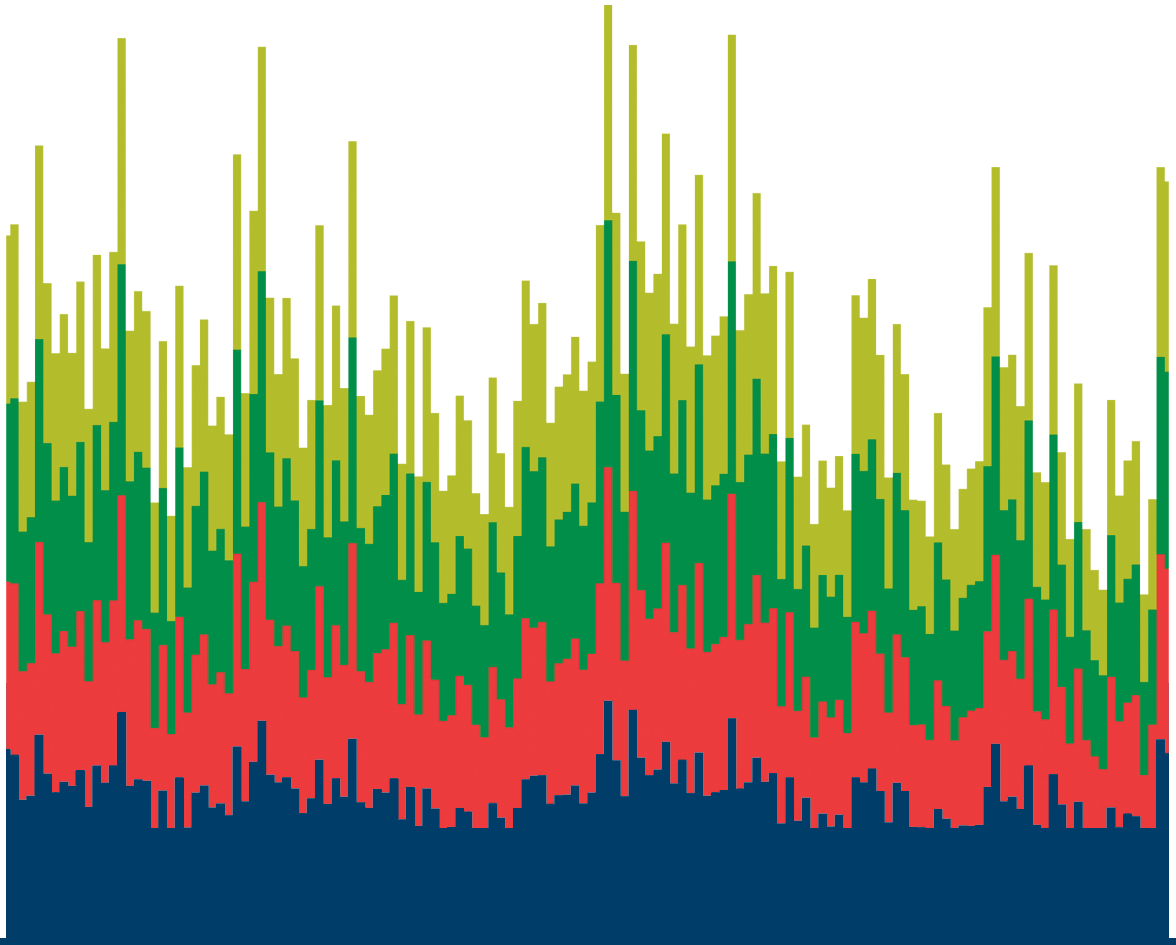


Analysis of I/O Requirements of Scientific Applications

Salem El Sayed Mohamed

IAS Series
Band / Volume 37
ISBN 978-3-95806-344-0

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
Forschungszentrum

Schriften des Forschungszentrums Jülich
Reihe IAS Band / Volume 37

Forschungszentrum Jülich GmbH
Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC)

Analysis of I/O Requirements of Scientific Applications

Salem El Sayed Mohamed

Schriften des Forschungszentrums Jülich
Reihe IAS

Band / Volume 37

ISSN 1868-8489

ISBN 978-3-95806-344-0

Bibliografische Information der Deutschen Nationalbibliothek.
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der
Deutschen Nationalbibliografie; detaillierte Bibliografische Daten
sind im Internet über http://dnb.d-nb.de abrufbar.

Herausgeber und Vertrieb:	Forschungszentrum Jülich GmbH Zentralbibliothek, Verlag 52425 Jülich Tel.: +49 2461 61-5368 Fax: +49 2461 61-6103 zb-publikation@fz-juelich.de www.fz-juelich.de/zb
Umschlaggestaltung:	Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH
Druck:	Grafische Medien, Forschungszentrum Jülich GmbH
Copyright:	Forschungszentrum Jülich 2018

Schriften des Forschungszentrums Jülich
Reihe IAS, Band / Volume 37

D 468 (Diss., Wuppertal, Univ., 2017)

ISSN 1868-8489
ISBN 978-3-95806-344-0

Persistent Identifier: urn:nbn:de:0001-2018071801

The complete volume is freely available on the Internet on the Jülicher Open Access Server (JuSER)
at www.fz-juelich.de/zb/openaccess

This is an Open Access publication distributed under the terms of the Creative Commons Attribution License 4.0,
which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Abstract

Analysis of I/O Requirements of Scientific Applications

The advance in both computation and data storage size in High Performance Computing (HPC) has not been matched by a similar advance in I/O connections. Emerging technologies have promised to overcome this gap. These could require scientific applications to change their I/O behaviour to benefit from the improvements. Therefore, a deeper analysis of applications' I/O behaviour on modern HPC systems is required. This work defines I/O analysis criteria by which I/O behaviour can be systematically evaluated. Using the defined criteria a large set of collected I/O logs on a petascale Blue Gene/P installation, namely JUGENE, was analysed. To further the understanding of I/O architectures and their effect on I/O, a simplified parametric I/O model was developed. Results show that the implemented model has a comparable I/O behaviour to that of JUGENE, and therefore is used to evaluate new I/O technologies.

Acknowledgements

Thanks to Prof. Dirk Pleiter and Prof. Matthias Bolten for their generous support and guidance, without which this work would not have been possible. Thanks to my colleagues at the Juelich Supercomputing Center for allowing me to be part of the team and their helpful notes and tips. Thanks to my family and friends, who have not failed to help and support.

But most of all, I'm grateful to my wife for her everlasting support, courage and endurance that replaced the hardships with blessings.

To
Maria & Carolina

Contents

List of Figures

List of Tables

Abbreviations

HPC	**H**igh **P**erformance **C**omputing
I/O	**I**nput/**O**utput
MPI	**M**essage **P**assing **I**nterface
POSIX	**P**ortable **O**perating **S**ystem **I**nterface for **U**nix
NSD	**N**etwork **S**hared **D**isk
SAN	**S**torage **A**rea **N**etwork
IOP/IOPs	**I**nput/Output **OP**eration(**s**)
IOPS	**I**nput/Output **O**perations **P**er **S**econd
HDD	**H**ard **D**isk **D**rive
SSD	**S**olid **S**tate **D**isk
GPFS	**G**eneral **P**arallel **F**ile**S**ystem
CIOD	**C**ontrol and **I**nput/**O**utput **D**aemon
CNK	**C**ompute **N**ode **K**ernal
CN	**C**ompute **N**ode
ION	**I**nput/**O**utput **N**ode

Chapter 1

Introduction

1.1 Motivation

High Performance Computing (HPC) systems have advanced from the tera-scale to the peta-scale and are advancing onwards to the exa-scale. These systems are being supported by large ever growing storage units that already can store on the order of tens of petabytes. However, the advance in both computation and storage size has not been met with a matching increase in the input/output (I/O) performance. This threatens the overall gain from employing such large scale HPC systems for the benefit of scientific applications. The problem is further complicated by HPC systems using an ever growing number of parallel computing components. These have to be serviced by the system's available I/O [1]. The increasing number of parallel components employed in HPC systems leads to possible higher failure rates. To prevent data and progress loss, applications have to save their intermediate status, further straining the I/O systems [2]. Many HPC systems have a high cost attached to its networking infrastructure. These to some extend dictate the size and performance of the I/O. As a result simply extending or increasing the I/O system is in many cases limited by overall system cost.

To provide HPC systems with the needed access to data, application's I/O requests have to traverse various software and hardware components. These can be thought of as a stacked layer of units, where each serves specific and interchanging functionalities. The I/O stack has evolved over a long period of time. As I/O transcends into a bottleneck and due to constant technology advancement, some I/O stack layers require an update.

Emerging I/O concepts and technologies promise upgrading and improving I/O performance. These provide changes to one or more layer in the I/O stack. The performance increase comes from better understanding the underlying systems and the tasks they need to fulfil to run a more optimized I/O. While some of these optimizations require scientific applications to change their implementations, others do not. Changing an application's implementation is usually connected to large efforts that scientists and application developers have to exert. Meanwhile, optimizations that do not require application changes focus on system or I/O library improvements.

The benefit from improving I/O can depend on paring the correct I/O behaviour with a more thoughtful I/O system configuration. Future I/O architectures should employ a co-design approach, where design decisions are made on the basis of a good understanding of application I/O behaviour. Many studies have been undertaken to provide a better understanding of the I/O system and it's functionalities, these include [3], [4], [5], [6], [7], [8], [9] and many more. As for the I/O behaviour of applications, there is an overall absence for understanding application's interactions with the underlying storage system [10]. Since I/O system designers have no control over the I/O behaviour of applications, many are forced to implement I/O systems on the basis of speculations [11]. As a result, many I/O systems are designed without full information on the workload they will have to serve [12].

There are significant efforts ongoing in the HPC community to monitor and analyse the I/O behaviour of applications such as [13], [10], [12] and [14]. Understanding application I/O and its impact on the I/O stack layers and the benefit of available I/O optimizations, is key in improving I/O performance of current peta-scale and future exa-scale systems. Knowledge of the application's I/O behaviour can be used in relation to available I/O optimizations to determine the extend of benefit applications will perceive, while considering the needed effort for implementation. Therefore, modern computing centers should analyse the I/O behaviour of the scientific applications running on their infrastructure. Configuration, I/O optimizations and upgrades of the I/O system should be done with reference to the understanding obtained from such I/O analysis.

1.2 Research Goals

Due to the complexity of the I/O stack and the massive parallelism integrated into modern HPC systems, the I/O behaviour is a complex compound of various observables. Analysing the I/O behaviour can be made easier by designing an analysis map. Therefore, this study proceeds by providing a scientific method for systematic analysis of I/O behaviour by outlining a set of I/O criteria. These are constructed and selected with contemplation on both modern scientific applications and I/O architecture. While some approach the study of I/O behaviour by investigating the I/O of a single or a limited group of applications [15][16], others opt for mass application I/O behaviour analysis [14]. Therefore, the I/O criteria are designed to provide both possibilities. The I/O criteria also attempts being applicable to many modern I/O architectures as well as to different methods of I/O measuring techniques.

The conclusions made according to any analysis method chosen is strongly dependent on the I/O measuring procedure. To provide a case study for applying the I/O criteria, a mass analysis on a large quantity of filesystem logs for a modern HPC system is performed. This tests the I/O criteria in the wild and allows feeding the information gained back into their design. The case study also gains system and application developers insights into scientific application's perceived I/O patterns created on a modern I/O system.

Collecting and analysing I/O behaviour of applications can feed into the design cycle of I/O systems that are or will be used by current or future HPC systems. One method for using such I/O behaviour information is to create simulations of I/O systems and experiment with I/O stack changes that could benefit the application. This study provides a simulation of a modern HPC I/O system with it's underlying components. By feeding the model with the collected I/O information, I/O architectural changes are investigated.

The approach taken by defining I/O criteria, using them to investigate I/O behaviour and simulating I/O stack changes, could greatly benefit I/O system and application developers.

1.3 HPC I/O Architecture

Fulfilling an application's requirement for storage access, takes the I/O request on a path through multiple components of the I/O system. The path taken is referred to as the I/O stack. The layers involved have the task to address and store the data, as well as providing an I/O interface. Applications are constantly growing in both computing and data requirements. As a response HPC systems had to grow in computational power, parallelism and data storage size and access performance. This lead to the need of further developing and improving the individual I/O stack layers.

A representation of the I/O stack layers that are part of a traditional I/O system are given in Fig. 1.1. Applications create I/O requests using either a high level I/O library or an I/O interface such as POSIX I/O or MPI-IO. The I/O interface translates the I/O requests to the underlying filesystem. These have the task of managing the storage infrastructure and the data stored on them. Since HPC systems employ a large scale storage infrastructure, local filesystems are not sufficient to take on the task. Therefore, the I/O system employs parallel filesystems that can span over multiple servers and many storage devices. The main difference to a standard desktop I/O stack is in the scale and size of the storage infrastructure and the parallel filesystem that is needed to operate it.

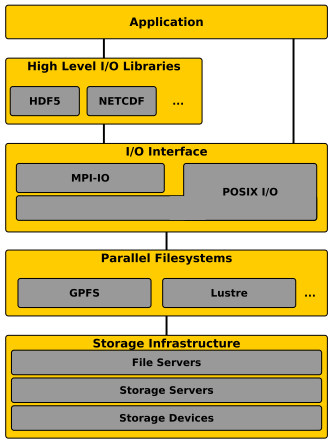


FIGURE 1.1: HPC traditional I/O stack

HPC systems are steadily growing in computational power. To achieve this, the systems incorporate large scale parallelism, employing multi-core multi-thread processors on a large number of nodes. To accommodate these changes the I/O systems had to scale their bandwidth and IOPS. Bandwidth is defined as the rate of data transfer that a system can achieve in a given time, and carries the unit data quantity over time[1]. I/O Operations Per Second (IOPS) is defined as the number of I/O Operations (IOPs) that a system can perform in one second.

The I/O stack as described here implies a separation between the computation and I/O system. Both need to implement or execute different parts of the I/O stack. However, the parallel filesystem considered can be accessed from different computing systems without requiring redesign, as long as these use the correct interface. It is therefore possible to distinguish between three terms: HPC or compute system, I/O system and I/O subsystem. The compute system includes the compute nodes, possibly specialized I/O nodes and an internal and external network for communication. An I/O system refers to the parallel filesystem along with the storage infrastructure attached. It dictates the interface the compute system will have to use for data access. The I/O subsystem refers to the union of the computation and I/O system components that are used in the fulfilment of an I/O request. For example, the external network connections of the compute system are also considered part of the I/O subsystem. The use of these terms allow differentiating between updates that would require changing the I/O system versus these that require changes to both the compute and I/O system, i.e. I/O subsystem.

Storage infrastructure

The storage infrastructure contains the physical storage units, storage servers and file servers as well as switches and routers that provide data access. Most storage systems use Hard Disk Drive (HDD) as storage devices. These are densely packed in storage servers and could be accessed through file servers. Due to HDD's limited bandwidth and IOPS, the storage infrastructure has to use in the order of thousands of HDDs in parallel to deliver adequate performance. These are placed into many storage servers. The increasing number of HDDs can result in reliability and cost issues. New storage

[1]Bandwidth has the unit B/s for bytes per second and bps for bits per second. The standard used here dictates that, 1KiB (Kibibyte) = 1024B, 1MiB (Mebibyte) = 1024KiB, 1GiB (Gibibyte) = 1024KiB, 1TiB (Tebibyte) = 1024GiB and 1PiB (Pebibyte) = 1024TiB

device technologies such as Solid State Disks (SSD) are being introduced, offering better bandwidth and IOPS. However, due to their high cost per storage unit and occasionally reliability issues they are as of now not capable of fully replacing HDDs. Some systems attempt combining the low cost per storage unit of HDD with the high performance of SSDs in so called hybrid systems [17].

Parallel filesystems

To manage the large number of storage servers, large scale parallel filesystems are used. Contrary to local filesystems, parallel filesystems can span multiple servers and offer access to a large number of clients. Therefore they offer global access to the storage infrastructure from all HPC system nodes [3]. To utilize the full bandwidth, IOPS and storage space offered by the storage and file servers, parallel filesystems stripe data across storage servers and HDDs within the storage server. Filesystems use metadata to track file and data locations, among other data information. While bandwidth and IOPS are mostly limited by storage infrastructure, metadata operations highly depends on the parallel filesystem used. Metadata operations include opening, closing, creating and deleting of files. Some parallel filesystem such as Lustre use dedicated metadata servers, others such as GPFS distribute the data across the available storage servers [3]. Parallel filesystems have to provide data reliability and cope with storage device failure. To prevent data corruption data access synchronization, such as data locks, have to be implemented.

Due to their complexity parallel filesystems offer a large number of configuration parameters. One such parameter is the filesystem's block size, the size of the smallest addressable unit in the filesystem. Configuring a parallel filesystem for optimal performance is a difficult task due to the large parameter space. The correct configuration is also dependent on the I/O behaviour resulting from the application in combination with high level I/O libraries and the used I/O interface. For example, smaller requests than the block size could result in an overhead [5] and possibly wasting storage space.

I/O interface

Most (if not all) HPC systems still implement and use the POSIX-I/O[2] standard, this can be expected to continue for a long period of time. Since POSIX-I/O was originally introduced to manage local filesystems, other I/O interfaces that offer more parallel functionality were introduced. One such interface is the MPI-IO, an extension to the Message Passing Interface (MPI) standard. Nonetheless, many applications still directly employ POSIX-I/O [14].

High level I/O libraries

Applications can either utilize the available I/O interface directly or use higher level I/O libraries. These offer added functionality such as complex data managing tools. Higher level I/O libraries primarily focus on improved user functionality rather than optimizing performance [3].

1.3.1 Emerging I/O Architectures

The increase in computation and storage size have not been met with a similar increase in I/O performance. While applications get offered more from both, the compute nodes might spend valuable time idling for I/O. As a result, many improvements to the I/O system have been offered. Emerging I/O architectures have also indicated ways to improve the I/O performance. These architectural changes can span over a single or multiple I/O stack layers. To benefit from or implement some of these improvements application changes might be required.

It is possible to group optimizations of I/O into two sets, system and applications optimizations. Application specific I/O optimization builds on changing the I/O behaviour to adapt and better fit to an existing I/O system. For that purpose a deeper understanding of the application's I/O algorithm and available I/O libraries and tools is required. When improving unoptimized application I/O, a significant performance increase can be obtained. As a disadvantage the optimization will only be applied to single application. Both [18] and [19] are examples of improving I/O of an application.

[2]Portable Operating System Interface for Unix

System optimization or improved I/O architectures can benefit a whole set of applications. The improvements build on either hardware or software changes that could benefit I/O performance. Occasionally system optimizations or changes require adapting the applications. Even when application changes are not required the overall benefit could depend on the I/O behaviour exhibited. As a result, not all applications are equally improved when implementing system optimizations.

The following explain a short set of I/O optimizations and emerging I/O architectures. While the focus remains on system optimizations, when suitable application optimizations are mentioned. The main target from this study remains observing the I/O behaviour of scientific applications on HPC systems. However, the explained optimizations are later used to possibly shed light on how the observed I/O behaviour can be utilized for improved performance or reducing overall system cost. As a result, the I/O optimization list is not exhaustive, leaving room for matching observed I/O behaviour with future technologies and architectures.

I/O forwarding

To provide more computing power HPC systems are rapidly scaling into order of tens of thousands of compute nodes with the possibility of running in the order of millions of processes. Although parallel filesystems are designed for multi-client use, these scales could significantly degrade performance.

I/O forwarding decreases the number of process accessing a given I/O system. Dedicated I/O nodes are tasked to access data on behalf of the compute nodes [20]. As a result the I/O system has to service a smaller number of processes. Network and system costs are also significantly reduced. Fig 1.2 shows the change of I/O forwarding on the I/O stack [20].

I/O forwarding is a combination of hardware and software changes. As the compute node and I/O node implementation performs the task of I/O forwarding, there is no need for application changes. However, while reducing the number of nodes accessing the filesystem improves performance, application developer should be aware of I/O parallelism and its benefits. Each I/O node has it's own link to the I/O system. Therefore, to fully utilize the available system bandwidth application developers should use as many

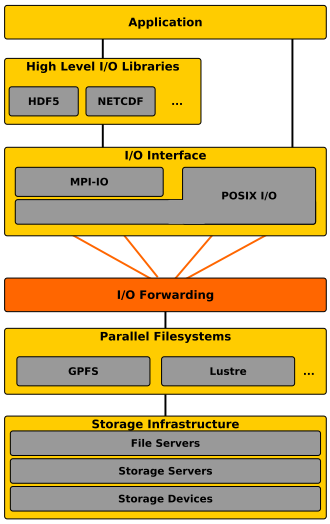


FIGURE 1.2: I/O stack with I/O forwarding

I/O nodes as available in their allocation. I/O forwarding can open the door for more I/O optimizations, such as improved I/O scheduling and asynchronous data staging [21].

Burst buffers

Burst buffers aim at overcoming low bandwidth bottlenecks for high short I/O bursts. Often to implement this the burst buffers exploit high internal network bandwidth. The I/O bursts are then buffered and slowly moved to the external storage. To provide fast I/O, burst buffers use SSDs. The benefit from burst buffers depends on applications and how much they exhibit a bursty behaviour. In [1] the observation of bursty I/O behaviour is considered to model and test the benefits from burst buffers, while in [22] a physical test system is implemented and GPFS is used to manage it. Burst buffers can also be extended to operate as a cache between the computation and I/O system. This could increases the benefit seen on the read path.

Prefetching

A prefetching mechanism reads data from the I/O system ahead of time. The data is therefore preloaded when requested by the application. To achieve a benefit prefetching

can use application I/O idle periods. Two terms can be distinguished in this context. The first is prefetching; the act of system preloading data based on system access pattern recognition during or before application start. Preloading data before application start is called pre-execution prefetching and can be used to hide I/O latency [23]. Prefetching can also be guided by application hints while running. The second term is readahead; which can be considered the act of an application using asynchronous I/O to preload data[3]. Prefetching requires the knowledge or the ability to recognize the exhibited access patterns [24]. The benefits from both implementing prefetching and readahead highly depends on the application I/O behaviour.

Active storage

Most I/O optimizations target improving the I/O access to the external storage system. Active storage is a new concept in which the I/O stack is significantly altered. Storage units are directly embedded into the computation system to be directly accessible [25]. By employing new storage technologies the embedded storage can be used as a burst buffer or as an extension for internal computation memory. The storage can be embedded into all compute nodes, creating a homogeneous system or only placed into a smaller set of nodes such as the I/O nodes creating a heterogeneous system. The embedded storage can use a local filesystem or different data representations such as key/value object store [25]. Benefiting from an active storage configuration requires applications that can utilize the added storage through different interfaces. Determining the extend of that benefit requires understanding the relationship between the applications computation and I/O access patterns. Studying the computational behaviour of application's is beyond the scope of this study. Yet primary knowledge of I/O application behaviour can benefit active storage design and possibly help initiate selection of I/O intensive applications which can benefit from such I/O architectures.

Other optimizations

There are many more I/O optimizations. These include collective I/O and data sieving. Both attempt increasing request sizes by restructuring the access pattern. This builds

[3]The two terms readahead and prefetching are often interchangeably used. The distinction is made in this study to separate system optimizations from application optimizations.

on the assumption that non-contiguous I/O reduces performance [26]. In collective I/O a group of processes can coordinate their I/O access [27]. A non-contiguous group of I/O requests from different processes can therefore be combined. In [28] the benefit of collective I/O is increased by sharing the data layout with the filesystem. Meanwhile, data sieving merges separate I/O requests from a single process by reading or writing data physically allocated between the requests. For writing a read-modify-write operation is performed [26].

The changes for collective I/O and data sieving suggested in [26] and [27] are implemented into the MPI-I/O interface. As a result, all applications that use the MPI-I/O interface can benefit. The extent of performance improvement depends on several factors including system settings and application I/O patterns [29].

Chapter 2

I/O System Architecture

As previously shown, large supercomputers require a complex I/O stack. These support the I/O requirements of the multi-core and multi-thread applications. In Chp. 1, the extend of possible variations on the I/O stack has been introduced. The following explains a practical example on the basis of JUGENE, a BlueGene/P system operated in Juelich supercomputing center between 2009 and 2012. JUGENEs 72 racks offered a peak performance of 1 PetaFlop/s[1] with 288×10^3 compute cores and had 144 TiB of main memory [5].The interest in JUGENE for this study is availability of collected I/O logs over a large period of time. These offer the possibility of further studying the I/O behaviour of scientific applications on a large supercomputing system.

2.1 JUGENE I/O Stack

JUGENE's I/O stack is similar to the one introduced in Chp. 1. The applications can either communicate with the filesystem using POSIX-I/O or employ various higher level I/O libraries, such as HDF5 and NetCDF [5]. These offer an extended API. The main difference to a traditional I/O stack is the use of I/O forwarding. Compute nodes forward their I/O requests to dedicated I/O nodes. These then execute the I/O request on behalf of the compute node. JUGENE had an I/O node to compute node ratio of 1:128, i.e. each rack contains 8 I/O nodes. The last rack is an exception and had a ratio of I/O node to compute nodes of 1:32, i.e. contains 32 I/O nodes. BlueGene/P

[1] Floating point operations per second.

offered two internal networks, a 3D torus and a binary tree for collective communication. Compute nodes employed the binary tree network to forward I/O requests to the I/O nodes.

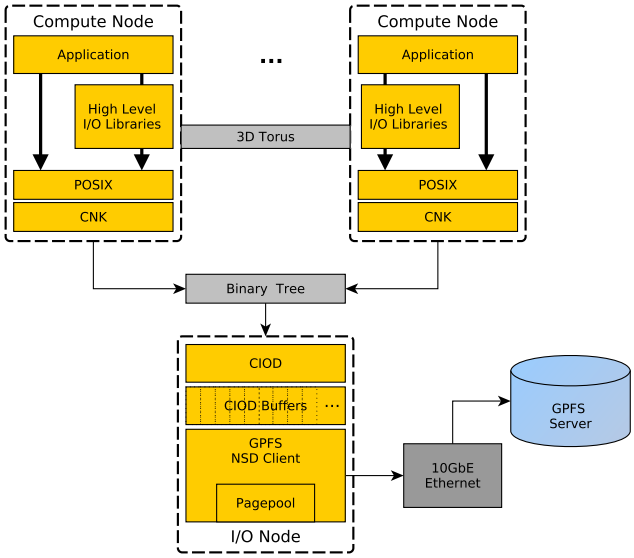


FIGURE 2.1: Blue Gene/P I/O stack, adapted from [5]

Fig. 2.1 shows the I/O forwarding on BlueGene/P. The application initiates the I/O request using either directly POSIX-I/O or higher level libraries. Higher level libraries such as MPI-IO might perform collective operations. These in turn might use the 3D torus to communicate across the compute nodes. The Compute Node Kernel (CNK) then forwards the I/O request via the BlueGene/P binary tree network to the I/O node. The Control and I/O Daemon (CIOD) receives the I/O request for further processing and forwarding to the filesystem. CIOD offers 1 buffer for each process running on the compute node group attached to the I/O node. BlueGene/P has a virtual node (VN) mode that runs 4 processes on each compute node. Therefore the CIOD offers 4 buffers for each compute node [5]. Each compute node process uses its allocated CIOD buffer for forwarding the request to the filesystem. Therefore, if the request is larger than the CIOD buffer allows, the request is split. This takes place on the compute node. The CIOD buffer size has been set on JUGENE to 4MiB [5].

On behalf of the compute node, the I/O node forwards the I/O requests to the filesystem. I/O nodes run a full Linux and employ a 10GbE Ethernet to communicate with the filesystem server. JUGENE uses a storage cluster built on the IBM General Parallel File System (GPFS) [30].

2.2 Storage Infrastructure

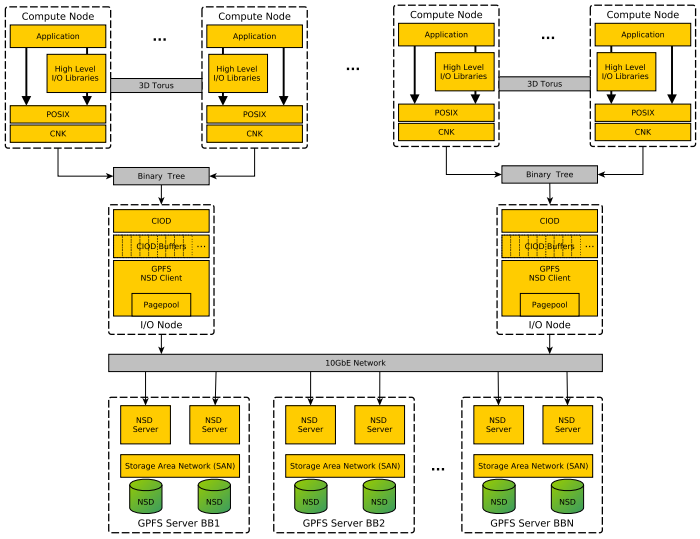


FIGURE 2.2: JUGENE I/O architecture, adapted from [5]

The storage cluster supporting JUGENE has an aggregate performance of 66GiB/s. GPFS is used to handle the disks and servers that are required to achieve this performance. This allows applications to access the filesystem in parallel. GPFS uses Network Shared Disk (NSD) servers. Since compute nodes in a traditional system do not contain any storage, GPFS offers NSD clients. In a traditional system, compute nodes would access the GPFS by running the GPFS NSD clients. As previously explained, BlueGene/P compute nodes forward their I/O requests to dedicated I/O nodes. As shown in Fig. 2.2 the I/O node uses the NSD client to fulfil the I/O requests, that were initiated by the compute node. This construction shows the advantage of I/O forwarding. By using a smaller set of nodes to access the GPFS, only a few NSD clients have to run. This in turn can reduce the computation time spent in I/O organizational operations

and filesystem locks. GPFS client on the I/O nodes contain a pagepool buffer, which is used to cache data and metadata. The pagepool buffer is set to 1024MiB [5]. It is worth noting here that the GPFS file servers are split into several filesystems. These are accessible using shell environment variables. Users are encouraged to use the $\$WORK$ scratch filesystem for I/O during job execution. The other filesystems are used for user home directories or as archives.

2.2.1 GPFS I/O Counters

GPFS provides a method for monitoring I/O performance, which is described in [31, Chapter 8]. The monitoring is based on using the *mmpmon* command (Further details of the command's options can be found in [32, Chapter 7]). The counters registered by the *mmpmon* are shown in table 2.1.

Keyword	Description
_n_	IP address of the node responding. This is the address by which GPFS knows the node.
_nn_	The hostname that corresponds to the IP address (the _n_ value).
_rc_	Indicates the status of the operation.
_t_	Indicates the current time of day in seconds (absolute seconds since Epoch (1970)).
_tu_	Microseconds part of the current time of day.
_cl_	Name of the cluster that owns the file system.
_fs_	The name of the filesystem for which data is being presented.
_d_	The number of disks in the filesystem.
_br_	Total number of bytes read, from both disk and cache.
_bw_	Total number of bytes written, to both disk and cache.
_oc_	Count of open() call requests serviced by GPFS. This also includes creat() call counts.
_cc_	Number of close() call requests serviced by GPFS.
rdc	Number of application read requests serviced by GPFS.
_wc_	Number of application write requests serviced by GPFS.
dir	Number of readdir() call requests serviced by GPFS.
_iu_	Number of inode updates to disk.

Table 2.1: GPFS I/O (*mmpmon*) counters [31].

Many I/O profiling tools exist and offer the ability to monitor the I/O of an application or a system. These tools can be compared on the basis of two properties. The first is the form and quantity of I/O profiling offered. While the second is related to the position of the I/O counters within the I/O stack. The *mmpmon* or GPFS I/O counters are present in the GPFS NSD clients on the I/O nodes. This is shown in Fig. 2.3. Deciding

which profiling tool and which layer in the I/O stack to monitor has an impact on the analysis process. The limitations and benefits from having the I/O counters present in the filesystem on the I/O nodes are discussed in Chp. 4.

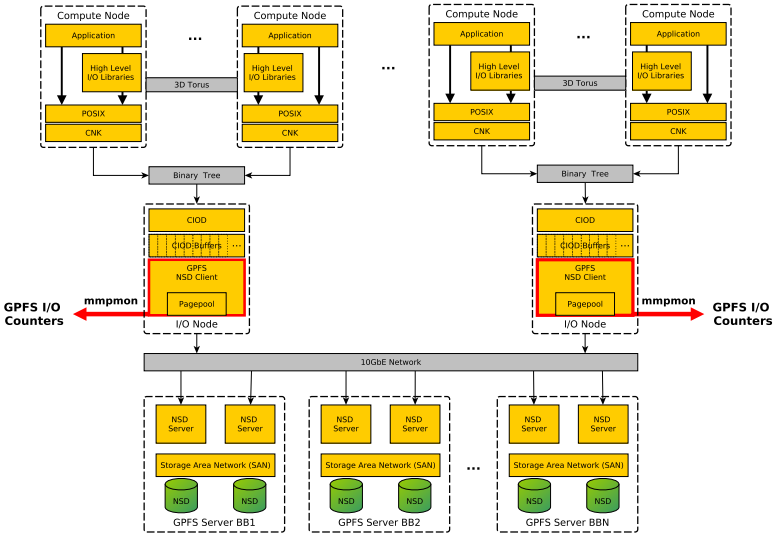


FIGURE 2.3: Location of GPFS I/O counters in JUGENE I/O stack, adapted from [5]

Chapter 3

Methodology: I/O Criteria

Analysing the I/O behaviour of applications has become a crucial part in locating performance bottlenecks. Many tools provide tracing and profiling of application I/O. However, due to the complex underlying systems and libraries that perform the I/O requests, the many dimensions of I/O behaviour are seldom explained by the measured quantities. The complexity of I/O architectures results in a large range of quantities to measure. Therefore a clear characterization of I/O criteria is necessary to analyse the I/O behaviour of an application. These I/O criteria have to be chosen with a careful consideration of both application and I/O system architecture. Understanding the I/O system of HPC brings forward the known I/O metrics, such as bandwidth and I/O Operations Per Second (IOPS). On the other hand, understanding how scientific applications perform I/O brings forward metrics that define the I/O behaviour. The I/O criteria should allow for a wide analysis and comparison of a large set of applications' I/O information. In turn this helps defining the I/O load that is exerted on a modern HPC I/O system.

The complexity of modern HPC systems could lead to an overwhelming number of measurable quantities. As a result, it is helpful to limit the I/O criteria by predefining some I/O architectural bounds. The here defined I/O criteria assumes the existence of a filesystem that manages the available storage through files, thereby ignoring the underlying block layers and the filesystem inner workings. The filesystem allows to perform read, write and seek operations. Furthermore a set of operations are considered for manipulating the filesystem's metadata. These operations are restricted to those

considered to dominate in a typical HPC workload and therefore include file create, open, close and delete.

The selection of I/O criteria requires consideration of three fundamental aspects, which are relevance, ease of measurement and I/O architecture dependency. Relevance generally questions the information attained from measuring and analysing the I/O criteria. However, judging the relevance of I/O criteria to an analysis highly depends on the analysis goal. The I/O criteria here aim for a general survey of I/O behaviour, therefore these consider various criteria that can be ignored for other analysis targets. The second aspect to consider is the ease of measurement. Effectively I/O criteria should only include measurable or quantifiable values. However, judging the ability to measure an I/O criteria highly depends on the I/O architecture and the used method of measuring. As a result, the I/O criteria considered here is limited to those reasonably easy to measure using various techniques that are listed when required. The third and final aspect to consider when selecting I/O criteria is I/O architecture dependency. The values measured to analyse the I/O criteria should attain an independence from the used I/O architecture. In case a full independence is not possible, the I/O criteria should be applicable to a broad range of architectures. For example, the maximum filling of buffers in an I/O system could be considered an interesting I/O criteria to measure. However, as some I/O architecture do not employ large buffers, such I/O criteria would not be applicable to a broad range of I/O systems. A counter example is I/O parallelism, where it is safe to consider that most I/O subsystems employed by modern HPC systems would involve a parallel access to the I/O storage.

For practical use, an analysis using the I/O criteria has to observe two aspects, application dependent changes and I/O measuring conditions. Application changes include problem size, implementation or code revision and run or partition size. These have to be defined in relation to any I/O profiling of an application. On the other hand, I/O measuring conditions define the state of the machine and the layer of the I/O stack at which I/O is measured.

The state of the machine has to be considered for the definition of I/O criteria, as it is not possible or not desirable to perform a machine independent I/O profile. This is due to machine I/O system and compiler software changing the I/O behaviour of the application being profiled. Furthermore, the I/O behaviour measured for an application

might also be changed depending on the total I/O load produced by all applications running on the machine at the time of measurement. The second component of I/O measuring conditions is the layer or level of I/O stack at which I/O is measured. Buffering and collective I/O for example, change the measured number and timing of I/O requests. Some machine architectures, such as BlueGene provide I/O forwarding nodes, thus changing the number of I/O accessing processes on various I/O stack layers. Meanwhile, I/O libraries change requests sizes, timings and even number of files accessed. It is therefore crucial to define these unknowns or observe the limitations they set on the I/O profiling. Once these conditions are met, the defined I/O criteria can be used to profile the I/O behaviour of an application.

The description provided here for the selected I/O criteria occasionally define some practical issues with evaluating and measuring individual I/O criteria. To investigate the full potential of the I/O criteria, Chp. 4 provides a practical example for evaluating the I/O criteria using the GPFS I/O counters introduced in Sec. 2.2.1.

3.1 Related Work

Most found related work does not provide an all around summary of I/O metrics and criteria that can be used for a full evaluation of application I/O behaviour. This is often due to the existence of a specific target for the I/O analysis, thereby reducing the considered I/O criteria to those relevant to the pre-set goal. For example, [7], [24] and [33] introduce some I/O criteria, focusing the scope on only those needed for applying some special optimizations. However, the intention here is analysing the general I/O behaviour of applications, from which the design of new I/O architecture can benefit. Thus the I/O criteria introduced requires consideration of many more available I/O metrics, including those found in various sources.

In [7] a clear focus on storage infrastructure evaluation is given. Here the common I/O metrics such as bandwidth and IOPS in the form of throughput is introduced. Additionally, capacity, reliability and cost are considered. These traditional metrics are useful when comparing different storage systems. The approach taken here differs by considering that I/O storage infrastructure performance depends on application I/O behaviour. To fulfil this analysis the given I/O metrics are redefined as seen by the application.

The then defined application I/O requirements can be used to judge the total benefit of a superior system in terms of these metrics. The assumptions is that understanding maximums and totals of bandwidth, IOPS and needed capacity of application's in relation to available system resources, could allow for achieving a higher I/O performance and/or reducing cost.

In [24] I/O access patterns are defined, to implement and optimize prefetching. Meanwhile, in [33] the ability of applications to decouple I/O from processing is introduced to exploit asynchronous I/O. By aggregating and regrouping these introduced I/O criteria it is possible to formulate a wide view for a better understanding of overall application I/O behaviour. Thereby allowing for weighing the benefits from introducing new I/O architectures to current application I/O behaviour.

The approach of evaluation of a large set of applications' I/O is also taken in some studies. For example, [15] tackles I/O system optimization with an attempt to perform a closer study of application I/O behaviour. Although [15] contains many useful I/O criteria, the analysis presented here complements these by using many more. Additionally, [15] focuses the analysis on a limited set of applications. In comparison the I/O criteria selected here should allow for mass analysis and comparison of applications' I/O behaviour.

The best attempt found for introducing I/O criteria and metrics that can be used for a wide application I/O analysis is the Charisma project [11], [34] and [35]. Although the intention of a wide I/O application study is given, the focus remains on file access patterns to improve filesystem's performance. To achieve this goal the project limited itself to analysing production I/O without relating the I/O behaviour to specific applications. In contrast, the I/O criteria selection introduced here is application based, thereby requiring connecting the I/O behaviour and the application.

Other sources extrapolate on the subject of I/O interfaces. Some criteria can be inferred from these sources. As an example [36] deals with designing a portable I/O interface. It therefore explains the difficulties that applications face in adapting and optimizing I/O. As result, further application details are relevant to the I/O criteria and the analysis of I/O behaviour. These might include I/O interface and libraries used, as well as implemented I/O optimizations. Some I/O criteria related to application details can be found in App. A.

3.2 Basic Quantities

To clearly the formulation of the I/O criteria, it is necessary to define basic quantities in relation to I/O. These quantities are organized into three groups defining the application, the I/O requests and the file or filesystem operations.

3.2.1 Application Quantities

Three quantities are considered to define an application for I/O analysis:

t_{start} ***Time when execution of application starts.***

t_{end} ***Time when execution of application ends.***

P ***Number of processes.*** Where each process is identified by a number p $(0 \leq p < P)$.

3.2.2 I/O Request Quantities

Data or I/O requests on the other hand can be defined by time t, size s and type as read r or write w. Since the I/O criteria intends the evaluation of overall application I/O behaviour, it is easier to consider the aggregate number of operations a basic quantity. This leads to the definition of the following primary quantities:

$D_{\text{r}}(s,t)$ ***Number of read operations with request size*** s ***Bytes*** $(s \geq 0)$ ***executed during time interval from*** t_{start} ***until time*** t $(t_{\text{start}} \leq t \leq t_{\text{end}})$. By definition $D_{\text{r}}(s, t_{\text{start}}) = 0$.

$D_{\text{w}}(s,t)$ ***Number of write operations with request size*** s ***Bytes*** $(s \geq 0)$ ***executed during time interval from*** t_{start} ***until time*** t $(t_{\text{start}} \leq t \leq t_{\text{end}})$. By definition $D_{\text{w}}(s, t_{\text{start}}) = 0$.

Using the basic quantities some derived quantities can be defined:

$D_{\mathrm{r}}(t)$ ***Number of read operations with any request size executed during the time interval from*** t_{start} ***till time*** t***.***

$$D_{\mathrm{r}}(t) = \sum_{s} D_{\mathrm{r}}(s,t) \tag{3.1}$$

$D_{\mathrm{w}}(t)$ ***Number of write operations with any request size executed during the time interval from*** t_{start} ***till time*** t***.***

$$D_{\mathrm{w}}(t) = \sum_{s} D_{\mathrm{w}}(s,t) \tag{3.2}$$

$\widetilde{D}_{\mathrm{r},\Delta\mathrm{t}}(s,t)$ ***Number of read operations with request size*** s ***Bytes (*** $s \geq 0$***) executed during the time interval*** $[t-\Delta t, t]$***.***

$$\widetilde{D}_{\mathrm{r},\Delta\mathrm{t}}(s,t) = \Delta t \frac{\partial D_{\mathrm{r}}(s,\tau)}{\partial \tau}|_{\tau=t} \tag{3.3}$$

In case of sampling the number of read operations in discrete time intervals $t_k = t_0 + k\Delta t$ this reduces to:

$$\widetilde{D}_{\mathrm{r},\Delta\mathrm{t}}(s,t) = D_{\mathrm{r}}(s,t_k) - D_{\mathrm{r}}(s,t_{k-1}) \tag{3.4}$$

$\widetilde{D}_{\mathrm{w},\Delta\mathrm{t}}(s,t)$ ***Number of write operations with request size*** s ***Bytes (*** $s \geq 0$***) executed during the time interval*** $[t-\Delta t, t]$***.***

$$\widetilde{D}_{\mathrm{w},\Delta\mathrm{t}}(s,t) = \Delta t \frac{\partial D_{\mathrm{w}}(s,\tau)}{\partial \tau}|_{\tau=t} \tag{3.5}$$

In case of sampling the number of write operations in discrete time intervals $t_k = t_0 + k\Delta t$ this reduces to:

$$\widetilde{D}_{\mathrm{w},\Delta\mathrm{t}}(s,t) = D_{\mathrm{w}}(s,t_k) - D_{\mathrm{w}}(s,t_{k-1}) \tag{3.6}$$

$\widetilde{D}_{\mathrm{r},\Delta\mathrm{t}}(t)$ ***Number of read operations with any request size executed during the time interval*** $[t-\Delta t, t]$**.**

$$\widetilde{D}_{\mathrm{r},\Delta\mathrm{t}}(t) = \sum_{s} \widetilde{D}_{\mathrm{r},\Delta\mathrm{t}}(s,t) \tag{3.7}$$

$\widetilde{D}_{\mathrm{w},\Delta\mathrm{t}}(t)$ ***Number of write operations with any request size executed during the time interval*** $[t-\Delta t, t]$**.**

$$\widetilde{D}_{\mathrm{w},\Delta\mathrm{t}}(t) = \sum_{s} \widetilde{D}_{\mathrm{w},\Delta\mathrm{t}}(s,t) \tag{3.8}$$

3.2.3 Filesystem Metadata Operation Quantities

Filesystem metadata operations can be described using time t, size s of file and type of file operation, which can be either create, open or close. Using this the following quantities can be defined:

$F_{\mathrm{create}}(s,t)$ ***Number of files with size*** s ***(*** $s \geq 0$***) created during the time interval from*** t_{start} ***till*** t ***(*** $t_{\mathrm{start}} \leq t \leq t_{\mathrm{end}}$***).*** For any s, $F_{\mathrm{create}}(s, t_{\mathrm{start}}) = 0$.

$F_{\mathrm{open}}(t)$ ***Number of file open operations executed during the time interval from*** t_{start} ***till*** t ***(*** $t_{\mathrm{start}} \leq t \leq t_{\mathrm{end}}$***).*** By definition $F_{\mathrm{open}}(t_{\mathrm{start}}) = 0$.

$F_{\mathrm{close}}(t)$ ***Number of file close operations executed during the time interval from*** t_{start} ***till*** t ***(*** $t_{\mathrm{start}} \leq t \leq t_{\mathrm{end}}$***).*** By definition $F_{\mathrm{close}}(t_{\mathrm{start}}) = 0$.

For some criteria it is necessary to link the I/O request with the process that initiated the request and the file on which the request is carried out. For these the following quantities are defined:

$R(p,f,i)$ ***i-th I/O request from process p for a given file handle f.*** i is in the range $[0, n_{\mathrm{p,f}}]$, where $n_{\mathrm{p,f}}$ is the total number of I/O requests from process p to file f.

$R_{\mathrm{off}}(p,f,i)$ ***Offset of the i-th I/O request from process p for a given file handle f with respect to the file start.***

Additional filesystem operations can be defined using time t, file handle f and type being either directory operation or inode update. The following quantities are then defined:

$F_{\mathrm{dir}}(t)$ ***Number of directory operations during the time interval from*** t_{start} ***till*** t ***(***$t_{\mathrm{start}} \leq t \leq t_{\mathrm{end}}$***).*** By definition $F_{\mathrm{dir}}(t_{\mathrm{start}}) = 0$.

$F_{\mathrm{inode}}(t)$ ***Number of inode updates executed during the time interval from*** t_{start} ***till*** t ***(***$t_{\mathrm{start}} \leq t \leq t_{\mathrm{end}}$***).*** By definition $F_{\mathrm{inode}}(t_{\mathrm{start}}) = 0$.

3.3 Category 1: Aggregate Performance Numbers

The aggregate performance numbers represent a quantitative evaluation of application I/O and therefore assess the magnitude of I/O the system has to withstand. Most common benchmarks and I/O analysis use these metrics to measure I/O system bottlenecks. As a consequence many assume that by achieving the highest requirements of application I/O in terms of these metrics the I/O has been thoroughly optimized. However, these metrics only offer a single dimension of optimization and might result into over-assessing application I/O requirements and missing the opportunity of creating a more cost effective I/O system. One reason for this misconception is the ease by which the performance metrics can be measured. A wide range of benchmarks are available to measure the system I/O, even using the same access patterns that applications produce. Additionally, optimization based on these metrics allow system administrators to independently reconfigure the I/O system with the hope for better performance without the need for application changes. Therefore, these metrics are still valuable to know when analysing I/O, but should be critically observed in light of more detailed qualitative I/O analysis metrics.

Classification 1.1 Total amount of data read/written

The total amount of data read or written by the execution of an application can be defined by:

$$S_{\mathrm{r}} = \sum_{s} sD_{\mathrm{r}}(s, t_{\mathrm{end}}) \tag{3.9}$$

$$S_{\mathrm{w}} = \sum_{s} sD_{\mathrm{w}}(s, t_{\mathrm{end}}) \tag{3.10}$$

S_r and S_w are important to understand the capacity of data that the I/O storage system will deal with. It is also beneficial to know the data quantities an application works on when using various buffering techniques. It should be noted that total amount of data read or written mostly does not depend on the I/O stack layer where it is measured. One exception would be the use of compression on one or more of the I/O layers. Another exception is the use of data sieving [26], where data that lies between non-contiguous I/O requests is also read.

Classification 1.2 Total number of IOPs

Total number of read or write I/O requests initiated by the application defined as:

$$N_\mathrm{r} = \sum_s D_\mathrm{r}(s, t_\mathrm{end}) \tag{3.11}$$

$$N_\mathrm{w} = \sum_s D_\mathrm{w}(s, t_\mathrm{end}) \tag{3.12}$$

N_r and N_w do not include filesystem operations such as file creation and file opening. The IOP definition here only relates to application initiated I/O requests. The total number of IOPs heavily depends on the I/O stack layer it is measured on. This is due to various I/O optimizations such as data sieving and collective I/O [26]. Reducing the number of IOPs on various layers has been mostly driven by the use of HDDs, these offer better performance for reduced contiguous IOPs. It is worth noting that some emerging storage technologies benefit less from this optimization [37, Chapter 5].

Classification 1.3 Read/Write bandwidth

The read and write bandwidth at time t can be defined as:

$$B_\mathrm{r}(t) = \frac{d}{d\tau} \sum_s s D_\mathrm{r}(s, \tau)|_{\tau=t} \tag{3.13}$$

$$B_\mathrm{w}(t) = \frac{d}{d\tau} \sum_s s D_\mathrm{w}(s, \tau)|_{\tau=t} \tag{3.14}$$

As D_r and D_w might not be continuous but rather measured at discrete points of time, the bandwidth measured becomes a local average over Δt, where Δt is the measuring

interval. Using the discrete time interval definition of $\widetilde{D}_{\mathrm{r},\Delta\mathrm{t}}(s,t)$ and $\widetilde{D}_{\mathrm{w},\Delta\mathrm{t}}(s,t)$ given in Eq. 3.4 and 3.6 respectively, the bandwidth can be redefined as:

$$B_{\mathrm{r}}(t) = \frac{\sum_s s\widetilde{D}_{\mathrm{r},\Delta\mathrm{t}}(s,t)}{\Delta t} \tag{3.15}$$

$$B_{\mathrm{w}}(t) = \frac{\sum_s s\widetilde{D}_{\mathrm{w},\Delta\mathrm{t}}(s,t)}{\Delta t} \tag{3.16}$$

The maximum read and write bandwidth for an application can be defined as:

$$B_{\mathrm{max,r}} = \max_{t_{\mathrm{start}} \le t \le t_{\mathrm{end}}} [B_{\mathrm{r}}(t)] \tag{3.17}$$

$$B_{\mathrm{max,w}} = \max_{t_{\mathrm{start}} \le t \le t_{\mathrm{end}}} [B_{\mathrm{w}}(t)] \tag{3.18}$$

Using total amount of data read and written defined in Eq. 3.9 and Eq. 3.10 respectively, the average read and write bandwidth over the application's runtime can be defined as:

$$\overline{B}_{\mathrm{r}} = \frac{S_{\mathrm{r}}}{t_{\mathrm{end}} - t_{\mathrm{start}}} \tag{3.19}$$

$$\overline{B}_{\mathrm{w}} = \frac{S_{\mathrm{w}}}{t_{\mathrm{end}} - t_{\mathrm{start}}} \tag{3.20}$$

Comparing the maximum bandwidth with the available network and I/O system bandwidth helps indicate how often the I/O subsystem is really challenged by the application. As a result, an application that operates at the maximum available bandwidth might benefit from an I/O system bandwidth increase. However, without further analysing the applications I/O behaviour a simple increase in system bandwidth might be a missed opportunity for more cost effective solutions. For example, an application that is only bottlenecked over a few bursts by the bandwidth limitation could be optimized using burst buffers [1], which are explained in Sec. 1.3.1. Such possible use of emerging I/O architectures will be further introduced when evaluating the I/O criteria using the GPFS I/O counters in Chp. 4. On the other hand, the average bandwidth in relation to the total amount of data read or written (Classification 1.1) would allow for comparing I/O magnitude of different applications. This could be of interest for HPC system administrators that are looking for applications that require further attention on the I/O system.

Classification 1.4 Read/Write IOPS

Read or write IOPS (Γ) at time t can be defined as:

$$\Gamma_{\mathrm{r}}(t) = \frac{dD_{\mathrm{r}}(\tau)}{d\tau}|_{\tau=t} \tag{3.21}$$

$$\Gamma_{\mathrm{w}}(t) = \frac{dD_{\mathrm{w}}(\tau)}{d\tau}|_{\tau=t} \tag{3.22}$$

Similar to the definition of bandwidth, D_{r} and D_{w} might not be continuous but rather measured at discrete points of time. Thus the measured IOPS becomes a local average over Δt, where Δt is the measuring interval. Read and write IOPS can therefore be redefined as:

$$\Gamma_{\mathrm{r}}(t) = \frac{\widetilde{D}_{\mathrm{r},\Delta\mathrm{t}}(t)}{\Delta t} \tag{3.23}$$

$$\Gamma_{\mathrm{w}}(t) = \frac{\widetilde{D}_{\mathrm{w},\Delta\mathrm{t}}(t)}{\Delta t} \tag{3.24}$$

Where $\widetilde{D}_{\mathrm{r},\Delta\mathrm{t}}(s,t)$ and $\widetilde{D}_{\mathrm{w},\Delta\mathrm{t}}(s,t)$ in Eq. 3.7 and Eq. 3.8 are evaluated using the discrete time interval definition given in Eq. 3.4 and Eq. 3.6 respectively.

The maximum read and write IOPS achieved during application run can be defined as:

$$\Gamma_{\mathrm{max,r}} = \max_{t_{\mathrm{start}} \le t \le t_{\mathrm{end}}} [\Gamma_{\mathrm{r}}(t)] \tag{3.25}$$

$$\Gamma_{\mathrm{max,w}} = \max_{t_{\mathrm{start}} \le t \le t_{\mathrm{end}}} [\Gamma_{\mathrm{w}}(t)] \tag{3.26}$$

Using the total number of read and write I/O requests defined in Eq. 3.11 and Eq. 3.12 respectively, the average read and write IOPS of an application run can be defined as:

$$\overline{\Gamma}_{\mathrm{r}} = \frac{N_{\mathrm{r}}}{t_{\mathrm{end}} - t_{\mathrm{start}}} \tag{3.27}$$

$$\overline{\Gamma}_{\mathrm{w}} = \frac{N_{\mathrm{w}}}{t_{\mathrm{end}} - t_{\mathrm{start}}} \tag{3.28}$$

Similar to maximum bandwidth, maximum IOPS of an application can help indicate how often the I/O subsystem is really challenged by the application's I/O. It is important to note that bandwidth and IOPS are linked through the request size. In other words, an application limited by IOPS but not by bandwidth, could benefit from increasing the

request sizes. Indeed many libraries and I/O stack layers attempt increasing the request size by using data sieving and collective I/O [26]. Using buffers write can be delayed to combine spatially adjacent requests. Meanwhile, using readahead and prefetching larger chunks of data can be read, thereby increasing the request sizes. As a result, the number of I/O requests executed changes on different I/O stack layers, directly leading to a change in the IOPS value measured. On the other hand, average IOPS in relation to total number of I/O requests (Classification 1.2) allows for comparing I/O request magnitudes of different applications.

Classification 1.5 Total number of files created

Total number of file creations can be defined as:

$$F_{\text{create}} = \sum_{s} F_{\text{create}}(s, t_{\text{end}}) \tag{3.29}$$

The purpose is to have initial information on the metadata operations required by the application. Many applications are designed using task-local files, that is each process creates and uses its own file. As the number of cores and processes on supercomputers increases, filesystems have to endure an ever growing number of metadata operations. To prove this limitation, a test shown in [38] resulted in JUGENE's attached filesystem taking $33min$ to create a total of $256K$ files. [38] continues to describe SIONlib a library that reduces the number of physical open files. Therefore when using SIONlib the number of open or created files could change between I/O stack layers, which should be acknowledged when measuring the number of created files. As additional information each file size might be analysed. Knowing the maximum $F_{\text{max,s}}$ and minimum $F_{\text{min,s}}$ file sizes might be useful for backup and filesystem design.

Classification 1.6 I/O intensity

The I/O intensity can be defined as the number of time intervals where read, write or a mix of read/write operations dominate, divided by the total execution time ($t_{\text{end}} - t_{\text{start}}$) of the application. I/O in the form of read or write is considered to dominate if it surpasses a certain threshold c. The threshold can be tuned so as to avoid small I/O, such as logging, to influence the I/O intensity. Other I/O intensity definitions

exist, such as the execution time without I/O divided by the execution time with I/O. However, compared to the definition used here some of these lack ease and accessibility of measuring I/O intensity for a large set of applications.

An issue with defining I/O intensity is that it considers the time taken by an I/O event. So far the duration of an I/O request was not considered, as only the number of I/O events is counted in a given time interval. While an I/O event can be counted at the moment of initiation or conclusion, the I/O request data stream would last for a given period of time. This is relevant when considering the property of I/O used to define dominating I/O intervals. The property can either be I/O requests or I/O quantity and would be compared accordingly to a reasonably selected threshold c.

Assuming that modern HPC systems employ multiprocessing, to formulate the I/O intensity the following definition is needed for each process p:

$$h_{\Delta \mathrm{t,p}}(t) \quad = \quad \begin{cases} 1 & \text{if} \quad \phi_{\mathrm{r}} > c \quad \text{or} \quad \phi_{\mathrm{w}} > c, \\ 0 & \text{else}, \end{cases} \tag{3.30}$$

Where ϕ depends on the I/O property used to evaluate the dominating I/O intervals. When using I/O requests, $\phi_{\mathrm{r}} = \sum_s \widetilde{D}_{\mathrm{r},\Delta \mathrm{t}}(s,t)$ and $\phi_{\mathrm{w}} = \sum_s \widetilde{D}_{\mathrm{w},\Delta \mathrm{t}}(s,t)$. Meanwhile, when using I/O quantity $\phi_{\mathrm{r}} = \sum_s s\widetilde{D}_{\mathrm{r},\Delta \mathrm{t}}(s,t)$ and $\phi_{\mathrm{w}} = \sum_s s\widetilde{D}_{\mathrm{w},\Delta \mathrm{t}}(s,t)$. It should be noted that despite defining the I/O quantity in terms of I/O requests, the measured I/O quantity should be viewed as a flux of data over time disregarding the related I/O requests.

Based on $h_{\Delta \mathrm{t,p}}$ the following can be defined:

$$H_{\Delta \mathrm{t}}(t) \quad = \quad \begin{cases} 1 & \text{if for at least one } p, \quad h_{\Delta \mathrm{t,p}}(t) > 0, \\ 0 & \text{else}, \end{cases} \tag{3.31}$$

Finally the I/O intensity can then be defined as:

$$\begin{aligned} I \quad &= \quad \frac{\sum_t H_{\Delta \mathrm{t}}(t)}{(t_{\mathrm{end}} - t_{\mathrm{start}})/\Delta t} \\ &= \quad \Delta t \frac{\sum_t H_{\Delta \mathrm{t}}(t)}{t_{\mathrm{end}} - t_{\mathrm{start}}} \end{aligned} \tag{3.32}$$

In essence the I/O intensity computes the fraction of time during which I/O is performed. Therefore the I/O intensity is both machine and application dependent. A computationally powerful machine would make applications seem more I/O intense, while a machine with a larger I/O subsystem would make the same applications seem less I/O intense. Therefore the measured I/O intensity cannot be compared across different systems. Additionally modern I/O subsystems tend to serve a larger set of separate systems running separate applications. Therefore the real time I/O intensity would not only depend on the I/O operations of the application but on other applications that ran at the same time.

Although the principle might seem simple, measuring I/O intensity requires further issues to be covered. As I/O might be done asynchronous or buffered, the I/O stack layer on which the fraction of time spent in I/O is measured, effects the I/O intensity. For example, using asynchronous I/O measuring on the application level would yield a different I/O intensity than measuring in the filesystem. Furthermore, in case of the presence of buffering it should be made clear when an I/O operation has been concluded. For example, a write request can be considered done either when it has reached the final storage or once the data has been buffered.

A closer study of I/O intensity can help begin to evaluate the usefulness applications would have from introducing asynchronous I/O, when measured on a synchronous I/O system. This can be done in relation to the busy work parameter defined in [9] as the computational complexity needed by an application to hide the I/O operations on a given system. The practicality of such a study however depends on the I/O stack layer where the I/O intensity is measured.

3.4 Category 2: I/O Pattern Analysis

For the purpose of improving prefetching, Byna et al. [24] suggested to classify I/O patterns using five dimensions:

1. Request size
2. Type of I/O operation
3. Spatiality

4. Temporal intervals

5. Repetitive behaviour

Others reiterate these access pattern dimensions adding various details or focusing on a subset [39], [15], [11] and [40]. To allow for analysing access patterns, the information from these sources are combined and symbolically defined.

For parallel I/O in multiprocessing systems, the I/O can be viewed in terms of local or global I/O. Local I/O refers to I/O as performed by individual processes, while global I/O refers to the overall application I/O behaviour. Shown in [40] the local I/O access patterns can be combined to form a global I/O access pattern view of an application. For some HPC systems employing different I/O architectures, analysing the I/O per process might not be relevant or feasible. For these systems the local I/O can be redefined as needed. For example, local I/O can be defined as either the I/O as performed by individual nodes or in case of I/O forwarding as the I/O performed by the dedicated I/O nodes. However, the number of nodes or I/O nodes in a system is usually smaller than the number of processes available to an application. Therefore, using node or I/O nodes instead of processes as a local view would change the I/O patterns analysed. In all cases, since most modern HPC I/O subsystems support large scale parallel I/O, the I/O access pattern dimensions should allow for analysing both the local and global I/O access patterns. The request size is excluded from this rule, as it has the same local and global view. An example for the need of having a local and global view of I/O access patterns is a local I/O showing repetitive behaviour that is not visible on the global view.

Both the time and the I/O stack layer of analysis are critical points in evaluating application I/O patterns. While most aggregate performance numbers can only be obtained from measuring a run of the application, I/O pattern behaviour could also be inferred from the I/O access style encoded into the application's algorithm. This is due to application I/O pattern analysis being a qualitative rather than a quantitative evaluation. However, caution has to be taken when comparing real time I/O access pattern with expected or coded I/O pattern. For example, I/O requests built into individual processes might not occur at the same time if interprocess synchronization on I/O does not take place. Additionally over the I/O layers request sizes and timing of requests might change, which also should be accounted for on both the local and global view.

3.4.1 Request Size

The request size can be defined as the I/O quantity transferred as either read or write when executing the I/O request and is a crucial factor that links IOPS to bandwidth. Up to a limit and depending on the I/O subsystem and storage technology, when increasing the request size, bandwidth increases and IOPS decrease. Many including [24] consider small request sizes as a factor that degrades I/O performance. A good request size could be considered as one that does not limit performance by reaching the limit of possible IOPS.

In general each I/O request induces an overhead given by the network and the filesystem. Therefore, the assumption is that a group of small I/O requests should result in a larger latency than a single I/O request with the same total request size. However depending on the I/O subsystem and the used storage technology this assumption might be mistaken. For example, an HDD might quickly reach its peak IOPS when presented with a large set of scattered small I/O requests. Meanwhile, an SSD could handle these scattered small I/O requests faster if it can employ more on board storage chips [37, Chapter 5]. Hence, analysing the request sizes of applications should observe the I/O subsystem's performance using various request sizes.

Classification 2.1 Distribution of request sizes

The distribution of the probabilities $p_{\mathrm{r}}(s)$ and $p_{\mathrm{w}}(s)$ of read and write operations with requests size s can be defined as:

$$p_{\mathrm{r}}(s) = \frac{D_{\mathrm{r}}(s, t_{\mathrm{end}})}{\sum_s D_{\mathrm{r}}(s, t_{\mathrm{end}})} \tag{3.33}$$

$$p_{\mathrm{w}}(s) = \frac{D_{\mathrm{w}}(s, t_{\mathrm{end}})}{\sum_s D_{\mathrm{w}}(s, t_{\mathrm{end}})} \tag{3.34}$$

An equally useful tool for investigating request sizes is the cumulative distribution function (CDF). In [15] the CDF of the percentage of read and write operations versus the request size is investigated for different scientific applications.

Many I/O stack layers employ I/O techniques that lead to changing the number and subsequently the size of I/O requests. As a result, the I/O stack layer on which the I/O

request sizes are measured highly affects the distribution of request sizes. This should be accounted for when analysing the evaluation of the distribution.

Filesystems designate a block size (s_{block}) as the smallest I/O operation size. An I/O request that is not a multiple of the filesystem block size might increase the transferred I/O. Generally, I/O requests larger than or equal to the filesystem block size ($s \leq s_{\text{block}}$) should be acceptable, as these already start to reduce partial accesses to filesystem blocks. Consequently, a request size distribution with higher probability of multiple or larger than filesystem block size (s_{block}) requests could have a better overall performance.

Classification 2.2 Percentage of small I/O requests

A request size can be considered small if it is smaller than or equal to a predefined value s_{small}. Thus the fraction of small read and write requests can be defined as:

$$f_{\text{r}}(s_{\text{small}}) = \frac{\sum_{s \leq s_{\text{small}}} D_{\text{r}}(s, t_{\text{end}})}{\sum_{s} D_{\text{r}}(s, t_{\text{end}})} \tag{3.35}$$

$$f_{\text{w}}(s_{\text{small}}) = \frac{\sum_{s \leq s_{\text{small}}} D_{\text{w}}(s, t_{\text{end}})}{\sum_{s} D_{\text{w}}(s, t_{\text{end}})} \tag{3.36}$$

The most reasonable value to identify a small request size is to use $s_{\text{small}} = s_{\text{block}}$, where s_{block} is the filesystem block size. Since filesystems conduct every I/O operation in multiples of s_{block}, smaller than s_{block} requests could result in an overhead. Other values for s_{small} might also be used. As discussed when introducing request sizes, small request sizes could result in higher latency as the application becomes IOPS limited. Therefore s_{small} could be identified as the request size that would not allow for the IOPS limit to be reached.

For practical evaluation, the percentage of small I/O requests measured depends on the I/O stack layer at which the request sizes are measured. This is due to same reasons as discussed for the distribution of request sizes (Classification 2.1).

Classification 2.3 Request size: Variable vs. fixed

The request size of an I/O operation is considered fixed, if for a given problem size the i-th I/O request size is fixed. That is the i-th I/O operation will always have the same size (s_i = constant), irrelevant of the problem state or other factors. The number of variable size I/O requests can be defined as:

$$\sum_i \sigma(i) \qquad \text{where:} \qquad \sigma(i) \;=\; \begin{cases} 0 & \text{if} \quad s_i = \text{constant}, \\ 1 & \text{else}, \end{cases} \tag{3.37}$$

Meanwhile a request size is considered variable if it varies, e.g. depending on the problem state. A variable request size s_i of the i-th operation can be the result of the input data, even though the problem size has not changed. Applications with an increase in fixed request sizes are more likely to exhibit repetitive behaviour either in the temporal or spatial dimensions. Meanwhile, an increase in variable request sizes could complicate understanding the I/O behaviour of the application as this could in turn depend on the problem state or other factors. Furthermore, adapting underlying I/O architectures for improved performance could be less complicated for fixed I/O requests than for variable ones.

Variability of request sizes can be investigated using several runs of the same application, while comparing s_i of each run with its previous value. Another more suitable option is to investigate the I/O algorithm built into the application. An application that by design uses uniform or constant I/O request sizes will most likely have fixed request sizes. Another application designed using work queues or work stealing might allow different processes to take different work loads, thus changing s_i. It is important to note that in the context of fixed and variable request sizes there is no interest in the location and/or data itself that is being requested. That is, if s_i of the i-th I/O request is constant then the request size is considered fixed even if it contains different data with each run. The aspect of spatiality and timing of I/O requests is covered in the other dimensions of the I/O access pattern.

3.4.2 Type Of I/O Operation

An I/O operation can be typed as read, write or a mix of read and write. The type of an I/O operation results in different behaviour of the I/O architecture. Indeed some new I/O architectures might benefit one I/O operation type over others. Therefore, to understand the applications' I/O behaviour and it's impact on the underlying I/O subsystem, the quantity of each type should be evaluated. Furthermore, to analyse the type of I/O operations in relation to application's temporal behaviour, time intervals can be assigned a dominating I/O operation type.

Classification 2.4 Percentage of I/O type

The fraction of I/O type for read and write can be defined based on the number of I/O requests:

$$p_{\mathrm{r,IOPs}} = \frac{\sum_s D_{\mathrm{r}}(s, t_{\mathrm{end}})}{\sum_s (D_{\mathrm{r}}(s, t_{\mathrm{end}}) + D_{\mathrm{w}}(s, t_{\mathrm{end}}))} \tag{3.38}$$

$$p_{\mathrm{w,IOPs}} = \frac{\sum_s D_{\mathrm{w}}(s, t_{\mathrm{end}})}{\sum_s (D_{\mathrm{r}}(s, t_{\mathrm{end}}) + D_{\mathrm{w}}(s, t_{\mathrm{end}}))} \tag{3.39}$$

It can also be defined based on the amount of data read or written in bytes:

$$p_{\mathrm{r,Bytes}} = \frac{\sum_s sD_{\mathrm{r}}(s, t_{\mathrm{end}})}{\sum_s s(D_{\mathrm{r}}(s, t_{\mathrm{end}}) + D_{\mathrm{w}}(s, t_{\mathrm{end}}))} \tag{3.40}$$

$$p_{\mathrm{w,Bytes}} = \frac{\sum_s sD_{\mathrm{w}}(s, t_{\mathrm{end}})}{\sum_s s(D_{\mathrm{r}}(s, t_{\mathrm{end}}) + D_{\mathrm{w}}(s, t_{\mathrm{end}}))} \tag{3.41}$$

The percentage of I/O type can give a hint to the application's category in terms of I/O. For example, a simulation is likely to read a small amount of configurations and produce larger amounts of data to write. On the other hand a data analysis application is likely to read more data than it will write. However, caution should be observed when defining the I/O behaviour of the application in terms of percentage of I/O type. For example, the same data could have been re-read multiple times. Additionally, buffering, collective I/O and data sieving among others might change the concluded percentage of I/O type on different I/O stack layers. In general, the percentage of I/O type does not observe data distribution in space or time, yet it still offers a good insight into application I/O behaviour. In fact, disregarding data distribution in space and time

limits the percentage of I/O type to a single value per application. Thereby allowing for comparison of type of I/O operations across a large set of analysed applications.

Classification 2.5 Dominating I/O operation type

Each time interval Δt can be assigned a dominating I/O operation type when the ratio of I/O operation type exceeds a predefined threshold ϵ (with $0 < \epsilon \ll 1$). This creates a temporal distribution of the I/O operation type. Assigning a dominating I/O operation type to a time interval Δt can be done using the following ratios:

$$\text{Read} \quad if \quad \frac{\widetilde{D}_{\mathrm{r},\Delta\mathrm{t}}(t)}{\widetilde{D}_{\mathrm{r},\Delta\mathrm{t}}(t) + \widetilde{D}_{\mathrm{w},\Delta\mathrm{t}}(t)} > \epsilon \tag{3.42}$$

$$\text{Write} \quad if \quad \frac{\widetilde{D}_{\mathrm{w},\Delta\mathrm{t}}(t)}{\widetilde{D}_{\mathrm{r},\Delta\mathrm{t}}(t) + \widetilde{D}_{\mathrm{w},\Delta\mathrm{t}}(t))} > \epsilon \tag{3.43}$$

$$\text{Read/Write} \quad if \quad \frac{\widetilde{D}_{\mathrm{r},\Delta\mathrm{t}}(t) + \widetilde{D}_{\mathrm{w},\Delta\mathrm{t}}(t)}{\sum_s (D_{\mathrm{r}}(s, t_{\mathrm{end}}) + D_{\mathrm{w}}(s, t_{\mathrm{end}}))} > \epsilon \tag{3.44}$$

An alternative definition for the read/write domination is:

$$\text{Read/Write} \quad if \quad \frac{\widetilde{D}_{\mathrm{r},\Delta\mathrm{t}}(t)}{\widetilde{D}_{\mathrm{r},\Delta\mathrm{t}}(t) + \widetilde{D}_{\mathrm{w},\Delta\mathrm{t}}(t)} \approx 0.5 \tag{3.45}$$

$$OR$$

$$if \quad \frac{\widetilde{D}_{\mathrm{w},\Delta\mathrm{t}}(t)}{\widetilde{D}_{\mathrm{r},\Delta\mathrm{t}}(t) + \widetilde{D}_{\mathrm{w},\Delta\mathrm{t}}(t)} \approx 0.5 \tag{3.46}$$

It is possible to analyse the dominating I/O operation type's temporal distribution by visualizing the read or write I/O type ratio over time. The ratios given here are described using the number of I/O requests. Another option is to use the I/O quantity for evaluating the ratio of I/O operation type in a given time interval Δt. The suggested dominating I/O operation type is limited to the evaluation using the number of I/O requests. This reduces the number of values to examine in a large scale I/O analysis.

The ability to evaluate the I/O operation type on the time scale gives a deeper application I/O access pattern description. Knowing that an I/O burst occurs at a specific time in the application is even more useful when the dominating I/O operation type of the burst is known as well. Due to some I/O stack layers shifting the I/O request's

timing, the temporal distribution of dominating I/O operation type might change, when measured on different I/O stack layers.

3.4.3 Spatiality Of I/O Requests

The spatial component of an I/O request is it's address or location in the file or filesystem. As previously discussed access patterns can be analysed with a local, meaning process, node or I/O node view or using a global application view. Therefore, it is possible to distinguish between a local and a global spatiality of I/O requests. The first relates processes to files, while the second relates I/O requests in a given file to processes. The files used by an application can be defined according to the local spatial view as being either task-local or shared files. A task-local file is created, opened, read or written by only one process, while a shared file is operated on by all or some of the application's processes. Building on the local view, global spatial view is the location, i.e. the offsets, of I/O requests within a given file. It therefore combines both the process and the application view of I/O requests' location in a given task-local or shared file.

Classification 2.6 Task-local vs. shared

A file can be considered either task-local, i.e. accessed by one process, or shared, i.e. access by more than one process. This can be defined as:

$$Task-local \quad if \quad \sum_{p=0}^{P} n_{\mathrm{p,f}} = n_{\mathrm{p,f}} \tag{3.47}$$

$$Shared \quad if \quad \sum_{p=0}^{P} n_{\mathrm{p,f}} > n_{\mathrm{p,f}} \quad \text{for all } p \ (0 \leq p < P) \tag{3.48}$$

That is a file is task-local if the sum of all I/O operations from all processes to file f ($n_{\mathrm{p,f}}$), is equal to only one process's (p) I/O operations to the same file. On the other hand, the file is shared if the sum of all $n_{\mathrm{p,f}}$ from all processes is larger than any individual $n_{\mathrm{p,f}}$ of all processes. Using this definition a file is considered shared if more than one process accesses the file. Therefore the fraction of processes accessing a shared

file can be defined as:

$$a_{\mathrm{p,f}} = \begin{cases} 1 & \text{if} \quad n_{\mathrm{p,f}} > 0, \\ 0 & \text{else}, \end{cases}$$

$$A_{\mathrm{f}} = \frac{\sum_{p=0}^{P} a_{\mathrm{p,f}}}{P} \tag{3.49}$$

The resulting I/O behaviour and it's performance from using task-local or shared files highly depends on the filesystem and the I/O libraries employed. For example, shared files could suffer from file locks, leading to I/O requests from separate processes to be handled sequentially rather than in parallel. The handling of these file locks depends on the filesystem's implementation. Additionally, the use of task-local files' performance depends on the capability of the filesystem to handle the creation and opening of a large number of files. This issue becomes relevant with the fast and extreme increase of parallelism in modern HPC systems. As a result the metadata could become a bottleneck for task-local files depending on the filesystem's capability. This problem is addressed in [38], where SIONlib maps task-local files on a reduced number of physical shared files. It is worth noting that the use of SIONlib would result in changing the number of shared or task-local files on different I/O stack layers. Finally, many I/O libraries including MPI-IO change their behaviour attempting to optimize access for shared files. Due to the impact the filesystem and I/O libraries could have on task-local and shared files, their resulting I/O behaviour has to be analysed while occasionally considering the I/O architecture.

On the other hand, the use of task-local or shared files could open other design dimensions for the I/O architecture. For example the use of task-local might allow for some I/O subsystems to use local filesystems attached to each process or a subset of processes. This is because task-local files clearly separate I/O from different processes. However, such an implementation using task-local files could limit the use of collective I/O and data sieving.

It is possible to add a third category of files called single-file sequential, defined in [38]. This describes a file that is task-local, yet the owning process uses this file to read or write data on behalf of all other processes. The process using the task local file might go so far as being a designated I/O process that does not participate in computation. In

a sense it becomes an application I/O forwarding process. Single-file sequential works well for shared memory architectures, but is quite limited with distributed memory [38]. This category of files is not considered here since defining it requires observation of inter-process communication which is beyond the scope of the here given I/O criteria.

Classification 2.7 Spatial access pattern classification

Spatial access pattern observes the location or offset relation between a consecutive set of I/O requests ($0 \leq i < n_f$) that use the same file f (with $n_f \gg 1$). Following [39] it is possible to distinguish 5 types of spatial access patterns, defined as:

Sequential Sequential file access means for all consecutive requests in a series that:

$$R_{\text{off}}(p, f, i) \quad > \quad R_{\text{off}}(p, f, i+1) \tag{3.50}$$

Contiguous Contiguous file access is a special case of sequential access with:

$$R_{\text{off}}(p, f, i) \quad = \quad R_{\text{off}}(p, f, i-1) + s_{\text{i}} \tag{3.51}$$

where s_{i} is the request size of the i-th I/O request.

Simple strided access Simple strided access is another sequential access with:

$$R_{\text{off}}(p, f, i) = R_{\text{off}}(p, f, i-1) + o_i \tag{3.52}$$

where o_i is a constant stride and could be a multiple of the request size

Nested strided access Nested strided access is yet another sequential access, in which there are several levels of strides nested together. A two level strided access would therefore have an internal and an external stride and could be defined as:

$$R_{\text{off}}(p, f, i) \quad = \quad R_{\text{off}}(p, f, i_0) + o_i \left(1 - \delta_{\left\lfloor \frac{i}{n_{\text{s}}} \right\rfloor, \frac{i}{n_{\text{s}}}}\right) + l_{\text{s}} \left\lfloor \frac{i}{n_{\text{s}}} \right\rfloor \tag{3.53}$$

where:

$$
\begin{aligned}
i_0 &= \text{first I/O request in the access} \\
o_i &= \text{internal stride} \\
n_\mathrm{s} &= \text{number of requests in external stride} \\
\left(1 - \delta_{\left\lfloor \frac{i}{n_\mathrm{s}} \right\rfloor, \frac{i}{n_\mathrm{s}}}\right) &= \begin{cases} 0 & \text{if at start of external stride,} \\ 1 & \text{else,} \end{cases} \\
l_\mathrm{s} &= \text{external stride length} \\
l_\mathrm{s} \left\lfloor \frac{i}{n_\mathrm{s}} \right\rfloor &= \text{sets beginning of external stride}
\end{aligned}
$$

Random The relation between $R_{\mathrm{off}}(p, i, f)$ and $R_{\mathrm{off}}(p, i-1, f)$ is random and not a function of s.

Despite defining the spacial pattern classification given here for one process or a local view, the full concept behind simple and nested strided access can only be seen from a global or application view. An example for simple strided access is given in Fig. 3.1. Shown is the commonly found simple strided access of parallel I/O for P processes reading or writing consecutive chunks of data of length s with $o_i = Ps$.

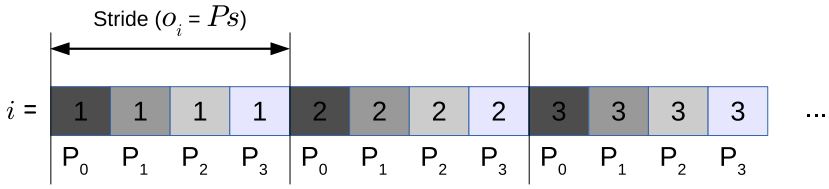


FIGURE 3.1: Example of simple strided access of a file f in a parallel application using $o_i = Ps$. Adapted from [39]

Defining nested strided access can be done using Eq. 3.53. It can also be imagined as a substitution of one request from a process with another level of strided access [39]. The example shown in Fig. 3.2 has an internal stride for each process of $o_i = 2s$, an external stride length of $l_\mathrm{s} = 12s$ and $n_\mathrm{s} = 2$ requests per external stride. The nested strided access pattern is used by many scientific applications [39].

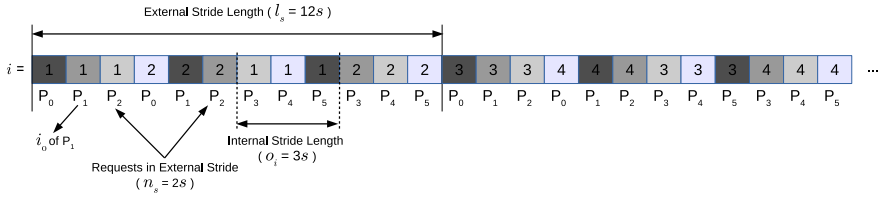


FIGURE 3.2: Example of nested strided access of a file f in a parallel application using $n_\mathrm{s} = 2s$, $l_\mathrm{s} = 12s$ and $o_i = 3s$ where s is a uniform request size. Adapted from [39]

Analysing spatial patterns on the fly from an application I/O request set is difficult. The major problem is the need to link every I/O request to its process, file and offset. This tends to drastically increase the size of the I/O analysis data collected and require implementation of complex pattern detectors. It is therefore preferred to analyse spatial access patterns from the application's I/O algorithm.

In [24] both post-execution and runtime analysis are used to predict future I/O. Therefore understanding the spatial patterns commonly exhibited by analysed applications could allow for designing prefetching or readahead of I/O requests. However, prefetching requires an application with a regular, consistent and predetermined I/O access pattern, that can be easily detected and/or signalled to the I/O subsystem. The I/O subsystem has to then be able to react and subsequently prefetch the data as needed to achieve any performance improvements. Another option for an I/O subsystem to use spatial patterns exhibited by applications is to improve data storage. For example, understanding the spatial pattern could lead the filesystem to change the data allocation, thereby allowing for an improvement of data write or later it's read back. Indeed, some local filesystems already allocate adjacent blocks to written ones in anticipation of sequential contiguous file access, which has an improved performance on HDD [37, Chapter 5]. Additionally, I/O subsystems can improve collective I/O or data sieving if the spatial pattern of an application is known. However, prior to altering the system to accommodate any special access patterns these have to be established as being common among applications.

3.4.4 Temporal Intervals

Temporal intervals analyses an application in time, compared to spatiality which analyses the application in space. By observing the application's I/O behaviour over time it is possible to judge the uniformity of I/O over the execution time. It could be expected that many application's exhibit periods of intense I/O that might delay computation. By detecting and applying various techniques such as asynchronous I/O prefetching and burst buffers it is possible to overlap computation and I/O.

Classification 2.8 Temporal distribution of I/O

The temporal distribution of I/O considers the total number of I/O requests and total amount of data read or written during a given interval $[t - \Delta t, t]$ as a function of time and can be defined as:

$$S_{\mathrm{r,w}}(t) = \sum_{s} \left(\widetilde{D}_{\mathrm{r},\Delta\mathrm{t}}(s,t) + \widetilde{D}_{\mathrm{w},\Delta\mathrm{t}}(s,t) \right) \tag{3.54}$$

$$N_{\mathrm{r,w}}(t) = \sum_{s} \left(\widetilde{D}_{\mathrm{r},\Delta\mathrm{t}}(s,t) + \widetilde{D}_{\mathrm{w},\Delta\mathrm{t}}(s,t) \right) \tag{3.55}$$

The temporal distribution of I/O can also be split into read and write, i.e. splitting $S_{\mathrm{r,w}}(t)$ into $S_{\mathrm{r}}(t)$ and $S_{\mathrm{w}}(t)$ for read and write respectively, similarly splitting $N_{\mathrm{r,w}}(t)$ into $N_{\mathrm{r}}(t)$ and $N_{\mathrm{w}}(t)$. A visualization of I/O distribution overtime could be very descriptive of the I/O behaviour of an application. Indeed the value of $S_{\mathrm{r,w}}(t)$ and $N_{\mathrm{r,w}}(t)$ would show the bandwidth and IOPS respectively required over time. Meanwhile the width of a burst of I/O on an $S_{\mathrm{r,w}}(t)$ plot against time would show the magnitude and could therefore define the size and external or internal bandwidth of burst buffers needed to hide such a burst.

Classification 2.9 Burstiness parameter

A burst can be considered a sudden increase in I/O with a short duration. The burstiness parameter attempts evaluating the overall burst behaviour of an application. Considering $H_{\Delta\mathrm{t}}(t)$ defined in Eq. 3.37. It is possible to define a burstiness parameter for read

and write as:

$$\rho = \begin{cases} 1 - \tanh\left(\frac{l_{\mathrm{IO}}}{l_{\mathrm{noIO}}}\right) & \text{if} \quad l_{\mathrm{noIO}} > 0, \\ 0 \quad \text{else.} \end{cases} \tag{3.56}$$

Where:

$$l_{\mathrm{IO}} = \text{Average of consecutive time slices } t \text{ during which } H_{\Delta \mathrm{t}}(t) > 0$$
$$l_{\mathrm{noIO}} = \text{Average of consecutive time slices } t \text{ during which } H_{\Delta \mathrm{t}}(t) = 0$$

The used equation assumes that a burst of I/O is a continues time slice where either read or write is above a given threshold c. The value of the threshold c is used to evaluate $h_{\Delta \mathrm{t,p}}(t)$ in Eq. 3.30, which is subsequently used to evaluate $H_{\Delta \mathrm{t}}(t)$ in Eq. 3.37. As l_{IO} or the average duration of I/O bursts increases, ρ tends towards 0. In this case I/O is more distributed over the execution time. Meanwhile when l_{noIO} or the average duration of no I/O increases, ρ tends towards 1, meaning that I/O appears in bursts.

The burstiness parameter can be considered a quantifiable variable informing on the temporal distribution of I/O (classification 2.8). It is therefore an attempt to define a simple and relevant indicator from a potentially complex variable time series. Which is needed as the use of a visualization of I/O overtime is difficult when comparing different applications and matching the I/O behaviour to a suitable I/O architecture.

As the name indicates burst buffers [1] are used to catch and hide the delay of I/O bursts. Understanding the burstiness of an application could indicate the usefulness and specification of burst buffers. For example, longer and bigger bursts require larger buffers to hide them. Therefore by observing the burstiness parameter in relation with I/O intensity (Classification 1.6), bandwidth (Classification 1.3) and IOPS (Classification 1.4) it is possible to quantify the suitability of burst buffers, asynchronous I/O and prefetching. A bursty application with sufficient time between bursts can benefit from asynchronous I/O overlapped with computation. It can also benefit from a high bandwidth to a burst buffer, while coping with a lower bandwidth to the external storage. To allow equal benefits to read as much as write from these I/O architectures might require the use of prefetching for read data to the available buffers. The prefetching can be triggered by

the application asynchronously or initiated by a spatial pattern recognition as explained in classification 2.7. Filesystems and I/O libraries can also benefit from understanding the exhibited burst behaviour of applications as quantified by the burstiness parameter.

3.4.5 Repetitive Behaviour

Repetition indicates that a certain I/O pattern will repeat itself. The pattern does not need to occur in a constant periodic fashion to be considered repetitive. Repetition can therefore take place in either space or time. For example, when two files are accessed using the same access pattern, the application is said to exhibit a spatial repetitive behaviour. Finding repetition can give the opportunity of using prefetching or prolong buffering or caching of the read data.

Classification 2.10 Access pattern repetitive behaviour

A repetitive access pattern would mean that within a sequence of requests $(0 \leq i < n)$the following is found:

$$R(p, i, f) \quad = \quad R(p, i + k\Delta, f') \tag{3.57}$$

For:

fixed Δ

$k = 1, 2, 3, ...$

and potentially different f and f'

Two I/O requests are the same given in Eq. 3.57 if either both involve the same data in which case $f = f'$, or both use the same offsets within the given files, i.e. $R_{\text{off}}(p, i, f) = R_{\text{off}}(p, i+k\Delta, f')$. If the same data is involved in two read requests caching and buffering could make use of the repetition. However, if the same offsets, i.e. the same spatial access pattern is used, prefetching can employ this knowledge to overlap I/O with computation.

It is also possible to consider the case for $R(p, i, f) = R(p', i + \Delta, f)$ where $p \neq p'$. That is two processes read the same data using the same access pattern at different occasions. Processes with shared memory could then also benefit from such an access

pattern repetition by prolonging caching and/or buffering of the read data. A burst buffer can also use this information to prolong data buffering.

As detection of access pattern repetition requires the analysis of spatial access patterns, the same difficulties apply. Not only is it necessary to link each I/O requests with its process, file and offset, but it is also required to detect and be able to compare the resulting access patterns. It is therefore preferable to analyse the repetition through careful study of application's I/O algorithm. Once a repetition is detected, the challenge of informing the I/O subsystem or detecting the pattern and its repetition remains. The I/O subsystem would also need modification to be able use techniques for improving the access once a repetitive access is detected.

Classification 2.11 Dominating I/O operation repetitiveness

Defining the dominating I/O operation type is given by the classification 2.5. To identify a repetitiveness in the I/O operation type the count of number of occasions where during a time interval $\Delta t'$ the same I/O operation type dominates can be defined as:

$$n_{\mathrm{x},\Delta \mathrm{t}'}(\delta t) \tag{3.58}$$

Where:

$x =$ r (read), w (write) or rw (read/write)

and the same I/O operation dominates at t_1 and $t_2 = t_1 + \delta t$

here $(t_{\mathrm{end}} \leq t_1 < t_2 \leq t_{\mathrm{start}})$

Detecting a repetition of dominating I/O can be done by plotting $n_{\mathrm{x},\Delta \mathrm{t}'}(\delta t)$ for a given $\Delta t'$ as a function of δt. It is worth noting that $\Delta t \neq \Delta t'$, where Δt can be considered the time interval between the discrete measured values and $\Delta t'$ is a portion of the application's runtime duration. It is possible to propose that $\Delta t < \delta t < \Delta t' \leq t_{\mathrm{end}} - t_{\mathrm{start}}$.

It can be expected from many scientific applications to exhibit a repetitive dominating I/O operation type. These applications could be using the pattern *read - compute - write*. For such a cyclic pattern, overlapping computation and asynchronous I/O could reduce I/O time.

3.5 Category 3: Parallel I/O

Parallel I/O describes the behaviour of the combination of the I/O of individual processes in the temporal dimension. As previously discussed access patterns can be divided into local and global [24]. Local access patterns describe the I/O of a single process. Global access pattern describes the I/O of a parallel application across it's multiple of processes to a single or multiple of files. In Sec. 3.4 the I/O access patterns (Category 2) have analysed the spatial access pattern of both the local and global application views. The temporal behaviour however, is analysed on the local view only and does not account for the combination of process I/O. As a result, the application has to be analysed for parallel I/O.

As introduced before, for some I/O subsystems analysing process I/O is not relevant or feasible, thereby redefining local I/O as node or I/O node I/O. Since in many cases the number of nodes or I/O nodes is smaller than the number of processes on a given HPC system, the redefinition of local I/O changes the depth of available measurable I/O parallelism. This should be observed when analysing the parallel I/O behaviour of applications.

Classification 3.1 Parallel I/O intensity

The parallel I/O intensity can be defined as the number of time intervals Δt during which more than one process performs read or write versus the total number of time intervals during which I/O operations are performed. By this definition a real-time concurrency is meant.

Considering $h_{\Delta \mathrm{t,p}}(t)$ defined in Eq. 3.30 the following quantity can be defined:

$$\rho_{\Delta \mathrm{t}}(t) = \frac{\sum_p^P h_{\Delta \mathrm{t,p}}(t)}{P} \tag{3.59}$$

Eq. 3.59 defines the fraction of processes which during the time interval $[t - \Delta t, t]$ participated in the I/O access. Using $\rho_{\Delta \mathrm{t}}(t)$ and $H_{\Delta \mathrm{t}}(t)$ defined in Eq. 3.37, it is possible to define the unnormalized parallel I/O intensity as:

$$P'_{\mathrm{IO},\Delta \mathrm{t}} = \frac{\sum_t \rho_{\Delta \mathrm{t}}(t)}{\sum_t H_{\Delta \mathrm{t}}(t)} \tag{3.60}$$

If in all time intervals where I/O is performed all process are involved $P'_{\mathrm{IO},\Delta\mathrm{t}} = 1$ is maximized. $P'_{\mathrm{IO},\Delta\mathrm{t}} = 1/P$ is the minimal value, meaning that during all time intervals with I/O only a single process was involved.

The normalized parallel I/O intensity can be then defined as:

$$P_{\mathrm{IO},\Delta\mathrm{t}} = \frac{P'_{\mathrm{IO},\Delta\mathrm{t}} - 1/P}{1 - 1/P} = \frac{PP'_{\mathrm{IO},\Delta\mathrm{t}} - 1}{P - 1} \tag{3.61}$$

The extreme case of no I/O, that is $h_{\Delta\mathrm{t,p}}(t) = 0$ for all t will lead to a parallel I/O intensity of -1. This can be the result of choosing a too high I/O threshold c in Eq. 3.30 or analysing an application with too little I/O. Although an application with no or little I/O that is analysed using a well defined threshold c is uninteresting, this case has to be handled for analysis of a large number of applications. As a result Eq. 3.61 can be extended:

$$P_{\mathrm{IO},\Delta\mathrm{t}} = \begin{cases} 0 \quad if \quad P'_{\mathrm{IO},\Delta\mathrm{t}} = 0, \\ \frac{PP'_{\mathrm{IO},\Delta\mathrm{t}} - 1}{P-1} \quad \text{else.} \end{cases} \tag{3.62}$$

$P_{\mathrm{IO},\Delta\mathrm{t}}$ is a quantity in the range $0 \leq P_{\mathrm{IO},\Delta\mathrm{t}} \leq 1$. Where a $P_{\mathrm{IO},\Delta\mathrm{t}} = 0$ indicates that only one process does I/O at any given time interval Δt. $P_{\mathrm{IO},\Delta\mathrm{t}} = 1$ indicates that for any time interval Δt I/O is performed by all processes. $P_{\mathrm{IO},\Delta\mathrm{t}}$ is a measurement of real-time parallelism. That is, $P_{\mathrm{IO},\Delta\mathrm{t}}$ gives no indication as to whether all processes doing I/O in parallel are involved in the same I/O task. Whether the application uses collective I/O spanning over all process is not measured. Timing and duration of I/O is changed over different I/O stack layers, thereby changing the parallel I/O intensity. I/O forwarding also changes the number of processes that access the I/O system and therefore changes the possible depth of I/O parallelism.

The $P_{\mathrm{IO},\Delta\mathrm{t}}$ gives the ability to understand the real parallel behaviour, which is a result of I/O libraries, parallel filesystem and application interprocess synchronization. Recognizing the efficiency of applications in doing real parallel I/O is useful for measuring the parallel load the filesystem has to endure.

Classification 3.2 I/O operation concurrency

I/O operation concurrency $\rho_{\Delta t}(t)$ shows the number of processes involved in the I/O operations over time. $\rho_{\Delta t}(t)$ is defined in Eq. 3.59.

While parallel I/O intensity (Classification 3.1) defines the total parallelism of an application, I/O operation concurrency defines parallelism at a given time. The advantage of using parallel I/O intensity is the ability to analyse a large set of applications and compare the I/O parallelism using a single value. Meanwhile, I/O operation concurrency is a temporal distribution representing the parallel I/O behaviour and can be used to further analyse a smaller set of applications' I/O parallelism.

Classification 3.3 Parallel I/O distribution

The aggregate performance numbers (Category 1) described in Sec. 3.3 analyses the total performance numbers of the application. Showing the contribution of each process to the total performance numbers allows for observing the distribution of I/O on the individual processes. Table 3.1 shows the performance numbers that can be measured for each process.

Number of read operations	$N_{\mathrm{p,r}} = \sum_s D_{\mathrm{r}}(p, s, t_{\mathrm{end}})$
Number of write operations	$N_{\mathrm{p,w}} = \sum_s D_{\mathrm{w}}(p, s, t_{\mathrm{end}})$
Amount of read data	$S_{\mathrm{p,r}} = \sum_s sD_{\mathrm{r}}(p, s, t_{\mathrm{end}})$
Amount of write data	$S_{\mathrm{p,w}} = \sum_s sD_{\mathrm{w}}(p, s, t_{\mathrm{end}})$
Number of open operations	$F_{\mathrm{p,open}}$
Number of close operations	$F_{\mathrm{p,close}}$

TABLE 3.1: Parallel I/O distribution metrics.

Together there are three I/O criteria for temporal parallelism of I/O. The first is parallel I/O intensity (Classification 3.1) and represents the overall I/O parallelism of an application. The second is I/O operation concurrency (Classification 3.2) and represents ratio of I/O parallelism overtime. Finally, the third is parallel I/O distribution (Classification 3.3) and defines the magnitude of contribution from each process to the application's I/O. The parallel I/O distribution can also be represented as a temporal distribution

of the performance numbers. This is done by exchanging the aggregate performance numbers with their temporal counterparts for each process.

Classification 3.4 Same file access concurrency

Same file access concurrency level indicates the number of processes accessing a given file at the same time versus the total number of processes. The following definition is needed for each process p:

$$a(p,f,t) \quad = \quad \begin{cases} 1 & \text{if} \quad n_{\mathrm{p,f}}(t) > 0, \\ 0 & \text{else}, \end{cases} \tag{3.63}$$

where $n_{\mathrm{p,f}}(t)$ is the total number of I/O requests from process p to file f during the time interval $[t - \Delta t, t]$. Then the same file access concurrency can be defined as:

$$A(f,t) \quad = \quad \frac{\sum_p^P a(p,f,t)}{P} \tag{3.64}$$

The definition in classification 2.6 of task-local versus shared only labelled files and set the fraction of processes accessing a shared file. Same file access concurrency however gives the fraction of processes accessing a file over time. To avoid racing conditions the I/O stack has to implement various synchronization and locking mechanisms. These in turn can impact performance when shared files are used. The extend of that impact can be concluded from the same file access concurrency. A file accessed by all processes at different times could induce less synchronization effects than a file always accessed by all processes at the same time. The same file access concurrency can be considered to analyse the spatial component of an application's parallel I/O.

3.6 Summarizing I/O Criteria

The introduction of a collective I/O criteria, allows for analysing I/O behaviour of a single or a large set of applications using a standardized formulation. Studying a large set of applications' I/O allows for analysing and further understanding the overall I/O behaviour of scientific applications on modern HPC systems and how these operate on the given I/O architecture. Using this information new I/O architectures that would

redesign the I/O subsystem can be described to better accommodate the I/O requirements of scientific applications. However, this analysis process should consider that the I/O behaviour changes as the I/O subsystem offers different capabilities. Therefore, evaluating overall application I/O behaviour can be complemented by further analysing a subset of I/O intensive applications. The I/O criteria overall analysis of I/O behaviour would facilitate identifying the relevant subset of I/O intensive applications that can be further analysed. The I/O criteria can also be used for evaluating I/O behaviour of individual applications and possibly selecting suitable optimization techniques.

The I/O criteria presented here can be grouped into single values such as total amount of data read/written (Classification 1.1) and I/O distributions over time, request sizes or processes, which are listed below:

- Classification 2.1 Distribution of request sizes
- Classification 2.5 Dominating I/O operation type
- Classification 2.8 Temporal distribution of I/O
- Classification 3.2 I/O operation concurrency
- Classification 3.3 Parallel I/O distribution
- Classification 3.4 Same file access concurrency

When analysing a large set of applications an additional dimension is added to the analyses process namely the analysed applications. In such a case single I/O criteria values are best visualized as a distribution over the applications. On the other hand, I/O distributions are difficult to visualize and/or analyse for a larger set of applications. The I/O criteria are designed to overlap and some single values illustrate some idea of their I/O distribution counterparts. For example, analysing the temporal distribution of I/O (Classification 2.8) for a large set of analysed applications is partly covered by the burstiness parameter (Classification 2.9).

Once the large set of applications are analysed and the I/O criteria evaluated, it is possible to pick a smaller representative subset of applications. These would facilitate the demonstration of I/O behaviour and allow for visualizing the I/O distribution criteria. Tab. 3.2 provides an overview for all single value I/O criteria and can be used in relation

to I/O distribution criteria to further analyse the I/O behaviour of the subset. To complement the analysis process further application information must be provided, this includes number of process P and runtime duration represented by time when execution of application starts t_{start} and ends t_{end}.

			Read	Write
1	Aggregate performance numbers			
1.1	Total amount of data		S_{r}	S_{w}
1.2	Total amount of IOPs		N_{r}	N_{w}
1.3	Bandwidth	Max	$B_{\text{max,r}}$	$B_{\text{max,w}}$
		Avg	$\overline{B}_{\text{r}}$	$\overline{B}_{\text{w}}$
1.4	IOPS	Max	$\Gamma_{\text{max,r}}$	$\Gamma_{\text{max,w}}$
		Avg	$\overline{\Gamma}_{\text{r}}$	$\overline{\Gamma}_{\text{w}}$
1.5	Total number of files created		F_{create}	
1.6	I/O intensity		I_r	I_w
2	I/O pattern analysis			
2.2	Percentage of small I/O requests		$f_{\text{r}}(s_{\text{small}})$	$f_{\text{w}}(s_{\text{small}})$
2.3	Number of request with variable size		$\sum_i \sigma_r(i)$	$\sum_i \sigma_w(i)$
2.4	Percentage of I/O type	in IOPs	$p_{\text{r,IOPs}}$	$p_{\text{w,IOPs}}$
		in Bytes	$p_{\text{r,Bytes}}$	$p_{\text{w,Bytes}}$
2.6	Task-local vs. shared	Task-local	$\sum f_{task-local}$	
		Shared	$\sum f_{shared}$	
2.7	Spactial access pattern		Exhibited spatial patterns	
2.9	Burstiness		ρ_r	ρ_w
2.10	Access pattern repetitiveness		yes (period Δ) or no	
2.11	Dominating I/O repetitiveness		yes (period δ) or no	
3	Parallel I/O			
3.1	Parallel I/O intensity		$P_{\text{IO},\Delta\text{t}}$ for read	$P_{\text{IO},\Delta\text{t}}$ for write

TABLE 3.2: Analysis map listing all single value I/O criteria to facilitate application analysis.

Chapter 4

Performance Characterization: Analysing GPFS I/O Counters

The GPFS I/O counters are explained in section 2.2.1 and were used to collect the GPFS I/O logs[1] for a period of approximately 19 months on JUGENE. Every 120 sec the counters are logged for every I/O node. Additionally a large database containing all jobs that ran on JUGENE over the same time period is kept. By equating the counters with the start and end time of the application's run a large scale analysis of I/O can be performed. The analysis process presented here uses the I/O criteria described in Chp. 3 to evaluate the I/O behaviour of applications as seen by the GPFS I/O counters. This allows for demonstrating the I/O criteria as an investigative tool, while commenting on the relationship between analysis and measuring of I/O.

4.1 Related Work And I/O Profiling Tools

As the computation increases along with the available data storage, the awareness for the need of investigating and optimizing I/O increases. As a result, many analysis and profiling tools are created or updated to trace and log the I/O behaviour of scientific applications on HPC systems. Part of the necessity to using various tracing tools is rooted in the complexity of the I/O stack. It therefore becomes inherently difficult to

[1] The term **GPFS I/O logs** is used to indicate the resulting data from logging the GPFS I/O counters periodically.

relate application I/O requests or behaviour to reactions of individual I/O stack layers or total performance of the I/O subsystem [13][41][42].

Three steps can be attributed to the process of any I/O monitoring and/or profiling technique. First is selecting a method of data collection, thereby deciding on the data resolution (i.e. amount or type of data to collect) and I/O stack layer to measure I/O on. Second is analysing or deciding on an analysis process for the collected data, which includes the goal or behaviour to be analysed. Third and final is the use of the gained knowledge towards improving I/O or application performance, system operation[2] or cost. Although all would agree on the need for I/O tracing, the approach taken by many differs. As a result, not all techniques involve all three steps for I/O monitoring and profiling. Despite the need to practically perform the three steps in order, many interchange the order of selecting the methods or goals involved in them.

The following describes some methods and tools used in the process of I/O monitoring and/or profiling. These methods are dissected across the different approaches they take to the three steps. The list given and discussed here is by no means exhaustive. Yet it allows for comparing the efforts made in this study with others. Additionally, it evaluates some of the complementary work done.

4.1.1 I/O Measuring Tools

As inferred by first introducing the I/O criteria the approach taken here first decides on the analysis process independent of the I/O measuring technique[3]. Many have a different approach. In it the data collection method is created first and only later the decisions are made on the needed analysis process. An example of such approach includes many profiling tools, which instrument key functions to monitor and collect the relevant events. Profiling of applications is regularly used to investigate performance bottlenecks. Many tools were either extended or created to allow for instrumenting of I/O calls. One such tool is VampirTrace [43]. Here all I/O events are recorded with varying details.

[2]Some studies use monitoring of system I/O for detecting performance degradation due to failing hardware or to automatically steer the system, this approach is taken in Scalable I/O for Extreme Performance (SIOX) [13]

[3]The introduction of I/O criteria mentions the limitations that the measuring method has on the evaluation and analysis process. It also confirms that the selection of which I/O criteria to evaluate is relevant to the measuring method. Despite the ability to interchangeably select either the analysis target or the measuring technique first, both are intricately linked and the decisions related to each effects the other.

This allows for a full coverage of all I/O events and the related information, thereby opening the possibility of full analysis of application I/O behaviour. Profiling tools can also provide sophisticated methods for visualizing collected data, along with statistics of the I/O calls. For example, the Vampir tool set provides Vampir and VampirServer for visualization of VampirTrace collected data [43]. Nonetheless, the actual analysis and investigation of I/O behaviour of application is mostly left to the application developers.

Many I/O profiling or tracing tools leave instrumentation of application to users. As a result, only individual applications have their I/O measured as users employ the given I/O profiling tool. An alternative, used by DARSHAN, is to instrument user space libraries, thereby not requiring source code changes or instrumentation of individual applications [10]. Nonetheless, I/O profiling tools require the instrumentation of events to be observed. Due to the availability of various functions to users for performing I/O, tool developers need to create appropriate instrumentation for all of these functions. Otherwise, I/O events will be missed and subsequently not logged in the collected data. For example, DARSHAN cannot monitor a job that does not use MPI [14].

Another method for data collection on I/O behaviour is the use of system counters. Many of the I/O subsystem components implement counters that regularly collect statistics. These can be either used or modified to regularly log their values. In [44] the RAID controllers' tools for querying performance and status is periodically polled and logged. Meanwhile, [12] complements the DARSHAN collected data by using system level tools to collect storage device activity and filesystem contents. These and others are similar to the method used for collecting the GPFS I/O counters. The approach allows for coverage of all I/O operations as seen by the logged counter on the specific I/O stack layer. However, matching logged I/O to applications requires additional effort. In some cases, the relationship to the application is ignored. For example, in [44] the analysis of the RAID controllers' counters have not been linked to specific applications.

A final method for data collection is to create dedicated systems. The Scalable I/O for Extreme Performance (SIOX) creates a separate system that can collect I/O data on all I/O layers for each I/O request [13].

Resolution of Data Collection

Due to the high resolution of data collection, profiling tools create a large quantity of data during application runtime. This problem becomes more visible as the number of recorded I/O events and the number of processes per application increase, which is a common trend in today's HPC systems. The produced data could therefore limit scaling the trace to larger runs of an application or investigating a large number of application I/O behaviour. Furthermore, only instrumented applications will have their I/O traced. As a result, these tools are mostly limited to the analysis of individual applications.

Some profiling tools attempt eliminating the obstacles for data scalability by improving the I/O performance when storing I/O traces. For example, in [45] I/O forwarding techniques are used to improve the I/O of Vampir. Another option for allowing data scalability of profiling tools is decreasing data produced by reducing the number of recorded events necessary to recreate an I/O trace of the instrumented application [46]. Meanwhile, ScalaIOTrace uses event compression and aggregation of timing information [47]. To reduce the data produced during I/O profiling, DARSHAN only records overall statistics of the I/O events. Therefore, DARSHAN can be used for mass I/O analysis of application's access patterns [10].

Usually when using system counters to log I/O information, the logged instances are not related to I/O events. Rather system counters are periodically captured. The data resolution is therefore temporal and the period of logging can be changed to increase or decrease the resulting data size. The GPFS I/O counters used here are logged with a period of 2min.

Since SIOX attempts the investigation of all I/O events on all I/O stack layers, the data resolution should be high. As result, a large quantity of data could be collected. This requires the implementation of a specialized system that can coup with a flood of data [13].

Measuring I/O Stack Layer

Application profiling tools measure and instrument the I/O behaviour as created by the application itself. Indeed, DARSHAN developers in [10] consider this a necessity for understanding the application's interaction with the storage system, arguing that I/O

behaviour is changed as it moves through I/O stack layers. Nonetheless, to increase data captured while using DARSHAN, [12] simultaneously collects storage device activity and filesystem contents. In comparison, [44] characterizes the workload of storage systems by only collecting data from RAID controllers.

Meanwhile, many attempt investigating the full operation of the I/O stack layers. SIOX create a system for the collection and linking of I/O events on all I/O stack layers simultaneously [13]. IOVIS attempts catching and visualizing the full end-to-end events triggered by an I/O request [41]. In [42] the user specifies the portion of the application to be instrumented and an automatic tracing of the I/O stack is performed. ScalaIOTrace can be extended to collect I/O events on any I/O stack layer [47].

As described in Sec. 2.2.1 the GPFS I/O counters are logged on the filesystem layer. The I/O criteria as an analysis tool described in Chp. 3 attempts to be I/O stack layer and architecture agnostic. It therefore recognizes the effect of the I/O stack layer, but can fit I/O collected on most of them. Analysing the GPFS I/O logs as provided here can be considered a demonstrative tool for the I/O criteria and it's ability to investigate the I/O behaviour of individual or a large number of applications.

4.1.2 Analysis Process

As mentioned, the analysis process, as in the goal or I/O behaviour to be analysed, is in many cases selected after collection of I/O traces. A main reason for choosing what to analyse prior to monitoring the data is the quest for observing specific I/O behaviours. For example, in [24] and [48] a classification of access patterns is offered, which is then used to analyse the collected I/O signatures. The reason for this predetermined I/O analysis is to offer improved prefetching.

As mentioned most profiling tools offer visualization of collected data for the user to perform the analysis. However, since DARSHAN offers the mass collection of application I/O it is used for long term characterization of I/O behaviour [12][14]. This most resembles the approach for mass application analysis taken here. Indeed, some of the findings in [14] and [12] correlate with the analysis results using the GPFS I/O counters to evaluate the I/O criteria. Comparing the two approaches reveals some complementary results. For example, the investigation using DARSHAN in [14] lacks the temporal

behaviour of I/O, which can be analysed from the GPFS I/O logs. The analysis in [12] attempts simultaneously collecting storage device activity and filesystem contents to retrieve some temporal I/O information, such as burstiness. On the other hand, the GPFS I/O logs almost looses all spatial and metadata information which are analysed in [14].

4.1.3 Using Analysis Information

Admittedly the overall target from all tools or methods for monitoring or profiling I/O is recognising and understanding the I/O behaviour of applications and/or the I/O subsystem. Nonetheless, the method by which this understanding can improve performance or cost differs. There are two main approaches, distinguished by which part is to be optimized. While one optimizes the application, the other optimizes the I/O subsystem on which the application runs.

Profiling tools can target the optimization of the instrumented applications. This is true for the methods used in [42], Vampir [43] and DARSHAN [10]. Application developers can be supported by hints to possible and available optimizations. The large scale analysis using DARSHAN performed in [14] adds the ability for selecting the applications that necessitate optimization. There the overall gain for the system is considered a result of finding an eliminating applications with less than optimal I/O performance.

The second approach is improving the I/O subsystem by analysing I/O. Existing parameters of the I/O subsystem can be improved to fit application I/O behaviour. In [24] the access pattern is analysed and fed to a tool-kit that performs prefetching. SIOX intends finding I/O bottlenecks and proposing optimizations for the I/O middleware [13].

The I/O criteria as evaluated by the GPFS I/O logs can be used to improve application and I/O subsystem performance. The main focus as presented in Chp. 3 is understanding I/O behaviour to possibly suggest improvement from different I/O architectures. These improvements are sparsely mentioned when analysing the GPFS I/O logs.

4.2 Reformatting GPFS I/O Counters

Tab. 4.1 shows the information logged by the I/O counters. Due to the crude form of logging, the data has to be reformatted to facilitate analysis. The reformatting entailed creating discrete values from the logged accumulative GPFS I/O counters and dealing with counter resets occurring at seemingly random points in time. To achieve a discretized data set each log has to be subtracted from it's previous value or the previous log. As a consequence the first recorded value has to be set to zero. In some cases the time between logs (Δt) is longer than 120sec, possibly due to maintenance periods. To keep track of such instances and other time factors, a record of $\Delta t = t_{current} - t_{prev}$ is kept for each log. This allows tracing the log back to the previous log used in the subtraction. Fig. 4.1 shows an example of the analysis time line for a single filesystem. Although the analysis encounters a logged data point at the very beginning, it cannot be used as there is no previous counter value available. The counters are therefore logged as zero for that first point and the I/O requests in region (1) are lost. Meanwhile the logged values in both region (2) and (5) can be discretized as each log has a previous value.

Keyword	Description
_t_	Indicates the current time of day in seconds (absolute seconds since Epoch (1970)).
_fs_	The name of the filesystem for which data is being presented.
_br_	Total number of bytes read, from both disk and cache.
_bw_	Total number of bytes written, to both disk and cache.
_oc_	Count of open() call requests serviced by GPFS. This also includes creat() call counts.
_cc_	Number of close() call requests serviced by GPFS.
rdc	Number of application read requests serviced by GPFS.
_wc_	Number of application write requests serviced by GPFS.
dir	Number of readdir() call requests serviced by GPFS.
_iu_	Number of inode updates to disk.

Table 4.1: GPFS I/O counters [31].

Another factor in reformatting the GPFS I/O logs are counter resets leading to loss of information. Fig. 4.1 shows a reset point which leads to I/O requests not being logged. Region (3) I/O requests are lost due to resetting the GPFS I/O counters. To increase benefit from I/O data analysis resets can be used as previous values for later logs. Opposite to region (1)'s I/O requests that were lost, region (4)'s I/O requests can be discretised using the time-stamp logged for the GPFS I/O counter reset. Indeed most

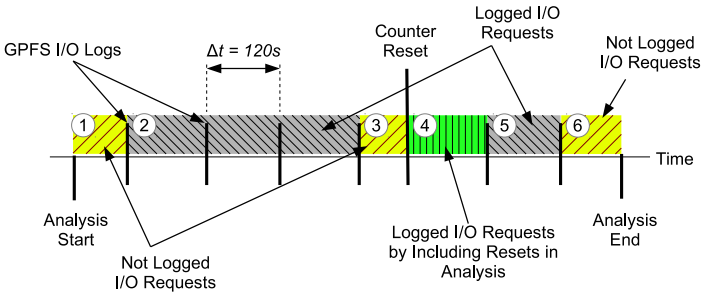


FIGURE 4.1: Analysis of GPFS I/O log time line where regions 1,3 and 6 represents I/O requests that have not been logged, regions 2 and 5 are logged I/O requests and region 4 is logged by including resets in analysis.

found of the logging start with a GPFS I/O counter reset, leading to region (1) losses to be minimized. Finally region (6) was not logged and therefore has no information recorded on the I/O requests.

Tracking the changes to the I/O data and the correctness of the reformatting process is challenging. GPFS I/O counters do not just change their behaviour over time with resets, they also change in space over multiple filesystems. On each I/O node multiple I/O filesystems are accessible and for each a separate GPFS I/O counter set exists. Therefore the process of forming discrete values from the accumulative GPFS I/O counters has to be repeated separately for each filesystem.

Due to the size of the GPFS I/O logs, it is important to implement optimized analysis scripts. Optimized scripts allow for analysing the complete data set within a reasonable time frame. The major difficulty with reformatting the data is not altering the information contained in the GPFS I/O logs. The later analysis and the correctness of the conclusions built on this set of GPFS I/O logs depends on keeping the information as accurate as possible. Therefore the analysis process has to check and compare results with the original GPFS I/O logs. Checking a random sample of GPFS I/O counters by hand adds to checking the GPFS I/O logs reformatting correctness.

Originally the GPFS I/O counters are periodically logged into text files. To analyse the I/O behaviour of an application the complete file of GPFS I/O logs has to be parsed to extract the relevant time span. The bottleneck for such a file parse is the

amount of data that has to be read in before the analysis can be conducted. Most of the GPFS I/O logs will then be discarded, since these logs lie outside the runtime of the application to be analysed. Therefore, a large data quantity would be loaded repeatedly to analyse different applications, leading to a decrease of the analysis process's performance. Large number of applications that should be analysed and the amount of data, means the analysis cannot be concluded in a reasonable time. Therefore it is necessary to place the reformatted GPFS I/O logs into a suitable database. This provides an efficient and optimized data access. Further database improvements, such as indexing, are implemented to allow for better performance. Once again checks are performed to ensure the correctness of the resulting database contents.

4.2.1 GPFS I/O Log Database

Due to the large size of the gathered GPFS I/O logs, placing all logs into a single table would prolong later the query process. Therefore the data has to be divided into different tables. The GPFS I/O logs are placed in tables corresponding to the I/O nodes on which they were recorded, leading to a total of 600 tables for the JUGENE system. Since the application runtime information include the I/O nodes used, the database query for the relevant GPFS I/O logs of the application is optimized. Rather than searching through the database for the entries corresponding to both the time and the I/O nodes for the application, the analysis would already provide the I/O node information by selecting the correct tables. Table entries are indexed using a time stamp corresponding to the time at which the GPFS I/O counter was logged and the filesystem it was logged for.

Table 4.2 shows some information relating to the size of the GPFS I/O log database. Here I/O requests can be referred to as read/write commands[4].

Mismatch of open to close could be explained by lost requests. Close commands lie more often at the end of a job. From Sec. 4.2 some regions are not fully logged. These mostly exist at the end of the GPFS I/O logs. Therefore it can be expected that some close commands would go missing.

Figure 4.2 represents a first look into the GPFS I/O logs for each I/O node. Since jobs on JUGENE cannot have a smaller allocation than a full midplane[5] (excluding the

[4]In this study I/O requests are synonyms to I/O commands and both are used interchangeably.
[5]Half a rack is a midplane, which contains 512 compute nodes with 4 I/O nodes.

Database size	222428.09375 MiB
Total number of GPFS I/O logs	754,365,752, with an average of 1,257,276 entries per I/O Node
Total data read	28.4169 PiB
Total data written	16.6596 PiB
Total read commands	15.3×10^9
Total write commands	18.07×10^9
Total file open commands	11.14×10^9
Total file close commands	10.91×10^9

TABLE 4.2: Information on the GPFS I/O log database.

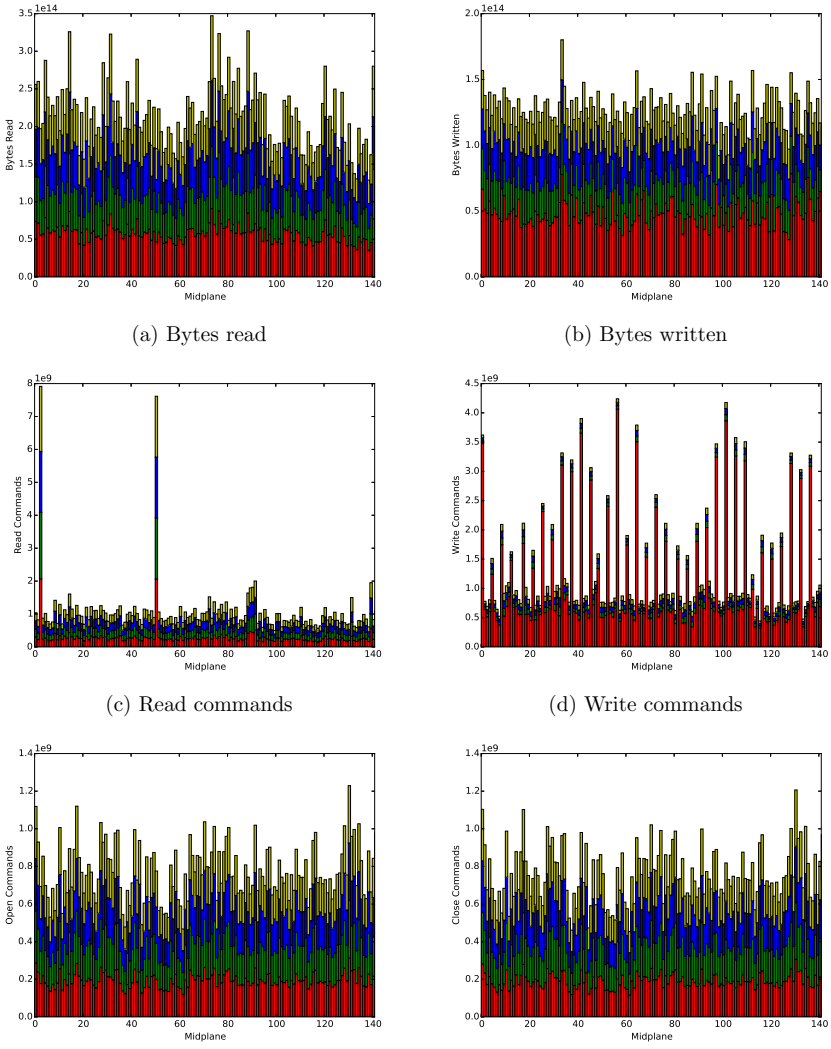
last rack), the four I/O nodes on each midplane should behave similarly. A perfect parallelism expects the amount of read or written data to be equal across the four I/O nodes. Such behaviour is observed in Fig. 4.2. One exception to this is the number of write commands shown in Fig. 4.2-(d), where the first I/O nodes seem to issue a lot more write commands than the other three. This could indicate that GPFS has some method for write request aggregation. Furthermore, some midplanes appear to be performing more write requests and the behaviour appears periodic across JUGENE. This could indicate a possible step wise aggregation between different midplanes. The existence of such aggregation has not been confirmed by studying GPFS.

4.3 Job Database

Number of jobs	1108814
Average duration of jobs	3039 sec ($\simeq$ 50min)
Average number of compute nodes	694 compute nodes
Average number of I/O nodes	6 I/O nodes

TABLE 4.3: Information on the job database.

The term job refers to a single run (i.e. the execution) of an application. Since an application can run several times using same or different parameters, it can result in many jobs. Meanwhile a job can only be related to one application. As part of the administration of JUGENE a job database is kept. This contains information on the jobs that ran on the system, including start and end time. Table 4.3 shows some information on the job database. The job information in the database allows matching GPFS I/O logs to the corresponding jobs that initiated the I/O requests. Therefore the job database can be limited to jobs that ran during the period of logging the GPFS I/O counters.



(a) Bytes read

(b) Bytes written

(c) Read commands

(d) Write commands

(e) Open commands

(f) Close Commands

FIGURE 4.2: Sum of GPFS I/O logs for midplanes divided among the four I/O nodes.

Jobs are indexed using a `JOBID`, which will be used throughout the analysis to refer to individual jobs. This provides user anonymity.

4.4 Verifying Analysis Process

The major challenge of a large scale data analysis are the occasional missing data or meta information needed to conduct an in depth analysis. Although the collected GPFS I/O logs represents a powerful insight into I/O on a full scale machine, the resolution of the GPFS I/O logs is limited. The GPFS I/O counters are logged every 120sec, thereby loosing accuracy of the temporal component. Additionally the spatiality is completely lost as I/O requests cannot be linked to addresses or files. Still by observing the limitations set on the I/O measurement, the analysis of the GPFS I/O logs can yield some conclusions on large scale scientific application I/O. The GPFS I/O logs can be viewed as information on the flow of data between I/O nodes and the storage system. Thereby many of the details of the inner workings of the storage system and GPFS cannot be observed. By subsequently limiting the GPFS I/O log analysis to focus on the application behaviour this might be an advantage. The I/O criteria evaluated can focus on the measurable end I/O behaviour of the applications, without being complicated by the storage system details.

The collected GPFS I/O logs can be considered a series of values representing the change of any GPFS I/O counter as $v(t_i)$ during the time interval $[t_{i-1}, t_i]$, where $t_{i-1} < t_i$. Here $v(t_i)$ can be the number of read/write requests, bytes read/written, number of open/close commands or any other GPFS I/O counter. As described the GPFS I/O counters are logged at 120sec intervals, thus $t_{i-1} - t_i \simeq 120sec$[6]. It can be assumed that their exists a known upper limit for a GPFS I/O counter change $v_{\max}$, where $0 \leq v(t_i) < v_{\max}$. The upper limit can be either determined theoretically (e.g. based on nominal hardware peak performance numbers) or empirically.

To be able to investigate the GPFS I/O logs it is necessary to link the logged requests to the jobs that initiated them. For that $t^{(\mathrm{k})}_{\mathrm{start}}$ and $t^{(\mathrm{k})}_{\mathrm{end}}$ are needed, which are the k-th job start and end time respectively. Jobs are expected to run consecutively on an I/O node, thus $.. < t^{(\mathrm{k-1})}_{\mathrm{start}} < t^{(\mathrm{k-1})}_{\mathrm{end}} < t^{(\mathrm{k})}_{\mathrm{start}} < t^{(\mathrm{k})}_{\mathrm{end}} < t^{(\mathrm{k+1})}_{\mathrm{start}} < t^{(\mathrm{k+1})}_{\mathrm{end}} < .. $. The target of matching the GPFS I/O logs to job (k) can be reduced to estimating the aggregate change of the counter values in the interval $[t^{(\mathrm{k})}_{\mathrm{start}}, t^{(\mathrm{k})}_{\mathrm{end}}]$. This allows for simplifying the description of the following matching methods. The matched GPFS I/O logs can be used in relation to

[6] As previously mentioned, the GPFS I/O logs are occasionally more than 120sec apart. The methods described here are designed to deal with variations in Δt

their timestamps to create the temporal I/O behaviour as described later. The methods provided here are described for a single I/O node, but can be expanded by repeating the process for each job sequence on each I/O node associated with job (k).

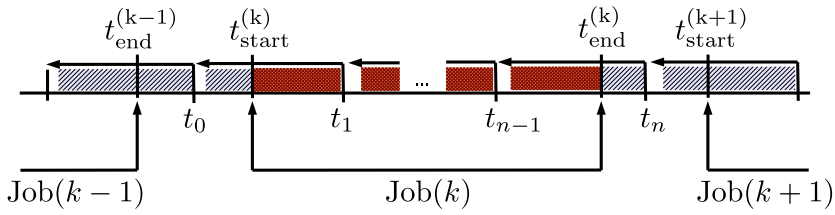


FIGURE 4.3: Special case for matching job runtime with GPFS I/O logs.

Figure 4.3 shows a special case where a value $v(t_0)$ exists between job $(k-1)$ and job (k), where $t_{\text{end}}^{(\text{k}-1)} < t_0 < t_{\text{start}}^{(\text{k})}$ and a value $v(t_n)$ exists between job (k) and job $(k+1)$, where $t_{\text{end}}^{(\text{k})} < t_n < t_{\text{start}}^{(\text{k}+1)}$. In such a case the estimation of the aggregate change of counter values is straight forward and can be defined as:

$$\tilde{V}^{(k)} = \sum_{i=1}^{n} v(t_i) \tag{4.1}$$

Where:

$$t_{\text{end}}^{(\text{k}-1)} < t_0 < t_{\text{start}}^{(\text{k})} < t_1 < .. < t_{n-1} < t_{\text{end}}^{(\text{k})} < t_n < t_{\text{start}}^{(\text{k}+1)}$$

For this special case it can be expected that $\tilde{V}^{(k)}$ is identical to the true value $V^{(k)}$. Here $V^{(k)}$ can be considered the value obtained if the GPFS I/O counters were logged at time $t_{\text{start}}^{(\text{k})}$ and $t_{\text{end}}^{(\text{k})}$[7].

The main problem with matching GPFS I/O logs to the job runtime, is observed when a value $v(t_i)$ does not exist between two jobs. Considering no measured value between job $(k-1)$ and job (k), meaning $t_0 < t_{\text{end}}^{(\text{k}-1)} < t_{\text{start}}^{(\text{k})} < t_1$. Therefore, the value $v(t_1)$ cannot be fully attributed to job $(k-1)$ or job (k) and can therefore be considered in conflict. The most plausible assumption is that both job $(k-1)$ and job (k) have contributed to the value $v(t_1)$ with an unknown ratio. Fig. 4.4 shows the worst case

[7]This is not strictly guaranteed as I/O operations might not be completed by the job's end.

where job (k) exhibits a conflicted value at both the start and end of the job's runtime. Here $t_0 < t_{\text{end}}^{(\text{k}-1)} < t_{\text{start}}^{(\text{k})} < t_1$ as well as $t_{n-1} < t_{\text{end}}^{(\text{k})} < t_{\text{start}}^{(\text{k}+1)} < t_n$. Conflicts such as these need to be resolved in order to allow for matching GPFS I/O logs to job runtime. Ultimately the method used will not deliver a fully accurate estimation or $\tilde{V}^{(k)} = V^{(k)}$. Therefore different strategies should be evaluated.

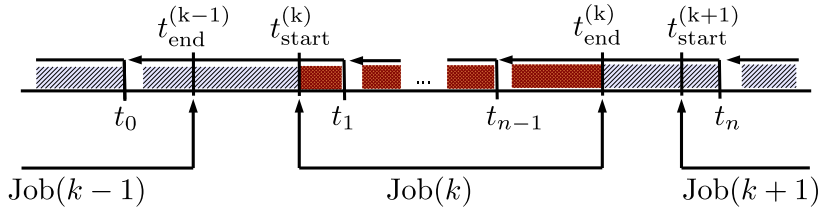


FIGURE 4.4: Worst case, with missing values $v(t_i)$ between jobs, for matching job runtime with GPFS I/O logs.

The first strategy is to discard all values at which such conflicts occur. For the worst case shown in Fig. 4.4 this means discarding values $v(t_1)$ and $v(t_n)$. This results in an aggregate change of GPFS I/O counters for job (k) that can be defined as:

$$\tilde{V}^{(k)} = \sum_{i=2}^{n-1} v(t_i) \tag{4.2}$$

Where:

$$t_0 < t_{\text{end}}^{(\text{k}-1)} < t_{\text{start}}^{(\text{k})} < t_1 < .. < t_{n-1} < t_{\text{end}}^{(\text{k})} < t_{\text{start}}^{(\text{k}+1)} < t_n$$

It is also possible to evaluate an upper limit to the estimation error using the discard strategy and $v_{\max}$, that can be defined as:

$$|V^{(k)} - \tilde{V}^{(k)}| \leq 2v_{\max} \tag{4.3}$$

The second strategy is to double-count conflicted values i.e. to attribute a value $v(t_i)$ to two jobs. For the worst case example given in Fig. 4.4, $v(t_1)$ will be aggregated to both job $(k-1)$ and job (k). Meanwhile, $v(t_n)$ will be aggregated to both job (k) and

job $(k+1)$. Using this strategy the aggregate change of GPFS I/O counters for job (k) can be defined as:

$$\tilde{V}^{(k)} = \sum_{i=1}^{n} v(t_i) \tag{4.4}$$

Where:

$$t_0 < t_{\text{end}}^{(\text{k}-1)} < t_{\text{start}}^{(\text{k})} < t_1 < .. < t_{n-1} < t_{\text{end}}^{(\text{k})} < t_{\text{start}}^{(\text{k}+1)} < t_n$$

The upper limit for estimation error for double-count should be similar to the one given in Eq. 4.3.

It is worth noting that the discard strategy can be considered the lower limit, while the double-count strategy can be considered the upper limit of the aggregate value $V^{(k)}$. It is therefore possible to create both an upper and a lower limit for each job. This however complicates the analysis and increases the time required to perform an analysis on all jobs. In general it is better to have a single method of matching I/O logs to job runtime. The method should reduce as much as possible the inaccuracy, while avoiding implementing too many assumptions into the matching method.

A third strategy is to attempt to resolve conflicts using a weighted-count. The strategy assumes that that no I/O occurs between two jobs in the interval $[t_{\text{end}}^{(\text{k})}, t_{\text{start}}^{(\text{k}+1)}]$. It also assumes all jobs perform I/O operations at approximately the same rate and therefore the number of I/O operations in the time intervals $[t_i, t_{\text{end}}^{(\text{k})}]$ and $[t_{\text{start}}^{(\text{k}+1)}, t_{i+1}]$. Using these assumptions it is possible to aggregate the change of GPFS I/O counters for job (k) as seen in the worst case scenario given in Fig. 4.4 using weighted measurements for job (k) as $\tilde{v}^{(k)}$:

$$\tilde{v}^{(k)}(t_1) = w_{\text{start}}^{(k)} v(t_1) \tag{4.5}$$

$$\tilde{v}^{(k)}(t_n) = w_{\text{end}}^{(k)} v(t_n) \tag{4.6}$$

Where the weighted factors can be defined as:

$$w_{\text{start}}^{(k)} = \frac{t_1 - t_{\text{start}}^{(\text{k})}}{t_{\text{end}}^{(\text{k}-1)} - t_0 + t_1 - t_{\text{start}}^{(\text{k})}} \tag{4.7}$$

$$w_{\text{end}}^{(k)} = \frac{t_{\text{end}}^{(\text{k})} - t_{n-1}}{t_{\text{end}}^{(\text{k})} - t_{n-1} + t_n - t_{\text{start}}^{(\text{k}+1)}} \tag{4.8}$$

It is worth noting that, $w_{\text{start}}^{(k)}$ can be considered the ratio of time job (k) spent doing I/O that contributed to the value $v(t_1)$ as seen in Fig. 4.5. As a result, $0 < w_{\text{start}}, w_{\text{end}} < 1$ and $w_{\text{end}}^{(\text{k}-1)} + w_{\text{start}}^{(k)} = 1$. The same reasoning would make $w_{\text{end}}^{(k)}$ the ratio of job (k) spent doing I/O that contributed to the value $v(t_n)$ and $w_{\text{end}}^{(k)} + w_{\text{start}}^{(\text{k}+1)} = 1$

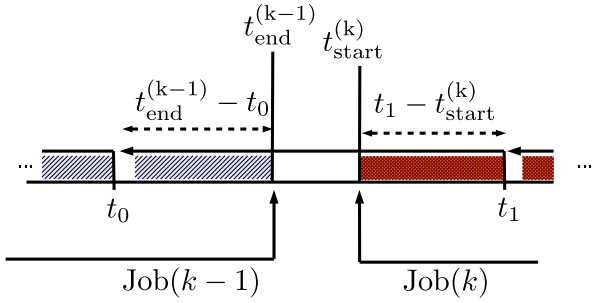


FIGURE 4.5: The weighted fraction $w_{\text{start}}^{(k)}$ can be considered the ratio of time job (k) spent doing I/O that contributed to the value $v(t_1)$.

Considering the worst case scenario given in Fig. 4.4 it is now possible to define the aggregate change of GPFS I/O counters for job (k) as:

$$\tilde{V}^{(k)} = \tilde{v}^{(k)}(t_1) + \sum_{i=2}^{n-1} v(t_i) + \tilde{v}^{(k)}(t_n) \tag{4.9}$$

Where:

$$t_0 < t_{\text{end}}^{(\text{k}-1)} < t_{\text{start}}^{(\text{k})} < t_1 < .. < t_{n-1} < t_{\text{end}}^{(\text{k})} < t_{\text{start}}^{(\text{k}+1)} < t_n$$

For the conflict resolution method or weighted-count the limit of error can be defined as:

$$|V^{(k)} - \tilde{V}^{(k)}| \leq (w^{(k)}_{\text{start}} + w^{(k)}_{\text{end}})v_{\max} < 2v_{\max} \tag{4.10}$$

Note that all so far described strategies will directly aggregate any non-conflicted values. For example, applying the conflict resolution strategy on the special case given in Fig. 4.3, the aggregation defaults to Eq. 4.1. As a result, all three strategies discard, double-count and conflict resolution have the need to identifying conflicts. To distinguish conflicted values requires knowledge of both previous and subsequent jobs and their runtime information. This could slightly complicate the implementation of the matching scripts. To simplify the matching for implementation, it is possible to only consider values that were logged within the runtime of the job being analysed, irrelevant of conflicts.

The fourth and final strategy is a simple matching, which only counts the GPFS I/O logs within the runtime of the job, i.e. in the interval $[t^{(k)}_{\text{start}}, t^{(k)}_{\text{end}}]$. Such a strategy would result in an aggregate change of GPFS I/O counters for job (k) that can be defined as:

$$\tilde{V}^{(k)} \quad = \quad \sum_{i=1}^{n-1} v(t_i) \tag{4.11}$$

Here Eq. 4.11 remains the same for the special and worst case described in Fig. 4.3 and Fig. 4.4. The upper limit of the estimation error for the simple method can be defined as:

$$|V^{(k)} - \tilde{V}^{(k)}| \leq v_{\max} \tag{4.12}$$

Both the discard and the double-count strategies are simple and easy to implement. However, both have the disadvantage of the values being discarded or duplicated among jobs. As a result, aggregation of values at the global level and a per job level will not match when using these methods. Such aggregation comparison can allow for checking correctness of implemented analysis tools. Therefore both strategies complicate the debugging and checking for faults in the matching of GPFS I/O logs to jobs. Although the simple strategy suffers from the same disadvantage, it is easier to implement. In

comparison, the conflict resolution strategy does not suffer from this issue. And despite being more complicated, the implementation of conflict resolution is still manageable.

The upper limit of error is lower for both the simple and conflict resolution strategies compared to both the discard and double-count strategies, which have equal upper limits of error. However, determining which approach is better when comparing the simple and conflict resolution is more complicated. This is due to the ratio's used in the conflict resolution being part of the upper limit error estimation. It is therefore worth testing and comparing the two strategies of matching I/O logs. It is also necessary to verify the analysis process. To achieve a good comparison and to verify the matching process and the accuracy of the GPFS I/O logs, an I/O benchmark is used.

It is worth noting that the error rising from mismatches such as conflicts is reduced as job runtime increases. It is therefore expected that once longer running jobs are selected the conflicts will only marginally effect the overall analysis.

4.4.1 Verification Of GPFS I/O Counters Using I/O Benchmark

To verify the I/O data and the analysis process, knowledge of the I/O done by the running job is needed. This can be achieved by running an I/O benchmark while the GPFS I/O counters are being logged. Most available I/O benchmarks are complex, making it difficult to determine individual I/O requests timing. Therefore a simplified micro-benchmark is created. To parse the possible application I/O behaviours the micro-benchmark reads in a set of parameters that allow altering the test conditions. The parameters include I/O request size, number of requests and file being shared or task-local. An additional option allows for creating random I/O request sizes. The micro-benchmark can even switch the I/O interface by using either POSIX-I/O or MPI-IO.

The test scheme is kept as simple as possible, with the individual I/O requests being recorded with timestamps after conclusion. The time-line is in order an open, write, close, open, read and close. Meaning, the micro-benchmark starts with opening a set of files and writes the given number of I/O requests to those files and closes them. The files are then reopened and the data read-back with the same set of given I/O requests.

As a check the read buffers are compared with the write buffers for correctness[8]. The scheme allows for creating a reasonable amount of I/O including opening and closing of files, that all will be logged by the GPFS I/O counters.

The micro-benchmark ran on 32 compute nodes and 1 I/O node with different requests sizes, using either MPI-IO or POSIX-I/O and opening either a single shared file or task-local files. For some request sizes the test is repeated using 64 compute nodes and 2 I/O nodes. This allows for covering a wide range of parameters. It is worth noting that all tests using the micro-benchmark are conducted using 4 process per compute node. Therefore a 32 compute node test contained 128 processes, while a 64 compute node test contained 256 processes.

Table 4.4 shows the micro-benchmark results for using POSIX-I/O with task-local files. The values are given in error percentage from the I/O registered by the micro-benchmark for both the simple (S) E_{simp} and conflict (C) E_{conf} resolution matching. Error percentage is calculated using the following equations:

$$E_{\text{simp}} = \frac{\tilde{V}^{(k)}_{\text{simp}} - V^{(k)}}{V^{(k)}} * 100 \tag{4.13}$$

$$E_{\text{conf}} = \frac{\tilde{V}^{(k)}_{\text{conf}} - V^{(k)}}{V^{(k)}} * 100 \tag{4.14}$$

where:

$\tilde{V}^{(k)}_{\text{simp}}$ is $\tilde{V}^{(k)}$ calculated using simple matching.

$\tilde{V}^{(k)}_{\text{conf}}$ is $\tilde{V}^{(k)}$ calculated using conflict resolution matching.

$V^{(k)}$ is the true value as seen by the I/O benchmark.

The values $\tilde{V}^{(k)}_{\text{simp}}$, $\tilde{V}^{(k)}_{\text{conf}}$ and $V^{(k)}$, and therefore E_{simp} and E_{conf}, could be Bytes Read (BR), Read Commands (RdC), Bytes Written (BW), Write Commands (WC), Open Commands (OC) or Close Commands (CC).

The results given in table 4.4 seem to suggest that, in most cases the conflict resolution matching arrives at a lower or equal error compared to the simple method. However, for

[8]The is an added check to acknowledge that the data has traversed the I/O stack and should have been recorded by the GPFS I/O counters. Since the interest is not in computation nor in performance measurement, the time spent in comparing the buffers can be spared.

32 Nodes (1 I/O node)												
Request Size[Bytes]	BR[%]		RdC[%]		BW[%]		WC[%]		OC[%]		CC[%]	
	C	S	C	S	C	S	C	S	C	S	C	S
4Ki	0	-4	1	0	0	-5	0	0	50	-25	0	-65
16Ki	0	-11	0	0	0	-12	0	0	50	-24	0	-74
64Ki	0	-8	0	0	0	-8	0	0	50	-24	0	-74
Random max 64Ki	0	-7	0	0	0	-7	0	0	50	-24	0	-69
256Ki	0	-8	0	0	0	-8	0	0	50	-24	0	-74
Random max 512Ki	0	-6	0	0	0	-6	0	0	50	-24	0	-65
1024Ki	0	-3	0	0	0	-3	0	0	50	-24	0	-61
64 Nodes (2 I/O nodes)												
4Ki	0	-17	1	0	0	-17	0	0	50	-24	0	-74
Random max 512Ki	0	-13	0	0	0	-13	0	0	50	-24	0	-74
1024Ki	0	0	0	0	0	0	0	0	50	-24	0	-50

TABLE 4.4: Micro-benchmark using POSIX I/O and task-local files. GPFS I/O logs matching error is shown for simple (S) and conflict (C) resolution matching methods, given for Bytes Read (BR), Read Commands (RdC), Bytes Written (BW), Write Commands (WC), Open Commands (OC) and Close Commands (CC).

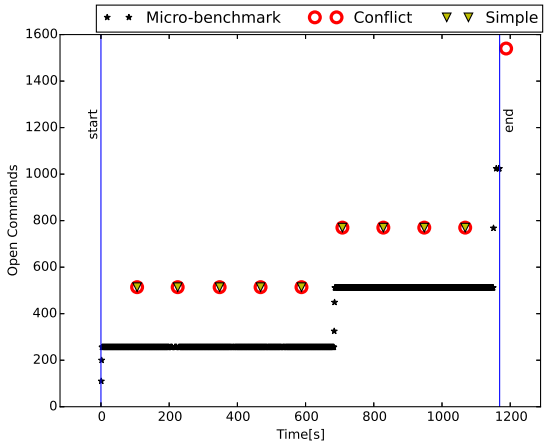


FIGURE 4.6: Open Command (OC), for matching POSIX-I/O task-local files 4KiB 64 node test to GPFS I/O logs.

Open Commands (OC) the simple method seem to arrive at a better estimate. Fig. 4.6 shows the logs for the Open Commands (OC) accumulating over time, of the POSIX-I/O task-local files 4KiB test with 64 nodes. The figure shows that the GPFS I/O counters logged many more open commands from the very beginning of the job runtime. This appears to repeat for other tests as well, making it difficult to estimate the open commands initiated directly by the job. A possible explanation for the excess of open

commands are system operations and the opening of binary files to be executed by the job.

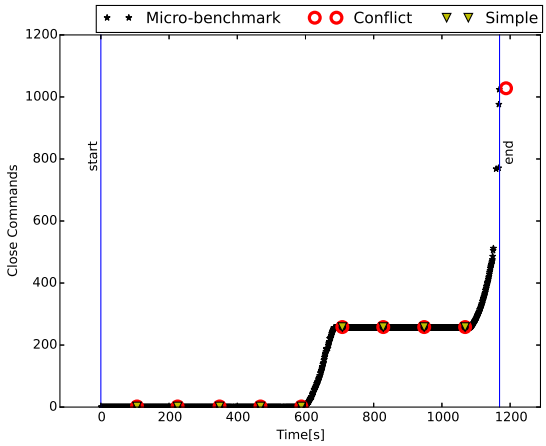
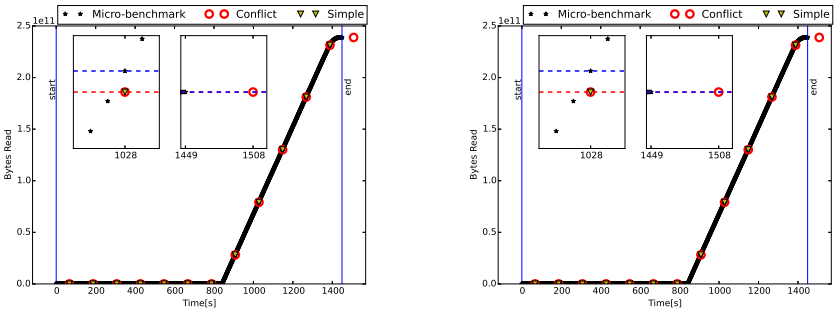


FIGURE 4.7: Close Command (CC), for matching POSIX-I/O task-local files 4KiB 64 node test to GPFS I/O logs.

In comparison Close Commands (CC) shown in Fig. 4.7 seem to be easier to estimate using the GPFS I/O logs. It has been previously mentioned that there is an unexplained mismatch between the total number of open commands and close commands in Sec. 4.2.1.

An analysis based on the GPFS I/O logs would also be interested in the temporal distribution of the I/O behaviour. To verify the accuracy of the intermediate values of I/O, Fig. 4.8 zooms on individual values of the logs. As seen for both Bytes Read (BR) and Read Commands (RdC), intermediate values may vary slightly from actual job I/O. The variation appears to show a small delay in adding the I/O commands or bytes to the GPFS I/O counters. However, the delay appears fairly small and seems to affect all values equally, as can be inferred when comparing Fig. 4.8-(a) and Fig. 4.8-(b). Despite the slight time shift, the final value of both benchmark and GPFS I/O logs analysed using conflict resolution matching appear to be almost equal as given by Tab. 4.4. This is also observable in the second zoom-in in Fig. 4.8-(a) and Fig. 4.8-(b). This outcome can be seen for all measured values with the exception of Open Commands (OC) and Closed Commands (CC). As a result the temporal distribution of I/O can be analysed using GPFS I/O logs, however the slight possible inaccuracy has to be observed.



(a) Bytes Read (BR) (b) Read Commands (RdC)

FIGURE 4.8: Bytes Read (BR) and Read Commands (RdC), for matching POSIX-I/O task-local files 1024KiB 32 node test to GPFS I/O logs.

Tab. 4.5 shows another set of micro-benchmark runs using POSIX-I/O and shared files. All processes here used a single shared file to write and read from. The results appear to suggest the same. In most cases the conflict resolution matching appears to contain less error in its estimation.

32 Nodes (1 I/O node)												
Request Size[Bytes]	BR[%]		RdC[%]		BW[%]		WC[%]		OC[%]		CC[%]	
	C	S	C	S	C	S	C	S	C	S	C	S
4Ki	0	-6	0	-6	1	0	0	0	25	-49	0	-74
16Ki	0	-70	0	-70	0	0	0	0	25	-49	0	-74
64Ki	0	-25	0	-25	0	0	0	0	25	-49	0	-74
Random max 64Ki	0	-2	0	-2	0	0	0	0	27	-48	1	-57
256Ki	0	-7	0	-7	0	0	0	0	25	-49	0	-74
Random max 512Ki	0	0	0	0	0	0	0	0	25	-49	0	-52
1024Ki	0	-11	0	-11	0	0	0	0	25	-49	0	-74
64 Nodes (2 I/O nodes)												
4Ki	-5	-10	-5	-10	203	0	0	0	-6	-49	-42	-74
Random max 512Ki	0	0	0	0	0	0	0	0	25	-49	0	-49
1024Ki	0	-9	10	9	-1	0	-1	0	57	25	42	0

TABLE 4.5: Micro-benchmark using POSIX I/O and shared files. GPFS I/O logs matching error is shown for simple (S) and conflict (C) resolution matching methods, given for Bytes Read (BR), Read Commands (RdC), Bytes Written (BW), Write Commands (WC), Open Commands (OC) and Close Commands (CC).

In the case of Bytes Written (BW) using 4KiB and 64 compute nodes an estimation error of 203% is given for the conflict resolution matching. Fig. 4.9 shows the logs accumulative over time for the Bytes Written (BW) of the POSIX-I/O shared file 4KiB test with 64 nodes. From the figure it appears that the conflict resolution matching has

matched a portion of the bytes written in job $(k+1)$ to job (k). An error that can occur when two jobs are closely scheduled. Additionally the jobs have to create many requests at the job boundary to effect the other job matching estimation. From the test results this appears to be an uncommon occurrence.

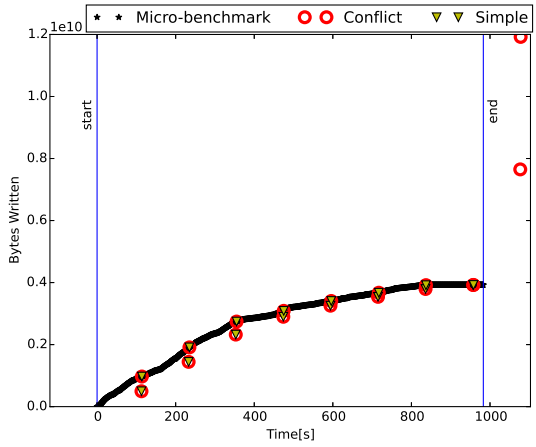


FIGURE 4.9: Bytes Written (BW), for matching POSIX-I/O shared file 4KiB 64 node test to GPFS I/O logs.

To complete the scan of the dimensions given by the micro-benchmark parameters, Tab. 4.6 and Tab. 4.7 show the results for using MPI-I/O with task-local files and a shared file respectively. Tab. 4.6 appears to suggest the same as previous tests. The conflict resolution results in a higher accuracy in most cases compared to the simple matching.

Compared to previous results, 4.7 shows large estimation errors for both Read Commands (RdC) and Write Commands (WC) when using either conflict resolution or simple matching. This observation can be a result of the MPI-I/O for shared file performing collective I/O. Thereby reducing the number of I/O requests seen by the filesystem. This reiterates the importance of noticing the effect of the I/O stack layer on which I/O is measured.

With a few mentioned exceptions, the conflict resolution method shows that matching of GPFS I/O logs to job runtime can be achieved with relatively low error margins. As a result, conflict resolution matching will be used for job I/O analysis in the following sections.

32 Nodes (1 I/O node)												
Request Size[Bytes]	BR[%]		RdC[%]		BW[%]		WC[%]		OC[%]		CC[%]	
	C	S	C	S	C	S	C	S	C	S	C	S
4Ki	0	-8	0	-8	1	0	0	0	75	0	25	-43
16Ki	0	-24	0	-24	0	0	0	0	75	0	25	-49
64Ki	0	-17	0	-17	0	0	0	0	75	0	25	-49
Random max 64Ki	0	-3	0	-3	0	0	0	0	75	0	25	-39
256Ki	0	0	0	0	0	0	0	0	75	27	25	-16
Random max 512Ki	0	0	0	0	0	0	0	0	75	0	25	-24
1024Ki	0	-71	0	-71	0	0	2	0	75	0	25	-49
64 Nodes (2 I/O nodes)												
4Ki	0	-9	0	-9	1	0	0	0	75	0	25	-49
Random max 512Ki	0	-12	0	-12	-1	0	0	0	16	0	6	-43
1024Ki	0	-10	0	-10	0	0	0	0	75	0	25	-49

TABLE 4.6: Micro-benchmark using MPI-IO and task-local files. GPFS I/O logs matching error is shown for simple (S) and conflict (C) resolution matching methods, given for Bytes Read (BR), Read Commands (RdC), Bytes Written (BW), Write Commands (WC), Open Commands (OC) and Close Commands (CC).

32 Nodes (1 I/O node)												
Request Size[Bytes]	BR[%]		RdC[%]		BW[%]		WC[%]		OC[%]		CC[%]	
	C	S	C	S	C	S	C	S	C	S	C	S
4Ki	0	-7	-99	-99	1	0	-99	-99	25	-49	0	-74
16Ki	0	-6	-98	-98	0	0	-98	-98	25	-49	0	-74
64Ki	0	-3	-93	-93	0	0	-93	-93	25	-49	0	-74
Random max 64Ki	0	-8	-93	-94	0	0	-93	-93	25	-49	0	-74
256Ki	0	-1	-93	-93	1	0	-93	-93	13	-49	-27	-74
Random max 512Ki	0	-1	-93	-93	0	0	-93	-93	25	-49	0	-74
1024Ki	0	1	-74	-74	0	0	-74	-74	38	100	28	75
64 Nodes (2 I/O nodes)												
4Ki	0	0	-99	-99	2	1	-99	-99	44	25	-5	-24
Random max 512Ki	0	-1	-93	-93	0	0	-93	-93	25	-49	0	-74
1024Ki	0	-11	-75	-78	0	0	-74	-73	6	-49	6	-49

TABLE 4.7: Micro-benchmark using MPI-IO and shared files. GPFS I/O logs matching error is shown for simple (S) and conflict (C) resolution matching methods, given for Bytes Read (BR), Read Commands (RdC), Bytes Written (BW), Write Commands (WC), Open Commands (OC) and Close Commands (CC).

4.5 Evaluating JUGENE Job I/O

As the Sec. 4.4 shows, it is possible to evaluate the I/O done by a job using the GPFS I/O logs. Such evaluation has the value of showing the I/O behaviour of scientific applications on large HPC systems under real conditions. Furthermore, using the I/O criteria discussed in Chp. 3, it is possible to form some general conclusions on the

overall I/O behaviour of scientific applications. In addition to that, the I/O criteria analysis allows for the selection of interesting jobs or applications that can be taken as representative for some I/O conditions and later further analysed.

Since JUGENE was decommissioned in 2012 which ended the GPFS I/O logging for this system, one may consider the GPFS I/O logs collected on JUGENE to be outdated. However, there are two main reasons why it is still worthwhile to analyse the GPFS I/O logs collected on JUGENE using the I/O criteria and thereby evaluate job I/O behaviour. The first reason is to show the applicability of evaluating job I/O using I/O criteria. The purpose therefore becomes demonstrating the methodology described in Chp. 3. The second reason is maturity of job I/O. While analysing the latest HPC system's I/O is tempting, application developers require time to port their application onto the newly available hardware. As a result, many jobs running at the early stages of an HPC system can be considered porting experiments and might not achieve the system's full potential. Furthermore, large collections of I/O logs that allow for a large scale job I/O analysis requires time. For example, to have the possibility of analysing the number of jobs discussed in Sec. 4.3 necessitated collecting I/O logs for a period of 19 months.

There are further reasons why an I/O analysis using the GPFS I/O logs collected on JUGENE is worthwhile. It is possible to argue that many applications develop their I/O behaviour slowly and would therefore change relatively little in their I/O behaviour overtime. Thus making the measured I/O behaviour using JUGENE's GPFS I/O logs relevant to modern I/O subsystems. Additionally, a mass I/O log analysis might reverse the process with the intention of analysing the impact I/O subsystem architectural changes have on I/O behaviour. In such a case a supercomputing center might be interested in performing such long term I/O analysis over two or more generations of HPC systems. Using the results some conclusions can be drawn on the next I/O subsystem architecture that would best serve the I/O behaviour of scientific applications.

4.5.1 Filtering The Job List

Tab. 4.3 in Sec. 4.3 shows the complete size of the jobs that ran on JUGENE during the time the GPFS I/O counters are logged. Despite the temptation to analyse the entire set of jobs that ran during that period, some filtering is necessary. There are

two main reasons for reducing the number of jobs to be analysed. First, the analysis process is time consuming and/or requires large processing power, storage, Random Access Memory (RAM) and systems with high I/O bandwidth. Such effort can be acceptable and depends on the available resources. However, the second prime reason for not analysing the full set of jobs, is the existence of short or fail-run jobs that might taint the overall analysis and as a result the conclusions drawn from it.

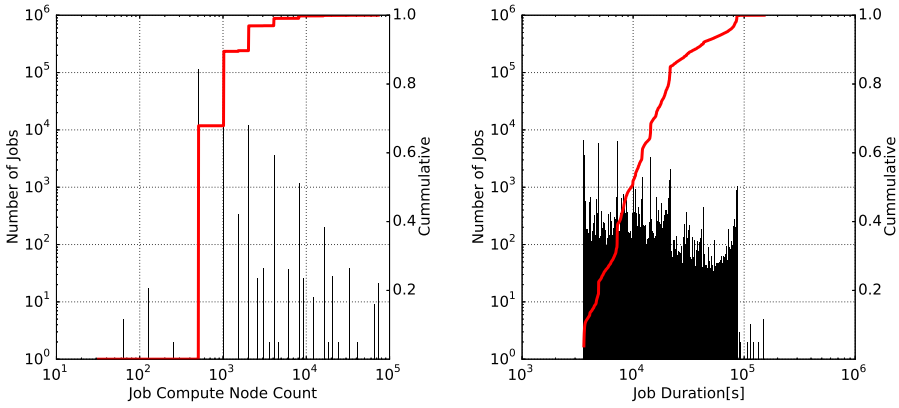
Avoiding overly complicating or extending the I/O analysis requires a mere reduction of the number of analysed jobs, which can be performed in random or using simple criteria or a combination of both. However, filtering the job list to remove irrelevant or erroneous jobs, requires the identification of job specification that correctly select the jobs to be filtered out. The target is to attempt recognizing production jobs, which are defined as jobs that are a full run of a scientific application and that possibly ended without errors. Correctly identifying production jobs requires the involvement of application developers that ran the job. As this is not feasible, production jobs have to be inferred from the available information in the job database. Two key aspects that can be used is the scale or number of compute nodes used by the job and the duration of the job's runtime. While at first choosing larger scale jobs appears reasonable, it risks leaving scientific applications that run on a smaller scale unanalysed. Additionally, large jobs are expected to preform more I/O and could therefore skew results.

Job duration can be used to reasonably identify possible production jobs. The assumption is that a job that ran for at least 1 hour has performed many or all of the intended duties. Therefore the resulting I/O measured by the GPFS I/O logs should contain the I/O behaviour exhibited by the application. Furthermore, a 1 hour job allows for at least 30 GPFS I/O logs to be available for the job. As a result, the error of matching GPFS I/O logs to job runtime, described in Sec. 4.4, is reduced. In addition to filtering using job duration, any jobs known to be I/O benchmarks or performed no read or no write are discarded.

Filtering jobs that ran for less than 1 hour reduces the job list to 166971 jobs from originally 1108814. In numbers the filtered list only represent 15% of the jobs that ran while logging GPFS I/O counters. However, in terms of duration the filtered list occupied over 80% of the compute time on JUGENE during logging. That is 80% of the logged I/O would be analysed when using the filtered list. Furthermore, when

considering both job duration and node count, the filtered list constitutes 67% of the duration over node count.

For the filtered jobs, Fig. 4.10-(a) shows the job distribution over compute node count and indicates that about 68% of jobs occupied 512 compute nodes. Another 21% of jobs occupied 1024 compute nodes. Out of the 166971 selected jobs, only 21 used the complete JUGENE system. On the other hand, Fig. 4.10-(b) shows the job distribution over runtime duration. The cut off of 1hour or 3600s is clearly visible. When summed up, the total duration of all jobs is around 2.7×10^9s, 55% of that sum can be attributed to only 20% of filtered jobs. Meanwhile, the remaining 80% of filtered jobs have a duration of 5.82hours or less.



(a) Job compute node count (b) Job duration

FIGURE 4.10: Histogram and cumulative distribution of job compute node count and job duration

4.5.2 Revisiting I/O Criteria

The I/O criteria discussed in Chp. 3, introduced the limitations set by the I/O measuring method on the evaluation process. Specifically the total I/O system load while measuring I/O and the layer of the I/O stack at which I/O is measured greatly effect the resulting analysis. Since the GPFS I/O counters are logged on the filesystem and in an I/O forwarding layer, the resulting analysis sets some limitations on the evaluation of the I/O criteria.

To allow for analysing the GPFS I/O logs using the I/O criteria, the basic quantities discussed in Sec. 3.2 need to be defined in terms of GPFS I/O counters given in Tab. 4.1. The relationship is drawn in Tab. 4.8 and shows why the I/O criteria are expressed in a specific format.

Keyword	Description	Expression
`_t_`	Indicates the current time of day in seconds (absolute seconds since Epoch (1970)).	t
`_br_`	Total number of bytes read, from both disk and cache.	$\sum_s sD_\mathrm{r}(s,t)$
`_bw_`	Total number of bytes written, to both disk and cache.	$\sum_s sD_\mathrm{w}(s,t)$
`_oc_`	Count of open() call requests serviced by GPFS.	$F_\mathrm{open}(t)$
`_cc_`	Number of close() call requests serviced by GPFS.	$F_\mathrm{close}(t)$
`_rdc_`	Number of application read requests serviced by GPFS.	$D_\mathrm{r}(t)$
`_wc_`	Number of application write requests serviced by GPFS.	$D_\mathrm{w}(t)$
`_dir_`	Number of readdir() call requests serviced by GPFS.	$F_\mathrm{dir}(t)$
`_iu_`	Number of inode updates to disk.	$F_\mathrm{inode}(t)$

TABLE 4.8: GPFS I/O counters matched to basic quantities.

Additionally the application quantities such as t_start, t_{end} and number of I/O nodes used by the job are given by the job database. As discussed in Chp. 3 it is possible to use the number of I/O nodes instead of the number of processes P. This is required here as the GPFS I/O counters are logged on a per I/O node basis.

Some limitations on analysing the GPFS I/O logs using the I/O criteria comes from a missing set of basic quantities that cannot be determined from the GPFS I/O logs. Specifically $F_\mathrm{create}(s,t)$ the number of files created, $R(p,f,i)$ relating I/O requests to files and processes, and $R_\mathrm{off}(p,f,i)$ relating I/O requests to offset in file. As a result the number of created files and spatial patterns cannot be evaluated using the GPFS I/O logs.

The GPFS I/O counters are logged with an interval of 2min on the I/O nodes, leading to a reduction in the temporal resolution. This should be observed during GPFS I/O log analysis. Furthermore, evaluating overall I/O criteria for jobs requires summing the individual GPFS I/O logs across different I/O nodes. This is difficult as the logging is not synchronized. To settle these issues the assumption is made that the distribution of I/O is equal within the logged interval. This leads to some I/O criteria to be evaluated as an average over the 2min. To have synchronized values across I/O nodes that can be aggregated, the GPFS I/O logs are interpolated. This is achieved by distributing equally the change in the GPFS I/O log over the 2min with a $\Delta t = 1s$. However the

lost portion of the temporal distribution is not regained by the interpolation. Therefore, the effect the 2min interval has on the temporal distribution has to be observed when discussing temporal I/O behaviour.

Due to these limitations and observations not all I/O criteria can be evaluated using the GPFS I/O logs or the results have some attached constraints. This can be considered a price to pay for system level logging of performance data of such long period of time. The following sections describe the evaluation of the I/O criteria using the GPFS I/O logs, while enumerating any observation and/or limitation.

4.5.3 Category 1: Aggregate Performance Numbers

Classification 1.1 Total amount of data read/written

Fig. 4.11 shows the total amount of data read against data written for the 166971 jobs that were analysed. The Fig. 4.11-(a) uses hexagonal binning to reduce the scatter points. The colour of the hexagons indicate the number of jobs contained within it resulting in a heat map. Fig. 4.11-(b) and 4.11-(c) show histograms for distribution of jobs over the x and the y axis respectively. The histograms use binning to reduce the number of bars to a 1000 bins and show the number of jobs within each bin. Finally, to clearly convey the distribution of jobs, the cumulative distribution is placed onto the histograms. Fig. 4.11 is a good visualization of read and written data for the analysed jobs. The same data representation is used for other analysed I/O criteria when possible.

In total over 14PiB were read and over 11PiB were written. These are surprisingly small quantities, when considering the overall capacity of the storage system. This could indicated that the amount of transient data hitting the external storage is relatively small. Jobs averaged around 92GiB of read and 73GiB of write. However averages in this case are misleading, as 80% of the jobs did below 13GiB of read and below 16GiB of write. The average is simply increased by outliers, such as the largest read job which did 109TiB of read and the largest write job which did 22TiB of write. In fact 20% of the jobs are responsible for over 97% of both the read and write I/O in terms of bytes. This imbalance of I/O quantity distribution over jobs can be seen from the cumulative distributions in Fig. 4.11-(b) and 4.11-(c).

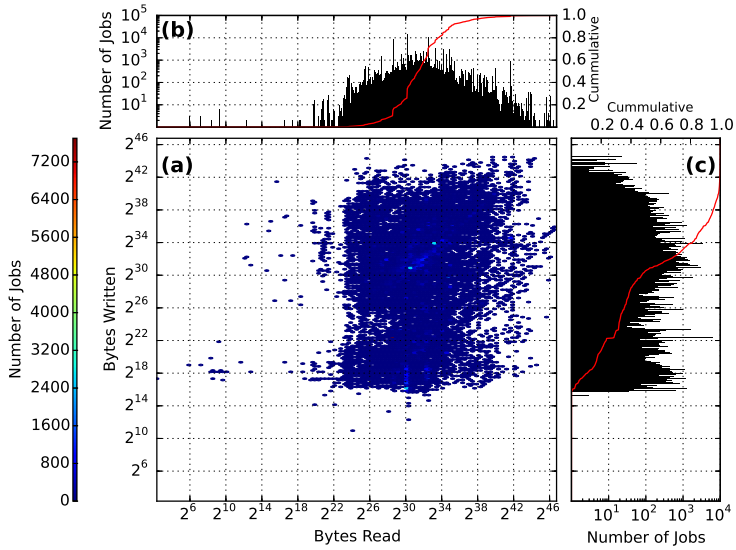


FIGURE 4.11: Bytes read and written for analysed jobs. (a) Scatter plot of bytes read and written with a heat map for job count, (b) Histogram of bytes read and (c) Histogram of bytes written.

One very useful tool for analysing large data sets is the use of clustering, which could be defined as the attempt of sorting jobs into different groups according to similar properties. The target is to find a small limited number of groups or clusters with little jobs being labelled as noise. These clusters can then be further analysed and used as basis for architecture or application changes. However, early attempts to provide job clustering using Density-based spatial clustering of applications with noise (DBSCAN) [49] on the basis of I/O quantity yields no reasonable clustering. Using various parameters for the DBSCAN results in either a few clusters with a lot of jobs being labelled as noise or in thousands of clusters. Both cases do not provide further insight into job I/O.

A similar conclusion, on the ability to cluster jobs according to read and written data, can also be inferred from Fig. 4.11. The heat map in Fig. 4.11-(a) indicates the absence of large job collections performing similar quantities of I/O. For data with large or obvious clusters, regions of different colours would be visible on the heat map. The jobs appear to be evenly distributed, with a few very small clusters, as indicated by the uniformity of colour. It is also not possible to create reasonable clusters using either read or write individually. Both Fig. 4.11-(b) and Fig. 4.11-(c) show that hardly any

group of jobs stands out by containing more than 10000 jobs.

Fig. 4.11 compares the read and written bytes of all jobs irrelevant of their size. It is therefore possible to assume that when jobs are normalized over their size some pattern or clusters might appear. Fig. 4.12 shows the average of bytes read and written over the I/O nodes of each job. As the interest is in job I/O the number of I/O nodes in a job are used to represent job size, rather than the number of compute nodes.

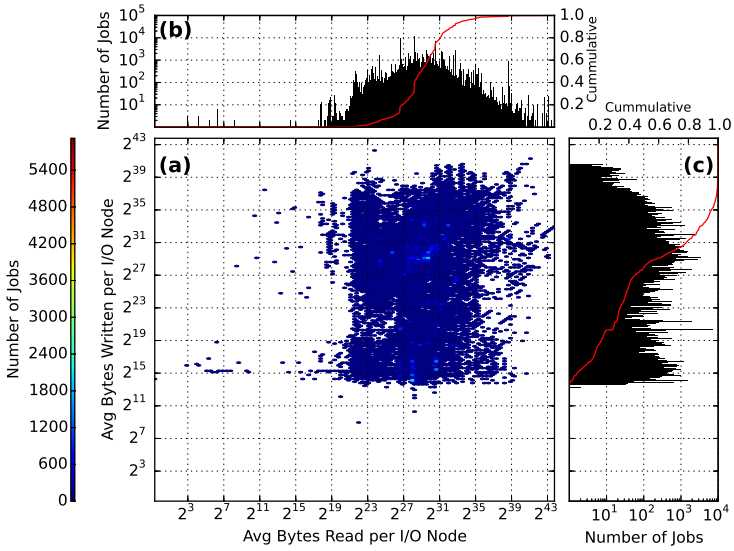


FIGURE 4.12: Bytes read and written average over I/O nodes for analysed jobs. (a) Scatter plot of bytes read and written averaged over the I/O nodes with a heat map for job count, (b) Histogram of bytes read averaged over the I/O nodes and (c) Histogram of bytes written averaged over the I/O nodes.

Despite Fig. 4.12 showing average bytes read and written per I/O node for each job, the distribution appears similar to that given in Fig. 4.11. This suggests that when a job grows, the amount of I/O produced grows accordingly. As a result the same conclusions can be drawn for job distribution over bytes read and written from both Fig. 4.11 and Fig. 4.12.

Classification 1.2 Total number of IOPs

As explained in the I/O criteria, the I/O stack layer at which I/O is measured strongly influences the measured results. Measuring IOPs is more influenced by the I/O subsystem than bytes read or written. Many I/O stack layers might perform collective I/O or buffering and result in different number of I/O requests arriving at the filesystem. Since this is the position at which I/O is measured, it is to be expected that the measured number of I/O requests does not exactly correspond to the number of I/O requests initiated by the jobs. An example for this effect has been shown by Tab. 4.7 in Sec. 4.4. Therefore, the term IOPs used here, refers to the I/O requests that are carried out by the filesystem and are registered by the GPFS I/O counters, after higher I/O stack layers have performed any I/O request changes. Since most I/O stack layers attempt reducing the number of I/O requests, it is possible to expect jobs to be actually performing a larger number of IOPs than measured by the GPFS I/O counters.

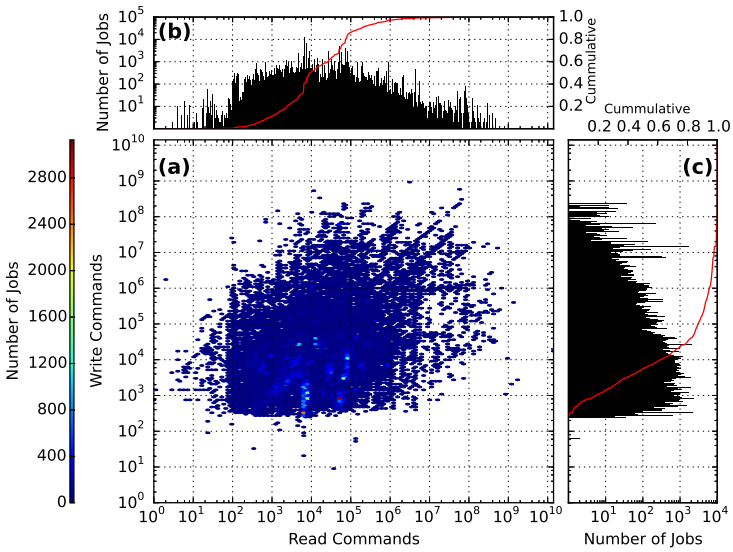


FIGURE 4.13: Read and write commands for analysed jobs. (a) Scatter plot of read and write commands with a heat map for job count, (b) Histogram of read commands and (c) Histogram of write commands.

Fig. 4.13 shows the number of read commands against the number of write commands for analysed jobs. In total over 102×10^9 read commands and over 145×10^9 write commands are performed. The average is over 615×10^3 for read and over 870×10^3 for writes.

Outliers of read and write have again skewed the averages, proven by the fact that 80% of jobs perform less than 77×10^3 read commands and 33.2×10^3 write commands. Where the maximum of write is over 924×10^6, the maximum of read is quite high at 14×10^9. Here 20% of jobs are responsible for over 97% of read commands and 99% of write commands. This I/O command distribution is shown by Fig. 4.13-(b) and Fig. 4.13-(c).

As the hexbin in Fig. 4.13-(a) indicates, the distribution of jobs over the read and write commands is semi-uniform. Only a few areas show some colour change, indicating small clusters of jobs. However, when compared to the total of 166971 jobs, these clusters seem too small to be significant. Clustering algorithms are therefore expected to produce a large number of small clusters, which would not help in further analysing job I/O behaviour. The same can be inferred from the histogram of both read and write, shown in Fig. 4.13-(b) and Fig. 4.13-(c) respectively. Here a single bin achieves a count of 1×10^4 jobs for read, while hardly any bin achieves 3×10^3 for write.

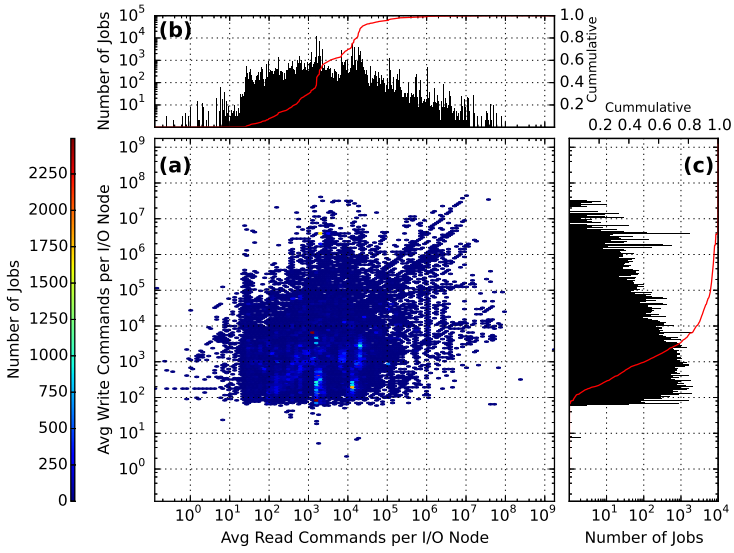


FIGURE 4.14: Read and write commands average over I/O nodes for analysed jobs. (a) Scatter plot of read and write commands average over I/O nodes with a heat map for job count, (b) Histogram of read commands average over I/O nodes and (c) Histogram of write commands average over I/O nodes.

Fig. 4.14 shows the average I/O commands over I/O nodes for analysed jobs. Similar to bytes read and written, the number of I/O commands change with the job size. As

a result the distribution given in Fig. 4.14 is very similar to that given in Fig. 4.13 and should lead to the same conclusions.

Classification 1.3 Read/Write bandwidth

Contrary to quantity of I/O and number of I/O commands, bandwidth is physically more limited by the used system. A job can hardly fill the large capacity storage, usually available for an HPC system. It can also use as many I/O commands as it needs. In comparison, a job cannot exceed the available bandwidth that the I/O subsystem provides. Therefore, it is necessary to observe the limitations set on the bandwidth by the I/O subsystem when analysing job bandwidth. It is reasonable to assume that most jobs will not achieve the maximum available bandwidth. This is due to most jobs only using part of the I/O subsystem and/or jobs performing I/O with less than efficient parameters.

Since the GPFS I/O counters are logged every 2min, the temporal I/O distribution has a lower resolution. As previously mentioned the I/O performed is interpolated over the 2min. As a result, the exact bandwidth at any given time and by deduction the exact maximum bandwidth is no longer available. However, by averaging the GPFS I/O counters over the 2min, it is possible to perceive some of the I/O variation over time. As a consequence the term maximum bandwidth, used here, refers to the maximum average bandwidth over the 2min. The real bandwidth could be higher than what can be measured using the GPFS I/O counters.

Fig. 4.15 shows the distribution of jobs over maximum read versus maximum write bandwidth. The average maximum bandwidth for read is over 201MiB/s and for write over 97MiB/s. Again the averages are skewed by outliers, such as the maximum measured read bandwidth of 103GiB/s and write bandwidth of 120GiB/s. These maximums are too high, when compared to the limit of 66GiB/s offered by the JUGENE I/O subsystem. It is possible to assume that any bandwidth measured that exceeds the limit, is caused by either buffering or by an error in matching GPFS I/O logs to job runtime. It is worth noting that an error in matching GPFS I/O logs to job runtime could strongly effect the maximum bandwidth measured if it falls within a wrongly matched GPFS I/O log.

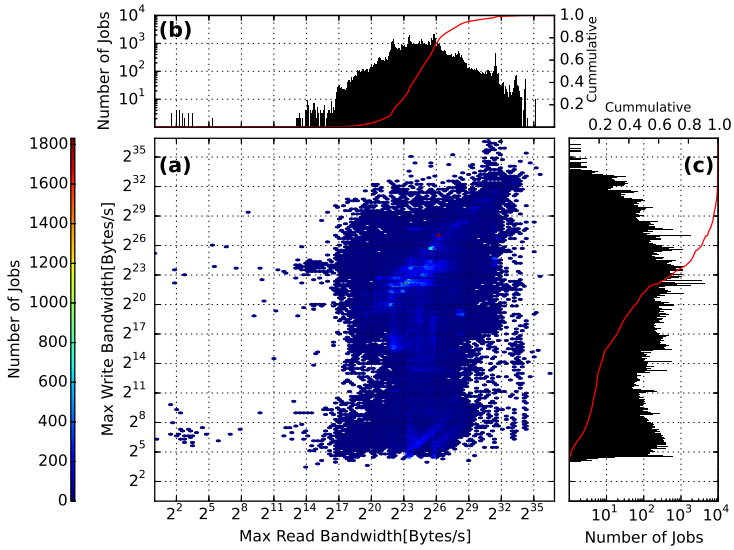


FIGURE 4.15: Read and write maximum bandwidth for analysed jobs. (a) Scatter plot of read and write maximum bandwidth with a heat map for job count, (b) Histogram of read maximum bandwidth and (c) Histogram of write maximum bandwidth.

In total 1 job for read measured a maximum bandwidth over the 66GiB/s limit, while 9 jobs for write are over the limit. When disregarding these jobs, the average of read maximum bandwidth slightly drops to 200MiB/s and for write drops to 93MiB/s. Nonetheless, the averages are still skewed by outliers, as the maximum measured bandwidth for read becomes 45GiB/s and for write 65GiB/s. Although the maximum does not exceed the peak available bandwidth, it can still be considered high, as jobs are usually expected to not reach almost full peak bandwidth. For read 80% of jobs have remained below 84MiB/s for maximum bandwidth, while for write 80% of jobs remained below 19MiB/s. It is also possible to view the maximum bandwidth achieved by individual I/O nodes for each job. However, previously dividing I/O quantity across I/O nodes has shown no change in distribution. Therefore, the same can be expected for maximum bandwidth.

The heat map in Fig. 4.15 shows the jobs rather evenly distributed over the read/write bandwidth. The exception being a coloured area on the diagonal pushing towards the upper right corner. Under perfect conditions, it would be expected that all jobs read and write with the maximum available bandwidth. In such a case the jobs would form a

large cluster on the scatter plot in the upper right corner. However, due to jobs sharing the bandwidth, buffering, file locks, request size variation and many other conditions effecting available bandwidth, the jobs attempting to reach maximum read and write bandwidth are more spread on the diagonal. This clearly indicates that the architecture of the I/O subsystem directly affects the measured bandwidth.

While maximum bandwidth is strongly limited by the I/O subsystem, an average bandwidth is a factor of the job's I/O quantity and job's duration. Additionally average bandwidth is less prone to errors in matching GPFS I/O logs to job runtime. For read 80% of analysed jobs performed an average bandwidth below 1.17MiB/s and for write below 1.25MiB/s. The maximum average bandwidth is fairly high at about 5.3GiB/s for read and 1.3GiB/s write.

Fig. 4.16 shows the jobs distribution over the average bandwidth. Compared to the maximum bandwidth, the coloured area seen in in Fig. 4.15 dissipates in Fig. 4.16. It is interesting to see that the distribution for the average bandwidth in Fig. 4.16, resembles the distribution of bytes read and written in Fig. 4.11. This could prove that job I/O increases with both job size and job duration.

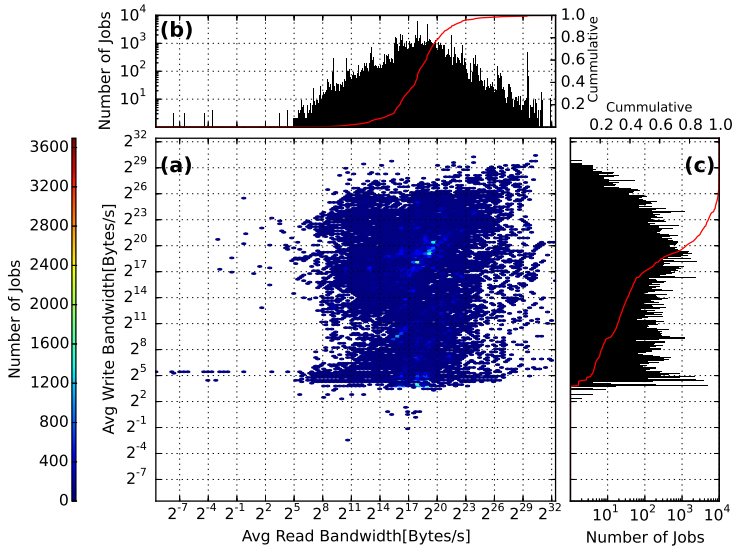


FIGURE 4.16: Read and write average bandwidth for analysed jobs. (a) Scatter plot of read and write average bandwidth with a heat map for job count, (b) Histogram of read average bandwidth and (c) Histogram of write average bandwidth.

Classification 1.4 Read/Write IOPS

Similar to bandwidth, IOPS are also limited by the physical attributes of the I/O subsystem. Although jobs cannot exceed the maximum available IOPS, it is more difficult to find an exact limit. Maximum IOPS depends on the details of the transport protocols and latencies, which are more difficult to determine without measurements. By ignoring caching effects, an upper limit to the I/O bandwidth can be determined by looking for the smallest nominal bandwidth along the data path. This is beyond the scope of this study and therefore no jobs are disregard on the basis of a too high IOPS rate. Large scale storage systems are often designed to offer high bandwidth. Since IOPS and bandwidth are closely related through the request size, it is difficult to determine which limit (i.e. bandwidth or IOPS) is reached first.

As previously discussed, measured IOPS are more influenced by the I/O subsystem than bandwidth. This is a result of the number of I/O requests being changed across different I/O stack layers. Hence, the term IOPS used here, defines the number of I/O requests per time unit conducted by the filesystem and registered by the GPFS I/O counters. While most I/O stack layers lead to a reduction in the number of I/O requests performed, it is difficult to suggest whether the actual number of I/O requests initiated by the job would finish in less or more time. As a result, it is difficult to speculate on the actual IOPS seen by the jobs when compared to the here measured IOPS.

Similar to bandwidth, the 2min logging interval of the GPFS I/O counters leads to a reduction of accuracy of the measured IOPS value. As a consequence, the maximum IOPS, referred to here, is the maximum average IOPS as measured over the 2min. It is possible for the actual maximum IOPS seen by the filesystem to be higher.

The analysed jobs average 1.5×10^3IOPS for read and 295IOPS for write. In this case the average is closer to the center than for previous I/O criteria, as 80% of jobs performed a maximum under 1.26×10^3IOPS for read and a maximum under 57IOPS for write. The maximum job observed IOPS are however still skewing the average, at 1.23×10^6IOPS for read and 732.11×10^3IOPS for write.

The distribution over read versus write IOPS is shown in Fig. 4.17. It is worth noting that IOPs cannot be fractioned, resulting in the lines formed at the bottom for jobs with low write IOPS and on the left for jobs with low read IOPS. This can also be explaining

the slight coloured areas at the bottom of the heat map. From these it appears that many jobs, about 50%, have performed a maximum IOPS equal or lower than 10 for write. Meanwhile, only 10% of jobs have performed a maximum IOPS equal or lower than 10 for read. This result is indicated by Fig. 4.17-(b) and Fig. 4.17-(c) respectively.

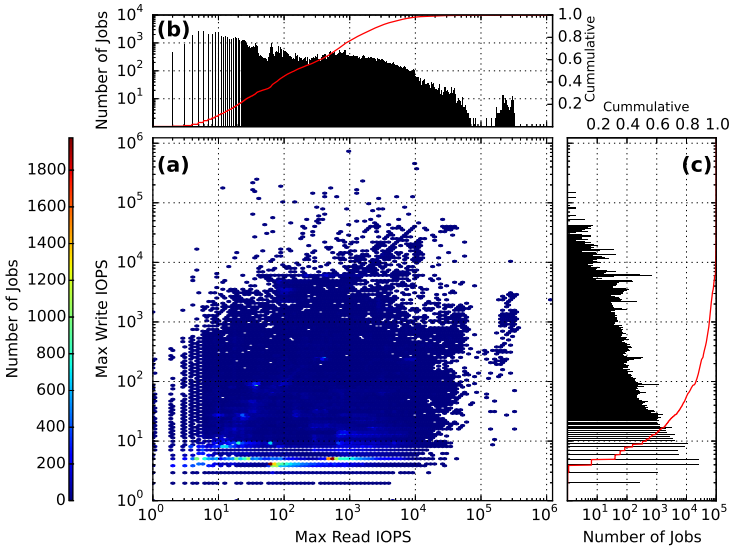


FIGURE 4.17: Read and write maximum IOPS for analysed jobs. (a) Scatter plot of read and write maximum IOPS with a heat map for job count, (b) Histogram of read maximum IOPS and (c) Histogram of write maximum IOPS.

Average IOPS over job runtime duration depends on the number of I/O requests initiated and the duration of the job. Therefore, contrary to maximum IOPS, average IOPS can have fractions. The average IOPS for read remained under 10.45IOPS and for write under 2.48IOPS for 80% of jobs. The maximum of the average IOPS is about 315×10^3 for read and 83×10^3IOPS for write. Without knowledge of request sizes, it is difficult to suggest whether these jobs are IOPS limited or not.

Fig. 4.18 shows the distribution of jobs over read versus write average IOPS. The distribution suggests no visible clusters. However when comparing Fig. 4.13 for read versus write commands, with Fig. 4.18 for average IOPS, there are slight similarities in the distribution. This again suggests that jobs increase the number of requests when increasing job duration.

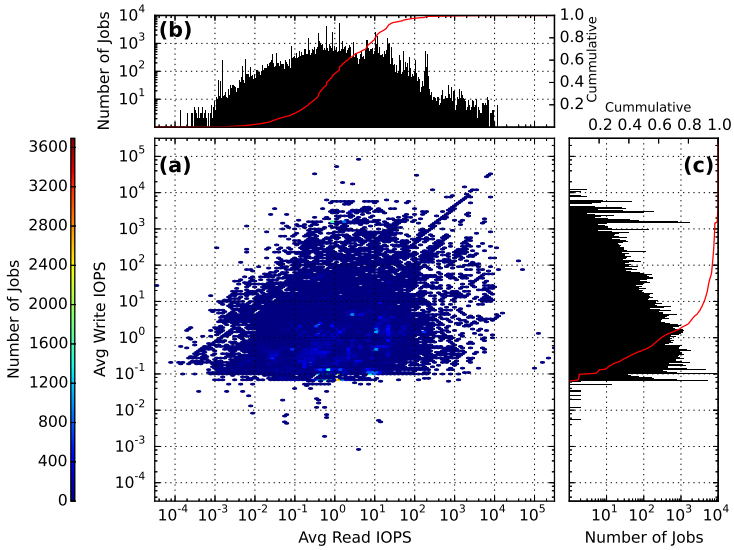


FIGURE 4.18: Read and write average IOPS for analysed jobs. (a) Scatter plot of read and write average IOPS with a heat map for job count, (b) Histogram of read average IOPS and (c) Histogram of write average IOPS.

Classification 1.5 Total number of files created

While total number of files created is an interesting criteria to measure, the GPFS I/O counters do not allow for it. Only file open and close commands are counted. It is not possible to distinguish newly created files from old files that are being opened. Although the GPFS I/O counters count the number of inode updates, inferring created files from it might result in unreliable results. This clearly indicates the limitations that a measuring method might set on the resulting I/O criteria that can be evaluated. Logging file creation and deletion requires access to metadata services.

The I/O criteria, files created, as presented here investigates the file open and close commands as seen on the filesystem. The analysis should observe the errors in the counts of open and close commands when matching to job runtime as discussed in 4.4.1. When observing that the matching error found is almost constant across the jobs, it is possible to assume that all have been created by the job. Even open and close commands initiated by the system to load and run the job can be attributed to the job's operation. The emphasize is therefore on the filesystem, which has to fulfil these

commands irrespective of them being initiated by the job or the system as a result of the job running.

Fig. 4.19 and Fig. 4.20 show the histograms of both open and close commands distribution over jobs respectively. Both figures indicate a relatively large number of opened and closed files. A total of about 66×10^9 opened commands and a total of about 64×10^9 close commands were initiated by the analysed jobs. The maximum lies in a job that created around 969×10^6 from each open and close commands. However, 80% of jobs used less than 38×10^3 open commands and less than 33×10^3 close commands. As a result 20% of jobs are responsible for almost 99% of both open and close commands.

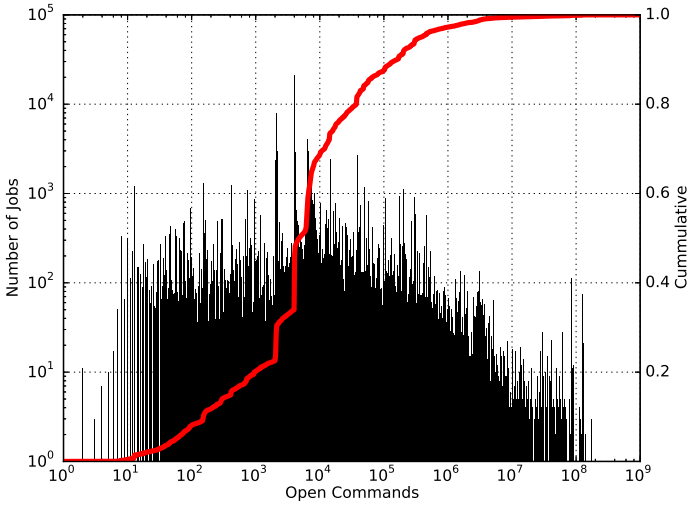


FIGURE 4.19: Open commands for analysed jobs.

Although it is possible to expect that the number of open files is equal to the number of closed files, this has not been always the case for the analysed jobs. A total of 140755 have more open commands, while 13441 have more close commands. Only 12744 have equal number of open and close commands. Around 80% of the jobs have open and close commands divert from each other by a relative difference (i.e. the difference divided by the total number of open and close commands) of about 11.5%. Nevertheless, Fig. 4.19 and Fig. 4.20 show almost equal distribution of open and close commands respectively, over jobs.

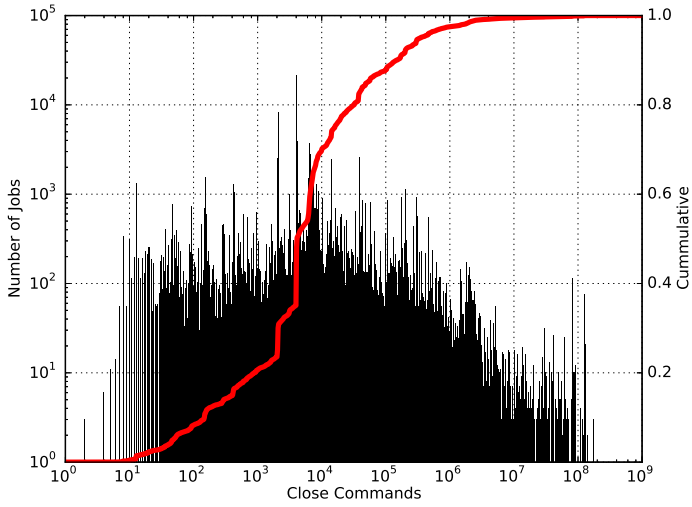


FIGURE 4.20: Close commands for analysed jobs.

While numbers ranging in the thousands for open and close commands appear high, the relative system size should be observed. JUGENE contains a total of 73728 compute nodes. Each can run upto 4 processes resulting in 294912 possible processes. A job running in task-local scheme can easily open a relatively large number of files. When dividing the number of open or close commands over the number of I/O nodes an almost equal distribution of jobs, as the ones given in Fig. 4.19 and Fig. 4.20, can be created. Jobs could be operating with a larger number of files when increasing in size.

Classification 1.6 I/O intensity

As discussed in Chp. 3 (Classification 1.6), I/O intensity is the fraction of time spent in I/O, and can therefore be influenced by many parameters. This results in many issues to be observed when analysing job I/O behaviour using I/O intensity. One such issue is the I/O stack layer on which the I/O is measured. The time spent in I/O could be changed relative to the job's I/O time when moving down the I/O stack layers. In the case of computing the I/O intensity of jobs using the GPFS I/O logs, the I/O intensity would be effected by any operation that decouples the job from it's I/O before logging. Since the GPFS I/O logs are measured on an I/O forwarding layer and in the filesystem, the I/O

requests timings could be changed before being logged. For example, any intermediate layer might perform buffering. The GPFS I/O counters would then register the I/O when such buffer contents are being written to the filesystem. Asynchronous I/O would have a similar effect. Therefore, the I/O intensity analysed here, can be redefined as the time during a job in which the filesystem performed I/O.

Furthermore, the I/O intensity is expected to be sensitive to the temporal granularity at which the measurements are performed. Therefore, another issue to be observed on computing I/O intensity, is the requirement of the full temporal information as to start and end time of I/O requests. The GPFS I/O counters are logged every 2min leading to a reduced temporal resolution. For the purpose of analysing the GPFS I/O logs the values are interpolated over the span of the 2min to achieve a resolution of $\Delta t = 1$, resulting in a possibly higher I/O intensity than actually performed by the jobs. To clarify this effect, an example can be given of a job that features phases of 30sec of intensive I/O every 2min. The I/O intensity would have the values 0.25, 0.5 and 1 when measuring with an interval of 30sec, 60sec and 2min respectively.

To increase the information delivered by the I/O intensity and to attempt decreasing the influence of the GPFS I/O log 2min span, the intensity factor can be tuned. The I/O intensity definition allows for 2 parameters that change the definition of I/O dominating time periods ($\Delta t = 1$). The first is the threshold that computes the time in which I/O is dominating, given by the variable c in Eq. 3.30. The second, is the I/O property, as either I/O quantity (bytes read or written) or I/O commands (IOPs), which is compared to the defined threshold. As a result, computing the I/O intensity at various thresholds for different properties and by analysing and comparing the distribution, it is possible to further understand the I/O intensity of the analysed jobs.

A multi-threshold analysis of I/O intensity as seen by the GPFS I/O logs is given in Fig. 4.21. The figure shows the distribution of jobs over the I/O intensity for different thresholds. While Fig. 4.21-(a) shows I/O intensity using I/O commands with a threshold of $c = 0$, the rest of the figure shows I/O intensity using bytes read and written with various thresholds. With the exception of Fig. 4.21-(b) computed using bytes read/written and threshold $c = 0$B, all distributions are similar with jobs experiencing a shift to lower I/O intensities as the threshold increases.

Using a bytes read/written with a threshold $c = 0$B for computing I/O intensity given in Fig. 4.21-(b) shows that more than 99% of jobs achieve an I/O intensity of 1.0. It is possible to conclude that I/O is continuously performed. As this conclusion cannot be made when the threshold is increased, this might just be a result of interpolating the GPFS I/O logs over the span of the 2min. There is also the possibility that jobs continuously perform small I/O, e.g. write every minute a short message into a log file. Since there is no interest in such small I/O operations, it makes sense to introduce the threshold and select it high enough such that intervals where only small logging occurs are filtered out. In addition to that, when equating the bandwidth of the I/O subsystem available to JUGENE of 66GiB/s (about 112MiB/s when divided over I/O node), a threshold below 1MiB appears small in comparison. Therefore, the I/O intensity distribution is likely to resemble the one with the threshold of 1MiB, with 80% of jobs having an I/O intensity below 0.2. As a result, it can be concluded that most jobs while running do not lead the filesystem on the I/O nodes to spend a lot of time in I/O. Some jobs achieve an I/O intensity of 1 even when using a threshold of $c = 1$MiB. These jobs could be suitable for further study on various I/O architectures.

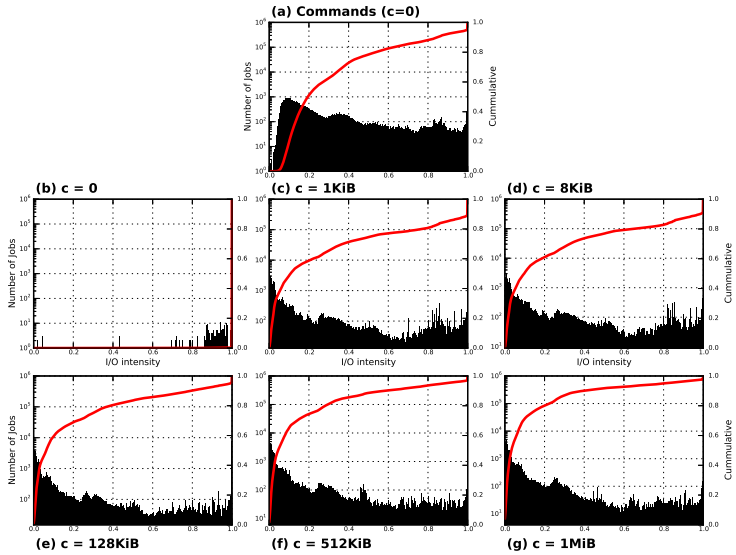


FIGURE 4.21: I/O intensity for analysed jobs computed using various thresholds c. (a) I/O commands $c = 0$, (b) $c = 0$B, (c) $c = 1$KiB, (d) $c = 8$KiB, (e) $c = 128$KiB, (f) $c = 512$KiB and (g) $c = 1$MiB.

The I/O intensity can also be split into read and write I/O intensity and plotted for

jobs against each other. This has the advantage of showing whether read or write is responsible for a high I/O intensity. The results of such analysis can be seen in the scatter plots shown in Fig. 4.22. To easier understand the scattering of jobs the dotted red lines are added to represent the 80% marker. That is 80% of jobs have a read intensity below the horizontal red line and 80% of jobs have a write intensity to the left of the vertical red line. From these it is possible to conclude that write is for many jobs responsible for the high I/O intensity. This is proven by the bulk of jobs scattering to the left as the threshold is increased. Additionally, the I/O intensity computed using bytes read/written and threshold $c = 0$B has many jobs in the left upper corner. Leading to the conclusion that write is fairly spread across the many jobs runtime. However, the almost constantly visible diagonal line on which many jobs are scattered for higher thresholds, indicates that other jobs have an equal I/O intensity for both read and write.

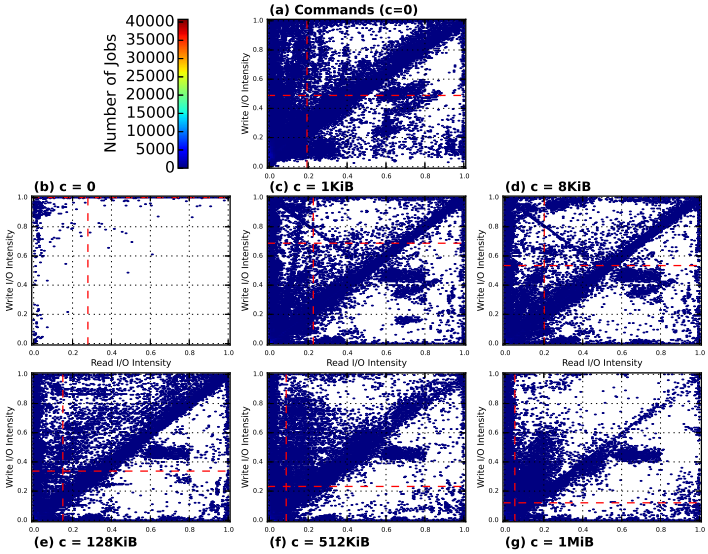


FIGURE 4.22: Read and write I/O intensity for analysed jobs computed using various thresholds c. (a) I/O commands $c = 0$, (b) $c = 0$B, (c) $c = 1$KiB, (d) $c = 8$KiB, (e) $c = 128$KiB, (f) $c = 512$KiB and (g) $c = 1$MiB.

To complement the information given in Fig. 4.22 and to highlight the individual distribution of I/O intensity over read and write, Fig. 4.23 shows the scatter and histograms for I/O intensity computed using bytes read and written at a $c = 1$MiB threshold. As seen when comparing Fig. 4.23-(b) and Fig. 4.23-(c), many jobs seem to have a higher write I/O intensity. This is proven by 80% of jobs having a read I/O intensity below

0.05 and a write I/O intensity below 0.12, which is more than double. Nonetheless, the conclusion remains that most jobs do not have a high I/O intensity, remaining under 0.2 for 80% of the jobs. This could indicate that either jobs are not I/O intense or that the I/O subsystem has a higher capability than the jobs require.

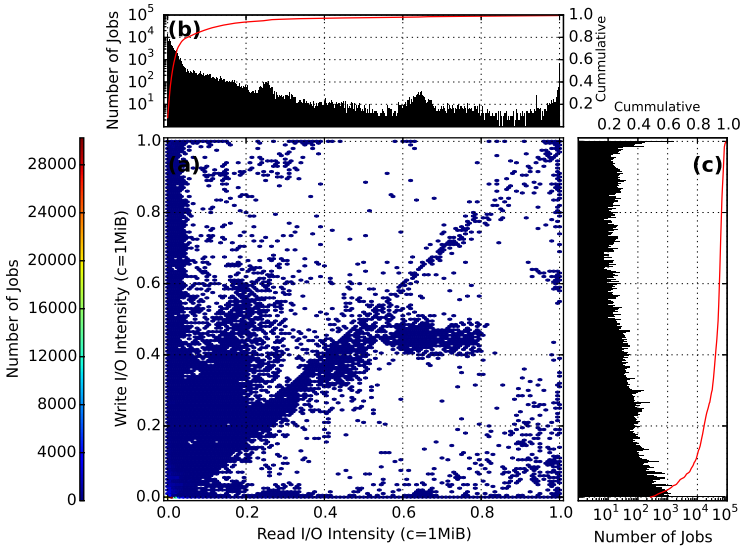


FIGURE 4.23: Read and write I/O intensity ($c = 1$MiB) for analysed jobs. (a) Scatter plot of read and write I/O intensity with a heat map for job count, (b) Histogram of read I/O intensity and (c) Histogram of write I/O intensity.

4.5.4 Category 2: I/O Pattern Analysis

Classification 2.1 Distribution of request sizes

As previously mentioned, the number and therefore the size of I/O requests can be changed while traversing different I/O stack layers. For measuring I/O using the GPFS I/O logs any collective I/O performed in higher I/O stack layers would result in changing the I/O request size. On the basis of collective I/O and data sieving, till now the discussion only introduced a possible decrease in the number of I/O requests and a resulting I/O request size increase. However, some I/O operations and I/O stack layers require decreasing the size of I/O requests. An example for this behaviour on JUGENE is introduced in Sec. 2.1. The CIOD buffers on the I/O nodes are set to 4MiB. A

larger I/O request has to be split into multiple requests on the compute node. As a result the number and size of I/O requests can be changed compared to the I/O directly requested by the application. Buffering could also result in an increase of request size. For example, multiple I/O requests could be collected and bundled together on the I/O node in the GPFS pagepool.

GPFS I/O counters count the number of read and written bytes and the number of I/O commands. There however is no relationship set between the two counts. That is, bytes read or written cannot be related back to a specific I/O request. Therefore the I/O request size is not measured when using the GPFS I/O logs. As the I/O logging is done with an interval of 2min, it is possible to create a 2min average I/O request for both read and write. The I/O request sizes presented here are 2min averages. The delay of I/O counting given in Sec. 4.4.1, specifically Fig. 4.8, might effect the measured average I/O request size. As the figure shows the delay is almost identical for both bytes read and read commands and therefore should result in the same estimate for requests size average. Such influences could result in variation of average request size and therefore should be considered.

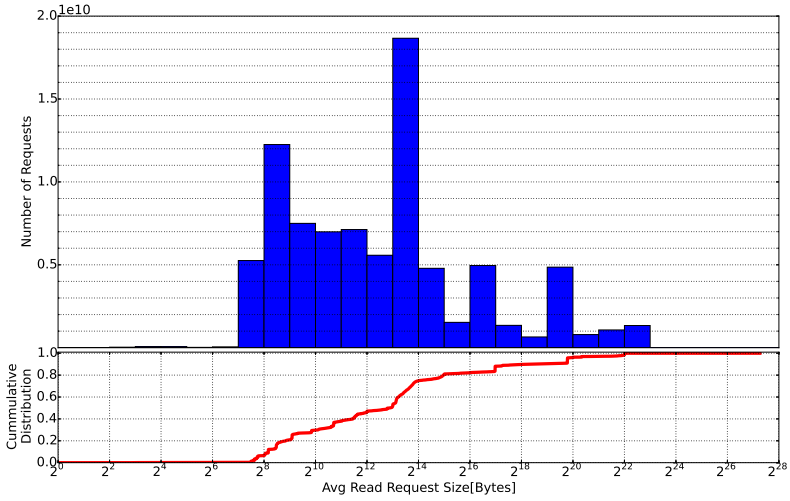


FIGURE 4.24: Distribution of read request sizes.

Fig. 4.24 shows the distribution of all read requests sizes. More than 80% of requests are below 64KiB, which is rather small given that the filesystem block is 4MiB. On the

other hand, Fig. 4.25 shows the distribution of write request sizes, which indicates that write has smaller request sizes than read. Above 80% of write requests are below 1KiB. Although the CIOD buffer should not allow for larger than 4MiB request sizes, it is possible that the few larger I/O requests are caused by buffering.

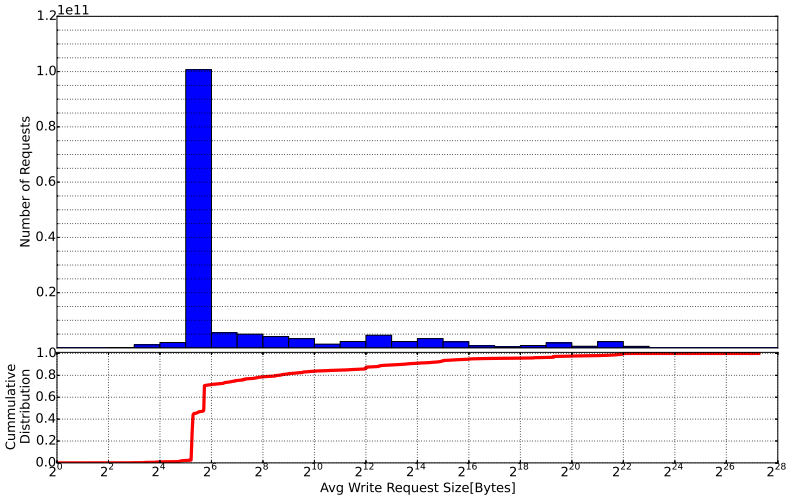


FIGURE 4.25: Distribution of write request sizes.

Showing the distribution of I/O request sizes for every analysed job is not possible. For the purpose of comparison, the distributions given in Fig. 4.24 and Fig. 4.25 do not relate I/O request sizes to jobs. To partially link I/O request sizes to jobs, Fig. 4.26 shows the distribution of jobs over job average requests sizes, which are computed by dividing the total bytes read or written by the total number of read or write commands for each job. The distribution shows that almost all jobs do not have an average job read or write request size above 4MiB as dictated by the use of CIOD buffers. In total, about 80% of jobs have an average job read request size of 1.7MiB and an average job write request size of 881KiB.

Generally for the period analysed, more write than read requests are counted. Write is about 59% of all I/O commands. Meanwhile, the number of total bytes read are more than the number of bytes written. Read bytes represents about 55% of all bytes read or written. As concluded from the analysis, smaller write than read requests are initiated by jobs.

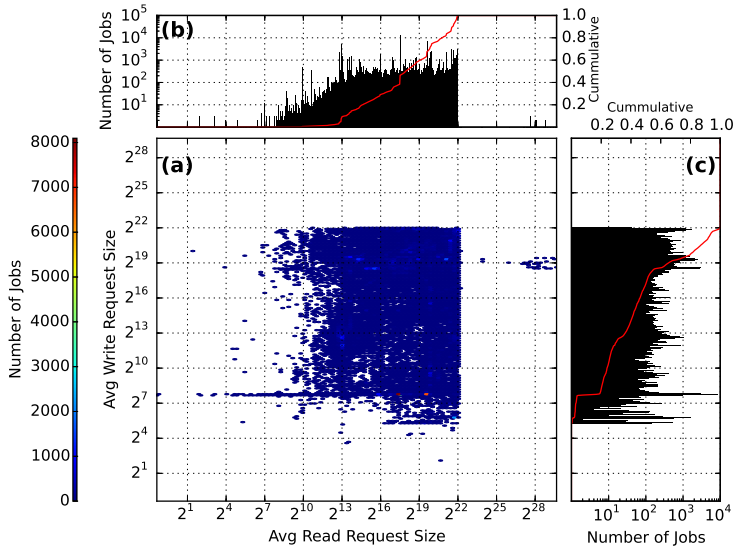


FIGURE 4.26: Read and write average request size for analysed jobs. (a) Scatter plot of read and write average request size, (b) Histogram of read average request size and (c) Histogram of write average request size.

Classification 2.2 Percentage of small I/O requests

In Chp. 3 (Classification 2.2), small I/O is defined as any I/O with request size smaller than a given limit s_{small}. As previously explained, the request sizes used here are averages over 2min and measured on the filesystem. Many I/O stack layers attempt improving I/O, forming larger requests by combining small I/O requests. With the exception of CIOD buffers breaking very large (>4MiB) I/O requests, the application itself could be expected to create smaller I/O requests and as a consequence have a larger percentage of small I/O.

When introducing percentage of small I/O in Chp. 3, it is suggested that a reasonable value for s_{small} is the filesystem block size (s_{block} =4MiB). Tab. 4.9 shows that most I/O requests are indeed smaller than s_{block}. To further understand the percentage of small I/O, Tab. 4.9 expands on different possible s_{small} sizes. The results given in the table confirm conclusions in Classification 2.1, as larger percentage of small I/O exists for write compared to read, irrelevant of s_{small}.

s_{small}	4KiB	8KiB	64KiB	128KiB	512KiB	1MiB	4MiB
Read small I/O[%]	46	52	82	88	90	96	98
Write small I/O[%]	86	89	94	95	96	97	99

TABLE 4.9: Percentage of small I/O for various s_{small}

One advantage for the use of percentage of small I/O is reducing the distribution of I/O request sizes to a single number. It therefore becomes possible to present the distribution of small I/O percentage for the analysed jobs. Fig. 4.27 and Fig. 4.28 show the distribution of small read and write I/O respectively over analysed jobs for various s_{small} values. Both figures show the same conclusions as Tab. 4.9, that most jobs use relatively small I/O requests. A few jobs manage to have a low small I/O percentage for read or write despite increasing s_{small}.

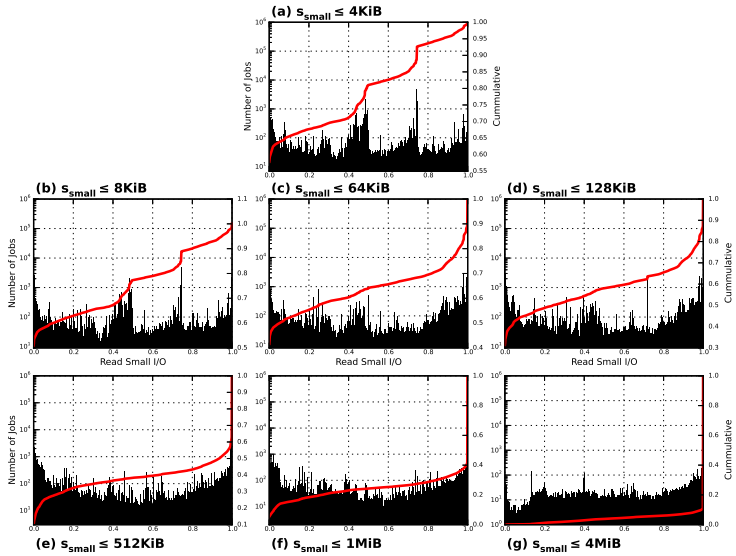


FIGURE 4.27: Percentage of small read I/O for analysed jobs computed using various s_{small}. (a) $s_{small} < 4$KiB, (b) $s_{small} < 8$KiB, (c) $s_{small} < 64$KiB, (d) $s_{small} < 128$KiB, (e) $s_{small} < 512$KiB, (f) $s_{small} < 1$MiB and (g) $s_{small} < 4$MiB

On the basis of analysed jobs, it is reasonable to assume that large scale I/O subsystems have to be capable of performing well when dealing with small I/O. Given that the GPFS I/O counters are logged in the filesystem, it appears that the storage system cannot even depend on higher I/O stack layers forming larger I/O requests using collective I/O or other methods. However, it is unknown if the I/O subsystem serving JUGENE suffers from the use of small I/O. When possible, tests should be conducted along side analysing

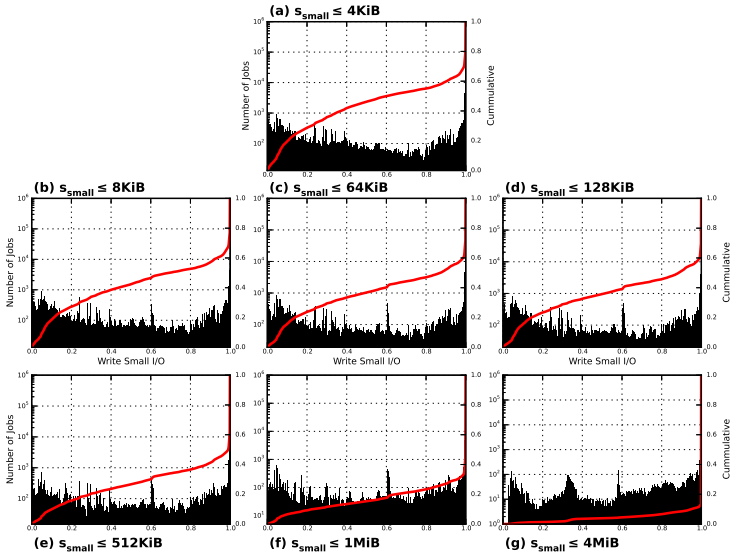


FIGURE 4.28: Percentage of small write I/O for analysed jobs computed using various s_{small}. (a) $s_{small} < 4$KiB, (b) $s_{small} < 8$KiB, (c) $s_{small} < 64$KiB, (d) $s_{small} < 128$KiB, (e) $s_{small} < 512$KiB, (f) $s_{small} < 1$MiB and (g) $s_{small} < 4$MiB

percentage of small I/O to determine it's impact on the I/O performance. Even then, changing I/O subsystem parameters to accommodate a large percentages of small I/O should be done with caution. While the number of small I/O operations might be very large, the remaining large I/O operations could be responsible for writing/reading most of the data. In such a case the system would not be positively effected from possibly increasing delay of large I/O to accommodate more small I/O requests.

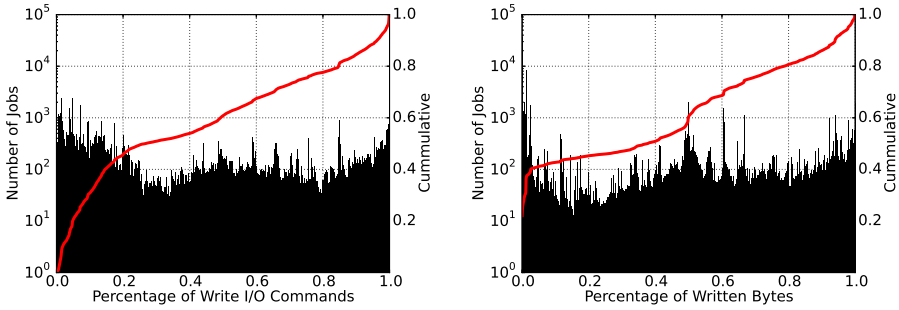
Classification 2.3 Request size: Variable vs. fixed

To determine whether given I/O requests are variable or fixed requires analysing several runs of the same application. Additionally, it requires understanding the parameters and input datasets. As the GPFS I/O logs analyse jobs that cannot be easily linked to the application, it is difficult to analyse the variability of request sizes. Another issue that leads to this limitation, is the use of 2min average I/O request sizes for the analysis. These result in a difficulty when using the GPFS I/O logs to determine whether a specific I/O request changes its size when the application is rerun.

Classification 2.4 Percentage of I/O type

The percentage of I/O type as defined in Chp. 3 (Classification 2.4) can be calculated using either I/O quantity in terms of bytes or I/O commands. Since the number of I/O commands can be changed on various I/O stack layers, the percentage of I/O type based on I/O commands might be different than the application when measured using GPFS I/O logs. Excluding any error in matching the GPFS I/O logs to the job's runtime, when using I/O quantity the percentage of I/O should resemble that expected by the application.

Fig. 4.29 shows the distribution of write percentage of I/O for both I/O commands and bytes read/written in Fig. 4.29-(a) and 4.29-(b) respectively. Since the same figure for read would simply be a flipped mirror of write, any information for read can be concluded from the write figures. Therefore, the read figures are omitted.



(a) Percentage of write I/O commands from total I/O commands

(b) Percentage of written bytes from total read and written bytes

FIGURE 4.29: Percentage of write for analysed jobs

When comparing the two write distributions in Fig. 4.29, the faster rise of the percentage of write bytes becomes obvious, indicating that almost 40% of jobs have performed almost 100% of bytes as read. For percentage of write I/O commands the rise is slower. However once past the 20% write marker both distributions are similar. In fact both arrive at the 50% write marker at 60% of the jobs, that is for both 40% of jobs have performed above 50% write.

It is mentioned in classification 2.1 that in total 59% of I/O commands are write, while only 45% of bytes are written. By comparing the distribution of write command percentage to written bytes percentage, it is possible find the jobs contributing to increasing the total percentage of write commands, while not equally contributing to the bytes written. Equal conclusions can be made from the read percentage distributions.

Classification 2.5 Dominating I/O operation type

Dominating I/O operation type evaluates the distribution of I/O type over time. Due to the temporal dimension it is difficult to visualize for the entire set of analysed jobs. Although some visualization methods exist, it is hard to see a benefit from it. Any conclusions on the entire set of analysed jobs using dominating I/O operation type can also be drawn from classification 2.4; percentage of I/O type. Dominating I/O operation type is more suitable to understand the distribution of I/O types within a job. Individual job analysis using I/O criteria is performed for a selected set of jobs in Sec. 4.5.6.

The analysis of dominating I/O operation type using GPFS I/O logs has similar issues to consider as those mentioned for classification 2.4 percentage of I/O type. I/O stack layers that change the number of I/O commands could effect the distribution of the dominating I/O operation type. In addition to that, the GPFS I/O counters being logged once every 2min could influence the temporal resolution of the analyses.

Classification 2.6 Task-local vs. shared

Analysing the percentage of task-local to shared files for jobs could lead to interesting conclusions. However, it cannot be perceived using the GPFS I/O logs. This is due to the open and close commands registered by the GPFS I/O counters not being linked to the individual opened or closed files. As a result, there is no way of recognizing two open commands as belonging to either the same or different files, whether on the same or on different I/O nodes. This is a basic requirement for differentiating task-local from shared files.

Classification 2.7 Spatial access pattern classification

As mentioned in the introduction of this section, the GPFS I/O counters do not register addresses of I/O requests. Therefore the spatial behaviour of jobs cannot be retrieved using the GPFS I/O logs. Generally, investigating the spatial access pattern behaviour is difficult for a large set of jobs. This is due to the need of saving all files and file offsets for each I/O request. Additionally, identifying the spatial access pattern is in itself a difficult task for which it is hard to create reliable algorithms. As a result, it could be considered that spatial access pattern classification is more suitable for deeper analysis of a limited set of fully available applications. Many other approaches allow for analysing the spatial pattern classification and have been discussed in the related work Sec. 4.1.

Classification 2.8 Temporal distribution of I/O

The temporal distribution brings the time factor into the total amount of data read-/written and into the total number of IOPs. Similar to classification 2.5; dominating I/O operation type, the temporal distribution is difficult to visualize for a large set of jobs. Therefore it is performed on a limited set of analysed jobs in Sec. 4.5.6.

The temporal distribution analysis has to observe the GPFS I/O counters being logged every 2min. This leads to a change in the resolution of the temporal distribution. Any I/O stack layers altering the data read/written or the number of IOPs, would also result in a change in the temporal distribution.

Classification 2.9 Burstiness parameter

The main difficulty of analysing the temporal I/O behaviour of a large set of data, is visualization. The burstinesss parameter reduces the temporal I/O behaviour to a quantifiable parameter that informs on the overall temporal behaviour of jobs. As with all time based analysis, I/O stack layers might shift or change I/O requests in time using buffering or other methods. As a result the burstiness parameter is subject to changing compared to job burstiness as I/O requests pass through the I/O stack layers before being logged by the GPFS I/O logs. Therefore, the burstiness measured here is

redefined as the bursts seen by the filesystem on the I/O nodes during the job's runtime. Additionally the 2min resolution of the GPFS I/O logs could change the burstiness of analysed jobs. As a consequence of interpolating the GPFS I/O counters over the 2min, the burstiness is expected to be reduced. Therefore, the actual burstiness of applications could be higher than that computed using the GPFS I/O logs.

Similar to I/O intensity, the computation of burstiness can employ various thresholds c on either I/O quantity (bytes read or written) or I/O commands (IOPs). The threshold c therefore dictates the height a burst must achieve to influence the burstiness parameter. This prevents small I/O operations which have little impact on the I/O system to dominate the analysis.

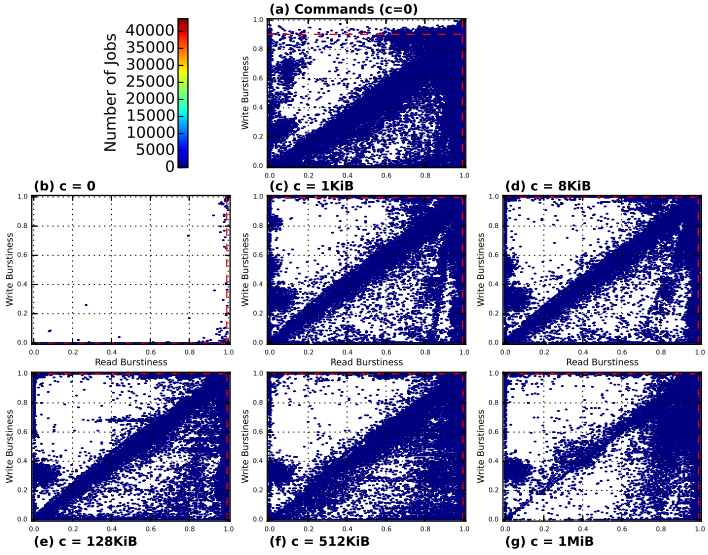


FIGURE 4.30: Read and write burstiness of analysed jobs computed using various thresholds c. (a) I/O commands $c = 0$, (b) $c = 0$B, (c) $c = 1$KiB, (d) $c = 8$KiB, (e) $c = 128$KiB, (f) $c = 512$KiB and (g) $c = 1$MiB.

Fig. 4.30 represents a study of the burstiness of analysed jobs using various thresholds. As seen from the scatter plots, it appears that read exhibits a higher burstiness than write, visible by the jobs scattering towards the right. This is proven by using a threshold $c = 0$B shown in Fig. 4.30-(b), where 99% of jobs had a write burstiness of 0, while almost 75% of jobs exhibit a read burstiness of over 0.8. As the threshold is increased, more

write burstiness becomes visible and as expected both read and write burstiness increase. However, read burstiness remains as a whole slightly larger than write burstiness.

Another interesting aspect seen from Fig. 4.30 is the almost constantly visible diagonal on which many jobs scatter. This indicates that for these jobs the burstiness for both read and write are almost equal. Finally, to complement the conclusions made from the scatter plots, Fig. 4.31 shows the scatter plot of read versus write burstiness and the individual distribution of read and write burstiness at threshold $c = 1\text{MiB}$. In this case the analysed jobs average 0.93 read burstiness with a standard deviation of 0.17. For write burstiness the average is 0.87 with a standard deviation of 0.25.

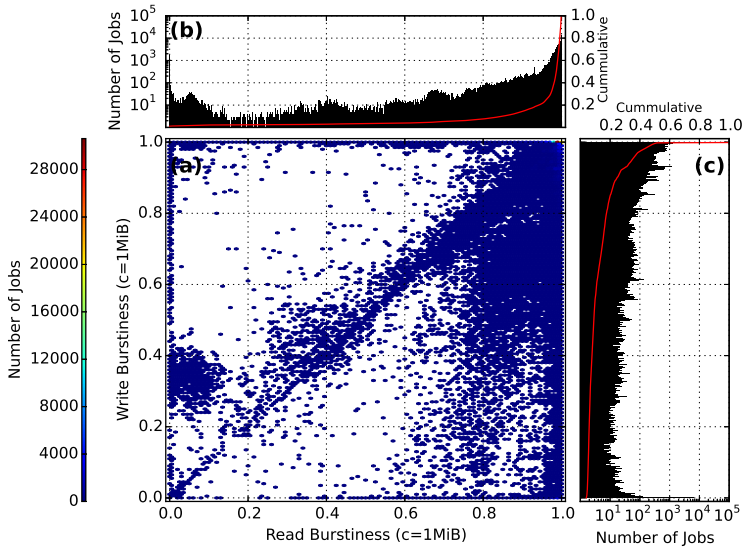


FIGURE 4.31: Read and write burstiness for analysed jobs with $c = 1\text{MiB}$. (a) Scatter plot of read and write burstiness, (b) Histogram of read burstiness and (c) Histogram of write burstiness.

Classification 2.10 Access pattern repetitive behaviour

As defined in Chp. 3 (Classification 2.10), access pattern repetitive behaviour requires analysing the spatial access pattern and comparing it between different files and processes. Since the GPFS I/O logs do not allow for a spatial access pattern analysis, it is not possible to form any conclusions on the access pattern repetitiveness. As a result, similar difficulty exists for both analysing spatial access patterns and their repetitiveness.

Analysing access pattern repetitive behaviour requires the ability to not only recognize the access pattern, but to be able to compare and conclude whether two exhibited patterns are equal. This adds a dimension of difficulty for spatial access pattern analysis when analysing a large set of job's I/O behaviour.

Classification 2.11 Dominating I/O operation repetitiveness

Detecting a repetitive behaviour in the time distribution of dominating I/O type requires fine tuning the period δt for which a repetition is expected. As a result, in Chp. 3 (Classification 2.11) when introducing dominating I/O operation repetitiveness, it is suggested to plot the count of number of occasions the same I/O operation type dominates (Eq. 3.58) as a function of the period δt. A repetition can be detected if the count for same I/O operation dominance in a given time interval is high. In practice performing such an analysis for a large set of jobs is a complex process and difficult to visualize. Additionally, the process of selecting a count that would determine whether a repetition exists could lead to mistaken assumptions.

The GPFS I/O logs pose further limit a large scale dominating I/O operation repetitiveness analysis. The 2min intervals for logging GPFS I/O counters limit the available resolution of the temporal distribution of I/O and therefore also the temporal distribution of dominating I/O operation. As a result a repetition might be detected that does not exist for the original application when using higher temporal resolution analysis. Any I/O stack layer that might change the temporal distribution of I/O would change the temporal distribution of dominating I/O and it's repetitiveness.

It is preferable to attempt detecting repetitiveness in dominating I/O type for individual applications directly from it's algorithm. Using GPFS I/O logs when analysing individual jobs, it is possible to use the analysis methods suggested in Chp. 3 and Eq. 3.58. However, the notes attached to using the GPFS I/O logs should be observed. Finally, it is possible to form some conclusions on the dominating I/O operation repetitiveness using a visualization of the temporal distribution of the dominating I/O operation type.

4.5.5 Category 3: Parallel I/O

Classification 3.1 Parallel I/O intensity

Parallel I/O intensity as described in Chp. 3 (Classification 3.1), is a measurement of real I/O parallelism and depends on the temporal distribution of analysed jobs. As a result any I/O stack layer changing the temporal distribution will result in a change in the parallel I/O intensity. Using the GPFS I/O logs measured in the filesystem therefore results in a different parallel I/O intensity than if, for example, measured directly from the application.

Enabling analysis of the full range of parallelism of a job, requires the ability to distinguish the I/O of each individual process. In most cases logging I/O counters from within a filesystem obscures the process performing I/O and leads to loss of I/O parallelism information. Although GPFS I/O logs are measured in the filesystem I/O stack layer, the logging is done on the I/O nodes. This allows for the ability to partially retrieve the I/O parallelism information.

Jobs analysed could be using a large number of processes running on the compute nodes, which forward I/O requests to their I/O node serving as many as 128 compute nodes. The ratio of processes served by I/O nodes can increase when each compute node runs the maximum of 4 processes, in which case I/O nodes could serve up to 512 processes. Therefore, measuring parallelism using GPFS I/O logs is a reduction of possible actual available parallelism of JUGENE. The effect of this parallelism reduction is increased since, as reported in Sec. 4.5.1 and shown by Fig. 4.10a, most jobs employ only 512 compute nodes and therefore only have 4 I/O nodes.

It is possible to argue that measuring parallelism passed the I/O forwarding layer is more appropriate. The argument rests on the I/O nodes already reducing the number of processes or nodes accessing the filesystem in parallel. Therefore, measuring I/O node I/O parallelism might be more interesting than measuring compute node I/O parallelism. Additionally I/O nodes through I/O forwarding could be performing I/O collection and/or buffering, which change I/O request timing, thereby decoupling compute node I/O parallelism from actual filesystem access parallelism.

Similar to measuring I/O intensity, the threshold c is used to prevent small I/O from dominating the analysis of parallel I/O . Fig. 4.32 shows the parallel I/O intensity for jobs computed using various thresholds. With the exception of Fig. 4.32-(b) using $c = 0$B, for the most part the distributions are similar and as expected the parallel I/O intensity is reduced when increasing the threshold. Additionally the distributions suggest that jobs almost evenly range from no to full parallel I/O intensity.

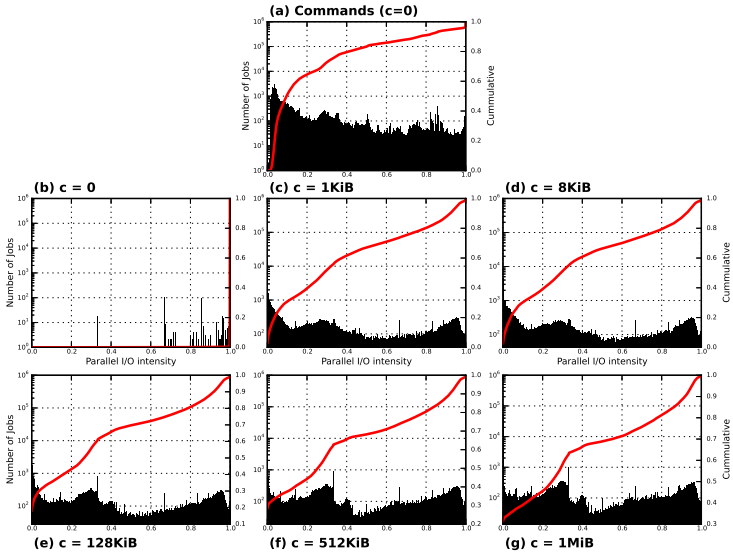


FIGURE 4.32: Parallel I/O intensity for analysed jobs computed using various thresholds c. (a) I/O commands $c = 0$, (b) $c = 0$B, (c) $c = 1$KiB, (d) $c = 8$KiB, (e) $c = 128$KiB, (f) $c = 512$KiB and (g) $c = 1$MiB.

Studying the parallel I/O intensity in Fig. 4.32, the distribution in Fig. 4.32-(b) for $c = 0$B appears different than other thresholds. To find whether read or write is responsible for a high or low parallel I/O intensity, it is helpful to untangle read from write and form a 2 dimensional distribution as given by the scatter plots in Fig. 4.33. Specifically, from Fig. 4.33-(b) it is possible to deduce that write causes the high parallel I/O intensity observed in Fig. 4.32-(b).

To facilitate tracking the changes of the distributions in Fig. 4.33 when increasing threshold c a vertical and a horizontal 80% marker is placed on each scatter plot. That is 80% of jobs write with a parallel I/O intensity below the horizontal line. Meanwhile, 80% of jobs are located left of the vertical line and read with a parallel I/O intensity

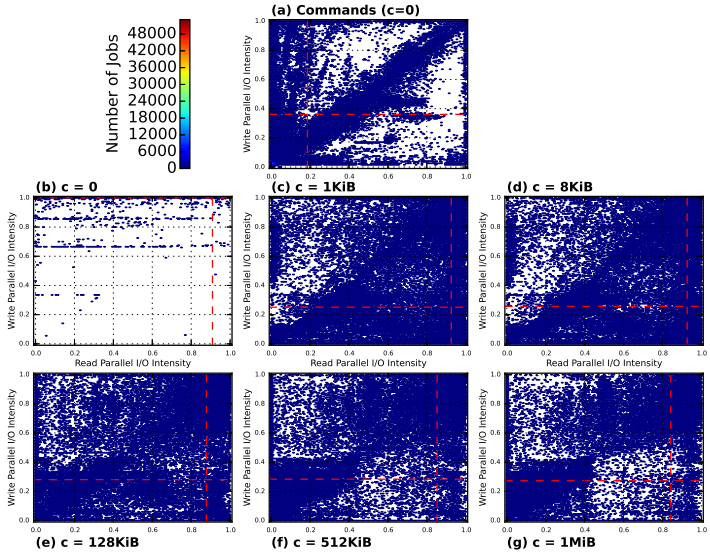


FIGURE 4.33: Read and write parallel I/O intensity for analysed jobs computed using various thresholds c. (a) I/O commands $c = 0$, (b) $c = 0$B, (c) $c = 1$KiB, (d) $c = 8$KiB, (e) $c = 128$KiB, (f) $c = 512$KiB and (g) $c = 1$MiB.

below the vertical lines position on the x-axis. From these it is possible to see that a major drop in write parallel intensity happens when increasing threshold from $c = 0$B to $c = 1$KiB, while read parallel intensity remains almost the same[9]. Increasing the threshold further appears to only slightly effect the overall parallel I/O intensity of all jobs, while maintaining a higher read parallel intensity.

To complement the discussion on parallel I/O intensity distribution, Fig. 4.34 shows the histograms of read and write parallel intensity distributions for $c = 1$KiB. The figure shows that 80% of jobs have read parallel intensity below 0.92 and a write parallel intensity below 0.25. Furthermore, it shows that there is a large group of jobs (around 64%) with a write parallel intensity of zero, while only around 1.6% of jobs have a read parallel intensity of zero.

From the given analysis, it is possible to conclude that most analysed jobs perform better parallelism on the I/O node level for read than for write. This is the opposite of I/O

[9]It is possible for parallel I/O intensity to increase when increasing threshold c as seen when comparing read parallel intensity's 80% marker in Fig. 4.33-(b) with Fig. 4.33-(c). This happens when the higher threshold c reduces $\sum_t H_{\Delta t}(t)$ faster than reducing $\sum_t \rho_{\Delta t}(t)$ in Eq. 3.60

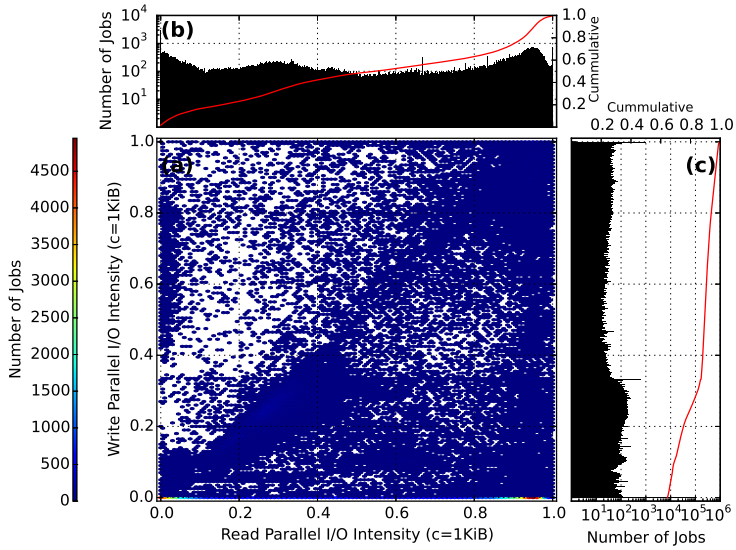


FIGURE 4.34: Read and write parallel I/O intensity for analysed jobs with $c = 1$KiB. (a) Scatter plot of read and write parallel I/O intensity with a heat map for job count, (b) Histogram of read parallel I/O intensity and (c) Histogram of write parallel I/O intensity.

intensity where write had a higher intensity than read. Therefore, from the analyses it appears that jobs spend more time writing than reading, but read more often in parallel.

Classification 3.2 I/O operation concurrency

While parallel I/O intensity defines parallelism in a single value, I/O operation concurrency defines the distribution of concurrency over time. This is similar to the relationship between percentage of I/O type (Classification 2.4) and dominating I/O operation type (Classification 2.5). Therefore, it can be similarly argued that the visualization of I/O operation concurrency for all analysed jobs is difficult and would not contribute much to the analysis process. As a result, the I/O operation concurrency is a tool better used to further analyse a limited set of jobs. It allows for measuring the contribution of an individual process or an I/O node to a parallel I/O operation.

Being both an analysis of parallelism and of it's temporal distribution, I/O operation concurrency has the same observations when measured using the GPFS I/O logs. The I/O parallelism depth is limited by the available data which only includes I/O node I/O

request information. Additionally, the temporal distribution and by deduction the I/O operation concurrency changes when measured on different I/O stack layers and using different logging intervals.

Classification 3.3 Parallel I/O distribution

The parallel I/O distribution dissects the I/O, showing the contribution of each process. While I/O operation concurrency describes temporal distribution of parallelism, parallel I/O distribution describes the magnitude of contribution from each process. Therefore, parallel I/O distribution has similar observations to be considered as the ones made for I/O operation concurrency when using GPFS I/O logs for evaluation, for both the I/O parallelism depth is measured on an I/O node level. The parallel I/O distribution, similar to aggregate performance numbers, can change when measured in different I/O stack layers. For example, IOPs is effected by the I/O stack layer it is measured on, which remains true when defining it for individual I/O nodes.

To further the analysis, parallel I/O distribution could add the temporal distribution. In this case the parallel I/O is further dissected into it's temporal I/O distribution, which for GPFS I/O logs has a resolution of 2min, the logging interval.

Parallel I/O distribution can therefore unfold into 3 dimensions, number of process or I/O nodes; various measured performance numbers such as IOPs and finally their distribution over time. As a result the visualization of parallel I/O distribution for the large number of analysed jobs is both difficult and hardly contributes to the analysis process. It is possibly interesting to perform a parallel I/O distribution for a limited set of jobs.

Classification 3.4 Same file access concurrency

Same file access concurrency combines the temporal and spatial accessing for the sake of understanding parallelism. While labelling files as either shared or task-local defines the file access pattern, same file access concurrency describes the magnitude of a files sharing in time. Due to the need of the spatial component required for measuring same file access concurrency, it is not possible to evaluate it using the GPFS I/O logs. The missing relationship between open/close commands, I/O requests and files leads to

the missing spatial component. Same file access concurrency combines file access with parallelism while describing them in time and should therefore observe the I/O stack layer on which it is measured and the temporal resolution of the measurement.

4.5.6 Further Analysing A Subset Of Jobs

As discussed in Sec. 3.6, the ability to select and further analyse a smaller set of applications allows for visualizing I/O behaviour using various observables. Analysing a large set of applications allows for comparing them according to various I/O behaviours. Meanwhile, analysing a smaller number of applications allows to focus on specific I/O behaviours and questioning their direct impact on the I/O architecture and possible methods of improving I/O.

Selecting applications or jobs to analyse depends on the objective of analysis. A general survey of application's I/O should incorporate the analysis of jobs with commonly found I/O behaviour as analysed by the distribution of the I/O criteria. On the other hand, a focused analysis of a specific I/O behaviour, while carefully considering the I/O architecture and it's operation, requires a more cautious selection process. In such a case, jobs exhibiting a specific I/O behaviour should be selected, and the percentage of jobs exhibiting that I/O behaviour should be objectively analysed. Overall, to avoid bias in the analysis process the selected jobs should be complemented with a randomly selected group.

The purpose of the given analysis is to survey the job I/O behaviour as recorded by the GPFS I/O logs, while allowing for investigation of the I/O criteria and their applicability. However, due to the absence of clear clusters or groups of jobs with common I/O behaviour in the analysed job distributions, the selection method is simplified, using among others maximum, minimum and median values to select jobs. The given list of jobs is therefore not exhaustive, yet offers well considered examples on the analysis process. For that purpose, it becomes less necessary to fully inspect all characteristics of selected jobs, but to only consider interesting aspects as they arise.

Sec. 3.6 offers an analysis map in Tab. 3.2, that can be employed for single value I/O criteria. As seen from the process of analysing the GPFS I/O logs it is not possible to evaluate all I/O criteria. Therefore, a restructuring of the analysis map is needed.

Tab. 4.10 shows a reduced I/O criteria analysis map that can be used in relation to the GPFS I/O log analysis. While it is possible to investigate the dominating I/O repetitiveness from the GPFS I/O logs for a limited number of jobs, it has been omitted from the analysis mapping. As discussed in Sec. 4.5.4 (Classification 2.11) for analysing dominating I/O repetitiveness using the GPFS I/O logs, it is preferred to infer it from investigating the temporal I/O distribution.

			Read	Write
1	Aggregate performance numbers			
1.1	Total amount of data		S_{r}	S_{w}
1.2	Total amount of IOPs		N_{r}	N_{w}
1.3	Bandwidth	Max	$B_{\mathrm{max,r}}$	$B_{\mathrm{max,w}}$
		Avg	$\overline{B}_{\mathrm{r}}$	$\overline{B}_{\mathrm{w}}$
1.4	IOPS	Max	$\Gamma_{\mathrm{max,r}}$	$\Gamma_{\mathrm{max,w}}$
		Avg	$\overline{\Gamma}_{\mathrm{r}}$	$\overline{\Gamma}_{\mathrm{w}}$
1.5	File commands	Open	F_{open}	
		Close	F_{close}	
1.6	I/O intensity	I	I_r	I_w
2	I/O pattern analysis			
2.2	Percentage of small I/O requests		$f_{\mathrm{r}}(s_{\mathrm{small}})$	$f_{\mathrm{w}}(s_{\mathrm{small}})$
2.4	Percentage of I/O type	in IOPs	$p_{\mathrm{r,IOPs}}$	$p_{\mathrm{w,IOPs}}$
		in Bytes	$p_{\mathrm{r,Bytes}}$	$p_{\mathrm{w,Bytes}}$
2.9	Burstiness		ρ_r	ρ_w
3	Parallel I/O			
3.1	Parallel I/O intensity	$P_{\mathrm{IO},\Delta\mathrm{t}}$	$P_{\mathrm{IO},\Delta\mathrm{t}}$ for read	$P_{\mathrm{IO},\Delta\mathrm{t}}$ for write

TABLE 4.10: Analysis map of I/O criteria for analysing GPFS I/O logs.

The I/O criteria analysis for I/O intensity (Classification 1.6), burstiness (Classification 2.9) and parallel I/O intensity (Classification 3.1) is performed using many thresholds c. It is possible to assume that any modern HPC I/O subsystem can cope well with jobs generating I/O requests of less than 1MiB/s, making c =1MiB. The high value for threshold observes the size of the I/O subsystem serving JUGENE. For a system with a bandwidth of 66GiB/s, 1MiB/s is relatively small. As the target is to find jobs that strain the I/O subsystem and might require new I/O architectures, it is reasonable to select a threshold that reflects such a job. Furthermore, lower I/O rates might be due to

writing continuously short status messages at low frequency. Tacking such I/O activities into account could lead to wrong conclusions, e.g. jobs might appear to be I/O intensive, while aggregate amount of data read/written is small.

As previously mentioned, having a limited set of jobs allows for analysing I/O distributions over time, request sizes or processes. Therefore, to provide further insight into jobs I/O behaviour, the selected jobs I/O distributions are visualized including dominating I/O operation type (Classification 2.5); temporal distribution of I/O (Classification 2.8); I/O operation concurrency (Classification 3.2) and parallel I/O distribution (Classification 3.3). Here too when thresholds are needed the value is set to $c = 1\text{MiB}$, with the exception of studying read or write commands where the threshold is set to $c = 0$ commands. Distribution of request sizes (Classification 2.1) is analysed when necessary.

The following sections re-list a subset of the I/O criteria and uses some of these to select a list of jobs for further analysis. Specifically jobs are selected using total amount of data read and written (Classification 1.1); total number of IOPs (Classification 1.2); read/write maximum bandwidth (Classification 1.3); read/write maximum IOPS (Classification 1.4) and I/O intensity (Classification 1.6). In addition to that, for some of the selected jobs comments are made on the distribution of request sizes (Classification 2.1); the percentage of small I/O requests (Classification 2.2); burstiness (Classification 2.9) and parallel I/O intensity (Classification 3.1).

To complement the analysis of the selected jobs, possible benefits from different I/O architectures are mentioned. Occasionally comments are made on the applicability of burst buffers and readahead or prefetching techniques. These function as a demonstration for considering I/O architectures in relation to job I/O analysis. As an example, reasonable size of burst buffers are questioned as seen from the burst behaviour of jobs. However, these examples are simplified and suggested improvements could depend on many additional parameters. For instance, the performance enhancement from using burst buffers assumes a larger bandwidth to the burst buffer than available to the storage system. Meanwhile, the readahead or prefetching performance improvement assumes the ability for jobs to recognize the needed data in advance. Fully judging and investigating the applicability of different I/O architectures and their positive and negative impact on the I/O behaviour would require detailed analysis of a larger set of jobs, including the application's algorithm and internal behaviour.

4.5.7 Analysing Jobs Using Category 1: Aggregate Performance Numbers

Classification 1.1 **Total amount of data read/written**

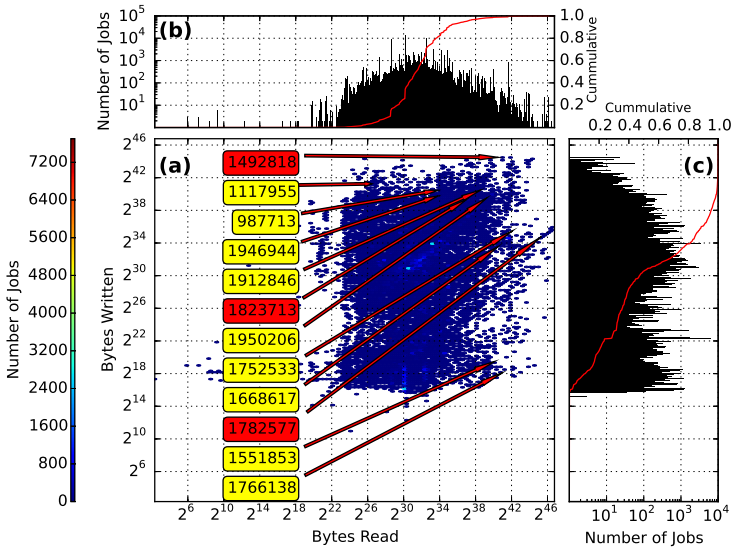


FIGURE 4.35: Bytes read and written for selected jobs. (a) Scatter plot of bytes read and written with a heat map for job count, (b) Histogram of bytes read and (c) Histogram of bytes written.

Using bytes read and written 3 jobs are selected. Job 1782577; has the maximum of bytes read, job 1492818; has the maximum of bytes written and job 1823713; is the bytes written median of jobs with over 1TiB read or write. The numbers indexing the jobs are called `JOBID` and have been discussed in Sec. 4.3. The median[10] selection is a method of quasi random selecting a job and has been done for jobs with at least 1TiB of read or write to observe the size of the I/O subsystem serving JUGENE. For PiB of storage and a bandwidth of 66GiB/s a job with I/O of less than 1TiB and a runtime of longer than 1hour can be considered in I/O terms relatively small.

To solidify the understanding of the connection between analysing and comparing a large batch of jobs versus analysing a smaller subset, it is helpful to see the position of the selected jobs within the overall distribution. Fig. 4.35 shows all selected jobs and their

[10]The median is the job at the middle of the sorted evaluated I/O criteria. Although not fully selected at random, the job could exhibit any I/O behaviour.

position within the distribution of read and written bytes. The 3 jobs selected based on bytes read and written are highlighted in red. The remaining jobs are selected using other evaluated I/O criteria.

Job 1782577 Maximum of total bytes read

Duration[s]	86389
I/O node count	8
Compute node count	1024

TABLE 4.11: Info of job 1782577; maximum of total bytes read.

1	Aggregate performance numbers			Read	Write
1.1	Total amount of data			109.543 TiB	63.067 GiB
1.2	Total amount of IOPs			1.38×10^8	1.99×10^7
1.3	Bandwidth	Max		1.624 GiB/s	5.946 MiB/s
		Avg		1.298 GiB/s	765.501 KiB/s
1.4	IOPS	Max		3982	13806
		Avg		1597.17	230.49
1.5	File commands	Open		6.74×10^7	
		Close		6.74×10^7	
1.6	I/O intensity	$c = 1$MiB	1.00	1.00	0.00
2	I/O pattern analysis				
2.2	Percentage of small I/O requests ($s \leq 4$KiB)			0.01	0.98
2.4	Percentage of I/O type	in IOPs		0.87	0.13
		in Bytes		1.00	0.00
2.9	Burstiness	$c = 1$MiB		0.00	1.00
3	Parallel I/O				
3.1	Parallel I/O intensity	$c = 1$MiB	0.99	0.99	0.00

TABLE 4.12: I/O criteria analysis map of job 1782577; maximum of total bytes read.

Job 1782577 measured the highest amount of bytes read at 109TiB. Tab. 4.11 introduces the size of job 1782577 in both time and number of both I/O nodes and compute nodes. Evaluating I/O criteria is better understood in terms of the job's size. Tab. 4.12 presents the evaluated I/O criteria map for job 1782577; maximum of total bytes read. As seen, the job has performed a large amount of read to a relatively small write. The mismatch of read to write manifests itself in almost all I/O criteria.

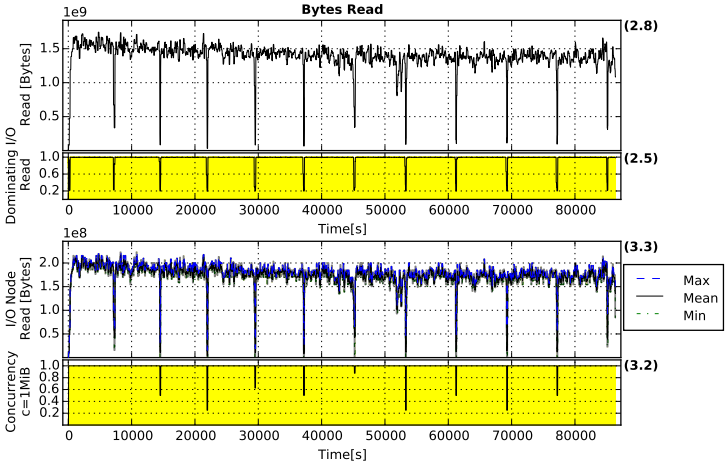


FIGURE 4.36: Bytes read for job 1782577; maximum of total bytes read.

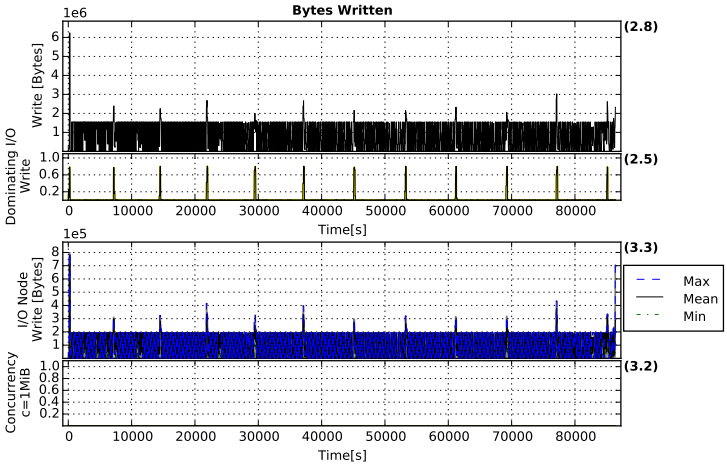


FIGURE 4.37: Bytes written for job 1782577; maximum of total bytes read.

Fig. 4.36 presents the I/O distributions of job 1782577. The plots depict, from top to bottom, temporal distribution of bytes read (Classification 2.8); dominating I/O type for bytes read (Classification 2.5) or percentage of bytes read over time; parallel bytes read distribution (Classification 3.3) and bytes read concurrency (Classification 3.2). For reference the number of the classification is repeated besides the I/O distribution. While all plots in Fig. 4.36 have a single value at any point in time, Fig. 4.36-(3.3) for parallel I/O distribution could have 3. The plot depicts the maximum, minimum and mean value of temporal I/O distribution of the I/O nodes belonging to the analysed job. From this it is possible to infer the range of values between the I/O nodes and as a

result the parallel I/O distribution. The method given in Fig. 4.36 is a compact method to evaluate many of the I/O distributions from the I/O criteria and is therefore used to analyse the selected jobs.

Fig. 4.37 presents the bytes written distributions. When analysed in relation to the bytes read in Fig. 4.36, job 1782577 appears to be operating with a repetitive cycle. The job almost continuously reads with the exception of some periodic dips of read amount which seem to coincide with slight increase in bytes written. Such job behaviour could benefit from an increase in available read bandwidth. In general any improvement on the read path can increase performance. As read appears to be continues a prefetching or readahead mechanism could require relatively large buffers.

Job 1492818 Maximum of total bytes written

Duration[s]	86032
I/O node count	64
Compute node count	8192

TABLE 4.13: Info of job 1492818; maximum of total bytes written.

Job 1492818 measured the highest amount of bytes written at over 22TiB. Tab. 4.13 introduces the job's information, while Tab. 4.14 evaluates the I/O criteria map for the job. The job appears to have a less mismatch between read and write compared to job 1782577. Job 1492818 is a larger job occupying 11% of JUGENE. Meanwhile, the maximum read job 1782577 occupied only an eighth of that size.

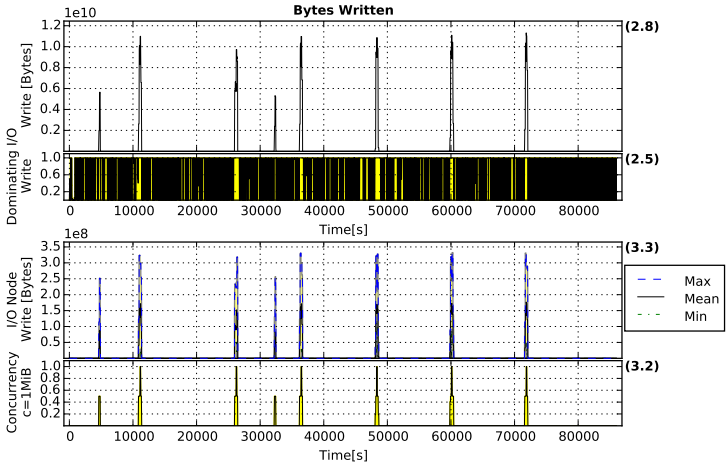


FIGURE 4.38: Bytes written for job 1492818; maximum of total bytes written.

1	Aggregate performance numbers			Read	Write
1.1	Total amount of data			3.307 TiB	22.259 TiB
1.2	Total amount of IOPs			9.12×10^5	6.47×10^6
1.3	Bandwidth	Max		14.172 GiB/s	10.537 GiB/s
		Avg		40.304 MiB/s	271.297 MiB/s
1.4	IOPS	Max		3783	2766
		Avg		10.60	75.21
1.5	File commands	Open		4.82×10^5	
		Close		3.08×10^5	
1.6	I/O intensity	$c = 1$MiB	0.05	0.00	0.04
2	I/O pattern analysis				
2.2	Percentage of small I/O requests ($s \leq 4$KiB)			0.00	0.05
2.4	Percentage of I/O type	in IOPs		0.12	0.88
		in Bytes		0.13	0.87
2.9	Burstiness	$c = 1$MiB		0.99	0.95
3	Parallel I/O				
3.1	Parallel I/O intensity	$c = 1$MiB	0.55	0.75	0.53

TABLE 4.14: I/O criteria analysis map of job 1492818; maximum of total bytes written.

Fig. 4.38 presents the bytes written distributions for job 1492818. Write appears in bursts and could be considered almost periodic. Given the time between bursts it is possible to perform burst buffering, which can also be understood from the value of burstiness given in Tab. 4.14. The buffer has then sufficient time to move the data out to the storage system. As a result the burst buffers might reduce the time spent in I/O significantly. However, when considering a write I/O intensity of only 0.04, the burst buffers would hardly improve performance of the overall job.

Job 1823713 Bytes written median of jobs with over 1TiB read or write

Duration[s]	17993
I/O node count	32
Compute node count	4096

TABLE 4.15: Info of job 1823713; bytes written median of jobs with over 1TiB read or write.

1	Aggregate performance numbers			Read	Write
1.1	Total amount of data			256.125 GiB	1.0 TiB
1.2	Total amount of IOPs			1.49×10^5	6.04×10^5
1.3	Bandwidth	Max		3.152 GiB/s	1.846 GiB/s
		Avg		14.576 MiB/s	58.276 MiB/s
1.4	IOPS	Max		1842	1029
		Avg		8.26	33.58
1.5	File commands	Open		8.20×10^4	
		Close		8.21×10^4	
1.6	I/O intensity	$c = 1$MiB	0.07	0.00	0.06
2	I/O pattern analysis				
2.2	Percentage of small I/O requests ($s \leq 4$KiB)			0.00	0.02
2.4	Percentage of I/O type	in IOPs		0.20	0.80
		in Bytes		0.20	0.80
2.9	Burstiness	$c = 1$MiB		1.00	0.93
3	Parallel I/O				
3.1	Parallel I/O intensity	$c = 1$MiB	0.85	0.92	0.84

TABLE 4.16: I/O criteria analysis map of job 1823713; bytes written median of jobs with over 1TiB read or write.

Job 1823713 is selected being the median of bytes written. The selection method is simple and provides a job almost at random that does not have as large I/O as the maximums previously presented. Tab. 4.15 provides job 1823713 information and indicates that the job's duration is less than for the two maximum selected jobs. On the other hand, Tab. 4.16 evaluates the job's I/O criteria and indicates a larger write than read by about 4 times.

Similar to job 1492818; maximum of total bytes written, job 1823713 exhibits bursty write as seen in Fig. 4.39 and evaluated by the burstiness parameter in Tab. 4.16. As a result similar benefits can be achieved for using burst buffers and at an I/O intensity of 0.06 job 1823713 could achieve a slight improvement. Nonetheless, the burst buffers would only minimally effect the overall job performance.

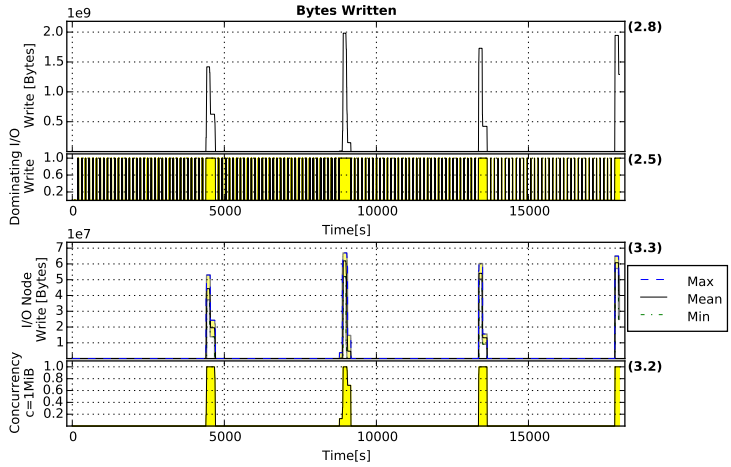


FIGURE 4.39: Bytes written for job 1823713; bytes written median of jobs with over 1TiB read or write.

Classification 1.2 Total number of IOPs

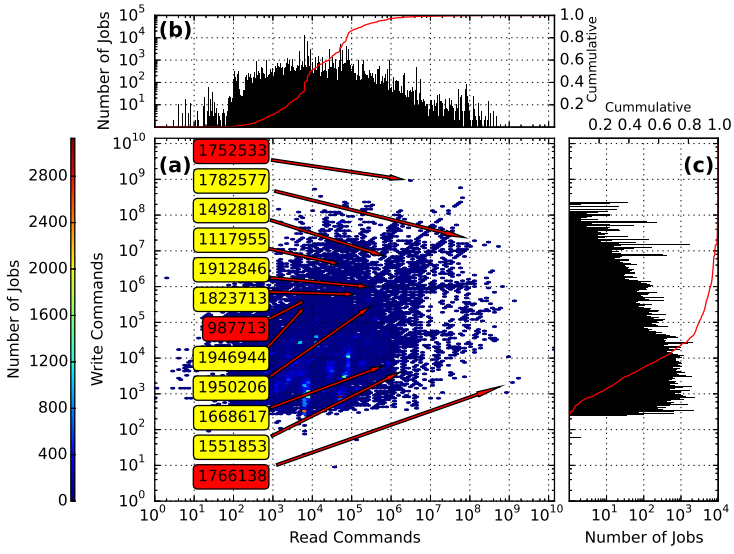


FIGURE 4.40: Read and write commands for selected jobs. (a) Scatter plot of read and write commands with a heat map for job count, (b) Histogram of read commands and (c) Histogram of write commands.

Using total number of IOPs 3 jobs are selected. Job 1766138; has the maximum of total read commands, job 1752533; has the maximum of total write commands and

job 987713; is the write commands median of jobs with over 1TiB read or write. The selected jobs are shown in the distribution of read and write commands for jobs in Fig. 4.40.

Job 1766138 Maximum of total read commands

Duration[s]	3878
I/O node count	32
Compute node count	4096

TABLE 4.17: Info of job 1766138; maximum of total read commands.

1	Aggregate performance numbers			Read	Write
1.1	Total amount of data			4.575 TiB	444.211 KiB
1.2	Total amount of IOPs			1.22×10^9	2.12×10^3
1.3	Bandwidth	Max		1.579 GiB/s	749.0B/s
		Avg		1.208 GiB/s	117.268B/s
1.4	IOPS	Max		332364	32
		Avg		315707.05	0.55
1.5	File commands	Open		2.87×10^4	
		Close		2.87×10^4	
1.6	I/O intensity	$c = 1$MiB	1.00	1.00	0.00
2	I/O pattern analysis				
2.2	Percentage of small I/O requests ($s \leq 4$KiB)			0.43	1.00
2.4	Percentage of I/O type	in IOPs		1.00	0.00
		in Bytes		1.00	0.00
2.9	Burstiness	$c = 1$MiB		0.00	1.00
3	Parallel I/O				
3.1	Parallel I/O intensity	$c = 1$MiB	1.00	1.00	0.00

TABLE 4.18: I/O criteria analysis map of job 1766138; maximum of total read commands.

Job 1766138 created the largest number of read commands at about 1.22×10^9 commands. The job's information is given in Tab. 4.17. The evaluation of the job's I/O criteria in Tab. 4.18, shows the large amount of read the job has performed compared to a relatively very small write.

Fig. 4.41 depicts the I/O distributions of read commands for job 1766138. As seen the job produces almost continuously read requests and is therefore similar to job 1782577;

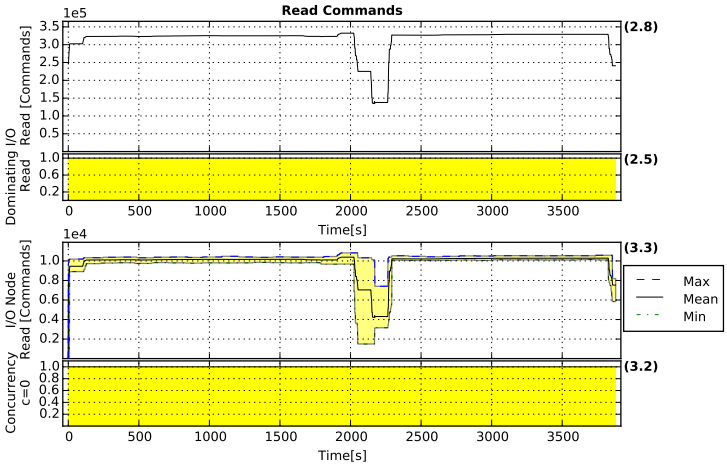


FIGURE 4.41: Read commands for job 1766138; maximum of total read commands.

maximum of total bytes read. Both the maximum read bytes job and maximum read command job appear to achieve similar read bandwidth when comparing Tab. 4.12 and Tab. 4.18. Therefore, it could be assumed that no improvement could be made to job 1766138; maximum of total read commands, by increasing the read request sizes and as a result decreasing the number of read commands. However, considering that job 1782577; maximum of total bytes read, achieved this read bandwidth using 8 I/O nodes, while job 1766138; maximum of total read commands, achieved the same bandwidth using 32 I/O nodes, it shows that with the larger number of read requests a smaller read bandwidth is achieved for the same I/O quantity.

For job 1766138; maximum of total read commands, prefetching or readahead could require large buffers. However, a prefetching mechanism can attempt merging small read requests to improve performance. The job could also benefit from an increase in available read IOPS or read bandwidth. Generally any improvement on the read path, specially in terms of dealing with a large quantity of read requests, could lead to an improvement in the job's I/O performance. Such improvement could have a large impact on the overall job's performance given the measured I/O intensity of 1.0.

Job 1752533 Maximum of total write commands

Duration[s]	30251
I/O node count	64
Compute node count	8192

TABLE 4.19: Info of job 1752533; maximum of total write commands.

1	Aggregate performance numbers			Read	Write
1.1	Total amount of data			10.575 TiB	82.16 GiB
1.2	Total amount of IOPs			3.12×10^6	9.25×10^8
1.3	Bandwidth	Max		36.481 GiB/s	40.282 MiB/s
		Avg		366.556 MiB/s	2.781 MiB/s
1.4	IOPS	Max		9346	470733
		Avg		103.00	30568.18
1.5	File commands	Open		1.04×10^7	
		Close		1.03×10^7	
1.6	I/O intensity	$c = 1$MiB	0.19	0.17	0.02
2	I/O pattern analysis				
2.2	Percentage of small I/O requests ($s \leq 4$KiB)			0.00	1.00
2.4	Percentage of I/O type	in IOPs		0.00	1.00
		in Bytes		0.99	0.01
2.9	Burstiness	$c = 1$MiB		0.80	0.97
3	Parallel I/O				
3.1	Parallel I/O intensity	$c = 1$MiB	0.12	0.13	0.01

TABLE 4.20: I/O criteria analysis map of job 1752533; maximum of total write commands.

Job 1752533 created the largest number of write commands at about 9.25×10^8 commands. The job's information is given in Tab 4.19. Tab. 4.20 evaluates job 1752533 I/O criteria. Despite the large number of write commands performed by the job, a relatively small amount of bytes were written. As a result the write requests are rather small as seen in Tab. 4.20 from percentage of small I/O requests. This could also be resulting in a reduced bandwidth as the job becomes limited by the available I/O system IOPS.

To analyse job 1752533; maximum of total write commands, Fig. 4.42 presents the read command distributions, while Fig. 4.43 presents the write command distributions. As seen from the two figures, it appears job 1752533 starts with read interleaved with write,

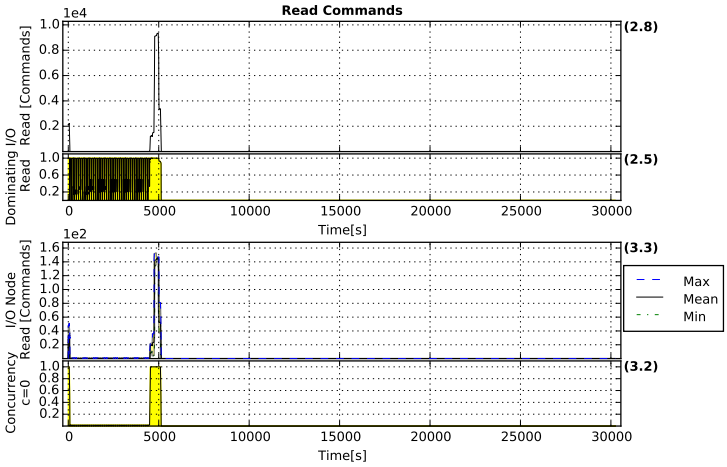


FIGURE 4.42: Read commands for job 1752533; maximum of total write commands.

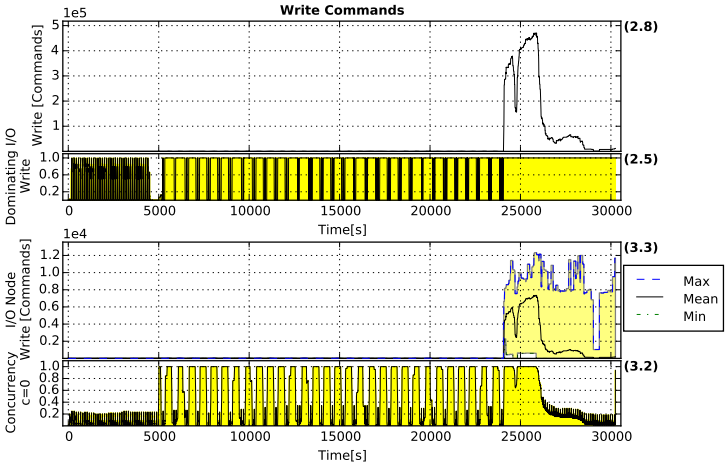


FIGURE 4.43: Write commands for job 1752533; maximum of total write commands.

moves into a large read burst and ends with a write burst. Between the two bursts write appears to be small but continuous. Furthermore, the final write burst is not shared by all I/O nodes, as given by Fig. 4.43-(3.3).

It is possible to imagine that job 1752533; maximum of total write commands, can benefit from a larger write IOPS rate on the I/O subsystem. It is also possible to achieve an improvement by using buffers with a combination of merging small write requests. Such collective I/O techniques requires a spatial access pattern that allows I/O requests to be combined.

Job 987713 Write commands median of jobs with over 1TiB read or write

Duration[s]	82349
I/O node count	8
Compute node count	1024

TABLE 4.21: Info of job 987713; write commands median of jobs with over 1TiB read or write.

1	Aggregate performance numbers			Read	Write
1.1	Total amount of data			25.562 GiB	1.526 TiB
1.2	Total amount of IOPs			6.88×10^3	4.20×10^5
1.3	Bandwidth	Max		415.136 MiB/s	303.799 MiB/s
		Avg		325.49 KiB/s	19.431 MiB/s
1.4	IOPS	Max		112	80
		Avg		0.08	5.10
1.5	File commands	Open		1.25×10^4	
		Close		9.48×10^3	
1.6	I/O intensity	$c = 1$MiB	0.23	0.00	0.23
2	I/O pattern analysis				
2.2	Percentage of small I/O requests ($s \leq 4$KiB)			0.00	0.03
2.4	Percentage of I/O type	in IOPs		0.02	0.98
		in Bytes		0.02	0.98
2.9	Burstiness	$c = 1$MiB		1.00	0.71
3	Parallel I/O				
3.1	Parallel I/O intensity	$c = 1$MiB	0.00	0.00	0.00

TABLE 4.22: I/O criteria analysis map of job 987713; write commands median of jobs with over 1TiB read or write.

To complement the I/O distributions studied of the largest 2 jobs in terms of read and write commands, job 987713; write commands median of jobs with over 1TiB read or write, is selected. Tab. 4.21 shows the job's information. The evaluated I/O criteria for job 987713, given in Tab 4.22, suggest a job with a larger write than read I/O.

The I/O distribution of write commands for job 987713, given in Fig. 4.44 and the burstiness parameter seems to suggest a bursty write access, with almost no parallel I/O. It might therefore benefit from introducing burst buffers, which at a write intensity

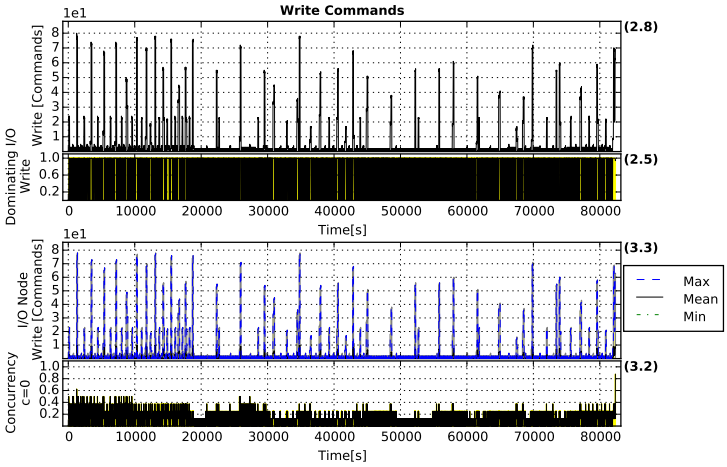


FIGURE 4.44: Write commands for job 987713; write commands median of jobs with over 1TiB read or write.

of 0.23 might add some improvement to the job's performance. Additionally, job 987713 appears to move through two I/O stages, in the first stage the write access bursts are more frequent, while in the second stage the bursts period increases.

Classification 1.3 Read/Write bandwidth

One reason for having the bandwidth as an I/O criteria is finding jobs that are limited at the maximum read or write bandwidth available by the I/O subsystem. However, as discussed in Sec. 4.5.3 (Classification 1.3), the maximum bandwidth has some inaccuracies when analysed using a coarse temporal resolution. As a result, selecting the job that performed the highest bandwidth might not be as effective in understanding the job I/O behaviour. For the purpose of surveying the I/O behaviour of analysed jobs 2 jobs are selected. Job 1912846; having equal read and write bandwidth and job 1668617; is the read bandwidth median of jobs with over 1TiB read or write. The term bandwidth here refers to the maximum bandwidth as measured by the GPFS I/O logs. Fig. 4.45 shows the selected jobs location in the distribution of read and write bandwidth over the analysed jobs.

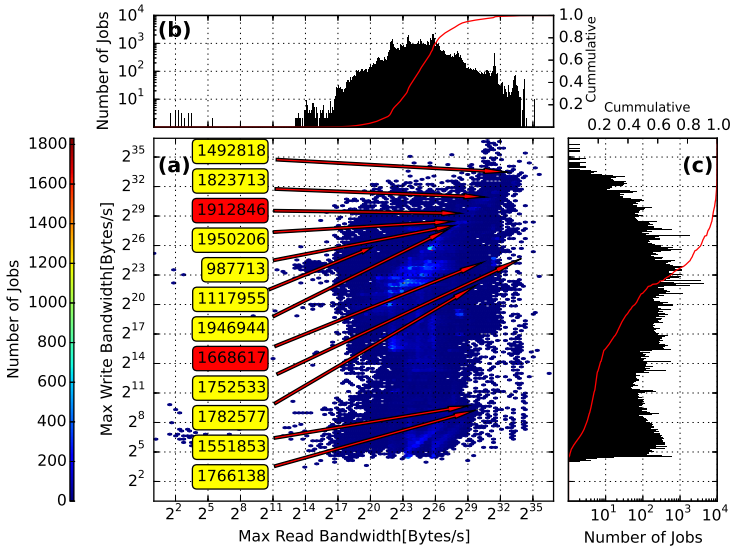


FIGURE 4.45: Read and write maximum bandwidth for selected jobs. (a) Scatter plot of read and write maximum bandwidth with a heat map for job count, (b) Histogram of read maximum bandwidth and (c) Histogram of write maximum bandwidth.

Job 1912846 Equal read and write bandwidth

Duration[s]	14203
I/O node count	16
Compute node count	2048

TABLE 4.23: Info of job 1912846; equal read and write bandwidth.

Tab. 4.23 gives the information of job 1912846 which achieved an equal read and write maximum bandwidth. Despite that the job appears to have performed a larger amount of write than read as described by Tab. 4.24, which evaluates the I/O criteria for job 1912846.

Fig. 4.46 and Fig. 4.47 depict an interesting I/O sequence for job 1912846. The job appears to start with a small read burst, moves in a stage of many write bursts with a complete absence of read. The job then proceeds to perform periodic write bursts interleaved with periodic read bursts. While this behaviour might not benefit much from an increase in bandwidth, it could benefit from burst buffers and prefetching or readahead. With maybe the exception of the first write bursts, the timing and period after each write burst could allow a burst buffer to trickle the data to the storage system

1	Aggregate performance numbers			Read	Write
1.1	Total amount of data			810.044 GiB	2.039 TiB
1.2	Total amount of IOPs			3.65×10^5	9.54×10^5
1.3	Bandwidth	Max		614.409 MiB/s	614.409 MiB/s
		Avg		58.402 MiB/s	150.54 MiB/s
1.4	IOPS	Max		276	284
		Avg		25.73	67.19
1.5	File commands	Open		5.65×10^5	
		Close		3.28×10^5	
1.6	I/O intensity	$c = 1$MiB	0.48	0.19	0.41
2	I/O pattern analysis				
2.2	Percentage of small I/O requests ($s \leq 4$KiB)			0.00	0.00
2.4	Percentage of I/O type	in IOPs		0.28	0.72
		in Bytes		0.28	0.72
2.9	Burstiness	$c = 1$MiB		0.76	0.37
3	Parallel I/O				
3.1	Parallel I/O intensity	$c = 1$MiB	0.68	0.54	0.68

TABLE 4.24: I/O criteria analysis map of job 1912846; equal read and write bandwidth.

while job I/O is idling. Meanwhile, the period of read bursts could allow a readahead mechanism to slowly read the needed data from the storage into a buffer before it is needed by the job. At an I/O intensity of 0.48 the burst buffering and readahead could have a beneficial impact on job 1912846.

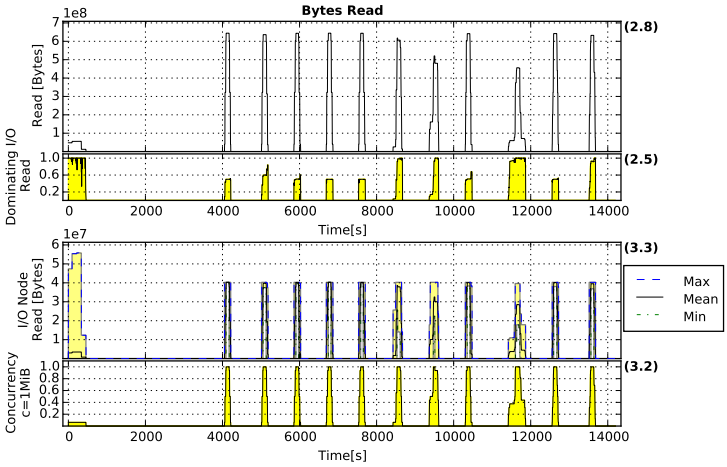


FIGURE 4.46: Bytes read for job 1912846; equal read and write bandwidth.

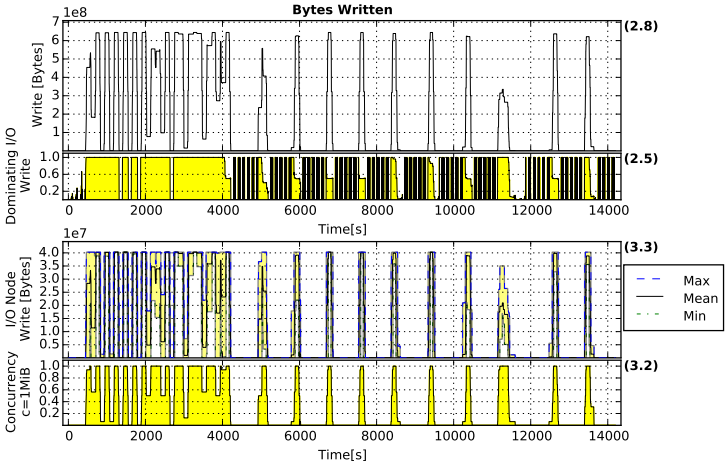


FIGURE 4.47: Bytes written for job 1912846; equal read and write bandwidth.

Job 166861 Read bandwidth median of jobs with over 1TiB read or write

Duration[s]	4754
I/O node count	8
Compute node count	1024

TABLE 4.25: Info of job 1668617; read bandwidth median of jobs with over 1TiB read or write.

Job 1668617 is selected as the read bandwidth median of jobs with over 1TiB read or write. The job's information are presented in Tab. 4.25 and the I/O criteria of the job

1	Aggregate performance numbers			Read	Write
1.1	Total amount of data			3.253 TiB	22.781 GiB
1.2	Total amount of IOPs			8.80×10^5	6.95×10^3
1.3	Bandwidth	Max		2.582 GiB/s	26.75 MiB/s
		Avg		717.379 MiB/s	4.906 MiB/s
1.4	IOPS	Max		906	15
		Avg		185.09	1.46
1.5	File commands	Open		1.07×10^5	
		Close		1.06×10^5	
1.6	I/O intensity	$c = 1$MiB	0.76	0.63	0.41
2	I/O pattern analysis				
2.2	Percentage of small I/O requests ($s \leq 4$KiB)			0.00	0.14
2.4	Percentage of I/O type	in IOPs		0.99	0.01
		in Bytes		0.99	0.01
2.9	Burstiness	$c = 1$MiB		0.06	0.39
3	Parallel I/O				
3.1	Parallel I/O intensity	$c = 1$MiB	0.77	0.92	0.00

TABLE 4.26: I/O criteria analysis map of job 1668617; read bandwidth median of jobs with over 1TiB read or write.

are evaluated in Tab. 4.26. It appears that job 1668617 performed a far larger amount of read than write as reflected by the I/O criteria. Nonetheless the write I/O intensity is at 0.41 which is not far behind the read I/O intensity of 0.63.

Fig. 4.48 describes the bytes read I/O distributions of job 1668617. The job appears to exhibit wide step bursts. When examining Fig. 4.48-(3.3) and Fig. 4.48-(3.2) it is possible to see that all I/O nodes experience these step bursts in parallel.

Fig. 4.49 depicts the bytes written I/O distributions of job 1668617. When compared to Fig. 4.48 it is possible to see an overall I/O behaviour of read and write. While performing read, some I/O nodes additionally perform write which occasionally coincides with the lower of the two steps of the read bursts. This behaviour can be observed from Fig. 4.48-(2.5) where the read I/O domination occasionally falls below 1.0. The opposite is also true, where write performance drops coincide with the occurrence of a read burst and can be observed from Fig. 4.49-(2.5). However, from the analysis of the GPFS I/O

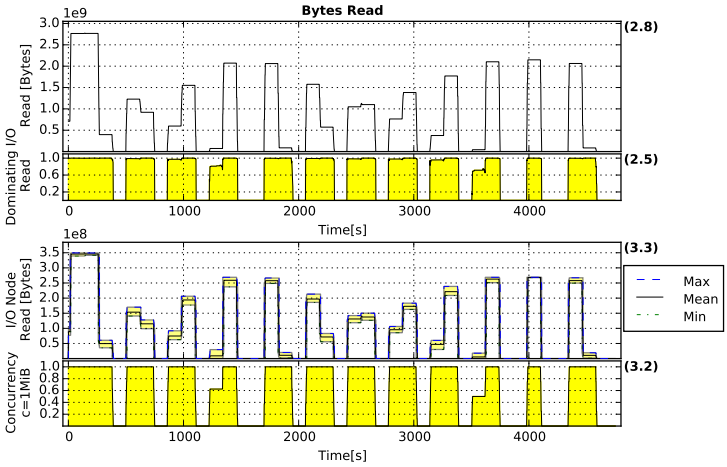


FIGURE 4.48: Bytes read for job 1668617; read bandwidth median of jobs with over 1TiB read or write.

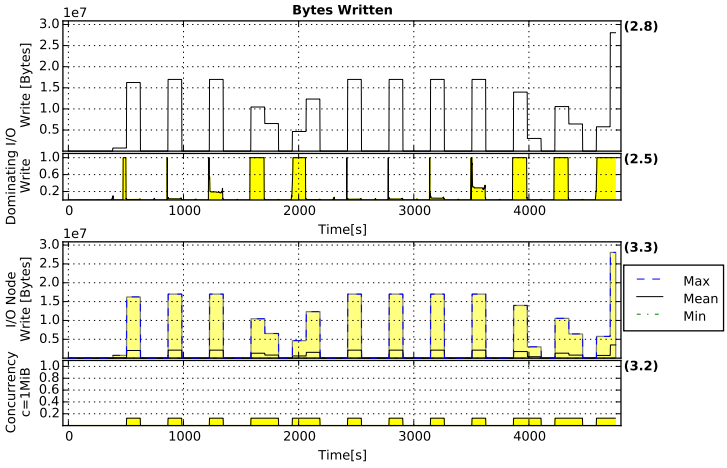


FIGURE 4.49: Bytes written for job 1668617; read bandwidth median of jobs with over 1TiB read or write.

logs it is not clear whether this decrease of read performance coinciding with write and vice versa is an application design or due to the I/O subsystem. Small I/O systems such as accessing an HDD could exhibit a drop in read and write bandwidth when performing simultaneous read and write.

For job 1668617, prefetch or readahead mechanism could benefit the read performance. however the implementation might require the use of relatively large buffers to hold the prefetched data. The size of the required buffers depends on the size of the bursts and

the time between bursts. Write might also benefit from a burst buffer. At a fairly high I/O intensity of 0.76 these I/O architectural changes might have a high impact on the job's performance.

Classification 1.4 Read/Write IOPS

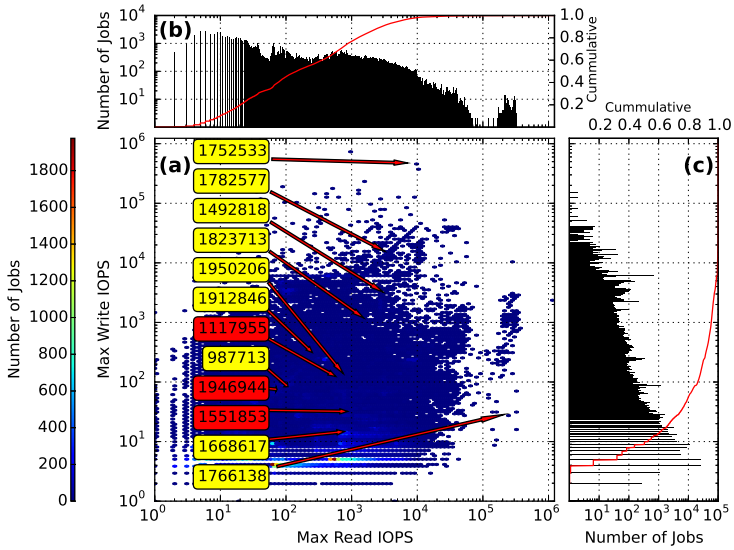


FIGURE 4.50: Read and write maximum IOPS for selected jobs. (a) Scatter plot of read and write maximum IOPS with a heat map for job count, (b) Histogram of read maximum IOPS and (c) Histogram of write maximum IOPS.

The analysis of job IOPS is similar to analysing bandwidth. As discussed in Sec. 4.5.3 (Classification 1.4), the maximum IOPS has some inaccuracies when analysed with a coarse temporal resolution. Therefore the study of jobs with highest maximum IOPS might not give correct conclusions on the I/O behaviour of analysed jobs. Using IOPS analysis of GPFS I/O logs 3 jobs are selected to survey the I/O behaviour. Job 1946944; having equal read and write IOPS, job 1551853; is the read IOPS median of jobs with over 1TiB read or write and job 1117955; is the write IOPS median of jobs with over 1TiB read or write. IOPS here refers to maximum achieved IOPS as measured using the GPFS I/O logs. Fig. 4.50 shows where these jobs are located in the distribution of maximum read and write IOPS over jobs. The IOPS measured are averages over 2min, the GPFS I/O counter logging interval, which should be considered when observing relatively low IOPS rates.

Job 1946944 Equal read and write IOPS

Duration[s]	67159
I/O node count	8
Compute node count	1024

TABLE 4.27: Info of job 1946944; equal read and write IOPS.

1	Aggregate performance numbers			Read	Write
1.1	Total amount of data			25.543 GiB	1.048 TiB
1.2	Total amount of IOPs			6.82×10^3	2.90×10^5
1.3	Bandwidth	Max		264.184 MiB/s	282.133 MiB/s
		Avg		398.816 KiB/s	16.367 MiB/s
1.4	IOPS	Max		75	75
		Avg		0.10	4.32
1.5	File commands	Open		8.35×10^3	
		Close		6.42×10^3	
1.6	I/O intensity	$c = 1$MiB	0.37	0.00	0.37
2	I/O pattern analysis				
2.2	Percentage of small I/O requests ($s \leq 4$KiB)			0.00	0.03
2.4	Percentage of I/O type	in IOPs		0.02	0.98
		in Bytes		0.02	0.98
2.9	Burstiness	$c = 1$MiB		1.00	0.48
3	Parallel I/O				
3.1	Parallel I/O intensity	$c = 1$MiB	0.00	0.00	0.00

TABLE 4.28: I/O criteria analysis map of job 1946944; equal read and write IOPS.

Tab. 4.27 lists job 1946944 information, while Tab. 4.28 shows it's evaluated I/O criteria. Job 1946944 performed an equal maximum IOPS for both read and write. Despite that the job appears uneven, having performed mostly write. This suggests that the maximum achieved IOPS is not directly related to the number of I/O requests a job needs to perform.

Due to the large write performed by job 1946944, the write command's distributions are shown in Fig. 4.51. As seen from the figure the write commands are continuous with interleaved bursts. These appear to be periodic, with the period decreasing over the jobs runtime. Additionally, the write bursts appear to come in different magnitudes, with the larger bursts being followed by smaller ones. Such periodic burst behaviour allows

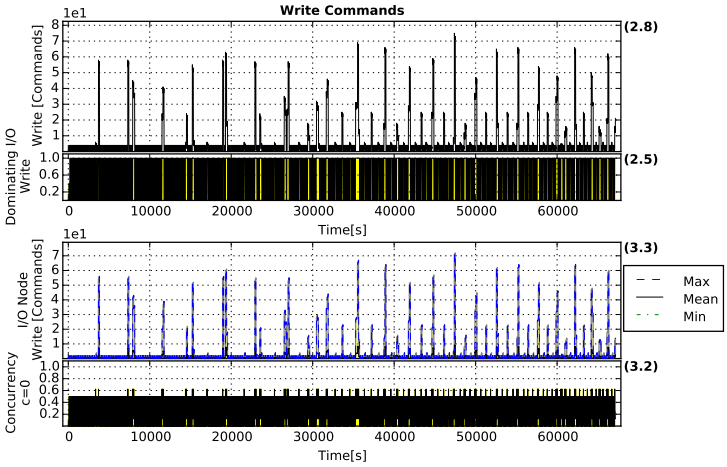


FIGURE 4.51: Write commands for job 1946944; equal read and write IOPS.

for better burst buffer design. Using burst size and period, buffer size and bandwidth between buffer and storage can be chosen.

Job 1551853 Read IOPS median of jobs with over 1TiB read or write

Duration[s]	3690
I/O node count	32
Compute node count	4096

TABLE 4.29: Info of job 1551853; read IOPS median of jobs with over 1TiB read or write.

Job 1551853 is the read IOPS median of jobs with over 1TiB read or write. The job's information is given in Tab. 4.29 and it's evaluated I/O criteria in Tab. 4.30. Job 1551853 appears to have performed a large amount of read compared to a very small amount of write.

The read behaviour of job 1551853 is described in the read commands distributions shown in Fig. 4.52. The figure indicates continuous read throughout the job's runtime. However, as seen from Fig. 4.52-(3.3) and Fig. 4.52-(3.2), the read lacks in parallelism. Out of the 32 I/O nodes occupied by the job, only a few appear to be involved in the read operations at any given time. If allowed by the job's algorithm, distributing the read over the available I/O nodes might help utilizing more bandwidth. Another method of achieving better I/O node parallelism for the job, is for the I/O nodes to redistribute

1	Aggregate performance numbers			Read	Write
1.1	Total amount of data			2.472 TiB	1.05 MiB
1.2	Total amount of IOPs			2.60×10^6	5.12×10^3
1.3	Bandwidth	Max		1021.766 MiB/s	935.0B/s
		Avg		702.163 MiB/s	298.433B/s
1.4	IOPS	Max		1019	32
		Avg		703.27	1.39
1.5	File commands	Open		7.55×10^3	
		Close		7.55×10^3	
1.6	I/O intensity	$c = 1$MiB	1.00	1.00	0.00
2	I/O pattern analysis				
2.2	Percentage of small I/O requests ($s \leq 4$KiB)			0.00	1.00
2.4	Percentage of I/O type	in IOPs		1.00	0.00
		in Bytes		1.00	0.00
2.9	Burstiness	$c = 1$MiB		0.00	1.00
3	Parallel I/O				
3.1	Parallel I/O intensity	$c = 1$MiB	0.16	0.16	0.00

TABLE 4.30: I/O criteria analysis map of job 1551853; read IOPS median of jobs with over 1TiB read or write.

some of the I/O load among themselves. However, such design must avoid using too much of the internal available network for the I/O nodes to bring the data to the process that initiated the read requests.

Given that job 1551853 continuously performs read, a readahead mechanism would only interfere with the ongoing read operations. A better implementation would attempt initiating prefetch prior to the start of the job. Such a design requires large buffers that allows for keeping the data long enough till it is needed by the job. For such applications understanding the data read could allow for better I/O architectures. Since the GPFS I/O logs do not allow identifying I/O requests, some data might be reread several times. In such a case a buffering the data or using active storage that brings the processing closer to the data, might allow for better I/O performance. Given job 1551853 high I/O intensity of 1.0, this might merit a deeper analysis of the application's I/O.

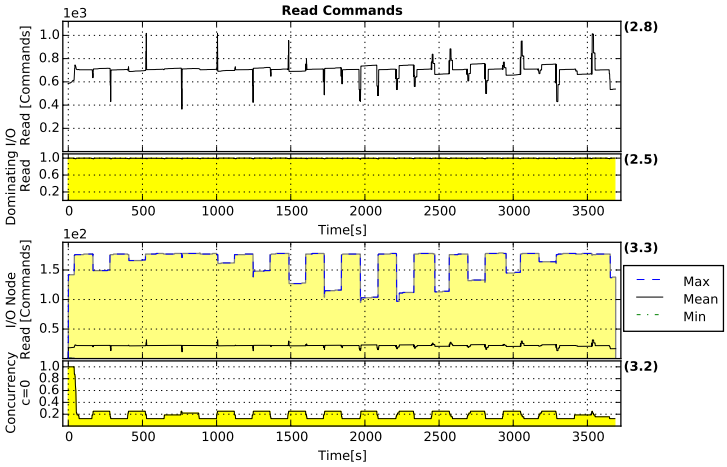


FIGURE 4.52: Read commands for job 1551853; read IOPS median of jobs with over 1TiB read or write.

Job 1117955 Write IOPS median of jobs with over 1TiB read or write

Duration[s]	69342
I/O node count	16
Compute node count	2048

TABLE 4.31: Info of job 1117955; write IOPS median of jobs with over 1TiB read or write.

Job 1117955 is the write IOPS median of jobs with over 1TiB read or write. The job's information is given in Tab. 4.31 and it's evaluated I/O criteria in Tab. 4.32. Similar to many of the so far analysed jobs the read and write quantities show a mismatch. In the case of job 1117955 the write quantity dwarfs the read. Despite that the read appears to achieve a higher maximum IOPS than write. This might indicate the lack of relationship between maximum IOPS and the total number of I/O requests a job requires.

The I/O behaviour of job 1117955 is given by the read command distributions shown in Fig. 4.53. As observed from the figure the maximum IOPS write occurs at the beginning of the job's runtime. As described in Sec. 4.5.3 (Classification 1.4), the maximum IOPS can suffer from errors in matching GPFS I/O logs to job runtime. Therefore the maximum write IOPS recorded for job 1117955 becomes less trust worthy as it might be resulting from a previous job.

1	Aggregate performance numbers			Read	Write
1.1	Total amount of data			123.818 MiB	2.401 TiB
1.2	Total amount of IOPs			5.60×10^{4}	4.11×10^{6}
1.3	Bandwidth	Max		1.61 MiB/s	66.281 MiB/s
		Avg		1.828 KiB/s	36.312 MiB/s
1.4	IOPS	Max		656	114
		Avg		0.81	59.34
1.5	File commands	Open		1.23×10^{7}	
		Close		8.18×10^{6}	
1.6	I/O intensity	$c = 1$MiB	1.00	0.00	1.00
2	I/O pattern analysis				
2.2	Percentage of small I/O requests ($s \leq 4$KiB)			1.00	0.00
2.4	Percentage of I/O type	in IOPs		0.01	0.99
		in Bytes		0.00	1.00
2.9	Burstiness	$c = 1$MiB		1.00	0.00
3	Parallel I/O				
3.1	Parallel I/O intensity	$c = 1$MiB	0.90	0.00	0.90

TABLE 4.32: I/O criteria analysis map of job 1117955; write IOPS median of jobs with over 1TiB read or write.

Job 1117955 write distribution appears to be interesting, as seen in Fig. 4.53, performing continuous write in parallel over all 16 I/O nodes. Despite that the maximum and average read bandwidth and IOPS appear small. It is therefore questionable whether the job was limited by the available I/O subsystem bandwidth and hence might not benefit from increasing it. Given the job's long runtime, it is difficult to attribute the diminished write bandwidth to other jobs running in parallel. This might also be an effect of measuring the I/O intensity with a lower log temporal resolution of 2min. Using a higher resolution might yield a smaller I/O intensity. The continuity of the write operations might require too large burst buffers to accommodate such large write without throttling performance.

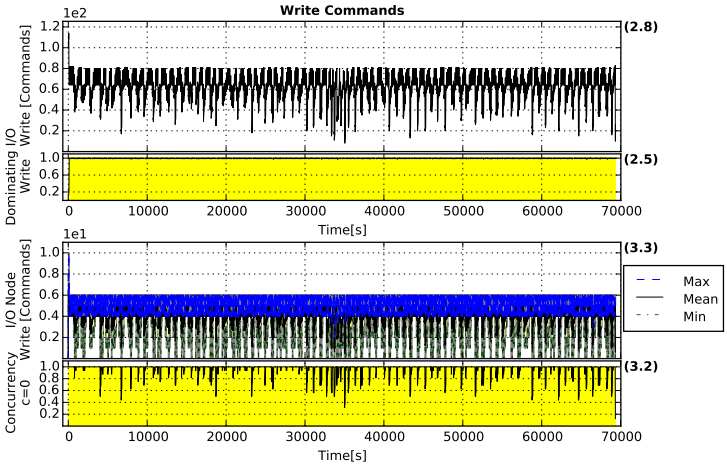


FIGURE 4.53: Write commands for job 1117955; write IOPS median of jobs with over 1TiB read or write.

Classification 1.6 I/O intensity

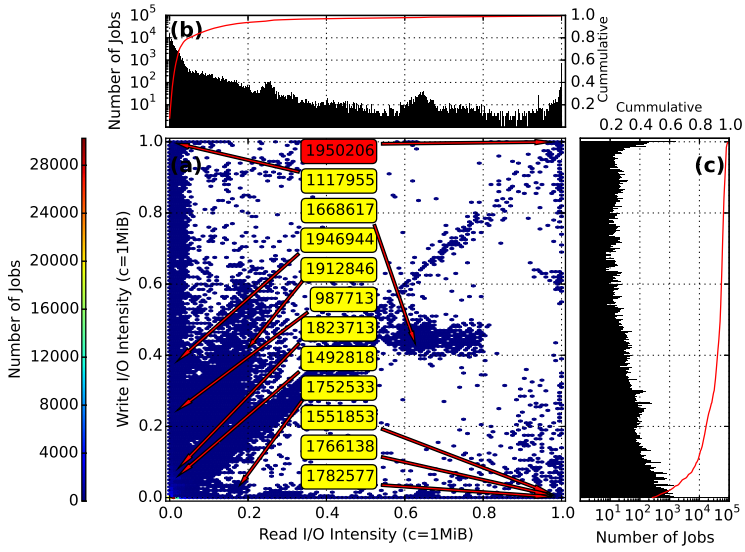


FIGURE 4.54: Read and write I/O intensity ($c = 1$MiB) for selected jobs. (a) Scatter plot of read and write I/O intensity with a heat map for job count, (b) Histogram of read I/O intensity and (c) Histogram of write I/O intensity.

Jobs with high measured I/O intensity are more likely to be I/O limited. Studying the I/O behaviour of such jobs might indicate methods for improving the I/O architecture.

As a result, job 1950206 with an I/O intensity of 1.0 is selected. Fig. 4.54 shows the job's location on the I/O intensity distribution. The figure shows that both the read and write intensity of the job are high.

Job 1950206 I/O intensity of 1.0

Duration[s]	4471
I/O node count	16
Compute node count	2048

TABLE 4.33: Info of job 1950206; I/O intensity of 1.0.

1	Aggregate performance numbers			Read	Write
1.1	Total amount of data			1.304 TiB	1.312 TiB
1.2	Total amount of IOPs			4.12×10^5	3.57×10^5
1.3	Bandwidth	Max		363.329 MiB/s	365.489 MiB/s
		Avg		305.689 MiB/s	307.699 MiB/s
1.4	IOPS	Max		863	114
		Avg		92.14	79.78
1.5	File commands	Open		5.87×10^4	
		Close		5.87×10^4	
1.6	I/O intensity	$c = 1$MiB	1.00	1.00	1.00
2	I/O pattern analysis				
2.2	Percentage of small I/O requests ($s \leq 4$KiB)			0.00	0.00
2.4	Percentage of I/O type	in IOPs		0.54	0.46
		in Bytes		0.50	0.50
2.9	Burstiness	$c = 1$MiB		0.00	0.00
3	Parallel I/O				
3.1	Parallel I/O intensity	$c = 1$MiB	1.00	1.00	1.00

TABLE 4.34: I/O criteria analysis map of job 1950206; I/O intensity of 1.0.

Tab. 4.33 introduces job 1950206 information, while Tab. 4.34 evaluates it's I/O criteria. From this the almost balanced read and write performance can be seen.

Fig. 4.55 and Fig. 4.56 show the bytes read and written distributions respectively. By comparing the two figures, the almost equal distribution of read and write is visible. Both are almost continuously performed in parallel from all I/O nodes. As a result,

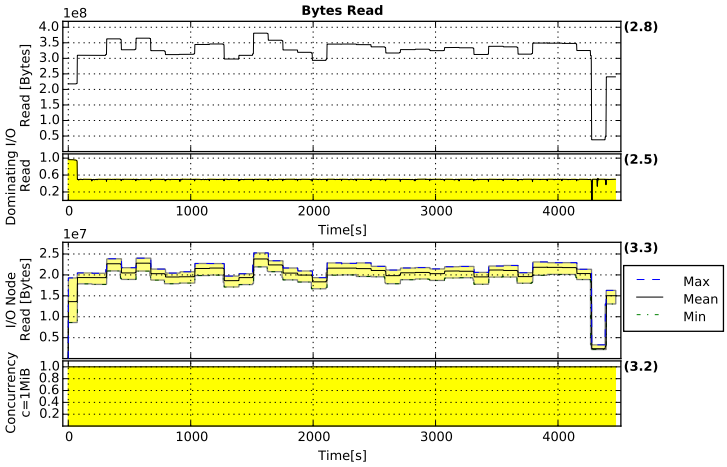


FIGURE 4.55: Bytes read for job 1950206; I/O intensity of 1.0.

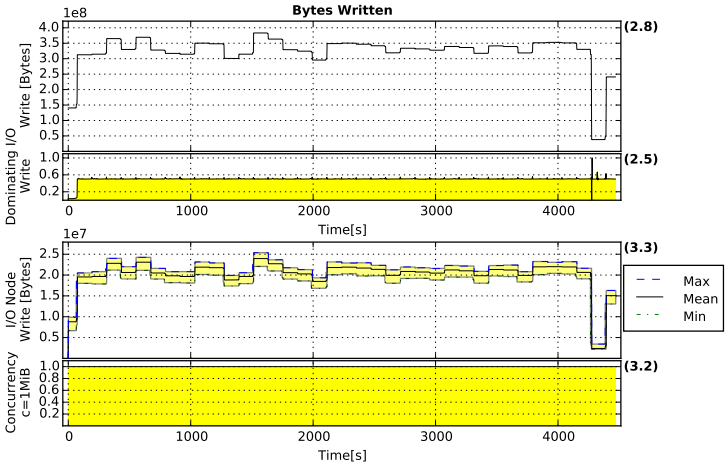


FIGURE 4.56: Bytes written for job 1950206; I/O intensity of 1.0.

both Fig. 4.55-(2.5) and Fig. 4.56-(2.5) are set for almost the entire runtime at 50% dominating read or write.

Since job 1950206 has an I/O intensity for write of 1.0 and a zero burstiness, burst buffers should have a rather low performance improvement, which depends on the size of the burst buffers. On the other hand, prefetching could introduce some limited improvement to the I/O if performed prior to the job's start. This is due to both burst buffers and readahead or prefetching requiring I/O idle times in which asynchronously the buffers are emptied or data can be readahead from storage. The size of the burst or

readahead buffers should consider the size and the duration of the I/O bursts. The idle time between bursts needed to fill or empty the buffer depends on the buffer size and the ratio between internal (buffer to I/O node) and external (buffer to storage system) bandwidth. Whether the job can benefit from increasing bandwidth is difficult to judge. Any observed limit on the bandwidth could also be the result of other factors such as filesystem locks or internal job algorithm. Due to the large I/O intensity of 1.0, it is safe to assume that any improvement of the I/O performance could result in a significant improved overall job performance.

4.5.8 Analysing Jobs Using Category 2: I/O Pattern Analysis

Classification 2.1 Distribution of request sizes

Analysing distribution of requests sizes allows for further understanding the I/O behaviour of jobs. It also allows for locating possible bottlenecks, such as added read or write due to request sizes smaller than filesystem block size. However, as introduced in Sec. 4.5.4 (Classification 2.1), distribution of request size cannot be introduced for all analysed jobs. As a result, jobs cannot be selected from a large scale GPFS I/O log analysis based on having an interesting request size distribution. However, jobs can be selected based on their percentage of small I/O.

Classification 2.2 Percentage of small I/O requests

It is possible to combine the percentage of small I/O requests with the distribution of request sizes to analyse job's request size behaviour. Here the percentage of small I/O can be used to select jobs for which to represent the I/O request size distribution. Tab. 4.35 shows the percentage of small I/O of both read and write for the analysed jobs with $s_{\text{small}} = 1\text{MiB}$. As described in Sec. 2.1 the CIOD buffer limits the request sizes to 4MiB. As a result selecting 4MiB, despite being the suggested filessytem blocksize, would result in all jobs reporting high percentage of small I/O. Therefore, Tab. 4.35 uses $s_{\text{small}} = 1\text{MiB}$.

Analysing the values given in Tab. 4.35 suggests that percentage of small I/O for analysed jobs is either high or low. This could suggest that the job's I/O request sizes

JOBID	Selection	Percentage of small read	Percentage of small write
1782577	Maximum of total bytes read	1.00	1.00
1492818	Maximum of total bytes written	0.01	0.06
1823713	Bytes written median of jobs with over 1TiB read or write	0.00	0.03
1766138	Maximum of total read commands	1.00	1.00
1752533	Maximum of total write commands	0.06	1.00
987713	Write commands median of jobs with over 1TiB read or write	0.04	0.03
1912846	Equal read and write bandwidth	0.00	0.00
1668617	Read bandwidth median of jobs with over 1TiB read or write	0.02	0.14
1946944	Equal read and write IOPS	0.03	0.03
1551853	Read IOPS median of jobs with over 1TiB read or write	1.00	1.00
1117955	Write IOPS median of jobs with over 1TiB read or write	1.00	1.00
1950206	I/O intensity of 1	0.15	0.00

TABLE 4.35: Percentage of small I/O ($s_{\text{small}} = 1\text{MiB}$) of selected jobs.

do not exhibit large variations. To further investigate job's request sizes the request size distribution of a selected group of jobs is provided.

Job 1782577 has a percentage of small read of 100% and its read request size distribution is given in Fig. 4.57. Despite being the job with the most bytes read, almost all I/O requests appear to be just short of 1MiB in size.

On the other hand, job 1492818 performed the most bytes written and exhibits zero percent of small write requests. The distribution for the write request sizes is given in Fig. 4.58. The figure suggest that most I/O requests are between 2MiB and 4MiB.

Both shown request size distributions seem to suggest jobs operate mostly with a limited constant range of request sizes. Since GPFS I/O log measured request sizes are averages,

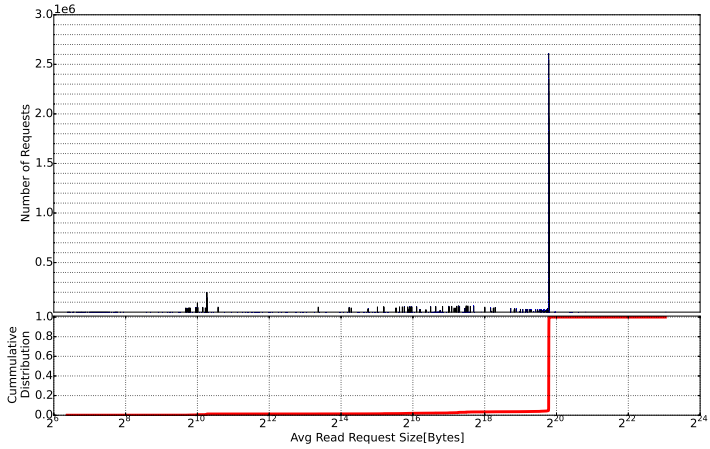


FIGURE 4.57: Distribution of read request sizes for job 1782577.

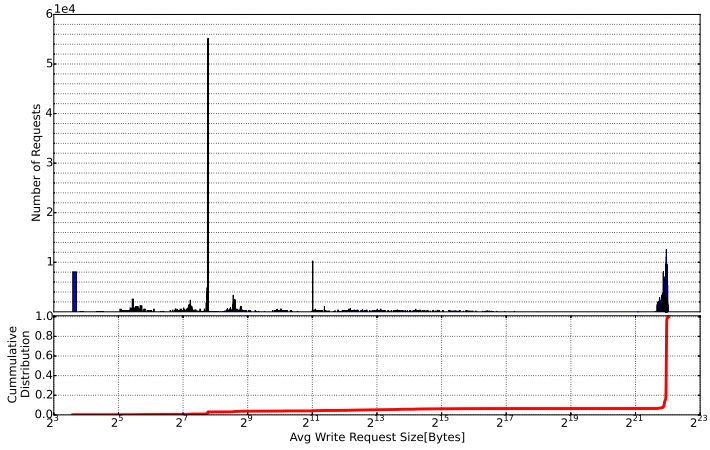


FIGURE 4.58: Distribution of write request sizes for job 1492818.

it might even be possible that these jobs have a single preferred read or write request size that is used by the application. Furthermore, in Sec. 4.5.4 (Classification 2.2) it is suggested that write requests are overall smaller than read requests. Through analysing the distribution of request sizes it is possible to expand on such observation. For example, although both distributions in Fig. 4.57 and Fig. 4.58 perform most of it's I/O requests at certain values, smaller than the job's preferred write request size

appear to be slightly more frequent. It is therefore possible that the overall smaller write requests sizes suggested are the result of job's other tasks such as logging. As a counter suggestion it is possible that the smaller write requests appearing in Fig. 4.58 are simply the CIOD buffer breaking some I/O requests that are slightly larger than 4MiB in two requests.

While small I/O is considered a cause for I/O delay, in Sec. 4.5.4 (Classification 2.2), this fact is questioned and testing the effect of small I/O on the delay is suggested. Another interesting factor is to attempt changing the filesystem blocksize. Indeed in a system where the I/O request size is trimmed at 4MiB it might be reasonable to use a smaller filesystem blocksize, thereby possibly avoiding write and read overhead.

Classification 2.9 Burstiness parameter

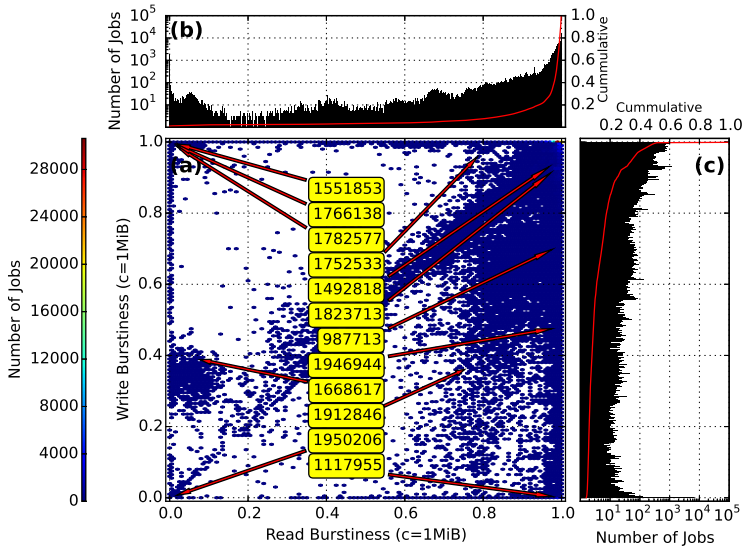


FIGURE 4.59: Read and write job burstiness ($c = 1$MiB) for selected jobs. (a) Scatter plot of read and write burstiness, (b) Histogram of read burstiness and (c) Histogram of write burstiness.

The burstiness has already been sparsely introduced for various analysed jobs. The analysed jobs are highlighted in Fig. 4.59. By observing the analysed job's temporal I/O distribution and comparing it to the evaluated burstiness parameter, it appears that

the burstiness parameter has managed to describe to a good degree the I/O behaviour of analysed jobs.

In general a bursty job could benefit from introducing burst buffers for write and read-ahead or prefetch for read. These would utilize idle I/O times and spread bursts across them. How much a job benefits from these I/O architectural improvements is dependent on many factors. For example, a job with higher write I/O intensity could benefit more from burst buffers, if they manage to spread bursts across the job's computation.

4.5.9 Analysing Jobs Using Category 3: Parallel I/O

Classification 3.1 **Parallel I/O intensity**

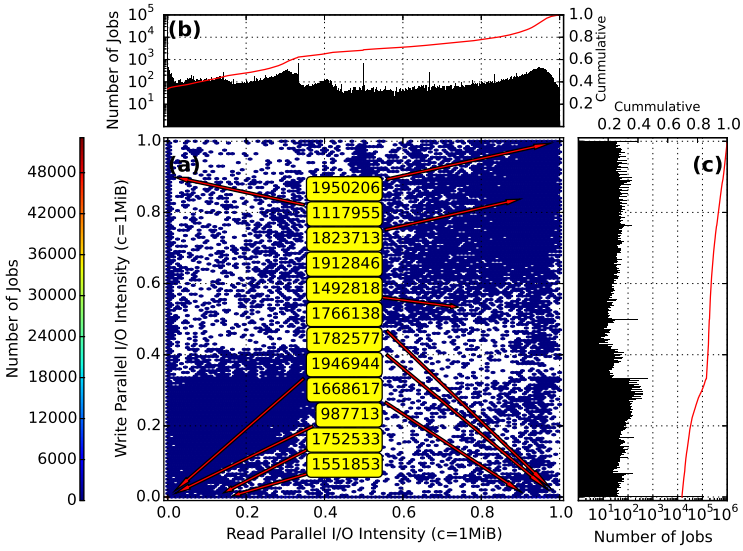


FIGURE 4.60: Read and write parallel I/O intensity ($c = 1$MiB) for selected jobs. (a) Scatter plot of read and write parallel I/O intensity with a heat map for job count, (b) Histogram of read parallel I/O intensity and (c) Histogram of write parallel I/O intensity.

Similar to burstiness, the parallel I/O behaviour has been described for many analysed jobs using the parallel I/O intensity in relation to the I/O operation concurrency and parallel I/O distribution. The analysed jobs have therefore shown that the parallel I/O intensity as an I/O criteria allows for correctly categorizing the I/O parallelism of a

large set of jobs. Fig. 4.60 shows the selected jobs on the distribution of parallel I/O intensity over analysed jobs.

Jobs with low parallel I/O loose part of the available bandwidth, given that less I/O links are utilized to the storage system. While this can be improved by optimizing the I/O parallelism of the application, another solution is to allow for I/O nodes to share the I/O load. As a result I/O nodes with higher loads would use the internal network to send I/O requests to other I/O nodes. However overloading the internal network with I/O requests should be avoided.

4.6 General Notes on Analysing the GPFS I/O Counters

From the analysis of the GPFS I/O logs it is possible to conclude that jobs exhibit a wide range of I/O behaviour. As a result it is not possible to cluster or group the jobs into categories according to the analysed I/O criteria. In general the jobs are found to have a relatively low I/O intensity. This could be the result of the I/O subsystem's capability to handle the magnitude of I/O driven by the analysed jobs. It could also be due to the knowledge of users that I/O operations are potentially expensive and subsequently avoided.

The analysed I/O behaviour suggests the need for considering and experimenting with many possible I/O architectures. This is due to the large number of analysed jobs and their diverse I/O behaviour. The different I/O optimizations could cater for specific I/O behaviours that arise more frequently among jobs. The use of I/O architectures that do not require application changes are recommended.

Despite the demonstrated ability to use the GPFS I/O logs for large job I/O analysis, the possible limitations and observations mentioned should be considered. Other I/O measuring methods could offer more insight into job's I/O behaviour and allow for analysing more I/O criteria. These however come at a cost, such as producing probably much larger data sets to be analysed. Other I/O measuring techniques might also not allow for mass analysis of job I/O. As a result, to better evaluate the I/O behaviour of jobs it is suggested to couple the GPFS I/O logs analysis with other methods. The conclusions of the GPFS I/O logs would direct the attention to a smaller group of jobs

with interesting I/O behaviour, that would later be analysed using more detailed procedures. As an overall principle, the analysis demonstrated here suggests the importance of evaluating the measuring technique and asses the I/O criteria, which have been found to greatly facilitate the I/O behaviour analysis process.

Chapter 5

Performance Modeling: Modeling JUGENE I/O

While the I/O criteria and GPFS I/O log analysis provide an insight into the I/O behaviour of applications, the possibilities to optimize and test various architectures is not provided. In general, reinventing and tuning an I/O system is a complex and difficult procedure and could be hindered by limited financial and hardware resources. However, beyond studying the I/O systems, tuning and optimizing future architectures is necessary to further the understanding of application's I/O behaviour.

Simulations offers an alternative to building several test systems and/or changing a running existing system. The investigated new I/O architecture can be decoupled from existing production systems, thereby avoiding the disturbance that might be caused by repeated changes for users running their applications.

An advantage of simulation is the full control over the setup and the involved parameters. In comparison, analysing real running systems and their architectural changes could suffer from parameter sensitivity.

As noted in Chp. 1, I/O architectures are fairly complex. The target is therefore to create a model of an I/O system that is simple, but still well represents the I/O system's behaviour. Simplifying the model eases the implementation and offers a limitation on the number of changing parameters. The task of creating a simulation becomes defining a set of parameters that can adequately represent the simulated system and be used to

tune for future architectures. For example, a storage system can incorporate the details of the inner workings of each disk. The storage model can also be simplified to only simulate individual storage server or even further simplified to a single module with a specific bandwidth.

The GPFS I/O logs can be used to test the reactions of a simulated I/O system. This provides the simulation with real world I/O patterns. As the GPFS I/O logs have been collected on JUGENE, an appropriate model of JUGENE's I/O subsystem is created and used for investigating future I/O architectures.

5.1 Related Work

Other work has already pursued the path of simulating systems for studying I/O architectures. As a reference, two system simulations that relate to the work done here are presented. The first and closer to this study is the Co-design of Exascale Storage System (CODES) [50], which is later used to simulate burst buffers in [1]. The second is the Hybrid Parallel I/O and Storage System Simulator (HPIS3) [17]. Generally there are many factors by which simulation of I/O architectures can be differentiated. These include simulation purpose, complexity of simulation, which components to simulate, methods for parameter fitting, input data and tools used to perform the simulation.

CODES demonstrates a simulation of a large scale I/O subsystem, particularly a Blue-Gene/P installation [50]. Due to this the simulation bares some similarities to the efforts made here. The target for CODES is the simulation of an end-to-end storage system and to accurately represent storage system software protocols [50]. As a result, components are detailed and encapsulate the protocols of communication between them. Parameters such as link throughput and access latency are selected using micro-benchmarks that ran on a real system [50]. To validate the model, it is fed with IOR (a known I/O benchmark) access patterns. The resulting behaviour is compared to runs of IOR on the real system. To implement this simulation CODES employs the Rensselaer Optimistic Simulation System (ROSS), which is a simulation framework that allows for parallel discrete-event simulations [50].

CODES was extended to simulate burst buffers [1]. To create this extension along with simulating additional components, burst buffers had to be included into the protocols

of the modelled system. The parameters of the burst buffer, such as internal bandwidth and size, were varied to test the effect it has on the simulated configuration [1]. In addition to IOR, the simulation was fed with simulated application I/O patterns as observed on the original system [1].

The HPIS3 simulation intends creation of a framework to facilitate simulation of parallel storage systems. The focus is on testing the creation of better performing system configurations, largely for hybrid systems, which employ both HDD and SSD [17]. The simulation focuses on the storage system with the underlying filesystems and file-servers. HPIS3 has more details and simulates all the way down to some of the inner operations of the storage devices, including HDD and SSD [17]. Parameters can be fitted using benchmarked results from the intended simulated system. The simulation input data is acquired using IOSIG, an I/O monitoring tool, which collected the I/O pattern generated on a real system using IOR [17]. Similar to CODES, HPIS3 uses ROSS as a simulation framework.

5.2 Modeling Framework (OMNET++)

With the exception of organizational tasks, an I/O system generally operates by reacting to I/O requests triggered by compute nodes. Therefore discrete event simulations are well suited for simulating an I/O system. These reduce the simulated system to a set of components that react to a set of events in time. As a result, simulated time only moves forward between two given events, hence the name discrete event simulation.

Although it is possible to implement discrete event simulations from scratch, using available established simulation environments is less prone to errors. Many available packages and environments exist in several programming languages. It is necessary to select the simulation framework that most fits the task and if possible decrease effort to implement and debug the model.

Modern I/O systems can be viewed as a set of complex components that are interconnected using a set of networks. OMNeT++ is a discrete event simulator targeting the simulation of communication networks, multiprocessors and distributed systems [51]. This makes OMNeT++ suitable for simulating the JUGENE I/O subsystem.

An OMNet++ simulation is based on connecting various modules. These can exist in a hierarchy, where a compound module is a collection of simple or other compound modules. The connections and communication between modules, i.e. the model's topology is described using the OMNET++ defined Network Description (NED) language [51]. Modules exchange messages, which trigger events. To simulate passage of time, a module can send a scheduled message or event to itself. The module's behaviour or reaction to events is defined in C++ [51].

Depending on events to signal progress in simulated time makes the simulation independent of the system it is running on. Since the inner workings of I/O system modules are rather complicated, they can be split into several internal modules. This avoids creating large modules that are complex and error prone and simplifies validating and debugging the simulation. Internal modules can be connected using zero delay networks.

5.3 Modelling JUGENE I/O

Modelling JUGENE I/O requires translating complex I/O behaviour of system components into a simulated version. The resulting simulation has to react to I/O requests in a similar manner when compared to the original I/O system. This requires I/O model verification, which is performed by comparing measured I/O on the original system to it's simulated version. For JUGENE I/O the verification process uses the GPFS I/O logs described in Sec. 2.2.1 and analysed in Chp. 4. The verification cycle is shown in Fig. 5.1.

JUGENE and its I/O network has been previously introduced in Chp. 2. To model the I/O system of JUGENE, it is necessary to study each component and determine the level of details to simulate. Since the interest here is in I/O, other components can be ignored. For example, only the binary tree network of BlueGene/P is part of an I/O request, therefore there is no need to simulate the 3D torus. Other components need to be introduced. For example, a request generator, that triggers compute nodes to create I/O requests, is necessary to simplify control over simulated modules.

The level of details to which the JUGENE I/O subsystem can be modelled is limited by the information available on its I/O behaviour and the GPFS I/O logs that are used to verify the model. The interest here is in overall data movement by applications based

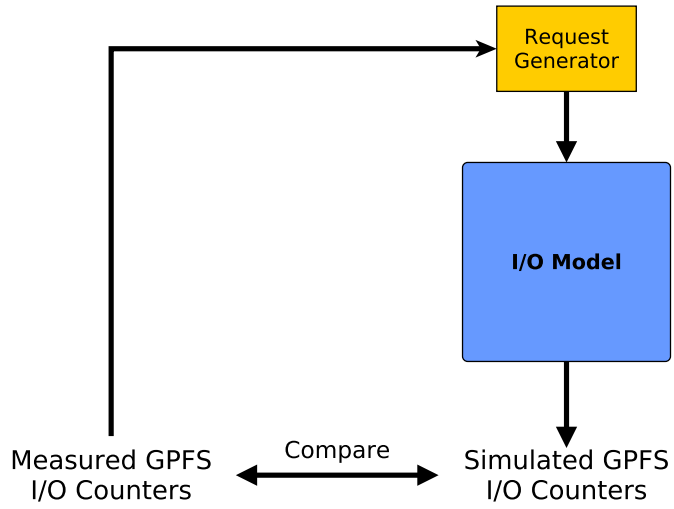


FIGURE 5.1: JUGENE I/O model verification cycle

on the GPFS I/O logs. Simulating the inner workings of some components such as disks or the GPFS filesystem would complicate the implementation and add a wide set of unknown parameters. These components are simplified and the GPFS I/O logs are used to verify their overall I/O behaviour. While in [50], handshakes and protocols have been included, the I/O model here does not perform any kind of pre-communication setup between components.

5.3.1 I/O Model Components

Fig. 5.2 shows the components of the JUGENE I/O model. It is constructed from a compute node group, connected through the binary tree by an I/O node. All I/O nodes are then connected to a common GPFS server, which in turn is connected to the Disk module. The inner workings of each component has been simplified, avoiding complex protocols and handshakes. The model does not simulate metadata or file read write operations. The reason behind this simplification is the use of the GPFS I/O counters to test the model. As the I/O data does not offer a link between files and data read or written, it is not possible to model metadata accesses. In [50] complex protocols have been added to the model to simulate file access protocols. Such protocols can

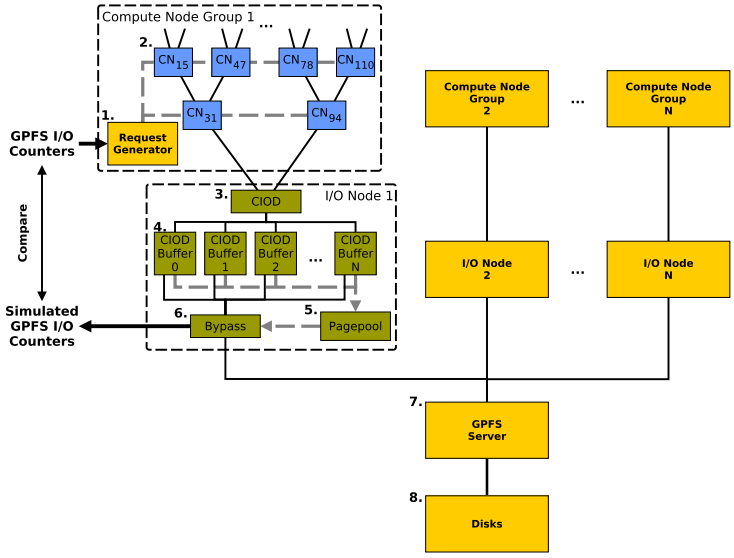


FIGURE 5.2: JUGENE I/O model components.

be later implemented into the JUGENE I/O model, should the necessity rise for such changes. The I/O model here is more focused on the overall data movement in relation to application I/O access. And this is tested and verified using the long term collected I/O logs from the GPFS I/O counters.

The following explains each component of the JUGENE I/O model and its behaviour:

1. **Request Generator,**

   creates the I/O requests and distributes them over the connected compute nodes. The requests can be constructed in different configurations. This allows for various tests and validations to be used. The most relevant configuration employs GPFS I/O logs, which the request generator divides over the compute node group. As the I/O logs only contain information on total data read/written and total number of read/write requests per 2min, the generator creates average sized requests. These are then distributed in a round-robin on the available compute nodes. If an I/O request is larger than the 4MiB allowed by the CIOD buffer (see Sec. 2.2.1), the request generator splits the request. Although this case should not occur per I/O system design, the GPFS I/O logs average an I/O request size that occasionally is larger than 4MiB (see Sec. 2.1). This behaviour might add some I/O requests that

do not exist in the GPFS I/O logs. Another factor worth mentioning, is the distribution of I/O requests by the request generator over time. Since the GPFS I/O counters are logged every 2min, the request generator lacks the timing information of each I/O request as performed by the application. To avoid adding additional unknown parameters, the I/O requests logged within the 2min are generated at the beginning of the interval.

2. **Compute Node (CN),**

   receives the I/O request over a zero delay connection from the request generator. The compute node then forwards these requests over the binary tree network. Each link in the binary tree has a bandwidth of 850MiB/s. I/O operations are synchronously assigned, therefore a maximum of one I/O request is in flight per compute node at any given time.

3. **CIOD,**

   represents the control and I/O daemon (CIOD) of the I/O node. It receives the request over the binary tree network links and forwards the message to the CIOD buffer belonging to the compute node that initiated the request.

4. **CIOD Buffer,**

   is a 4MiB buffer and only holds requests from a single compute node. Requests have been previously split if larger than 4MiB, therefore no overflow is possible using one I/O request. Although, compute nodes perform only synchronous I/O, the buffer is designed as a queue and can hold more than one request. This could be used in the future to simulate asynchronous I/O. In that case, possible buffer overflow from multiple requests has to be taken into account. The CIOD buffer can either forward the request internally to the pagepool if space is available, or sends the request directly to the GPFS server.

5. **Pagepool,**

   is offered by the GPFS client as a cache. Although JUGENE's 1024MiB pagepool offered both read and write buffering, it is difficult to perform read caching in the simulation. This is due to missing addresses and information on data allocations for the GPFS I/O logs. In other words, it is not possible to determine if two requests address the same data or not. As a result, the model's pagepool only offers write buffering. It is also not possible for the simulated pagepool to perform

any merge or sort of the buffered I/O requests. Once a write request is forwarded to the pagepool, it responds with an acknowledgement to the compute node, which can then proceed with the next request. Once the path to the GPFS server is free, the pagepool would drain the stored write requests to the storage.

6. **Bypass,**
   offers a possible bypass to the pagepool. It has the task of forwarding the I/O requests from either the CIOD buffers or the pagepool to the GPFS server. Additionally it collects the simulated GPFS counters. It is reasonable to assume that the GPFS counters are located below the pagepool, which could explain the presence of larger than 4MiB I/O requests. These could be the result of merging several smaller I/O requests in the pagepool. It should be noted here that the bypass is not a buffer in itself. Therefore the pagepool and CIOD buffers are not allowed to forward any I/O requests to the bypass, unless the link to the GPFS server is free and can be used by the forwarded I/O request. Using this the bypass creates collision free traffic on the 10GbE link to the GPFS server.

7. **GPFS Server,**
   operates as a switch, connecting the 10GbE link from the I/O nodes with the 66GiB/s link of the disks. In comparison, real GPFS servers and the filesystem used are far more complicated and contain a multitude of components, to stripe and distribute the data over the different servers and disks.

8. **Disks,**
   represents the end of the path where data is either placed into the disks and responded to by an acknowledgement or read and forwarded back to the compute nodes. There is no delay in either reading or writing the data. The delay to the disks come solely from the bandwidth of the link between the GPFS server and the disks.

9. **Messages,**
   are the events sent between the I/O model components and represent requests and acknowledgements simulating the GPFS I/O logs. A message can either be a request, an acknowledgement (ack) or a data transfer. Requests and acks have zero size and a higher priority than data transfer messages. However, when travelling along links with limited bandwidth or delay, requests and acks still need to wait

in case the link is busy or other requests are being transferred. The flow graph of a write request is given in Fig. 5.3, and shows the decisions the request has to traverse. When a write data transfer arrives at the disk, the disk creates an ack and sends that in return to the compute node. On receiving the ack message the bypass checks if the write request has been previously acknowledged by the pagepool. If it has been, the bypass does not forward the ack message to the compute node. The bypass still needed to receive the ack from the disk, as it is collecting the simulated GPFS counters. A read request traverses the same path, excluding the pagepool, as seen in Fig. 5.4. As a result, a read request has less decisions to make on its path.

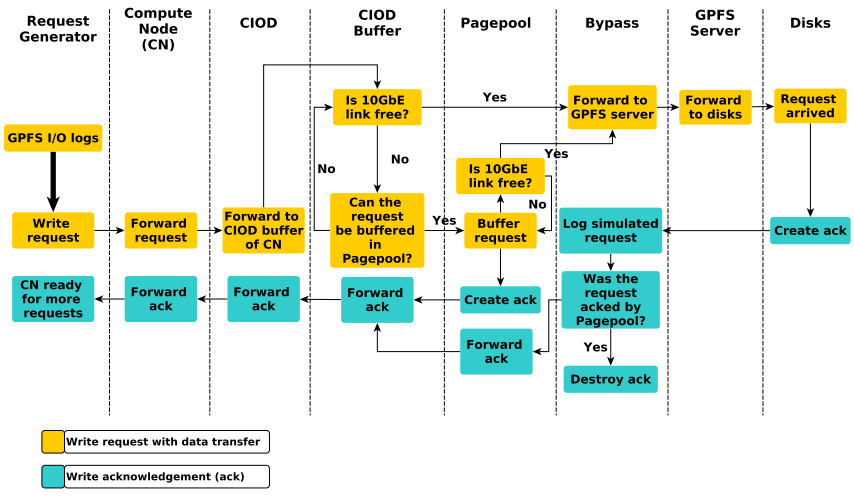


FIGURE 5.3: I/O model write flow graph.

While the three figures, Fig. 5.2, Fig. 5.3 and Fig. 5.4 make no distinction between the links connecting the components, there are four connection types. The following describes the connection types and provides the original JUGENE system bandwidth for each link if available:

Internal zero delay are the links connecting internal components of either the I/O nodes or the compute node groups. These include the link connecting the request generator to compute nodes; the CIOD to CIOD buffers; the CIOD buffers to both

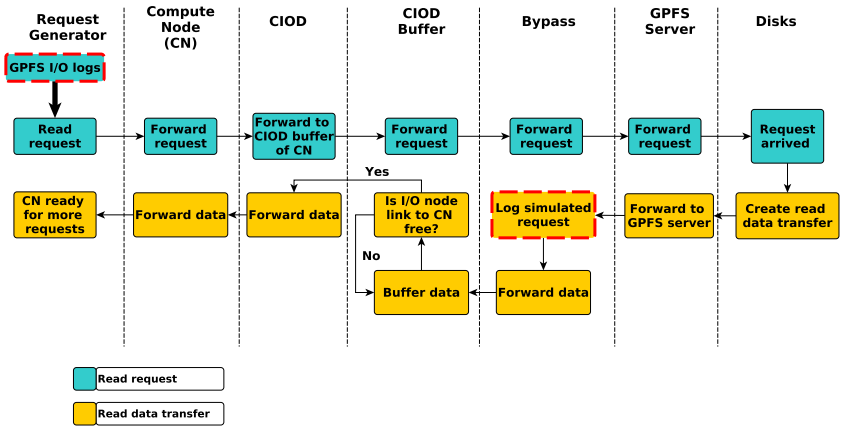


FIGURE 5.4: I/O model read flow graph.

pagepool and bypass; and pagepool to bypass. As the name suggests, these links have no delay.

Binary tree are the links connecting the compute nodes to each other and the compute nodes to the I/O nodes. These have a bandwidth of 850MiB/s.

10GbE are the links connecting the I/O nodes to the GPFS server. These have a bandwidth of 10Gbps

GPFS server to disk provides the bandwidth limitation to storage access and has a value of 66GiB/s.

5.4 I/O Model Verification

The verification process involves comparing the I/O behaviour of the model to the I/O behaviour of the original system. To provide as close as possible approximation of the I/O system, the model's parameters are changed. This is termed parameter fitting, which is the process of selecting parameter values to bring the simulation closer to the operation and delays of the real system. The main target is to use the GPFS I/O logs to drive the I/O model. As a result, parameters that cannot be evaluated using the GPFS I/O logs are not considered. The parameters of the I/O model mainly include the

bandwidth of the links. These can be gradually changed until the overall performance resembles the real I/O system.

As discussed earlier, their exists four link types, internal zero delay, binary tree, 10GbE and GPFS server to disk. The internal zero delay communicates between functional components within the compute and I/O nodes and therefore cannot be tuned due to missing information on internal node operations. The GPFS I/O logs are logged on the I/O nodes filesystem clients and as a result the binary tree bandwidth cannot be tuned using the GPFS I/O logs. This leaves two links for tuning, the first is the 10GbE connecting the I/O nodes to the GPFS server and the second is the GPFS server to disk link.

In [5] it is shown that the I/O nodes are unable to drive the filesystem at the full rate available by the 10GbE. Meanwhile, the GPFS system's peak bandwidth of 66GiB/s is an aggregate value that might not be achievable by the I/O nodes. In fact this bandwidth is divided among different filesystems [5]. Both of these links might require tuning in accordance to the verification done using the GPFS I/O logs.

5.4.1 Parameter Fitting Using GPFS I/O Logs

The GPFS I/O logs are used to generate I/O requests that drive the full I/O model simulation of JUGENE for longer periods of simulated time. This allows for an objective evaluation of the chosen parameters. It also allows for the simulation to be driven using real application I/O, making it possible to compare the JUGENE I/O model's behaviour to possible future I/O architecture models. Another method for parameter fitting the I/O model using an I/O micro-benchmark can be found in App. B.

Using GPFS I/O logs for parameter fitting is subject to some limitations. The 2min interval between logging GPFS I/O counters gives no clear indication of the exact duration and timing of each I/O request. Fig. 5.5 shows a possible real I/O request distribution over the 2min that is captured by the GPFS I/O log as four values, the total bytes read and written and the number of read and write requests. As a result, the exact timings and sizes of the I/O requests is not reflected in the GPFS I/O logs. To mitigate this, the I/O model generates all logged I/O requests at the beginning of the 2min interval as seen by the simulated I/O requests in Fig. 5.5. The figure also shows for the simulated

I/O requests that the total I/O quantity logged by the GPFS I/O counters is distributed equally across the logged number of I/O requests. Therefore, the simulated I/O requests have the average read/write request size over the 2min.

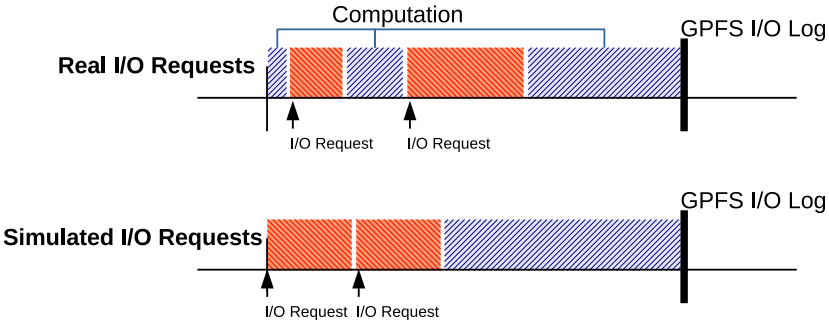


FIGURE 5.5: Real versus simulated GPFS I/O log.

Due to the missing timing and size information on individual I/O requests, the simulation cannot be exactly tuned to perform the same requests. Fig. 5.6 shows how the GPFS I/O logs can help in testing and tuning the I/O model. Only logs in which pure I/O was done for the complete 2min can detect a mismatch of performance between I/O model and real I/O logs. In case the duration of the I/O during the simulation is longer than 2min, tuning the parameters would be needed. However, if the simulation runs through the I/O requests faster than the 2min, no mismatch will be detected.

Although I/O model to real I/O request mismatch might indicate a possible error in tuning the I/O model, it should not effect the conclusions drawn from the simulations. The purpose is not to create an over all accurate depiction of the I/O operations on an I/O subsystem. Rather the aim is to create a sufficient modelling of I/O that would allow comparing different I/O architectures. As long as the parameters and model fulfil real I/O system conditions the comparison is reasonable. This point will be further clarified when introducing the future I/O architectures in Sec. 5.5.1.

To achieve reasonable tuning of I/O model it is necessary to run the simulation for extend simulated time. It should therefore include many I/O logs. Tab. 5.1 shows the number of error logs and the average error across the logs for 24 hours simulated time using the original bandwidth values for the links, i.e. 10Gbps for the 10GbE and 66GiB/s for the GPFS server to disk link.

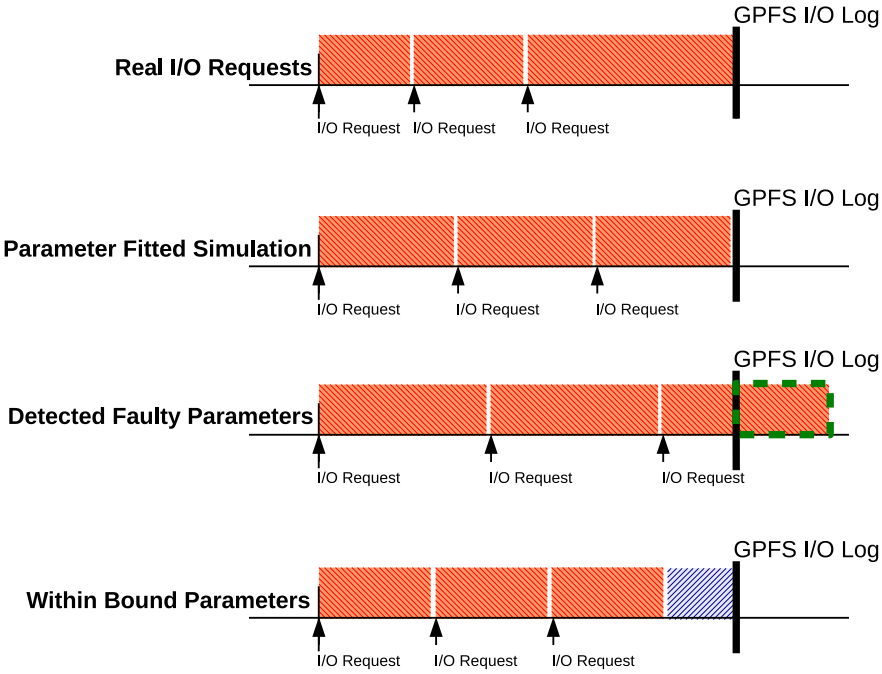


FIGURE 5.6: Using GPFS I/O logs for I/O model parameter fitting.

	Logs with error	Average error of logs [%]
Bytes Read	15856/425227	0.023
Read Commands	713/425227	0.45
Bytes Written	241147/425227	0.2
Write Commands	5108/425227	2.08

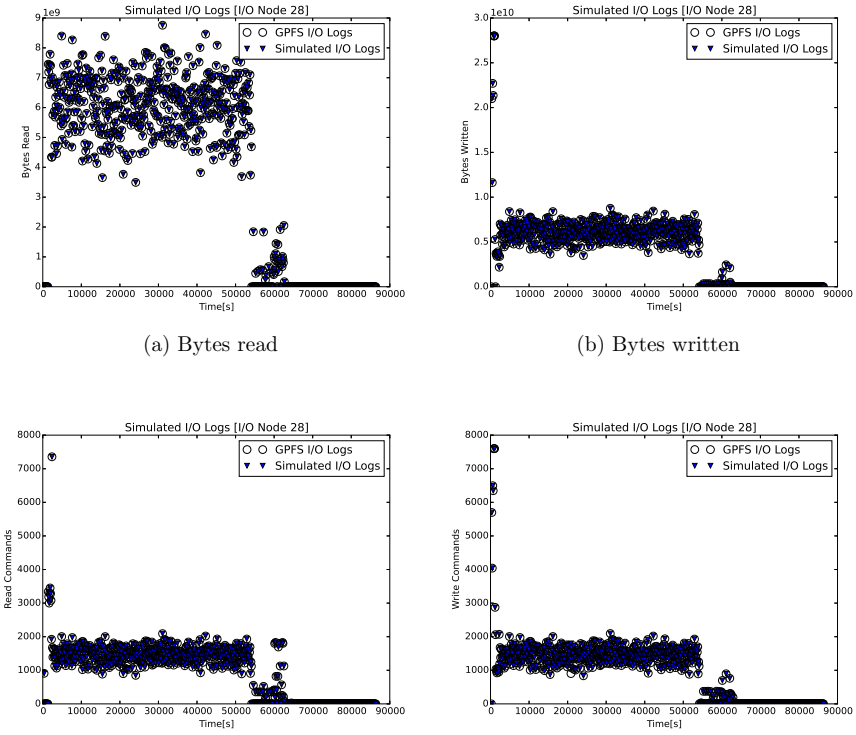
TABLE 5.1: Number of logs and average error for 24hours simulated time.

The final decision on the accuracy of the I/O model rests on the average deviation of simulated I/O from the GPFS I/O logs given in Tab. 5.1. Here the average error of logs, that is the deviation of simulated logs from real logs, is quite small, with a worst case of only 2% for write commands.

The exact reason for this slight variation between real and simulated I/O is unknown, but can be, at least partially, attributed to floating point errors. Another possible contributor to the error is the existence of larger than 4MiB requests in the GPFS I/O

logs. These are, per I/O model design, dissected into several smaller requests, thereby changing the count of I/O requests for either read or write. This shows that debugging and correctly configuring such a large model is a delicate and difficult task. As a result slight variation between real and simulated I/O are to be expected and can hardly be avoided. As will be discussed in Sec. 5.5.1 the main focus is on changes in the I/O behaviour when subjected to various new I/O architectures. As long as the I/O resembles real system I/O, within a limited margin, the I/O behaviour of the I/O model can be compared with a changed model. The conclusions are drawn on the changes and not on the accuracy of the real I/O on the I/O model. The I/O model should behave realistic when driven with real I/O logs which can be deduced from the results shown in Tab. 5.1.

Fig. 5.7 shows an example I/O node comparing the GPFS I/O logs to the simulated I/O. From the figure it can be seen that the simulated and real I/O logs correlate.



(a) Bytes read

(b) Bytes written

(c) Read commands

(d) Write commands

FIGURE 5.7: Example of an I/O node's 24hour I/O model simulation.

The time spent in I/O observed by the simulation is also needed for simulating and comparing I/O architecture changes. Fig. 5.8 shows the time spent in I/O for each I/O node. It is also worth observing the time each I/O node spent executing jobs in relation to the I/O time. This is given in Fig. 5.9 as the percentage of job execution time spent in I/O.

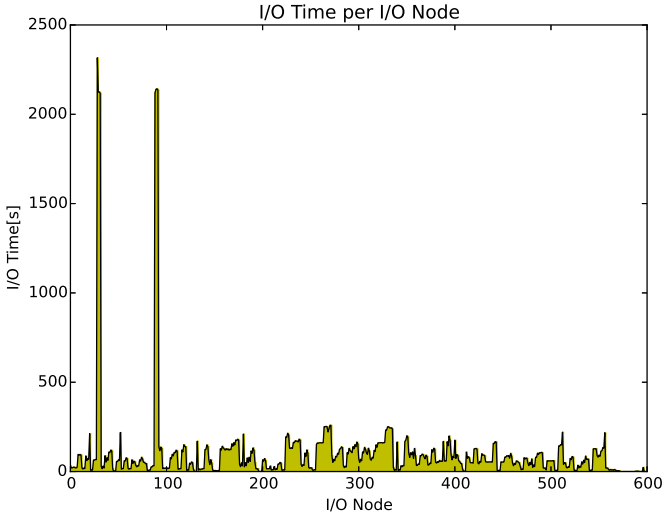


FIGURE 5.8: Simulated time spent in I/O for each I/O node.

5.5 Future I/O Architectures

As previously mentioned, the main reason for modelling and simulating JUGENE I/O is to experiment with architecture changes. By comparing a standard I/O model using the I/O logs with different I/O subsystem architectures, some conclusions can be drawn. These simulations can give an insight to the usability and effectiveness of I/O architectures under real I/O loads. In general performing simulations for new I/O architectures is advised to evaluate their impact on I/O prior to implementing them. This is specially true for I/O architectures that require massive or difficult changes from both the hardware and the scientific applications.

Different I/O architectures vary in the improvements and changes they offer to the I/O system. The main thoughts for most I/O system updates are performance and

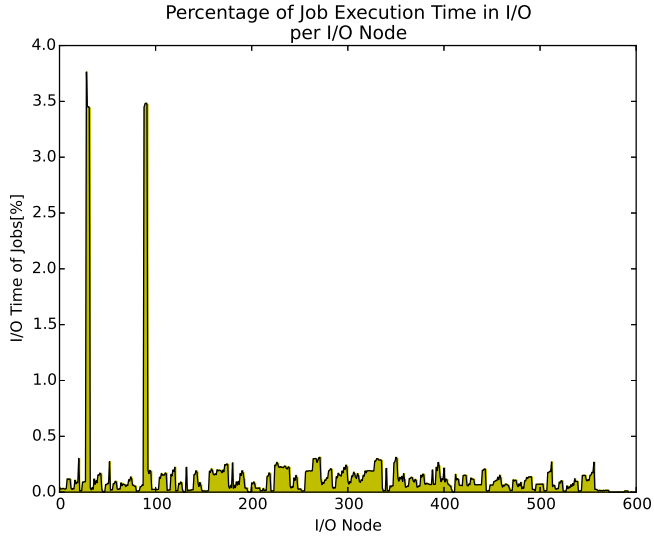


FIGURE 5.9: Simulated percentage of job execution time in I/O for each I/O node.

storage space. While storage space is not modelled here, performance improvement is considered to be less time spent in I/O for an application. However, an I/O architecture can bring more than just a better performance. By additionally considering cost, some I/O architectures might become interesting by achieving equivalent performance for less cost.

5.5.1 I/O Model Changes

To model different I/O architectures the I/O model requires the addition of new components and/or changing component's behaviour. As a direct result the I/O model as a whole changes behaviour. By comparing the difference between the original and the updated I/O model the effect of the changes on the I/O behaviour is modelled. Since the conclusions are drawn from the comparison, there is no need to recreate an exact detailed I/O model or replicate the exact I/O requests. As long as the I/O model entails the main components and replicates their main behaviour under real I/O load the comparison can yield useful insights into I/O architecture changes. All this should be done while observing the limitations of the simulation and the I/O measuring limitations discussed in Chp. 3 and Chp. 4.

Correct conclusions can be drawn from comparing the original to changed I/O model only if the differences are limited as much as possible to the tested I/O system changes. Updating the I/O model could lead to changing the time taken to conclude the given I/O requests. As a result, I/O requests are shifted in time and the I/O load as a whole would change. An additional issue is the possible prolonged I/O request delay that might lead the GPFS I/O log to be overdrawn. In such a case the simulation of individual GPFS I/O logs could overlap or require shifting.

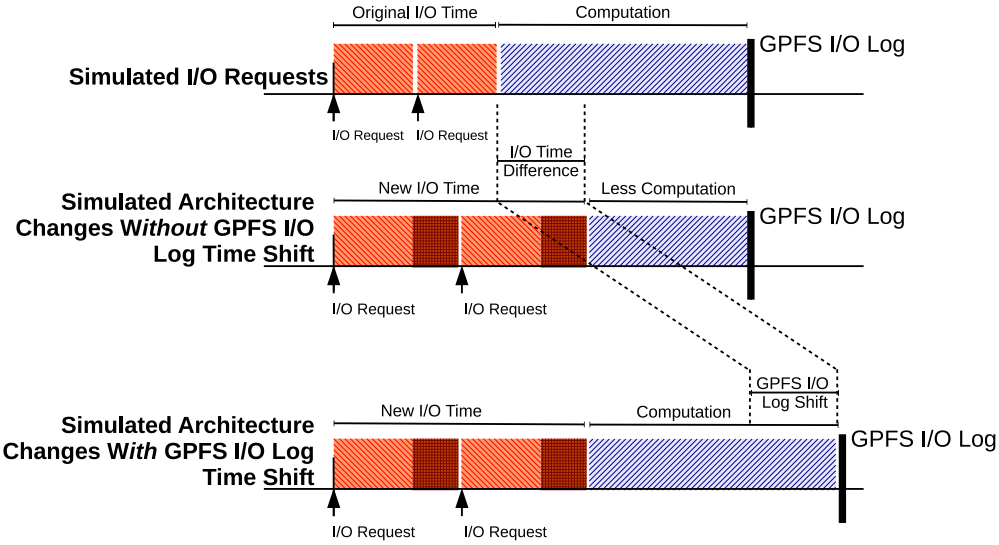
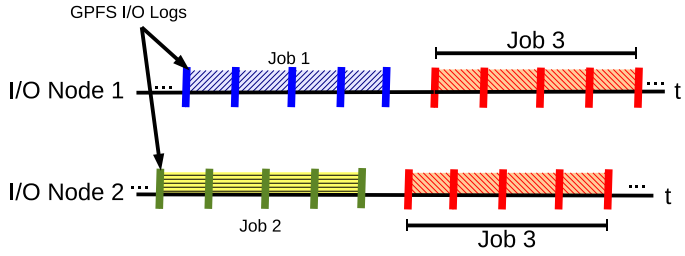


FIGURE 5.10: Time shifting GPFS I/O logs in the I/O model.

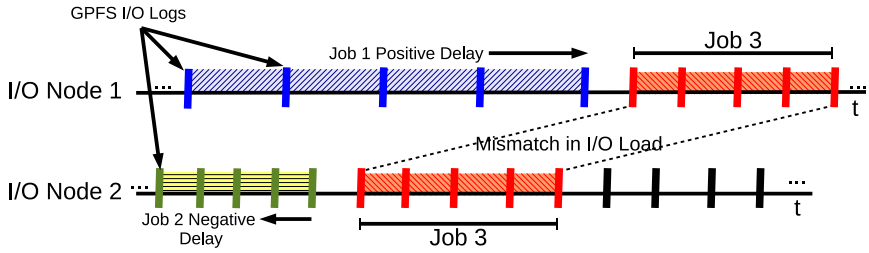
To allow for minimal I/O load changes to affect the comparison, a unified rule for shifting GPFS I/O logs is required. Fig. 5.10 shows the principal of shifting GPFS I/O logs according to the difference in time spent in I/O requests. The main concept is preservation of computation time. The I/O architectural changes should only effect the I/O time. Comparing the delay for the original I/O model with the new delay for the changed I/O model, the shift required for the GPFS I/O log can be measured. This is shown in Fig. 5.10, where comparing the time taken by simulated I/O requests with simulated architecture changes yields the I/O log shift. This holds true for both positive and negative I/O time changes. The difficulty here is to keep track of two simultaneous

time-lines, the GPFS I/O log timings and the simulated shifted timings. As a result more care has to be given into simulation correctness, component behaviour and event logging to ensure proper simulation.

As GPFS I/O logs shift, the synchronization between I/O nodes can change. The I/O requests are not always going to be equally shifted as the I/O model changes might lead to variable delays. An example of such a shift can be seen in Fig. 5.12-(a). As a result two or more I/O nodes involved in the same I/O load of a job could perform the job's I/O at different times. Fig. 5.11-(b) shows the miss shifting of GPFS I/O logs making I/O nodes unsynchronized.



(a) Simulated job I/O synchronized across I/O nodes



(b) Unsynchronized job I/O due to GPFS I/O log time shift

FIGURE 5.11: Job I/O mismatched timing on different I/O nodes due to GPFS I/O log time shift.

To achieve an almost equal load on both the original and the changed I/O model it is necessary, to resynchronize the I/O of a job across the I/O nodes. This is achieved by adding an additional component to the JUGENE I/O model called Re-Scheduler. It's

task is to delay performing the I/O requests of the GPFS I/O logs until all I/O nodes involved in the job are ready. As a result the I/O logs of the job are resynchronized. Such process is shown in 5.12.

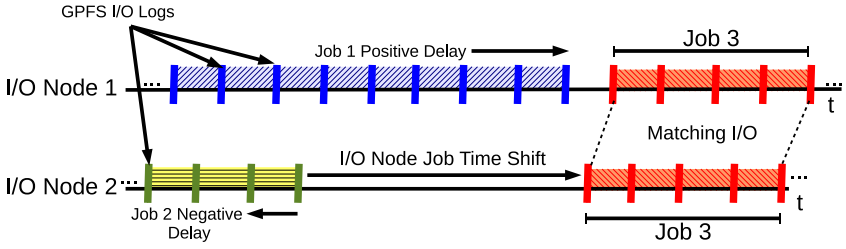


FIGURE 5.12: Job I/O resynchronizing of GPFS I/O logs on different I/O nodes.

Despite the use of the re-scheduler to execute GPFS I/O logs of a job across multiple I/O nodes at the same time, there is no resynchronization occurring while a job's GPFS I/O logs are executed. Any shift in a job's internal I/O logs across I/O nodes is tolerated as there is no information on the relationship between the separate I/O requests. Indeed the shifting of GPFS I/O logs for job I/O synchronization leads to empty spots in which no I/O is performed. As a result the I/O load may be lessened over certain periods of time. Such observation correlates with the intention that all is kept constant between original and updated I/O model with exception of tested I/O architecture changes. Even the job scheduling decisions are kept the same.

5.5.2 Burst Buffers

As a counter measure to increasing I/O bandwidth some systems opt for placing burst buffers in the form of SSDs integrated into the computing system or on the I/O subsystem to catch I/O bursts. Burst buffers are therefore bridging the gap between fast and slower connections. Applications can dump their data on the close burst buffers and return to computation. The burst buffer then over time moves the data onwards to the storage system. Although burst buffers can also operate as caches by retaining data for rereading, they mainly optimize the write path.

In [1] the observation of bursty application I/O is stated as a well known issue. This coincides with the findings made while analysing the burstiness of job I/O using the GPFS I/O logs in Sec. 4.5.3 (Classification 2.9). This could indicate the usefulness of burst buffers to the I/O subsystem. As described in Sec. 5.1, [1] has also performed simulations to test the use of burst buffers.

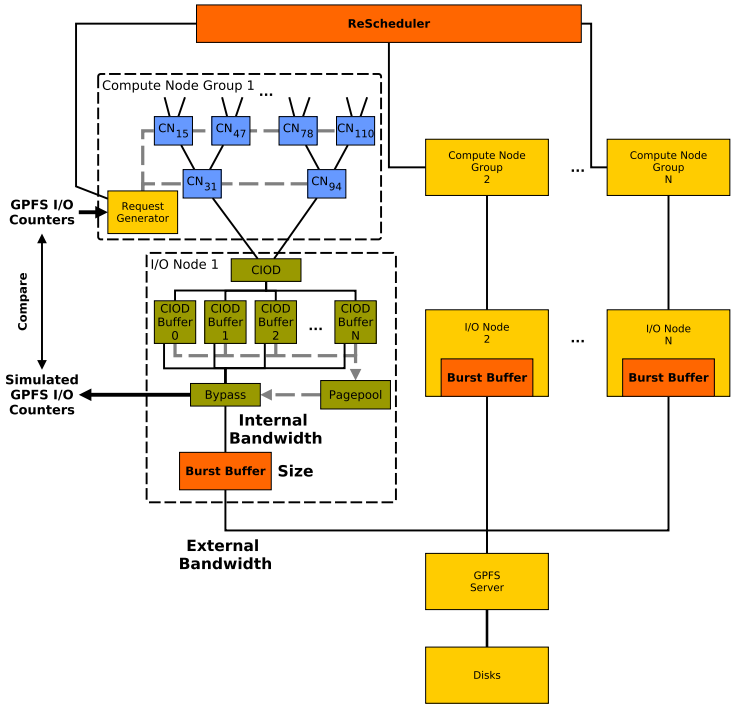


FIGURE 5.13: Burst buffer I/O model.

Fig. 5.13 shows the change needed to simulate burst buffers. Each I/O node has an added burst buffer. As a result the model includes three additional parameters. The first is the internal bandwidth connecting the I/O node's I/O to the burst buffer. The second is the size of the burst buffer. Finally the third is the external bandwidth connecting the burst buffer to the GPFS servers. Being part of the internal workings of the I/O node the internal link can be considered equivalent to the links connecting the compute nodes with the I/O node. Therefore a reasonable value for the internal bandwidth is 850MiB/s. This leaves two parameters that can be varied, burst buffer size and the

external bandwidth[1].

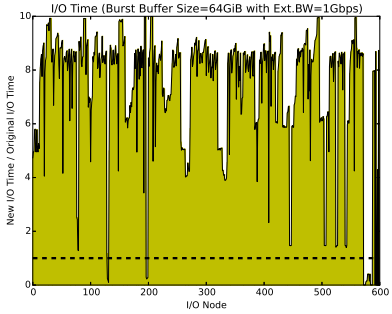
The two remaining parameters, external bandwidth and burst buffer size, need to fulfil real conditions and simulate intended improvements. The original external bandwidth of JUGENE is 10Gbps, which is difficult to improve. Additionally [5] shows that the limiting factor lies in the disk bandwidth. However as the 10Gbps Ethernet cards represent a significant cost factor, the opportunity of using smaller external bandwidth without loss of performance could significantly reduce total system cost. To test such an opportunity, the simulation can be run with several available external bandwidth and burst buffer size combinations. The first possible value for the external bandwidth is a 1Gbps representing the use of a 1Gbps Ethernet card. The second external bandwidth is 4Gbps representing the use of 4x1Gbps Ethernet cards, a common implementation in various systems. It is possible to start the burst buffer size at 16GiB and increase it upto 256GiB. Although SSDs are well suited for the task of burst buffers, the limited number of write cycles offered by SSDs should be considered when regarding total system cost.

Fig. 5.14-(a) shows the relative I/O time change of I/O nodes for using a 64GiB burst buffer with an external bandwidth of 1Gbps. For most I/O nodes the I/O time is increased, in some cases up to 10 times, while for a few I/O nodes the I/O time is decreased. The benefit from burst buffers depends on many factors, such as I/O burstiness. Additionally, read is not improved by the placement of burst buffers and would only be slowed down by the limited external bandwidth.

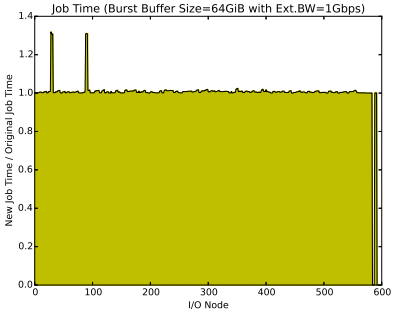
To further investigate the effect of using burst buffers both the burst buffer size and the external bandwidth can be changed. Fig. 5.15-(a) shows the relative I/O time of I/O nodes change for using a 16GiB burst buffer with an external bandwidth of 4Gbps. As the figure shows, the I/O time is well improved compared to having a 64GiB burst buffer with 1Gbps external bandwidth. The improvement could indicate that the external bandwidth is the bottleneck rather than the burst buffer's size.

As previously mentioned, a performance improvement or a cost reduction can justify the use of a burst buffer. Although the simulation results indicate reduced performance for I/O time, both Fig. 5.14-(b) and Fig. 5.15-(b) show that the simulated jobs' execution

[1]There are other possible parameters that can be added to the burst buffer model, which are not considered here, such as fill level.

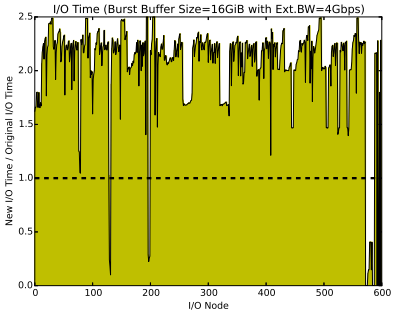


(a) Change in I/O time per I/O node

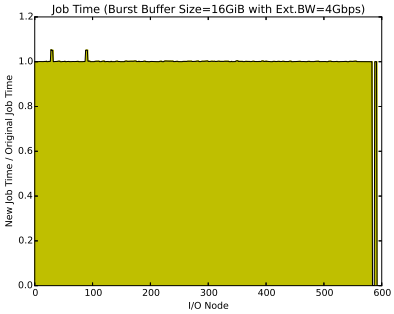


(b) Change in job execution time per I/O node

FIGURE 5.14: Change of I/O and job time per I/O node using burst buffers of size 64GiB and an external bandwidth of 1Gbps.



(a) Change in I/O time per I/O node



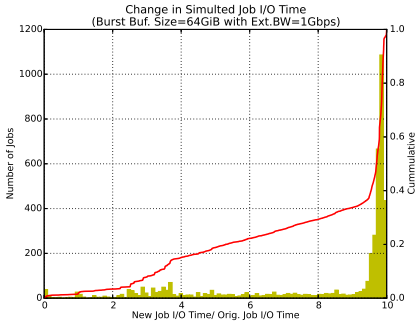
(b) Change in job execution time per I/O node

FIGURE 5.15: Change of I/O and job time per I/O node using burst buffers of size 16GiB and an external bandwidth of 4Gbps.

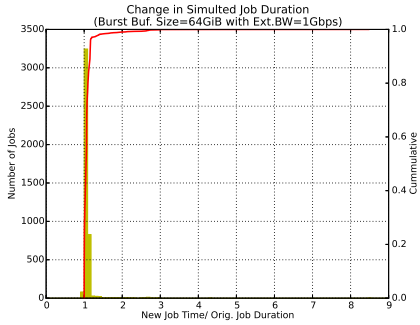
time per I/O node is less effected. Therefore it is still possible to reduce system cost by the use of burst buffers, if most jobs exhibit low I/O times.

The effect of the burst buffers on the jobs' I/O and execution time are given in Fig. 5.16 and Fig. 5.17, showing the use of burst buffer size 64GiB and 16GiB with 1Gbps and 4Gbps external bandwidth respectively. Around 4300 jobs are simulated. The figures show the jobs' distribution over the change of I/O and execution time. The I/O time

of a job is selected as the maximum of the job's I/O nodes I/O time during the job's simulation.

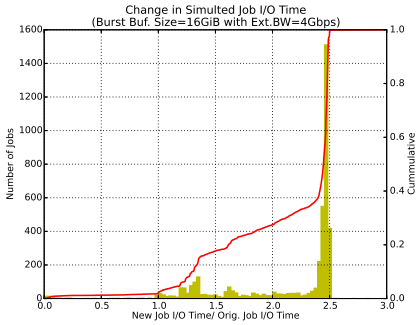


(a) Job I/O time

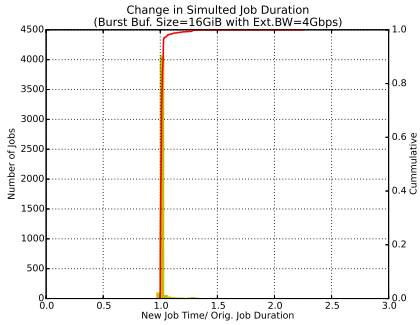


(b) Job execution time

FIGURE 5.16: Change of job I/O and execution time using burst buffers of size 64GiB and an external bandwidth of 1Gbps.



(a) Job I/O time



(b) Job execution time

FIGURE 5.17: Change of job I/O and execution time using burst buffers of size 16GiB and an external bandwidth of 4Gbps.

Tab. 5.2 and Tab. 5.3, in relation to Fig. 5.16 and Fig. 5.17, clarify the effect of using burst buffers with reduced external bandwidth. While the job I/O time can increase by up to 10 times, the execution time of most jobs will only slightly change. However some jobs will experience relatively large slow downs. About 20% of simulated jobs have a slow down above 10% for a 64GiB burst buffer with 1Gbps external bandwidth. As the external bandwidth increases to 4Gbps with a 16GiB burst buffer, only 2% suffer from

Burst buffer size = 64GiB with Ext.Bw = 1Gbps		
	Ratio of I/O time (New I/O time / Orig. I/O time)	Ratio of job execution time (New job time / Orig. job time)
Average (standard deviation)	8.1 (2.7)	1.08 (0.2)
Maximum	10.0	8.47
Median	9.7	1.05
Ratio > 1.1	97%	20%

TABLE 5.2: Statistics on the change of job I/O and execution time using burst buffers of size 64GiB and an external bandwidth of 1Gbps.

Burst buffer size = 16GiB with Ext.Bw = 4Gbps		
	Ratio of I/O time (New I/O time / Orig. I/O time)	Ratio of job execution time (New job time / Orig. job time)
Average (standard deviation)	2.14(0.52)	1.01(0.03)
Maximum	2.96	2.25
Median	2.44	1.01
Ratio > 1.1	96%	2%

TABLE 5.3: Statistics on the change of job I/O and execution time using burst buffers of size 16GiB and an external bandwidth of 4Gbps.

a slow down of 10% or more. These jobs and the effect of the I/O delay on the overall system utilization have to be considered for the use of burst buffers with a reduced external bandwidth. Using more simulations and the I/O analysis it is possible to find the percentage of all applications running on a particular system that are effected by such I/O performance reduction. Additionally, by employing I/O analysis based on the I/O criteria these applications could be further analysed to decide on other I/O architectural changes that might benefit them.

5.6 Conclusions On Modelling System I/O

The purpose of the modelled I/O is to provide basic understanding of the I/O subsystem's components interaction and how the I/O behaviour reacts to changes in the I/O architecture. It also allows for increased control of the I/O system's parameters and investigate subtle changes to I/O behaviour. As shown, the complexity of the simulation highly depends on the available information on the individual components and the used input data. The verification and parameter fitting observes real application I/O behaviour to allow for valid conclusions on the reaction to the I/O architectural changes.

The decision on which components to simulate and the required detail of the simulation is linked to the purpose of the I/O model. The I/O architectural changes to simulate are driven by analysing the I/O behaviour of applications on the existing I/O system to determine the need for possible improvements.

As more I/O architecture modifications and configurations are suggested, the necessity of using simulations will grow. This rises from the need of understanding the effect these changes have on the I/O behaviour. While the simulation done here is by no means exhaustive, it paves the way for experimenting with more simulations of future I/O system architectures.

Chapter 6

Conclusion

Modern HPC systems offer substantial compute power to scientific applications, which grows at an exponential rate following Moore's law. These HPC systems are supported by large growing data storage infrastructures. The evolution of both compute power and data storage quantities has not been matched by an equal growth of the I/O capabilities available to modern HPC systems. This threatens the gain achieved by scientific applications using such large scale systems. To combat this threat, many suggestions for improving I/O performance are made. These could require costly modifications to HPC systems and I/O systems. In some cases the scientific applications have to be modified as well. The benefit from using many future I/O architectures depends on the I/O behaviour exhibited by the scientific applications. This study attempts establishing both the need and possible methods for investigating the I/O behaviour of applications as observed on current modern HPC systems.

The complexity of I/O systems could lead to an overwhelming number of measurable quantities. The I/O criteria described provides an analysis map that can be used to reduce quantities that require evaluation. This is achieved by selecting quantities that are relevant, easy to measure and applicable to a wide range of modern I/O systems. The I/O criteria is designed to be usable for analysis of different data sets, that were collected using different methods and on different I/O stack layers. They also provide the possibility of analysing individual applications or perform a mass I/O behaviour analysis and comparison of a large number of applications. This is provided by a well formulated and condensed group of I/O criteria that are selected with careful consideration of

the functionalities of modern I/O systems. The I/O criteria offer a starting point for investigating the overall application I/O behaviour. They can be extended, reduced or modified to fit other I/O architectures or to further evaluate specific I/O behaviours.

To establish the usability of the I/O criteria and analyse the I/O behaviour of applications on modern HPC systems, a large set of I/O measurements were analysed. The I/O measurements were logged GPFS I/O counters on the I/O nodes of a peta-scale modern HPC system, namely JUGENE.

Analysing the large data quantity of I/O measurements given has demonstrated the benefit from using well defined I/O criteria. These allow a standardized repetition and comparison of such an analysis for different HPC and I/O system combinations or for different methods of I/O measurements on possibly different I/O stack layers. The I/O criteria were updated by understanding the analysis results. This improves on the I/O criteria or filters out quantities that do not further the description of application and system I/O behaviour. Such cyclic feedback from I/O criteria to their analysis and back, provides a refining process for the I/O criteria.

The analysis of the GPFS I/O logs demonstrates the effect the I/O measurement technique has on the evaluation of I/O criteria. I/O measurement methods vary in their resolution, the I/O stack layer and the collected quantities. These variations can limit the I/O criteria evaluation's accuracy. Missing quantities, such as temporal or spatial information could restrict the number of I/O criteria that can be evaluated.

The analysis of the GPFS I/O logs has resulted in evaluating many of the I/O criteria of over 166×10^3 jobs. The I/O behaviour exhibited by these jobs is observed to vary widely. Given the overall capacity of the storage system supporting JUGENE, the transient data hitting the external storage system was indicated by the analysis to be relatively small. Most jobs were found to have a low I/O intensity. Small I/O requests are found to dominate the I/O, more so for write than for read. The analysis of the GPFS I/O logs demonstrated the bursty I/O behaviour of analysed jobs. For many jobs the real time parallel I/O on the I/O nodes was observed to be relatively low. These analysis results among others demonstrates the need for considering and experimenting with many different I/O architectures. Various I/O improvements could cater to subsets of different I/O behaviours exhibited by the analysed jobs. Many jobs

and their applications might merit using other I/O measuring techniques to evaluate missing I/O criteria and further investigate their I/O behaviour.

The target of the analysis was to give an overview of the I/O behaviour of applications on modern I/O systems. While evaluating the I/O criteria for the analysed jobs, an effort was made to pair some observations to possible I/O architecture changes. As the target of the analysis and the analysed system changes, so would the suggested future I/O architectures change. This is a result of the different reactions various I/O systems have to different I/O behaviour. Supercomputing centers are therefore urged to perform such analysis when considering future I/O architectures or different configuration to their existing I/O system. Emphasis should then be made on the I/O criteria paired with a suitable I/O measuring technique, that would give the most insight into the specific desired I/O changes.

Testing new I/O architectures and configuring an I/O system can be a complex and costly procedure. The I/O criteria used for the analysis of I/O behaviour can suggest which I/O configurations can yield better I/O performance. Modelling the I/O of an existing I/O subsystem can complement the I/O analysis process, providing more information on the I/O behaviour. It also provides control over I/O system parameters.

An I/O model of the JUGENE was created and verified using the GPFS I/O logs. The details of the I/O model's components was shown to depend on both the available information on internal I/O system operations and the input data used for verification. By carefully adapting the I/O model it was demonstrated that future I/O architectures can be modelled and simulated. Comparing the original I/O model to the changed model provides basic understanding of the effect I/O architectural changes have on the I/O behaviour. Due to the observed bursty I/O behaviour of analysed jobs, the effect of burst buffers on the I/O performance is modelled and simulated. Creating a more accurate comparison between original I/O model and burst buffers, required carefully limiting the differences between the two models and their simulations. Many I/O system and application I/O optimizations target improved I/O performance. The simulation here showed that adding burst buffers with reduced external bandwidth can result in prolonged I/O time for simulated jobs. However, cost is an additional factor to consider for improving an I/O system and selecting appropriate configurations. Since the overall execution time of simulated jobs was not increased as much, burst buffers remain a viable

I/O configuration choice to reduce overall system cost. In this case some applications might suffer from reduced performance due to their high I/O intensity. These can be further analysed and possibly catered for with other I/O improvements.

The number of I/O system configurations, modifications and improvements will increases as the scientific applications' need for improving I/O increases. Prior to implementing these changes the I/O behaviour of the scientific applications need thorough analysis. The conclusions from the analysis can be complemented with simulating the I/O system and the suggested future architectures. Such process will allow supercomputing centers to build better performing and more cost effective future I/O systems that can better support the scientific community.

6.1 Future Work

The I/O criteria are designed to reflect the main quantities representing the I/O behaviour. As the necessity rises more criteria can be added, while the existing definitions can be refined. This can be done to observe new I/O behaviours or to analyse specific I/O architectures. The I/O criteria can also be further edited to reflect the effect of the I/O measuring technique used in the analysis process.

The I/O criteria or a refined version can be used to analyse the I/O behaviour across different systems and different I/O measuring techniques. The I/O behaviour of jobs on JUGENE can then be compared to I/O behaviour on newer systems. The reaction of jobs to larger computation power, more I/O bandwidth and more available storage can be investigated. Using different data sets from different I/O measuring techniques allows exploring the impact different measuring methods have on the analysis and it's observations. These can then be compared according to their benefits and cost effectiveness. Applications can be selected for further analysis and their evaluated I/O criteria compared across various I/O measuring techniques. The analysis process as a whole can be automated giving periodic information of system or application I/O behaviour to system administrators or application developers.

The I/O model investigated can be extended and further new I/O architectures and I/O optimizations can be modelled. More details can be added to the I/O model by using different input data for I/O model verification and finding more information on

system component's inner operations. The impact these details have on the I/O model can then be investigated. By simultaneously changing a set of parameters in a real I/O system and it's I/O model the effectiveness of I/O simulations in determining parameters for I/O system configurations can be evaluated. Other I/O subsystems with radically different I/O architectures can be modelled and their I/O behaviour investigated.

Appendix A

I/O Criteria - Category 4: Application Details

The introduced I/O criteria in Chp. 3 focused on measurable quantities. Application details describe application specific information. These in general require closer analysis of the application's implementation. The challenge for the application details is to determine the I/O criteria which could require manual code analysis. Some information can only be acquired from the application developers directly. As a result, some of the application details are more likely to be evaluated using questionnaires directed at HPC system users. The following are short notes on some possible application details worthy of investigating.

Classification 4.1 Problem size dependency

The problem size dependency can be considered the data volume as a function of application input parameters. These should be reduced to the input parameters that dictate the problem size.

Classification 4.2 Library dependencies and I/O interface used

Listing used I/O libraries and the I/O interface can help determine overall application benefit from specific I/O improvements. For example, only applications using MPI-IO can benefit from improving collective I/O in the MPI-IO interface [26] [27]. The number of applications using a specific I/O library and/or I/O interface determine the overall benefit from improving these libraries and/or interfaces.

Classification 4.3 Options and willingness to change I/O routines

Application developers might not always be willing to change or improve their implemented I/O routines. Reasons for this can vary. The I/O routines could have been implemented and optimized for a specific platform. In some cases, changing community used codes might be restricted. System, I/O library or I/O interface improvements are more suitable to optimize the I/O of applications where developers are not able or willing to change I/O routines.

Classification 4.4 I/O purpose classification

According to [15], the purpose of I/O can be classified into compulsory, checkpoint and out-of-core. Compulsory I/O are unavoidable I/O operations such as reading initialisation files, reading input data sets or writing output data sets. Checkpoints are I/O operations performed for the purpose of saving application progress or restarting the application at a given point. Out-of-core are I/O operations performed due to the limited primary memory, as a result the application is forced to swap data with the storage system. To these an additional class can be added, namely workflow I/O. For this class the I/O is a result of data being transferred from one component to another through storing it in the storage system.

The purpose of I/O might change the approach for improving the I/O performance. For example, compulsory I/O cannot be eliminated but only optimized, while out-of-core I/O can be reduced by increasing the primary memory. Meanwhile, many I/O optimizations target improved applications' checkpointing performance.

Classification 4.5 I/O task dependency

I/O tasks can be defined as application tasks that are formed of I/O operations and no (or very little) computation. The remaining application tasks can be either dependent or independent of the I/O tasks. According to [33] the dependency of application to I/O tasks can be grouped into fully coupled, decoupled in space and decoupled in time. When an application is fully dependent on the I/O tasks it is considered fully coupled. In this case I/O will be performed synchronously and on the source node [33]. Independent application from I/O task can either be decoupled in space or time. When decoupled in space the application I/O can be performed on a separate node such as an I/O node. Decoupled in time means that I/O can be performed asynchronously while the application proceeds with other tasks [33].

Appendix B

I/O Model Parameter Fitting Using An I/O Benchmark

Parameter fitting using the GPFS I/O logs is described in Sec 5.4.1. Another method for parameter fitting uses an I/O micro-benchmark, the cycle for which can be seen in Fig. B.1 and is described here.

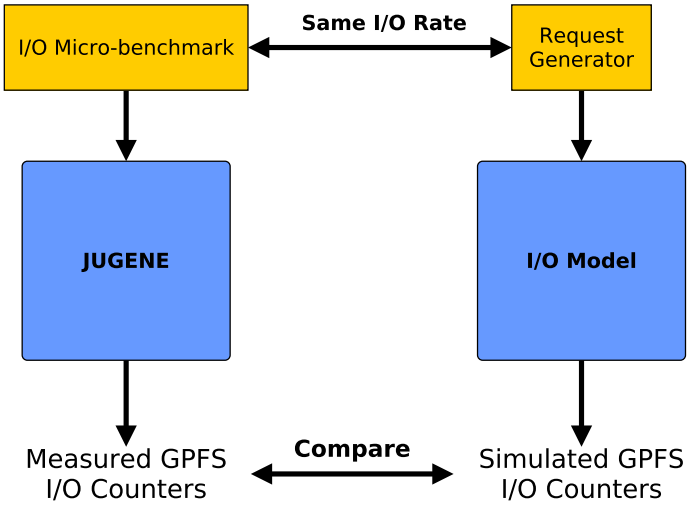


FIGURE B.1: JUGENE I/O model verification cycle using an I/O micro-benchmark.

In this method for establishing the I/O model parameters, a single I/O node and the connected compute nodes are simulated. The request generator, generates I/O requests

at the rate of an I/O micro-benchmark, which is the same as the one described in Sec. 4.4.1. The simulation is repeated multiple times, while changing the bandwidth available between the GPFS server and the disk. This allows finding an appropriate value for the disk bandwidth using a single I/O node.

Fig. B.2 shows an example of changing the bandwidth for parameter fitting one of the I/O benchmark runs, specifically the use of POSIX-I/O with task-local files and request size of 1024KiB[1]. The red dashed line represents the I/O behaviour registered by the GPFS I/O logs, while the other lines show results of simulating with different disk bandwidth. From the figure it appears that an I/O node can drive the disks in the I/O model at 2.3Gbps for write operations which resembles the value described in [5]. This value for bandwidth is acquired by observation from Fig. B.2. To achieve a more accurate bandwidth, the simulation has to be repeated several times for different benchmarks runs with different input parameters. The resulting data from all simulation runs has then to undergo a linear regression to find the best fitting value for disk bandwidth. This process has to be repeated for both write and read.

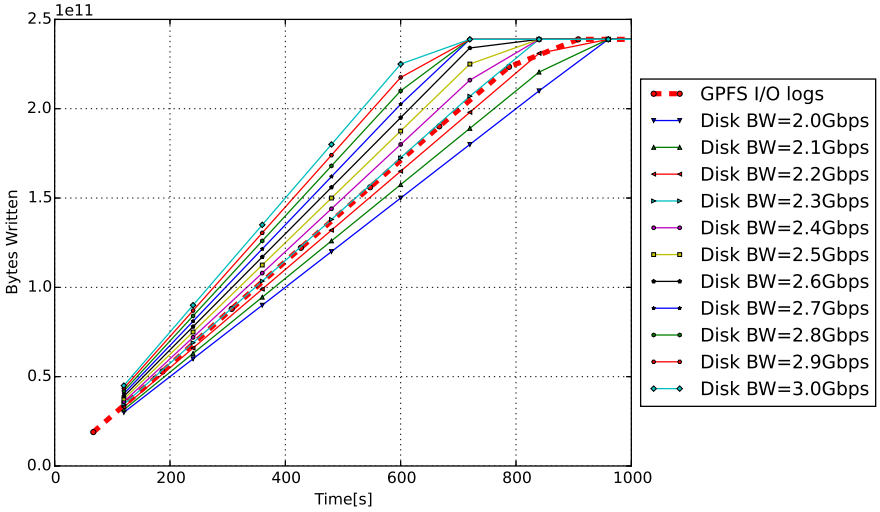


FIGURE B.2: Example of I/O model parameter fitting using I/O benchmark for write

Observed deviation of the simulated bandwidth in this method from the available peak performance of JUGENE, can be due to other applications running at the same time.

[1]See Sec. 4.4.1 for micro-benchmark description

In other words, during collection of the GPFS I/O logs for the I/O benchmark, the remaining I/O nodes were also creating I/O requests, thereby consuming I/O resources that could have led to better results for the I/O benchmark. To confirm these parameter values for all I/O nodes in the simulation, this verification method has to be repeated for the complete machine or while the remaining machine is empty (eg. after maintenance).

Bibliography

[1] 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 *Proceedings of the 2012 IEEE Conference on Massive Data Storage*, 2012.

[2] P. C. Roth, "Characterizing the I/O behavior of scientific applications on the cray XT," in *Proceedings of the 2Nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*, PDSW '07, (New York, NY, USA), pp. 50–55, ACM, 2007.

[3] A. Jackson, F. Reid, J. Hein, A. Soba, and X. Saez, "High performance I/O," in *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, pp. 349–356, February 2011.

[4] R. L. Cloud, "Problems in modern high performance parallel I/O systems," *CoRR*, vol. abs/1109.0742, 2011.

[5] W. Frings and M. Hennecke, "A system level view of petascale I/O on IBM Blue Gene/P.," *Computer Science - R&D*, vol. 26, no. 3-4, pp. 275–283, 2011.

[6] E. L. Miller, "Towards scalable benchmarks for mass storage systems," *5th NASA Goddard Conference on Mass Storage Systems and Technologies, College Park, MD*, pp. 515–527, September 1996.

[7] P. Chen and D. Patterson, "Storage performance-metrics and benchmarks," *Proceedings of the IEEE*, vol. 81, pp. 1151–1165, August 1993.

[8] H. Shan, K. Antypas, and J. Shalf, "Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference*, pp. 1–12, November 2008.

[9] J. Borrill, L. Oliker, J. Shalf, and H. Shan, "Investigation of leading HPC I/O performance using a scientific-application derived benchmark," in *Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference*, pp. 1–12, November 2007.

[10] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale I/O workloads," in *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference*, pp. 1–10, August 2009.

[11] D. Kotz and N. Nieuwejaar, "File-system workload on a scientific multiprocessor," *Parallel Distributed Technology: Systems Applications, IEEE*, vol. 3, pp. 51–60, Spring 1995.

[12] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," in *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium*, pp. 1–14, May 2011.

[13] M. Wiedemann, J. Kunkel, M. Zimmer, T. Ludwig, M. Resch, T. Bönisch, X. Wang, A. Chut, A. Aguilera, W. Nagel, M. Kluge, and H. Mickler, "Towards I/O analysis of HPC systems and a generic architecture to collect access patterns," *Computer Science - Research and Development*, vol. 28, no. 2-3, pp. 241–251, 2013.

[14] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao, "A multiplatform study of I/O behavior on petascale supercomputers," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, (New York, NY, USA), pp. 33–44, ACM, 2015.

[15] E. Smirni and D. A. Reed, "Lessons from characterizing input/output bahavior of parallel scientific applications," *INTERNATIONAL JOURNAL*, vol. 33, pp. 27–44, 1998.

[16] P. Crandall, R. Aydt, A. Chien, and D. Reed, "Input/output characteristics of scalable parallel applications," in *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*, pp. 59–59, 1995.

[17] B. Feng, N. Liu, S. He, and X.-H. Sun, "HPIS3: towards a high-performance simulator for hybrid parallel I/O and storage systems," in *Proceedings of the 9th Parallel*

Data Storage Workshop, PDSW '14, (Piscataway, NJ, USA), pp. 37–42, IEEE Press, 2014.

[18] A. Deuzeman, S. Reker, and C. Urbach, "Lemon: an MPI parallel I/O library for data encapsulation using LIME," *Computer Physics Communications*, vol. 183, no. 6, pp. 1321–1335, 2012.

[19] R. Latham, C. Daley, W. keng Liao, K. Gao, R. Ross, A. Dubey, and A. Choudhary, "A case study for scientific I/O: improving the FLASH astrophysics code," *Computational Science and Discovery*, vol. 5, no. 1, p. 015001, 2012.

[20] N. Ali, P. H. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. B. Ross, L. Ward, and P. Sadayappan, "Scalable I/O forwarding framework for high-performance computing systems," in *Name: Proceedings of the 2009 IEEE International Conference on Cluster Computing*, (New Orleans, LA, USA), 08/2009 2009.

[21] V. Vishwanath, M. Hereld, K. Iskra, D. Kimpe, V. Morozov, M. Papka, R. Ross, and K. Yoshii, "Accelerating I/O forwarding in IBM Blue Gene/P systems," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference*, pp. 1–10, November 2010.

[22] S. El Sayed, S. Graf, M. Hennecke, D. Pleiter, G. Schwarz, H. Schick, and M. Stephan, "Using GPFS to manage NVRAM-based storage cache," in *Supercomputing* (J. Kunkel, T. Ludwig, and H. Meuer, eds.), vol. 7905 of *Lecture Notes in Computer Science*, pp. 435–446, Springer Berlin Heidelberg, 2013.

[23] Y. Chen, S. Byna, X.-H. Sun, R. Thakur, and W. Gropp, "Hiding I/O latency with pre-execution prefetching for parallel applications," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, (Piscataway, NJ, USA), pp. 40:1–40:10, IEEE Press, 2008.

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

[25] B. G. Fitch, A. Rayshubskiy, M. C. Pitman, T. J. C. Ward, and R. S. Germain, "Using the active storage fabrics model to address petascale storage challenges,"

in *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, (New York, NY, USA), pp. 47–54, ACM, 2009.

[26] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in ROMIO," in *Frontiers of Massively Parallel Computation, 1999. Frontiers '99. The Seventh Symposium*, pp. 182–189, February 1999.

[27] K. Coloma, A. Ching, A. Choudhary, W.-K. Liao, R. Ross, R. Thakur, and L. Ward, "A new flexible MPI collective I/O implementation," in *Cluster Computing, 2006 IEEE International Conference*, pp. 1–10, September 2006.

[28] Y. Chen, X.-H. Sun, R. Thakur, H. Song, and H. Jin, "Improving parallel I/O performance with data layout awareness," in *Cluster Computing (CLUSTER), 2010 IEEE International Conference*, pp. 302–311, September 2010.

[29] J. Kunkel, M. Zimmer, and E. Betke, "Predicting performance of non-contiguous I/O with machine learning," in *High Performance Computing* (J. M. Kunkel and T. Ludwig, eds.), vol. 9137 of *Lecture Notes in Computer Science*, pp. 257–273, Springer International Publishing, 2015.

[30] F. Schmuck and R. Haskin, "GPFS: a shared-disk file system for large computing clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, (Berkeley, CA, USA), USENIX Association, 2002.

[31] "GPFS version 3.5 (2013) advanced administration guide," *IBM publication*, June 2013.

[32] "GPFS version 3.5 administration and programming reference," *IBM publication*, no. SA23-2221-08.

[33] M. Payne, P. Widener, M. Wolf, H. Abbasi, S. McManus, P. G. Bridges, and K. Schwan, "Exploiting latent I/O asynchrony in petascale science applications," in *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, (Washington, DC, USA), pp. 410–411, IEEE Computer Society, 2008.

[34] A. Purakayastha, C. Ellis, D. Kotz, N. Nieuwejaar, and M. Best, "Characterizing parallel file-access patterns on a large-scale multiprocessor," in *Parallel Processing Symposium, 1995. Proceedings., 9th International*, pp. 165–172, April 1995.

[35] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Ellis, and M. Best, "File-access characteristics of parallel scientific workloads," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 7, pp. 1075–1089, October 1996.

[36] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS)," in *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, CLADE '08, (New York, NY, USA), pp. 15–24, ACM, 2008.

[37] S. El Sayed, "Analysis and optimization of storage IO in distributed and massive parallel high performance systems," masterarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, November 2011.

[38] W. Frings, F. Wolf, and V. Petkov, "Scalable massively parallel I/O to task-local files," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, (New York, NY, USA), pp. 17:1–17:11, ACM, 2009.

[39] R. Ge, X. Feng, S. Subramanya, and X.-H. Sun, "Characterizing energy efficiency of I/O intensive parallel applications on power-aware clusters," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium*, pp. 1–8, April 2010.

[40] T. Madhyastha and D. Reed, "Learning to classify parallel input/output access patterns," *Parallel and Distributed Systems, IEEE Transactions*, vol. 13, pp. 802–813, August 2002.

[41] C. Muelder, C. Sigovan, K.-L. Ma, J. Cope, S. Lang, K. Iskra, P. Beckman, and R. Ross, "Visual analysis of I/O system behavior for high-end computing," in *Proceedings of the Third International Workshop on Large-scale System and Application Performance*, LSAP '11, (New York, NY, USA), pp. 19–26, ACM, 2011.

[42] S. J. Kim, Y. Zhang, S. W. Son, R. Prabhakar, M. Kandemir, C. Patrick, W.-k. Liao, and A. Choudhary, "Automated tracing of I/O stack," in *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface*, EuroMPI'10, (Berlin, Heidelberg), pp. 72–81, Springer-Verlag, 2010.

[43] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. Müller, and W. Nagel, "The vampir performance analysis tool-set," in *Tools for High Performance Computing* (M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, eds.), pp. 139–155, Springer Berlin Heidelberg, 2008.

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

[45] T. Ilsche, J. Schuchart, J. Cope, D. Kimpe, T. Jones, A. Knüpfer, K. Iskra, R. Ross, W. Nagel, and S. Poole, "Optimizing I/O forwarding techniques for extreme-scale event tracing," *Cluster Computing*, vol. 17, no. 1, pp. 1–18, 2014.

[46] M. Kluge, A. Knupfer, and W. Nagel, "Efficient pattern based I/O analysis of parallel programs," in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference*, pp. 144–153, September 2010.

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

[48] Y. Yin, S. Byna, H. Song, X.-H. Sun, and R. Thakur, "Boosting application-specific parallel I/O optimization using IOSIG," in *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium*, pp. 196–203, May 2012.

[49] M. Ester, H. peter Kriegel, J. S, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *In Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining (KDD-96)*, pp. 226–231, AAAI Press, 1996.

[50] N. Liu, C. Carothers, J. Cope, P. Carns, R. Ross, A. Crume, and C. Maltzahn, "Modeling a leadership-scale storage system," in *In Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics*, 2011.

[51] A. Varga and R. Hornig, "An overview of the OMNeT++ simulation environment," in *Simutools '08: Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, (ICST,

Brussels, Belgium, Belgium), pp. 1–10, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

Band / Volume 25
Numerical simulation of gas-induced orbital decay of binary systems in young clusters
A. C. Korntreff (2014), 98 pp
ISBN: 978-3-89336-979-9
URN: urn:nbn:de:0001-2014072202

Band / Volume 26
UNICORE Summit 2014
Proceedings, 24th June 2014 | Leipzig, Germany
edited by V. Huber, R. Müller-Pfefferkorn, M. Romberg (2014), iii, 60 pp
ISBN: 978-3-95806-004-3
URN: urn:nbn:de:0001-2014111408

Band / Volume 27
Automatische Erfassung präziser Trajektorien in Personenströmen hoher Dichte
M. Boltes (2015), xii, 308 pp
ISBN: 978-3-95806-025-8
URN: urn:nbn:de:0001-2015011609

Band / Volume 28
Computational Trends in Solvation and Transport in Liquids
edited by G. Sutmann, J. Grotendorst, G. Gompper, D. Marx (2015)
ISBN: 978-3-95806-030-2
URN: urn:nbn:de:0001-2015020300

Band / Volume 29
Computer simulation of pedestrian dynamics at high densities
C. Eilhardt (2015), viii, 142 pp
ISBN: 978-3-95806-032-6
URN: urn:nbn:de:0001-2015020502

Band / Volume 30
Efficient Task-Local I/O Operations of Massively Parallel Applications
W. Frings (2016), xiv, 140 pp
ISBN: 978-3-95806-152-1
URN: urn:nbn:de:0001-2016062000

Band / Volume 31
A study on buoyancy-driven flows: Using particle image velocimetry for validating the Fire Dynamics Simulator
by A. Meunders (2016), xxi, 150 pp
ISBN: 978-3-95806-173-6
URN: urn:nbn:de:0001-2016091517

Schriften des Forschungszentrums Jülich
IAS Series

Band / Volume 32
Methoden für die Bemessung der Leistungsfähigkeit multidirektional genutzter Fußverkehrsanlagen
S. Holl (2016), xii, 170 pp
ISBN: 978-3-95806-191-0
URN: urn:nbn:de:0001-2016120103

Band / Volume 33
JSC Guest Student Programme Proceedings 2016
edited by I. Kabadshow (2017), iii, 191 pp
ISBN: 978-3-95806-225-2
URN: urn:nbn:de:0001-2017032106

Band / Volume 34
Multivariate Methods for Life Safety Analysis in Case of Fire
B. Schröder (2017), x, 222 pp
ISBN: 978-3-95806-254-2
URN: urn:nbn:de:0001-2017081810

Band / Volume 35
Understanding the formation of wait states in one-sided communication
M.-A. Hermanns (2018), xiv, 144 pp
ISBN: 978-3-95806-297-9
URN: urn:nbn:de:0001-2018012504

Band / Volume 36
A multigrid perspective on the parallel full approximation scheme in space and time
D. Moser (2018), vi, 131 pp
ISBN: 978-3-95806-315-0
URN: urn:nbn:de:0001-2018031401

Band / Volume 37
Analysis of I/O Requirements of Scientific Applications
S. El Sayed Mohamed (2018), XV, 199 pp
ISBN: 978-3-95806-344-0
URN: urn:nbn:de:0001-2018071801

Weitere ***Schriften des Verlags im Forschungszentrum Jülich*** unter
http://wwwzb1.fz-juelich.de/verlagextern1/index.asp

IAS Series
Band / Volume 37
ISBN 978-3-95806-344-0

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
Forschungszentrum