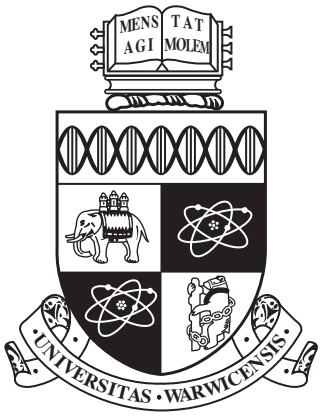


Towards Application-centric I/O Benchmarking for Parallel Scientific Applications

by

James Dickson

A thesis submitted to The University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

Doctor of Philosophy

Department of Computer Science

The University of Warwick

September 2018

Abstract

High performance computing (HPC) systems are undergoing an explosion in the variation and complexity of their hardware components and architectures. By the early 2020s, it is predicted that exascale systems will be in operation, and it is the pursuit of this capability that is currently shifting the parallel computing landscape.

Storage systems have developed alongside the other supercomputing components, but have struggled to keep pace with the rate of progress of computational hardware in particular. The resulting performance bottlenecks are problematic, as the data storage demands for the purposes of analysis and resilience are unlikely to be reduced by the current and future generations of HPC platforms.

The principle focus of the work presented in this thesis is to enable the benchmarking and analysis of I/O in a meaningful way, such that the performance expected from a scientific application can be accurately measured. Specifically, this thesis presents a case study of the profiling of multi-physics workloads for the purpose of extracting workload characteristics that are not limited by commercial sensitivity, as with the application itself. A flexible and portable I/O proxy application is developed and validated, before being deployed with the previously defined workloads to benchmark I/O performance on a number of current generation systems. This thesis ends with an analysis of the replicated workloads' sensitivity to the configuration of different elements of the I/O software stack. An evaluation is also performed on some alternative high level I/O implementation strategies that can be adopted to simplify the adoption of new burst buffer storage architectures.

Acknowledgements

I first joined the Department of Computer Science in October 2010 and during this time I have been fortunate to work with and enjoy the company of many great people. In the first instance, I would like to thank my supervisor, Professor Stephen Jarvis, for affording me the opportunity to undertake a Ph.D. Equally, I wish to express gratitude to Dr. Satheesh Maheswaran for fulfilling the role of my industry supervisor, and I offer my apologies for the rafts of paperwork you were subjected to on my behalf.

I am certain that without the advice, support and distractions offered by friends and colleagues, past and present, I would not have been able to complete this Ph.D. In particular, I wish to thank Dr. James Archbold, Dr. Robert Bird, Dr. Richard Bunt, Dr. Adam Chester, Dr. Peter Coetzee, Dr. James Davis, Dr. Tim Law, Dr. James Marchant, Dr. Faiz Sayiid, Dr. Phillip Taylor and Dr. Steven Wright. Thank you all for listening to ideas, proof reading drafts of work, and for the day to day entertainment that has kept me going.

Furthermore, I have found the Department of Computer Science a unique place to spend the past 8 years of my life, and I have enjoyed the professionalism and assistance of both academic and support staff during this time. Some of those I am thankful to include Dr. Abhir Bhlereo, Mike Cribdon, Sharon Howard, Professor Graham Martin, Lynn McLean, and Dr. Roger Packwood.

Outside of Warwick, I have been fortunate to collaborate with industrial contacts who have offered me assistance and opportunities without which I would have not have been successful. Many of the members of the Applied Computer Science team at AWE have not hesitated to spare their valuable time to assist me, for which I am very grateful. This are also true of a number of individuals I have been fortunate to work with during my time at Lawrence Livermore National Laboratory, in particular Dr. Kathryn Mohror, Dr. Mark

Miller and Dr. Elsa Gonsiorowski.

I owe an immeasurable debt of gratitude to my family for their love and support. Thank you to Mum, Dad, Jon and my wonderful Nana for your unwavering belief and encouragement. Finally but by no means least, I have to say how grateful I am to you Danielle, I hope you know I couldn't have managed any of this without you.

Declarations

This thesis is submitted to the University of Warwick in support of the author's application for the degree of Doctor of Philosophy. It has been composed by the author and has not been submitted in any previous application for any degree. The work presented (including data generated and data analysis) was carried out by the author except in the cases outlined below:

- I/O profiling data in Chapter 5 for an AWE multi-physics application run on the SpruceA supercomputer were collected by Dr. Duncan Harris (AWE)
- The original development work for the MACSio proxy application featured in Chapter 6 was carried out by Mark Miller at Lawrence Livermore National Laboratory (LLNL)

Parts of this thesis have been previously published by the author in the following:

[26] J. Dickson. Parallel I/O Libraries. In *1st Symposium of the Centre for Computational Plasma Physics, University of Warwick*, November 2015

[27] J. Dickson. Investigating Application I/O. In *JOWOG 34 Meeting, Atomic Weapons Establishment*, June 2016

[28] J. Dickson. Replicating I/O Behaviour in Production Applications. In *JOWOG 34 Meeting, Los Alamos National Laboratory*, June 2017

[29] J. Dickson. I/O Performance Analysis with Proxy Applications. In *JOWOG 34 Applied Computer Science Meeting, Sandia National Laboratory*, February 2018

[30] J. Dickson, A. Herdman, S. Maheswaran, S. a. Wright, J. a. Herdman, and S. a. Jarvis. MINIO : an I/O benchmark for investigating high level parallel libraries. In *27th ACM/IEEE International Conference for*

High Performance Computing, Networking, Storage and Analysis (SC'15), pages 5–6, September 2015. ISBN Dickson, James, Maheswaran, Satheesh, Wright, Steven A., Herdman, J. A. and Jarvis, Stephen A. (2015) MINIO : an I/O benchmark for investigating high level parallel libraries. In: 27th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SCâĂŹ15), Austin, Texas, USA, 15-20 Nov 2015 (In Press). URL `http://wrap.warwick.ac.uk/73143/20/WRAP_ExtendedAbstract%2BConferenceposter.pdf`

[32] J. Dickson, S. Wright, S. Maheswaran, A. Herdman, M. C. Miller, and S. Jarvis. Replicating HPC I/O workloads with proxy applications. In *Proceedings of PDSW-DISCS 2016: 1st Joint International Workshop on Parallel Data Storage and Data Intensive Scalable Computing Systems - Held in conjunction with SC16: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 13–18, September 2017. ISBN 9781509052165. doi: 10.1109/PDSW-DISCS.2016.007

[31] J. Dickson, S. Wright, S. Maheswaran, A. Herdman, D. Harris, M. C. Miller, and S. Jarvis. Enabling Portable I / O Analysis of Commercially Sensitive HPC Applications Through Workload Replication. In *Cray User Group 2017 Proceedings*, pages 7–12, May 2017

Sponsorship and Grants

The research presented in this thesis was made possible by the support of the following benefactors and sources:

- Atomic Weapons Establishment, United Kingdom:
  AWE Technical Outreach Programme (2014-2018, CDK0724)
- EPSRC ARCHER Resource Allocation Panel (RAP) Grant:
  "Preparing for Exascale with Mini-applications" (2015-2017, e402)

Abbreviations

ADIOS	Adaptable I/O System
ANL	Argonne National Laboratory
API	Application Programming Interface
ATA	Advanced Technology Attachment
AWE	Atomic Weapons Establishment
B/W	Bandwidth
CPU	Central Processing Unit
DFS	Distributed File System
FLOPS	Floating Point Operations Per Second
GB	Gigabyte - 1024^3 Bytes
GPFS	IBM General Parallel File System
HDD	Hard Disk Drive
HDF5	Hierarchical Data Format version 5
HPC	High Performance Computing
I/O	Input/Output
LANL	Los Alamos National Laboratory
LLNL	Lawrence Livermore National Laboratory
MB	Megabyte - 1024^2 Bytes
MDS	Metadata Server
MDT	Metadata Target
MGS	Management Server
MPI	Message Passing Interface
NFS	Networked File System
NVMe	Non-Volatile Memory
ORNL	Oak Ridge National Laboratory
OSS	Object Storage Server

OST	Object Storage Target
PFS	Parallel File System
POSIX	Portable Operating System Interface
RAID	Redundant Array of Independent Disks
RPM	Revolutions Per Minute
SAN	Storage Area Network
SSD	Solid State Drive
TB	Terrabyte - 1024^4 Bytes
UFS	Unix File System
VFS	Virtual File System

Contents

Abstract ii

Acknowledgements iii

Declarations v

Sponsorship and Grants vii

Abbreviations viii

List of Figures xvii

List of Tables xxi

1 Introduction 1

1.1 Motivation 4

1.2 Thesis Contributions 6

1.3 Thesis Overview 8

2 Performance Analysis and Engineering 11

2.1 Parallel Computing 12

2.2 Parallel I/O and Data Storage 13

2.2.1 Issues in Parallel I/O and Data Storage 16

2.2.2 Parallel File Systems 19

2.2.3 I/O-aware Scheduling 19

2.2.4 Parallel I/O Software Stack 21

2.3 Performance Analysis and Engineering 27

2.3.1 Benchmarking 27

2.3.2 Profiling 29

2.3.3 Simulation 30

2.4 Summary 32

3 Parallel Hardware and Software Overview 33

3.1 Storage Hierarchy 34

3.1.1 Hard Disk Drive 34

3.1.2 Solid State Drive 35

3.2 File Systems 37

3.2.1 The Extended File Systems 37

3.2.2 ZFS 40

3.2.3 XFS 41

3.3 Clustered File Systems 43

3.3.1 The Lustre Parallel File System 45

3.3.2 The IBM Spectrum Scale File System 49

3.4 Computing Platforms 50

3.5 I/O Benchmarking Applications 54

3.6 Summary 55

4 Experimental Setup 57

4.1 Common Methodology: I/O Measurement 57

4.2 Chapter 5: Profiling Multi-physics I/O Workloads 58

4.2.1 Bookleaf Mini-Application 58

4.2.2 AWE01 Multi-Physics Application 58

4.3 Chapter 6: Application Workload Replication 59

4.3.1 Proxy Application Validation: Bookleaf 59

4.3.2 Proxy Application Validation: FLASH-IO 60

4.4 Chapter 7: I/O Performance Benchmarking and Optimisation . . 61

4.4.1 Tuning the Parallel I/O Software Stack: Middleware . . . 61

4.4.2 Parallel File System Performance 63

4.4.3 I/O Library and File Strategy Comparisons 64

4.5 Summary 65

5 Profiling Multi-physics I/O Workloads **67**

5.1 Application Pattern Identification 68

5.1.1 Runtime Profiling 70

5.1.2 Bookleaf Mini-Application 74

5.1.3 AWE01 Multi-Physics Application 76

5.1.4 Multi-Physics Checkpoint Analysis 80

5.2 Summary 81

6 Application Workload Replication **83**

6.1 The MACSio Proxy Application 84

6.1.1 Application Overview 84

6.1.2 Modifications 87

6.2 Proxy Application Validation 94

6.2.1 Bookleaf 96

6.2.2 FLASH-IO 103

6.3 Summary 107

7 I/O Performance Benchmarking and Optimisation **110**

7.1 Approach 111

7.2 Tuning the Parallel I/O Software Stack 112

7.2.1 Middleware 112

7.2.2 Parallel File System 123

7.3 I/O Library and File Strategy Comparisons 128

7.3.1 TyphonIO Efficiency 129

7.3.2 N-M Parallel File Modes 133

7.4 Discussion 138

7.5 Summary 139

8 Discussion and Conclusions **141**

8.1 Limitations 143

8.2 Future Work 145

8.3 Final Remarks 146

Bibliography **147**

Appendices **163**

A Profiling Multi-physics I/O Workloads **163**

B Application Workload Replication **164**

C I/O Performance Benchmarking and Optimisation **166**

List of Figures

2.1 Parallel I/O Strategies 17

3.1 Basic structure of a hard disk drive assembly 35
3.2 Basic structure of a solid state drive assembly 36
3.3 Internal structure of an XFS allocation group 42
3.4 An example Lustre configuration with one MGS, four OSSs, four OSTs, and one MDT with two MDSs in failover. 46
3.5 An example GPFS configuration with four OSSs and four OSTs connected over a switching fabric 49

5.1 An example of the general pattern of simulation phases found in scientific applications. 69
5.2 Execution pattern for Bookleaf simulating the noh_large problem at 1, 2, 4, 8, 16, 32, and 64 Nodes. These patterns of execution were collected on Archer with a default stripe count of 4. 75
5.3 Execution pattern for AWE01 Simulation A 77
5.4 Execution pattern for AWE01 Simulation B 78
5.5 Dataset Growth of Input B 79
5.6 Execution pattern for AWE01 Simulation D 80

6.1 MACSio application components 84

6.2 I/O timings representing the best case (10 repititions) for Bookleaf and the MACSio replication running on Archer (Lustre stripe count 48). For these results the best case is shown with the variation across repetitions being less than 9.4% for each run scale. Timings shown: (a) Start to finish time for a checkpoint (b) Cumulative time spent by all processors performing I/O (c) Time for the slowest write operation issued by a processor during each checkpoint. 98

6.3 I/O timings representing the best case (10 repititions) for Bookleaf and the MACSio replication running on Tinis (GPFS). For these results the best case is shown with the variation across repetitions being less than 11% for each run scale. Timings shown: (a) Start to finish time for a checkpoint (b) Cumulative time spent by all processors performing I/O (c) Time for the slowest write operation issued by a processor during each checkpoint. . 100

6.4 FLASH-IO 3D mesh decomposition to a 3D processor grid and the corresponding file layout. 105

6.5 Best case I/O timings for FLASH-IO and the MACSio replication collected on Archer (stripe count 48) showing: (a) Start to finish time for a checkpoint (b) Cumulative time spent by all processors performing I/O (c) Time for the slowest write operation issued by a processor during each checkpoint. 106

7.1 Checkpoint bandwidth for Bookleaf and Flash workloads run independently and collectively on Archer, Quartz and Ray. 114

7.2 Perceived checkpoint bandwidth for the AWE01 workloads using independent and collective calls on Archer, Quartz and Ray. The Lustre stripe count used on Archer and Quartz is 4 for (a) and 20 for (b) and (c). These results represent the best observed performance across 10 repetitions. 116

7.3 Perceived checkpoint bandwidth for the Bookleaf workload on Archer with different (a) aggregator node count (b) collective buffer size. 120
7.4 Perceived checkpoint bandwidth for the Flash workload on Archer with different (a) aggregator node count (b) collective buffer size. 121
7.5 Perceived file bandwidth achieved for AWE01 workloads with simple file striping strategies on the Spruce A, Titan, Archer, Cab, Taurus, and Tinis platforms. 124
7.6 Perceived file bandwidth achieved for AWE01 workload B over the duration of a run using five compute nodes. Results are shown for Spruce A, Titan, Archer, Cab, Taurus, and Tinis platforms with standard and increased striping (stripe count is shown in brackets). 127
7.7 Layout of different data writing modes tested in MACSio. 129
7.8 Checkpoint performance for AWE01 on Archer through standard TyphonIO, a TyphonIO-like layout using raw HDF5, and an alternative user-defined hyperslab arrangement also written using raw HDF5. 130
7.9 Checkpoint performance for AWE01 on Quartz through standard TyphonIO, a TyphonIO-like layout using raw HDF5, and an alternative user-defined hyperslab arrangement also written using raw HDF5. 131
7.10 File bandwidth achieved for the AWE01 Problem D workload on Quartz when using Sequential and Parallel N-M access patterns. 134
7.11 File bandwidth achieved for the AWE01 Problem D workload on Archer when using Sequential and Parallel N-M access patterns. 134
7.12 File bandwidth achieved for the AWE01 Problem D workload on Cab when using Sequential and Parallel N-M access patterns. . . 135

7.13 File bandwidth achieved for the AWE01 Problem D workload on Ray's GPFS file system and node-local burst buffers with N-M access patterns. 136

List of Tables

3.1 Hardware specification of the Titan and ARCHER supercomputers. 51
3.2 Hardware specification of the Spruce A and Cab supercomputers. 52
3.3 Hardware specification of the Quartz and Ray supercomputers. . 53
3.4 Hardware specification of the Taurus and Tinis supercomputers. 54

4.1 Bookleaf profile gathering setup 58
4.2 AWE01 profile gathering setup 59
4.3 MACSio-Bookleaf Validation on Archer 60
4.4 MACSio-Bookleaf Validation on Tinis 60
4.5 MACSio-FLASH Validation on Archer 60
4.6 Collective Operation Scaling on Archer 62
4.7 Collective Operation Scaling on Quartz 62
4.8 Collective Operation Scaling on Ray 62
4.9 cb_nodes and cb_buffer_size parameter settings tested 63
4.10 Collective Buffering Parameter Performance on Archer 63
4.11 Parallel file system striping performance 64
4.12 I/O library and file strategy comparison experiments 65
4.13 Parallel file mode comparison experiments 66

5.1 Summary of AWE Multi-Physics application profile data showing the calculated number of distinct I/O phases in a simulation where an open-write-close cycle operates on a visualisation file and the number of states actually observed in the file. 73
5.2 Checkpoint statistics for Bookleaf checkpoints at scales between 1 and 64 nodes collected on Archer with default stripe count of 4. 74
5.3 Checkpoint statistics for each problem class run by the AWE01 multi-physics application. 81

6.1 Configurable parameters in generated MACSio datasets 85
6.2 Input parameter values for MACSio validation runs of Bookleaf . 97
6.3 Input Parameter values for scaling Bookleaf validation runs. . . . 97
6.4 I/O timings for Bookleaf and MACSio replication checkpoints run on Archer with 10 repetitions. 99
6.5 Checkpoint file size comparison between Bookleaf and a MACSio replication run on Archer. 101
6.6 Checkpoint file breakdown for the Bookleaf noh large problem and MACSio replication run on a single node. 102
6.7 Checkpoint file breakdown for the Bookleaf noh large problem and MACSio replication run on 1 and 64 Nodes. 103
6.8 Input parameter values for MACSio validation runs of FLASH-IO 104
6.9 Checkpoint file size comparison between FLASH-IO and a MAC-Sio replication. 107
6.10 Checkpoint file breakdown for a FLASH-IO run on a single node. 107

7.1 Summary of the experimental target applications used as workloads inside MACSio. 112
7.2 Write sizes issued to the parallel file system during a Bookleaf checkpoint on Archer performed independently and collectively . 117
7.3 Write sizes issued to the parallel file system during a Bookleaf checkpoint on Quartz performed independently and collectively . 117
7.4 File bandwidth changes for checkpointing over the course of the AWE01 Problem B workload. 126
7.5 Performance improvement of Ray burst buffers over GPFS file system under parallel N-M file modes. 137
7.6 Bandwidth achieved per number of files on Ray when writing to burst buffers. 137

A.1 Darshan MPIIO counters 163

B.1 I/O timings for Bookleaf and MACSio replicated checkpoints on Archer. 164
B.2 Cumulative I/O timings for Bookleaf and MACSio replicated checkpoints on Archer. 164
B.3 Slowest I/O operation time for Bookleaf and MACSio replicated checkpoints on Archer. 164
B.4 I/O timings for Bookleaf and MACSio replicated checkpoints on Tinis. 165
B.5 Cumulative I/O timings for Bookleaf and MACSio replicated checkpoints on Tinis. 165
B.6 Slowest I/O operation time for Bookleaf and MACSio replicated checkpoints on Tinis. 165

C.1 Perceived checkpoint bandwidth for the Bookleaf workload on Archer. 166
C.2 Perceived checkpoint bandwidth for the Bookleaf workload on Quartz. 166
C.3 Perceived checkpoint bandwidth for the FLASH-IO workload on Archer. 166
C.4 Perceived checkpoint bandwidth for the FLASH-IO workload on Quartz. 167
C.5 Perceived checkpoint bandwidth for the Bookleaf workload on Ray. 167
C.6 Perceived checkpoint bandwidth for the FLASH-IO workload on Ray. 167
C.7 Write bandwidth for the Bookleaf workload running on Archer with different collective buffering node counts. 167
C.8 Write bandwidth for the FLASH-IO workload running on Archer with different collective buffering node counts. 168

C.9 Perceived checkpoint bandwidth for the AWE01 Problem A workload running on the Spruce A, Titan, Archer, Cab, Taurus and Tinis systems. 168
C.10 Perceived checkpoint bandwidth for the AWE01 Problem A workload running on the Spruce A, Titan, Archer, Cab, Taurus and Tinis systems. 168
C.11 Perceived checkpoint bandwidth for the AWE01 Problem B workload running on the Spruce A, Titan, Archer, Cab, Taurus and Tinis systems. 169
C.12 Checkpoint performance for AWE01 on Archer through standard TyphonIO, a TyphonIO-like logically contiguous layout using raw HDF5, and an block contiguous hyperslab arrangement in HDF5. 169
C.13 Checkpoint performance for AWE01 on Quartz through standard TyphonIO, a TyphonIO-like logically contiguous layout using raw HDF5, and an block contiguous hyperslab arrangement in HDF5. 170
C.14 File bandwidth achieved for the AWE01 Problem D workload on Archer when using Sequential and Parallel N-M access patterns. 170
C.15 File bandwidth achieved for the AWE01 Problem D workload on Quartz when using Sequential and Parallel N-M access patterns. 170
C.16 File bandwidth achieved for the AWE01 Problem D workload on Cab when using Sequential and Parallel N-M access patterns. . . 170
C.17 File bandwidth achieved for the AWE01 Problem D workload written through parallel N-M to GPFS and burst buffers on Ray. 171

CHAPTER 1

Introduction

The term 'Super computing' traces its origins back to as early as 1931 and the electromechanical tabulating machines built by IBM to process census data stored on punch cards. Thirty-three years later, the CDC 6600 was built by the Control Data Corporation with the capacity to perform 3 million floating point operations per second (FLOPS), what is now considered to be the first *supercomputer* [86]. In the modern day, the Summit machine at Oak Ridge National Laboratory (ORNL) can achieve 122.3 petaflops (122.3 quadrillion floating point operations per second) and in June 2018 was listed as the most powerful supercomputer in the world.

Over this period of several decades, the application of computing resources to scientific problems has dramatically changed the way that these problems can be approached. Computational simulation of real world phenomena can offer answers to problems that would otherwise be too time consuming, expensive, or even unsafe to attempt to solve with physical experimentation. The growth in power of supercomputing systems inevitably advances that which can be achieved through computational simulation, with larger and more complex systems allowing calculations to be performed much faster and at higher resolutions than were previously possible. Larger and higher resolution simulations running on highly parallel machines process and produce data at exceptional rates. This data represents the true value of a simulation, and it is the analysis and visualisation of this data that ultimately leads to knowledge discovery. Consequently, the failure of a simulation and loss of valuable data and expensive processing time is an important issue to be addressed by computational scientists, and those in the field of High Performance Computing (HPC). Studies

conducted at ORNL investigating failures on large-scale systems have demonstrated that the mean time between failures for a range of platforms studied over two generations ranged from 9 hours to 37 hours [43]. Here, failure is defined as a hardware or system-software related error that causes an application running on some part of the system to crash rather than a complete system failure. Protecting simulation data against loss from these types of failures requires transferring it to persistent storage, which can both stall the progress of a simulation and interfere with resources needed by other system users.

Despite the importance of storage systems in supercomputing platforms, they have not developed at the same rate as other system components. Furthermore, the obscurity of the data storage process has lead to it being underprioritised by application developers. Failures to address the storage needs of some applications has lead to this becoming a bottleneck; an issue that will be exacerbated by advanced system architectures increasing the density of compute power in a node and a corresponding increase in the data flowing in and out.

Consider a hypothetical scientific application that is being developed to simulate some real world phenomenon. The ultimate goal of the application would be to produce an answer to some proposed scientific hypothesis. For the users of this application, their aims are best served by simulating the largest possible problem space or producing the most high resolution representation of their phenomenon allowed by the size of their platform, servicing both of these aims will contribute to the production of greater physical quantities of data. Added to this, the more points in the simulation that they can capture a snapshot of its state, the greater the opportunity to understand what is happening inside the simulation. All of these motivating factors contribute to domain scientists wanting to generate greater quantities of I/O from their applications. The underlying issue for I/O in HPC is derived from the fact that these desires to generate greater quantities of I/O are rarely met with an equivalent investment of time or effort into how that I/O is carried out. Furthermore, there is no guarantee that the workloads generated by an application will be able to pro-

duce optimal I/O patterns, or the user workflow explicitly prevents optimal I/O behaviours from being observed.

A straightforward example to highlight a problem experienced by HPC applications is the variability in two of the most straightforward approaches, shared or multiple file based I/O. For a number of applications, a simple approach of writing a single file per processor is sufficient and performant enough to work well. In other cases where follow-on processing or data portability is required, a large number of files is unmanageable and the use of a single shared data file is required to be compatible with the user's workflow. These two approaches are both common, but behave optimally under different and conflicting storage configurations. It is the responsibility of the user to decide how they wish to implement their I/O workloads, but the responsibility of those operating the HPC system to deliver the best possible service to all users. Clearly, balancing requirements means that storage systems much be designed to work best in the average case but the variability this introduces prevent users from gaining a strong understanding of why their workloads are not performing optimally.

An example to illustrate this problem can be taken from the I/O workloads commonly run at Los Alamos National Laboratory (LANL). The applications that use the most resources at LANL utilise the single shared file strategy for committing data from multiple distributed processors to a single output file. The implementation of this strategy generates small unaligned I/O access patterns for the storage system, however the available storage systems are not able to perform this type of access well. The motivation for using such an approach was user driven, due to the ease of data reuse for different sized jobs making it well suited to users' needs. The conflict between the workloads that users are wanting to run and what can work well on the available platforms is one of the biggest issues for I/O and data storage.

The size of the parameter space that can be explored with respect to a particular I/O workload has also prevented applications from running with the best possible configurations. The modern I/O stack gives freedom to implement

patterns of read and write operations at a low level, but equally to employ high level libraries with simplified interfaces abstracting the mechanics away from the developer. At each level in the software stack there are elements that can be configured or parameters to be set that are critical to generating a well performing I/O workload. As an example, the widespread use of HDF5 library routines in applications allows for parallel operations to be performed with much less technical implementation overhead than lower level approaches. However, the way in which the library is instructed to marshal its processes, data, and file accesses is still very much in the hands of the user. Exploring the state space to establish how best to configure the implementation is time consuming for a large application with many thousands of lines of code and tightly coupled I/O routines. Flexibility in exploring the parameter space in terms of alternative software libraries, I/O schemes and tuning parameters is clearly required to uncover where poor performance is being generated.

1.1 Motivation

With the data storage requirements of applications growing, the ability to analyse the I/O behaviour of an application and make informed decisions about how it is likely to perform on new system architectures is key. Furthermore, the ability to test out optimisations and improvements to current I/O software and practices with the needs of a particular application in mind has the potential to improve performance on current systems. Decoupling the I/O workload from simulations offers opportunity for greater flexibility in both the performance optimisation and benchmarking activities.

To illustrate this point, the HPC and I/O landscape inside a large organisation charged with delivering critical scientific understanding is considered. Lawrence Livermore National Laboratory (LLNL) is one of three vitally important Department of Energy laboratories that exist in the United States. A key mission of LLNL is to deliver scientific understanding and provide experi-

mentally based support to a number of cutting edge problems. HPC drives the production of useful science at LLNL, with hundreds of domain scientists making use of parallel simulations to produce answers to their problems. Modern complex simulation codes are at the stage where they are the size of an entire software eco-system and for these codes to be maintained requires an exceptional level of both domain knowledge and performance engineering skill. The ability to separate concerns and allow I/O specialists to address problems in the libraries and mechanisms used by scientific simulations, without the complexities and overheads of working with the simulation itself, has the potential to deliver more efficient I/O in scientific codes and ultimately accelerate discovery.

This thesis demonstrates work that has been carried out in the development and validation of an application designed to generate I/O activity on a system from high-level descriptions of scientific datasets and I/O patterns. It is shown that it is possible to recreate common I/O behaviours from the perspective of scientific-like datasets. In addition, it is demonstrated how the high-level profiles of an application's I/O workload can be constructed based on the limited data available through lightweight I/O characterisation [80] and accurately recreated by the proxy application.

Additionally, this thesis contains a demonstration of the use of proxy I/O workloads to benchmark a range of current day systems, an activity that is vital for the purposes of system design and procurements. An investigation into how both the I/O software stack installed on a system, and the configuration of different software components can improve or degrade performance is presented. Specifically, the sensitivity of the parallel file system, middleware and high level I/O library to improper use is highlighted. Finally, a demonstration of the potential performance of an alternative parallel file strategy suitable for deployment on new burst buffer architectures is given.

1.2 Thesis Contributions

The research presented in this thesis makes the following contributions:

- Standard profiling techniques are used to enable novel application understanding of a commercially sensitive production multi-physics application and smaller scale hydrodynamics application. These applications are representative of a real world scientific code used by a critical industrial organisation and a more lightweight representation designed to represent a scientific simulation in a more portable form. An open source profiling and characterisation library is used to collect data on the two applications and their I/O patterns summarised from analysis of where I/O hotspots occur and how data storage is used to preserve scientific data.. Due to the size and complexity of the production application, more comprehensive I/O tracing is rendered infeasible and hence straightforward observations about the target applications are used to supplement the available profiling data. Profiles of the Bookleaf hydrodynamics application demonstrate how the I/O demands change in relation to the simulation as a whole and the characterisation of the AWE01 application defines three distinct categories of I/O pattern that it generates. Working towards the goal of enabling application focused I/O performance, the understanding of real applications workloads is a fundamental building block in ensuring any performance insights and optimisations that can be gained will be applicable to HPC applications in practice.

- Development contributions and validation of an I/O proxy application (MACSio) is described in detail. MACSio has been designed to recreate the characteristics of real scientific datasets and generate I/O activity based on a number of potential configurations. The I/O activity performed by MACSio is generated at a high level by libraries currently used in real world HPC applications, making it indistinguishable from an I/O operation that would be performed by a genuine scientific simulation. Fea-

tures such as additional library plugins and file management routines have been introduced into the MACSio code base to enable the faithful replication of profiled I/O workloads. In particular, by looking at I/O function performance and dataset accuracy it is shown that MACSio successfully replicates the Bookleaf and FLASH applications through the use of parameters informed by the profiling work covered previously. As mentioned previously, the decoupling of I/O workloads from large fully formed scientific applications enables a separation of concerns for domain scientists and HPC specialists. Validation of MACSio as a portable tool that I/O engineers can work with in a streamlined way makes it possible for application workloads to be investigated and optimised independently of the large amount of application overhead that would normally form a barrier to entry.

- Using MACSio, the performance of Bookleaf, FLASH and AWE01 are benchmarked against several platforms providing file systems of contrasting architectures and scales. Analysis of their performance is carried out with considerations of each layer in the parallel I/O software stack. Specifically, the impact of configuration changes at the parallel file system, middleware and high level library layers are explored, demonstrating that traditional advice on how to tune these components can cause performance degradation for the targeted workloads. It is identified that the approach to performance tuning is heavily dependent on the target workloads and software configuration, demonstrated by performance losses when naively spreading I/O work over Lustre targets. A comparison of different data layouts used by high level libraries is performed, highlighting potential pitfalls when using HDF5 hyperslabs. This work culminates in an evaluation of an alternative file mode to the single shared file approach used through this thesis. To simplify I/O on future burst buffer systems, an N-M I/O mode is demonstrated through MACSio and regularly achieves a speedup of more than 2× over both the single shared file and sequential

N-M strategies used by the TyphonIO and Silo libraries. New libraries and techniques widen the parameter space that exists for configuration of the I/O stack, introducing new options that much be intelligently set for an application to make best use of the storage systems available. This performance study for a group of scientific applications provides guidance to those looking to use said applications and the I/O strategies that they employ. Moreover, the demonstration of a new and promising file access pattern to work alongside an established parallel I/O library opens up the possibility of incorporating burst buffer technology for organisations like AWE, who lack the flexibility in their large scale production applications to adopt such techniques easily.

1.3 Thesis Overview

The remainder of this thesis is structured as follows:

Chapter 2 provides an overview of some of the key underlying concepts in high performance computing that underpin the work throughout this thesis. Furthermore, an overview is given of current work in the field with a particular focus on I/O and file systems. Specifically, current issues in parallel I/O are covered, along with elements of parallel file systems, the I/O software stack, profiling, and performance benchmarking in HPC.

Chapter 3 details the experimental processes followed to gather the results presented throughout the rest of this thesis. The chapter exists as a reference point for the use of different systems and configurations in experiments shown at later stages in the work.

Chapter 4 contains a brief description of hardware and software components used in this thesis. The chapter begins by introducing the storage hierarchy, as

typically found in HPC systems, from the operations of spinning and solid state disks up to the distributed file systems that are built up of described components. The conclusion of this chapter is an overview of the computing platforms and applications that are used throughout this thesis.

Chapter 5 describes the process undertaken to profile and reconstruct I/O workload patterns for the Bookleaf and AWE01 applications. Darshan lightweight I/O characterisation is described and and example of the data that can be extracted from the resulting logs is shown. An example of how Darshan counters are used to calculate features of a workload is shown for AWE01, which is followed by description and illustration of the application patterns that are important for the later work in this thesis.

Chapter 6 introduces the MACSio I/O proxy application, and describes in detail some of the components that have been developed in MACSio to enable effective workload replication and I/O exploration to be performed. A demonstration of MACSio replication of the behaviour of Bookleaf and FLASH is presented, which is accompanied by validation studies showing that the behaviour and performance of the replication effectively matches the original application. In particular, these applications demonstrate MACSio operating with the HDF5 and TyphonIO plugins that are important for the performance study that follows.

Chapter 7 combines the work of the previous two chapters and presents a series of benchmarking and performance analysis activities carried out on a collection of different scale HPC systems. An investigation is carried out on the effect of simple parameter tuning on elements in the parallel I/O stack. The effect of Lustre striping configurations, middleware optimisations and high level library implementation on the Bookleaf, FLASH and AWE01 workloads replicated through MACSio is shown, highlighting difficulties that exist in controlling

I/O behaviours at low level with poor system configuration. A comparison of the TyphonIO library is made with the HDF5 library that it implements under the covers, demonstrating a large improvement in performance that can be achieved by careful use of the HDF5 hyperslabs mechanism. Finally, this chapter demonstrates the use of a new N-M file scheme that has been enabled through MACSio. By controlling some of the management of files accessing storage targets in parallel, a notable speedup can be achieved over both standard single shared file and Silo-like grouped sequential parallel access.

Chapter 8 concludes the thesis, and discusses the implications of the work presented. The limitations of the research presented are outlined, and the potential for future work is suggested.

CHAPTER 2

Performance Analysis and Engineering

Computational techniques are key to mathematics, science and engineering in both an academic and industrial setting. Consequently, developing advanced techniques to increase computational power and efficiency has been an ever present goal of scientists seeking to reduce the time to solution or tackle computational problems of increasing scale and complexity.

Advances in computer design and algorithm optimisation are the two fundamental components that drive improvements in solving computational problems. In what could be considered a watershed moment, the introduction of parallel computers and the associated parallelisation of computational techniques has formed the basis of one of the most important advances experienced in the field; they have been seen to reduce the time to solve some problems by orders of magnitude, the time to simulate certain natural phenomena taking only seconds and minutes rather than weeks and months.

Exploiting hugely parallel machines is, however, not a straightforward task. Modern extreme scale systems posses tremendous amounts of raw computational power, but the mapping of applications and processes to these systems must be carried out in such a way as to tightly couple with the available subsystems to avoid performance bottlenecks. It is here that the field of performance analysis and engineering has contributed a substantial amount of work in understanding application behaviour and continuously optimising out inefficiencies where they are found to exist.

This chapter contains a summary of: *(i)* some of the core concepts of parallel computation and terminology used in high performance and scientific computing; *(ii)* an overview of the fundamental elements of parallel input/output and

data storage in parallel computation; *(iii)* an introduction to performance analysis and engineering principles that are used to investigate, understand and predict computational performance in high performance systems.

2.1 Parallel Computing

When executing a programmed sequence of instructions, computers rely on a number of different types of operation involving the movement, transformation, and storage of data. In particular, the processing of data through arithmetic operations is the basis of the majority of scientific applications. The ability to perform more than one of these operations at once is the defining characteristic of modern parallel computing. When a set of operations has been designed such that they can be carried out concurrently, the addition of more hardware components then increases the rate at which computation is completed, and therefore performance. However, the process of scaling out the size of a machine to generate greater parallel compute power is far from straightforward. The addition of more components can increase complexity disproportionately, and sophisticated algorithms and software is required to manage the optimal use of these components.

This phenomenon is illustrated by *Amdahl's law* [6], proposed by Gene Amdahl in 1967, which states that for a parallel program, the speedup S_n for n processors is given by:

$$S_n \leq \frac{1}{r_s + \frac{r_p}{n}} \tag{2.1}$$

where r_s and r_p represent the serial and parallel ratios of a program respectively and $r_s + r_p = 1$. This places an upper bound on the performance improvement when scaling a problem of a fixed size to an increased number of processing units, and is considered to be linked to the idea of *strong scaling*. It is clear from this formalism that any components of a program that cannot be effectively parallelised limit the benefits of increasingly large parallel machines.

A re-evaluation of Amdahl's law was proposed in 1988, known as *Gustafa-*

son's law [44]. This formalism describes the theoretical speedup gained by increasing the availability of parallel processing units. The important characteristic of this law is in the consideration of a fixed time window, and demonstrating how the parallel portion of a program allows for larger problem sizes to be solved in this time window. Gustafason's law can be expressed as:

$$S_n = n - r_s(n - 1) \tag{2.2}$$

where similarly S_n is the the scaled speedup for n processors and r_s is the fixed serial code portion that does not benefit from parallelism. As more processors are added, larger scale problem sizes can be solved within the fixed time window despite r_s being present. In the same way that Amdahl's law has been used to represent strong scaling behaviour, Gustafason's law can be considered to represent the practice of *weak scaling*. In this practice, adding parallel hardware to the solution of a problem sees the size of the problem grow proportionally to be solved in a similar time frame.

2.2 Parallel I/O and Data Storage

The increase in computational capacity of new machines allows higher resolution problems to be solved and a greater number of concurrent simulations to be performed. Both of these advances will inherently lead to greater volumes of data being generated by simulations, for both analysis and visualisation purposes. The difficulty that is presented by this situation is the increase in failure rate that accompanies increasingly large and complex systems [15, 43, 53, 79]. Failures and resilience in HPC systems has been well studied to understand the ways in which a system can experience failure and attempt to predict how frequently action should be taken to mitigate the risk of data or resource loss. A large number of works to understand system failures focuses on specific hardware components with memory failures occupying a large percentage of these [48, 61, 76, 82]. This is hardly surprising given many studies such as the one carried out by El-

Sayed and Schroeder [36], Snir et al. [79] showing that of the errors detected for compute nodes a high proportion of these were attributable to CPU or memory failures. These works are however not in agreement as to what the single most common cause of node failure is, with Snir et al. [79] suggesting 80% of compute node failures are attributable to memory where El-Sayed and Schroeder [36] observes only 20% of hardware failures to be memory related with 40% cause by CPU errors. More generally, hardware failures are observed to be the most common category of failure, this supported by a survey of a number of large-scale HPC systems at Los Alamos National Laboratory (LANL) [37]. This work highlights that in all of the systems studied hardware failures alone count for more than 54.15% of failures with this number as high as 77.68% for one of the studied systems. Moreover, similar field data provided by LANL has been used to show that hardware or node failures in a system have a knock on effect and increase the likelihood of follow up failures in the same node, nodes in the same rack, and nodes in the system as a whole [36]. Software errors, while accounting for a smaller percentage of job failures, can also be responsible for an application failing to successfully run to completion and a loss of valuable simulation data as well as a degradation in system availability. In particular, Martino et al. [60] have found that while software errors accounted for 20% of node failures in the Blue Waters HPC system, this corresponded to 53% of the node repair hours to keep the system operational.

Given the likelihood of failures in HPC systems and the consequences for users attempting to conduct important and expensive experiments, it is important that measures are taken to protect against the loss of valuable solution data due to component failure or calculation instabilities [74]. Time and resources devoted to checkpointing the the avoidance data loss, while necessary, are essentially wasting CPU cycles that could be devoted to valuable scientific discovery. It is therefore desirable to minimise the time and resources that are required for this task. Minimising this effort can be broadly approached in two different ways, being the reduction in the number of checkpoints written or a

reducing the time required to commit the required data to storage. The first of these approaches is based heavily in the realm of reliability and dependable systems, with a focus on detecting and modelling error behaviour in systems with a view to informing the optimal checkpointing frequency. El-Sayed and Schroeder [37] provide a methodology for computing the checkpoint interval to use based on much older work proposed by Young [100] in 1974. Specifically, their work attempts to demonstrate a practical application of the checkpointing formula that accounts for an up to date estimation of the mean time to failure for the system and the type of failures that have occurred most recently. This approach demonstrates performance comparable to that seen with an optimal checkpoint interval, but vitally relies on the data required to support the components of the methodology being accurate, up-to-date, and crucially made available to users. The authors recognise that even under optimal checkpoint placement, the limits of traditional coordinated checkpointing might be reached soon, and alternative techniques will be required to maintain viability.

The field of performance optimisation and engineering to reduce data storage times is one that is much more aligned to traditional HPC method and practices. While informing checkpointing frequency with dependability analysis can go some way to reducing data storage load, it is the case that checkpoints, scientific results and visualisation data will always need to be preserved. In the case of results and visualisations, these are crucial for the purposes of analysis and reporting experimental outcomes. The need to store data in these categories exists as a component of the reliability problems discussed here but also as a motivation for performant storage independently of reliability issues. In other words, simulation data will need to be stored regardless of the likelihood of a system failure occurring. It is the study of data storage and I/O performance that is the focus of the work contained in this thesis, with the goal of the research outcomes being relevant to the writing of checkpoint file and other simulation data alike.

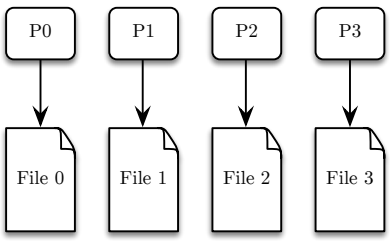
2.2.1 Issues in Parallel I/O and Data Storage

The process of moving data from a large number of distributed compute elements to a potentially equally large number of storage devices introduces a number of complexities that do not typically plague serial I/O activities. At a high level, there are a number of schemes governing the mapping of data from the compute element that generated it to an output file. The approaches commonly adopted can be seen in Figure 2.1.

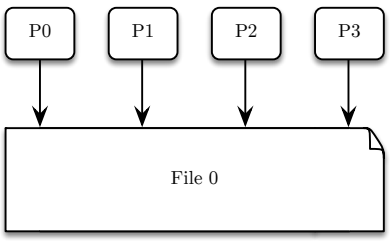
One of the simplest 'parallel' strategy that can be considered is known as file per process, or N-to-N where we have N processes, with each process writing to its own file (shown in Figure 2.1(a)). Due to the fact no two processes are writing to the same file, this approach is not considered to be 'true' parallel I/O. Similarly, it is possible to commit data to a single output file using a single nominated writer, traditionally process 0. Figure 2.1(c) demonstrates how each rank communicates its data to process 0, which is responsible for aggregating and writing data to the single output file. As with N-to-N, this Aggregated N-to-1 scheme is not truly operating in parallel with regards to I/O, as the distributed simulation data is essentially serialised by the aggregating process to be committed to storage.

The single shared file, or N-to-1, approach is what is most commonly considered as 'true' parallel I/O. In this mode, every process accesses a single file, either collectively or independently committing their data to different regions in the file.

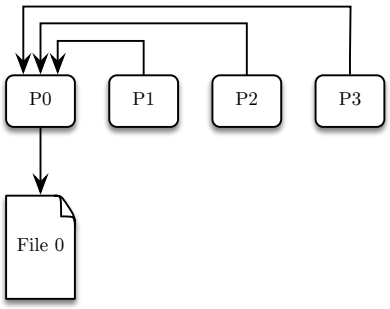
Additionally, there are hybrid approaches to file I/O, which select a number of output files between 1 and N. Within this N-to-M mapping, some of the previously mentioned approaches can be incorporated to control the access of processes in each of the M groups. For example, within each of the M subgroups an aggregating process can be nominated or each process can access the groups file in the N-to-1 scheme. An additional I/O pattern sometimes deployed for N-to-M access serialises access to each of the M files for the processes in the subgroup. A 'poor man's' parallel access pattern is used, with each process



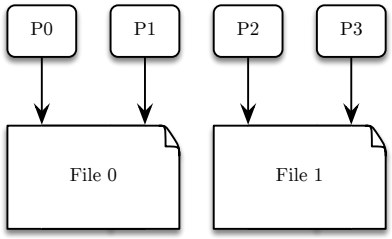
(a) N-to-N



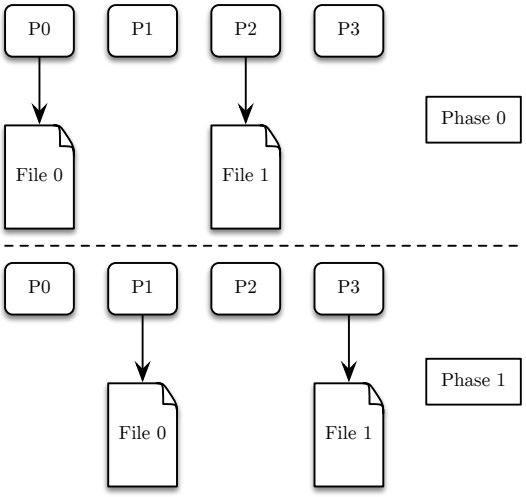
(b) N-to-1



(c) N-to-1 Aggregated



(d) N-to-M



(e) N-to-M Sequential

Figure 2.1: Parallel I/O Strategies

waiting for control of a baton before it opens the file and commits data.

Each of these approaches to I/O in a parallel system deliver different performance characteristics, running into different limiting factors and exploiting system architectures in contrasting ways. In general, the use of a single file per process (as per Figure 2.1(a)) is often the most performant strategy, but creates issues with regards to file management at larger process counts. Additionally, attempting to restart a simulation using a different number of processes is not possible without additional computational cost or data movement.

Conversely, the approach shown in Figure 2.1(b) is one that is commonly used but has the potential to deliver poor performance if not optimised well for the system hardware and software. Often performing I/O in parallel to a single-shared-file is done with the use of middleware and higher level libraries, which can simplify the implementation at the application level but requires a well designed parallel file system and library implementations to offer performance comparable to other approaches.

The potentials for trade off between complexity and performance for different file schemes is interesting in the context of application focused performance. In particular, the availability of libraries that implement these approaches goes some way to ensuring that more intelligent parallel I/O are accessible to application developers, some of whom may lack detailed domain knowledge of I/O. Moreover, a transition period for HPC seeing I/O burden being migrated to alternative burst buffer style systems will necessarily facilitate the adoption of alternative techniques where, for instance, a straightforward single shared file pattern can no longer be used. One of the objectives identified for this work, namely the performance study of representative HPC applications, seeks to identify the role that different file access strategies plays in the context of I/O libraries and future hardware. An evaluation of these different approaches would appear to fall firmly in the scope of this objective.

2.2.2 Parallel File Systems

Permanent data storage in a regular personal or desktop computer is a relatively straightforward process, usually involving the transfer of data from memory to an on-board hard disk drive (HDD) or solid state drive (SSD). This system, coupling a single computing unit with a single storage device, is generally sufficient for the I/O requirements of desktop applications. However, distributed systems comprised of multiple computing units and storage devices require a decoupling of these elements to parallelise data operations. Doing so avoids the serialisation of accesses to a single device, which is itself serial in its operation due to all the data being routed via a single connector. The mechanics of HDDs and SSDs are discussed in more detail in Chapter 3.

To provide a much greater storage capacity and eliminate unmanageable contention from concurrent access to a single disk, distributed file systems (DFS) are widely deployed in distributed computer networks and high performance clusters. The basic principle behind the design of many DFSs is to have data spread across independent storage devices, and furthermore spread storage requests across multiple storage servers to widen the available channels for the movement of data from the host to the storage medium. This structure can also be achieved through the use of some versions of a Redundant Arrays of Independent Disks (RAID) [69], which is deployed widely as a component of DFS.

2.2.3 I/O-aware Scheduling

Parallel file systems (PFS) such as Lustre (described in Section 3.3.1) are a shared resource accessible by all users and nodes in a HPC system. The size of performance of the PFS in relation to the scale of the machine and number of users therefore has a great deal of impact on the likelihood of a job running on a portion of the machine from suffering I/O resource contention. To illustrate this, between generations of IBM Blue Gene systems at Argonne National Laboratory

(ANL) there was an 20 times increase in compute resource but only a 3 times increase in the total I/O throughput of the attached PFS [4]. Comparison of the expected performance of different system components shows that a fraction of the compute nodes are able to saturate the bandwidth of the PFS, in the case of Mira at ANL this fraction is around a quarter. Focusing on I/O performance as a system wide problem rather than with an application centric focus, the reduction of interference between jobs on the system is something that has the potential to improve the quality of service experienced by all users.

I/O-aware scheduling is an approach that has been investigated with the intention of mitigating I/O congestion using a system wide batch scheduling approach. This approach is managed at the level of the batch scheduler for the reasons that a global coordinated knowledge of jobs and resource availability is needed to successfully manage the jobs attempting to stress the I/O subsystems. Specifically, the scheduler must have the ability to initiate, suspend, and terminate user jobs based on its scheduling policies independent of user activity [91, 92]. Herbein et al. [46] put forward an I/O-aware scheduling component to be integrated into existing batch scheduling system used at Lawrence Livermore National Laboratory (LLNL). Their work demonstrates how the use of a scheduler with an internal I/O contention model maximises the percentage of runtime individual jobs spend doing blocking I/O. Furthermore, the consistency of a job's performance under an I/O-aware scheduler is seen to increase as a job will not be launched unless the required I/O resources are known to be available. A key point identified by both Herbein et al. [46] and Zhou et al. [102] is that while a number of the performance and contention metrics used to evaluate I/O resource usage show scheduling to be an effective technique at job level, the overall system utilisation and turnaround times for jobs as a whole fail to improve and in some cases are worse than the baseline. Because of these results and the continued focus on compute utilisation in production systems, I/O-aware scheduling techniques are not currently implemented in live systems and the importance of application level I/O performance continues to persist.

The application centric focus of I/O performance that marks a key objective of this thesis is one that has the potential to benefit from I/O-aware scheduling. As discussed, this technique operates at a system-wide level and deploying workloads on such a system have been shown to operate with greater consistency. However, the approach of blocking an individual user's job from running until I/O resources are unimpeded does not necessarily reduce the time to that user completing their required workflow. Furthermore, improper use of I/O libraries and the underlying parallel file system are not addressed by a scheduling based approach, instead these techniques would be complementary to the work in this thesis rather than an alternative approach to achieving the performance study objectives.

2.2.4 Parallel I/O Software Stack

The file systems deployed alongside HPC systems enable data to be stored in parallel efficiently. For applications to perform their data transfer to these files systems in parallel, a great deal of programming complexity must be overcome to avoid collisions from distributed processes. Protecting data output from race conditions and file corruption requires careful calculation of file offsets or a defensive file locking approach. To reduce this complexity, a variety of parallel I/O middlewares and higher level libraries exist to provide an interface to the file system with simpler programming semantics.

Middleware

Portable Operating System Interface (POSIX) provides a set of standards describing an API and semantics for a widely portable interface to file systems. However, the POSIX interface lacks the required semantics for optimized parallel I/O patterns, and hence a higher level middleware abstraction was required.

The first step of achieving a portable I/O abstraction for parallel applications was introduced by the Message Passing Interface (MPI) Forum as part of the MPI-2 standard [85]. The I/O specific portion of the standard, called MPI-

IO, outlines an extensive parallel API with specific mechanisms for performant operation.

Underpinning the original goal of the MPI-IO standard is the idea of modelling I/O as being similar in execution to how message passing is carried out between distributed processes. Against this paradigm, the practice of writing to a file can be compared to sending a message. Similarly, processes in a simulation can view reading from a file in the same way as if a message was being received from a neighbouring process.

As MPI-IO is itself a specification there have been a number of portable and machine-specific implementations [40, 51, 70, 83]. Of these implementations, ROMIO [84] is by far the most commonly utilised and underpins OpenMPI, MPICH2, and a number of vendor-specific MPI distributions.

Implementations of MPI-IO permit concurrent I/O from distributed processes via *independent* or *collective* operations. The former carrying out file operations with no coordination from any other processes in the simulation, while the latter requires each process in a communicator group to participate. Furthermore, collective operations can allow the implementation or underlying file system to analyse the individual I/O requests as a whole, enabling request merging and optimisations to be made and improving performance.

MPI-IO introduces two important performance features to optimise the requests that are issued to parallel file systems. The first of these is referred to as *collective buffering* or *two-phase collective I/O*. Data is re-organised between processes to a subset of aggregators in a communication phase, following which data can then be sent to the file system. By aggregating data, a smaller number of processes can issue larger requests utilising more of the available bandwidth and better matching the data layout in file. Collective buffering can generate improvements for both small and large requests; in the former case, combining small accesses into larger ones and the latter splitting operations into multiple phases with overlapping communication and I/O.

Secondly, data sieving is a mechanism that can reduce the performance hit of

issuing many small I/O requests. Instead of reading or writing to multiple non-contiguous locations in a file the entire region is read or written via a temporary local buffer. In the case of writing, a contiguous block of data is read into the buffer and small non-contiguous regions are modified before writing the entire block as a single large write. While this has been shown to improve performance for a number of applications, many modern parallel file systems are capable of performing a similar optimisation and allowing MPI-IO based data sieving can inadvertently affect performance [24].

Collective operations, collective buffering, and data sieving are now well established mechanisms for MPI based parallel I/O to be handled in user applications. For this reason they can often be overlooked by application developers as important strategies and optimisations that can be applied both directly and indirectly to I/O operations. The uptake of higher level abstractions to handle the lower level details of how I/O will be performed by an applications further reduce the focus on what workloads are doing at the middleware level, this in part driven by the complexity and domain knowledge required to hand craft efficient MPI procedures. Given a key objective of this thesis is to understand I/O behaviour with a representative application centric focus, an evaluation of these techniques applied to real workloads is a valuable avenue for exploration. Furthermore, the access to and control over middleware operations from higher level interfaces is a pertinent question to consider with the popularity of these interfaces among applications and scientists.

A strategy that is implemented by many MPI distributions to reduce bottlenecks is to allow for the overlapping of operations. This is achieved through the use of calls to non-blocking functions, with program execution allowed to continue without waiting for the function to complete. The use of non-blocking over blocking functions enables an application to perform MPI operations asynchronously, meaning that the operation will return immediately regardless of completion and program execution can carry on to the next instruction. The typical model for asynchronous MPI operations is designed around point-to-

point non-blocking communication, where a send and receive are called on two communicating processes and both continue their execution without any expectation of the communication completing successfully [20]. Any benefits that can be drawn from asynchronous progress require an MPI implementation that supports progress threads, these being dedicated threads that poll the non-blocking operation for the completion status [96]. Dickens and Thakur [25], Ma et al. [59] demonstrated how this mechanism could be used for collective I/O performance, but found that naïve usage of threads could decrease performance. Furthermore, Patrick et al. [68] presented a similar approach spawning a thread when non-blocking MPIIO functions are called which in turn will manage the operation of its blocking counterpart. The use of asynchronous non-blocking I/O routines has potential to alleviate the bottleneck on application progression, however this relies on an MPI distribution being available on the system that supports these routines in addition to this support being built in at compile time. From the perspective of an application developed, asynchronous I/O through MPIIO may not be possible due to the requirement for data to remain unchanged until the routines transferring it have been confirmed to have completed to avoid data corruption. In order to continue to operate on simulation data that is being asynchronously written to storage, the application would have to wait for the progress thread to have completed or create a fixed snapshot of the data. This in turn places additional load on the available node memory and would be undesirable for memory intensive applications. The use of higher level libraries, discussed in the next subsection, also removes some of the control of which MPIIO operations are used to perform data transfer and users wishing to make use of these libraries are limited by the implementation choices of the library developers. As a technique that requires fundamental change to application design to enable uncorrupted overlap of computation and I/O, this work does not currently explore the use of asynchronous I/O, but recognises it as a suitable candidate for future continuation of this research. Moreover, a focus on the industry specific TyphonIO library for the replication and performance

studies carried out in this thesis preclude invasive changes to the internal implementations of the I/O stack as this is not currently a practical avenue of exploration for the industry project.

High Level Libraries

While MPI-IO provides a portable software layer with some of the key mechanisms required for parallel performance, the data files that are produced are themselves not standardised or portable between applications or systems. Higher level libraries have been developed in part to address this problem by defining self describing data models that can be easily written and read by library calls or standard tools.

Furthermore, libraries have been developed to simplify I/O from the perspective of the application developer by abstracting away low level details such as the calculation of file offsets. In doing so, some of the burden of managing I/O operations to maximise performance has also been taken on by these libraries to streamline I/O across the board as much as possible.

The Hierarchical Data Format version 5 (HDF5) is a data model, software library, and file format that can be built on top of I/O middleware to deliver the required performance and portability for applications [41]. As the name suggests, the HDF5 data model and file format is based around a hierarchical structure similar to a directory and file structure, with data being organised into containers and labelled with user-level attributes. When built with support for parallel operations, HDF5 can operate independently or collectively using the mechanisms provided by middleware. Due to its portability and continuing development, HDF5 continues to be used for storing mission-critical data by a number of industrial and academic organisations [22, 47, 99].

Even at this higher level of abstraction, there is still a great deal of responsibility on application developers to shape their I/O routines in a way that is scalable and optimised for their target system. This inevitably results in a design decision being made on a per application basis as to whether a single

shared file or file per process approach is taken, neither of which scale well to future exascale machines [57]. To address the need for flexibility, the Adaptable I/O System (ADIOS) [56] decouples the descriptive parameters of how data is structured and how it should be handled from the library routines built into an application. I/O behaviour can therefore be changed transparently as far the application is concerned simply by adapting an external XML file. ADIOS specifies a number of available transport methods for handling data under the covers, including classic POSIX; independent or collective MPI-IO; MPI-IO optimised for the Lustre parallel file system (Lustre is described in Chapter 3); an aggregate MPI-IO approach; and even other high level libraries such as parallel HDF5.

Similarly to how ADIOS adds a further abstraction to the software stack, HPC institutions have long since devoted considerable efforts to developing their own I/O layers and interfaces, both to handle specific application needs and in attempts to standardise practices across the code bases. Moreover, bringing application suites together with regard to their data handling increases the interoperability of output files and reduces the complexity of optimising I/O performance for all application users. Libraries such as HDF5 and NetCDF are used as the basis for in-house packages, as is the case with the TyphonIO [88] and Silo [62] libraries developed by AWE and LLNL respectively. Both of these examples persist data with the help of the HDF5 file format and its library routines, while imposing their own scientific data model and I/O pattern on top.

As discussed in the context of MPIIO middleware, a key objective of this thesis is to understand I/O behaviour with a representative application centric focus. The application centric focus is a crucial part of this for organisations like AWE as application performance and user experience is understandably the critical factor that progress is made towards. Specifically, targeting specific production applications with a top-down approach is common practice, this is where high level libraries such as TyphonIO and HDF5 are deployed to handle

the burden of performing complicated parallel I/O routines outside the expertise of domain scientists. Correct use of these libraries is not always guaranteed, and it is important to undergo performance studies to highlight how different application workloads are making use of I/O libraries and the scope that exists to improve performance within the eco-system as a whole. An example of such work being the combination of a vanilla TyphonIO build and an alternative file access strategy detailed in an earlier section.

2.3 Performance Analysis and Engineering

The driving force for HPC is to enable faster, more accurate scientific results. Procuring larger, more powerful, systems seeks to fulfil this goal but is heavily dependent on the practice of *performance engineering* to guarantee success. Motivations for performance engineering are two-fold. Firstly, achieving what is known to be the peak performance for an operational system; complemented by efforts to predict performance, and secondly support the procurement of new machines to efficiently and economically provide greater capabilities than are currently available.

A collection of different approaches to performance engineering are used to measure, predict, model and replicate the behaviour of parallel applications. These techniques often require sophisticated software tools to capture and understand the performance intricacies of complex workloads interacting with equally complex systems.

2.3.1 Benchmarking

The first port of call for many performance engineers is to gain an accurate representation of what a system can and is achieving with regards to performance. Benchmarking is the practice of physically executing code on a system to obtain data to compare to expected values or to serve as the baseline to be compared against. This practice can be performed for both hardware and software.

By far the most famous HPC specific benchmark is the LINPACK linear solver code, designed to measure a system's peak floating point computing power [34]; the metric used for this purpose being floating point operations per second (FLOPS). The results returned by LINPACK are used to rank the most powerful supercomputers in the well established TOP500 list [33].

Having an understanding of precisely what a particular benchmark is measuring is crucial to building up an accurate picture of performance. For example, LINPACK represents a largely compute bound problem and hence naively reports peak FLOPS that may be unobtainable by any real calculation due to the host of other performance limiting factors experienced by large systems. Features of memory bandwidth, network communication, and data storage are missing from the LINPACK performance value. This fundamental limitation has been widely acknowledged, but demonstrates the risks involved in attempting to benchmark complex interoperating sub-systems as a whole.

One approach taken to ensure benchmarking results remain representative of the features they are targeting is to use heavily simplified or low-level applications known as *micro-benchmarks*. By exercising specific components as independently as possible from the rest of the system, a more accurate picture of the performance thresholds for the respective components can be learned.

Much as a great deal of work has been done to benchmark the components of a system responsible for computational performance, so too has this process been carried out for parallel file systems and data storage capabilities. In this area, tools such as b_eff_io [71], iozone [66], mpi-tile-io [7], The Flexible File System Benchmark [7], benchio [38], and IOR [35] are all synthetic benchmarks with a focus on benchmarking storage bandwidth to a parallel file system under different scenarios. As file system performance is dependent on factors other than just storage bandwidth, applications like mdtest focus specifically on metadata performance for file creations and deletions to asses the rate at which these can be processed by the file system.

The highly specialised approach to benchmarking seen in many of the ap-

plications listed suffers the drawback of failing to capture the characteristics of interconnected components and consequently is not suitable for gaining comprehensive system performance insights.

Moving the focus away from evaluating just the file system, application benchmarks and mini-applications seek to exhibit behaviour akin to that found in a real application or software library. The focus in this case is on benchmarking the data storage pattern taken from an existing simulation to assess how well the parent application might perform on a new system or using an alternative I/O library. These benchmarks have also successfully been used as optimisation tools; their representative nature and relative simplicity making the performance tuning process much simpler for performance engineers. This process has been demonstrated by the FLASH-IO [103], S3D-IO [18], MADBench2 [13], and Chombo I/O [7] benchmarks which were all derived by extracting the I/O kernels from their respective parent applications.

2.3.2 Profiling

Tracking the time it takes to execute an application or benchmark is a straightforward way of comparing relative performances of machines or their components. Effective performance engineering requires a great deal more insight, specifically capturing the characteristics of a run with more fine grained data. This is achieved through the intermittent sampling of system state or through capturing individual function calls and recording parameters and other data instrumentation. Both of these approaches are used in the practice of system and application profiling. When making use of application profiling techniques, the output generated is usually in the form of a profile or trace file that can be interpreted to show characteristics or a full record of the application during its run. For the case of a straightforward profile, a high level representation of the run can be used to assess overall behaviour through counters and statistic based metrics. Alternatively, a trace file can allow for the run to be replayed function by function and so low level code execution behaviour can be investigated.

For the parallel computing domain, profiling requires the collection of data from a potentially large number of distributed processes executing different portions of code. Consequently, simple profiling tools such as *gprof* [42] and *perf* [23] must be replaced by more sophisticated parallel profilers.

Intel Vtune Amplifier [50] and Arm MAP [8] are profiling tools that identify where performance hotspots occur in an application, in addition to monitoring features such as cache usage and communication synchronisation. Data is available on individual function calls as well as on hardware counters to identify and address performance bottlenecks and measure how effectively hardware is utilised. Similar tools like mpiP [90], TAU [78], Caliper [12], and the HPC Toolkit [2] offer similar capabilities with varying degrees of sophistication and usability.

The majority of monitoring and profiling tools available will provide some I/O capability alongside more general execution data. This can often be limited in detail for I/O specific counters and fail to profile higher level library calls. *Darshan* is a scalable I/O profiling tool that is specifically designed to capture an accurate picture of application I/O behaviour [16]. The lightweight interception and aggregation of file operations at the POSIX, MPI-IO and HDF5 levels enable Darshan to efficiently capture data over the duration of a run. Moreover, sites such as Argonne National Laboratory (ANL) are enabling Darshan on production systems by default to make I/O profiles available to users with no intervention.

For the work carried out in this thesis, profiling is a key technique that is used to capture application workloads to allow for their characteristics to be compared to synthetic workload replications.

2.3.3 Simulation

In a number of situations performance analysis goals cannot be achieved through execution on real hardware. Particularly when looking to evaluate the performance of new hardware architectures that aren't yet physically deployed, or

to drill down into the characteristics of an application where data collection is difficult, simulation is a valuable tool for performance engineers.

The Structural Simulation Toolkit (SST), developed by Sandia National Laboratory (SNL), is a framework for micro-, meso-, and macro-scale simulation [72]. Contained in SST are a number of built-in models for hardware components and network topologies, enabling novel programming models and hardware organisations to be explored. SST has been effective at studying specific execution characteristics such as memory access and network communications, but as far simulation of parallel file systems there is a notable gap in capabilities.

Replicating the behaviour of an individual HDD at a low level can be a challenging task in and of itself; specifics such as data layout and cache utilisation can dramatically affect the behaviour observed despite possessing a good understanding of the hardware specification. Despite this difficulty, there are a number of storage simulators that focus on imitating drive and RAID array behaviour, DiskSim [14], RAIDFrame [49], Pantheon [95], HRaid [21], StorageSim [73], and SIM-Array [98] to name some notable examples.

Simulation of networked parallel file system presents added difficulty, with additional components such as the separate data and metadata concerns in architectures like Lustre adding behavioural complexity. The IMPIOUS parallel file system simulator attempts to approximate the behaviour of real systems with just enough detail to observe important characteristics [65]. This is achieved with the use of abstract simulation models of object-based parallel file systems, driven by parallel I/O traces. It was demonstrated that IMPIOUS's accuracy was sufficient to observe the trends of the simulated file system, but results underestimated performance values by a factor of ten.

A more recent example of work to produce PFS simulation capability can be seen in the Co-design of Exascale Storage Systems (CODES) framework [19]. CODES has been developed to evaluate storage designs for upcoming exascale supercomputers, notably including up-to-date storage technologies such as SSD

and NVMe based burst buffers and I/O accelerators [54].

2.4 Summary

High performance computing is a field in constant motion, delivering more powerful supercomputers in the pursuit of greater scientific understanding. The evolution of HPC systems, and the parallel techniques required to exploit them, is a source of constant research and development. This chapter has presented an overview of parallel computing techniques and research focused on the problem of parallel I/O.

Parallel file systems have been deployed by organisations to deliver a degree of storage performance to their HPC machines, and the I/O software stack makes this accessible to application users and developers. Despite this I/O never ceases to trouble scientific applications, and breaking the exascale milestone is likely to require a more complex storage hierarchy. Engineering I/O performance from new systems relies on a three-fold approach, combining sensible application behaviour, library optimisations and file system tuning (examined in further detail in Chapter 7). The research presented in this thesis examines the process of I/O performance analysis of scientific applications through workload replication, contrasting to traditional synthetic and kernel benchmarking approaches. It will also address the question of how relative performance can be measured for applications against theoretical and realistic system expectations, quantifying the performance envelope with which developers can confidently work within.

CHAPTER 3

Parallel Hardware and Software Overview

The work contained in this thesis is supported by, and makes use of, a number of hardware and software components. A proportion of these are specific to high performance and parallel computing, however some are more widely applicable to computing in general. This chapter provides an overview of these components, focusing on storage systems in particular, as well as computing platforms and benchmarking applications used in this work.

The material in this chapter is provided to give a complete understanding of the components that are used to construct modern day parallel file systems to handle the I/O traffic generated by parallel compute clusters. In particular, one of the objectives of this work is to understand how applications can employ new and existing file access modes or strategies to avoid contention and similar limitations on parallel file systems like Lustre. To aid in this understanding, the description given of Lustre and some of its fundamental components provide valuable context when attempting to reason about where inefficiencies in I/O performance may be coming from in later chapters. Furthermore, details of physical storage devices are explained here to provide a basis for understanding the limitations of the parallel file systems that are tested. One possible solution to these limitations is the inclusion of node-local burst buffers in HPC systems which motivate some of the later experiments in Chapter 7, highlighting here how their physical design can offer greater storage performance than traditional hard disk drives gives a point of reference point on the results we can expect when testing one of these systems. These storage devices offer great potential improvements over current generation parallel file systems, a quantification of which is made possible by the work covered in later chapters.

3.1 Storage Hierarchy

In modern computing systems, data storage components support a number of important functions. The differing requirements of these functions call for a hierarchy of storage devices with different physical mechanisms; the purpose of such a hierarchy is to effectively handle the competing requirements of capacity, performance and cost to service the use cases of a system.

3.1.1 Hard Disk Drive

Storage technology has continuously developed to offer greater capacity and performance at lower cost though the adoption of solid state drive (SSD) devices. However, the current generation of parallel storage systems still rely heavily on the conventional mechanical spinning hard disk drive (HDD). Consequently, understanding the characteristics of a HDD is important when considering the performance of these parallel storage systems.

The key mechanism underpinning the operation of a conventional HDD is the storage of data on spinning magnetic disk platters. The basic internal structure of the drive assembly can be seen in Figure 3.1.

HDD are available at different grades, meaning different physical characteristics of drives give variable reliability and performance. The three most widely deployed types of HDD are:

- *Serial Attached SCSI* (SAS) - SAS disks offer the greatest reliability, maintain performance under more difficult conditions and offer a greater peak performance than other drive grades. The SAS interface contains a different feature set to that found in SATA that enable greater performance such as command queuing and concurrent data channels.

- *Nearline-SAS* (NL-SAS) - A NL-SAS drive is an enterprise SATA drive with a fully featured SAS interface. As a result, the drives are able to exploit some of the features of the SAS interface but still maintain the

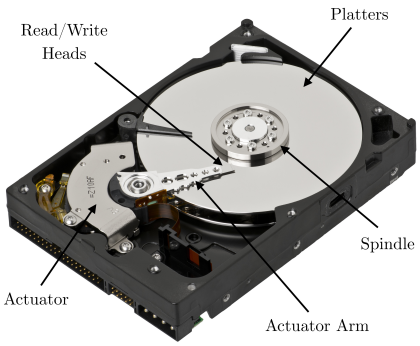


Figure 3.1: Basic structure of a hard disk drive assembly.[1]

reliability characteristics of a traditional SATA drive, with performance largely resembling that of SATA also.

- *Serial ATA* (SATA) - Of the three classes of drive, SATA offers the best price per capacity but maintains a order of magnitude worse reliability than SAS in terms of bit error rate. SATA class HDDs have lower rotational speeds than are used for SAS, by far the biggest indicator of the performance difference between the two.

The disk platters are made up of multiple layers of ferromagnetic disks, which are accessed via a moving read/write head. To perform a data read or write to the disk, the read/write head must seek to the correct position on the relevant magnetic disk to retrieve or set the polarity of a location; this polarity encoding either a 1 or a 0.

3.1.2 Solid State Drive

SSD storage devices have existed since as early as 1978, however the technology has remained too expensive for broad adoption. In contrast to the mechanical spinning disk based HDD, the design of SSDs contain no moving parts.

The mechanism underpinning the design of SSDs is the use of memory like

[1]Image includes resources from: https://commons.wikimedia.org/wiki/File:35-Desktop-Hard-Drive.jpg
[2]Image includes resources from: https://commons.wikimedia.org/wiki/File:Sf-ssd.jpg

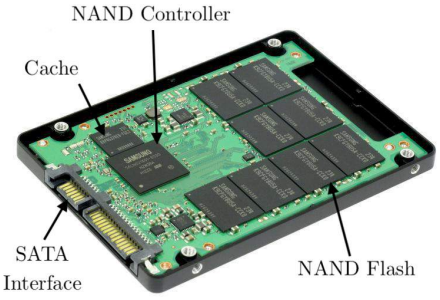


Figure 3.2: Basic structure of a solid state drive assembly.[2]

technologies as the storage medium, coupled with a controller to read and write states to the medium. Traditionally, the storage medium used in SSDs has been Dynamic random-access memory (DRAM). However, due to the lack of voltage persistence in the underlying capacitors, these devices are not considered to be non-volatile. To allow DRAM based drives to store data in the absence of power from the host system, an internal battery or external power adaptor is required. DRAM permits very fast data access, but has a relatively high cost per byte for a storage medium. As a result, a more commonly used SSD design relies on NAND flash memory as the base storage medium.

NAND flash memory is a non-volatile memory (NVMe) technology that traps an electron charge on a capacitor indefinitely in a no-power state. A high voltage pulse adds or removes electrons from the capacitors to write data to the storage medium. The charge of a NAND cell is sensed by circuitry on the chip, and data is read from the device using an analogue to digital converter. Devices using NAND flash can either be single-level cells (SLC) or multi-level cells (MLC); the difference between these is that SLC based devices can only store a single bit of information using one of only two voltage levels. MLCs will typically use four distinct voltage levels to store two data bits per cell. Intuitively, a MLC based SSD will have a much higher storage density, and hence much larger volume SSDs can be produced.

A characteristic that SSDs display which is not seen in traditional HDDs is the limit on the number of write cycles that a drive can undergo. For a write

to be performed, an erasure operation on the NAND flash chip must first be performed by supplying a large electrical charge to the chip. This charge causes a small amount of degradation to the physical semiconductor layer of the chip, and after a certain number of erase/write cycles the chip can no longer effectively store a data charge. The predicted write limit of an SSD is dependent on the geometry of component flash dies and whether or not SLC or MLC technologies are in use. The reduction in physical size of storage chips and increasing use of MLCs to increase the storage densities of modern SSDs both contribute to a reduction in the expected lifetime of a drive.

A combination of the hardware lifetime limits and price-per-byte characteristics of SSDs have contributed to the fact that, at the time of writing, these storage devices still sit behind HDDs as the most widely deployed technology for servicing large scale and intensive data storage requirements, particularly at industrial scale.

3.2 File Systems

A file system manages the interaction between a user and the physical storage hardware. Without file system layers providing structure, data written to a medium would exist in one continuous body without any indication of where distinct elements begin and end. The file system implementation in use will dictate how interactions between the operating system and the storage hardware are conducted, influencing performance.

3.2.1 The Extended File Systems

The extended file system family, originating with the first extended file system (ext), is a series of file systems created exclusively for the Linux kernel. Originally based on the metadata structure of the Unix File System (UFS) and developed to replace the MINIX file system, itself developed as a cut-down version of UFS, ext was the first implementation to make use of the Virtual File

System (VFS).

The original ext implementation improved upon some of the limiting factors of MINIX, crucially, addressing a maximum size limit of 2 gigabytes and 255 character limit on file names. However, ext still possessed a number of limitations and consequently was superseded by the second version extended file system (ext2) almost immediately.

An important feature missing from ext, addressed by ext2, included support for separate timestamps for file access, inode modification, and data modification. Additionally, the second iteration of the file system attempted to address performance issues that arose through the use of linked lists to keep track of free blocks and inodes; the performance becoming worse as file system fragmentation increased.

Around 8 years after the introduction of ext2, the third version extended file system (ext3) added the concept of journalling. A journalling file system being one that uses a dedicated portion of the file system to track changes in a 'journal' data structure or log; the benefit of journalling is that file systems are more robust to a system failure and can be recovered much faster. Moreover, in some situations where journal entries are written to a contiguous disk region, performance improvements can be seen. Conversely, in situations where metadata and file contents must be written twice, poorer performance can be observed. Three levels of journalling are implemented in ext3:

- *Journal* - The first level is the most robust, with both metadata and file contents being appended to the journal. This is done before the data is committed to the main portion of the file system.

- *Ordered* - Offering a greater level of risk than the first, ordered journalling commits only metadata and not file contents to the journal log. File contents must be committed to the main file system before metadata associated with the file can be marked as committed in the journal.

- *Writeback* - The final and highest risk journal level also stores metadata

and not file contents. However, the journal contents may be updated before or after file contents are committed to the main portion of the file system; allowing files modified just before a crash to become corrupted.

Journals can be stored on the file system they are in turn journalling, but in many cases can exist externally on a separate device. To improve performance, SSDs or battery supported non-volatile random access memory (RAM) can provide journalling capability, while a regular HDD handles the main file system data storage.

The newest version of the extended file system, version four (ext4), builds directly on ext3 with an ability to support individual file sizes of 16 terabytes and a maximum file system size of 1 exabyte. Furthermore, features such as journal checksumming, faster file system consistency checks, delayed allocation and multiblock allocation improve the performance and reliability of ext4 over its predecessors. Originally these extensions were developed for the Lustre file system [77], and were designed to maintain a large amount of backwards compatibility with ext2 and ext3. A number of additional extensions to ext4 have been developed, and make up the ldiskfs file system for use as the underlying file system in a Lustre deployment.

In each of the extended file system versions, a key building block object is the *inode* data structure, describing all of the information for a file or directory except its name and actual data contents. Upon file creation a name and a unique inode number are assigned, and when a file is referenced by a user or application the file name is used to perform a look up of its inode. In addition to storing the metadata for a file, the inode structure contains 15 pointers that are used to indicate the starting point of the data blocks containing the file contents. The first 12 of these pointers are used to indicate the starting point of file blocks directly, while the remaining pointers use one, two, and three levels of indirection respectively. An indirect block pointer references a data block, that itself contains a table of addresses that point to data blocks. A single indirect block pointer uses this lookup table to then point directly to blocks containing

file contents, with double indirection having to go through two pointer lookup blocks in turn before accessing file blocks. This structure enables faster data retrial for smaller files, but through levels of slower indirection, a larger number of data blocks increase the total possible file size.

The inode block mapping scheme described is implemented in ext2 and ext3, but was replaced in ext4 by the concept of *extents*. One of the benefits of changing the internal structure of inodes in this way is a reduction in the amount of metadata that is required to keep a record of data blocks for large files. An extent is a range of contiguous physical blocks of up to 128 MB (assuming a 4 KB block size). The 16 pointers previously stored inside an inode are replaced by 4 extents, with each extent reserved and addressed at once. Furthermore, the use of larger contiguous blocks reduces file fragmentation and in turn can improve performance.

3.2.2 ZFS

ZFS is a file system and logical volume manager, developed by Sun Microsystems for the Solaris operating system. Historically, storage systems were created on top of a single hardware device, and the combination of file system and volume manager in ZFS addresses the use of multiple devices and provides redundancy. ZFS is designed to run on a single server, with many attached storage devices potentially numbering up to thousands. These storage devices are pooled and managed as a single entity, with a theoretical upper limit on scalability of 256 quadrillion zettabytes (2^{128} bytes).

In managing physical storage devices, ZFS uses the concept of virtual devices (vdevs). A vdev can be either a single device, multiple devices using mirroring, or multiple devices in a ZFS specific RAID configuration (RAIDZ). Available vdevs are pooled into a *storage pool*, which acts as an arbitrary data store to be used to create a file system. The model used by ZFS storage pools is similar to that of virtual file system (VFS), where new hardware can be added to the pool and is immediately available for use by the file system without additional

management effort.

When a storage pool is first created, a form of data striping is used across all of the available vdevs to maximise file system performance. Consequently, each vdev must have a sufficient level of redundancy to protect the pool against the loss of any of the vdevs, which could in turn cause the loss of the pool as a whole.

To ensure the file system is always consistent on disk, ZFS adopts a transactional model. File system inconsistency is traditionally a problem experienced when a system failure occurs midway through the process of committing changes to disk. For example, when creating a file, the loss of system power between data block allocation and linking into a directory would leave the file system in an inconsistent state. A common solution to maintaining consistency is to use journalling, as discussed in Section 3.2.1. Alternatively, transactional file systems rely on copy-on-write (CoW) data management semantics. During this process, existing data is not overwritten in place and a sequence of operations is either marked as completely committed or ignored in its entirety. To modify a file's contents modified data is written to new data blocks, and at completion relevant metadata blocks can then be read, reallocated and written to complete the update. Requiring data copies will naturally involve a performance overhead, which can be reduced by grouping operations into transactional groups.

Similarly to ldiskfs (essentially ext4 with extensions), ZFS can be deployed as the backend storage in a Lustre parallel file system, which is discussed later on in this chapter. Motivating factors for this adoption include the aforementioned extreme scalability, good underlying write performance, and built in redundancy ZFS offers.

3.2.3 XFS

XFS is a 64-bit journalling file system originally developed for the SGI IRIX operating system. From inception, the file system was designed to excel in handling parallel and streaming I/O for large multiprocessor systems and disk

arrays rather than small single disk workstations.

The design of XFS is based on the concept of allocation groups (AG), equal sized subdivisions of the physical volume that keep track of their own free blocks and inodes. Each AG can conceptually be thought of as an individual file system with a maximum size of 1 TB, this upper limit being independent of the underlying device sector size. It is the availability of multiple AGs in a file system that permit concurrent file operations without introducing contention and performance degradation.

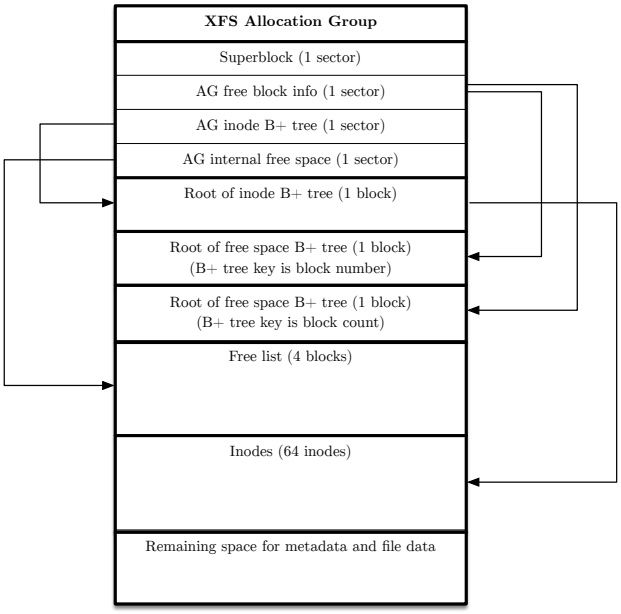


Figure 3.3: Internal structure of an XFS allocation group

The internal structure of an AG can be seen in Figure 3.3. The first block is made up of four sectors, beginning with the AG superblock. The superblock belonging to the primary AG is used to store aggregate file system information such as the total free space and number of inodes across the entire file system; subsequent superblocks are used only for recovery purposes on a primary superblock corruption.

Following the superblock sector is the free space block, which points to two B+ trees for tracking free space by block number and by block size. Each entry

in the B+ trees is a descriptor of a free extent in the AG, consisting of a starting block and length for the extent. By using two B+ trees with different indexing the process of searching for available free space is more efficient for the different types of allocation that are required. The other AG component required for free space management is a free list, containing an array of reserved space for growing the free space B+ trees that cannot be used by user data.

Storage of inodes is performed by a similar method to that of free space blocks with the use of a B+ tree. Inodes are allocated in chunks of 64 and the B+ tree is tasked with tracking these allocations, as well as the freeing of inode chunks. In particular, the file and metadata performance improvements XFS demonstrates over other file system approaches can be attributed to the efficiency of B+ tree, in part due to their ability to minimise the number of traversal operations to retrieve metadata and file data.

File system consistency is achieved in XFS with the use of journalling via a write-ahead transaction log. In contrast to the journalling implemented in ext3 however, journal updates can be performed both synchronously and asynchronously; the latter avoiding performance hits by decoupling the rate metadata is updated from the speed of the disks.

3.3 Clustered File Systems

File systems such as ext4 and ZFS are effective at managing direct-attached storage on a single local system; managing non-local storage between variable numbers of clients and servers in a way that provides unified file and directory structures however requires the use of clustered file systems. The term clustered file systems is a general one encompassing different categories of non-local storage, the distinctions of which are not necessarily universal. Furthermore, file system architectures may fulfil some definitions and design goals of more than one category. Consequently, the following terms are used to describe types of clustered file systems: networked file system, distributed file system, shared-disk

file system, and parallel file system.

Networked File System

A networked file system is one that allows a client to access data on remote storage devices via a network service. One of the most widely implemented networked file system solutions is the Network File System (NFS) developed by Sun Microsystems. In the NFS model, a server exports a directory structure to be mounted in a client's local file space alongside its direct-attached storage. The Remote Procedure Call (RPC) protocol enables the client to issue I/O commands over the network, which are translated by the NFS server into commands issued to its local file system. Internally, an NFS server accesses local storage devices via a traditional local file system (such as ext4); as such, NFS is not a traditional standalone file system and is sometimes referred to as a network abstraction to a file system.

Distributed File System

Distributed file systems (DFS) are considered to be those that build on the client/server model to present a unified file system backed by multiple servers. As with networked file systems there is no block level access to storage devices, with I/O requests handled by a network protocol. For this reason, DFSs are commonly categorised as or alongside networked file systems.

Spreading storage requirements across a number of devices, that are in turn accessed through multiple servers, generally yields better performance than would be achieved by a single server when scaling the number of clients and size of the network. Importantly, DFS protocols deliver a distributed storage capability transparently such that client nodes are unaware of the distinction between distributed and local files when they are accessed, and a consistent namespace for both is presented.

Shared-disk File System

A shared-disk file system differs from a DFS in that clients are able to gain direct block level access to storage devices. The connection between clients and file system servers is achieved across a storage-area network (SAN). Due to clients making direct disk accesses to remote storage, the translation from file level operations to block level operations must be done on the client side before being transmitted across the SAN.

Architectures for shared-disk file systems can differ, notably in how metadata is stored containing file information. A fully distributed architecture will balance metadata out across all of the file system's servers; the opposing approach relying on a centralized server to manage file information and records of where a file's data is stored.

Parallel File System

A parallel file system (PFS) is a particular type of clustered file system that distributes data across multiple storage elements to offer greater redundancy but most importantly better performance. PFSs have been designed to use both the distributed and shared-disk file system models, with examples such as Ceph [94] able to operate using both approaches depending on configuration.

3.3.1 The Lustre Parallel File System

Lustre is by far the most widely used parallel distributed file system for HPC systems. Furthermore, Lustre remains the file system of choice for the largest and most powerful machines in the world. Of the machines occupying the top 10 places in the global TOP500 list published in November 2017, seven use Lustre for their primary high performance storage systems[3].

The architecture of Lustre is based on the principle of object based storage, the key component of which is the management of data as objects instead of files

[3]The Sunway TaihuLight, K computer, and Gyoukou machines do not use Lustre; however, TaihuLight and the K computer have storage systems that are either based on or a branch of Lustre.

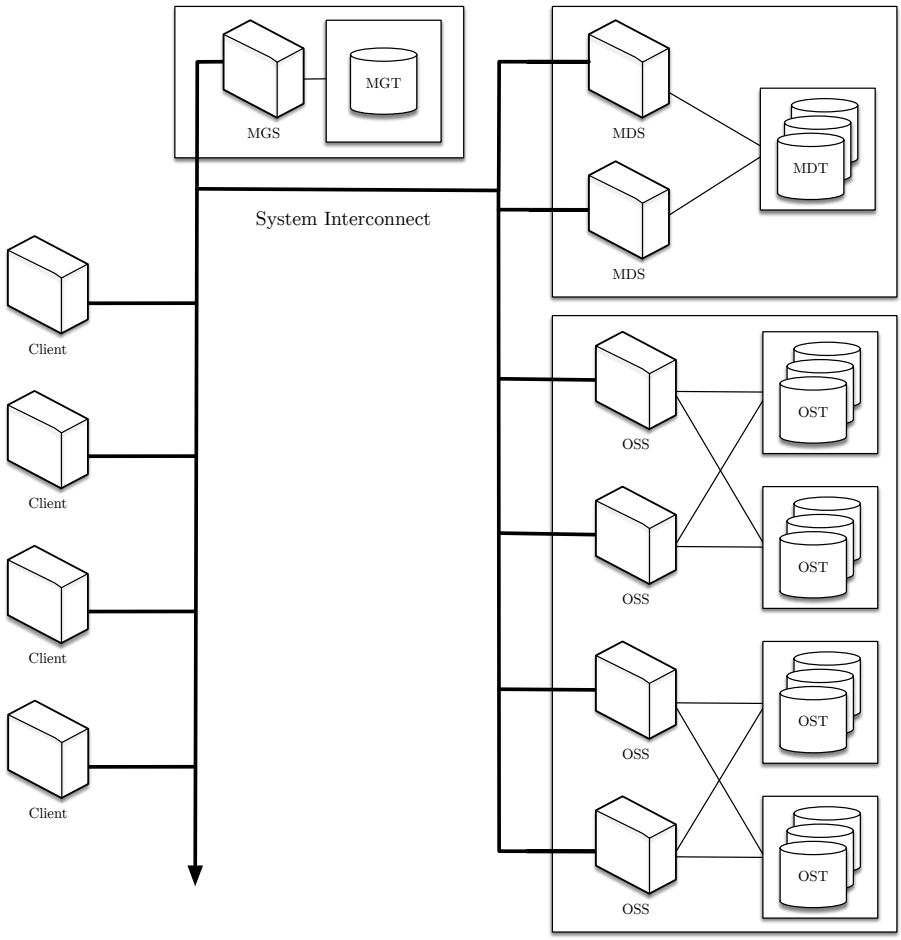


Figure 3.4: An example Lustre configuration with one MGS, four OSSs, four OSTs, and one MDT with two MDSs in failover.

or blocks. In designing storage this way, the concerns of metadata and actual file data can be separated. Most modern clustered file systems that share this architecture are based on six components, for which Lustre has a well defined naming convention. The structure of these file system components is shown in Figure 3.4.

Object Storage Target (OST)

The base unit of storage hardware, be it a single HDD or multiple drives organised into a RAID array, is referred to as an *Object Storage Target*. Each OST

manages a single local disk file system for performing block level data operations. The data blocks that make up a file are stored in stripes on one or many OSTs, with the total number of OSTs dictating the total capacity of the file system.

Object Storage Server (OSS)

Each of the servers directly connected to one of more OSTs are designated an *Object Storage Server*. Commonly an OSS will be primarily responsible for file data operations on a group of OSTs, with failover links to a second group of OSTs managed by another OSS. Overlapping the mapping of OSSs and OSTs in a file system provide high availability access to OSTs and protects the file system from individual OSS failure.

Metadata Target (MDT)

File metadata is stored separately to data on a dedicated storage device or devices. Similar to an OST, a *Metadata Target* can be made up of a single HDD or often a RAID array to provide greater performance and redundancy. Importantly, the namespace metadata stored on an MDT includes information such as filenames, access permissions and file layouts; unlike block-based PFSs no data on file block allocations are stored in a Lustre MDT, meaning the metadata server does not have to be directly involved in file I/O operations. Instead, block allocation data is managed internally by the OSTs.

Metadata Server (MDS)

A Lustre file system contains one or more dedicated servers controlling access to one or many MDTs, referred to as the *Metadata Server*. For a file system comprising multiple MDSs and MDTs, overlapping connections provides high availability and redundancy in the same way as can be achieved for OSSs and OSTs.

Management Server (MGS)

Configuration information for a Lustre file system required by servers and clients is managed by the central *Management Server*. It is possible for the MGS to run on a shared server alongside the MDS, however production file system typically have a dedicated MGS node.

Client

The Lustre client is mounted on a host's operating system to present a unified namespace for all of the files and data contained in the file system. Applications running on a host can interact with the file system using standard POSIX semantics to read and write data concurrently.

Communication between Lustre components is done through the *Lustre Network* (LNet) layer, which can operate across a number of interconnects. In HPC systems, InfiniBand is a common choice for high speed interconnect and is supported for LNet communication; alternatively Ethernet, TCP/IP and proprietary technologies such as Intel Omni-Path [11] or Cray Gemini [5] are compatible with the LNet layer.

Where made possible by the network infrastructure, Remote Direct Memory Access (RDMA) transfers can be exploited by Lustre. RDMA bypasses operating system buffers allowing the network adaptor to access data in application memory, reducing the CPU load and improving throughput when performing LNet communication.

Lustre Architecture

In a typical HPC system, the Lustre client is installed on each compute node's operating system allowing communication between the compute portion of the system and the Lustre OSSs and MDSs. With a Lustre file system mounted on each compute node, there exists an all-to-all mapping between distributed compute nodes and distributed storage components. Importantly, a HPC platform

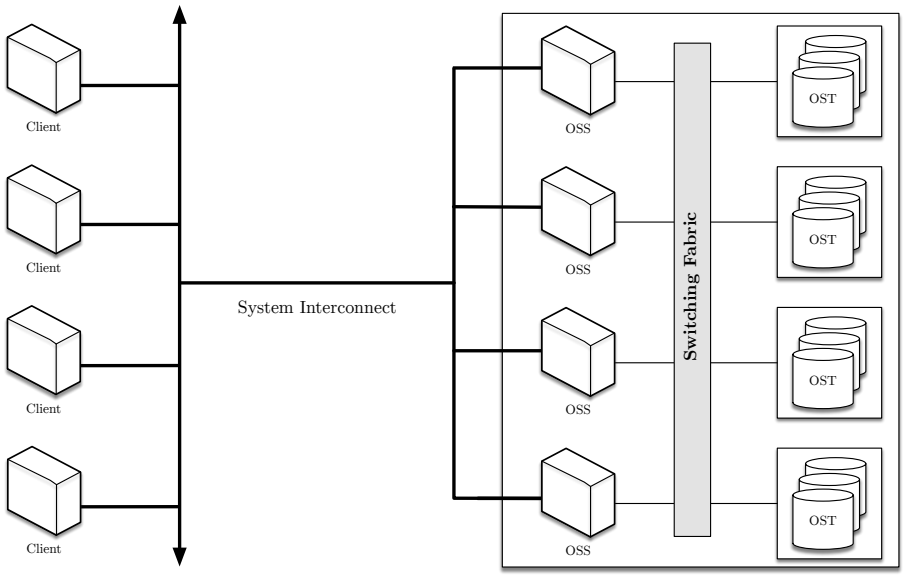


Figure 3.5: An example GPFS configuration with four OSSs and four OSTs connected over a switching fabric

can support multiple Lustre file systems concurrently each containing their own distinct MDS and storage target pools.

3.3.2 The IBM Spectrum Scale File System

The IBM Spectrum Scale File System (also known as the General Parallel File System [GPFS] [75]) is an alternative clustered file system, which is based on the shared-disk model. GPFS features in a number of machines in the TOP500, but is less widely adopted than Lustre; it does however share some common operational characteristics. In particular, both Lustre and GPFS distribute file data across multiple storage targets via a number of storage servers.

One of the key elements that differentiates GPFS from Lustre is the ability to operate with both distributed metadata and a centralised metadata target. When configured with distributed metadata, file information is stored alongside file data across all of the available OSTs in the system. This avoids a single point of failure and can improve performance for large numbers of concurrent metadata operations, which can be observed in the superior file creation rate

GPFS has over Lustre. A caveat to this is that file operations must be carried out in different directories due to the distributed locking GPFS implements to enable full POSIX semantics.

The OSS to OST mapping in GPFS has full connectivity, meaning all of the storage servers can communicate with all of the disks across a switching fabric. Having full connectivity in the file system maximises the potential for parallel data transfers to disk, which is performed under the control of the distributed lock manager. An example of this architecture is shown in Figure 3.5.

3.4 Computing Platforms

The work carried out in this thesis has made use of nine different HPC platforms. Two of the platforms are leading class supercomputers, entering the TOP500 list at the time of their commissioning and occupying rankings in the top 20. One of these was formally the most powerful machine in the world, hosted at Oak Ridge National Laboratory (ORNL) in the United States, while the other is the United Kingdom's national supercomputing service hosted at the Edinburgh Parallel Computing Centre (EPCC). A further three systems are built of commodity hardware, one hosted at the Atomic Weapons Establishment (AWE) and two at Lawrence Livermore National Laboratory (LLNL) in the United States. Also installed at LLNL is the Ray IBM cluster, which is an early access platform for the Sierra advanced technology system. Finally, two machines housed by universities have been used from the University of Warwick and Technische Universität Dresden in Germany.
Specifically, the machine configurations used are:

Titan

A capability system installed in the Oak Ridge Leadership Computing Facility (OLCF) at ORNL. Titan is a hybrid architecture Cray XK7 system consisting of 18,688 nodes, each housing a 16-core AMD Opteron processor and an NVidia

	Titan		ARCHER	
Processor	AMD Opteron 6274		Intel Xeon E5-2697v2	
CPU Speed	2.2 GHz		2.7 GHz	
Cores Per Node	16		24	
Memory Per Node	32 GB		64 GB	
Nodes	18,688		4,920	
Interconnect	Cray Gemini		Cray Aires	
File System				
I/O Servers		288		12
	Storage	Metadata	Storage	Metadata
Number of Disks	10,080	30	480	14
Disk Size	2 TB	900 GB	4 TB	600 GB
Spindle Speed	10,000 RPM	15,000 RPM	7,200 RPM	10,000 RPM
Bus Connection	NL-SAS	SAS	SAS	SAS
RAID Configuration	RAID 6 (8 + 2)	RAID 1 + 0	RAID 6 (8 + 2)	RAID 1 + 0

Table 3.1: Hardware specification of the Titan and ARCHER supercomputers.

K20X Kepler GPU with 14 streaming multiprocessors. Titan is serviced by two large centre-wide Lustre file systems, each housing 1008 OSTs, and nodes are connected via the Cray Gemini interconnect in a 3D torus topology. The full specification can be found in Table 3.1.

ARCHER

A capability system installed as the UK's national supercomputing service at EPCC. ARCHER is a Cray XC30 consisting of 4920 nodes, each containing two 12-core Intel Ivy Bridge processors connected via two QuickPath Interconnect (QPI) links. The Cray Aries interconnect is used to link the nodes together in a dragonfly topology, and the machine is serviced by three Lustre file systems. Two of these file systems contain 48 OSTs, with the third slightly larger housing 56 OSTs. A summary of the archer configuration is found in Table 3.1.

Spruce

A capacity system installed at AWE, partitioned into two halves that can operate independently. The partition used in this thesis was Spruce A, an SGI ICE X consisting of 2226 nodes containing two 10-core Intel Ivy Bridge EP processors running at 2.8 GHz. Spruce is serviced by a 140 OST Lustre file system, and the nodes are connected by FDR InfiniBand. A Summary of the Spruce

	Spruce A		Cab	
Processor	Intel Xeon E5-2680v2		Intel Xeon E5-2670	
CPU Speed	2.8 GHz		2.6 GHz	
Cores Per Node	20		16	
Memory Per Node	64 GB		32 GB	
Nodes	2,226		1,296	
Interconnect	InfiniBand FDR		InfiniBand QDR	
File System				
I/O Servers				32
	Storage	Metadata	Storage	Metadata
Number of Disks			4,800	30 (+2)
Disk Size			450 GB	147 GB
Spindle Speed			10,000 RPM	15,000 RPM
Bus Connection			SAS	SAS
RAID Configuration	RAID 6 (8 + 2)	RAID 1 + 0	RAID 6 (8 + 2)	RAID 1 + 0

Table 3.2: Hardware specification of the Spruce A and Cab supercomputers.

configuration is shown in Table 3.2.

Cab

A capacity system installed in the Open Compute Facility (OCF) at LLNL. Cab is a Cray built Xtreme-X cluster consisting of 1296 nodes, containing two 8-core Intel Sandy Bridge EP processors running at 2.6 GHz. Nodes are connected using an InfiniBand interconnect in a fat tree topology. Storage systems in the OCF are connected to compute platforms via a central storage network, allowing different Lustre installations to be accessed by different machines. In this thesis, the *lscratche* file system was predominantly used as one of the three Lustre file systems available to Cab. This particular file system contains 80 OSTs.

Quartz

The Quartz system (summarised in Table 3.3) exists to fill a similar role to Cab at LLNL, and indeed belongs to the generation of systems intended to take over from Cab. Quartz is a Penguin Linux cluster consisting of 2634 nodes, containing two 16-core Intel Broadwell CPUs running at a frequency between 2.1 GHz and 3.3 GHz. Nodes are connected via the Intel Omni-Path interconnect and a 36 OST Lustre file system (*lscratchh*) is mounted via the central site storage network.

	Quartz		Ray	
Processor	AMD Opteron 6274		IBM POWER8	
CPU Speed	2.2 GHz		2.6 GHz	
Cores Per Node	16		16	
Memory Per Node	32 GB		32 GB	
Nodes	18,688		1,296	
Interconnect	Cray Gemini		InfiniBand QDR	
File System				
I/O Servers		36		2
	Storage	Metadata	Storage	Metadata
Number of Disks	2,880	96	82	2
Disk Size	8 TB	800 GB	6 TB	400 GB
Spindle Speed	15,000 RPM	(SSD)	7,200 RPM	7,200 RPM
Bus Connection	SAS	SAS	NL-SAS	NL-SAS
RAID Configuration	RAIDZ2	RAID 1 + 0	RAID 6 (8 + 2)	RAID 1 + 0

Table 3.3: Hardware specification of the Quartz and Ray supercomputers.

Ray

Of the systems used in this thesis, the Ray early access cluster is the most distinct. Ray is installed at the OCF at LLNL, and is an IBM hybrid architecture system designed as a scaled down system for porting software and applications to the Sierra supercomputer. Ray consists of 62 nodes, each containing two 10-core IBM POWER8 processors and 4 NVidia Tesla P100 GPUs with 56 streaming multiprocessors each giving a total of 3584 GPU cores. A Mellanox EDR InfiniBand interconnect links the nodes and a reasonably limited size GPFS file system serves as the main parallel file system. Furthermore, Ray nodes each contain a 1.6 TB SSD burst buffer as a high bandwidth persistent storage layer. Ray is summarised in Table 3.3.

Taurus

A research system installed at the Centre for Information Services and High Performance Computing at Technische Universität Dresden. Taurus (summarised in Table 3.4) is a bullx DLC 720 system consisting of three tightly coupled islands each based on a different processing architecture. In this thesis, work has been performed using the phase two island of 1456 nodes, each containing two 12-core Intel Haswell processors. Nodes are connected in the system with FDR InfiniBand and storage is provided by a 96 OST Lustre parallel file system.

	Taurus		Tinis	
Processor	Intel Xeon E5-2680v3		Intel Xeon E5-2630 v3	
CPU Speed	2.5 GHz		2.4 GHz	
Cores Per Node	24		16	
Memory Per Node	64 GB		64 GB	
Nodes	1,456		203	
Interconnect	InfiniBand FDR		InfiniBand QDR	
File System				*GPFS*
I/O Servers		24		2
	Storage	Metadata	Storage	Metadata
Number of Disks	9,600	14	120	-
Disk Size	6 TB	900 GB	6 TB	-
Spindle Speed	10,000 RPM	10,000 RPM	7,200 RPM	-
Bus Connection	NL-SAS	SAS	NL-SAS	-
RAID Configuration	RAID 6 (8 + 2)	RAID 1 + 0	RAID 6 (8 + 2)	-

Table 3.4: Hardware specification of the Taurus and Tinis supercomputers.

Tinis

A research system installed at the Centre for Scientific Computing at the University of Warwick. Tinis consists of 203 Lenovo NeXtScale NX360 nodes, each housing two 8-core Intel Haswell processors. Nodes are connected with QLogic TrueScale 4x QDR InfiniBand and storage is provided by a small GPFS installation organised into six data disk pools.

3.5 I/O Benchmarking Applications

The work contained in this thesis focuses heavily on using software applications to benchmark I/O performance. For this purpose, benchmarks have been used throughout. Specifically these applications are:

MACSio

An application developed for I/O performance testing, as well as evaluation of trade-offs in data model interfaces and parallel I/O paradigms for multi-physics HPC applications. Two key design features of MACSio [63] set it apart from existing I/O proxy applications and benchmarking tools. The first is the level of abstraction at which MACSio operates and the second is the degree of flexibility MACSio provides in driving an HPC I/O workload through parametrised, user-

defined data objects and a variety of parallel I/O paradigms and I/O interfaces.

The MACSio proxy application is used particularly heavily throughout this thesis, and is the focus of the workload replication and optimisation presented in Chapter 6 and Chapter 7. More detail on MACSio will be discussed in these chapters.

Bookleaf

Bookleaf is a 2D unstructured Lagrangian hydrodynamics application, solving the Sod, Sedov, Saltzmann and Noh shock-hydro problems [89]. The application has a fixed checkpointing scheme that produces initial and final output files covering the complete dataset. All the I/O operations performed in Bookleaf are done through the TyphonIO library, writing to an underlying HDF5 file with an N-1 scheme.

3.6 Summary

Increasingly the hardware deployed by HPC sites is diverging as new architectures and approaches are required to forge the path to exascale. To handle the complexity of these architectures, advanced software and programming interfaces are being developed to maintain performance portability and user accessibility. With regards to HPC storage systems, the principles limiting performance have remained largely consistent across different sites due to the reliance on HDDs and well established DFS approaches. In this chapter the background and history of I/O and data storage in HPC has been presented, covering the base HDD storage device up to the widespread Lustre PFS. These components form the basis of the storage systems deployed in the HPC systems that are tested in later chapters, and understanding the limitations of the hardware and design of these components motivates a goal of this work in mitigating poor I/O pattern choice for the underlying storage system.

The parallel file systems described in this chapter share the common de-

sign philosophy of object storage. This architecture has proved successful for distributing storage to provide greater performance to large scale supercomputers, however even these sophisticated file systems remain a bottleneck to the heavy data requirements of scientific simulations. Adopting modern storage architectures and technologies, including things such as SSDs and NVMe devices in storage design, will alleviate some of the burden traditionally placed on the PFS if I/O approaches can effectively evolve to exploit these capabilities.

Finally, this chapter provides a comprehensive overview of supercomputer hardware configurations that are used throughout this thesis to analyse I/O performance. In addition, the applications that are used to test these systems are introduced here.

CHAPTER 4

Experimental Setup

In this chapter, a detailed run-down of the configurations and experimental procedures used throughout the remainder of this thesis is given. Experimental results for the configurations listed are not presented here, instead this is a point of reference for the experiments contained primarily in Chapters 5 to 7. For ease of navigation, the sections and subsections in this chapter have been arranged to reflect the location of the experiments in the respective chapters in which they are featured.

4.1 Common Methodology: I/O Measurement

All of the experimentation throughout this thesis is done with the purpose of measuring I/O related metrics, the most common being I/O times and bandwidths. In each experimental run, the target application was instrumented using the Darshan I/O profiling and characterisation library. Darshan can be statically linked or dynamically loaded at runtime, both of which were used based on the requirements of the MPI library installed, however the operation and data reported is the same for both usage methods. Darshan itself calculates and reports I/O timings based on the starting and ending timestamp for open, close, read and write operations. For verification of accuracy, all of the I/O timings used in these experiments were calculated using the timestamps for each operation. Darshan also records bytes that have been transferred by read and write operations, and these measurements were used in conjunction with the calculated timings to produce bandwidth or throughput figures. It is also worth nothing that throughout the results in this thesis the results that are quoted are measured in terms of binary multipliers, that is KB represents 1024 Bytes,

MB represents 1024 KB, and GB represents 1024 MB.

4.2 Chapter 5: Profiling Multi-physics I/O Workloads

The focus of the results collected in Chapter 5 is to extract representations of the I/O work performed by the targeted applications.

4.2.1 Bookleaf Mini-Application

A profile of the Bookleaf mini-application is presented in Figure 5.2. This profile was extracted from Darshan logs generated on the Archer platform. Bookleaf was configured to use the *noh_large* problem and to use the TyphonIO checkpointing sub-routines.

Setup Component	
Application	Bookleaf
Configuration	noh_large problem
Platform	Archer
Scale	1, 2, 4, 8, 16, 32, 64 Nodes
Lustre Stripe Count	4
Lustre Stripe Size	1 MB

Table 4.1: Bookleaf profile gathering setup

Ten repetitions at each node scale were collected and the best case in terms of overall runtime was selected for presentation.

4.2.2 AWE01 Multi-Physics Application

Five profiles of the AWE01 Application are presented in Section 5.1.3. Each of these profiles was extracted from Darshan logs generated on the Spruce A platform. The AWE01 application was configured to run five different problem input decks and to use TyphonIO checkpointing and visualisation writing.

Due to the extremely limited access to the machine to run these problems, the length of runtime required and the reasonable probability of a calculation

Setup Component	
Application	AWE01 Multi-Physica
Configuration	Problems A, B, C, D, and E
Platform	Spruce A
Scale	1 Node (Problem A) 5 Nodes (Problems B-E)
Lustre Stripe Count	-1
Lustre Stripe Size	1 MB

Table 4.2: AWE01 profile gathering setup

halting due to instability, only 5 repetitions of problems B-E were collected. For problem A the resource required was only a single node for a shorter runtime and so 10 repeats were possible. The best performing run in terms of overall runtime were selected for use in the relevant section.

4.3 Chapter 6: Application Workload Replication

The focus of the results in Chapter 6 is to validate the MACSio proxy application's ability to generate an I/O workload representative of Bookleaf and FLASH-IO.

4.3.1 Proxy Application Validation: Bookleaf

The experiments run in Section 6.2.1 used the Bookleaf mini-application with the noh_large problem input and the MACSio proxy application using a Bookleaf input parameter set. Two platforms were used to collect validation results for both of these applications, Archer and Tinis. All of the values presented in these results are taken from profiling logs collected via Darshan instrumentation and the values quoted verbatim.

On both systems, 10 repeats were collected at each scale and the best performing run in terms of I/O time is presented. Importantly, the best performing run at each scale was selected and all three measurements (I/O time, cumulative write time, and max write operation time) were taken from this run rather than the best result being cherry picked across different repeats for the three

Setup Component		
Application	Bookleaf	MACSio
Configuration	noh_large	Bookleaf profile
Platform	Archer	
Scale	1, 2, 4, 8 ,16, 32, 64 Nodes	
Lustre Stripe Count	-1	
Lustre Stripe Size	1 MB	

Table 4.3: MACSio-Bookleaf Validation on Archer

Setup Component		
Application	Bookleaf	MACSio
Configuration	noh_large	Bookleaf profile
Platform	Tinis	
Scale	1, 2, 4, 8 ,16, 32, 64 Nodes	
Lustre Stripe Count	N/A (GPFS)	
Lustre Stripe Size	N/A (GPFS)	

Table 4.4: MACSio-Bookleaf Validation on Tinis

measures. I/O timings are presented separately for the two checkpoints that are generated by both applications, the run with the best time across both of these checkpoints was selected at each node scale.

4.3.2 Proxy Application Validation: FLASH-IO

The experiments run in Section 6.2.2 used the FLASH-IO application and the MACSio proxy application using a derived FLASH input parameter set. Results for FLASH-IO and MACSio were gathered from the Archer platform and values were obtained directly from Darshan profiling logs.

Setup Component		
Application	FLASH-IO	MACSio
Configuration	3D Large	FLASH-IO Profile
Platform	Archer	
Scale	1, 2, 4, 8 ,16, 32, 64, 128 Nodes	
Lustre Stripe Count	-1	
Lustre Stripe Size	1 MB	

Table 4.5: MACSio-FLASH Validation on Archer

The results presented in Figure 6.5 represent the best case in terms of I/O

time across 15 repetitions. At each scale the best performing run was selected and all three measurements (I/O time, cumulative write time, and max write operation time) were taken from this run rather than the best result being cherry picked across different repeats for the three measures.

4.4 Chapter 7: I/O Performance Benchmarking and Optimisation

The results presented in Chapter 7 cover a wide spectrum of experiments comprising a varied performance study of I/O software and parallel file system tuning. These experiments were conducted using a wide range of platforms across a period of 42 months.

4.4.1 Tuning the Parallel I/O Software Stack: Middleware

Collective Operation Scaling

The experiments shown in Figures 7.1 and 7.2 used the MACSio proxy application to generate replicated Bookleaf, FLASH-IO, and AWE01 workloads on the Archer, Quartz, and Ray platforms. For each of the applications tested two sets of runs were performed, one with the I/O library enforcing independent I/O operations and one with the library attempting to use collective operations. Additionally, two sizes of workload were tested for Bookleaf and FLASH-IO. Bookleaf used a small 1800×720 and larger 14400×5760 problem size and FLASH-IO used the standard reference 3D problem and a large reference problem twice the size of the original.

For each machine, application profile, and problem size 10 repeats of the run were collected. The results presented represent the best case performance at each scale in terms of the observed I/O checkpoint bandwidth.

Setup Component	
Application	MACSIO
	Bookleaf Profile
Configuration	FLASH-IO Profile
	AWE01 Problem A, B, D
Platform	Archer
	1, 2, 4, 8 ,16, 32, 64, 128 Nodes (Bookleaf, FLASH-IO)
Scale	1 Node (AWE01 Problem A)
	5 Nodes (AWE01 Problem B, D)
Lustre Stripe Count	-1
Lustre Stripe Size	1 MB

Table 4.6: Collective Operation Scaling on Archer

Setup Component	
Application	MACSIO
	Bookleaf Profile
Configuration	FLASH-IO Profile
	AWE01 Problem A, B, D
Platform	Quartz
	1, 2, 4, 8 ,16, 32, 64, 128 Nodes (Bookleaf, FLASH-IO)
Scale	1 Node (AWE01 Problem A)
	5 Nodes (AWE01 Problem B, D)
Lustre Stripe Count	-1
Lustre Stripe Size	1 MB

Table 4.7: Collective Operation Scaling on Quartz

Setup Component	
Application	MACSIO
	Bookleaf Profile
Configuration	FLASH-IO Profile
	AWE01 Problem A, B, D
Platform	Ray
	1, 2, 4, 8 ,16, 32 Nodes (Bookleaf, FLASH-IO)
Scale	1 Node (AWE01 Problem A)
	5 Nodes (AWE01 Problem B, D)
Lustre Stripe Count	N/A (GPFS)
Lustre Stripe Size	N/A (GPFS)

Table 4.8: Collective Operation Scaling on Ray

Collective Buffering Parameters

Figures 7.3 and 7.4 show experiments to study the impact of changes to the collective buffering control parameters, specifically **cb_nodes** and **cb_buffer_size**.

The Bookleaf and FLASH-IO replicated workloads were run in MACSio on the Archer platform to produce these results. A set of seven parameter values are tested for the **cb_nodes** setting and a set of five used for **cb_buffer_size**.

cb_nodes	**cb_buffer_size**
1	16MB
2	32MB
4	64MB
8	128MB
16	256MB
32	
64	

Table 4.9: cb_nodes and cb_buffer_size parameter settings tested

Setup Component	
Application	MACSIO
Configuration	Bookleaf Profile
	FLASH-IO Profile
Platform	Archer
Scale	1, 2, 4, 8 ,16, 32, 64, 128 Nodes
Lustre Stripe Count	-1
Lustre Stripe Size	1 MB

Table 4.10: Collective Buffering Parameter Performance on Archer

Due to the number of combinations of parameters that were tested, these experiments were carried out with only 8 repetitions of each configuration. The results selected for presentation represent the best case performance at each scale in terms of the observed I/O checkpoint bandwidth.

4.4.2 Parallel File System Performance

Figures 7.5 and 7.6 present experimental results showing the file bandwidth achieved on a host of platform under different Lustre striping conditions. These experiments were run using the Spruce A, Titan, Archer, Cab, Taurus, and Tinis machines. The application used to gather these results was the MACSio proxy application running the replicated problem workloads of AWE01 problems A, B, and D.

Setup Component						
Application	MACSIO					
Configuration	AWE Problem A,B,D					
Scale	1 Node (Problem A) 5 Nodes (Problem B, D)					
Platform	Spruce	Titan	Archer	Cab	Taurus	Tinis
Lustre Stripe Count (Default)	8	4	4	1	96	N/A (GPFS)
Lustre Stripe Count (Increased/Striped)	N/A	80	48	80	N/A	N/A (GPFS)
Lustre Stripe Size	1 MB	1 MB	1 MB	1 MB	1 MB	1 MB

Table 4.11: Parallel file system striping performance

The number of repeats that were gather for these experiments was 10 for all platforms apart from Titan due to limited access to the machine and the lead-time of having jobs accepted and scheduled, hence only 6 repeats were possible. From the runs collected the best performing repeat was select with respect to the file bandwidth achieved on each platform under the two different Lustre configurations.

4.4.3 I/O Library and File Strategy Comparisons

TyphonIO Efficiency

Figures 7.8 and 7.9 demonstrate experiments to measure the performance variation of the MACSio AWE01 workloads when generating different file layouts on disk through two high level libraries. The selection of the different libraries and file schemes was achieved by setting the interface and layout parameters in MACSio. These experiments were run using the Archer and Quartz systems and each AWE01 problem workload was run at 1,2, and 4 times its standard node count.

Each data point selected for presentation represents the best achieved bandwidth for each platform, problem, scale, and file scheme. To collect these results, 10 repeats were used for each configuration on both machines.

Setup Component		
Application	MACSIO	
Configuration	AWE Problem A,B,D	
Library File Schemes	TyphonIO - Contiguous TyphonIO - Chunked HDF5 - Logically Contiguous HDF5 - Block Contiguous	
Scale	1, 2, 4 Nodes (Problem A) 5, 10, 20 Nodes (Problem B, D)	
Platform	Archer	Quartz
Lustre Stripe Count (Default)	48	80
Lustre Stripe Size	1 MB	1 MB

Table 4.12: I/O library and file strategy comparison experiments

N-M Parallel File Modes

Figures 7.10 to 7.13 demonstrate experiments to do a side-by-side comparison of classic single shared file, sequential N-M, and parallel N-M file modes. All of these experiments were conduced using the MACSio AWE01 Problem D replicated workload. The Quartz, Archer, Cab, and Ray platforms were used to collect these results at three different node scales. At each scale, an N-M sequential and N-M parallel run was taken with the files per node count varied from 1 to 16. In the case of the experiments run on Ray, these consisted of two versions of each configuration, one run with I/O to the parallel file system and one writing to node-local burst buffers. Despite the large number of combinations, each data point represents the best of 10 repeats in terms of the observed file bandwidth.

4.5 Summary

This chapter contains a reference of the experimental configurations used throughout the rest of these thesis. Details on the application, platform and experimental setting are presented here to aid in the interpretation of experimental results later on.

Setup Component				
Application	MACSIO			
Configuration	AWE Problem B			
File Modes	Single Shared File N-M Sequential N-M Parallel			
Platform	Quartz	Archer	Cab	Ray
Scale (Nodes)	5, 10, 20	5, 10, 20	8, 16, 32	4, 8, 16
Files per Node	1, 2, 4	1, 2, 4	1, 2, 4, 8	1, 2, 4, 8, 16
Lustre Stripe Count	80	48	80	N/A (GPFS+Burst Buffers)
Lustre Stripe Size	1 MB	1 MB	1 MB	N/A

Table 4.13: Parallel file mode comparison experiments

CHAPTER 5

Profiling Multi-physics I/O Workloads

Modern storage systems are being placed under increasingly heavy loads as supercomputers break the exascale barrier [52]. In particular, the trend of moving away from conventional CPU only architectures towards a hybrid of both CPU and accelerator components is increasing the density of computation power that can be packed in to a node; consequently, more data can be processed and generated by these nodes for the purpose of results gathering, resilience, and visualisation.

To properly study I/O performance in an application focused way, it is critically important to be able to generate I/O traffic in a system that closely mirrors that of the target application. Unfortunately, performance benchmarking and engineering work is difficult to perform in an agile way when constrained by large, fully formed scientific applications. Benchmark applications are typically used as a stand in for the purpose of testing out system I/O performance and attempting to uncover the inefficiencies in I/O libraries, but these lack resemblance to more complex and nuanced scientific codes and present a limited picture as to the true implications for the original application. Particular consideration is made to the IOR benchmark, which has become the de-facto tool for demonstrating I/O performance limits on a system. A fundamental drawback with IOR is that it is designed to marshal dummy blocks of data in a heavily simplified pattern that you would be unlikely to find in any real scientific application. Moreover, higher level scientific concepts can also influence the way that data is moved around a simulation or indeed committed to file as ultimately the data must be easily interpretable by a user.

It is hypothesised that the representative profiles of genuine scientific sim-

ulation codes is of great value for the goals of this thesis, to demonstrate and utilise an I/O proxy application stand-in to enable much more targeted investigations of current and future I/O hardware and practices. To achieve this goal, the work in this chapter demonstrates the use of standard application profiling techniques to collect data on two previously unstudied scientific applications. Profile descriptions are created for the I/O workloads of these novel applications and analysis is presented to outline the regularities and irregularities present in their pattern of execution. These profiles will be utilised in later chapters to verify and demonstrate the proxy application specific objectives of this work.

5.1 Application Pattern Identification

The execution pattern of many simulations has traditionally been made up of a number of distinct phases that occur in order. Dependent on the parallel programming model used and often the characteristics of the algorithm itself, the phases that can be identified in an application may vary drastically.

In a message passing based simulation, a portion of a calculation and its associated data is distributed across a number of nodes. The process that is usually followed is a setup phase with parameters and initial data being read from storage and the controlling elements of the run being determined. What follows is a repeating cycle of individual calculations by each processor and a communication of some portion of the resulting data to neighbouring processors or to a nominated controlling processor. After this communication phase, the simulation is in a coordinated state and a decision can be taken to repeat another sequence of the calculation, terminate the simulation, or perform some auxiliary function. This phase is where the greatest proportion of I/O activity is likely to take place. Logically the point of synchronisation marks a sensible stage at which to perform checkpointing or the preservation of results for visualisation purposes. A simple high-level execution pattern exhibited by simulations is shown in Figure 5.1.

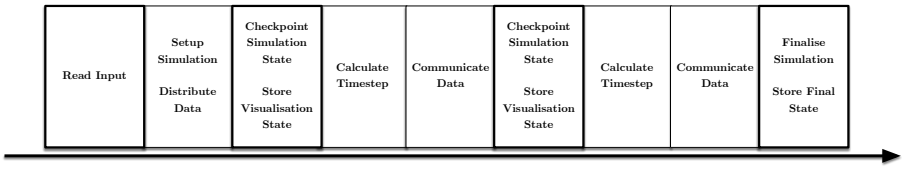


Figure 5.1: An example of the general pattern of simulation phases found in scientific applications.

Threading based parallel programming models, those favoured by GPU-like hardware, discretise units of work and spread them out across a number of available threads. Upon completion, a point of synchronisation is usually required to ensure the data is in a known consistent state before progressing on to the next phase of the calculation. Similar to how communication is used to coordinate process in the previous example, this point of synchronisation offers a logically desirable phase in which to perform coordinated I/O.

In both of the described simulation patterns there is a clear distinction made between the I/O phase and the phases responsible for calculation and data updates. As a result, the I/O pattern exhibited by an application will resemble bursts of concentrated activity at intervals that can be predictably regular or vary throughout execution. Bursty I/O is a commonly observed pattern in HPC applications, however this behaviour is not conducive to efficient use of parallel file systems. Approaches that attempt to mitigate the obvious bottleneck of concentrated I/O load generally approach the problem in one of two ways. The first approach would be to somehow stage the data in a high performance buffer comprising faster storage hardware. Burst buffer systems adopt this approach through the use of very fast node-local storage in the form of high performance flash based storage. In this instance the mechanism in place to alleviate the application becoming blocked by I/O is to stage the data in the local storage, either temporarily or permanently, with much greater speed than can be achieved from the main parallel file system. The decrease in solid state drive technology has facilitated an increase in uptake of burst buffer systems, and one of the elements of the performance study carried out in Chapter 7 seeks to

quantify the potential benefits of this approach.

An alternative to burst buffers for handling bursty I/O patterns is the use of asynchronous I/O, as reviewed in Chapter 2. This approach looks to amortise the I/O cost of transferring data to the parallel file system over subsequent computational steps and does not cause the application to block its progression. Equipping an application to successfully perform asynchronous I/O operations relies on support from the underlying MPIIO implementation available on the system, and while masking the issue from a user's perspective, does not ultimately reduce the stress on the parallel file system. For the purpose of the work carried out in this thesis, the burst buffer approach is evaluated and asynchronous I/O considered as a valuable avenue for future research.

A complete replication of an I/O profile will incorporate the frequency and predictability of activity bursts, in conjunction with more fine grained detail regarding the composition of a file and the operations issued to move data to and from the file. We have constructed profiles for two scientific applications based on coarse grained data collection for the purpose of further I/O analysis and performance engineering.

5.1.1 Runtime Profiling

Modelling the I/O patterns displayed by an application requires user knowledge and profiling data collected from a representative simulation run. The profiles that are described in Sections 5.1.2 and 5.1.3 are derived from logs generated by the Darshan I/O characterisation library [16].

Profiling methods that are capable of collecting data on each I/O specific operation rely on the interception and logging of function calls and associated data. Commonly this process is carried out in the form of function tracing where a record of each function call is stored along with associated parameters, timestamps and calling contexts. In a parallel simulation the volume of data that can be generated becomes prohibitively large, requiring profiling counters to be aggregated to avoid interfering with the footprint of the simulation itself. To

avoid overhead issues associated with this granularity of data storage, Darshan uses a lightweight process for logging I/O events from an application. A set of data counters (listed in Table A.1) are used to record a characterisation of the operations that are being performed, eliminating the need for each function invocation to be stored. The operations targeted here are primarily the open, close, read, and write based operations but these also include some of the lower level functions used in MPIIO to perform these actions. The minimal data collection overhead makes this technique a viable approach for the profiling of production applications with the potential to issue thousands of I/O specific function calls over the course of a run lasting multiple days. The lightweight nature of Darshan profiling has been demonstrated against production scale I/O benchmarks, with the most intensive application shutdown operation introducing less than 4.6 seconds of fixed overhead [17, 81].

Records generated by Darshan characterisation are produced in the format demonstrated in Listing 5.1. Function calls are intercepted by one of the POSIX, MPIIO, HDF5, or Lustre modules and if these correspond to any of the record counters then the counter is updated accordingly. For example, a simple MPIIO_File_write function call would be intercepted by the MPIIO module and this would cause values for the MPIIO_F_WRITE_TIME, MPIIO_INDEP_WRITES, and MPIIO_BYTES_WRITTEN counters to be updated among others. The counters returned at the end of the run reflect the operations captured at each of these levels. Furthermore, the correlation of records for a file generated at different levels in the I/O stack present the opportunity for characteristics of the workload introduced by I/O libraries to be derived.

The high level execution pattern describing points in the simulation where an I/O phase occurs can be constructed from the timestamp counters associated with each file. A limitation of Darshan's approach of not recording timestamps for every function call is that file reuse obscures the write access pattern. This affects situations such as the appending of visualisation states to a single file throughout a simulation. In this instance, we are able to determine an access

```
<rank>  <counter>                          <value>     <file name>
-1      MPIIO_F_OPEN_TIMESTAMP             0.448894    checkpoint.01.h5
-1      MPIIO_F_WRITE_START_TIMESTAMP      0.594944    checkpoint.01.h5
-1      MPIIO_F_WRITE_END_TIMESTAMP        1.428066    checkpoint.01.h5
-1      MPIIO_F_CLOSE_TIMESTAMP            1.440608    checkpoint.01.h5
-1      MPIIO_F_WRITE_TIME                 12.113718   checkpoint.01.h5
-1      MPIIO_F_META_TIME                  0.662519    checkpoint.01.h5
-1      MPIIO_INDEP_WRITES                 245         checkpoint.01.h5
-1      MPIIO_COLL_WRITES                  800         checkpoint.01.h5
-1      MPIIO_BYTES_WRITTEN                426989900   checkpoint.01.h5
...
```

Listing 5.1: Decompressed Darshan log demonstrating the record format produced by an instrumented application. The output file is shared between processors (ranks), which is indicated by the rank field containing a value of −1. Records additionally contain a module, record id, mount point, and file system field which are omitted for brevity.

pattern through correlation of open operation counters on files with repeated access and those containing a single checkpoint state. Analysis of the darshan profile logs collected shows that for single use files (i.e. opened and closed only once during the simulation) all processes will issue a collective open function call only once, consequently the MPIIO_COLL_OPENS counter will match the processor count. The same sequence of operations recorded by the POSIX module returns a POSIX_OPENS counter with a number one greater than the processor count due to the creation of the file by the first rank in the communicator adding an extra operation to the counter. This is demonstrated by the log extract shown in Listing 5.2. Recognising the relationship between the processor count and the file open counters makes it possible to determine the number of times visualisation files undergo an open-write-close cycle without requiring the files itself to be inspected for each run. For the first of the AWE Multi-Physics application problems the visualisation file viz.h5 records a MPIIO_F_COLL_OPENS value of 800, which when divided by the number of processors used gives 50 visualisation states. Manual verification of the file contents was able to confirm that this problem indeed generated 50 visualisation states and hence 50 open-write-close phases were generated for visualisation purposes. This counters extracted

```
<rank>  <counter>                 <value>   <file name>
-1      MPIIO_F_COLL_OPENS        16        checkpoint.01.h5
...
-1      POSIX_OPENS               17        checkpoint.01.h5
...
-1      MPIIO_F_COLL_OPENS        800       viz.h5
...
-1      POSIX_OPENS               801       viz.h5
...
```

Listing 5.2: Decompressed Darshan log extract showing the observed relationship between the MPIIO_F_COLL_OPENS and POSIX_OPENS counters for a checkpoint file that is opened and closed once by all processes and a visualisation file that is opened and closed 50 times.

for the verification are shown in Listing 5.2 and the relationship between processes, open count, and I/O phases is summarised in Table 5.1 for the AWE Multi-Physics application.

Simulation	Process Count	MPIIO Opens	Calculated I/O Phases	Actual States
A	16	800	50	50
B	80	2320	29	29
C	80	2320	29	29
D	80	9280	116	116
E	80	7040	88	88

Table 5.1: Summary of AWE Multi-Physics application profile data showing the calculated number of distinct I/O phases in a simulation where an open-write-close cycle operates on a visualisation file and the number of states actually observed in the file.

Recording and deriving timestamps for the I/O phases in a simulation allows us to construct a timeline of the components shown in Figure 5.1. Factoring in the information collected about the volume of data that is being transferred to the file system, and how this data is laid out in the application, a basic I/O pattern can be defined.

The POSIX_BYTES_WRITTEN record represents the total data transferred through calls to the POSIX library, specifically the write() function. Both the MPI-IO interface and the parallel file systems investigated in this thesis are POSIX compliant and therefore any I/O function calls made to any of the avail-

able libraries will consequently be translated to the POSIX level and captured in this record.

5.1.2 Bookleaf Mini-Application

Bookleaf is a 2D unstructured hydrodynamics mini-application for approximating the solutions to a collection of different physics problems. At the time of writing, input configurations for Bookleaf are available to solve the Sod shock tube problem, Sedov blast wave test, Saltzman planar shock problem, and Noh gas impact problem. Moreover, the application is capable of producing solutions to these problems using both Lagrangian and Eulerian methods.

	Nodes						
	1	2	4	8	16	32	64
Posix							
File Opens	25	49	97	193	385	769	1537
File Seeks	496	934	1810	3538	6994	13,906	27,730
File Writes	524	956	1820	3548	7004	13,916	27,740
Sequential Writes	457	859	1664	3238	6560	13,088	26,144
(Proportion of Total)	(87.2%)	(89.9%)	(91.4%)	(91.3%)	(93.7%)	(94.0%)	(94.2%)
Consecutive Writes	30	24	12	12	12	12	12
(Proportion of Total)	(6.6%)	(2.5%)	(0.7%)	(0.3%)	(0.2%)	(0.1%)	(<0.1%)
MPI-IO							
File Opens	24	48	96	192	384	768	1536
Collective Writes	0	0	0	0	0	0	0
Independent Writes	524	956	1820	3548	7004	13,916	27,740
File Syncs	24	48	96	192	384	768	1536
File Views	0	0	0	0	0	0	0

Table 5.2: Checkpoint statistics for Bookleaf checkpoints at scales between 1 and 64 nodes collected on Archer with default stripe count of 4.

The runtime characteristics displayed by Bookleaf can vary based on the problem definition given, but in terms of the general simulation pattern are largely fixed for any calculation performed. As a consequence the phases of I/O in a Bookleaf run occur predictably; the composition of the I/O activity depending on the scale of the simulation and problem itself.

Checkpointing behaviour is implemented at the beginning and end stages of the simulation, primarily for the purpose of verifying calculation correctness and visualisation. The execution pattern of Bookleaf is illustrated in Figure 5.2, showing the footprint and chronology of the important phases in the simulation.

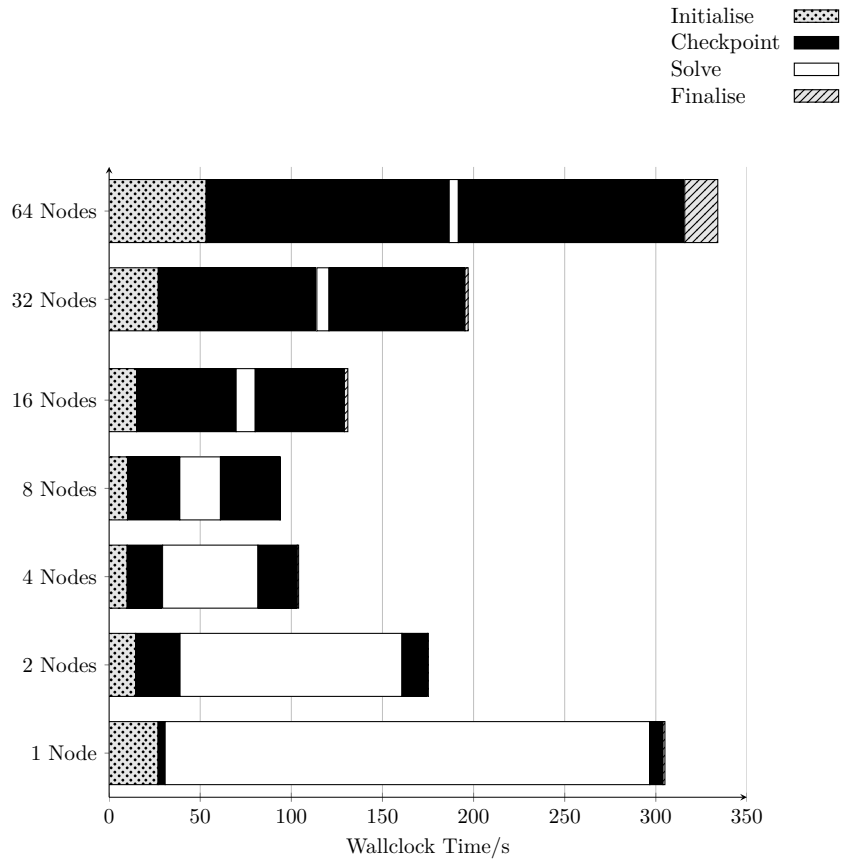


Figure 5.2: Execution pattern for Bookleaf simulating the noh_large problem at 1, 2, 4, 8, 16, 32, and 64 Nodes. These patterns of execution were collected on Archer with a default stripe count of 4.

What is interesting to observe from these profiles of the Bookleaf application is that the I/O overhead for the checkpointing phases does not scale with the node count as seen with the solve phase, or indeed remain constant. Instead the I/O overhead increases to dominate the overall execution time and deliver greatly disappointing performance. To illustrate this point, the checkpoint bandwidth for the single node profile is around 35 MiB/s with this dropping to just 1 MiB/s at 64 nodes. Considering serial I/O speeds are expected to be tens or hundreds of megabytes this is generally disappointing and a world away from what applications should be aiming for to make use of the current generation of I/O hardware as well as future innovations to support exascale problems.

A selection of the statistics extracted from Bookleaf are shown in Table 5.2. These statistics are taken from Darshan counter values reported in the logs produced by Darshan instrumented runs. Additional characteristics of the I/O being performed by Bookleaf can be observed in some of these statistics. First and foremost, the profiling identifies that none of the I/O operations used in writing a checkpoint are performed collectively. The number of writes to consecutive offsets is very small, both the actual count and proportion of the total writes issued. The proportion of writes to increasing offsets in the file is much larger, and increases as the application scales. These statistics indicate that on the creation of a checkpoint, all ranks are independently sweeping through the file and issuing smaller interleaved writes rather than writing multiple contiguous blocks.

5.1.3 AWE01 Multi-Physics Application

The AWE01 multi-physics application is a 2D production code primarily built around the simulation of hydrodynamics problems. The multi-physics capability in the application is provided by a suite of different physics packages that can be activated to simulate additional phenomenon within the core hydrodynamics problem. Due to the number of packages that are available in AWE01, both the execution pattern and the dataset composition can vary drastically between different simulation types and within a particular run.

There are five simulation configurations used for the work carried out in this thesis, which represent some important workloads generated by application users. These problem configurations were provided by a computational physicist recognised as a domain expert in the simulations performed by the multi-physics application. These inputs are labelled A to E and present an overview of their characteristics. In all of the execution patterns profiled both checkpoint/restart and visualisation data is written to the file system. In the case of checkpointing, a new file is created for each checkpoint phase whereas visualisation data is appended to a single file for the duration of the simulation.

Each of these application profiles has been collected from runs taken from the Spruce A platform as described in Section 4.2.2.

Simulation A

The first simulation type is the most simple to characterise in terms of its I/O behaviour, representing a baseline calculation run on a single compute node. Over the course of the 119 minute runtime, 48 checkpoints are written with 213 MB of data stored in each. Additionally there are 50 states written to a visualization file giving a total visualization output of 1.75 GB. This simulation represents a reference test problem and so is typically run on a single node, the run profile is shown in Figure 5.3. It is important to note that each checkpoint that occurs is represented by a thin black line rather than a solid block due to the required timeline scale. Similarly, the finalise region occurs at the end of the final Solve phase but is not clearly observable due to its relatively short time period.

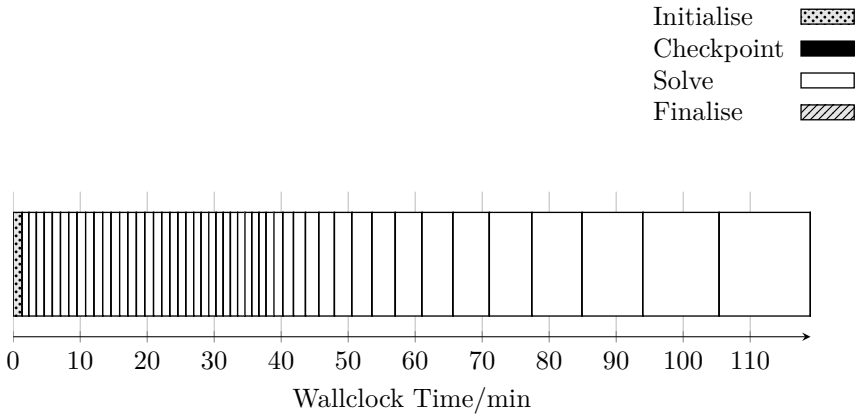


Figure 5.3: Execution pattern for AWE01 Simulation A

Simulation B

The second simulation (Figure 5.4) has the additional characteristic that the composition of the dataset used for checkpointing and visualization is varied

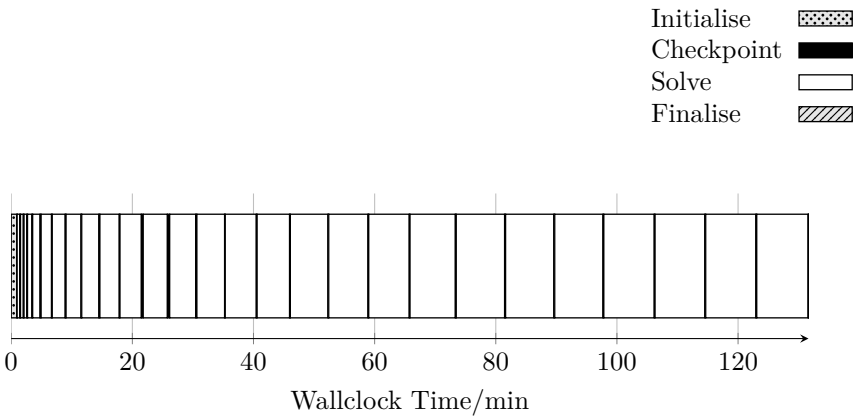


Figure 5.4: Execution pattern for AWE01 Simulation B

through the course of execution. Similarly to 5.3, individual checkpoint phases are represented by the black lines and the finalise region is too small to show up at this scale. The simulation runs on 5 compute nodes and a runtime of 131 minutes spans 27 checkpoints and 28 visualization states.

The dataset increases from 259 MB at the beginning of the simulation and reaches a total size of 1.9 GB, this pattern is shown in Figure 5.5. The total size of visualization data written reaches 285 MB.

It can be seen that there is a non-linear increase in the total checkpoint output, with a much greater rate of increase around the fifth checkpoint, which corresponds to the activation of additional simulation components.

From the perspective of understanding the I/O workload requirements of the simulation, it could be valuable to exploit knowledge of when this change will occur to influence a variation in I/O strategy such as modifying checkpoint frequency.

Simulation C

Simulation C performs the same experiment as Simulation B, but with an important additional physics package added. The I/O pattern of this simulation, including the observed dataset growth, matches Simulation B but a much larger

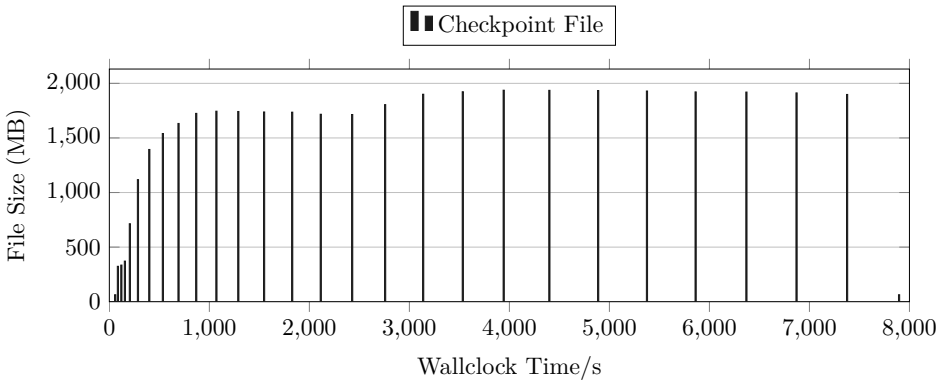


Figure 5.5: Dataset Growth of Input B

volume of data is required to visualize the additional data elements. As a result the total size of visualization data reaches 20.2 GB.

Simulation D

Simulation D, shown in Figure 5.6, represents a larger calculation typically run over multiple days. As with previous simulation patterns, the length of the runtime means checkpoint phases are represented by the vertical black lines. In this figure the initialise region at the beginning of the timeline and finalise region at the end are too small to show up at this scale.As with simulation problems B and C this run is performed with 5 compute nodes. Checkpointing is performed at a small number of key stages in the simulation, with 7.3 GB of data committed to file before and after a particularly important and unstable phase of the simulation, as well as the final state being saved upon finalisation. The visualization file data totals 1.8 GB, and is stored in 116 separate state outputs, giving a much higher temporal granularity to the results for visualisation and focusing less on checkpoint restart.

Simulation E

The final problem considered here performs a similar experiment to Simulation D. Similarly, three checkpoints of 7.7 GB are written at irregular intervals, while 1.4 GB of visualization data is produced across 88 simulation states.

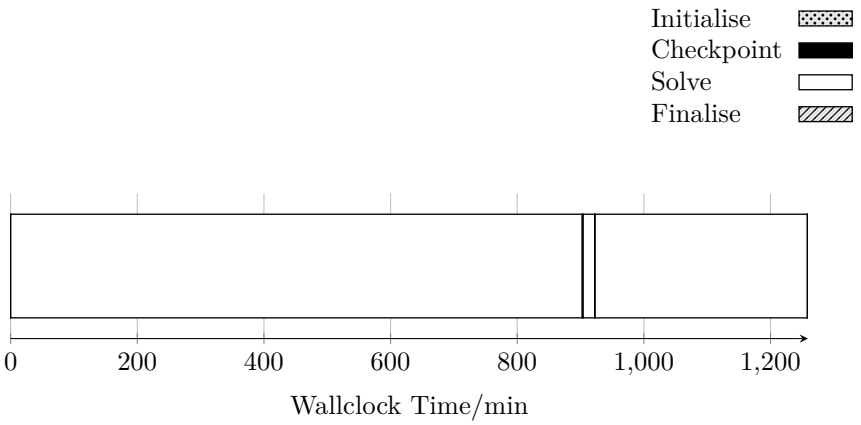


Figure 5.6: Execution pattern for AWE01 Simulation D

5.1.4 Multi-Physics Checkpoint Analysis

The statistics generated for the AWE01 workloads are summarised in Table 5.3. Unlike the Bookleaf counters shown earlier on in this chapter, checkpoint files here are written using collective operations. This leads to a greater proportion of the dataset being written to consecutive blocks in the file as aggregation is able to reconstruct the decomposed dataset before issuing a write call. Considering the number of collective writes issued, it is possible to approximate the number of different datasets stored in a file. In problem A a total of 864 collective calls were issued, which is the equivalent of 54 calls in the HDF5 layer being translated to MPI_File_write_at_all calls in the middleware layer. On inspection of the file, the total number of distinct variable arrays is identified at around 49, which when factoring in additional mesh related data comes out to be 54 total dataset objects.

Out of the five workloads analysed, there are broadly three different classes identified. In particular, the patterns established in problems B and C, as well as problems D and E are largely similar in their profiles and dataset compositions. For this reason, the B and D problems are selected as representative of their workload class, and with the addition of problem A will make up the set of three application profiles for AWE01 used through the remainder of this thesis.

	AWE01 Multi-Physics Application				
	Problem A	Problem B	Problem C	Problem D	Problem E
POSIX					
File Opens	17	325	485	241	401
File Seeks	1377	4482	6596	10,016	13,581
File Writes	3869	6870	7008	23,000	27,708
Sequential Writes	3687	6582	6705	21,477	26,262
(Proportion of Total)	(95.3%)	(95.8%)	(95.7%)	(93.4%)	(94.8%)
Consecutive Writes	2515	4347	4432	13,554	17,463
(Proportion of Total)	(65.0%)	(63.3%)	(63.2%)	(58.9%)	(63.0%)
MPI-IO					
File Opens	16	164	244	160	240
Collective Writes	864	1316	1316	25,360	25,360
Independent Writes	3815	6531	6669	21,738	26,446
File Syncs	32	168	328	320	480
File Views	1728	3912	6312	50880	65,280

Table 5.3: Checkpoint statistics for each problem class run by the AWE01 multi-physics application.

5.2 Summary

In order to study I/O performance of HPC system in a meaningful way, performance engineers must be able to stress the I/O subsystems in ways that are as representative of real production applications in use by experts and domain scientists. The first step to perform targeted I/O engineering work is to capture the profile of I/O workloads with regards to the size, composition and frequency of I/O phases.

In this chapter the application of the Darshan lightweight tracing library to capture I/O operations and translate these logs to high-level workload profiles has been presented. The profiling approach used by Darshan captures a series of I/O based counters rather than storing fine-grained trace records, making it ideal for near transparent data collection in production environments. This work focuses on profile data collected for a production multi-physics application and demonstrates a case study of how the I/O work generated by an application can vary dramatically depending on the simulation configuration. The configurations used in this work are designed by a domain expert in computational physics and hence provide a high degree of certainty as to their representativeness for production I/O workloads.

The construction of the profiles in this chapter, in particular A, B, and D, are intended to be deployed as representative workloads for the purpose of benchmarking and procuring systems with greater accuracy than is currently possible.

CHAPTER 6

Application Workload Replication

As HPC storage systems become more complex to cope with the demands of exascale computing, new I/O strategies and software libraries are being incorporated into applications. Traditional large scale spinning disk parallel file systems (PFS), such as Lustre and GPFS, are being supplemented by additional storage tiers made up of much faster SSD and NVMe devices. These so called *burst buffers* can be incorporated into storage systems in vastly different ways and hence application behaviour must be tailored for a system to obtain best performance.

The lack of flexibility of applications to support performance engineering activities for data storage creates an obstacle to exploiting maximum performance from new storage systems. I/O benchmarking addresses this inflexibility, but achieving accuracy and representativeness in benchmark workloads can be difficult due to the variation in data models and libraries used by applications.

This chapter documents the development and operation of the Multi-Purpose Application Centric Scalable I/O proxy application (referred to throughout the rest of this chapter by the acronym MACSio), described in [31, 32] to replicate the I/O behaviour of three scientific applications. MACSio is a parametrised proxy application for generating datasets to drive one of a number of high level library plugins using the I/O paradigms explored in detail in Chapter 2. The ability of MACSio to replicate the behaviours of real scientific applications allows portable benchmarking and exploration of alternative I/O libraries and paradigms.

The remainder of this chapter is structured as follows: Section 6.1 introduces the MACSio proxy application and describes the developments made to both

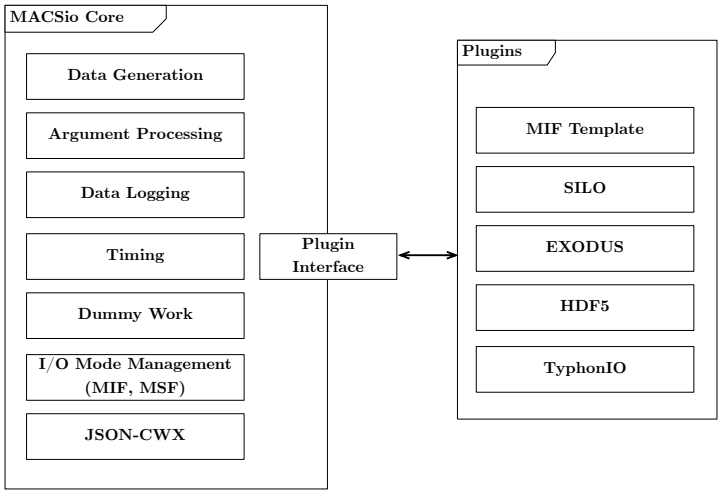


Figure 6.1: MACSio application components

the core and plugin components; Section 6.2 presents a validation of MACSio's ability to replicate the I/O workloads of two physics applications; finally, Section 6.3 summarises this research.

6.1 The MACSio Proxy Application

MACSio is a proxy application for generating I/O activity representative of that seen in scientific simulations. As a proxy application, MACSio is intended to act as a flexible and portable stand-in for the data generation and transfer portions of a fully fledged science code.

6.1.1 Application Overview

The design principle that affords MACSio the flexibility to investigate a multitude of I/O libraries is the use of a plugin based structure. Core application logic is contained in an upper-level driver, with control being passed on to one of a number of plugins at the point an I/O phase is entered. The general structure of the application is shown in Figure 6.1.

Generation of data for the purpose of performing I/O is controlled by a

Feature	Values
Mesh Dimensions	1D, 2D, 3D
Mesh Structure	Structured (rectilinear), Structured (curvilinear), Unstructured (unstructured cell data zoo), Unstructured (arbitrary shape)
Mesh Part Size (Bytes)	Integer values greater than 1
Mesh Part Decomposition	Integer value for each required mesh dimension
Distributed Parts Per Rank	Integer value greater than one and less than the total number of ranks
Mesh Part Dimensions	Integer value for each mesh dimension
Variables Per Mesh Part	
Rectilinear	Greater than 1
Curvilinear	Greater than the number of spatial dimensions
Unstructured	Number of spatial dimensions + $2^{\text{Number of topological dimensions}}$
Data Generation Algorithm	Constant value, Increasing value, Random / chaotic value, Sinusoidal, Spherical

Table 6.1: Configurable parameters in generated MACSio datasets

number of routines that produce distributed multi-dimensional meshes and arrays. The datasets that can be produced by MACSio are variable to approximate a broad spectrum of scientific applications in both dataset structure and magnitude. In particular, the variable features in a dataset are summarised in Table 6.1.

In the application start-up phase, workload parameters are read from the command line and missing values can be filled by either computing a sensible value from the set of defined parameters or reverting to known default values. For example, when creating and populating a mesh dataset the mesh part dimensions can be specified to generate data chunks of differing physical shapes. In the absence of explicit chunk dimensions, a layout for the data chunk is computed by factorisation of the total number of elements to achieve a predictable regular shape. The parameters required to generate a general workload with MACSio can be approximated by a user with a good working knowledge of

```
1  typedef struct MACSIO_IFACE_Handle_t
2  {    char                    name[MACSIO_IFACE_MAX_NAME];
3       char                    ext[MACSIO_IFACE_MAX_NAME];
4       int                     slotUsed;
5       ProcessArgsFunc         processArgsFunc;
6       DumpFunc                dumpFunc;
7       LoadFunc                loadFunc;
8       QueryFeaturesFunc       queryFeaturesFunc;
9       IdentifyFileFunc        identifyFileFunc;
10 } MACSIO_IFACE_Handle_t;
```

Listing 6.1: Interface handle structure for registering an I/O plugin with the application management routines.

mesh based datasets and the volume of data a particular size of mesh with generate when written. For more specific use cases like the replication work in this thesis, more careful parameter selection is required and this process is described in more detail in Section 6.2.

Different levels of coordination are required for distributed I/O depending on the library and I/O mode in use. Management of the creation and accesses of files in different modes can be marshalled in MACSio by a set of core utility functions. When operating in modes that require processes to coordinate, groups of processes are partitioned at initialisation and pass a baton **struct** to signal which process currently has control of the group and corresponding files.

I/O libraries are integrated into MACSio's core routines through a plugin interface. Each plugin populates the MACSIO_IFACE_Handle_t structure shown in Listing 6.1, which in turn is stored in an interface lookup table. At runtime the target I/O library plugin is retrieved from the lookup table and function pointers for data dump library functions used to dynamically hand off execution to a plugin.

Optionally, performance data can be recorded by MACSio during execution and stored in a text based log, demonstrated in Listing 6.2. Included in the recorded data is a record of data volumes, operation timings, and corresponding bandwidths for each I/O dump on a per process basis. Furthermore, statistics are gathered by process 0 and aggregated values for these metrics are reported

```
----------------------Processor 000000----------------------
Info:Dump 00 BW: 1.4365 Mi/ 0.1282 secs = 11.2007 Mi/sec
Info:Dump 00 Stat BW: 1.4445 Mi/ 0.1282 secs = 11.2637 Mi/sec
Info:Overall BW: 1.4365 Mi/ 128.2470 msecs = 11.2007 Mi/sec
Info:Summed BW: 22.3827 Mi/sec
Info:Total Bytes: 2.8729 Mi; Total I/O Time = 129.0784 msecs;
Total BW = 22.2570 Mi/sec

----------------------Processor 000001----------------------
Info:Dump 00 BW: 1.4365 Mi/ 0.1284 secs = 11.1821 Mi/sec
Info:Dump 00 Stat BW: 1.4445 Mi/ 0.1284 secs = 11.2450 Mi/sec
Info:Overall BW: 1.4365 Mi/128.4601 msecs = 11.1821 Mi/sec
...
```

Listing 6.2: Example log output for a MACSio run showing two processors performing a single checkpoint dump.

to show the performance of the application as a whole as well as individual processes.

6.1.2 Modifications

To enable MACSio to carry out the experiments presented in this thesis, a number of modifications have been made to the application. Each of these modifications is intended to increase the diversity of workloads that can be produced and similarity to real physics simulations.

Simulation Control and I/O Scheduling

The main control flow used in scientific simulations is based on a timestep loop, which contains the equations and logic to progress the state of the simulation by some amount of logical time. Importantly, the progression of logical simulation time is independent of both the system wall-clock and real-world time. Depending on the fidelity of time intervals used, an iteration of the timestep loop can move the logical simulation time on by an amount that can range from a single second to a millionth of a second.

The scheduling of regular I/O activity, such as checkpointing and storing visualisation states, is dictated by the progression of logical time-steps. It is

common for an I/O interval to be set in terms of the number of timestep iterations that have progressed, for example a checkpoint interval can be specified every 0.1 seconds of simulation time and a visualisation interval set similarly every 0.03 seconds. Scheduling I/O phases in this way can generate irregular patterns of access for an application writing to its output files, and hence an equivalent structure is required in MACSio.

Source code demonstrating the structure of the MACSio timestep loop is given in Listing 6.3. The condition to control whether a further iteration of the loop should be performed checks the current timestep t and the maximum simulation timestep maxT. Alongside these values, a dt value indicates the the time delta by which a timestep can be expected to progress the simulation time. Variation of the dt value throughout the course of a simulation allows an irregular pattern of checkpointing to be generated.

Mirroring a fully featured simulation, the first instruction in the control loop is an optional call to the MACSIO_WORK module. This module can perform different degrees of computation that are explained in greater detail later on in this section.

When a timestep is reached that indicates the next checkpoint interval, the function corresponding to the target plugin interface is called to take control of the I/O phase. This function is accessed through a function pointer to the dumpFunc member function of the target plugin interface struct.

After completion of an I/O phase, a tracking counter dumpNum is advanced and the next checkpoint timestep is calculated using the current dt value. Optionally a transformation can be applied to the I/O dataset by the MACSIO_DATA module.

TyphonIO Plugin

To demonstrate the flexibility of MACSio to investigate high level I/O libraries and perform faithful replication of the applications featured in this thesis, a plugin implementation for the TyphonIO library (introduced in Chapter 2) has been

```
1  // Main timestep loop controlled by time t
2  while (t < maxT){
3      // Allows computation to advance the current timestep
4      if (doWork){
5          MACSIO_WORK_DoComputeWork(&t, dt, work_intensity);
6      }
7
8      // Main checkpoint burst phase to be executed when
9      // we reach a checkpoint timestep interval
10     if (t >= tNextBurstDump || !doWork){
11         // Call the checkpoint function
12         // for the chosen plugin interface
13         (*(iface->dumpFunc))(argi, argc, argv, main_obj,
14                                          dumpNum, dumpTime);
15
16         // Advance dump counter and calculate
17         // next checkpoint interval
18         dumpNum++;
19         tNextBurstDump += dt;
20
21         // Perform alterations to the dataset
22         // if required for this simulation
23         if (factor > 1.0){
24             unsigned long long prev_bytes =
25                         MACSIO_UTILS_StatFiles(dumpNum-1);
26             int growth_bytes = (prev_bytes*factor)-prev_bytes;
27
28             if (growth_bytes > 0){
29                 MACSIO_DATA_EvolveDataset(main_obj,
30                     &dataset_evolved, factor, growth_bytes);
31             }
32         }
33     }
34
35     // Increase the timestep if computation routine isn't used
36     if (!doWork) t++;
37
38 } // End of timestep loop
```

Listing 6.3: Simplified source code extract demonstrating the main timestep loop in MACSio for checkpointing. Lines of code containing variable declarations and logging functionality have been omitted for brevity.

added. This plugin implements the interface functions defined in macsio_iface (see Listing 6.1) that are required to process plugin specific arguments and generate the desired I/O traffic.

Development work to produce a plugin for MACSio was largely completed over a period of 24 months, although this was not a concentrated effort and the estimated development time would be expected to be much less for an individual with experience of HPC software development. Importantly, an individual looking to develop an interface plugin for MACSio would require a good familiarity with the I/O library targeted by the plugin and a working understanding of the plugin interface routines in MACSio. It is anticipated that gaining a familiarity with the required routines in MACSio does not introduce a large barrier to uptake, and the availability of previously developed plugins provide a valuable resource to demonstrate the correct approach with which to take. Considering all of these factors, an expected development time to produce a functioning plugin could be estimated at between two and twelve months depending on experience and concentration of effort.

An incremental approach to plugin development was taken for TyphonIO, meaning that core functionality was implemented and immediately followed by a period of testing and experimentation. Following this, the scope of the plugin was increased to support a larger set of the features of both MACSio and TyphonIO. For example, initial plugin development and experimentation focused exclusively on the single shared file I/O routines in MACSio and TyphonIO with this applied strictly to a structured rectilinear mesh. Once this functionality was explored fully and verified to demonstrate agreement with the expected behaviour the remaining mesh routines in TyphonIO were able to be added to the plugin. Extensions to MACSio itself were also worked on during this period and once features such as the N-to-M file writing were developed the TyphonIO plugin could be re-evaluated to ensure it continued to generate behaviour that agreed with the applications targeted for replication.

The entry point for MACSio into the plugin is a common dump function, to

which a JSON container object is passed housing the application configuration parameters and generated datasets. Using the parallel_file_mode parameter, a decision is made to pass control to an appropriate I/O mode function in addition to performing a check that the given parameter values are valid for TyphonIO. At the top level, a distinction is made between I/O modes that will operate with parallel accesses and those where processes access the same file sequentially. Parallel access modes are specified by the Single Shared File (SIF) parameter, indicating the configuration where all processes write to a single file, or groups of processes write to multiple shared group files (MSF). The choice between these two I/O modes is indicated by the file count value that follows the SIF parameter, with a value of 1 opting for the former and a number between 1 and the total process count indicating the latter.

Non-parallel I/O modes are indicated by the parallel_file_mode parameter selecting Multiple Independent File (MIF). MIF accesses can be further divided into modes with a single file per process, indicated by the file count matching the process count, and a group shared file mode with sequential accesses to shared files.

TyphonIO is designed to natively operate in parallel shared file mode, meaning no coordination is required in MACSio when using a single shared file. In all other configurations, the I/O mode manager is used to set up and manage which ranks in each group are allowed to issue calls to the underlying library.

After each processor has created or opened their shared or individual output files, the transfer of data out of the application is handled by one of four writing functions. These functions are characterised by their handling of a structured or unstructured mesh, and whether a single part or the whole mesh is being committed to file in a single action. The combination of mesh type and I/O mode in use requires different processing of the JSON data container, the result of which is passed to specific TyphonIO functions for correct placement in the file. Ultimately, TyphonIO calls are made to create the desired type of mesh, define the characteristics of its data in the output file, and transfer all of the

```
TIO_Call( TIO_Create_Mesh(file_id, state_id, "mesh", &mesh_id,
                  mesh_type, TIO_COORD_CARTESIAN, TIO_FALSE,
                  "mesh_group", (TIO_Size_t)1,TIO_DATATYPE_NULL,
                  TIO_DOUBLE, (TIO_Dims_t)ndims, (TIO_Size_t)dims[0],
                  (TIO_Size_t)dims[1], (TIO_Size_t)dims[2],
                  TIO_GHOSTS_NONE, (TIO_Size_t)1, NULL, NULL, NULL,
                  NULL, NULL, NULL),
                  "Create Mesh Failed\n");
...
        TIO_Call( TIO_Set_Quad_Chunk(file_id, mesh_id, (TIO_Size_t)0,
            (TIO_Dims_t)ndims,0, dims[0]-1, 0, dims[1]-1, 0,
            dims[2]-1, 0, 0),
            "Set Quad Mesh Chunk Failed");
        TIO_Call( TIO_Write_QuadMesh_All(file_id, mesh_id,
            TIO_DOUBLE, coords[0], coords[1], coords[2]),
            "Write Mesh Coords failed\n");
...
json_object *vars_array = json_object_path_get_array(part_obj, "Vars");

for (int i = 0; i < json_object_array_length(vars_array); i++)
{
    TIO_Size_t var_dims[3];
    TIO_Object_t var_id;
    json_object *var_obj = json_object_array_get_idx(vars_array, i);
    json_object *data_obj = json_object_path_get_extarr(var_obj, "data");
    char const *varname = json_object_path_get_string(var_obj, "name");
    char *centering = strdup(
        json_object_path_get_string(var_obj, "centering"));
    TIO_Centre_t tio_centering = strcmp(centering, "zone") ?
                                 TIO_CENTRE_NODE : TIO_CENTRE_CELL;
    int ndims = json_object_extarr_ndims(data_obj);
    void const *buf = json_object_extarr_data(data_obj);

    TIO_Dims_t ndims_tio = (TIO_Dims_t)ndims;

    TIO_Data_t dtype_id =
        json_object_extarr_type(data_obj) == json_extarr_type_flt64 ?
            TIO_DOUBLE : TIO_INT;

    for (int j = 0; j < ndims; j++){
        var_dims[j] = json_object_extarr_dim(data_obj, j);
    }

    TIO_Call( TIO_Create_Quant(file_id, mesh_id, varname, &var_id,
        dtype_id, tio_centering, TIO_GHOSTS_NONE, TIO_FALSE, "qunits"),
                "Create Var Quant Failed\n");

    TIO_Call( TIO_Write_QuadQuant_Chunk(file_id, var_id, (TIO_Size_t)0,
        TIO_XFER_INDEPENDENT, dtype_id, buf, (void*)TIO_NULL),
                "Write Quad Quant Var Failed\n");
    TIO_Call( TIO_Close_Quant(file_id, var_id),
                "Close Quant Failed\n");

}
TIO_Call( TIO_Close_Mesh(file_id, mesh_id),
    "Close Mesh failed\n");
```

Listing 6.4: TyphonIO plugin source code demonstrating creation of a structured rectilinear (quadrilateral colinear) mesh and writing of the associated data.

associated data to the file as seen in Listing 6.4

Multiple Shared File I/O Mode

The MIF behaviour implemented in MACSio mirrors that used in the SILO library, with groups of processes accessing a shared file in sequence. In this scenario the MIF I/O mode manager is responsible for first partitioning processes into groups. Within a group, a leader is nominated to create the group's file and dump its data before handing over exclusive access to the next process in the group. This behaviour has been adapted to provide MACSio, and its underlying libraries, to perform multi-shared file I/O using true parallel accesses.

Mirroring the MIF I/O manager, a Multiple Shared File (MSF) mode module has been implemented to manage the initialisation of processes into groups such that groups are able to issue I/O requests concurrently without collision.

Each process in the simulation populates a MACSIO_MSF_baton_t struct when the MACSIO_MSF_Init() function is invoked by a plugin. The processor rank assigned by global MPI communicator is used by each processor to allocate itself to a group, at which point the global communicator is split to create a group communicator. For the purpose of coordinated parallel communications required by some I/O libraries, each processor in MSF mode must have knowledge of the ranks in its group and the global rank of the nominated group root. This information is distributed by means of an MPI_Allgather() call between all ranks of each group.

When invoked by a plugin, MSF mode is used in largely the same way as SIF single shared file I/O. Initially the creation and opening of files must be performed collectively by every rank in a group using the group communicator. Following this, parallel accesses are carried out by processors issuing library calls with reference to the shared group file handle, transferring their chunk of the dataset to non-overlapping regions of the file.

6.2 Proxy Application Validation

To assess the suitability of the MACSio proxy application to simulate I/O workloads, validation experiments have been carried out for two scientific applications. The applications used for this are the Bookleaf hydrodynamics mini-application and the FLASH-IO benchmark of the FLASH astrophysical thermonuclear explosion code. The steps of this validation involve, firstly, the profiling of behaviour and output files from the applications, followed by an analysis of the accuracy of their translation to a MACSio workload.

Modelling of applications workloads has been explored in detail in Chapter 5. From the profiles described by the previous chapter, the input parameter sets for both Bookleaf and FLASH-IO were constructed and fed into MACSio (see Tables 6.2 and 6.8). The replications were executed on two of the machines used in this thesis and profiled using Darshan to verify the I/O load when compared to the original applications.

The process of constructing MACSio parameters sets was completed mostly by manual definition with the inclusion of calculated components. Tables 6.2 and 6.3 contain the parameter values that were derived for Bookleaf at 1 Node to 64 Node scales. Considering the parameters listed in the table in turn, the first value specified is the plugin interface that will be used by MACSio. For the validation work in this chapter the TyphonIO and HDF5 plugin interfaces were used for Bookleaf and FLASH-IO respectively. This value is one determined from domain knowledge of the target application as profiling libraries are not able to extract this information. Similarly the parallel file mode, dimensionality, and part type are based on application domain knowledge as they represent higher level concepts than can be detect through analysis of I/O profiling data.

In the parameter table, part size per processor and checkpoint interval represent values that are calculated based on Darshan logs recorded from runs of the target applications. For part size this is determined based on a relationship between the overall size of the output file and how the data is distributed across

```
1  filesize = <MPIIO_BYTES_WRITTEN value>
2  variables = <Dataset variable count>
3  processor_count = <Processor count>
4
5  factor_a = 1070.7
6  constant_a = 10967.0
7
8  factor_b = 75.1
9  constant_b = 547.0
10
11 factor_c = 0.9
12 constant_c = 5.1
13
14 step1 = (filesize - (factor_a * variables) - constant_a) / processor_count
15 step2 = step1 - (factor_b * variables) - constant_b
16 step3 = step2 / ( (factor_c * variables) + constant_c )
17
18 partsize = step4 / 1024
```

Listing 6.5: Script used to calculate part sizes for MACSio based on target application output file size, number of array variables and processor count

processors and array variables. In the first instance a simple division of the MPIIO_BYTES_WRITTEN counter by the nprocs counter will give the volume of data written per processor. In order to further divide this between the array variables that make up a MACSio dataset, and in turn TyphonIO dataset, the number of variables must be factored in. Dividing the volume of data written per processor by the variable count gives an approximation of the part size required by MACSio. Initial attempts to translate Bookleaf file sizes to MACSio part sizes using a known fixed number of variables were found to contain a small margin of error due to the mesh coordinates and associated labelling (elements such as the coordinate array dimensions) influencing the total value of data that was actually written by MACSio. To account for this, file sizes were measured using MACSio while automating the varying of variable counts or part sizes. Measuring the increase in file size and factoring in the increase in variable counts and part sizes produced a series of scaling constants that are combined in the script shown in Listing 6.5. The part size values generated for Bookleaf were calculated using this script and were able to produce file sizes consistent with the original runs as demonstrated in the following section.

```
1  checkpoint file count := total checkpoints listed by Darshan MPIIO file summary
2
3  for n := checkpoint file count to 1:
4      open time := checkpoint_n(MPIIO_F_OPEN_TIMESTAMP)
5      close time := checkpoint_n-1(MPIIO_F_CLOSE_TIMESTAMP)
6      checkpoint interval_n-1 := open_time - close_time
```

Listing 6.6: Psedocode example for extracting checkpoint intervals from darshan timestamp counters.

The checkpoint intervals used to separate I/O phases during a simulation are extracted from the timestamps associated with each checkpoint file. A pseudo-code representation of the generation is shown in Listing 6.6. Similarly the checkpoint dump count is also extracted from the Darshan MPIIO per-file I/O summary which requires a simple count for the number of lines returned by this section of the log. The final parameter set in Table 6.2 is the no collective operations flag which is set to true for the original Bookleaf runs. The presence of collective operations is detected from the MPIIO_COLL_WRITES counter value in the Darshan log and this is read via a simple Python script similar to previously mentioned counters.

6.2.1 Bookleaf

Figure 6.2 shows the measured I/O times at differing process counts, specifically the total I/O time from start to finish of a checkpoint; the cumulative time spent by each processor performing I/O transfers; and the maximum time spent by any individual processor performing a single I/O operation. These results were collected using the Archer platform with a stripe count of 48 and represent the best case observed for both Bookleaf and MACSio across ten repetitions of the experiment. Each of the three graphs shown contain data points for four different categories, these being two different checkpoint files for the two applications. For clarity, MACSio #1 and Bookleaf #1 refer to the first checkpoint in the simulation and MACSio #2 and Bookleaf #2 refer to the second.

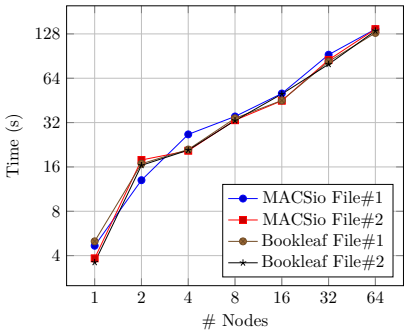
Parameter	Bookleaf Parameter Value Noh
Interface	TyphonIO
Parallel File Mode	SIF 1
Part Size per Processor	See Table 6.3
Number of Dimensions	2D
Part Type	Unstructured Mesh
Vars Per Part	9
Number of Checkpoint Dumps	2
Checkpoint Interval	See Table 6.3
Visualisation Part Size	N/A
Number of Visualisation Dumps	N/A
No Collective Operations	True
Dataset Growth Sequence	N/A

Table 6.2: Input parameter values for MACSio validation runs of Bookleaf

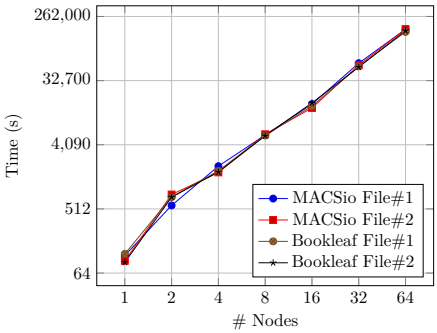
Parameter	1 Node	2 Nodes	4 Nodes	8 Nodes	16 Nodes	32 Nodes	64 Nodes
Processor Count	24	48	96	192	384	768	1536
Part Size per Processor	398.4 KB	197.5 KB	98.8 KB	49.4 KB	24.8 KB	12.4 KB	6.2 KB
Checkpoint interval	266s	120s	53s	22s	11s	7s	5s

Table 6.3: Input Parameter values for scaling Bookleaf validation runs.

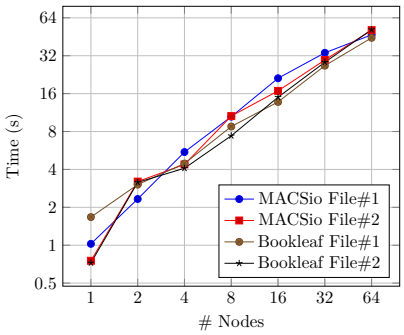
From these experiments it can be seen that the time spent executing the I/O workloads of the two applications are similar in their magnitude and trend as the applications scale. In particular, the average difference in the I/O time shown by Figure 6.2(a) has been calculated as 4%, with the largest difference being a 15% speedup for MACSio over Bookleaf. Similarly the cumulative, or aggregated, I/O times of each rank match up closely between the two applications and give a good degree of confidence in the spread of I/O work across ranks being suitably similar. The final graph shown in Figure 6.2(c) demonstrates the difference in the time taken for the longest recorded write operation issued in each of the application checkpoints. The variation in this metric is notably greater than in the previous two figures, with the average difference at the scale of a single node showing MACSio's slowest I/O operation taking 26% longer than Bookleaf. It is important to note however that the amount of time required for this operation on 1 node ranges between 0.72 and 1.67 seconds and



(a) I/O time



(b) Cumulative Write Time



(c) Max Write Operation Time

Figure 6.2: I/O timings representing the best case (10 repititions) for Bookleaf and the MACSio replication running on Archer (Lustre stripe count 48). For these results the best case is shown with the variation across repetitions being less than 9.4% for each run scale. Timings shown: (a) Start to finish time for a checkpoint (b) Cumulative time spent by all processors performing I/O (c) Time for the slowest write operation issued by a processor during each checkpoint.

at this scale variation due to machine noise is likely amplified.

Given that the best observed performance is shown up to this point, it is worth looking at the variation in performance in the repeated runs. In the graphs seen so far, the smallest I/O times are shown as these are judged to be the I/O timings that are least likely to display interference. However, to verify that there is reproducability in the experiments other results should be considered to ensure that the times reported here are not outliers. Table 6.4 shows the I/O timings corresponding to Figure 6.2(a). These values show the minimum timings in addition to the average and maximum values observed in the data. Inspecting the average values when compared to the plotted minimum

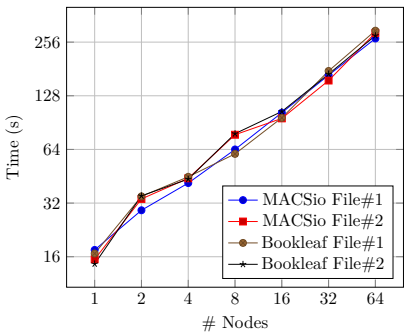
Nodes	Bookleaf Checkpoint #1			MACSio Checkpoint #1		
	Min	Average	Max	Min	Average	Max
1	3.63	3.97	4.57	4.66	4.79	4.96
2	16.49	17.33	17.78	15.87	17.27	18.83
4	20.81	23.09	27.55	26.61	26.81	27.03
8	33.16	34.68	35.51	35.26	35.99	36.74
16	50.11	50.49	51.18	50.44	52.26	54.78
32	79.66	86.36	89.83	92.47	93.03	93.89
64	134.20	136.65	138.94	137.81	138.17	140.26
Nodes	Bookleaf Checkpoint #2			MACSio Checkpoint #2		
	Min	Average	Max	Min	Average	Max
1	5.04	5.13	5.47	3.83	4.61	5.23
2	16.80	17.37	18.01	17.43	18.57	18.84
4	21.24	22.99	24.14	20.67	22.27	26.93
8	34.27	34.86	36.14	33.29	35.41	36.20
16	45.29	48.31	51.87	45.51	48.21	51.02
32	83.09	85.79	89.45	85.22	86.85	87.89
64	129.83	133.71	137.58	131.80	136.18	138.01

Table 6.4: I/O timings for Bookleaf and MACSio replication checkpoints run on Archer with 10 repetitions.

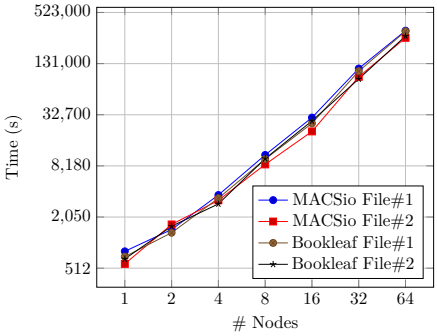
I/O times, there is minimum difference to be seen at each scale. While there is a small amount more variation between the minimum and maximum recorded values at larger scales, this is to be expected due to the larger timings overall. Regardless, the variation between the plotted best case values, the calculate mean average, and the maximum values are a handful of seconds or less in all of the cases. From these values there is confidence that the experimentation is demonstrating a fair representation of the system performance for the targeted workloads.

To increase the robustness of this validation the same set of experimental results have been collected on the Tinis platform. Tinis uses a GPFS file system and as such striping is not user controlled. As with the results taken from Archer, these results represent the best case recorded across 10 repeats in order to discount system noise as much as possible.

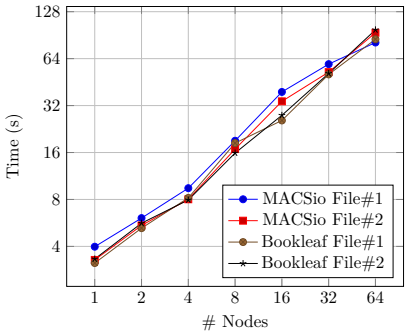
Figure 6.3 shows the three different time metrics measured on Tinis. As with



(a) I/O time



(b) Cumulative Write Time



(c) Max Write Operation Time

Figure 6.3: I/O timings representing the best case (10 repititions) for Bookleaf and the MACSio replication running on Tinis (GPFS). For these results the best case is shown with the variation across repetitions being less than 11% for each run scale. Timings shown: (a) Start to finish time for a checkpoint (b) Cumulative time spent by all processors performing I/O (c) Time for the slowest write operation issued by a processor during each checkpoint.

Archer, the results show a strong correlation between the I/O timings for MACSio and Bookleaf suggesting that the replication is indeed valid. Specifically, the average difference in I/O time shown by Figure 6.3(a) has been calculated as 9% with the largest different being an 18% difference at 8 nodes. The cumulative I/O time shown in Figure 6.3(b) again demonstrates that the distribution of I/O work across all of the processes performing I/O is in agreement between the two applications. Finally, the slowest write operation recorded by the two applications shows the MACSio replications taking slightly longer or on average 14%. This trend as the node count scales is notably more consistent that that observed on Archer.

Scale	Checkpoint Size (MB)		Size Difference (MB)	Size Difference %
	Bookleaf	MACSio		
1	124.05	124.38	0.32	0.26 %
2	124.24	123.93	0.30	0.24 %
4	124.50	123.43	1.06	0.85 %
8	125.87	123.46	1.41	1.13 %
16	125.41	123.08	2.33	1.86 %
32	126.20	122.55	3.64	2.89 %
64	127.35	122.24	5.11	4.01 %

Table 6.5: Checkpoint file size comparison between Bookleaf and a MACSio replication run on Archer.

The structure of the data that is transferred to the parallel file system by the original application and the similarly of that produced by MACSio is also important to the faithfulness of the replication. Inspection of the files produced by both applications demonstrate firstly that, when MACSio is not launched with the exact dimensions of the target dataset, a reasonably accurate estimation can be produced. Furthermore, by supplying the correct type of dataset and an estimate of the number of variables resident in the data, the checkpoint file produced is extremely close in size and contents to the original application. In Table 6.5 the variation in total file size between Bookleaf and MACSio shows that for our experiments up to 64 nodes a maximum difference of 2% is observed.

Closer analysis of the checkpoint files produced by the applications, detailed in Table 6.6, shows that the file elements in the MACSio generated dataset are faithful to a Bookleaf checkpoint in both their number and distribution. The differences observed between the data elements of the two files can be largely attributed to minor differences in the dataset dimensions, in addition to the selection of the datatype for each element. In particular, the physical topology of the dataset in both two and three dimensions has been identified as a component that can introduce variations in the number of data entries required to describe the coordinates and connectivity between nodes in the mesh. In our experiments however, the underestimation of the mesh data is counteracted by an overestimation of its associated variables. Of the nine data quantities checkpointed by Bookleaf, six of these use double precision floating point values

Bookleaf				MACSio			
Dataset Dimensions 1800 x 720				Dataset Dimensions 760 x 1596			
Element	Type	Count	Size (MB)	Element	Type	Count	Size (MB)
Mesh				*Mesh*			
Cell IDs	integer	1296000	4.94	Cell IDs	integer	1202040	4.59
Connectivity	integer	5184000	19.78	Connectivity	integer	4808160	18.34
I Coord	double	1298521	9.91	I Coord	double	1212960	9.25
J Coord	double	1298521	9.91	J Coord	double	1212960	9.25
Node IDs	integer	1298521	4.95	Node IDs	integer	1212960	4.63
	Sub Total (MB)		49.49		Sub Total (MB)		46.06
Variables				*Variables*			
Density	double	1296000	9.89	Constant	double	1202040	9.17
Energy	double	1296000	9.89	Constant_000	double	1202040	9.17
Material	integer	1296000	4.94	Noise	double	1212960	9.25
Node	integer	1298521	4.95	Noise Sum	double	1212960	9.25
Pressure	double	1296000	9.89	Random	double	1202040	9.17
X Velocity	double	1298521	9.91	Spherical	double	1202040	9.17
Y Velocity	double	1298521	9.91	X Layers	integer	1202040	4.59
csqrd	double	1296000	9.89	X Ramp	double	1212960	9.25
ielreg	integer	1296000	4.94	Y Sin	double	1212960	9.25
	Sub Total (MB)		74.21		Sub Total (MB)		78.29
	Total (MB)		123.69		Total (MB)		124.35
	Metadata (MB)		0.02		Metadata (MB)		0.02
	File Size (MB)		123.71		File Size (MB)		124.37

Table 6.6: Checkpoint file breakdown for the Bookleaf noh large problem and MACSio replication run on a single node.

occupying 64-bits per value. The remaining three quantities are represented by 32-bit integers. Due to the procedural method by which data arrays are generated in MACSio, the ratio of double to integer variables is eight to one, meaning there is a greater amount of replication data written to contiguous file regions. Finally, the aggregated data totals show that the same amount of metadata is written by each file for the purpose of labelling and indicating the location of data quantities.

In order to replicate the behaviour of a strong scaled application the part_size parameter value must be adjusted when running the same Bookleaf configuration across a greater number of processes. A comparison of the dataset generated in the largest and smallest scale experiments is presented in Table 6.7. It can be seen that the dataset dimensions generated by MACSio at larger scale produce a smaller number of mesh cells overall than at a single node, and indeed the original Bookleaf run. It is notable however, that the I Coord, J Coord, and Node IDs

		MACSio 1 Node		MACSio 64 Node		
Dataset Dimensions		760 x 1596		800 x 1536		
Element	Type	Count	Size (MB)	Count	Size (MB)	Difference
Mesh						
Cell IDs	integer	1202040	4.59	1142784	4.36	-5.05%
Connectivity	integer	4808160	18.34	4571136	17.44	-5.05%
I Coord	double	1212960	9.25	1228800	9.38	1.30%
J Coord	double	1212960	9.25	1228800	9.38	1.30%
Node IDs	integer	1212960	4.63	1228800	4.69	1.30%
Sub Total (MB)			46.06		45.23	-1.81%
Variables						
Constant	double	1202040	9.17	1142784	8.72	-5.05%
Constant_000	double	1202040	9.17	1142784	8.72	-5.05%
Noise	double	1212960	9.25	1228800	9.38	1.30%
Noise Sum	double	1212960	9.25	1228800	9.38	1.30%
Random	double	1202040	9.17	1142784	8.72	-5.05%
Spherical	double	1202040	9.17	1142784	8.72	-5.05%
X Layers	integer	1202040	4.59	1142784	4.36	-5.05%
X Ramp	double	1212960	9.25	1228800	9.38	1.30%
Y Sin	double	1212960	9.25	1228800	9.38	1.30%
Sub Total (MB)			78.29		76.73	-2.00%
Total (MB)			124.35		121.97	-1.93%
Metadata (MB)			0.02		0.17	152.27%
File Size (MB)			124.37		122.14	-1.81%

Table 6.7: Checkpoint file breakdown for the Bookleaf noh large problem and MACSio replication run on 1 and 64 Nodes.

arrays all contain 2% more elements at the larger scale. When considering the mesh specific elements as a whole, this works out at a 2% difference between the runs with the single node container the larger of the two data volumes. This pattern is mirrored by the remaining variables in the dataset and indeed the total overall size of the checkpoints.

6.2.2 FLASH-IO

Following the same process, the parameters in Table 6.8 were taken from a run of the standard FLASH-IO problem. Notably, FLASH-IO uses a 3D structured rectilinear mesh and performs I/O through the parallel HDF5 API. The decomposition that is used by FLASH-IO and replicated by MACSio is a regular 3D mesh decomposition across a 3D grid of processors which is illustrated by a simple example in Figure 6.4.

In contrast to Bookleaf the FLASH-IO application is configured to use a

Parameter	FLASH-IO Parameter Value
Interface	HDF5
Parallel File Mode	SIF 1
Number of Dimensions	3D
Mesh Part Dimensions	16 16 16
Part Type	Rectilinear Mesh
Vars Per Part	27
Number of Parts Per Rank	100/101
Number of Checkpoint Dumps	1
Checkpoint Interval	N/A
Visualisation Part Size	N/A
Number of Visualisation Dumps	N/A
No Collective Operations	True
Dataset Growth Sequence	N/A

Table 6.8: Input parameter values for MACSio validation runs of FLASH-IO

weak scaled problem, with an irregular number of fixed size data blocks per processor simulating a load imbalance. The size of the dataset, and consequently the resulting checkpoint file, is therefore easily predictable when running the application on different processor counts.

Figure 6.5 shows the I/O timings for FLASH-IO and the corresponding MACSio replication at different scales. As with the the Bookleaf replication experiments, the time spent performing checkpoint I/O for the application as a whole and cumulatively across all ranks are notably close for the original and replicated workloads. In particular, the average difference in time spent performing I/O between FLASH-IO and MACSio has been measured at 7% while the average cumulative time for all ranks differs by 19%. However, this relatively large cumulative time difference decreases as the scale is increased, suggesting that imbalance in individual I/O times is amortised as the scale of the run increases.

The timings for the slowest write operation performed during checkpointing demonstrate some interesting characteristics not present in the previous experiments. In Figure 6.5(c) the increase in maximum write operation time can be seen to be broadly linear as processor count increases, however also display a

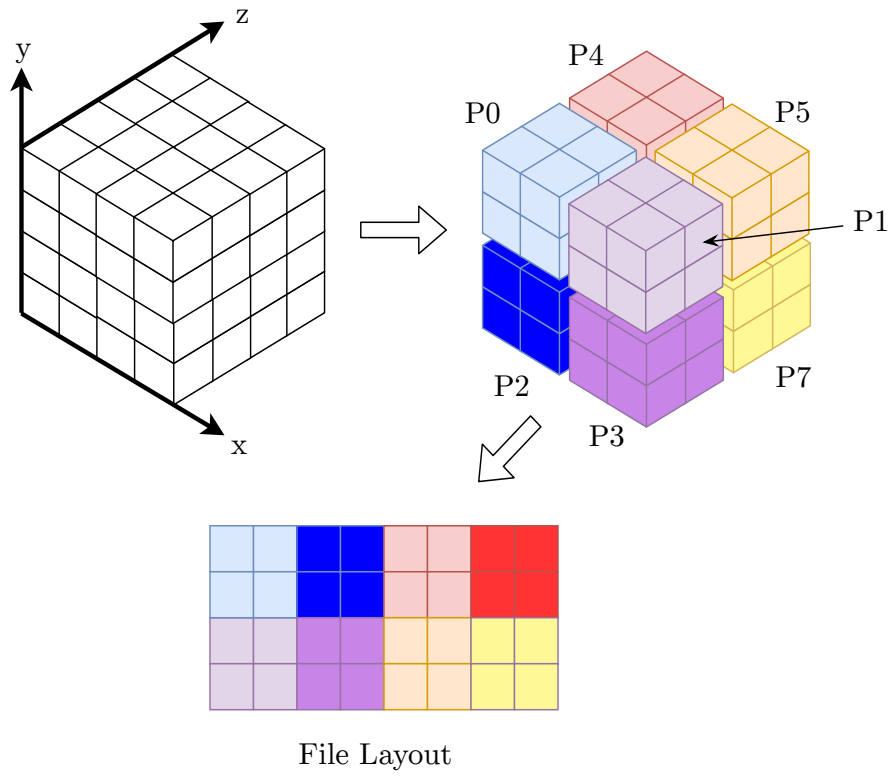
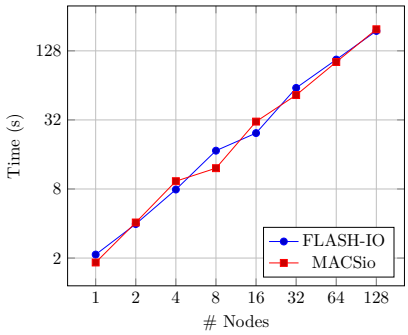


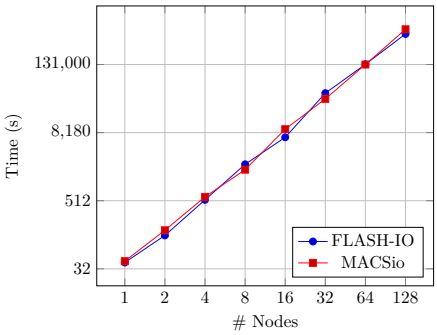
Figure 6.4: FLASH-IO 3D mesh decomposition to a 3D processor grid and the corresponding file layout.

saw-tooth pattern of a larger increase between consecutive experiment scales followed by a comparatively smaller increase. This trend is displayed by both the original application and replicated workload, suggesting that the dataset composition and I/O operations generated by this dataset share similar performance characteristics on ARCHER.

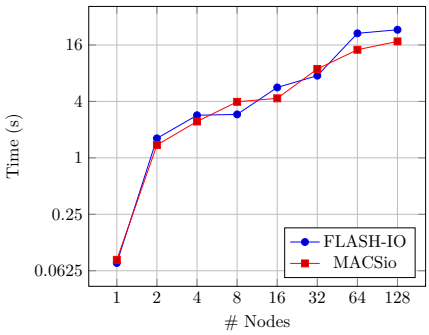
Checkpoint files generated by the FLASH-IO setup in these experiments are made up of variables in the form of HDF5 scalar datasets. The variable unknowns in the simulation that represent the majority of the data is stored in a number of four dimensional datasets, with the first three dimensions being that of the block size and the fourth the number of data blocks in the simulation. The data layout used in a FLASH-IO allows for each of the data blocks on a rank to be stored in one contiguous region in a checkpoint file and is achieved using



(a) I/O time



(b) Cumulative Write Time



(c) Max Write Operation Time

Figure 6.5: Best case I/O timings for FLASH-IO and the MACSio replication collected on Archer (stripe count 48) showing: (a) Start to finish time for a checkpoint (b) Cumulative time spent by all processors performing I/O (c) Time for the slowest write operation issued by a processor during each checkpoint.

the HDF5 hyperslabs functionality. This layout is mirrored by the hyperslab definitions used in the MACSio HDF5 plugin, and hence produce an equivalent dataset layout. The total volume of file data this generates at each experimental scale is shown in Table 6.9. It should be noted that MACSio produces a volume of data that is around 0.02% less than the equivalent FLASH-IO run. The source of this discrepancy is mainly due to additional metadata and labelling datasets that are produced by FLASH-IO to describe features such as the block size and block bounding box, the composition of which is shown in Table 6.10. Because of the specialised nature of these datasets, they are not easily reproducible in MACSio and hence result in the checkpoint differences observed. However, this proportion of data not represented in MACSio is extremely small when

Scale	Checkpoint Size (GB)		Size Difference (MB)	Size Difference %
	FLASH-IO	MACSio		
1	1.78	1.78	0.38	0.021 %
2	3.56	3.55	0.76	0.021 %
4	7.10	7.10	1.52	0.021 %
8	14.20	14.20	3.04	0.021 %
16	28.41	28.40	6.07	0.021 %
32	56.82	56.81	12.14	0.021 %
64	113.64	113.62	24.27	0.021 %
128	227.30	227.25	48.53	0.021 %

Table 6.9: Checkpoint file size comparison between FLASH-IO and a MACSio replication.

	FLASH-IO Checkpoint Composition			
Block Size		16x16x16		
Blocks Per Rank		100		
	Count	Dimension Sizes	Type	Dataset Size
Regular Datasets				
4D Unknown Variables	24	100x16x16x16	64-bit float	1.8 GB
Irregular Datasets				
Bounding Box	1	100x3x2	64-bit float	112.5 B
Block Sizes	1	100x3	64-bit float	56.3 B
Coordinate Count	1	100x3	64-bit float	56.3 B
Grid ID	1	100x15	32-bit int	140.6 B
Node Type	1	100	32-bit int	9.4 B
Refine Level	1	100	32-bit int	9.4 B
Variable Label		24x1	string	96 B

Table 6.10: Checkpoint file breakdown for a FLASH-IO run on a single node.

compared to the I/O load generated as a whole. Moreover, the closeness of the data volumes, dataset compositions and the performance characteristics of the two applications indicates that a faithful replication of the I/O work performed in FLASH-IO has been carried out by MACSio.

6.3 Summary

Modern supercomputer designs are becoming increasingly diverse, and changes to processing architectures are seeing the balance between compute and data sub-systems shift as the compute density of nodes featuring components such as GPUs far exceeds that of traditional CPU systems. To address data movement and reliability issues in these new platforms, storage system architectures are

also changing to introduce new hardware and software layers to prevent an even larger gulf opening between compute and I/O capability. However, evaluating the best way to deploy these new storage layers is not straightforward, with a need to asses the performance of the current application I/O practices in addition to new paradigms and software libraries. As such, representative yet flexible generation of application I/O is vital to enable successful research to be conducted.

This chapter has detailed work towards the development and operation of MACSio, a parametrised proxy application consisting of a set of core benchmark functionalities and I/O library plugins for the synthetic generation of workloads with a close resemblance to real scientific applications. This research has served to validate the ability of MACSio, with careful parameter selection, to generate scientific-like datasets and patterns of I/O that can be used to effectively stand-in for the data storage phase of a scientific application. The results presented demonstrate that for two applications that checkpoint with high level I/O libraries, MACSio recreates the resulting dataset to within 0.5% and 0.02%. In the former case, an approximation used to generate a data layout and achieves a high level of accuracy when provided with relatively little information about the target data. This accuracy is then improved up, with the second replication of the FLASH-IO code maintaining a consistent 99.98% similarity to the original dataset.

Analysis of the performance characteristics achieved by the target applications and their replications demonstrate that MACSio is able to produce an I/O workload achieving comparable performance on the system, with the largest difference in checkpoint duration measured at 15% and 19% and an average replication error of 4% and 7%.

The work in this chapter directly motivates a use case for proxy application based benchmarking and optimisation, which can build on the knowledge gained through library plugin development and deployment for the purpose of influencing I/O practices going forward. In the next chapter, we demonstrate

the benchmarking process for a number of application workloads, including those profiled in Chapter 5. Proxy applications like MACSio are uniquely well suited to this task due the the flexibility of a parameter based configuration and the ability to swap out I/O components with much less work than would be required in a real application.

CHAPTER 7

I/O Performance Benchmarking and Optimisation

The evolution of data storage systems, which can incorporate new storage layers and devices, has led to multi-layered software stacks obscuring the movement of data between an application and some end-point storage device.

For those looking to achieve optimal I/O performance, this has created a barrier which needs to be overcome. From the perspective of application developers, I/O performance is easily ignored as a problem to be solved by software library developers and system procurements. However, work invested in tuning the parameters of a high level library, MPI-IO implementation and parallel file system will often be highly specific and require repeating many times over. This can quickly become infeasible when dealing with large code bases that can take large amounts of time to build and run under experimental configurations. A number of research efforts have been carried out to demonstrate the value of tuning elements in this software stack for different applications in an effort to guide best practices for I/O. The research presented by Yu et al. [101], Acharya et al. [1], and Behzad et al. [9] all suggest that performance improvements upwards of an order of magnitude are possible through correct selection of parameters for the high level library, middleware, and parallel file system.

System administrators and those charged with procurements will often seek to solve their institutions I/O problems by providing larger storage systems promising a higher ceiling on IOPS and bandwidths. In reality, these are difficult to evaluate in a way that truly represents the applications that will be used in production. The problems this generates are two-fold where (a) the impact of introducing new hardware such as burst buffer layers cannot be easily quantified to make the most efficient procurement decisions, and (b) it is rarely clear to

users if they are achieving peak performance from these storage devices.

Replicating the characteristics of an I/O workload in a portable and flexible way presents a host of opportunities for optimisation in the form of both parameter tuning and more dramatic I/O strategy changes. For example, the substitution of the entire I/O library, or shift to a different file scheme alongside tradition tuning of Lustre stripes and MPI-IO hints. This chapter presents a collection of performance optimisation studies carried out using MACSio and workloads representative of three scientific applications. With similar aims to those of [101], [1], and [9], the first study contains an investigation of the parameter space to find optimal configurations for a set of workloads for each application. Secondly, comparisons of how different I/O libraries perform for the given workloads when also factoring in alternative file strategies to those used by the target application. Finally, an evaluation of two contrasting burst buffer technologies is demonstrated alongside a characterisation of how the presence of burst buffers impact the I/O performance of a system from the perspective of an application.

7.1 Approach

All of the experimentation performed in this chapter has used the MACSio proxy application to represent the I/O behaviour of three physics simulations. The first two of these targeted codes are the Bookleaf and FLASH-IO applications that have been previously used to validate the capabilities of MACSio. Additionally, a large production multi-physics application labelled AWE01 has been replicated; this is particularly valuable owing to the lack of portability of the application that results from both its size and commercial sensitivity.

The three source applications are represented by a total of seven MACSio workload configurations. Of these configurations, Bookleaf and FLASH-IO are both used with a standard and large workload while AWE01 is configured with three notable I/O pattern variations. The suite of experimental configurations

is summarised in Table 7.1.

	Bookleaf	FLASH-IO	AWE01
Physics Domain	Shock Hydrodynamics	High-energy Density Physics	Multi-Physics
I/O Library	TyphonIO	HDF5	TyphonIO
File Configuration	N-1	N-1	N-1
Default Scaling	Strong Scaling	Weak Scaling	Strong Scaling
Dimensions	2D	3D	2D
Problem Inputs	Standard, Large	Standard, Large	A, B, D

Table 7.1: Summary of the experimental target applications used as workloads inside MACSio.

The AWE01 application is made up of a core hydrodynamics package with a number of additional physics packages that can be enabled depending on the user supplied input deck. Consequently, the application can behave differently depending on the addition packages activated during a simulation. This can have a notable effect on the I/O that is performed, producing different profiles. The profiles used for these experiments were extracted from the target application and detailed in Chapter 5.

For reference, Chapter 4 contains details of the experimental configurations used in this and previous chapters.

7.2 Tuning the Parallel I/O Software Stack

The assessment of tuned I/O stack performance involved experiments run primarily on Archer, Quartz and Ray. However, a proportion of the results for the AWE01 workloads have also been collected from the Cab, Titan, Taurus, and Tinis machines[1].

7.2.1 Middleware

The entirety of the parallel I/O performed in this thesis is completed indirectly through the use of the MPI-IO middleware layer working underneath a higher

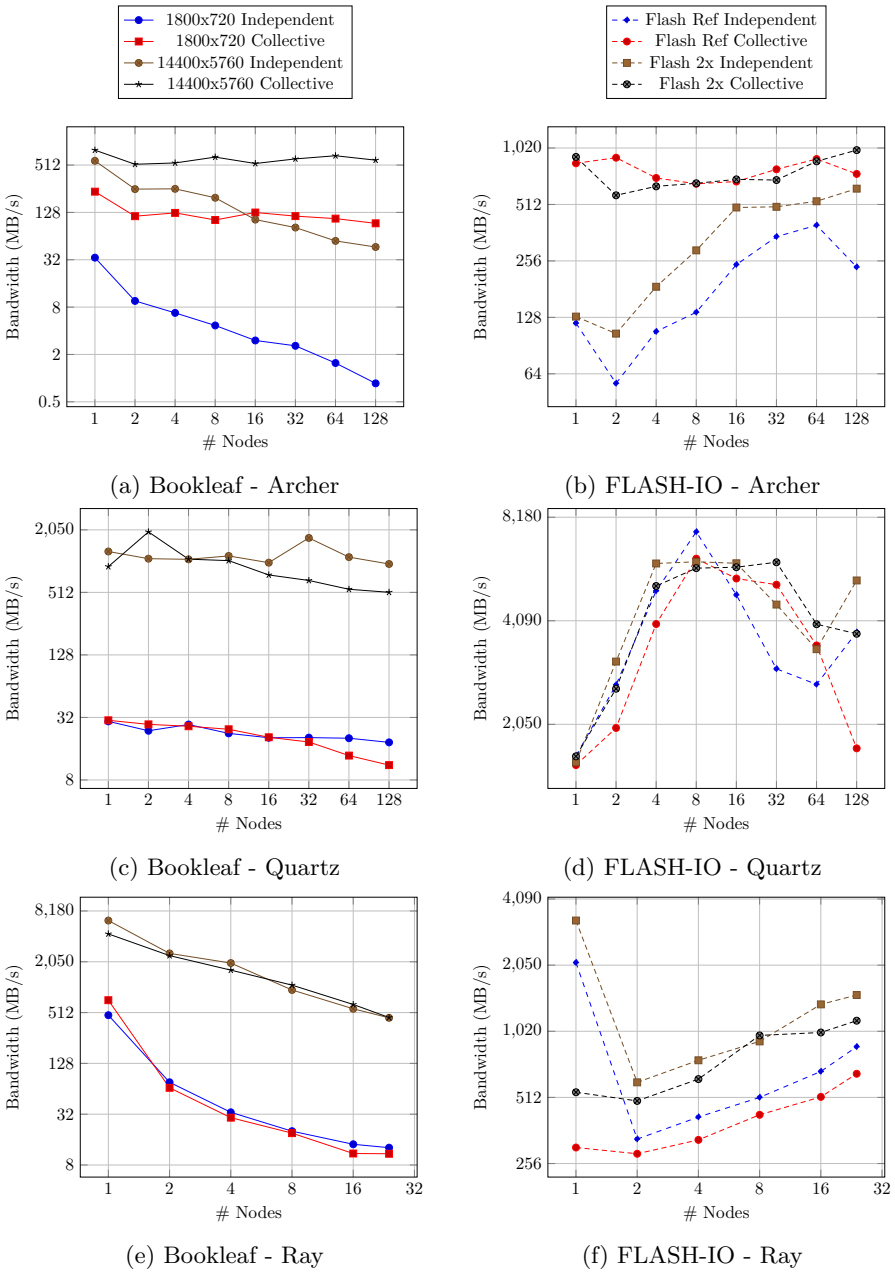
[1]Different groups of machines have been used for experimentation throughout this chapter due to the period of time over which data was collected and the variation in machine availabilities over this period.

level library. Consequently, performance can be gained or lost in all of the targeted workloads through correct tuning of the operations performed at this middleware layer.

File writes in MPI-IO can be performed using either the MPI_File_write_at or MPI_File_write_at_all functions, the former called independently between ranks and the latter called collectively by all ranks taking part in the I/O. Issuing collective calls affords the library the opportunity to perform a number of optimisations, namely the nomination of a subset of writers, aggregation of data into large requests, and reduction in file locking where parallel actions do not clash. The default behaviour implemented in both Bookleaf and FLASH-IO is to issue independent writes through the use of data transfer property list parameters, which instruct their I/O libraries to invoke MPI_File_write_at for performing data transfer.

Figure 7.1 demonstrates the checkpoint bandwidth for both workloads in their default configuration and with write calls made collectively. The impact of enabling collective operations varies between machines, with the largest difference apparent in the Bookleaf workload on Archer and Quartz. Interestingly, the checkpoint performance on the two machines is impacted in contrasting ways (as seen in Figure 7.1(a) and Figure 7.1(c)). On a single Archer node, a standard Bookleaf checkpoint sees a speedup of 7.8× with a large problem speedup of 1.4× when collective writes are enabled. As the node count increases, the independent checkpoint times experience a large slowdown despite the fixed checkpoint size. However, the checkpoints performed with collective writes remain largely consistent with regards to time taken. As a result of the difference in scaling behaviour, at 128 nodes the collective operation speedup grows to 101.1× and 12.7× for the standard and large workloads respectively.

The scaling behaviour of the two checkpoint classes on Quartz is broadly the reverse of that observed on Archer, but with a much smaller slowdown in checkpoint time as the number of nodes increases. Specifically, the slowdown in checkpoint time for collective calls dominates that of the independent con-



(a) Bookleaf - Archer

(b) FLASH-IO - Archer

(c) Bookleaf - Quartz

(d) FLASH-IO - Quartz

(e) Bookleaf - Ray

(f) FLASH-IO - Ray

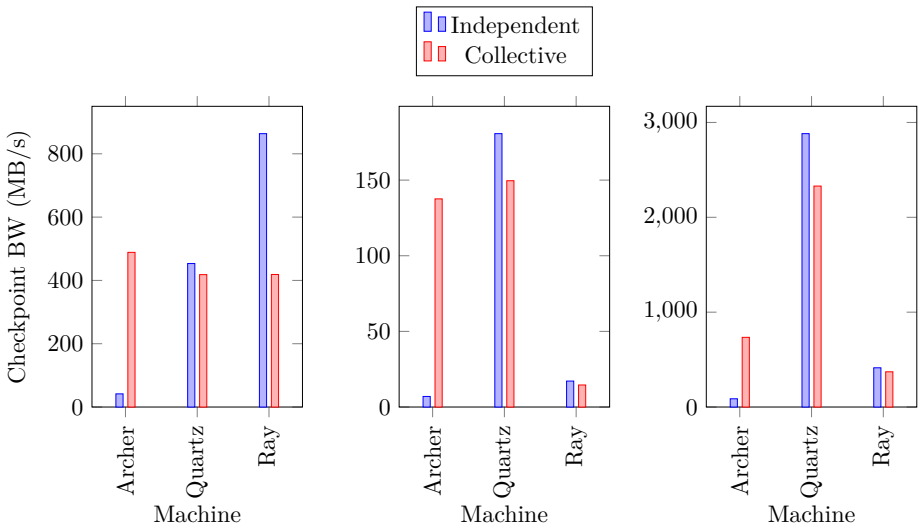
Figure 7.1: Checkpoint bandwidth for Bookleaf and Flash workloads run independently and collectively on Archer, Quartz and Ray.

figuration by 3.1× to 1.8× for the standard workload size and 1.8× to 1.3× for the large workload. This results in a 1.7× and 1.9× improvement in checkpoint time when using the default independent operations for the two workload sizes.

Performance characteristics of the FLASH-IO workloads run on the two Lustre systems further illustrate the pattern of collective calls outperforming independent calls on Archer but the trend being more varied on Quartz. Specifically, Figure 7.1(b) shows collective calls consistently outperforming independent calls by a minimum of 2.3× for a standard problem size and 1.4× for the large problem on Archer. Of note at the 128 node count is the fact that the collective run performance at reference scale decreases relative to the bandwidth at 64 nodes by around 150 MB/s. Coupled with this the continued increase of the larger independent run performance appears to show the two converging. It is inconclusive as to the strength of this trend without the availability of further data points, however the cause of this conversion may be the result of the reference problem size reaching a scale at which data stripes are colliding on the Lustre OSTs. A similar behaviour is not seen for the larger problem sizes as the additional efficiency of larger writes per process dominates any Lustre effects. The optimal configuration for FLASH-IO workloads on Quartz is less clear, and Figure 7.1(d) shows minimal performance differences between independent and collective runs until the node count reaches 128, where the independent timings dominate by 2.2× and 1.4×. The surprising lack of discrepancy between the different problem sizes and the the different write modes is not well understood currently, and the poor performance shown by collective operations similar to that observed with Bookleaf suggests that configuration of the underlying MPIIO library and file system setup may be conflicting to prevent the anticipated improvements.

The final experiments carried out in this set use the GPFS file system in Ray, and overall demonstrate a minor performance penalty when enabling collective operations. Bookleaf checkpoint times demonstrate identical scaling behaviour for both configurations with the maximum performance difference measured at 1.5× for the larger workload running on 8 nodes. Similarly, FLASH-IO checkpoints performed without collective calls are subject to a performance improvement of around 1.2× with the exclusion of the single node runs which

have a discrepancy of between 6.9× and 6.0× at standard and large problem sizes respectively.



(a) Problem A - 1 node (b) Problem B - 5 Nodes (c) Problem D - 5 nodes

Figure 7.2: Perceived checkpoint bandwidth for the AWE01 workloads using independent and collective calls on Archer, Quartz and Ray. The Lustre stripe count used on Archer and Quartz is 4 for (a) and 20 for (b) and (c). These results represent the best observed performance across 10 repetitions.

Figure 7.2 shows the result of applying the same configuration to three production workloads taken from the AWE01 application. Due to the simulation constraints, the calculation and consequently checkpointing can only be carried out by a limited number of nodes. Under the typical simulation configuration each of the three experimental platforms demonstrate similar performance characteristics to those seen previously. Archer experiences a minimum speedup of 8.4× for collective calls from what is a notably slow baseline when writes are performed independently. Such a low baseline figure suggests the uncoordinated writes are colliding at the OST and hence lock contention could be a key component limiting performance. This is supported by the large speedup to bandwidth figures that equate to those seen on Quartz for problem A and B collective runs which are made possible by coordination of writes eliminating lock contention at the OST level. The same behaviour is also observed for problem D, however

the use of collective operations does not match the 2000 MB/s performance seen on Quartz. Quartz demonstrates different behaviour in response to the introduction of collective writes, only achieving between 81% and 92% of the checkpoint bandwidth of the original runs. This response to coordinated writes suggests that Quartz was unlikely to have been experiencing a bottleneck due to OST lock contention, suggesting that the file system is not configured to hold file locks in the same way as that seen on Archer. The small drop in performance could then be attributed to the coordination overhead between writing processes which would normally be masked by the improved write speed.

Independent		Collective	
Access Size	Count	Access Size	Count
3712 B	24,576	1 MB	119
1856 B	12,288	284 B	20
4 B	6142	64 B	11
6720 B	3071	207 B	6

Table 7.2: Write sizes issued to the parallel file system during a Bookleaf checkpoint on Archer performed independently and collectively

Independent		Collective	
Access Size	Count	Access Size	Count
2640 B	36,864	2640 B	36,864
1320 B	12,288	1320 B	12,288
4 B	6142	4 B	6142
4640 B	3071	4640 B	3071

Table 7.3: Write sizes issued to the parallel file system during a Bookleaf checkpoint on Quartz performed independently and collectively

Allowing MACSio to issue collective operations to the I/O library for each target workload on Archer does indeed deliver the expected performance optimisations. However, the same effect is not observed for the equivalent runs on Quartz, despite both systems performing I/O to Lustre parallel file systems. Furthermore, the GPFS based file system in Ray also fails to demonstrate any performance improvement from the collective buffering. Closer analysis of the access sizes generated during each run identify differences in the middleware behaviours that are observed on these three platforms. Table 7.2 and Table 7.3 show the most common write sizes generated by the Bookleaf workload at 128

nodes on Archer and Quartz. By default the number of aggregators used for collective write buffering will attempt to match the stripe count used, which on Archer would be 48 due to this being the number of OSTs available for these experiments. In the case of Archer, the large number of writes generated with sizes in the range of thousands of bytes are replaced by comparatively few aggregated writes of 1 MB. In contrast, due to a lack of file system driver support the size of writes issued to the file system for the equivalent workload on Quartz remain identical despite collective routines being requested. In this case the MPIIO library supported by Quartz has been built without the inclusion of the Lustre Abstract-Device Interface for I/O (ADIO) driver. This prevents some MPIIO capability around collective buffering from operating correctly. A similar configuration also causes Ray to demonstrate the same lack of data aggregation for writes. The translation of many smaller writes to fewer larger writes that are aligned with 1 MB stripes on the parallel file system is responsible for both reducing the number of file locks that need to be held during the checkpoint, and maximising the write bandwidth that can be achieved by parallelising writes across storage targets. Consequently, the failure of the middleware implementation on Quartz and Ray to successfully aggregate data writes leads not only to a loss of potential performance, but due to added overheads for collective synchronisation experiences a slowdown in real terms when compared to the default routines. Importantly, routines in both the HDF5 and TyphonIO libraries identified the transfers carried out in these cases as collective, despite the discrepancy in behaviours further down the I/O stack.

Having quantified the performance improvements seen on Archer with collective buffering in the middleware layer, additional experiments have been carried out to determine the potential for tuning the parameters that control collective buffering. In particular, the total number of buffering nodes and the total buffer size that can be used are components that can be specified to affect how middleware optimisations are performed.

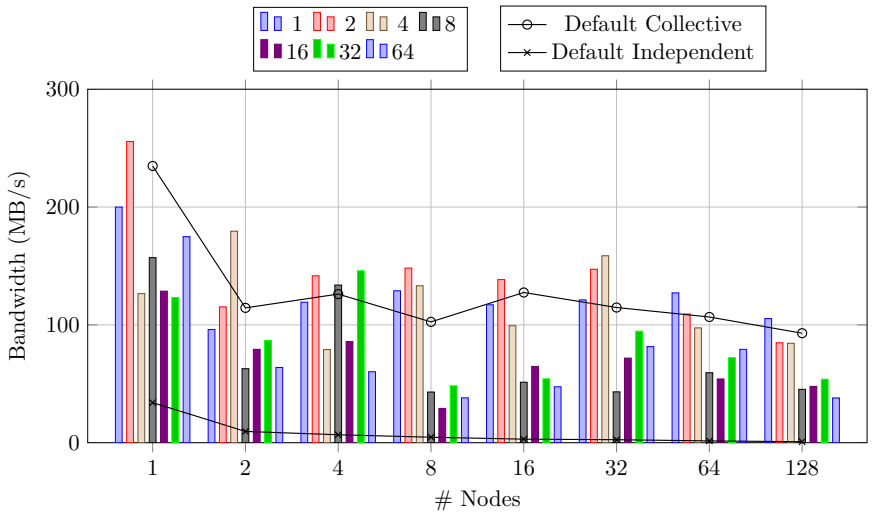
The purpose of these tests is to identify potential for a greater number

of aggregators or larger buffers to improve the throughput of data from the higher level libraries to the parallel file system. Firstly, changing the number of aggregators to identify if buffering data into optimal sized writes can be improved by a greater number of aggregator nodes issuing collective buffered writes in parallel. Additionally, increasing the size of the buffer available to aggregating ranks to eliminate any hidden buffering latency in the first phase of the two phase collective algorithm.

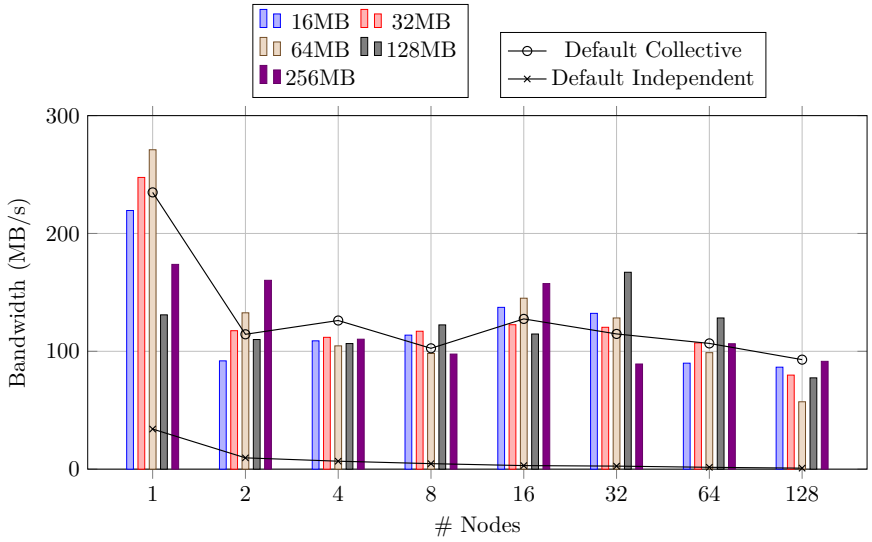
Figures 7.3 and 7.4 show the observed bandwidth for **cb_nodes** values between 1 and 64 nodes and **cb_buffer_size** values from 16 MB to 256 MB, in addition to the bandwidth achieved by the default configuration shown earlier.

For both target workloads, explicitly setting the parameters can be seen to cause large variations in checkpoint bandwidth, however tuning these components fails to demonstrate a consistent performance improvement. Selecting a lower number of aggregators, between 1 and 4 aggregator ranks, delivers comparable or slightly better performance than that measured in previous experiments. Importantly, these aggregator counts are less than or equal to the number of Lustre OSTs used in the experiments, which was kept at the system wide default of 4. When **cb_nodes** is configured to use an aggregator count greater than the number of storage targets available the middleware layer is unable to avoid write collisions with multiple writers attempting to access blocks in the same file region. When this occurs, one of the writers holds the file lock and the second must wait for this to be released before a write can be completed. The penalty for causing this collision on an OST clearly negating any potential benefits from attempting to increase aggregator count alongside the node count for the simulation. This is clearly demonstrated by the FLASH-IO workload running on 8 or more nodes, where collective performance with a large aggregator count drops to below that achieved by using non-collective routines.

Having identified lower numbers of aggregators (specifically less than the file stripe count) as the most effective value for **cb_nodes**, an increase in the size of the buffer available to these aggregators was trialled. As expected, the strongly



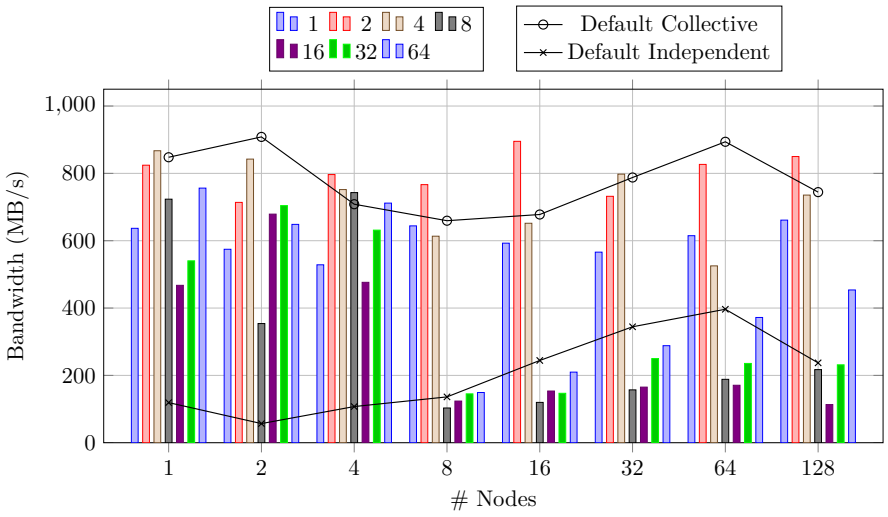
(a) **cb_nodes**



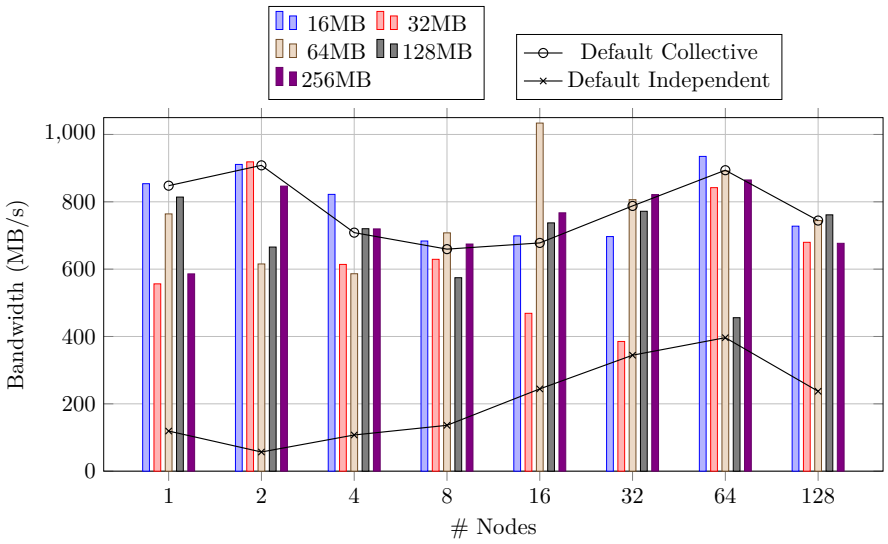
(b) **cb_buffer_size**

Figure 7.3: Perceived checkpoint bandwidth for the Bookleaf workload on Archer with different (a) aggregator node count (b) collective buffer size.

scaled Bookleaf workload exhibited very little performance deviation from the original run due to the amount of data per aggregator not changing as the simulation scales to larger node counts. More surprisingly, increasing the size of the buffer demonstrated an equal lack of performance improvement for FLASH. With the weak scaling behaviour of the workload producing around 40 MB



(a) **cb_nodes**



(b) **cb_buffer_size**

Figure 7.4: Perceived checkpoint bandwidth for the Flash workload on Archer with different (a) aggregator node count (b) collective buffer size.

of data per process coupled to a comparatively small number of aggregators, it would be expected that an increase in buffer size is required as the node count grows. For the 128 node setup, a total of 123 GB of data is generated, corresponding to the default number of four aggregators having to gather and transfer 30 GB of data each. With the performance of any of the buffer sizes

failing to provide a performance speedup (with the exception of 64 MB buffers on 16 nodes), it can be concluded that the buffering algorithm implemented in the middleware layer is able to operate sufficiently fast for our workloads as to not introduce a bottleneck. Furthermore, neither of sets of experiments identify any notable capacity for performance tuning at this level of the I/O stack.

A final observation from the attempts to tune MPI-IO middleware parameters is that while I/O performance cannot be improved by changing the behaviour of this layer in isolation, the impact that middleware configuration can have in the context of the parallel file system can be very important. This was demonstrated by the requirement that aggregator count remain less than or equal to the number of storage targets in use. Moreover, any change to the file system configuration should be followed by validation of the current parameter settings remains optimal.

Alongside the results presented in this section it is useful to consider a reference point for the performance that has been observed in another study on one of the experimental platforms. Turner et al. [87] presents a range of experiments designed to measure the peak bandwidth available on Archer with the use of a simplified benchmark problem. This work uses a simple 3D block decomposed problem to write to a single shared file and measure the maximum throughput available compared to the theoretical peak. The theoretical bandwidth achievable from the parallel file system on Archer is dictated by the number of scalable storage units that it is built from, in this case a theoretical 30 GiB/s is available from 5 storage units. Benchmarking work in this study shows that for the simplified problem, I/O performed through the NetCDF library (operating through HDF5) achieved 11.49 GiB/s in a setup configured to make use of Lustre striping. For reference, the performance observed on Archer for experiments using MACSio FLASH-IO workloads achieve just 10% of this performance, failing to display equivalent scaling behaviour. The reasons for this gulf in scalability are not immediately clear, and clearly the workload generated by MACSio is exhibiting much more complex I/O behaviour than the simplified test problem

preventing it from achieving anywhere near peak performance.

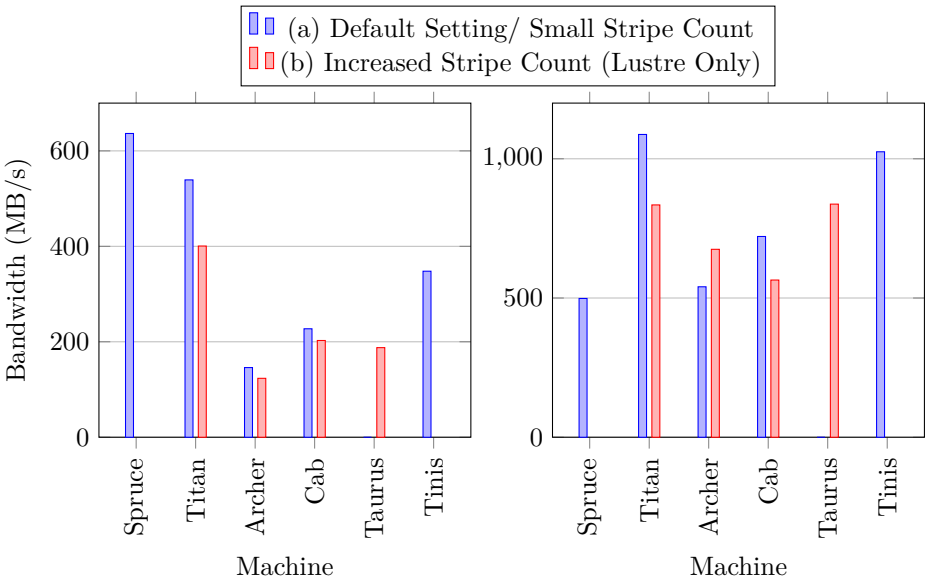
7.2.2 Parallel File System

Difficulties in tuning I/O middleware behaviour have demonstrated how failure to configure parallel file system parameters in addition to the rest of the I/O stack can cause dramatic degradation of performance. Furthermore, the scale and distribution of a real application's workload may be such that it is not possible to benefit from the peak available performance of a large parallel file system.

Making use of the rapid workload deployment enabled by MACSio, the three AWE01 problems described in Chapter 5 were run on the Titan, Archer, Cab, Taurus and Tinis platforms. This process simulated the type of file system benchmarking that may be performed for the purpose of evaluating a system for procurement. Initially, each problem was run in its default configuration, with Problem A occupying a single node and Problems B and D occupying 5 nodes. The provided system MPI installation with default drivers was used for each machine with no explicit tuning of library parameters to benchmark application performance of the system as is provided to users. For the file system configuration both the default stripe count and in increased stripe count were tested, the increased count being equal to the total number of cores used or the total OST count should this number be the smaller of the two.

Figure 7.5 shows the observed checkpoint bandwidth of the A and D workloads recorded on each machine in addition to the original application running on the AWE Spruce A system. Interestingly, the original simulation outperforms all other systems for the first problem achieving a peak of 636 MB/s; conversely, problem D performs better on the five benchmarked machines achieving only 498 MB/s on Spruce A.

Of the benchmarked systems, the highest throughput under the standard file system configuration comes from Titan delivering 539 MB/s and 1087 MB/s representing a 0.85× and 2.18× speedup. In addition, an increase in the number



(a) Problem A (b) Problem D

Figure 7.5: Perceived file bandwidth achieved for AWE01 workloads with simple file striping strategies on the Spruce A, Titan, Archer, Cab, Taurus, and Tinis platforms.

of storage targets used leads to a performance degradation of 35%. While the increase in storage devices and greater utilisation of available I/O servers offers a greater amount of aggregate parallel bandwidth to disk, the transfer sizes generated by the application and subsequent I/O stack are poorly aligned to stripe boundaries. The resulting collisions by overlapping processes prevent the storage target increase from having the desired effect.

Archer proves to be the benchmarked system offering the worst performance in these experiments achieving just under 23% of the bandwidth of Spruce. This may seem surprising given the scale of the system, Archer being a tier-1 system, however the Lustre file system alongside Archer is smaller than expected for a petabyte scale machine due to the hardware deployment strategy used; by comparison the file system available to Cab is around twice the size for a machine offering a quarter of the processing capacity. Three Lustre file systems are used to support data storage requirements, with users allocated to one of the three for the duration of their project. An architecture such as this makes

use of type of static load balancing, attempting to avoid file contention by allocating users to file systems without any consideration of the I/O footprints of their individual workloads. Statically load balancing across file system resources will inevitably improve overall storage throughput, however this is unlikely to perform as optimally as the dynamic load balancing that occurs in the Lustre stripe allocation policy. Additionally, the file system ratio of OSTs to OSSs in Archer is the highest of any of the benchmarked systems at 4 OSTs per server; a greater disk to server ratio further contributing to storage target contention with a greater number of requests processed by each server.

The final production platform in this list is Cab, which achieves checkpoint bandwidths of 227 MB/s and 721 MB/s under standard configuration. This performance is particularly noteworthy due to system being configured to use only a single storage target by default. Increasing the stripe count to make use of all available OSTs produces similar results to that observed on Titan with an 11% and 21% loss in bandwidth.

Aside from Titan, which exists is in a different class of system, the next largest file system by OST count is Taurus. By default data stripes are distributed across each of the 96 storage targets, delivering 188 MB/s and 837 MB/s checkpoint bandwidths. Due to an atypical system configuration, it was not possible to request a different stripe count during the launch of a job making it impossible to record a run using a smaller proportion of the file system as with the other platforms. Considering the large number of OSTs that must be used on Taurus, the perceived bandwidth in these results equates to less than 10 MB/s per OST which is a fraction of what a system like this would aim to achieve.

Tinis is the only system in these benchmarking tests that uses a GPFS file system for parallel data storage. This file system is of a comparatively small size comprising only 2 I/O servers managing a total of 12 storage volumes. Due to the expected constraints of such a small system, it is surprising that Tinis achieves higher file bandwidth than Archer, Cab, and Taurus for all configura-

tions of the two workloads. The cause of this level of performance is difficult to attribute to specific file system characteristics due to the dynamic striping used in GPFS that cannot be configured by the user at runtime.

Figure 7.6 shows the observed bandwidth for Problem B checkpoint plotted over the duration of a run across five nodes. Each figure is also overlaid with the size of the checkpoint at that point in the simulation to correlate changes in I/O performance with the change in dataset composition. From beginning to end a 1.7 GB change in checkpoint size occurs between the initial state of 256 MB to the terminating state at 1.9 GB, representing a 656 % increase in data volume. In conjunction with data increase, checkpoint performance also increases in each configuration that has been tested by different amount summarised in Table 7.4.

The reference run from the original application on Spruce shows that the first checkpoint achieves just 49.9 MB/s. The first major change in data volume occurs around the fifth checkpoint with a 94% increase in data triggering a 50% jump in throughput. Between here and checkpoint fifteen the dataset size increases at diminishing increments, which corresponds to an unpredictable fluctuation in I/O performance with the peak bandwidth achieved at 278.5 MB/s. After this point the checkpoint differences become more incremental, corresponding to stabilisation in performance with the exception of a 15% dip around the twentieth timestep. Overall a 229.5 MB/s swing is measured, which in fact corresponds to a 468% increase when considering the poor initial performance.

	Perceived File Bandwidth (MB/s)			
	Minimum	Maximum	Change	% Change
Spruce	49.9	278.5	229.5	468 %
Titan (Default)	489.5	961.0	471.5	96 %
Titan (Striping)	171.0	282.8	111.8	65 %
Archer (Default)	27.0	206.3	179.3	633 %
Archer (Striping)	25.7	284.4	258.8	1008 %
Cab (Default)	94.7	443.3	348.5	368 %
Cab (Striping)	34.7	201.7	167.0	481 %
Taurus	162.1	588.9	426.8	263 %
Tinis	51.0	399.9	348.5	684 %

Table 7.4: File bandwidth changes for checkpointing over the course of the AWE01 Problem B workload.

Out of the the machines surveyed, the greatest similarity in performance

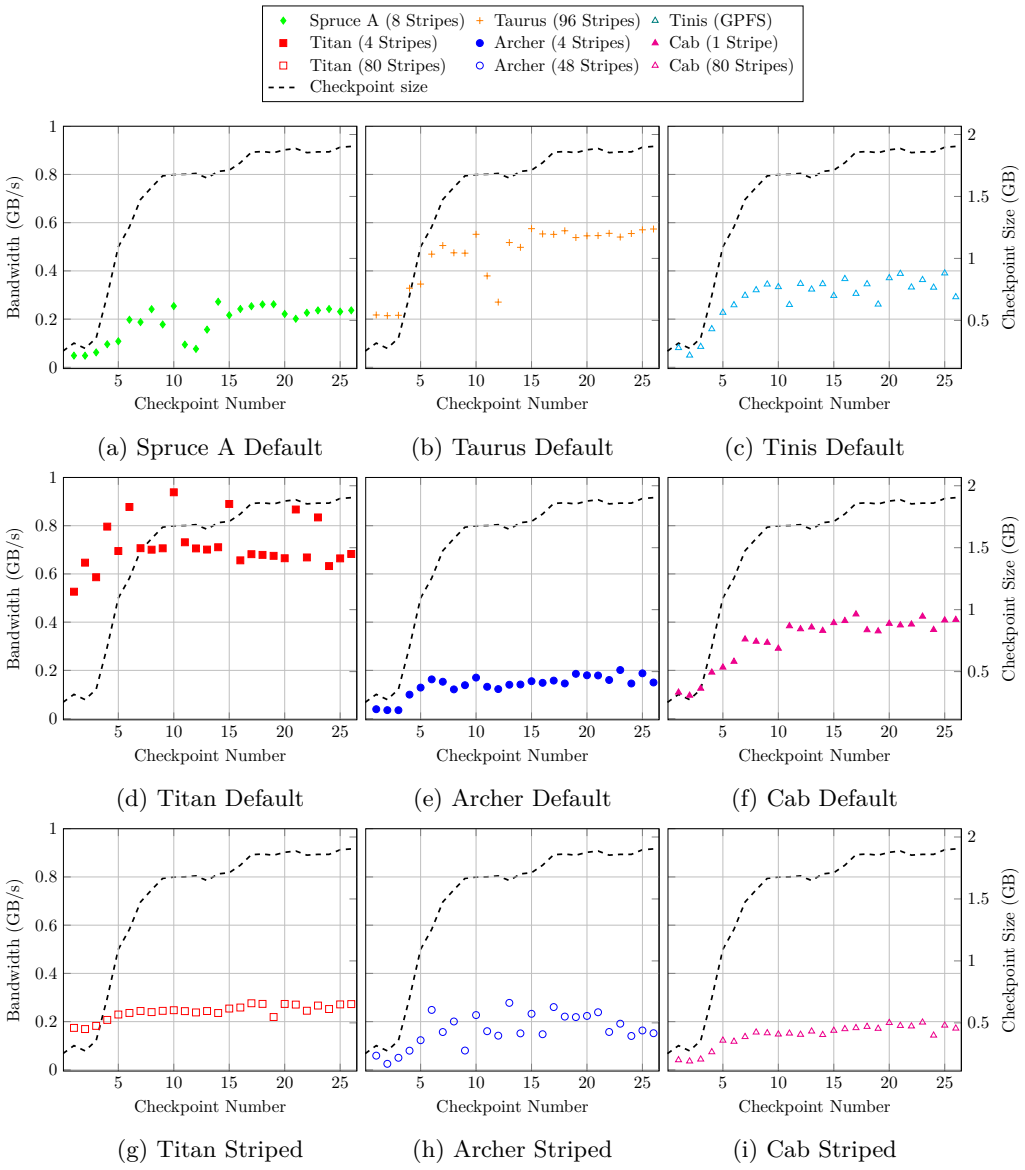


Figure 7.6: Perceived file bandwidth achieved for AWE01 workload B over the duration of a run using five compute nodes. Results are shown for Spruce A, Titan, Archer, Cab, Taurus, and Tinis platforms with standard and increased striping (stripe count is shown in brackets).

to Spruce comes from Tinis and the default configuration on Cab. Both of these performance profiles show a similar increase in throughput around the fifth checkpoint with almost identical range between the smallest and largest checkpoint bandwidths of 348.5 MB/s. Taurus is also able to match the general

profile of the run, notably experiencing the same fluctuations between the fifth and fifteenth checkpoints as the dataset experiences rapid growth. Furthermore, a greater bandwidth increase of 426.8 MB/s is noted but due to the initial performance starting off much greater than on Spruce, this only equates to a percentage improvement of 263%.

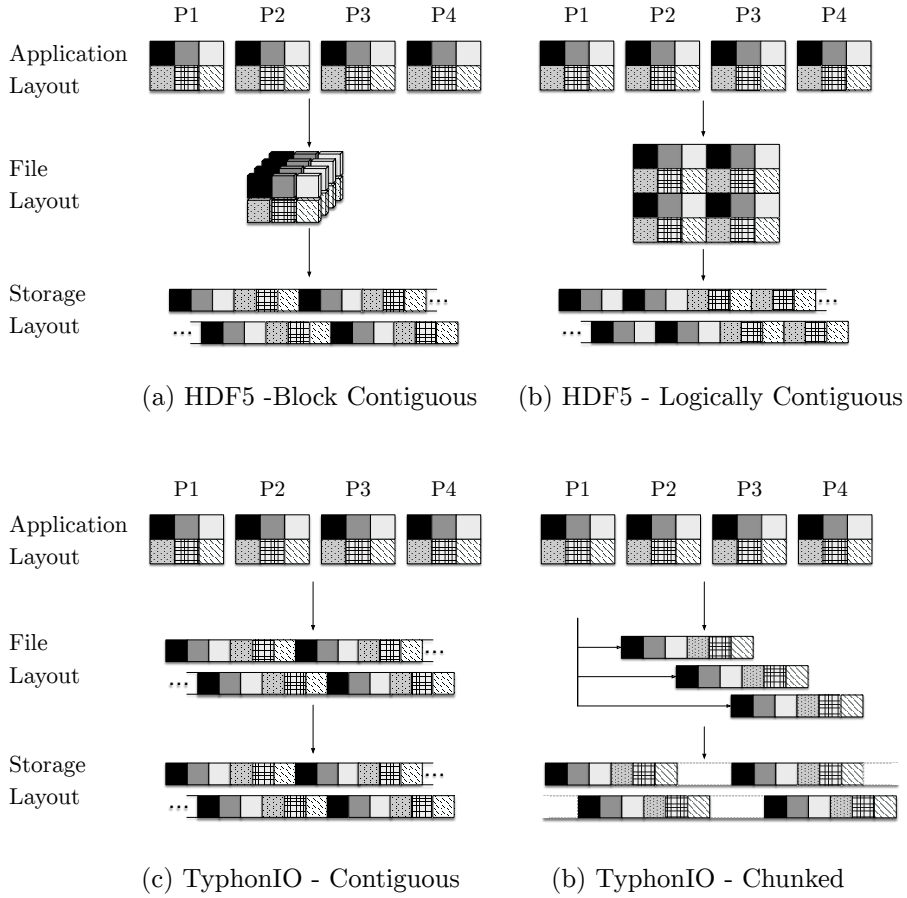
The three Lustre systems Titan, Archer, and Cab all repeat some of the characteristics seem previously with regards to the impact of their striping configuration on performance. Under the default conditions, the systems each achieve bandwidth numbers consistent with expectations drawn from their relative sizes and measurements taken up to this point. A previously unobserved volatility is noted on Titan which demonstrates a sort of two levelled performance profile with certain sized checkpoints causing the bandwidth to jump up to around 900 MB/s for it then to return to the baseline of between 600 MB/s and 700 MB/s. The increased striping configuration also introduced volatility on Archer, giving a higher peak for the same workload but fluctuating greatly between consecutive I/O phases.

7.3 I/O Library and File Strategy Comparisons

Preceding work in this chapter looks at the lower end of the I/O stack, namely the parallel file system and MPI-IO middleware layer. Application source code can be written to perform file I/O through MPI-IO calls (or indeed serially with calls directly to the POSIX layer); however, it is commonplace for production applications to make use of higher level libraries as their interface to storage, which is the case for the three applications focused on throughout this thesis. Higher level libraries are designed to abstract away complicated implementation details from application developers, however this does not remove the need for care to be taken ensuring I/O is carried out as efficiently as possible.

7.3.1 TyphonIO Efficiency

The AWE01, Bookleaf, and FLASH-IO applications each make use of the parallel routines in the HDF5 library; the first two applications do this through the TyphonIO library abstraction, while FLASH-IO makes calls to HDF5 directly. The mechanism used by HDF5 to transfer data in regions of memory to regions of a file in parallel is referred to as a *hyperslab*. A *hyperslab* selection can be both a logically contiguous collection of data elements or a regular pattern of blocks of data, this is true for data in memory and in file dataspaces.

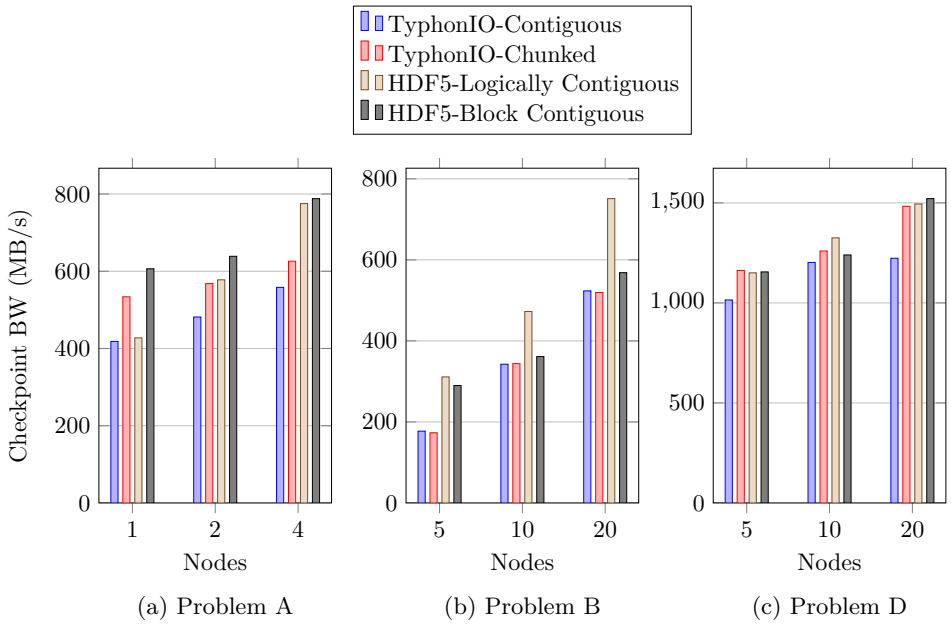


(a) HDF5 -Block Contiguous (b) HDF5 - Logically Contiguous

(c) TyphonIO - Contiguous (b) TyphonIO - Chunked

Figure 7.7: Layout of different data writing modes tested in MACSio.

TyphonIO imposes a hierarchical structured data model onto its output files. Hyperslabs are used to flatten application defined multidimensional data



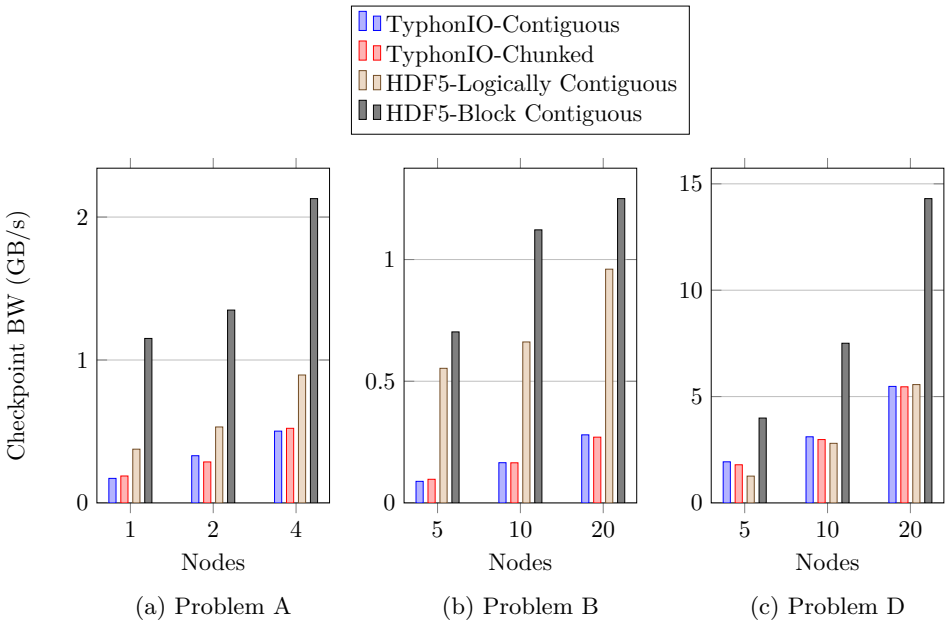
(a) Problem A (b) Problem B (c) Problem D

Figure 7.8: Checkpoint performance for AWE01 on Archer through standard TyphonIO, a TyphonIO-like layout using raw HDF5, and an alternative user-defined hyperslab arrangement also written using raw HDF5.

chunks into one dimensional arrays. These arrays can then be stored in either a contiguous or chunked dataset, depending on the library build. TyphonIO also stores chunk metadata to describe the logical structure of a dataset, allowing it to be re-assembled from the flat one dimensional dataspace. The design of the library delivers portability of output files and a degree of simplicity for users, however adding this layer on to an existing high level library limits some of the available flexibility.

An evaluation of TyphonIO and HDF5 has been carried out using the AWE01 application workloads on two platforms with Lustre file systems. For these experiments, the three I/O profiles were written to the parallel file system in four configurations: (a) TyphonIO writing contiguous datasets; (b) TyphonIO writing chunked datasets[2]; (c) HDF5 writing a contiguous dataset with block contiguity; (d) HDF5 writing a contiguous dataset with mesh contiguity. Figure 7.7 illustrates how data is translated from array chunks distributed across proces-

[2]Achieved by building with the −D_ENABLE_HDF5_CHUNK flag enabled.



(a) Problem A (b) Problem B (c) Problem D

Figure 7.9: Checkpoint performance for AWE01 on Quartz through standard TyphonIO, a TyphonIO-like layout using raw HDF5, and an alternative user-defined hyperslab arrangement also written using raw HDF5.

sors to a logical file dataspace, and ultimately its physical layout on disk for each of these configurations.

Figures 7.8 and 7.9 demonstrate library plugin performance of each workload at 1, 2 and 4 times the base problem definition on the Archer and Quartz systems. These experiments were carried out with the Lustre stripe count set to -1 to allow 1 MB blocks of data to be striped across all of the available OSTs, 48 on Archer and 36 on Quartz. Results show that on Archer, minor performance improvements can be obtained in some cases when bypassing TyphonIO and writing directly to one of the HDF5 configurations provided with MACSio. On Quartz, the performance difference between TyphonIO and the best performing HDF5 configuration is much greater with a speedup between 2.1× and 7.2× (average 4.4).

On Archer the use of non-contiguous chunked datasets in TyphonIO delivers equivalent performance to the contiguous strategy in the worst case. For the A and D inputs, switching to the chunked build actually improves checkpoint

bandwidth by up to 28% in the case of the smallest scale run for Problem A, representing a 115.5 MB/s increase in real terms. In this instance, the actual I/O requests issued to the file system are practically equivalent due largely to the middleware write aggregation that is occurring, The small improvements that are experienced in some cases can be attributed to the effects of chunk caching happening inside HDF5. When writing a chunked dataset, caching can be enabled for the entirety of a data chunk if the available cache is large enough. In the case of these experiments, the chunk cache has not been optimised and consequently only a subset of the chunk data can be cached to reduce write time. This accounts for the small and inconsistent improvements where this caching can benefit certain datasets but offers negligible benefits in other cases where the minor speedup is amortised over the time to write the rest of the dataset.

Performance results from Quartz differ notably from Archer, with the use of the 'block contiguous' HDF5 implementation achieving a peak bandwidth of 14,648 MB/s for the 20 node run on input D. Unlike Archer, no noticeable benefits are gained from the chunked datasets being turned on in TyphonIO and analysis of individual write operations shows that the two modes generate exactly the same set of operations at the middleware layer. Interestingly, the contiguous TyphonIO and block-contiguous HDF5 layouts both use I/O configurations mapping the same application data to the same layout on file, yet the raw HDF5 method notably outperforms its equivalent in TyphonIO as highlighted previously. In both cases, the transfer sizes generated by the libraries to store the bulk of the dataset are the same, the difference occurring in the volume and placement of metadata in addition to library management routines that are executed at the end of the checkpoint phase. This can be identified from a single metadata write issued to the file at finalise time which is preceded by a significant gap compared to all previous completed operations.

The logically contiguous layout in HDF5 generates the most distinct pattern of I/O behaviour out of the four tested. The complex hyperslab definitions used by this plugin to arrange the dataset in such as was as to match its logical layout

in the simulation causes the HDF5 library to perform a large amount of data aggregation on rank 0 in order to re-assemble the decomposed problem. What results is an all gather of data to rank 0, which essentially takes full control of outputting the entire dataset to file. The transfers generated by the aggregator alternate between 16 MB and 4.2 MB writes which are distributed to 17 and 5 OSTs respectively. Aggregation in this instance avoids contention from multiple ranks attempting to write to the same OST, however, as expected this method is outperformed by all ranks writing their own contiguous blocks should the number of OSTs be suitably large as to minimise collisions. Furthermore, for the largest of the three inputs the performance can be seen to be beginning to fall below that of both TyphonIO strategies.

7.3.2 N-M Parallel File Modes

Performance benchmarking carried out this chapter so far has focused on characteristics of the machine architecture, parallel file system and software stack. One of features of these experiments is that despite tuning efforts it can be difficult to influence how these components operate together to avoid contention at different stages. It is especially problematic when attempting these optimisations across a range of platforms providing different implementations and builds of the required components. To illustrate this point, the avoidance of file locking and contention in the shared file workloads we have considered requires the correct selection of high level library routines, middleware drivers, middleware hints, and parallel file system tuning parameters.

The simplest method of storing data in parallel from a software perspective has traditionally involved each rank writing to its own individual file (N-to-N), a strategy that was standard practice before the introduction of MPI-IO and is still used by some applications. Intuitively the use of independent files can help to avoid some performance limiting issues stemming from rank synchronisation and lock contention. Conversely, it is well understood that N-to-N based I/O lacks scalability due partially to bottlenecks at the file system metadata server [3]. In

this scenario there is still a requirement placed on the file system to distribute the files across the available storage targets to avoid ranks contending for the same resource.

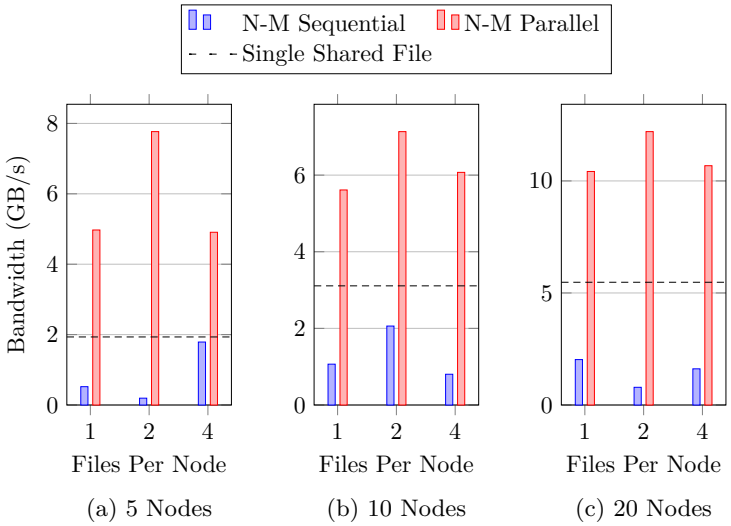


Figure 7.10: File bandwidth achieved for the AWE01 Problem D workload on Quartz when using Sequential and Parallel N-M access patterns.

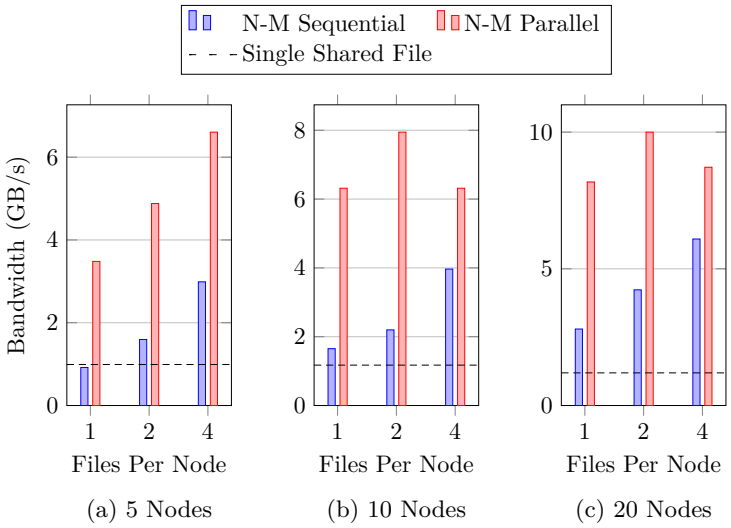
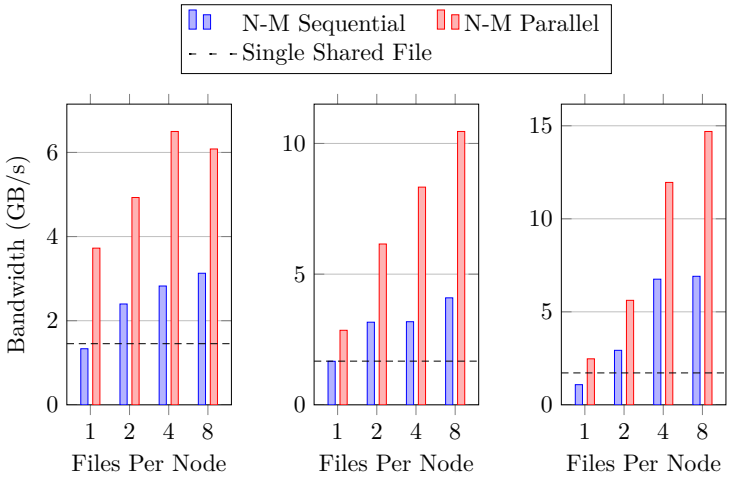


Figure 7.11: File bandwidth achieved for the AWE01 Problem D workload on Archer when using Sequential and Parallel N-M access patterns.

To investigate whether a mid point between these two strategies can be performant, a parallel N-M file strategy has been added to MACSio to operate

on top of the existing I/O plugins. This is motivated by the increasing adoption of systems featuring node local fast storage referred to as burst buffers. With these systems, libraries such as TyphonIO that are designed to write to a single shared file require different implementation inside an application to operate in this hybrid mode. Results from this approach are compared against an existing N-M approach used by the Silo library[62] where groups of ranks access multiple files sequentially.



(a) 8 Nodes (b) 16 Nodes (c) 32 Nodes

Figure 7.12: File bandwidth achieved for the AWE01 Problem D workload on Cab when using Sequential and Parallel N-M access patterns.

Figures 7.10 and 7.11 show the results of these runs on Quartz and Archer, for which observed checkpoint bandwidth is available at 5, 10, and 20 nodes with 1, 2 and 4 files per node (for clarity, 5 nodes with 1 file per node corresponds to 5 checkpoint files in total). As expected, both platforms demonstrate greater overall throughput when using the new parallel multiple-shared file writing approach. Specifically, the average speedup at each scale on Archer is measured at 3.02×, 3.00×, and 2.24×. Importantly, this approach outperforms the standard single shared file (SSF) method with a minimum speedup of 3.51× for the 5 node problem writing 1 file per node. In the best case, the observed checkpoint bandwidth increases by 837% from 1.2 GB/s to 8.4%.

On Quartz the greatest difference between the two approaches comes when writing to 2 files per node (18 ranks per file) with shared parallel achieving just over 7.7 GB/s to the sequential 198.8 MB/s. At each scale increasing the file count from two to four files per node causes a drop in throughput, of which the 5 node runs experience the biggest drop of 37%. Unlike Archer, the sequential approach on Quartz is unable to match the SSF performance in any of the tested configurations.

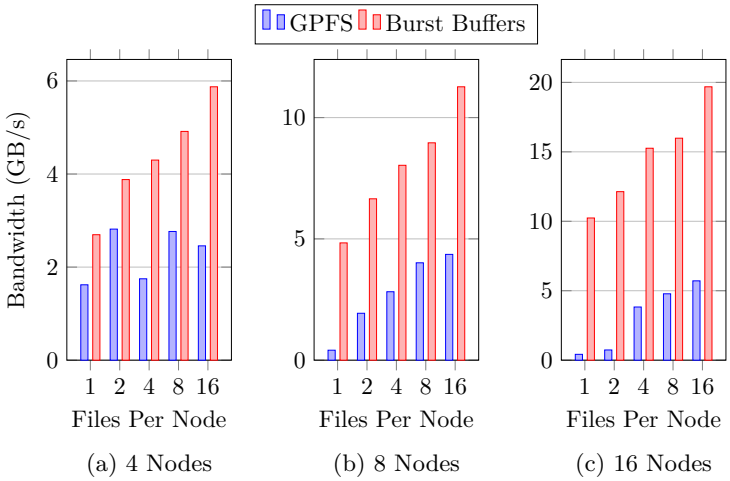


Figure 7.13: File bandwidth achieved for the AWE01 Problem D workload on Ray's GPFS file system and node-local burst buffers with N-M access patterns.

The same set of experiments is shown in Figure 7.12 on Cab, with the addition of an 8 file-per-node data point and the basic problem size starting at 8 nodes. Parallel shared file performance is on average double the original strategy, and peaks at 6.5 GB/s, 10.5 GB/s, 14.7 GB for the three problem sizes. SSF performance on Cab remains largely consistent as the simulation scales, as a result an improvement of 856% can be gained on 32 nodes writing to 8 files (4 ranks per node).

With the ability to checkpoint to decentralised storage being a necessity for node-local burst buffer systems, the new N-M file scheme has been evaluated on Ray using both GPFS and BB storage. In Figure 7.13, the results predictably see the burst buffer runs exceeding that of the GPFS file system by a minimum

of 37% (shown in Table 7.5) and delivering a peak checkpoint throughput of just under 20 GB/s when checkpoint data is spread across 16 files. The GPFS file system in Ray peaks at 5.7 GB/s for the largest problem size and file count, the mostly regular scaling for multiple files made possible by an update to GPFS disabling the directory level locking.

Files Per Node	4 Nodes	8 Nodes	16 Nodes
1	66%	1075%	2334%
2	38%	244%	1550%
4	146%	185%	299%
8	78%	123%	234%
16	139%	158%	245%

Table 7.5: Performance improvement of Ray burst buffers over GPFS file system under parallel N-M file modes.

Files Per Node	4 Nodes	8 Nodes	16 Nodes
1	2.70	4.83	10.24
2	1.94	3.33	6.07
4	1.07	2.01	3.81
8	0.61	1.12	2.00
16	0.37	0.70	1.23

Table 7.6: Bandwidth achieved per number of files on Ray when writing to burst buffers.

Writing to a greater number of files displays feasible scalability for the burst buffer setup, with these workloads not reaching the point of overloading the metadata capacity of the disk attempting to access multiple files concurrently. From a workflow perspective the generation of a large number of files may be undesirable due to added complexity when processing and visualising output files. For this reason, a trade-off for throughput against file count is of interest may be an important factor for both I/O library developers and application users. A summary of the 'bandwidth per file count' is listed in Table 7.6 showing that the performance scaling is not proportional to the increase in file count required, with the ratio decreasing such that some configurations achieve less than 1 GB/s per checkpoint file.

7.4 Discussion

An important observation to make about some of the results presented in this chapter (and in previous chapters) is how the performance of many of the experimental configurations used compares to theoretical and expected values. A large number of configurations are tested for parallel I/O, including variations in middleware, high level libraries, file system striping and parallel files modes. What is evident from many of these tests is that there are a number of factors that can be shown to increase, or decrease, the efficiency with which I/O is performed. An underlying trend throughout these experiments is the relative performance that is achieved for the different platforms when considered in the context of their expected throughput. To highlight this point, the file system deployed on Archer and the experimental results obtained from the machine can be compared.

Archer hosts a Lustre file system comprising 48 OSTs, which are expected to deliver 0.5 GB/s each. While there are a number of performance limiting factors that mean the total throughput on each OST is unlikely to ever be realised, an expected baseline for a well utilised Archer file system with little contention should be in the range of tens of GB/s. Comparing this to the figures obtained in Figure 7.8, the application workloads tested through MACSio achieve a maximum single shared file bandwidth of 1.5 GB/s. This represents one of the highest I/O throughput figures obtained from the platform, and many other results achieve considerably less than this. In terms of performance engineering, the gulf between what is theoretically possible on Archer and what has been measured is somewhat disappointing. Clearly the workloads that are used to represent scientific applications in these experiments are poorly suited to achieving an truly efficient return on I/O, which highlights some of the issues that individual applications will need to overcome to make best use of the available hardware. As a final point to note, the results shown in Figure 7.11 do achieve much greater file bandwidth performance figures when employing

a multiple shared file writing strategy. The peak throughput measured does reach around 10 GB/s, which represents a much more positive utilisation of the Archer file system. Achieving a greater throughput in this way suggests that the coordination to perform shared file I/O is not well implemented in the workloads tested earlier on in this work, and there are changes that can be made to better unlock the parallel I/O potential provided by HPC systems.

7.5 Summary

In this chapter, a comprehensive benchmarking study has been shown covering each level in the parallel I/O stack. However, it has been shown that optimisation efforts for different elements of this stack are not as effective as some literature suggests [9, 45, 97] when applied to a set of replicated production workloads due largely to variations in software environments and file system configuration. Similar findings have been reported in a small amount of literature [58], but these lessons have not been widely incorporated into the industry standard advice for performing efficient parallel I/O.

The previous chapter explored work towards building a flexible proxy application to rapidly deploy representative scientific workloads. MACSio has been shown to be capable of generating such workloads with a good degree of accuracy for the multi-physics applications studied. The work in this chapter provides a demonstration of how these portable workloads can be used to easily benchmark the performance that could be expected from production applications when deployed to a new system with a varied file system architecture. Furthermore, lightweight I/O profiling has been used in conjunction with these workloads to explore the sensitivity of different machine configurations to optimisation in the available software stack.

Using a variety of parameter sweeps aimed at the middleware and parallel file system components, the results in this chapter have shown that the system provided MPI-IO library on a number of production class systems are not built

with adequate support for efficient collective I/O operations. As a consequence, the expected performance that has been shown to be be achievable through striping data over parallel file systems cannot be easily obtained. In fact, some of the experiments conducted show that the absence of suitable software support can cause an application's checkpointing performance to suffer when distributing these workloads over a large number of OSTs.

Finally, this chapter is concluded with a top down look at the I/O stack. A comparison is made between four data layouts strategies controlled at the level of the HDF5 and TyphonIO parallel I/O libraries, demonstrating a degree of overhead introduced by the TyphonIO abstraction. Furthermore, careless use of hyperslabs to influence the file data layout in HDF5 can dramatically degrade the performance ceiling of a checkpoint phase.

A demonstration of a new parallel N-M file management scheme implemented by MACSio has been given, showing that this approach can offer significant improvement over the standard single shared file or Silo-like sequential strategies. Furthermore, introducing parallelism at this high level is able to insulate an application from a failure to perform contention avoiding optimisations further down the stack, ensuring that ranks attempts to access the same file regions or OSTs are minimised. Moreover, a strategy such as this is inevitably required to bridge the gap between PFS and burst buffer I/O strategies, hence a demonstration of this technique on a burst buffer enabled system has been conducted showing considerable performance improvements and highlighting this as a viable approach for I/O libraries going forward.

CHAPTER 8

Discussion and Conclusions

The work presented in this thesis outlines the current state of I/O in scientific applications, and offers some insight into how future developments can be made. Studies carried out at some of the largest supercomputing sites in the world have shown that the current generation of extreme scale machines are operating with a mean time between failures of just 9 hours, and so defensively writing data out to persistent storage is necessary to protect valuable simulation results [43].

A large proportion of the work that is done to improve I/O focuses on increasing the idealised peak bandwidth that can be achieved by a file system. For this reason, tools such as IOR have become industry standard as a simple way to generate I/O traffic to benchmark the performance ceiling [35]. However, a disconnect exists between the idealised performance of tools such as IOR and the real performance of large complex scientific applications. This thesis suggests that application focused benchmarking offers the solution of more accurate evaluations of future I/O systems and a greater amount of flexibility to facilitate prototyping and development of new I/O strategies.

Often, advice about how to maximise I/O performance is administered in the form of the 'rule of thumb' and little attention to detail is paid. In this thesis, it is demonstrated that unoptimised software environments and the complex interactions between layers in the I/O stack can make tuning attempts difficult to succeed in. This means that practically there is little incentive to perform such activities with large cumbersome code-bases that lack flexibility. Furthermore, the ability to measure accurate representations of the speedup offered by new I/O technologies on existing workloads can look to better advise supercomputing sites when procuring these expensive components.

Chapter 5 has demonstrated that data collected via lightweight profiling techniques can be used to extract the I/O characteristics of applications. A hydrodynamics mini-application and a large commercially sensitive production simulation are analysed and profiles of their I/O workloads are outlined. These I/O descriptions can inform users how their strategy selection plays out over the course of a simulation, and can be used to guide more accurate benchmarking and procurements.

Chapter 6 demonstrates the design and development of the MACSio I/O proxy application, which is able to generate application focused I/O workloads and execute these workloads through a variety of checkpoint strategies and high level libraries. A plugin for the TyphonIO library is developed, which enables the first portable and representative tool for benchmarking and re-engineering the library and its applications for future file systems. By drawing on the application profiling approach described in chapter 5, a validation of MACSio's ability to generate proxy I/O workloads for scientific applications is shown to be successful.

Finally, Chapter 7 combines the work of the preceding chapters and demonstrates the use of MACSio to conduct a range of benchmarking and optimisation tasks using the workload profiles of the studied scientific applications. A demonstration of the rapid deployment of MACSio to a number of platforms with file systems of contrasting architectures and scales is performed. Further, analysis is done on these workloads reacting to changes in configuration at the middleware and parallel file system layers, which is able to demonstrate that poor configuration renders some traditional I/O optimisation advice invalid for these workloads. With no modification to the parallel I/O software stack, settings intended to utilise greater file system parallelism are shown to degrade checkpointing performance in the test cases. These results were most notable when experimenting with collective buffering parameters, and in particular attempts to increase the number of aggregators for larger scale runs impacted the performance negatively due to the corresponding Lustre stripe count not being high

enough. This chapter also presents an evaluation of different file data layouts achieved through the HDF5 and TyphonIO libraries, showing that the use of hyperslabs as a mechanism for mapping simulation data to checkpoint files can lead to a loss of performance in cases where rearrangement of data blocks causes an aggregation bottleneck on the root process. The conclusion of this chapter demonstrates the performance of a multiple parallel file (N-M) I/O strategy that may be implemented in a library such a TyphonIO to future proof I/O against a shift to new storage layouts, in particular the use of fast node local burst buffers. This strategy also considerably outperforms the traditional shared file and multiple file approach used by Silo on current generation storage systems with no tuning of the rest of the I/O stack required.

8.1 Limitations

The primary limitation in this thesis is that experimentation and analysis focuses entirely on the write performance of parallel applications, with no consideration of improving read performance. In the type of application studied, initial states and configurations are read from a file at the initialisation stage after which point the flow of data generally is intermediate states being written out to storage for checkpointing and visualisation. Because the proportion of reads to writes over the course of a simulation is so skewed, write performance has been prioritised as the path to delivering the most notable improvements to an application overall. However, the tuning and selection of different I/O strategies featured in this work has been shown to reduce storage contention, a factor that will also improve the performance of parallel reads from the file system.

Another potential limitation is the choice of applications that have been studied for performance optimisation in Chapters 4 and 6. In particular, the AWE01 application is very specific in its pattern of usage and is not commonly used for large scale runs across the entirely of a machine. Nevertheless, it has been iden-

tified as one of the biggest I/O bottlenecks in the production workflow at AWE. Furthermore, as numerical methods and system architectures change the potential for the usage pattern similarly changes, meaning that lessons learned in the design and use of the I/O software environment at this scale will be valuable to the continued use of the application going forward.

A key limitation that was highlighted in Chapter 7 was the use of poorly configured MPI-IO middleware on multiple machines used for benchmarking. In particular, the lack of file system aware components in the MPI distribution prevented effective collective buffering to be performed. The MPI implementations used on all the experimental platforms was supplied as the system-wide default and built by administrators responsible for maintaining the user development environment. It is interesting to note that software provided to users would be unoptimised for the task of parallel I/O, and the remainder of the performance work carried out in the chapter was carried out with these unoptimised libraries. This way, a true representation of how systems will likely be typically used in gained.

A common limitation with I/O specific work is that by the nature of parallel file systems being shared by multiple users, it can difficult to avoid the effects of machine load and background noise. Each of the supercomputers used throughout this thesis were either production or publicly accessible research systems with other users sharing a proportion of the available resources. To account for the variability that can be caused by system noise, each of the experiments carried out were repeated multiple times, often at different times of day and spread out over a number of days or in some cases weeks. As the performance characteristics of interest in this thesis were specific to the I/O software environment and it's interaction with the storage system, the highest possible throughput measured is presented as this is likely the result demonstrating the truest performance of a system when removing external interference.

8.2 Future Work

The work in this thesis is focused primarily on the performance of current generation production system and the developments in I/O techniques that will be required to best utilise future systems, with a view to the procurement of these systems. It has been shown that some of the characteristics of real application I/O workloads lend themselves to poor performance, and a lack of understanding and continued optimisation efforts prevent this problem being successfully addressed. More recent efforts to update the libraries currently used to perform I/O, notably with tools such as ADIOS [55, 56], offer potential solutions to the problems hindering I/O optimisation work. In particular, ADIOS acts in a similar fashion to a domain specific language where a high level interface can be written into an application and control of the I/O routines handled by an external configuration file provided at runtime. The flexibility of this approach coupled with a tool like MACSio has a great deal of potential to benefit production applications, where optimisations can be discovered in MACSio and immediately implemented in the target code simply by supplying an I/O configuration file for users to adopt.

Machine architectures are currently undergoing a large shift in paradigm, and data storage is an important consideration. In particular, the widespread adoption of advanced architectures such as GPUs as the primary processing units in a node have meant that the density of compute power in a single node has increased. These devices have the potential to reduce the time to solution for a simulation, but only if the devices are suitably saturated with the data required to maximise the amount of parallel computation. The upshot of the increased compute density is therefore a similar increase in the density of data being processed and exported from each of the nodes in the system. Attempts to alleviate bottlenecks that halt the progress of a node when I/O is being performed have lead to the introduction of burst buffer devices [10, 54, 67, 93]. The purpose of a burst buffer is to provide fast SSD and NVMe based storage

to reduce the amount of time it takes for data to flow to and from the parallel file system. Competing architectures for deploying burst buffers have been put into production, and at the time of writing it has proved too early to identify which technique is likely to be accepted as the industry standard. On one hand, solutions such as Cray's DataWarp place a centralised collection of burst buffer nodes between the compute nodes and the file system to be shared by multiple nodes and jobs, where as node local burst buffers in machines like Summit [39] are dedicated for use by their host node only. A key consideration for the use of the second configuration is the incompatibility that decentralised storage devices has with the common practice of writing to shared files [64]. The N-M mode implemented in MACSio has shown potential as a solution for this issue, allowing a group of processors on each node to perform shared writes as they currently do, but to separate files stored locally on node. Bringing this capability down into the TyphonIO library to operate alongside the traditional method could offer an effective solution to the migration of I/O strategies to these modern and future storage architectures.

8.3 Final Remarks

This thesis has presented a set of steps towards representative and flexible I/O benchmarking and analysis. The existence of tools and techniques such as these will no doubt be vital in continuing to adapt I/O behaviour and tackle performance issues that arise with traditional storage systems and software libraries. Additionally, great pressure is placed on supercomputing sites to procure more advanced systems that meet the needs of their user base, meaning accurate benchmarking and analysis are fundamental to making correct decisions.

Ultimately, the explosion in complexity of system architectures, both general and storage specific, means that users and library developers must work to ensure their techniques continue to deliver I/O performance on the road to exascale systems.

Bibliography

[1] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. Hollingsworth, J. Saltz, and A. Sussman. Tuning the performance of I/O-intensive parallel applications. In *Proceedings of the Annual Workshop on I/O in Parallel and Distributed Systems, IOPADS*, IOPADS '96, pages 15–27, New York, NY, USA, 1996. ACM. ISBN 0-89791-813-4. doi: 10.1145/236017.236027. URL `http://0-doi.acm.org.pugwash.lib.warwick.ac.uk/10.1145/236017.236027`.

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: Tools for performance analysis of optimized parallel programs. *Concurrency Computation Practice and Experience*, 22(6):685–701, 2010. ISSN 15320626. doi: 10.1002/cpe.

[3] S. R. Alam, H. N. El-Harake, K. Howard, N. Stringfellow, and F. Verzelloni. Parallel I/O and the metadata wall. In *PDSW'11 - Proceedings of the 6th Parallel Data Storage Workshop, Co-located with SC'11*, pages 13–18. ACM, 2011. ISBN 9781450311038. doi: 10.1145/2159352.2159356.

[4] G. Almasi, S. Asaad, R. E. Bellofatto, H. R. Bickford, M. A. Blumrich, B. Brezzo, A. A. Bright, J. R. Brunheroto, J. G. Castanos, D. Chen, and Others. Overview of the IBM Blue Gene/P project. *IBM Journal of Research and Development*, 52(1-2):199–220, 2008.

[5] R. Alverson and D. Roweth. The gemini system interconnect. In *Proceedings - 18th IEEE Symposium on High Performance Interconnects, HOTI 2010*, pages 83–87. IEEE, 2010. ISBN 9780769542089. doi: 10.1109/HOTI.2010.23.

[6] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings - 1967*

Spring Joint Computer Conference, AFIPS 1967, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. doi: 10.1145/1465482.1465560.

[7] Argonne National Laboratory. Parallel I/O Benchmarking Consortium, 2011. URL `http://www.mcs.anl.gov/research/projects/pio-benchmark/`.

[8] Arm. Arm MAP. *Documentation at the URL: {https://static.docs.arm.com/101136/1822/userguide-forge.pdf}*, 2019. URL `https://www.arm.com/products/development-tools/server-and-hpc/forge/map`.

[9] 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 *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, SC '13, page 68, New York, NY, USA, 2013. ACM, ACM. ISBN 9781450323789. doi: 10.1145/2503210.2503278. URL `http://0-doi.acm.org.pugwash.lib.warwick.ac.uk/10.1145/2503210.2503278`.

[10] W. Bhimji, D. Bard, M. Romanus, D. Paul, A. Ovsyannikov, B. Friesen, M. Bryson, J. Correa, G. K. Lockwood, V. Tsulaia, S. Byna, S. Farrell, D. Gursoy, C. S. Daley, V. Beckner, B. V. Straalen, D. Trebotich, C. Tull, G. Weber, N. J. Wright, K. Antypas, and Prabhat. Accelerating Science with the NERSC Burst Buffer Early User Program. *Proceedings of the 2016 Cray User Group*, 2016. URL `https://cug.org/proceedings/cug2016_proceedings/includes/files/pap162.pdf`.

[11] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak. IntelÂő Omni-path Architecture: Enabling Scalable, High Performance Fabrics. In *Proceedings - 2015 IEEE*

23rd Annual Symposium on High-Performance Interconnects, HOTI 2015, pages 1–9. IEEE, 2015. ISBN 9781467391603. doi: 10.1109/HOTI.2015.22.

[12] D. Boehme, T. Gamblin, D. Beckingsale, P. T. Bremer, A. Gimenez, M. Legendre, O. Pearce, and M. Schulz. Caliper: Performance Introspection for HPC Software Stacks. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, pages 550–560. IEEE Press, 2017. ISBN 9781467388153. doi: 10.1109/SC.2016.46.

[13] J. Borrill, L. Oliker, J. Shalf, H. Shan, and A. Uselton. HPC global file system performance analysis using a scientific-application derived benchmark. *Parallel Computing*, 35(6):358–373, 2009. ISSN 01678191. doi: 10.1016/j.parco.2009.02.002.

[14] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger. The disksim simulation environment version 4.0 reference manual (cmu-pdl-08-101). *Parallel Data Laboratory*, page 26, 2008.

[15] F. Cappello. Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. *International Journal of High Performance Computing Applications*, 23(3):212–226, 2009. ISSN 10943420. doi: 10.1177/1094342009106189.

[16] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley. 24/7 characterization of petascale I/O workloads. In *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, pages 1–10. IEEE, 2009. ISBN 9781424450121. doi: 10.1109/CLUSTR.2009.5289150.

[17] P. H. Carns, K. Harms, R. Latham, and R. B. Ross. Performance Analysis of Darshan 2.2.3 on the Cray XE6 Platform. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2012.

[18] J. H. Chen, a. Choudhary, B. De Supinski, M. Devries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki,

R. Sankaran, S. Shende, and C. S. Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science and Discovery*, 2(1):15001, 2009. ISSN 17494680. doi: 10.1088/1749-4699/2/1/015001.

[19] J. Cope, N. Liu, S. Lang, P. H. Carns, C. D. Carothers, and R. B. Ross. CODES: Enabling Co-Design of Multi-Layer Exascale Storage Architectures. In *Workshop on Emerging Supercomputing Technologies 2011 (WEST 2011)*, volume 2011, pages 303–312, 2011.

[20] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snirt, B. Traversat, and P. Wong. Overview of the MPI-IO parallel I/O interface. In *Input/Output in Parallel and Distributed Computer Systems*, pages 127–146. Springer, 1996.

[21] T. Cortes and J. Labarta. HRaid: A Flexible Storage-system Simulator. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 772–778, 1999.

[22] F. De Carlo, D. Gürsoy, F. Marone, M. Rivers, D. Y. Parkinson, F. Khan, N. Schwarz, D. J. Vine, S. Vogt, S. C. Gleber, S. Narayanan, M. Newville, T. Lanzirotti, Y. Sun, Y. P. Hong, and C. Jacobsen. Scientific data exchange: A schema for HDF5-based storage of raw and analyzed data. *Journal of Synchrotron Radiation*, 21(6):1224–1230, 2014. ISSN 16005775. doi: 10.1107/S160057751401604X.

[23] A. C. de Melo. The New Linux 'perf' Tools. In *Linux Kongress*, volume 18, 2010.

[24] P. M. Dickens and J. Logan. A high performance implementation of MPI-IO for a Lustre file system environment. *Concurrency Computation Practice and Experience*, 22(11):1433–1449, 2010. ISSN 15320626. doi: 10.1002/cpe.1491.

[25] P. M. Dickens and R. Thakur. Improving collective I/O performance using threads. In *Proceedings of the International Parallel Processing Symposium, IPPS*, pages 38–45. IEEE, 1999.

[26] J. Dickson. Parallel I/O Libraries. In *1st Symposium of the Centre for Computational Plasma Physics, University of Warwick*, November 2015.

[27] J. Dickson. Investigating Application I/O. In *JOWOG 34 Meeting, Atomic Weapons Establishment*, June 2016.

[28] J. Dickson. Replicating I/O Behaviour in Production Applications. In *JOWOG 34 Meeting, Los Alamos National Laboratory*, June 2017.

[29] J. Dickson. I/O Performance Analysis with Proxy Applications. In *JOWOG 34 Applied Computer Science Meeting, Sandia National Laboratory*, February 2018.

[30] J. Dickson, A. Herdman, S. Maheswaran, S. a. Wright, J. a. Herdman, and S. a. Jarvis. MINIO : an I/O benchmark for investigating high level parallel libraries. In *27th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*, pages 5–6, September 2015. ISBN Dickson, James, Maheswaran, Satheesh, Wright, Steven A., Herdman, J. A. and Jarvis, Stephen A. (2015) MINIO : an I/O benchmark for investigating high level parallel libraries. In: 27th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SCâĂŹ15), Austin, Texas, USA, 15-20 Nov 2015 (In Press). URL `http://wrap.warwick.ac.uk/73143/20/WRAP_ExtendedAbstract%2BConferenceposter.pdf`.

[31] J. Dickson, S. Wright, S. Maheswaran, A. Herdman, D. Harris, M. C. Miller, and S. Jarvis. Enabling Portable I / O Analysis of Commercially Sensitive HPC Applications Through Workload Replication. In *Cray User Group 2017 Proceedings*, pages 7–12, May 2017.

[32] J. Dickson, S. Wright, S. Maheswaran, A. Herdman, M. C. Miller, and S. Jarvis. Replicating HPC I/O workloads with proxy applications. In *Proceedings of PDSW-DISCS 2016: 1st Joint International Workshop on Parallel Data Storage and Data Intensive Scalable Computing Systems - Held in conjunction with SC16: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 13–18, September 2017. ISBN 9781509052165. doi: 10.1109/PDSW-DISCS.2016.007.

[33] J. Dongarra, H. W. Meuer, E. Strohmaier, and H. Simon. The TOP 500 List, 2000.

[34] J. J. Dongarra, P. Luszczek, and A. Petite. The LINPACK benchmark: Past, present and future. *Concurrency Computation Practice and Experience*, 15(9):803–820, 2003. ISSN 15320626. doi: 10.1002/cpe.728.

[35] S. Ed and L. Berkeley. Using IOR to analyze the I/O performance for HPC. *Cray User Group 2007 Proceedings*, 2010.

[36] N. El-Sayed and B. Schroeder. Reading between the lines of failure logs: Understanding how HPC systems fail. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 1–12. IEEE, 2013. ISBN 9781467364713. doi: 10.1109/DSN.2013.6575356.

[37] N. El-Sayed and B. Schroeder. Checkpoint/restart in practice: When simple is better. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 84–92. IEEE, 2014.

[38] EPCC I/O Benchmarking applications. benchio, 2018. `https://github.com/EPCCed/benchio`, last accessed on September 2, 2018.

[39] M. Feldman. Oak Ridge readies Summit supercomputer for 2018 debut. *Top500. org, http://bit. ly/2ERRFr9*, 2017.

[40] S. a. Fineberg, O. Wong, B. Nitzberg, and C. Kuszmaul. PMPIO - a portable implementation of MPI-IO. In *Frontiers of Massively Parallel Computation - Conference Proceedings*, pages 188–195. IEEE, 1996.

[41] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson. An overview of the HDF5 technology suite and its applications. In *ACM International Conference Proceeding Series*, pages 36–47. ACM, 2011. ISBN 9781450306140. doi: 10.1145/1966895.1966900.

[42] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction, SIGPLAN 1982*, volume 17, pages 120–126. ACM, 1982. ISBN 0897910745. doi: 10.1145/800230.806987.

[43] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari. Failures in large scale systems: Long-term measurement, analysis, and implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017*, page 44. ACM, 2017. ISBN 9781450351140. doi: 10.1145/3126908.3126937.

[44] J. L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, May 1988.

[45] R. Hedges, B. Loewe, T. McLarty, and C. Morrone. Parallel file system testing for the lunatic fringe: The care and feeding of restless I/O power users. In *Proceedings - Twenty -second IEEE/Thirteenth NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 3–17. IEEE Computer Society, 2005. ISBN 0769523188. doi: 10.1109/MSST.2005.22.

[46] S. Herbein, D. H. Ahn, D. Lipari, T. R. Scogland, M. Stearman, M. Grondona, J. Garlick, B. Springmeyer, and M. Taufer. Scalable I/O-aware job scheduling for burst buffer enabled HPC clusters. In *HPDC 2016 - Proceedings of the 25th ACM International Symposium on High-Performance*

Parallel and Distributed Computing, pages 69–80. ACM, 2016. ISBN 9781450343145. doi: 10.1145/2907294.2907316.

[47] M. Howison, Q. Koziol, D. Knaak, J. Mainzer, and J. Shalf. Tuning HDF5 for Lustre File Systems. In *Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS10)*, 2010.

[48] A. a. Hwang, I. a. Stefanovici, and B. Schroeder. Cosmic rays don't strike twice: Understanding the nature of DRAM errors and the implications for system design. In *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, volume 47, pages 111–122. ACM, 2012. ISBN 9781450307598. doi: 10.1145/2150976.2150989.

[49] W. V. C. Ii, G. Gibson, M. Holland, L. N. Reilly, and J. Zelenka. RAIDframe : A Rapid Prototyping Tool for RAID Systems. Technical Report August, Carnegie-Mellon University, 1997.

[50] Intel Developer Zone. Intel VTune Amplifier. *Documentation at the URL: https://software.intel.com/en-us/intel-vtune-amplifier-xe-support/documentation*, 2018.

[51] T. Jones, R. Mark, J. Martin, J. May, E. Pierce, and L. Stanberry. An MPI-IO Interface to HPSS. Technical report, Lawrence Livermore National Lab., CA (United States), 1996.

[52] Kogge, Peter \emphet. al. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15:278, 2008.

[53] P. Kovatch, M. Ezell, and R. Braby. The malthusian catastrophe is upon us! Are the largest HPC machines ever up? In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence*

and Lecture Notes in Bioinformatics), volume 7156 LNCS, pages 211–220. Springer, 2012. ISBN 9783642297397. doi: 10.1007/978-3-642-29740-3-25.

[54] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership-class storage systems. In *IEEE Symposium on Mass Storage Systems and Technologies*, pages 1–11. IEEE, 2012. ISBN 9781467317450. doi: 10.1109/MSST.2012.6232369.

[55] J. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *CLADE - Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments 2008, CLADE'08*, pages 15–24. ACM, 2008. ISBN 9781605581569. doi: 10.1145/1383529.1383533.

[56] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, metadata rich IO methods for portable high performance IO. In *IPDPS 2009 - Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium*, volume 00, pages 1–10, 2009. ISBN 9781424437504. doi: 10.1109/IPDPS.2009.5161052. URL `doi.ieeecomputersociety.org/10.1109/IPDPS.2009.5161052`.

[57] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf. Managing variability in the IO performance of petascale storage systems. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010*, pages 1–12. IEEE, 2010. ISBN 9781424475575. doi: 10.1109/SC.2010.32.

[58] J. Logan and P. Dickens. Towards an understanding of the performance of MPI-IO in Lustre file systems. In *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, volume Proceedings of the 2008

IEEE International Conference on Cluster Computing, ICCC 2008, pages 330–335. IEEE, 2008. ISBN 9781424426409. doi: 10.1109/CLUSTR.2008.4663791.

[59] X. Ma, M. Winslett, J. Lee, and S. Yu. Improving MPI-IO output performance with active buffering plus threads. In *Proceedings - International Parallel and Distributed Processing Symposium, IPDPS 2003*, pages 10—-pp. IEEE, 2003. ISBN 0769519261. doi: 10.1109/IPDPS.2003.1213165.

[60] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *Proceedings - 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014*, pages 610–621. IEEE, 2014. ISBN 9781479922338. doi: 10.1109/DSN.2014.62.

[61] J. Meza, Q. Wu, S. Kumar, and O. Mutlu. A large-scale study of flash memory failures in the field. *Performance Evaluation Review*, 43(1):177–190, 2015. ISSN 01635999. doi: 10.1145/2796314.2745848.

[62] M. Miller. Silo: A Genrral-Purpose API and Scientific Database. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2015.

[63] M. C. Miller. Design & Implementation of MACSio. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA (United States), 2015.

[64] S. Mittal and J. S. Vetter. A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, 2016. ISSN 10459219. doi: 10.1109/TPDS.2015.2442980.

[65] E. Molina-Estolano, C. Maltzahn, J. Bent, and a. Brandt. Building a parallel file system simulator. In *Journal of Physics: Conference Series*,

volume 180, page 12050. IOP Publishing, 2009. doi: 10.1088/1742-6596/180/1/012050.

[66] W. D. Norcott and D. Capps. Iozone filesystem benchmark, 2003.

[67] X. Ouyang, S. Marcarelli, and D. K. Panda. Enhancing checkpoint performance with staging IO and SSD. In *Proceedings - 2010 International Workshop on Storage Network Architecture and Parallel I/Os, SNAPI 2010*, pages 13–20. IEEE, 2010. ISBN 9780769540252. doi: 10.1109/SNAPI.2010.10.

[68] C. M. Patrick, S. Son, and M. Kandemir. Comparative evaluation of overlap strategies with study of I/O overlap in MPI-IO. *Operating Systems Review (ACM)*, 42(6):43–49, 2008. ISSN 01635980. doi: 10.1145/1453775.1453784.

[69] D. a. Patterson, G. Gibson, and R. H. Katz. *A Case for Redundant Arrays of Inexpensive Disks (RAID)*, volume 17. ACM, 1988. doi: 10.1145/971701.50214.

[70] J.-P. Prost, R. Treumann, R. Hedges, B. Jia, and A. Koniges. MPI-IO/GPFS, an optimized implementation of MPI-IO on top of GPFS. In *Supercomputing, ACM/IEEE 2001 Conference*, pages 17–17. IEEE, 2001. doi: 10.1145/582034.582051.

[71] R. Rabenseifner, A. E. Koniges, J.-P. Prost, and R. Hedges. The Parallel Effective I/O Bandwidth Benchmark: b_eff_io. *Parallel I/O for Cluster Computing*, pages 107–132, 2004. URL `http://www.hlrs.de/people/rabenseifner/publ/cpj_b_eff_io_nov19.pdf`.

[72] a. F. Rodrigues, E. CooperBalls, B. Jacob, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, and P. Rosenfeld. The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):37, 2011. ISSN 01635999. doi: 10.1145/1964218.1964225.

[73] C. San-Lucas and C. L. Abad. Towards a fast multi-tier storage system simulator. In *2016 IEEE Ecuador Technical Chapters Meeting, ETCM 2016*, pages 1–5. IEEE, 2016. ISBN 9781509016297. doi: 10.1109/ETCM.2016.7750836.

[74] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. De Supinski, and S. Matsuoka. Design and modeling of a non-blocking checkpointing system. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, page 19. IEEE Computer Society Press, 2012. ISBN 9781467308069. doi: 10.1109/SC.2012.46.

[75] F. Schmuck and R. Haskin. {GPFS}: A Shared-Disk File System for Large Computing Clusters. *Proceedings of the First Conference on File and Storage Technologies*, pages 231–244, 2002. URL `http://portal.acm.org/citation.cfm?id=1083349`.

[76] B. Schroeder, E. Pinheiro, and W. D. Weber. DRAM errors in the wild: A large-scale field study. In *Communications of the ACM*, volume 54, pages 100–107. ACM, 2011. doi: 10.1145/1897816.1897844.

[77] P. Schwan. Lustre: Building a File System for 1,000-node Clusters. *Proceedings of the Linux Symposium*, pages 401–409, 2003. doi: 10.1.1.2.456.

[78] S. S. Shende and A. D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20 (2):287–311, 2006. ISSN 10943420. doi: 10.1177/1094342006064482.

[79] M. Snir, R. W. Wisniewski, J. a. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. a. Chien, P. Coteus, N. a. Debardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen. Addressing failures in exascale computing. *International Journal of*

High Performance Computing Applications, 28(2):129–173, 2014. ISSN 17412846. doi: 10.1177/1094342014522573.

[80] S. Snyder, P. Carns, R. Latham, M. Mubarak, R. Ross, C. Carothers, B. Behzad, H. V. T. Luu, S. Byna, and Others. Techniques for Modeling Large-scale {HPC} {I/O} Workloads. In *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*, page 5. ACM, 2015.

[81] S. Snyder, P. Carns, K. Harms, R. Latham, and R. Ross. Performance Evaluation of Darshan 3.0.0 on the Cray XC30. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2016. URL `http://www.osti.gov/scitech/`.

[82] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi. Feng Shui of supercomputer memory positional effects in DRAM and SRAM faults. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, page 22. ACM, 2013. ISBN 9781450323789. doi: 10.1145/2503210.2503257.

[83] K. Stockinger and E. Schikuta. ViMPIOS, a "truly" portable MPI-IO implementation. In *Proceedings - 8th Euromicro Workshop on Parallel and Distributed Processing, EURO-PDP 2000*, pages 4–9. IEEE, 2000. ISBN 0769505007. doi: 10.1109/EMPDP.2000.823386.

[84] R. Thakur, E. Lusk, and W. Gropp. Users guide for ROMIO: A high-performance, portable MPI-IO implementation. Technical report, Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, 1997.

[85] R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Annual Workshop on I/O in Parallel and Distributed Systems, IOPADS*, pages 23–32. ACM, 1999.

[86] J. E. Thornton. The CDC 6600 Project. *Annals of the History of Computing*, 2(4):338–348, 1980. ISSN 01641239. doi: 10.1109/MAHC.1980.10044.

[87] A. Turner, X. Guo, D. Sloan-Murphy, J. Rodriguez Herrera, C. Maynard, and B. Lawrence. Parallel I/O Performance Benchmarking and Investigation on Multiple HPC Architectures. Technical report, EPCC, The University of Edinburgh, 2017.

[88] UK Mini App Consortium. TyphonIO. `https://github.com/UK-MAC/typhonio`, 2013. last accessed on September 2, 2018.

[89] UK Miniapp Consurtium. Bookleaf Unstructured Lagrangian Hydro Miniapp. `https://github.com/UK-MAC`, 2016. last accessed on September 2, 2018.

[90] J. Vetter and C. Chambreau. mpiP: Lightweight, Scalable MPI Profiling Using mpiP. `http://gec.di.uminho.pt/Discip/MInf/cpd1415/PCP/MPI/mpiPLightweightScalableMPIProfiling.pdf`, 2005. last accessed on September 2, 2018.

[91] K. Wang, X. Zhou, H. Chen, M. Lang, and I. Raicu. Next generation job management systems for extreme-scale ensemble computing. In *HPDC 2014 - Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, pages 111–114. ACM, 2014. ISBN 9781450327480. doi: 10.1145/2600212.2600703.

[92] K. Wang, M. Lang, X. Zhou, B. McClelland, K. Qiao, and I. Raicu. Towards scalable distributed workload manager with monitoring-based weakly consistent resource stealing. In *HPDC 2015 - Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 219–222. ACM, 2015. ISBN 9781450335508. doi: 10.1145/2749246.2749249.

[93] T. Wang, S. Oral, Y. Wang, B. Settlemyer, S. Atchley, and W. Yu. BurstMem: A high-performance burst buffer system for scientific applications.

In *Proceedings - 2014 IEEE International Conference on Big Data, IEEE Big Data 2014*, pages 71–79. IEEE, 2015. ISBN 9781479956654. doi: 10.1109/BigData.2014.7004215.

[94] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[95] J. Wilkes. The Pantheon storage-system simulator. *Hewlett-Packard Laboratories Technical Report HPL-SSP-95-114*, 1995.

[96] M. Wittmann, G. Hager, T. Zeiser, and G. Wellein. Asynchronous MPI for the Masses. *arXiv preprint arXiv:1302.4280*, 2013. URL `http://arxiv.org/abs/1302.4280`.

[97] S. a. Wright and S. a. Jarvis. Quantifying the Effects of Contention on Parallel File Systems. In *Proceedings - 2015 IEEE 29th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2015*, pages 932–940. IEEE, 2015. ISBN 0769555101. doi: 10.1109/IPDPSW.2015.8.

[98] Z. Xiao, Z. Xiao-Nan, and C. Jian-Quan. SIM-array: A flexible storage system simulator. In *Proceedings - International Symposium on Computer Science and Computational Technology, ISCSCT 2008*, volume 2, pages 221–225. IEEE, 2008. ISBN 9780769534985.

[99] M. Yang, R. E. McGrath, and M. Folk. HDF5 - A high performance data format for earth science. In *85th AMS Annual Meeting, American Meteorological Society - Combined Preprints*, pages 1161–1166, 2005.

[100] J. W. Young. A First Order Approximation to the Optimum Checkpoint Interval. *Communications of the ACM*, 17(9):530–531, 1974. ISSN 15577317. doi: 10.1145/361147.361115.

[101] W. Yu, J. S. Vetter, and H. S. Oral. Performance characterization and optimization of parallel I/O on the cray XT. In *IPDPS Miami 2008 - Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium, Program and CD-ROM*, pages 1–11. IEEE, 2008. ISBN 9781424416943. doi: 10.1109/IPDPS.2008.4536277.

[102] Z. Zhou, X. Yang, D. Zhao, P. Rich, W. Tang, J. Wang, and Z. Lan. I/O-aware batch scheduling for petascale computing systems. In *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, volume 2015-October, pages 254–263. IEEE, 2015. ISBN 9781467365987. doi: 10.1109/CLUSTER.2015.45.

[103] M. Zingale. FLASH I/O Benchmark Routine. `http://www.ucolick.org/~zingale/flash_benchmark_io/`, 2001. last accessed on September 2, 2018.

APPENDIX A

Profiling Multi-physics I/O Workloads

MPIIO_INDEP_OPENS	MPIIO_COLL_OPENS
MPIIO_INDEP_READS	MPIIO_INDEP_WRITES
MPIIO_COLL_READS	MPIIO_COLL_WRITES
MPIIO_SPLIT_READS	MPIIO_SPLIT_WRITES
MPIIO_NB_READS	MPIIO_NB_WRITES
MPIIO_SYNCS	MPIIO_HINTS
MPIIO_VIEWS	MPIIO_MODE
MPIIO_BYTES_READ	MPIIO_BYTES_WRITTEN
MPIIO_RW_SWITCHES	MPIIO_MAX_READ_TIME_SIZE
MPIIO_MAX_WRITE_TIME_SIZE	MPIIO_SIZE_READ_AGG_0_100
MPIIO_SIZE_READ_AGG_100_1K	MPIIO_SIZE_READ_AGG_1K_10K
MPIIO_SIZE_READ_AGG_10K_100K	MPIIO_SIZE_READ_AGG_100K_1M
MPIIO_SIZE_READ_AGG_1M_4M	MPIIO_SIZE_READ_AGG_4M_10M
MPIIO_SIZE_READ_AGG_10M_100M	MPIIO_SIZE_READ_AGG_100M_1G
MPIIO_SIZE_READ_AGG_1G_PLUS	MPIIO_SIZE_WRITE_AGG_0_100
MPIIO_SIZE_WRITE_AGG_100_1K	MPIIO_SIZE_WRITE_AGG_1K_10K
MPIIO_SIZE_WRITE_AGG_10K_100K	MPIIO_SIZE_WRITE_AGG_100K_1M
MPIIO_SIZE_WRITE_AGG_1M_4M	MPIIO_SIZE_WRITE_AGG_4M_10M
MPIIO_SIZE_WRITE_AGG_10M_100M	MPIIO_SIZE_WRITE_AGG_100M_1G
MPIIO_SIZE_WRITE_AGG_1G_PLUS	MPIIO_ACCESS1_ACCESS
MPIIO_ACCESS2_ACCESS	MPIIO_ACCESS3_ACCESS
MPIIO_ACCESS4_ACCESS	MPIIO_ACCESS1_COUNT
MPIIO_ACCESS2_COUNT	MPIIO_ACCESS3_COUNT
MPIIO_ACCESS4_COUNT	MPIIO_FASTEST_RANK
MPIIO_FASTEST_RANK_BYTES	MPIIO_SLOWEST_RANK
MPIIO_SLOWEST_RANK_BYTES	MPIIO_F_OPEN_TIMESTAMP
MPIIO_F_READ_START_TIMESTAMP	MPIIO_F_WRITE_START_TIMESTAMP
MPIIO_F_READ_END_TIMESTAMP	MPIIO_F_WRITE_END_TIMESTAMP
MPIIO_F_CLOSE_TIMESTAMP	MPIIO_F_READ_TIME
MPIIO_F_WRITE_TIME	MPIIO_F_META_TIME
MPIIO_F_MAX_READ_TIME	MPIIO_F_MAX_WRITE_TIME
MPIIO_F_FASTEST_RANK_TIME	MPIIO_F_SLOWEST_RANK_TIME
MPIIO_F_VARIANCE_RANK_TIME	MPIIO_F_VARIANCE_RANK_BYTES

Table A.1: Darshan MPIIO counters

APPENDIX B

Application Workload Replication

Node Count	MACSio File 1 Write Time (s)	MACSio File 2 Write Time (s)	Bookleaf File 1 Write Time (s)	Bookleaf File 2 Write Time (s)
1	3.8	4.6	3.6	5.0
2	15.8	17.3	16.4	16.8
4	26.6	20.6	20.8	21
8	35.2	33.2	33	34.2
16	50.4	45	50	45.2
32	92.4	85.2	79.6	83.5
64	137.8	137.8	134.2	129.8

Table B.1: I/O timings for Bookleaf and MACSio replicated checkpoints on Archer.

Node Count	MACSio File 1 Write Time (s)	MACSio File 2 Write Time (s)	Bookleaf File 1 Write Time (s)	Bookleaf File 2 Write Time (s)
1	112.9	95.3	90.4	119.1
2	572.7	815.2	749.6	751.9
4	2043.5	1671.6	1693.8	1761.6
8	5613.1	5745.9	5466.1	5536.9
16	15289.6	13495.4	15881.8	14335.8
32	57565.4	53755.2	50650.9	52087.0
64	172951.6	173648.8	165415.9521	158804.0

Table B.2: Cumulative I/O timings for Bookleaf and MACSio replicated checkpoints on Archer.

Node Count	MACSio Op Time 1 (s)	MACSio Op Time 2 (s)	Bookleaf Op Time 1 (s)	Bookleaf Op Time 2 (s)
1	1.0	0.7	0.7	1.6
2	2.3	3.2	3.1	3.0
4	5.4	4.3	4.0	4.4
8	10.4	10.6	7.4	8.7
16	21.1	16.8	14.9	13.7
32	33.7	29.5	28.34	26.6
64	46.8	51.1	51.5	44.3

Table B.3: Slowest I/O operation time for Bookleaf and MACSio replicated checkpoints on Archer.

Node Count	MACSio File 1 Write Time (s)	MACSio File 2 Write Time (s)	Bookleaf File 1 Write Time (s)	Bookleaf File 2 Write Time (s)
1	17.4	15.4	14.5	16.6
2	29.1	33.7	35.1	34.8
4	41.4	44.0	43.5	44.9
8	63.9	77.3	78.4	60.3
16	102.8	95.5	104.4	96.2
32	166.9	155.7	169.3	176.2
64	268.0	288.2	278.2	296.5

Table B.4: I/O timings for Bookleaf and MACSio replicated checkpoints on Tinis.

Node Count	MACSio File 1 Write Time (s)	MACSio File 2 Write Time (s)	Bookleaf File 1 Write Time (s)	Bookleaf File 2 Write Time (s)
1	807.3	574.4	656.7	699.8
2	1459.0	1672.7	1593.1	1336.8
4	3716.0	3214.1	2920.3	3405.1
8	11012.4	8515.3	9971.6	9828.9
16	30316.9	20823.8	27675.2	25932.1
32	114610.8	91481.4	86406.6	106643.6
64	319602.3	263225.7	276152.2	314260.7

Table B.5: Cumulative I/O timings for Bookleaf and MACSio replicated checkpoints on Tinis.

Node Count	MACSio Op Time 1 (s)	MACSio Op Time 2 (s)	Bookleaf Op Time 1 (s)	Bookleaf Op Time 2 (s)
1	3.9	3.2	3.3	3.1
2	6.0	5.4	5.6	5.2
4	9.4	8.0	7.9	8.1
8	19.0	16.8	15.8	18.3
16	39.1	34.0	27.7	25.6
32	58.9	52.4	51.4	50.6
64	81.1	93.7	97.8	85.1

Table B.6: Slowest I/O operation time for Bookleaf and MACSio replicated checkpoints on Tinis.

APPENDIX C

I/O Performance Benchmarking and Optimisation

Node Count	Standard Problem Independent BW (MB/s)	Standard Problem Independent BW (MB/s)	Large Problem Independent BW (MB/s)	Large Problem Collective BW (MB/s)
1	34.02	234.86	579.39	792.09
2	9.58	114.43	252.94	523.49
4	6.75	126.13	255.03	543.97
8	4.67	102.53	196.41	645.46
16	3.01	127.49	103.57	535.34
32	2.58	114.74	82.13	612.52
64	1.56	106.69	55.54	671.80
128	0.86	92.92	46.46	590.51

Table C.1: Perceived checkpoint bandwidth for the Bookleaf workload on Archer.

Node Count	Standard Problem Independent BW (MB/s)	Standard Problem Independent BW (MB/s)	Large Problem Independent BW (MB/s)	Large Problem Collective BW (MB/s)
1	29.33	30.02	1266.13	905.97
2	23.81	27.43	1079.68	1955.84
4	27.33	26.31	1064.44	1067.85
8	22.46	24.61	1147.25	1034.05
16	20.40	20.60	988.28	752.88
32	20.45	18.56	1710.74	666.23
64	20.18	13.73	1117.66	547.95
128	18.41	11.12	960.03	512.42

Table C.2: Perceived checkpoint bandwidth for the Bookleaf workload on Quartz.

Node Count	Standard Problem Independent BW (MB/s)	Standard Problem Independent BW (MB/s)	Large Problem Independent BW (MB/s)	Large Problem Collective BW (MB/s)
1	119.16	847.87	129.13	917.31
2	56.90	908.32	104.78	572.34
4	107.45	708.41	186.29	638.96
8	136.16	659.41	291.16	663.52
16	244.33	677.85	492.85	696.12
32	344.40	787.75	497.05	690.28
64	396.44	893.83	532.03	868.70
128	237.24	744.32	621.23	996.88

Table C.3: Perceived checkpoint bandwidth for the FLASH-IO workload on Archer.

Node Count	Standard Problem Independent BW (MB/s)	Standard Problem Independent BW (MB/s)	Large Problem Independent BW (MB/s)	Large Problem Collective BW (MB/s)
1	1657.42	1557.22	1587.11	1651.09
2	2663.41	1996.93	3113.66	2596.28
4	4999.55	4003.84	6003.96	5163.16
8	7419.09	6201.30	6072.44	5818.89
16	4865.48	5429.00	6012.32	5855.46
32	2965.73	5209.04	4560.15	6053.40
64	2673.23	3469.22	3382.78	3998.92
128	3790.13	1742.09	5358.94	3751.30

Table C.4: Perceived checkpoint bandwidth for the FLASH-IO workload on Quartz.

Node Count	Standard Problem Independent BW (MB/s)	Standard Problem Independent BW (MB/s)	Large Problem Independent BW (MB/s)	Large Problem Collective BW (MB/s)
1	476.37	717.23	6303.23	4375.44
2	76.42	65.70	2571.65	2420.95
4	33.67	29.09	1976.40	1629.26
8	20.12	19.07	947.67	1081.76
16	14.07	10.94	567.33	638.17
24	12.83	10.87	443.37	447.11

Table C.5: Perceived checkpoint bandwidth for the Bookleaf workload on Ray.

Node Count	Standard Problem Independent BW (MB/s)	Standard Problem Independent BW (MB/s)	Large Problem Independent BW (MB/s)	Large Problem Collective BW (MB/s)
1	2101.17	302.60	3265.06	539.97
2	330.62	283.44	599.04	493.10
4	416.41	327.78	754.79	619.83
8	511.52	426.35	920.03	979.34
16	671.62	514.32	1355.97	1010.06
24	869.87	655.44	1495.75	1142.98

Table C.6: Perceived checkpoint bandwidth for the FLASH-IO workload on Ray.

Node Count	Collective Buffering Nodes						
	1	2	4	8	16	32	64
1	199.98	255.59	126.52	157.04	128.63	123.09	174.84
2	96.05	115.28	179.48	62.82	79.27	86.71	63.85
4	119.27	141.68	79.02	133.79	85.85	145.90	60.23
8	128.91	148.20	133.18	43.03	29.14	48.28	38.09
16	117.06	138.43	99.29	51.34	64.73	54.21	47.49
32	121.21	147.20	158.67	43.17	71.82	94.65	81.55
64	127.17	109.23	97.46	59.40	54.10	72.12	79.26
128	105.35	84.82	84.38	45.30	47.83	53.89	38.03

Table C.7: Write bandwidth for the Bookleaf workload running on Archer with different collective buffering node counts.

Node Count	Collective Buffering Nodes						
	1	2	4	8	16	32	64
1	636.83	824.29	867.23	723.39	467.51	540.25	756.23
2	574.60	713.83	842.38	353.97	678.99	704.25	648.51
4	528.48	796.49	751.91	742.91	476.45	631.47	711.67
8	644.08	766.58	613.42	103.21	123.77	145.19	149.19
16	592.72	895.18	651.85	119.85	153.62	146.86	209.72
32	566.06	731.89	797.69	157.00	165.21	249.98	288.26
64	614.92	826.64	525.29	188.34	170.74	235.56	372.06
128	661.17	850.01	735.52	217.16	113.33	231.41	453.75

Table C.8: Write bandwidth for the FLASH-IO workload running on Archer with different collective buffering node counts.

Machine	Default BW (MB/s)	Increased Stripe Count BW (MB/s)
Spruce	636.44	-
Titan	539.19	400.61
Archer	145.96	123.46
Cab	227.26	202.71
Taurus	-	187.7267512
Tinis	347.88	-

Table C.9: Perceived checkpoint bandwidth for the AWE01 Problem A workload running on the Spruce A, Titan, Archer, Cab, Taurus and Tinis systems.

Machine	Default BW (MB/s)	Increased Stripe Count BW (MB/s)
Spruce	498.38	-
Titan	1087.38	834.31
Archer	540.32	674.92
Cab	720.97	564.56
Taurus	-	837.18
Tinis	1025.17	-

Table C.10: Perceived checkpoint bandwidth for the AWE01 Problem A workload running on the Spruce A, Titan, Archer, Cab, Taurus and Tinis systems.

Checkpoint Number	System					
	Spruce A	Titan	Archer	Cab	Taurus	Tinis
0	49.00	489.54	27.04	94.79	162.11	65.85
1	49.87	538.96	39.92	111.48	222.41	82.87
2	49.56	662.28	36.01	98.30	219.49	51.01
3	63.50	600.27	35.76	127.96	221.11	87.95
4	98.20	815.28	101.81	196.36	336.09	162.91
5	110.81	711.59	131.77	217.71	353.71	232.78
6	202.22	898.74	166.64	242.33	480.76	264.71
7	192.15	723.61	156.58	337.14	517.33	305.04
8	247.08	717.37	124.11	327.50	486.41	328.42
9	182.16	723.16	141.86	322.92	484.96	352.77
10	260.22	961.01	174.17	297.75	565.02	341.67
11	96.61	748.75	135.20	393.12	388.31	265.98
12	78.36	722.68	125.44	380.40	277.10	355.27
13	160.28	717.84	143.87	387.94	530.00	331.31
14	278.55	727.67	144.88	373.49	509.71	354.29
15	221.52	911.53	158.85	406.53	588.95	304.35
16	247.94	672.03	151.96	415.34	566.62	375.84
17	259.68	698.18	161.44	443.34	564.78	313.02
18	267.32	695.24	149.20	376.32	579.99	353.76
19	267.88	691.23	190.18	371.44	551.20	267.65
20	226.87	680.97	183.92	403.45	558.58	379.55
21	206.32	887.98	183.23	396.74	559.51	397.91
22	231.72	684.39	164.01	400.51	569.09	340.07
23	242.01	854.30	206.31	433.91	553.34	371.66
24	247.72	647.65	149.09	377.29	568.28	339.22
25	236.70	680.42	192.26	417.89	584.07	399.87
26	242.14	699.13	153.41	420.08	587.49	298.80

Table C.11: Perceived checkpoint bandwidth for the AWE01 Problem B workload running on the Spruce A, Titan, Archer, Cab, Taurus and Tinis systems.

Problem	Nodes	HDF5 - Logically Contiguous	HDF5 - Block Contiguous	TyphonIO - Contiguous	TyphonIO - Chunked	HDF5 Speedup Over TyphonIO
A	1	427.54	606.35	418.37	533.83	1.13
	2	577.77	638.72	481.51	568.14	1.12
	4	775.15	787.97	558.22	625.96	1.25
B	5	311.16	289.99	177.24	173.24	1.76
	10	472.69	361.24	342.44	343.96	1.37
	20	751.18	568.42	523.46	519.51	1.44
D	5	1150.30	1155.04	1014.78	1162.38	0.99
	10	1325.31	1239.83	1202.39	1259.46	1.05
	20	1334.59	1321.20	1223.15	1482.87	0.90

Table C.12: Checkpoint performance for AWE01 on Archer through standard TyphonIO, a TyphonIO-like logically contiguous layout using raw HDF5, and an block contiguous hyperslab arrangement in HDF5.

Problem	Nodes	HDF5 - Logically Contiguous	HDF5 - Block Contiguous	TyphonIO - Contiguous	TyphonIO - Chunked	HDF5 Speedup Over TyphonIO
A	1	385.24	1178.56	176.00	193.17	6.10
	2	544.32	1381.70	338.38	293.81	4.08
	4	916.54	2180.09	514.52	534.69	4.08
B	5	566.39	719.57	90.90	99.55	7.23
	10	677.29	1148.89	169.35	168.92	6.78
	20	983.46	1280.29	286.53	276.79	4.47
D	5	1291.78	4086.45	1980.07	1837.67	2.06
	10	2869.79	7687.39	3184.56	3051.34	2.41
	20	5697.35	14648.22	5609.42	5590.70	2.61

Table C.13: Checkpoint performance for AWE01 on Quartz through standard TyphonIO, a TyphonIO-like logically contiguous layout using raw HDF5, and an block contiguous hyperslab arrangement in HDF5.

Files Per Node	Sequential N-M			Parallel N-M		
	5 Nodes	10 Nodes	20 Nodes	5 Nodes	10 Nodes	20 Nodes
1	941.08	1691.84	2865.08	3565.91	6468.04	8373.29
2	1634.09	2251.57	4332.94	4997.57	8135.71	10238.04
4	3059.95	4060.92	6236.89	6762.82	6467.09	8924.05

Table C.14: File bandwidth achieved for the AWE01 Problem D workload on Archer when using Sequential and Parallel N-M access patterns.

Files Per Node	Sequential N-M			Parallel N-M		
	5 Nodes	10 Nodes	20 Nodes	5 Nodes	10 Nodes	20 Nodes
1	533.32	1091.58	2075.61	5091.72	5747.33	10671.90
2	198.83	2110.02	808.92	7953.82	7306.43	12493.40
4	1831.74	822.10	1653.14	5025.47	6217.43	10935.81

Table C.15: File bandwidth achieved for the AWE01 Problem D workload on Quartz when using Sequential and Parallel N-M access patterns.

Files Per Node	Sequential N-M			Parallel N-M		
	5 Nodes	10 Nodes	20 Nodes	5 Nodes	10 Nodes	20 Nodes
1	1365.80	1707.05	1110.67	3814.42	2921.09	2533.60
2	2454.94	3237.81	3001.13	5046.70	6301.10	5752.49
4	2892.71	3256.90	6916.97	6654.19	8529.21	12242.21
8	3203.04	4191.97	7075.50	6228.06	10709.53	15046.39

Table C.16: File bandwidth achieved for the AWE01 Problem D workload on Cab when using Sequential and Parallel N-M access patterns.

Files Per Node	GPFS			Burst Buffers		
	5 Nodes	10 Nodes	20 Nodes	5 Nodes	10 Nodes	20 Nodes
1	1658.7	421.2	430.6	2760.7	4950.8	10483.0
2	2884.0	1980.0	753.0	3974.0	6812.0	12424.9
4	1790.8	2889.3	3920.3	4402.9	8226.8	15625.4
8	2832.0	4110.7	4893.8	5033.6	9175.3	16365.7
16	2515.46	4466.0	5848.4	6015.5	11540.5	20155.4

Table C.17: File bandwidth achieved for the AWE01 Problem D workload written through parallel N-M to GPFS and burst buffers on Ray.