Berkeley, CA, USA: USENIX Association, 2002.

[49] J. Strunk, E. Thereska, C. Faloutsos, and G. R. Ganger, "Using utility to provision storage systems," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, ser. FAST'08. USENIX Association, 2008, pp. 21:1–21:16.

[50] Y. Chen, M. Winslett, Y. Cho, S. Kuo, and C. Y. Chen, "Automatic Parallel I/O Performance Optimization in Panda," in *In Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1998, pp. 108–118.

[51] Y. Chen, M. Winslett, S.-w. Kuo, Y. Cho, M. Subramaniam, and K. Seamons, "Performance modeling for the panda array I/O library," in *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, ser. Supercomputing '96, 1996.

[52] Y. Chen, M. Winslett, Y. Cho, and S. Kuo, "Automatic parallel i/o performance optimization using genetic algorithms," in *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on*, jul 1998, pp. 155 –162.

[53] W. Yu, J. Vetter, and H. Oral, "Performance characterization and optimization of parallel I/O on the Cray XT," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, april 2008, pp. 1 –11.

[54] H. You, Q. Liu, Z. Li, and S. Moore, "The Design of an Auto-tuning I/O Framework on Cray XT5 System."

[55] B. Behzad, J. Huchette, H. Luu, R. Aydt, S. Byna, Y. Yao, Q. Koziol, and Prabhat, "A Framework for Auto-tuning HDF5 Application," in *Poster at ACM Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC 2013, 2008.

[56] S. Byna, J. Chou, O. Rübel, Prabhat, and et al., "Parallel I/O, analysis, and visualization of a trillion particle simulation," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012, pp. 59:1–59:12.

[57] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O performance challenges at leadership scale," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 40:1–40:12.

[58] K. Schulz, "Experiences from the Deployment of TACC's Stampede System," http://www.hpcadvisorycouncil.com/events/2013/Switzerland-Workshop/Presentations/Day_1/7_TACC.pdf, March 2013 2013.

[59] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO," in *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, ser. FRONTIERS '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 182–.

[60] S. J. Kim, S. W. Son, W. keng Liao, M. T. Kandemir, R. Thakur, and A. N. Choudhary, "Iopin: Runtime profiling of parallel i/o in hpc systems," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE Computer Society, Nov 2012, pp. 18–23.

[61] X. Wu, K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth, "Probabilistic Communication and I/O Tracing with Deterministic Replay at Scale," in *ICPP*. IEEE, 2011, pp. 196–205.

[62] C. Sigovan, C. Muelder, K.-L. Ma, J. Cope, K. Iskra, and R. B. Ross, "A visual network analysis method for large-scale parallel i/o systems." in *IPDPS*. IEEE Computer Society, 2013, pp. 308–319.

[63] "Pablo research group," http://www.renci.org/focus-areas/project-archive/pablo.

[64] A. Konwinski, J. Bent, J. Nunez, and M. Quist, "Towards an I/O Tracing Framework Taxonomy," in *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing '07*, ser. PDSW '07. New York, NY, USA: ACM, 2007, pp. 56–62.

APPENDIX A

MYSQL AND R SCRIPTS

This appendix lists the MySQL and R scripts used to create the graphs in Chapter 2. Example scripts are shown for Mira. Scripts for Edison and Intrepid are similar. Results from MySQL query are saved to .csv files to feed into R.

A.1 Section 2.2.4 scripts

Listing A.1: MySQL script to query data for Figure 2.4

```
mysql> select nprocs, total_bytes, "Mira" as platform
    from Jobs_header, Jobs_perf where Jobs_header.
    logfilename = Jobs_perf.logfilename;
```

Listing A.2: R scripts to draw Figure 2.4

```
> across <- read.csv("./jobsizeacross_platforms.csv")
> ylab=c("1","10","100","1K","10K","100K","1M")
> boxplot(log10(nprocs)~platforms, data = across,yaxt
    ="n", ylab="Number of processes", main = "Number
    of processes per job across platforms", cex.lab
    =1.5, cex.axis=1.5, cex.main=1.5, cex.sub=1.5)
> axis(2,at=seq(0,6,1),labels=ylab,las=2)
```

Listing A.3: MySQL script to query data for Figure 2.6

```
mysql> select exe_name, max(nprocs) as max_jobsize,
    max(total_bytes) as max_datasize, "Mira" as
    platform from Jobs_header, Jobs_perf where
```

Listing A.3 (Cont.):

```
    Jobs_header.logfilename = Jobs_perf.logfilename
    group by exe_name;
```

A.2 Section 2.5 scripts

Listing A.4: MySQL script to query Earth1 data

```
mysql>select (unique_iotime - unique_meta) as localio,
    unique_meta as localmeta, (
    shared_time_by_cumul_io_only -
    shared_time_by_cumul_meta_only) as globalio, (
    iotime - unique_iotime) as globalmeta, (runtime -
    iotime) as notio, total_bytes,
    allshared_posix_write from Jobs_header, Jobs_perf,
    Jobs_files where Jobs_header.logfilename =
    Jobs_perf.logfilename and Jobs_header.logfilename =
    Jobs_files.logfilename and exe_name like '%Earth1%'
```

A.3 Section 2.6.1 scripts

MySQL scripts to query data for Figure 2.12 and Figure 2.13 is similar to those in Appendix A.1.

Listing A.5: R script to create Figure 2.12

```
> maxperf <- read.csv("./maxperf3.csv")
> gg_color_hue <- function(n) {
    hues = seq(15, 375, length=n+1)
    hcl(h=hues, l=65, c=100)[1:n]}
> cols = gg_color_hue(3)
> ggplot(data = maxperf) + geom_line(aes(x=
    job_percentage, y=log(max_perf,1024),color=platform
    ), size = 2) + xlab("Applications") + ylab("I/O
```

Listing A.5 (Cont.):

```
    Throughput") + scale_y_continuous(breaks = seq
    (-1,2,1),labels=c("1 KB/s", "1 MB/s", "1 GB/s", "1
    TB/s")) + theme_bw() + theme(axis.text=
    element_text(size=18,color="black"),axis.title=
    element_text(size=24), panel.grid.major =
    element_line(colour = "grey40"), plot.title =
    element_text(size=24), legend.text = element_text(
    size=16), legend.position="top") + geom_abline(
    intercept=log(168*1024,1024),slope=0,colour = cols
    [1],size=2) + geom_abline(intercept=log
    (88*1024,1024),slope=0,colour = cols[2],size=2) +
    geom_abline(intercept=log(240*1024,1024),slope=0,
    colour = cols[3],size=2) + ggtitle("Applications'
    Max Throughput") + scale_x_continuous(breaks=c
    (0,0.25,0.5, 0.75,1),labels=c("0","25%","50%",
    "75%","100%"))
```

Listing A.6: R script to create Figure 2.13

```
> moverall <- read.csv("./mira_overall.csv")
> ggplot(moverall, aes(log2(total_bytes), log2(
    agg_perf_MB))) + stat_binhex(aes(fill = cut(..
    count.., c(0, 10, 100, 500, 1000, 5000,10000,
    100000),labels = c('1 - 10', '11 - 100','101 -
    500', '501 - 1k', '1k1 - 5k','5k1 - 10k','10k1 -
    100k')))) + xlab("Number of bytes transferred") +
     ylab("I/O Throughput") + scale_y_continuous(breaks
     = seq(-20,20,10),labels=c("1 B/s", "1 KB/s", "1 MB
    /s", "1 GB/s","1TB/s")) + scale_x_continuous(
    breaks = seq(0,50,10),labels=c("1 B", "1 KB", "1 MB
    ", "1 GB","1 TB","1 PB")) +
geom_hline(yintercept=log2(1024 * 240),colour="
    darkgreen",size=1) + geom_hline(yintercept=log2
    (1024 * 2.40),colour="purple",size=1) + theme_bw()
    + theme(axis.text=element_text(size=14,color="black
    "),axis.title=element_text(size=24), panel.grid.
```

Listing A.6 (Cont.):

```
    major = element_line(colour = "grey40"), plot.title
    =element_text(size=24))  + annotate("text", x = 15,
     y = 19, label = "System peak - 240 GB/s")+
    annotate("text", x = 8, y = 12.5, label ="1% peak")
     + ggtitle("Mira: Jobs I/O Throughput")  +  labs(
    fill = 'Jobs Count') +  scale_fill_manual(values =
    c("cadetblue1","dodgerblue","blue","green","yellow
    ","darkorange","red"))
```

Listing A.7: MySQL script for Figure 2.16

```
select * from (
select t1.real_exe, "0-1 GB" as category, ifnull(
   numjobs,0) as numjob from (select distinct real_exe
    from Jobs_header) t1 left join (select * from
   jobdatasize where category = "0-1 GB") t2 on t1.
   real_exe = t2.real_exe UNION
select t1.real_exe, "1-10 GB" as category, ifnull(
   numjobs,0)  as numjob from (select distinct
   real_exe from Jobs_header) t1 left join (select *
   from jobdatasize where category = "1-10 GB") t2 on
   t1.real_exe = t2.real_exe UNION
select t1.real_exe, "10-100 GB" as category, ifnull(
   numjobs,0)  as numjob from (select distinct
   real_exe from Jobs_header) t1 left join (select *
   from jobdatasize where category = "10-100 GB") t2
   on t1.real_exe = t2.real_exe UNION
select t1.real_exe, "100 GB-1 TB" as category, ifnull(
   numjobs,0)  as numjob from (select distinct
   real_exe from Jobs_header) t1 left join (select *
   from jobdatasize where category = "100 GB-1 TB") t2
    on t1.real_exe = t2.real_exe UNION
select t1.real_exe, "> 1 TB" as category, ifnull(
   numjobs,0)  as numjob from (select distinct
   real_exe from Jobs_header) t1 left join (select *
   from jobdatasize where category = "> 1 TB") t2 on
```

Listing A.7 (Cont.):

```
    t1.real_exe = t2.real_exe ) q1,
(select real_exe, count(*) as total_numjob, avg(
    agg_perf_MB) as avg_perf, max(agg_perf_MB) as
    max_perf, sum(total_bytes) as sum_bytes from
    Jobs_header, Jobs_perf where Jobs_header.
    logfilename = Jobs_perf.logfilename group by
    real_exe) q2
where q1.real_exe = q2.real_exe;
```

A.4 Section 2.6.2 scripts

Listing A.8: MySQL script to query top 15 big-time applications

```
mysql> select exe_name, sum(runtime) as totalruntime,
    sum(iotime) as totaliotime, sum(total_bytes) as
    bytes, count(*) as numjobs, sum(iotime)/sum(runtime
    ) as io_percent  from Jobs_header, Jobs_perf where
    Jobs_header.logfilename = Jobs_perf.logfilename
    group by exe_name order by sum(iotime) desc limit
    15;
```

Listing A.9: MySQL script for Figure 2.19

```
mysql> select @jobid:=@jobid+1 as jobid, @jobid/
    total_job_count as job_percentage, io_percentage, "
    Mira" as platform from (select count(*)  as
    total_job_count from Jobs_header) t2, (select
    @jobid:=0) r, (select  iotime/runtime as
    io_percentage from Jobs_header, Jobs_perf  where
    Jobs_header.logfilename = Jobs_perf.logfilename and
     iotime < runtime order by jobio_percentage  desc )
     t1;
```

Listing A.10: R script for Figure 2.19

```
> sysutil <- read.csv("sysutil.csv")
> ggplot(data=sysutil) + geom_line(aes(x=
    job_percentage,y=io_percentage,colour=platform),
    size=3) + scale_x_continuous (breaks=c
    (0,0.1,0.25,0.5,1.0),labels=c
    ("0","10%","25%","50%","100%")) +
    scale_y_continuous (breaks=c(0,0.5,0.9,1.0),labels=
    c("0","50%","90%","100%")) + xlab("Jobs") + ylab("
    Percentage of total system I/O time") + theme_bw()
    + theme(axis.text=element_text(size=18,color="black
    "),axis.title=element_text(size=24),panel.grid.
    major = element_line(colour = "grey40"), legend.
    position="top", legend.text=element_text(size=18))
```

Listing A.11: MySQL script for Figure 2.20

```
select appname, agg_perf_MB from top15, Jobs_header,
    Jobs_perf where top15.real_exe = Jobs_header.
    real_exe and Jobs_header.logfilename = Jobs_perf.
    logfilename  and agg_perf_MB is not null ;
```

Listing A.12: R script for Figure 2.20

```
> mtop15 <- read.csv("mira.csv")
> mtop15$newrank <- sprintf("%02d",mtop15$rank)
> mtop15$app <- paste(itop15$newrank,mtop15$appname)
> boxplot(log2(agg_perf_MB)~app, data=mtop15,xlab="",
    ylab="",las=2,yaxt="n",main="Mira's Top 15 Big Time
     Applications")
> labels <- c("1 KB/s","32 KB/s","1 MB/s","32 MB/s","1
    GB/s","32 GB/s","1 TB/s")
> axis(2,at=seq(-10,20,5),labels = labels,las=2)
```

Listing A.13: MySQL script for Mira data of Figure 2.23

```
mysql> select "Small_data,_Small_procs" as category,
```

Listing A.13 (Cont.):

```
    rank, appname, count(*) as numjob, avg(agg_perf_MB)
     as avg_perf, "Mira" as platform from top15,
    Jobs_header, Jobs_perf  where top15.exe_name =
    Jobs_header.exe_name and Jobs_header.logfilename =
    Jobs_perf.logfilename and nprocs < 2048 and
    total_bytes < 10 * 1024*1024*1024 group by appname
UNION
select "Big data, Small procs" as category, rank,
    appname, count(*) as numjob, avg(agg_perf_MB) as
    avg_perf, "Mira" as platform from top15,
    Jobs_header, Jobs_perf  where top15.exe_name =
    Jobs_header.exe_name and Jobs_header.logfilename =
    Jobs_perf.logfilename and nprocs < 2048 and
    total_bytes > 10 * 1024*1024*1024 group by appname
union
select "Small data, Big procs" as category, rank,
    appname, count(*) as numjob, avg(agg_perf_MB) as
    avg_perf, "Mira" as platform from top15,
    Jobs_header, Jobs_perf  where top15.exe_name =
    Jobs_header.exe_name and Jobs_header.logfilename =
    Jobs_perf.logfilename and nprocs > 2048 and
    total_bytes < 10 * 1024*1024*1024 group by appname
UNION
select "Big data, Big procs" as category, rank,
    appname, count(*) as numjob, avg(agg_perf_MB) as
    avg_perf, "Mira" as platform from top15,
    Jobs_header, Jobs_perf  where top15.exe_name =
    Jobs_header.exe_name and Jobs_header.logfilename =
    Jobs_perf.logfilename and nprocs > 2048 and
    total_bytes > 10 * 1024*1024*1024 group by appname;
```

Listing A.14: R script for Figure 2.23

```
> m15distri <- read.csv("./mira_top15_distri2.csv")
> e15distri <- read.csv("./edison_top15_distri2.csv")
> top15distri <- rbind(m15distri,e15distri)
```

Listing A.14 (Cont.):

```
> top15distri$newrank <- substr(top15distri$appname
    ,0,2)
> top15distri$app <- paste(top15distri$platform,
    top15distri$newrank)
> ggplot(data=top15distri,aes(x=appname,y=log2(
    avg_perf),colour=factor(category),shape=factor(
    platform))) + geom_point(size=5) + theme_bw() +
    theme(legend.position="top", axis.text.x=
    element_text(angle=90,hjust=1), axis.text.y=
    element_text(size=14),axis.title=element_text(size
    =24)) + xlab("") + ylab("I/O Throughput") +
    scale_y_continuous (breaks=c(0,5,10,14),labels=c("1
     MB/s","32 MB/s", "1 GB/s","16 GB/s")) +
    scale_shape_manual(name="",values=c(15,16)) +
    scale_colour_manual(name="",values=c("red","green
    ","blue","purple"))
```

A.5 Section 2.6.3 scripts

The MySQL scripts to query data for graphs in Section 2.6.3 which include application's time breakdown are similar to those in Appendix A.2. Graphs are created using Excel.

A.6 Section 2.6.4 scripts

Listing A.15: MySQL script for Mira data of Figure 2.27

```
select "POSIX" as interface, floor(log2(nprocs)) as
    numproc, count(*) as job_count from Jobs_header,
    Jobs_files where Jobs_header.logfilename =
    Jobs_files.logfilename and (allshared_mpi_count +
    partshared_mpi_count + unique_mpi_count) = 0 group
    by floor(log2(nprocs));
```

Listing A.15 (Cont.):

```
select "POSIX" as interface, floor(log2(nprocs)) as
    numproc , 0 as job_count from Jobs_header where
    floor(log2(nprocs)) not in (select floor(log2(
    nprocs)) from Jobs_header, Jobs_files where
    Jobs_header.logfilename = Jobs_files.logfilename
    and (allshared_mpi_count + partshared_mpi_count +
    unique_mpi_count) = 0) ;

select "MPI" as interface, floor(log2(nprocs)) as
    numproc, count(*) as job_count from Jobs_header,
    Jobs_files where Jobs_header.logfilename =
    Jobs_files.logfilename and (allshared_mpi_count +
    partshared_mpi_count + unique_mpi_count) > 0 group
    by floor(log2(nprocs))
UNION
select "MPI" as interface, floor(log2(nprocs)) as
    numproc, 0 as job_count from Jobs_header where
    floor(log2(nprocs)) not in (select distinct floor(
    log2(nprocs)) from Jobs_header where logfilename in
     (select logfilename from Jobs_files where (
    allshared_mpi_count + partshared_mpi_count +
    unique_mpi_count) > 0 ));
```

Listing A.16: R script for Figure 2.27

```
> minter <- read.csv("mira/mira_interface.csv")
> ggplot(data=minter,aes(x=numproc, y=job_count, fill=
    interface)) + geom_bar(stat="identity") +
    scale_x_continuous(breaks = seq(0,20,2),labels=c
    ("1", "4", "16", "64","256","1K","4K","16K","64K
    ","256K", "1M")) + scale_y_continuous(breaks=seq
    (0,30000,10000)) + xlab("Number of processes") +
    ylab("Number of jobs") + theme_bw() + theme(axis.
    text=element_text(size=18,color="black"),axis.title
    =element_text(size=24),panel.grid.major =
```

Listing A.16 (Cont.):

```
    element_line(colour = "grey40"), legend.position="
    none", legend.text=element_text(size=18)) + ggtitle
    ("Mira")
```

Listing A.17: MySQL script for Mira data of Figure 2.28

```
select count(*) from Jobs_header, Jobs_perf,
    Jobs_files where Jobs_header.logfilename =
    Jobs_perf.logfilename and Jobs_header.logfilename =
     Jobs_files.logfilename and (unique_mpi_count +
    partshared_mpi_count + allshared_mpi_count ) = 0
    and agg_perf_MB is not null and agg_perf_MB >=
    5*1024;
select count(*) from Jobs_header, Jobs_perf,
    Jobs_files where Jobs_header.logfilename =
    Jobs_perf.logfilename and Jobs_header.logfilename =
     Jobs_files.logfilename and (unique_mpi_count +
    partshared_mpi_count + allshared_mpi_count ) = 0
    and agg_perf_MB is not null and agg_perf_MB >= 1024
     and agg_perf_MB < 5*1024;
select count(*) from Jobs_header, Jobs_perf,
    Jobs_files where Jobs_header.logfilename =
    Jobs_perf.logfilename and Jobs_header.logfilename =
     Jobs_files.logfilename and (unique_mpi_count +
    partshared_mpi_count + allshared_mpi_count ) = 0
    and agg_perf_MB is not null and agg_perf_MB < 1024;

select count(*) from Jobs_header, Jobs_perf,
    Jobs_files where Jobs_header.logfilename =
    Jobs_perf.logfilename and Jobs_header.logfilename =
     Jobs_files.logfilename and (unique_mpi_count +
    partshared_mpi_count + allshared_mpi_count ) > 0
    and agg_perf_MB is not null and agg_perf_MB >=
    5*1024;
select count(*) from Jobs_header, Jobs_perf,
    Jobs_files where Jobs_header.logfilename =
```

Listing A.17 (Cont.):

```
    Jobs_perf.logfilename and Jobs_header.logfilename =
     Jobs_files.logfilename and (unique_mpi_count +
    partshared_mpi_count + allshared_mpi_count ) > 0
    and agg_perf_MB is not null and agg_perf_MB >= 1024
     and agg_perf_MB < 5*1024;
select count(*) from Jobs_header, Jobs_perf,
    Jobs_files where Jobs_header.logfilename =
    Jobs_perf.logfilename and Jobs_header.logfilename =
     Jobs_files.logfilename and (unique_mpi_count +
    partshared_mpi_count + allshared_mpi_count ) > 0
    and agg_perf_MB is not null and agg_perf_MB <
    1*1024;
```

A.7 Section 2.6.5 scripts

Listing A.18: MySQL script for Mira data of Figure 2.29

```
mysql> select count(*), "0-1 GB" as category from
    Jobs_header, Jobs_perf where Jobs_header.
    logfilename = Jobs_perf.logfilename and (
    unique_meta + shared_time_by_cumul_meta_only ) > (
    unique_iotime + shared_time_by_cumul_io_only - (
    unique_meta + shared_time_by_cumul_meta_only ) )
    and total_bytes < 1*(1024*1024*1024) UNION
select count(*), "1-10 GB" as category from
    Jobs_header, Jobs_perf where Jobs_header.
    logfilename = Jobs_perf.logfilename and (
    unique_meta + shared_time_by_cumul_meta_only ) > (
    unique_iotime + shared_time_by_cumul_io_only - (
    unique_meta + shared_time_by_cumul_meta_only ) )
    and total_bytes >= 1*(1024*1024*1024) and
    total_bytes < 10*(1024*1024*1024) UNION
select count(*), "10-100 GB" as category from
    Jobs_header, Jobs_perf where Jobs_header.
```

Listing A.18 (Cont.):

```
    logfilename = Jobs_perf.logfilename and (
    unique_meta + shared_time_by_cumul_meta_only ) > (
    unique_iotime + shared_time_by_cumul_io_only - (
    unique_meta + shared_time_by_cumul_meta_only ) )
    and total_bytes >= 10*(1024*1024*1024) and
    total_bytes < 100*(1024*1024*1024) UNION
select count(*), "100 GB-1 TB" as category from
    Jobs_header, Jobs_perf where Jobs_header.
    logfilename = Jobs_perf.logfilename and (
    unique_meta + shared_time_by_cumul_meta_only ) > (
    unique_iotime + shared_time_by_cumul_io_only - (
    unique_meta + shared_time_by_cumul_meta_only ) )
    and total_bytes >= 100*(1024*1024*1024) and
    total_bytes < 1024*(1024*1024*1024)UNION
select count(*), "> 1 TB" as category from
    Jobs_header, Jobs_perf where Jobs_header.
    logfilename = Jobs_perf.logfilename and (
    unique_meta + shared_time_by_cumul_meta_only ) > (
    unique_iotime + shared_time_by_cumul_io_only - (
    unique_meta + shared_time_by_cumul_meta_only ) )
    and total_bytes >= 1024*(1024*1024*1024);

mysql> select count(*) as total from Jobs_header;
```

A.8 Section 2.6.6 scripts

The MySQL scripts to query data for graphs in Section 2.6.3 which include application's time breakdown are similar to those in Appendix A.2. Graphs are created using Excel.

Listing A.19: MySQL script for Figure 2.30

```
select * from Jobs_header, Jobs_perf, Jobs_files where
    Jobs_header.logfilename = Jobs_files.logfilename
    and Jobs_header.logfilename = Jobs_perf.logfilename
```

Listing A.19 (Cont.):

```
    and total_count > 1000000;
```

Listing A.20: MySQL script for Figure 2.34

```
select count(*) from Jobs_files where allshared_count
   > 0 and allshared_mpi_count = allshared_count union
select count(*) from Jobs_files where allshared_count
   > 0 and allshared_posix_count = allshared_count
   union
select count(*) from Jobs_files where allshared_count
   > 0 and allshared_mpi_count < allshared_count and
   allshared_posix_count < allshared_count and
   allshared_mpi_count + allshared_posix_count =
   allshared_count ;
```

A.9 Section 2.7 scripts

Listing A.21: R script for Figure 2.38

```
ggplot(all3) + geom_boxplot(aes(x=platform,y=log2(
   total_bytes),fill=platform)) + theme_bw() +
   scale_y_continuous(breaks = seq(0,50,10),labels=c
   ("1B", "1KB", "1MB", "1GB","1TB","1PB")) +
   facet_wrap(~ appname,nrow=1) + ylab("Number of
   bytes") + xlab("") +theme(legend.position="top",
   axis.text=element_text(color="black",size=18), axis
   .title=element_text(size=24),strip.background =
   element_blank(), strip.text.x = element_blank(),
   legend.text=element_text(size=18)) +
   scale_x_discrete(breaks=NULL)
```

Listing A.22: R script for Figure 2.39

```
> mgwl <- read.csv("~/Dropbox/final_defense/data/
```

Listing A.22 (Cont.):

```
    mira_gwl_heatmap.csv")
> ggplot(mgwl, aes(log2(total_bytes), agg_perf_MB
    /1024)) +  stat_binhex(aes(fill = cut(..count.., c
    (0, 10, 100, 500, 1000, 5000),labels = c('0 - 10',
    '10 - 100','100 - 500', '500 - 1k', '1k - 5k')))) +
     scale_fill_manual(name="Jobs count",values = c("
    blue","green","yellow","red",'darkorchid4')) + xlab
    ("Number of bytes") +scale_x_continuous(limits=c
    (37,42.5),breaks=seq(38,42,2),labels=c("256GB","1TB
    ", "4TB")) + scale_y_continuous(limits=c(0,8.5),
    breaks=seq(0,8,2),labels=c("0", "2 GB/s", "4 GB/s",
     "6 GB/s","8 GB/s")) + theme_bw() + ggtitle("Mira")
     + theme(axis.text=element_text(size=12,color="
    black"),axis.title=element_text(size=16), panel.
    grid.major = element_line(colour = "grey40"), plot.
    title=element_text(size=18)) + ylab("I/O throughput
    ")
```

Listing A.23: R script for Figure 2.41

```
> ggplot(crossplat1 ,aes(x=log2(total_bytes),y=log2(
    nprocs),color= log2(agg_perf_MB))) + geom_point(
    size=5) +  facet_wrap(~platform,nrow=1) +
    scale_x_continuous(breaks = seq(0,50,10),labels=c
    ("1B", "1KB", "1MB", "1GB","1TB","1PB"))  + theme(
    legend.position="right",axis.text=element_text(
    color="black",size=16),axis.title=element_text(size
    =24)) + xlab("Number of bytes transferred") + ylab
    ("Number of processes") + scale_y_continuous(breaks
    =seq(0,15,5),labels=c("1","32","1K","32K")) +
    scale_color_gradientn(colours=rev(rainbow(8,start
    =0,end=0.7)),name="I/O Thruput",breaks=c
    (-3,0,10,14),labels=c("128 KB/s","1 MB/s","1 GB/s
    ","16 GB/s"))
```

Listing A.24: R script for Figure 2.44

```
> ggplot(weather1,aes(x=log2(total_bytes),y=log2(
    nprocs),color=log2(agg_perf_MB))) + geom_point(size
    =5) +  facet_wrap(~platform,nrow=1) + theme_bw() +
    scale_x_continuous( breaks=seq(25,45,5),labels=c
    ("32 MB", "1 GB", "32 GB", "1 TB", "32 TB")) +
    scale_y_continuous(breaks=seq(4,12,2),labels=c
    ("16","64","256","1K","4K")) + theme(panel.margin =
     unit(2, "lines")) + scale_color_gradientn(colours=
    rev(rainbow(8,start=0,end=0.7)),name="I/O Thruput",
     breaks=c(5,10,12,15),labels=c("32 MB/s","1 GB/s
    ","4 GB/s","32 GB/s")) +  xlab("Number of bytes")
    + ylab("Number of processes") + theme(axis.text=
    element_text(size=12,color="black"),axis.title=
    element_text(size=16), panel.grid.major =
    element_line(colour = "grey40"), plot.title=
    element_text(size=18),legend.position="top",legend.
    key.width=unit(3,"line"))
```