Parallel I/O in Low Latency Storage Systems

by
Raafat Feki

A dissertation submitted to the Department of Computer Science,

College of Natural Sciences and Mathematics

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in Computer Science

Chair of Committee: Jaspal Subhlok

Co-Chair of Committee: Edgar Gabriel

Committee Member: Shishir Shah

Committee Member: Panruo Wu

Committee Member: Rodolfo Ostilla Monico

University of Houston
May 2022

DEDICATION

In the name of Allah, the Entirely Merciful, the Especially Merciful. First and foremost, I would like to praise and thank the one God (Allah), the almighty, who has granted me countless blessing and immense strength and courage to accomplish this dissertation.

I would like to dedicate this work to my father soul, Salah Feki in the hope that he would be proud of me and to my mother Semia Jallouli who had dedicated her life to make me the man I am today. I would like to thank you for the endless love and encouragement you surrounded me during all these years.

My deepest gratitude goes to my dear brother Saber, who first ignited the idea of pursing the PhD degree and encouraged me to accomplish this work, and to his wife Malek Smaoui and children Emna and Hatem for their support and love. I would like to extend my sincere thanks and appreciation to my sister Asma who supported me during all my academic journey as well as her husband Hamdi Kchaw and children Julia and Jad for their immense love.

I would like to express my deepest appreciation to my second family in Houston and all my friends in Europe and back home in Tunisia for always being there for me.

Last but not least, I'm deeply grateful to my beloved wife Emna Gargouri for being my inspiration and motivation through the darkest moments and for her unlimited support, love and patience. You illuminated my life.

"Surely, Allah does not change what is in a people until they change what is in themselves."

Al-Quran

Chapter 13, Verse 11

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to to my advisor Dr. Edgar Gabriel for his invaluable advises, continuous support, and patience during my PhD study. His immense knowledge, plentiful experience and contagious enthusiasm towards research have been a motivation for me in all the time of my academic research and daily life. Words cannot express how thankful I am to have you as advisor and mentor.

I would also like to thank the chair of committee Dr. Jaspal Subhlok for agreeing to serve as my advisor during the last semester of my PhD study and for his broad support and help during this critical period. I would like to extend my sincere thanks and appreciation to the other committee members, Dr. Shishir Shah, Dr. Panruo Wu and Dr. Rodolfo Ostilla Monico for their valuable inputs and insightful discussions during my PhD proposal defense that was of immense help in planning and structuring my work. I would like to thank my colleagues at the Parallel Software Technologies Laboratory, Siddhesh Pratap Singh, Tanvir Rahman Pavel and particularly Mohammad Rajiur Rahman for their collaboration, stimulating discussions and all the fun we had in the last five years.

Finally, special thanks to Prof. Mohamed Said Ouerghi whom I consider the most influential professor in my Academic Journey to engineering and computer science and to my supervisor at Kalray, Dr. Marta Rybczynska, who first introduced me to the HPC field.

ABSTRACT

The high performance computing (HPC) systems have experienced a momentous evolution during the last two last decades. Embedding thousands of cores, with very powerful processing capabilities, today's supercomputers can process a tremendous amount of data in a matter of seconds. However, the evolution of the storage systems has not kept pace with it, which has led to a huge gap between I/O performance and processing performance. Therefore, computer scientists had mainly focused on improving the I/O performance by providing software solutions, e.g., collective I/O and asynchronous I/O, that constitute the foundation of parallel I/O. They proposed several algorithms and techniques in order to hide the I/O overhead and improve the overall performance of storage systems by targeting the bandwidth and the capacity. The Message Passing Interface (MPI) has been the most recognized parallel programming paradigm for large scale parallel applications. Starting from version 2 of the MPI specification, the standard has introduced an interface for parallel file I/O support, referred to as MPI-I/O. By extending the MPI concepts to file I/O operations, the programming model becomes more complete and offers more options for developers to exploit the performance benefits of parallel I/O.

While reaching the new era of Exascale computing, multiple innovative technologies has risen to the surface opening the door toward a balanced HPC ecosystem that incorporates low latency storage systems. Nevertheless, this evolution has also posed new challenges regarding parallel I/O optimizations. Whereas hardware latency was the main source of I/O overhead, software latency was usually negligible. However, in low latency storage systems, the equation changes since the former would be reduced to the same level as the latter. Therefore, we aim in this dissertation at solving the new equation by providing multiple optimization techniques to the existing parallel I/O solutions within MPI I/O context. In particular, this dissertation targets the communication overhead of collective I/O operations, the computation phase of complex access patterns within independent I/O operations and the file locking overhead of the Lustre parallel file system.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

The current standard in high performance computing (HPC) for the assessment of supercomputers performance capabilities is the rate of floating point operations that can be performed by a system in one second, usually referred to as FLOPS i.e floating-point operations per second. However, many experts of the field claim that this performance metric is not anymore harmonious with the changing pace of the HPC systems as it only reflects their processing speed [53].

Today's supercomputers are architecturally very complex consisting of thousands of compute nodes connected by a high-speed network interconnect, each having one or more multi-core processor with multiple threads per core. Their processing capabilities are exponentially growing in a manner that intensive calculations are not anymore the main cause of performance degradation but rather the increased amount of data that is produced. Such tremendous amount of data, measured in units of Petabytes and reaching the Exabytes scale, is producing an unprecedented I/O bottleneck that even high end computing systems often lack the file I/O performance required to manage it. The cost of I/O operations performed on parallel storage systems and that of the internal data transfer between the nodes have always been a source of overhead in parallel applications but it had never been a top priority like it is nowadays.

To cope with these challenges, most large scale parallel applications are using parallel programming models that incorporate parallel I/O solutions. MPI [41] is as of today still the most popular programming paradigm providing efficient inter-node communication, supporting all state-of-the-art network interconnects, and having a large software ecosystem. Starting from the second version of its specification, MPI incorporates interfaces for parallel file I/O, often referred to as MPI I/O. By extending the MPI concepts to file I/O operations, multiple processes can simultaneously access the same file to perform read and write operations. This shared-file access pattern propagated by MPI I/O reduces the number of files created and enhances the overall I/O performance. Yet, this performance is still sub-optimal due to hardware and software limitations.

1.1 Performance in Storage Systems

There are multiple factors that affect the performance of storage systems. We can categorize them into two main components that form a complete ecosystem:

1. Hardware component

   It mainly consists of 3 major elements that have direct impact on file I/O performance:

   (a) Storage devices

   They are the physical media on which we store or retrieve data. Whether it is a hard disk drive (HDD) or a flash memory device, e.g., solid-state drives (SSD) and PCI Express (PCIe) add-in cards, the characteristics of a storage device play the main role in defining the performance capabilities of the storage system in term of bandwidth, latency and capacity.

   (b) Central processing unit (CPU)

   The processing capability of CPUs has both direct and indirect impact on I/O performance. On the one hand, the CPU is responsible of processing the data, serving I/O demands of storage devices and initiating I/O operations. So it needs to create and dispatch enough data to exploit the full bandwidth of I/O buses. On the other hand, I/O operations are performed and controlled through a software stack that has heavy processing requirements. The capacity of the CPU to fulfill these requirements will hence indirectly affect I/O performance.

   (c) Network interconnect

   There are two logical types of network interconnects that are implied in I/O operations. The storage network that connects external storage devices to each other within a same network environment and the cluster network allowing the data transfer between compute nodes and I/O nodes within a cluster. Similarly to storage devices, the I/O performance is bounded to the network bandwidth and latency that are dependent on

the design of the network architecture to be deployed on the storage system and the choice of the network technology.

2. Software component

   This component incorporates all software layers needed for the control and scheduling of I/O operations starting from the operating system kernel to the end user level. Since we are interested in storage systems within parallel environments, we limit our scope to parallel I/O software stack, i.e., parallel I/O algorithms and parallel file systems.

These factors are so highly correlated to each other that optimal I/O performance cannot be achieved without setting up the right design to ensure a good balance between them. While storage systems are still bounded to the hardware specification such as the system bandwidth, we can rely on parallel algorithms to perform more efficient parallel I/O operations over multiple storage devices. However, a bad configuration of the parallel file system or a low cluster interconnect bandwidth can also overshadow the performance gain. With the current available technologies, we can overcome most of these dependency issues and find the right intersection point between these factors in order to create an optimal computing system. Nevertheless, no matter how much we can scale up the storage systems, there is still one major obstacle that has a negative impact on all of these factors: *"latency"*. Because of the limitation of storage devices commonly used in the deployed HPC systems, i.e., magnetic hard disk drives (HDD), we cannot cross over a certain performance threshold until we switch to low latency storage devices.

1.2 Parallel I/O Architecture

With the increasing need of parallel processing in different scientific fields e.g physics simulation, weather forecasting, the architecture of high performance distributed memory systems has evolved at an incredible pace during the last decades to meet these requirements. Typically, a parallel I/O architecture consists of two complementary components: a hardware component and a software component.

1.2.1 Hardware Component

An HPC system consist of thousands of nodes connected via a high performance network infrastructure. They can be either compute nodes dedicated for massively parallel processing, storage nodes mainly composed of storage devices or I/O nodes providing file I/O services and acting as an interface between the other two types of nodes. Those nodes are all connected by a cluster network and a storage network forming a mesh of high performance network interconnect allowing data distribution among them. As illustrated in Fig. 1, all of these components constitutes together a typical parallel I/O architecture in a high performance distributed memory system.

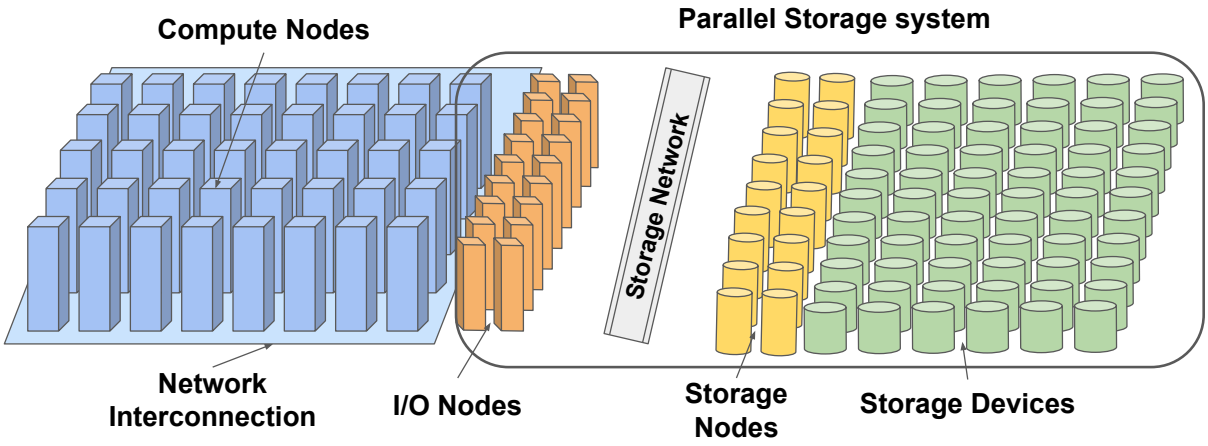

Figure 1: Typical parallel I/O architecture in a distributed memory system - hardware component.

1.2.2 Software Component

The parallel I/O software stack is a combination of multiple application programming interfaces (APIs) and software tools. It allows the users to exploit the HPC systems without taking care of the specificity of the hardware technology and enable full parallelism for file I/O operations. As depicted in Fig. 2, it follows a hierarchical architecture consisting of 4 different layers.

1. **Parallel File system**

   The parallel file system layer is considered as the most important layer of the software stack as it represents the junction point that relates the hardware component to the other software

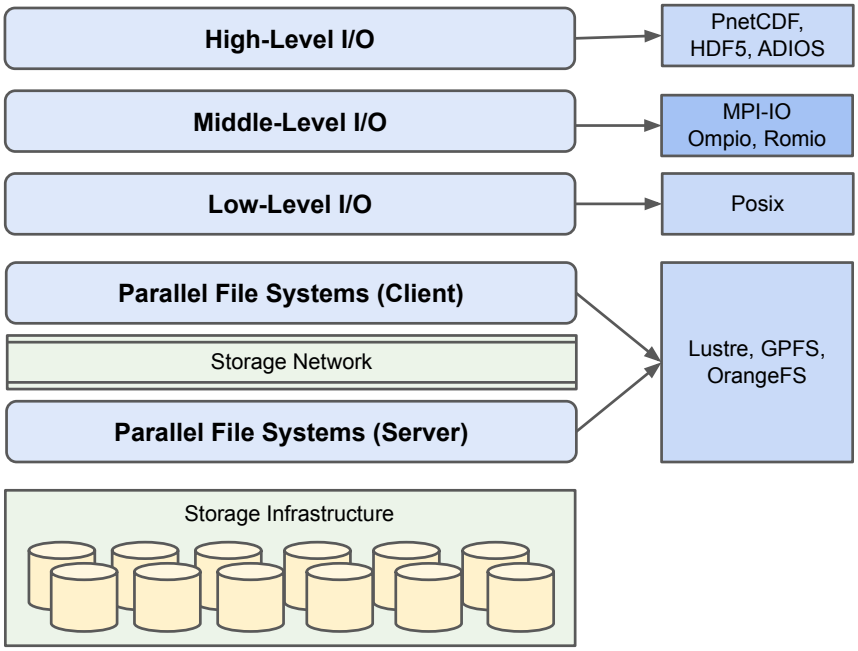

Figure 2: Typical parallel I/O architecture in a distributed memory system - software component.

layers. In general terms, a parallel file system is a software solution that defines an abstraction layer between the user and the storage system in order to store the data on multiple storage devices and thus allowing parallel I/O operations. Basically, it follows a client/server architecture in which the servers are running on top of the storage infrastructure (storage nodes) whereas the clients are running on the I/O nodes. They are both connected to each other through the storage network. In the next Section 1.4, we will dive into more details about parallel file systems and their mechanism.

2. **Low level I/O Library**

The second layer of the software stack encompasses the low level I/O libraries that defines the atomic function for file I/O operations, e.g., open/close, read/write. In spite of its limitation in HPC environments, the POSIX [3] interface remains the most commonly used library for low level I/O operations.

3. **Middle level I/O** Since the low level I/O libraries were originally designed for uni-processor systems, we cannot use the parallel I/O systems to good advantage by merely using them. Therefore, we are in need of a higher level interface that provides better mechanisms for parallel I/O operations and handles various optimization techniques. For distributed memory systems, MPI I/O remains the most dominant I/O middleware. It is an interface that incorporates a vast range of I/O functionalities specially dedicated for parallel I/O environments. It provides the user with multiple options like the ability to apply different optimizations techniques or to define complex I/O access patterns. As it is the foreground of our research projects, MPI I/O will be discussed in more detail in Section 2.1.

4. **High level I/O software**

   The last I/O layer consist of a set of a high level APIs positioned on top of the MPI I/O interface. It offers the scientific community a variety of specific functionalities that allows the manipulation of data in a more natural and convenient way. These libraries usually reduce the complexity of scientific problems and produce portable formats that facilitate data processing while taking advantage of the MPI I/O optimizations for parallel I/O operations. The most popular high level libraries are Parallel Hierarchical Data Formats (Parallel HDF5) [28], Parallel Network Common Data Form (PnetCDF) [30] and Adaptable IO System (ADIOS) [34].

1.3 Introduction to Parallel I/O

In a parallel environment, an application developer can follow different approaches in implementing file I/O operations. Basically, we distinguish two major types of file I/O: sequential I/O and parallel I/O. We begin with a brief introduction of sequential I/O then we focus the spotlight on parallel I/O and its different strategies.

1.3.1 Sequential I/O

In the early age of computers, we were mostly dealing with uni-processor systems. File systems at that time was designed in a manner that only one processor is responsible of accessing a single file

at once. Therefore, I/O operations were only performed sequentially in a per file basis, hence the notion of sequential I/O.

With the introduction of new different operating systems, there was a need to create a unified and portable programming interface compatible with all the systems. This has motivated the design of the Portable Operating System Interface (POSIX) [3]. It is an IEEE standard API that offers a complete programming interface, including an I/O support, for a variants of operating system. POSIX I/O is widely supported by most file systems and remains to this moment the standard API used for I/O operations. However, since it was primarily designed for a uni-processor system, POSIX was not suitable for parallel I/O accesses. In fact, in order to perform sequential I/O in a parallel environment, the data needs to be transferred from all nodes and aggregated into a single node that initiates the POSIX I/O functions. This approach led to I/O bottleneck within the node. In order to overcome these limitations, many research papers [63] [59] [44] proposed different extensions to the POSIX API to render it more efficient with parallel applications while others introduced new substitutes [1].

Nonetheless, since most existing applications are using POSIX as a standard API, switching to new APIs is straightforward and requires fundamental changes in the existing implementations. Besides, the new POSIX extensions has shown better performance on large scale parallel environments and offered several interesting options that can be combined with other parallel programming interface, e.g., MPI I/O.

1.3.2 Parallel I/O

Parallel I/O can be defined as the support of I/O operations in a multiprocessor system in a parallel environment. Unlike sequential I/O, multiple processors, and thus multiple nodes, can concurrently perform I/O operations on either a single file or multiple ones.

Consider a parallel application that performs massive calculations over several compute nodes. Each node is processing a part of the data and consequently producing new results that are temporary located in its local memory. We will focus in this example on write operations as they are more

complex and recurrent yet the same behavior applies to read operations. To write these results into a storage system, we can follow two different approaches illustrated in Fig. 3: Independent file I/O or Shared file I/O.

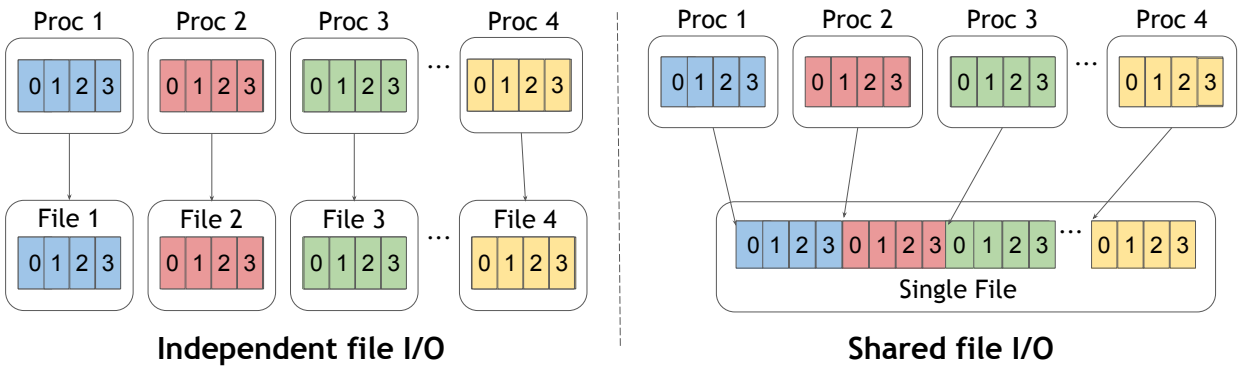

Figure 3: Independent file I/O and shared file I/O in a multiprocessor system.

Independent File I/O: In this approach, each individual process creates and opens its own file using the regular POSIX I/O API. Since it has access to its local results, it can individually initiate the write operation without the need to communicate with any other processes. Although this approach is very straightforward, easily implemented and leads to a good I/O bandwidth, it has many drawbacks. First and foremost, with the huge number of processes in today's supercomputers, it will produce thousands of files which would eventually overwhelm the metadata server of any parallel file system. Besides, in order to collect all these files into a single large file, we need to perform expensive processing that would wipe out the performance benefits. The same issue would rise if multiple processes have to read from a single file. In this case, the file has to be split into multiple smaller files matching the number of processes. Therefore, we cannot say that Independent I/O permanently eliminates the cost of data transfer between nodes but it rather delays it to a subsequent data rearrangement task.

Shared File I/O: Unlike the previous approach, there is only one single file shared between all processes. Depending on the parallel algorithm used, all processes or some of them will directly write their local data to the file at a specific offset. In fact, there are two possible scenarios in a shared file I/O approach. The first scenario is when each process writes its local data without having to exchange it with other processes. In this case, there is no need for data rearrangement since each process knows exactly where to write into the file. However, the processes might need to coordinate in order to guarantee the consistency of the file. This would generally occur when there are overlapped regions between the data chunks assigned to each process. The second scenario is when there is only a subset of processes that are designated to perform the I/O operations. In such a case, the data needs to be rearranged and redistributed among these processes before the actual I/O operation. Even though it seems to add more communication overhead, this approach, known as collective I/O operations, can significantly improve the overall I/O performance as we will explore with more details in the next sections. Consequently, shared file I/O has been the best adopted choice for parallel file I/O as it overcomes most of Independent I/O problems.

1.4 File Systems

The human abstraction of the data is based on the notion of files and directories. However, at the hardware level, the data is nothing but a sequence of bytes that can be stored and retrieved as blocks, hence the need for file systems.

A file system is basically a software solution for creating an abstraction layer defining specific data structures and implementing different methods for the management and organization of the data. It provides services such as file naming, setting of access rules and permissions, creation of file descriptors and pointers and others. John M. May summarized in his book [39] the main tasks a file system should provide:

1. Transfer of data between memory and storage devices efficiently.

2. Coordination of concurrent access by multiple processes to the same filename.

3. Allocation and reclamation of data blocks on storage devices.

4. Data recovery in case of system corruption.

The traditional file systems were designed to run on a uni-processor systems. Although multiple processes can access a file simultaneously, they are bounded to the file system semantics that impose restrictions on legitimate concurrent I/O operations which grantee sequential consistency. Because of these restrictions, the traditional file systems are ill-suited to multiprocessors distributed memory systems, hence the introduction of distributed and parallel file systems.

1.4.1 Distributed File Systems

A distributed file system is a network-based file system that allows multiple machines and thus processors to have access to a shared set of files using a network system. It consists of multiple storage nodes acting as servers. When a remote machine, called client, needs to access certain files, it logically mounts the directory tree of the server in its local directory hierarchy. The programs running on the client can act on the mounted file system as it is physically attached to the local system while most of I/O operations are actually done underneath the hood on the server side. The client sends the I/O requests to the sever which handles them and sends back the results. Many distributed file systems were commonly used such as DFS [29] (Distributed File System) and AFS [26] (Andrew File System) but NFS [48] (Network File system) remains the most prominent and widely used one. However, the main purpose of these file systems was not to enable data parallelism on storage systems but rather to share the data itself. Consequently, distributed file systems are not the best solution for parallel I/O, hence the need of parallel file systems.

1.4.2 Parallel File Systems

Despite its inadequacy for parallel I/O, distributed file systems were the first step toward a more complete file system. In order to enable storage parallelism and respond to the needs of parallel I/O software stack, the concept of parallel file systems raised to the surface.

A parallel file system is basically a distributed file system in which the data is divided into several blocks called 'stripes' and distributed over multiple storage devices. Since the data is split between several servers, a client can issue I/O requests in parallel to each server holding a part of the requested data. This design will provide a more efficient way to access a file concurrently by many processes and allow full parallelism, hence increasing overall system bandwidth. Depending on how a parallel file system manage the data and its structure, we can distinguish two different type of storage: *block-based* storage and *object-based* storage.

1. **Block-based parallel file systems**

   The main goal of parallel file systems is to stripe data over multiple servers. The trivial solution was to divide it into fixed-sized chunks of raw data called 'blocks' and identify them with their memory addresses. In this type of storage systems, the information describing the data, called metadata, is not easily accessible as in traditional file system that uses file-base storage technology. In traditional file system, since the data is defined as files, it is accessed by traversing the file system hierarchy and thus iterating over the file's metadata. However, in block storage system, the only identifier of the raw data is its address, hence we need another software layer for metadata tracking. In general, the operating system is usually in charge of the metadata management for block-based storage. The most popular Block-based parallel file system is IBM Spectrum Scale, also know as GPFS [49].

2. **Object-based parallel file systems**

   The lack of metadata in block-storage can lead to some extent to inefficiency in iterating big data in parallel applications. Therefore, object-based storage was designed to meet these needs. Instead of storing the data as raw blocks, we can combine it with its metadata and some additional information like a unique identifier and encapsulate them into one flat structure called 'object'. Even though there are many noted object-based parallel file system, e.g., Beegfs [23] and OrangeFS [4], Lustre [60] remains the most popular one.

Each of these two strategies have their benefits and drawbacks. The choice between them mainly

depends on the nature of the workload and the requirements from the deployed HPC system. For block-based storage, its strength and weakness is coming from the same source which is its way of handling metadata. Since the data is stored in raw format, block storage is more flexible and easy concerning data extraction and management. It offers a low-latency environment with a more predictable and consistent performance. Because of these benefits, block storage is more suitable for databases, virtual machines and transaction-based applications. However, when it comes to unstructured data, the lack of accessible metadata in block storage becomes a source of overhead and limits the scalability of the storage system.

Due to its search capabilities and unlimited scale, object-based storage would be the best option for these conditions. Since it can be seen as an independent entity that holds data and its own information within a same structure, an object can be easily searched and identified. Besides, the data can be altered and dynamically moved between storage devices with minimal interference of the metadata manager. Therefore, for whatever data sizes, the storage system is always capable of expanding by creating more objects as long as the physical storage devices can handle it. All these benefits make the object-based storage well-suited for extremely large data that requires more scalibility and faster data retrieval. It is also beneficial for data analytic applications that needs to have flexible access to the metadata. Yet, it still has one downside. In case of a dynamic data flow, many parts of data have to be re-written multiple times. Since an object is a discrete element, it has to be entirely modified for each I/O operation performed on a part of its content. As the cost of writing an object is a bit higher compared to raw data, recurrent I/O operations to a same data object would lead to a performance degradation.

1.5 Storage Systems

With the evolution of the CPU processing power and multi-core architectures, the HPC systems are capable of processing an unprecedented number of instructions, and hence data, in least amount of time that the current storage systems can keep pace with. Although these system have a very high bandwidth, the time they spend in accessing the actual storage devices known as *Latency* is

preventing them from reaching their fullest performance. In the past, the developers of parallel I/O algorithms tried to overcome the latency problem by changing the I/O access patterns to write/read bigger chunks of data. Nonetheless, as we are on the verge of the Exa-scale technology era and there is an increasing tendency toward exploiting high performance computing in different fields that have atypical I/O accesses, e.g., machine learning, scientific simulations and visualizations, the software solutions are not capable anymore of filling this performance gap. The necessity of deploying low latency storage systems has become more and more critical.

In this section, we start by explaining the main causes behind the high latency of the current storage systems. Then, we present the new technologies providing low latency environment. Finally, we will shed the light on its benefits and its impact on parallel I/O algorithms.

1.5.1 Hard Disk Drive (HDD) Based Systems

For decades and before the introduction of flash memory, all storage systems were based on hard disk drives. Hard drives basically consist of spinning platters, for data storage, that are coated with magnetic materials and disk arms with magnetic head on its tip. In order to write or read data, the disk arms needs to move to the right sector of the spinning platters. The rotation speed of the platters, quantified by its revolutions per minute (RPM), defines the access time needed to perform one I/O operation on the disk. This time delay is called latency. Whatever the speed it can reach, hard drives remains mechanical devices and their response time will be always in the order of milliseconds. During this time frame, the processor will be in an idle state for too many cycles waiting for data coming from the storage device.

1.5.2 Flash Memory (SSD)

Due to the limitation of the classical hard drives, new technology have been brought to the surface and started to conquer the industry of data storage. Solid-state drives (SSD), also known as flash memory, are made of integrated circuit based on either Nor or Nand logic gates. Unlike HDD, SSD does not have the same issues of a mechanical device which make them more resistant to shocks,

lighter, silent and faster. The latency has dropped from milliseconds to less than 100 microseconds which make them perfectly suitable for low latency storage systems. Yet, there are still multiple factors that might affect the overall performance.

1.5.3 Example of New Technologies

The main key of building a low latency storage system is in fact achieving the right balance between the processing capacity of CPUs and the characteristics of storage devices in term of throughput and latency. While the modern CPU has reached a tremendous performance pick, even the newest SSD technology are unable to match their evolution pace resulting in idle cycles on the CPU side. Although the high performance computing field is not totally ready for the transaction from classic to low latency storage systems, many supercomputer manufacturers like Intel and Cray are working on new technologies within this area.

Intel® Optane™ Technology

Even after the introduction of Nand-based SSDs, there is still a huge gap between bandwidth and latency in storage systems. This was the main motivation behind the creation of the Intel Optane™ SSD technology. It is a memory technology that can achieve an access speed that is very close to that of a regular DRAM while adding the non-volatility feature of persistent storage. It is manufactured using an undisclosed materiel different from the regular components of DRAM and flash memory. When compared to regular SSDs, Intel Optane™ SSD is much faster and shows unprecedented lower latency even with considerably heavy I/O loads [21]. Nonetheless, this performance improvement is less perceivable with the write operations than read operations. Since the first goal was to improve the overall performance of a system memory, this product was not meant as a replacement of the DRAM but rather as an extension to the main memory.

Using its newest technology, Intel proposed a new architecture for memory and storage by introducing its Intel Optane DC Persistent Memory. It is a memory module based on the Intel Optane™ SSD dedicated for data centers. It is directly plugged into DIMM slots and can be used as

an alternative for both main memory and storage memory. One interesting feature is that memory, unlike the traditional storage devices that use blocks for data access, is addressed at a byte level. Furthermore, it has a higher endurance that can reach up to 60 Drive Writes per Day. That means that its storage can be entirely overwritten more often than regular SSDs. This feature is considered as one of the most important requirements for data storage in HPC systems. This new module has shown lower latency even when compared to the Intel Optane™ SSD. The greatest benefit from using the Intel Optane DC Persistent Memory is that it allows applications to bypass the operating system storage stack by directly accessing it. This will completely remove the software overhead and thus reduce the overall latency of the system to its minimal value [22]. All of these characteristics make the Intel Optane™ DC Persistent Memory an excellent replacement for the Nand-based SSDs especially when deployed on a high performance system dedicated for low latency storage systems.

1.5.4 Impact on Parallel I/O

As explained in the previous sections, high latency is the main source of overhead in parallel I/O applications. Low latency storage will unlock the full power of super-computing by maintaining the optimal balance between the processing speed, the system bandwidth and the latency. While using the same hardware platforms, the overall system performance will reach higher rates by processing more data and performing more I/O operations compared to the previous infrastructure within the same time frame. Besides, the HPC systems will be able to respond to the needs of multiple parallel applications that requires multiple small I/O transactions on big data with random access patterns.

Based on the current available technologies and the ones coming in the future, low latency storage systems are expected to be deployed in most HPC systems. This will raise new challenges for the existing parallel I/O algorithms. Since the hardware latency will become merely equal to the software latency, this will open the door for new possible optimizations on the algorithms dealing with I/O distribution within parallel environments. In fact, parallel I/O algorithms usually consist of two parts: i) a pre-processing phase that involves all required setups needed for the subsequent

I/O calls such as the processing of data layouts and the redistribution of data over I/O nodes; and i) an I/O phase that incorporates the actual file access operations. The benefit of optimizing the first phase was always camouflaged by the long I/O operations. Therefore, the main focus was always on "what to do" in the pre-processing phase to reduce the cost of the second phase rather then "how to do" it efficiently.

And here raises the question, since the hardware latency in low latency storage systems are reduced to the same level as software latency, what further optimization we can apply to the latter to increase the performance of the existing parallel I/O algorithms.

1.6 Research Goals

In the last decades, the researchers had focused more on improving the overall bandwidth and capacity of storage systems thus the introduction of parallel I/O solutions, e.g., collective I/O and asynchronous I/O. On the one hand, these different techniques and algorithms tried to hide the I/O cost of the parallel applications and reduce the number of accesses to the storage system. On the other hand, they provided elegant methods to describe the data layout in memory and files in order to reduce its complexity.

Most of these software solutions has significantly improved the performance of I/O operations and offered more flexibility to developers. In fact, they usually needed to add some internal operations that result into extra processing requirements such as communication in collective I/O. Nevertheless, the overhead caused by these operations was always negligible when compared to the overall performance gain. This fact remains true as long as the HPC systems are using traditional storage systems. However, the HPC community is gradually switching toward low latency storage systems because of the new requirements stated in the previous sections,

These revolutionary changes has reshaped the current perspective of parallel I/O problems. Since we were always bounded to the physical limitations of storage devices, latency has always made I/O operations (write/read to disks) the main source of overhead. However, with the introduction of the new technologies constituting the core of low latency storage systems, we can reach

a new level of I/O performance by targeting all the layers of the I/O software stack. Since the computation overhead of parallel I/O software would become more perceivable, we would have a wider room for new optimizations of the current available solutions. This is in fact the centerpiece of this dissertation.

In a general sense, we are looking at achieving in this dissertation the following tasks:

- Improving collective I/O operations by reducing the communication overhead.
- Improving individual I/O operations by introducing a full multithreaded solution.
- Improving MPI I/O operations on the parallel file system Lustre by supporting a new file locking mechanism.

In the first part, we focused on optimizing collective I/O operations by developing an approach that overlaps I/O operations with communication. We proposed several possible solutions by investigating different techniques for data transfer and altering the "synchronicity" of I/O operations. This work mainly targeted the inter-processes communication overhead of collective I/O operations [15].

The second task in this dissertation targeted and other type of parallel shared file I/O operations, i.e., individual I/O. Although it does not involve any kind of synchronization between the processes, it incorporates an additional computation phase that translates the access pattern of the file I/O as well as the input data layout to lists of offsets and lengths recognizable by the low level I/O functions. Reducing the cost of this phase is our primary goal in this work. As a first step, we proposed a solution to parallelize the current implementation of the computation phase through POSIX Multithreading. Then, we extended the parallelism to cover the actual I/O calls for the purpose of establishing a hybrid programming model that supports both inter-node and intra-node parallelism [16].

In the last part, we mainly focused on another source of overhead in parallel I/O applications which is strictly related to the first layer of the I/O software stack, i.e., parallel file systems. Basically, each parallel file system has its own specificity and defines a customized architecture that

has its own advantages and drawbacks. In this research topic, we focused on the locking mechanism of the parallel file system "Lustre" and its effect on I/O operations. Lustre has introduced a new locking mechanism to enhance the performance of I/O operations. We incorporated this new feature into the MPI I/O implementation by investigating the different possible approaches in supporting it for both collective and individual I/O operations. By doing so, we discovered new further optimizations that can reduce the communication overhead of the current collective I/O implementation for Lustre.

1.7 Dissertation Outline

In this section, we present the organization of this dissertation. First, we start, in Chapter 2, with a full description of the different subjects, notions and related works we build our research projects on. Then, we provide a comprehensive explanation of each proposed project supported by theoretical and experimental justifications, In Chapter 3, we discuss collective I/O operations and the different techniques and algorithms we proposed to improve the overall I/O performance. Chapter 4 presents our solution to improve individual I/O operations by applying multithreading on the original algorithm. As in Chapter 5, we focus on the new optimizations of MPI I/O on the Lustre parallel file system. Finally, we summarize, in Chapter 6, the scientific contributions of this dissertation and present our perspectives for the possible future work in this research area.

2 Background and Related Work

In this chapter, we focus on the main topics on which we founded our perspective toward a high performance parallel I/O solution for low latency storage systems. First, we provide a comprehensive overview of MPI I/O while focusing on the internals of both collective and individual I/O operations and discussing the most relevant related works that were done in the past. Then, we introduce the notion of hybrid programming model along with the different research that targeted this area of parallel I/O. Finally, we present some details about Lustre parallel file system and its file locking protocol while focusing on the previous research that targeted the file locking problem.

2.1 MPI I/O

Message Passing Interface (MPI) [41] is a specification of a portable message-passing standard created by a group of research scientists from different backgrounds. It is not an actual library but rather a paradigm that sets the theoretical basis of a distributed memory model and defines the syntax and semantics to follow for the implementation. There are many efficient implementations of MPI such as Open MPI [17] [45], MPICH [19] and Intel MPI [27]. Starting from version 2 of the MPI specification [18], the standard introduced an interface for parallel file I/O support, referred to as MPI I/O. By extending the MPI concepts to file I/O operations, the programming model becomes more complete and offers more options for developers to exploit the performance benefits of parallel I/O such as defining the file access pattern and the data layout and using different types of I/O operations (collective/individual, synchronous/asynchronous). As explained in the previous section, parallel application cannot achieve optimal I/O performance without unleashing the full capabilities of a storage systems through parallel file systems. The de facto standard for I/O operations is POSIX as most parallel file systems implement a support for its semantics. However, the POSIX interface lacks essential features needed to enable parallel I/O which has amplified the need for a high level interface on top of the parallel file system that offers more flexibility and portability to file I/O, and hence the introduction of MPI I/O.

2.1.1 MPI Derived Datatype

As a general definition, MPI datatype is an opaque object that defines a sequence of basic datatypes (int, float etc..) and their arrangement specified by displacements. It can be either predefined or customized. The predefined type corresponds to the regular types of variable in common programming languages, e.g., MPI_INT, MPI_DOUBLE. The customized type, often refereed to as 'derived datatype', is a complex structure defined by the MPI users by combining different predefined datatypes at different positions. MPI offers multiple functions to construct derived datatypes that can match different data layout like contiguous arrays, indexed arrays and strided blocks of datatypes. The main goal of MPI derived datatype is to add more flexibility and effectiveness to MPI functions in particular communication and I/O operations. Most parallel applications are dealing with non contiguous data, therefore they have to explicitly pack the data into contiguous buffers before the send operations and unpack it after the receive. Derived datatype can prevent such explicit operations and provide better optimizations to such operations. Timo Schneider et al. [50] argue that data packing, whether it is explicit or implicit, is very crucial since it can make up to 90% of the communication time for common data access pattern in real world applications. On the other hand, there are many articles [24] [35] that proves the efficiency of derived datatype over explicit packing. Furthermore, derived datatype allows MPI programs to support different memory representations at the same time which will reduce the number of in-memory copies. For example, one process can send data from its local buffer using a particular memory representation while another process can receive it directly into his local buffer with a total different layout. Similarly, we can use the same approach within MPI I/O to describe the data layouts in both memory buffers and file views.

2.1.2 Data Layout Definition

In parallel application, multiple processes are concurrently accessing a file in an interleaved manner. The data can be partitioned into unlimited number of possible layouts that can be contiguous, irregular and sometimes overlapped. To solve this problem, parallel file systems and some parallel

programming paradigms introduced the notion of "file modes" that defines the semantics for concurrent I/O operations [10]. Yet, MPI I/O has adopted a different approach that uses the notion of derived datatypes to describe the data layout.

Typically, each process is interested in only a subset of the overall data file, namely the part containing the data that has been assigned to this process based on the data partitioning strategy of the parallel application. This can be for example a certain number of rows or columns of a large matrix, or more complex patterns such as a 2-D subset of a much larger 2-D matrix. MPI relies on the notion of `derived datatypes` to express what part of a file a process intends to read or write. Generally speaking, two different derived datatypes are being used during a MPI I/O file access operation a **file view** and a **buffer view**.

File View

A process has a *file view*, which defines the parts of the file that this process plans to access. A file view is the access pattern that a process will follow when operating on a shared file. Since each process will access a specific region of the file, it has to define its own file view. This operation needs to be synchronized between all processes before starting the actual I/O operations, therefore it is a collective function in which all processes must participate. Setting the file view can be very beneficial. On the one hand, it enables efficient non-contiguous access to the file. On the other hand, it may create wider windows for different optimization techniques that improve file I/O performance since it provides a global preview of the access pattern of each process and their file offsets.

To create a file view, each process needs to provide an initial offset that defines the starting point in the file, a derived data type that describes the data distribution and an elementary type which represents the basic element of the data composition, e.g., MPI INT. Derived datatypes used to create a file view typically provide relative displacement to the beginning of the file for each data block. For example, a file view could contain a description that a process intends to access $1,024$ bytes starting from offset $4,096$ in the file, and $2,048$ bytes starting from offset $16,384$.

Buffer View

Each process has the data structure to be read/written in its local memory and will use a derived datatype during the `MPI_File_read/write` operation to provide a description of the element it intends to read/write. This is similar to sending a non-contiguous data structure using `MPI_Send`. Note that the derived datatype used to describe a data structure in memory does not have to be identical to the derived datatype used to describe a file view: a process could write less (or more) data than represented by the file view with a single function call. Furthermore, the data structures themselves might not be identical, e.g., the data structure in memory could contain halo-cells that don't have to be written to disk.

2.1.3 Data Access Options

A process can access a file by performing one of these two I/O operations: Read or Write. Yet, data access is more complex and encompasses multiple options. The MPI Forum [41] has defined three orthogonal aspects for data access in MPI I/O: positioning, synchronism and coordination. The positioning refers to the method that I/O operation uses to know the position in the file. It can be defined explicitly by an offset or implicitly by a file pointer. A file pointer can be shared between all processes or replicated individual for each one. Synchronism defines the access mode whether it is blocking, non-blocking or split collective. Coordination is the nature of synchronization level between the processes. If the data access routine requires synchronization between processes, it is called a collective operation otherwise it is a non-collective operation. MPI I/O provides a full set of I/O functions that define all possible combinations derived from these aspects. This richness is the main reason that makes MPI I/O more flexible and suitable to parallel applications that deal with a variety of irregular patterns of data.

2.2 Collective I/O

The data decomposition used by parallel applications often leads to a non-contiguous data layout that usually results into a large number of small I/O requests which will cause a significant performance degradation in file I/O operations. In order to solve this problem, MPI I/O introduced the concept of collective I/O operations.

Collective I/O represents a high level abstraction layer of file I/O operations that requires the collaboration and synchronization of a group of processes to perform I/O operations. It incorporates some internal optimizations such as rearranging data across processes to match the data layout at the file system level and merging many small I/O requests into fewer and larger global requests. These optimizations will significantly reduce the contention at the file system level, increase I/O bandwidth and hence improve the overall file I/O performance.

2.2.1 Two-phase I/O

The ability of processes to coordinate file I/O requests is generally considered to be the most relevant aspect for achieving good performance for I/O operations in parallel applications. The most widely used approach in collective I/O as of today is the two-phase I/O [11] algorithm. The main idea of the algorithm is to exploit the fact that I/O operations are much faster when applied on large contiguous data chunks. So, by shuffling the data among the MPI processes and rearranging it in a contiguous way that matches the data layout of the file, a subset of processes, so-called aggregators, are designated to perform the I/O operations. As its name would suggest, the algorithm consists of two phases:

1. **Communication phase (Shuffle phase)**

   In this phase, the data is redistributed over the aggregators by using inter-process communication primitives and rearranged such as each aggregator would be in charge of a contiguous sub-region of the file. Although it will create a new source of overhead, i.e., inter-process communication, the shuffle phase will subsequently lead to a significant performance gain at

the second phase that will overshadow it.

2. **File access phase (I/O phase)** After the first phase, each aggregator ends up with a large chunk of contiguous data in its local buffer known as "collective buffer". It can hence issue the file I/O requests to the file system to perform the actual read/write operations.

As mentioned in Section 2.1.2, MPI I/O allows all processes to share the entire view of the shared file through the `MPI_File_set_view` function and acquire the required information about the data distribution. Consequently, each process creates its own data structure about the data it holds (in term of offsets and lengths) and the aggregator that is responsible of it. Usually collective buffers have a fixed size because the primary memory cannot hold the whole data volume at once. Therefore, the so-explained process of the two phase I/O algorithm is performed in multiple cycles such that the size of the processed data in each cycle matches the collective buffer size. This parameter, as well as the number of aggregators are configurable using some run-time parameters of the MPI I/O implementations.

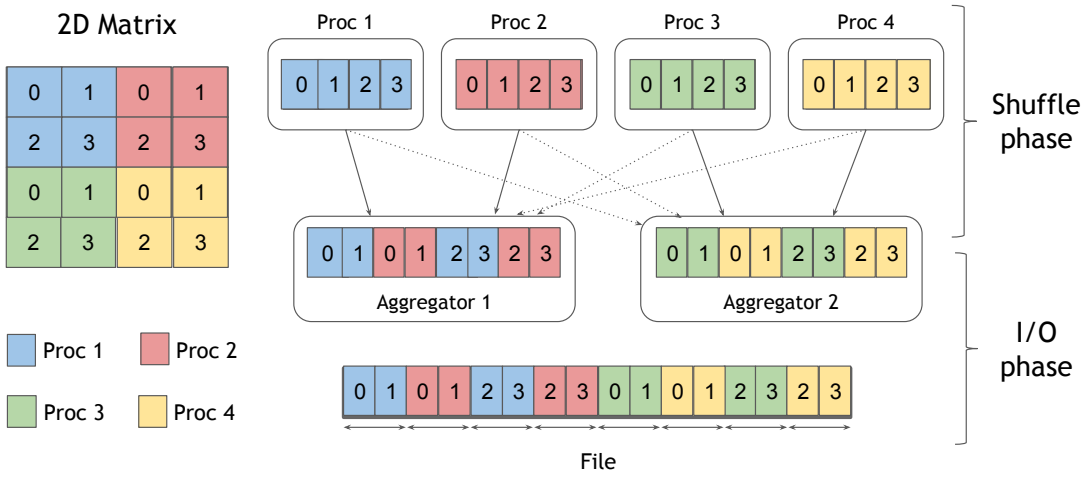

Figure 4: Example of the two-phase algorithm on a 2D data decomposition.

Fig. 4 shows an example of the two-phase algorithm applied on a 2D data decomposition. Consider a 2D matrix divided between 4 processes into sub-regions where only two of them are

designated as aggregators. In order to build one large contiguous data, each two consecutive rows should be assigned to a different aggregator. Therefore, process 1 and process 2 send their data to the first aggregator while the other two send it to the second aggregator. When the shuffle phase finishes, the aggregators should already have their collective buffers filled with the new contiguous data and can hence start the I/O operations.

2.2.2 File Domain Partitioning

Toady's storage systems incorporate a large number of disks managed by multiple I/O servers. In parallel file systems, the data is typically divided into several chunks called *"stripes"* and distributed over I/O servers. By doing so, a client can issue parallel I/O requests to the I/O server holding the requested data. This design provides an efficient way to concurrently access a file. However, parallel file systems have to guarantee the consistency of data by implementing their own file locking protocol. Depending on the locking protocol as well as the lock granularity, the data redistribution in collective I/O should be re-adjusted to be aligned with the lock boundaries.

The choice of the partitioning scheme has a significant impact on parallel I/O performance. The original implementation of the two-phase I/O algorithm is based on the **"even partitioning"** method in which the data is evenly redistributed over the aggregators so that each one is assigned to one contiguous sub-domain of the file view called "file partition".

In addition to this method, Liao et al. explored three other different file domain partitioning methods and evaluated them on two parallel file system, i.e., Lustre and GPFS [31]. The results show that a strategy called *"Group-cyclic partitioning"* leads to the best collective I/O performance on the Lustre file system. In this methods, the aggregators are split into groups of size equal to the number of I/O servers. Each group internally follows a more generic partitioning method called *"Static-cyclic partitioning"*. We are mainly interested in this method as it is currently adopted by the ROMIO and OMPIO implementations of MPI collective I/O for Lustre.

The **"Static-cyclic partitioning"** method partitions the file view into equal blocks of data matching the stripe size of Lustre, i.e., the locking granularity. Then, it statically re-distributes

the data over the aggregators following a round-robin fashion analogous to the Lustre file striping pattern.

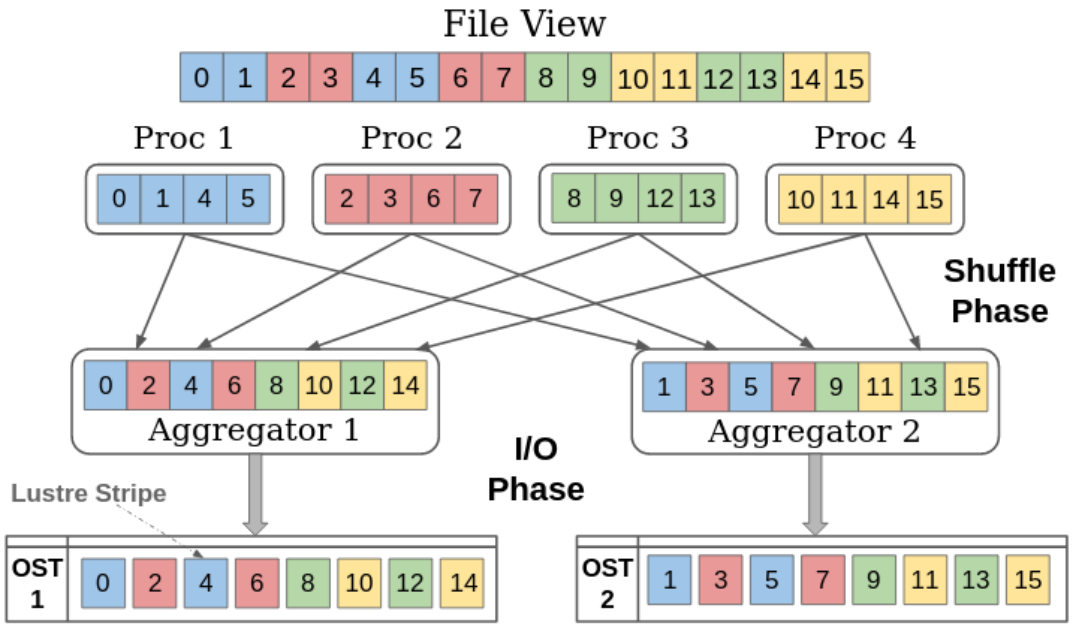

Figure 5: Example of a collective write operation of a 2D data distribution on the Lustre file system using the two-phase I/O algorithm with a static-cyclic partitioning scheme.

As an example, let us assume that we have to write a 2D data distribution with the file view illustrated in Fig. 5. The file consists of 16 Lustre stripes. Initially, each of the 4 processes holds 4 different stripes of data in its local memory. Usually, the number of aggregators should match the number of the *Object Storage Targets (OSTs)*, i.e., Lustre I/O servers. Therefore, only two processes are designated as aggregators and each one of them is assigned to a specific OST.

Since the file has to be striped across the OSTs in a round-robin fashion in Lustre, the processes has to re-arrange the data in a strided manner into the collective buffers of the aggregators. Once the shuffle phase is finished, Aggregator 1 (resp. 2) should have received only the stripes of even (resp. odd) indices. Finally, in the I/O phase, each aggregator iterates over the stripes aggregated in its local buffer and issues a POSIX write operation for each.

2.2.3 Optimization Techniques in Collective I/O

Numerous researchers have explored numerous techniques to optimize collective I/O operations. Some of them focused on collective buffering techniques by exploring different possible approaches [43, 38, 37]. Others have tried to extend the two phase algorithm to make it more efficient for various scenarios such as datatype I/O [54], view-based collective I/O [2], resonance I/O [64], and dynamic segmentation and static segmentation algorithms [9]. Liao et.al. analyzed various file domain partitioning methods for improving the performance of collective I/O operations [31]. Wang at el. introduced internal pipelining to manage collective I/O operations and limit the number of processes posting I/O requests to a storage server [62]. Most of these algorithms tweaked on how the shuffle and I/O operations are performed, yet they used the same fundamental approach of the two-phase algorithm.

One research project focused on optimizing the implementation of two-phase I/O by proposing a topology-aware solution for data aggregation. Weifeng Liu et al. [32] utilized the Linear Assignment Problem (LAP) to efficiently assign file domains to aggregators. The new data distribution over the aggregators reduces the total hop-bytes of collective I/O operation and thus improving the performance. Yuichi Tsujita et al. targeted different optimization techniques in their paper [57] by aligning the aggregator data to the striping access pattern and enhance it with a round robin algorithm for data distribution over multiple aggregators within a same node in order to alleviate the data transfer contention. In addition, they proposed the I/O throttling technique that consists on limiting the number of I/O requests issued by each aggregator in order to overcome the limitation of the file systems storage servers in term of the number of possible I/O threads and queued requests.

As part of their work on View-based I/O, Blas et al. [2] also reported on improved collective I/O performance by using read-ahead for collective read operations, and use multiple threads to perform these operations in the background, effectively overlapping the read operations with other ongoing operations.

The two phase algorithm is performed within multiple cycles, therefore many researchers tried to exploit this point for optimization by trying different techniques such as overlapping the shuffle

phase and the file access phase. This is in fact the focal point of our research introduced and explained in more details in Section 3. The most closely related work to this paper was done by Sehrish et al. [51]. The authors proposed an overlapping scheme similar to one of our proposed models, implementing the operations using non-blocking point-to-point operations. This paper however did not consider what part of the collective I/O operations can be overlapped, nor have they explored the use of different data transfer primitives. Similarly, Tsujita et al. [58] applied a similar approach by using asynchronous multi-threading. Nevertheless, very little work has tried to deeply explore **what** operations are being overlapped, and **how** the shuffle phase can be implemented.

2.3 Individual I/O

In individual I/O, the processes are completely independent of each others. Each one has its own file view and buffer view and performs the required I/O operations individually without requiring any data transfer or synchronization among the processes.

Internal Operations

In the following, we provide some details on the internals of an MPI I/O individual I/O function. The description will be based on the OMPIO parallel I/O frameworks of Open MPI [8]. Note that for large data volume, the internal I/O operations are usually performed in multiple cycles in order to keep the additional memory requirements of the MPI processes within reasonable limits.

The first step in any MPI I/O function is to decode the file view and the buffer view, i.e., translate the derived data types into arrays of I/O vectors (*iovec* structure). Each element of an I/O vector comprises a pointer and an integer value. The integer represents the number of bytes of the data block while the pointer represents either a memory address or an offset.

The main I/O algorithm for both read and write operations consist of two phases:

1. The *Build IO* phase, which performs file offset calculations, explained in more details below.
2. The *File Access* phase, which performs the actual file I/O.

The main goal of the *Build IO* phase is to calculate the parameters required for the subsequent *File Access* phase. A read/write operation requires three essential items: 1) The address of a data block in the local memory to be read or written; 2) the offset in the file where to write the data to or read the data from; and 3) the length of data block. This triplet represents the parameters of a single file I/O call. Therefore, the code parses the two arrays of I/O vectors defining the file view and buffer view, *fh_decoded_iov* and *decoded_iov*, and perform the necessary calculations to create an array of these triplets, called the *io_array*.

Let us explain the creation process of the *io_array* using the write operation illustrated in Fig. 6. We suppose that both buffer view and file view have 5 elements. Each element represents a contiguous data chunk defined by a memory address *"Addr i"* or an offset *"Off i"* and its length in bytes *"$S_X i$"* where *"i"* is the index of the element and *"X"* is either *"B"* or *"F"* standing respectively for buffer view and file view.

The first data chunk in the buffer can completely fit into the first element of the file view as it is smaller. We can hence define the first I/O operation by the triplet (Addr 1, off 1, $S_B 1$) previously explained. A data of size *"$S_B 1$"* is written from the buffer with address *"Addr 1"* to the file at offset *"Off 1"*. The second element of the buffer view is bigger than the remaining size of the file view. Therefore, we have to split it into two sub-elements such as the size of the first sub-element is equal to the size of the smallest data chunk that can fit in the rest of the file view element. The same process keeps running until identifying all the I/O operations that will be performed in the corresponding cycle so that the sum of the data lengths of all created elements is always less or equal to the cycle buffer size. As soon as the Build IO phase is done, the resulted *"io_array"* is passed as a parameter to the internal OMPIO write/read function. Then, it iterates over all elements of the array and call the corresponding POSIX I/O functions, i.e., `pwrite/pwritev or pread/preadv`.

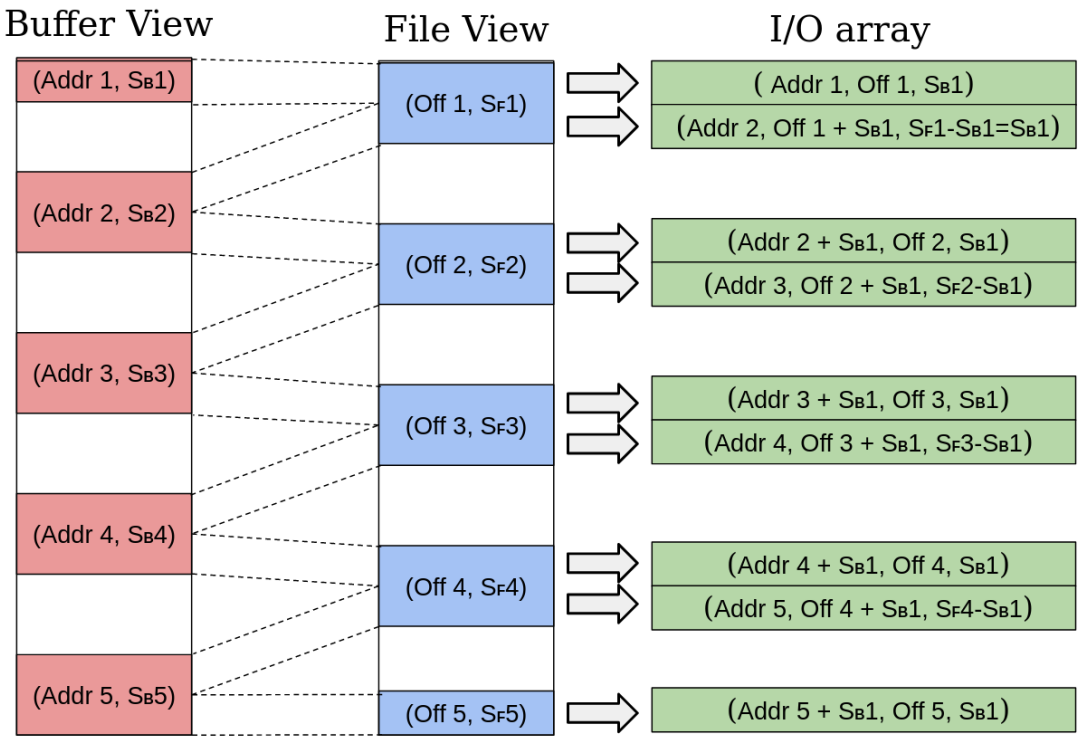

Figure 6: Example of the creation process of an io_array for a given cycle.

2.4 Hybrid Programming Model

Todays compute clusters are architecturally very complex. High-end computer systems contain thousands of compute nodes connected by a high-speed network interconnect, each node having one or more multi-core processor with multiple threads per core. Applications developer have to invest unprecedented efforts to develop scalable code, such as taking the process/thread and memory affinity into account; use asynchronous and/or remote direct memory access (RDMA) based communication operations; and deploy a multi-layer file I/O strategy to manage very large data sets.

To cope with these challenges, most large scale applications have started to use multiple parallel programming models simultaneously. MPI [41] is as of today still the most popular programming

paradigm providing efficient inter-node communication, supporting all state-of-the-art network interconnects, and having a large software eco-system. Intra-node parallelism is typically based on multi-threaded programming models such as OpenMP [46] or POSIX Threads [6]. They offer a simpler programming model based on shared memory without the necessity of explicit communication between threads or managing halo-cells. The combination of these two programming paradigms (MPI+threads) allows to maximize compute performance within a node and minimize communication costs between processes on different nodes. Yet, as of today, no parallel I/O library (MPI I/O [41], HDF5 [20], PNetCDF [5], ADIOS [33]) have support for hybrid MPI+threads based parallel applications. File I/O operations are executed by a single (master) thread per node, introducing de-facto an intra-node serialization and hence a performance bottleneck.

While no parallel I/O library has full support for the MPI + threads programming model, using multiple threads for file I/O has been explored to various extents. Within the context of shared-memory parallel programming, Wang et al. introduced in iHarmonizer [61] a user-level scheme that exploits OpenMP threads for file I/O operations. It dynamically manages I/O requests based on data availability by using a separate I/O thread for pre-fetching requested data and notifying the other threads of its availability. While this approach improves the efficiency of multi-threaded programs, it does not improve I/O operations themselves. To address this drawback, Mehta et al. designed and implemented parallel I/O [40] interfaces for OpenMP, demonstrating benefits of using multiple threads in file I/O operations.

Multi-threading has been also introduced in distributed memory programs. More et al. designed and implemented a multi-threaded runtime library in which uses two separate threads for computation and file I/O. Their approach in implementing multi-threaded collective I/O allowed I/O operations to be overlapped with computation. However, there were some limitation concerning the collective buffer size causing a performance degradation. Dickens et al. studied the parallelization of collective I/O functions using threads. They suggested using threads for some aspects of the collective I/O operation while executing other parts of the operation in the foreground [12].

Using a similar concept, Ma et al. proposed a multi-threaded solution to overlap file I/O with

inter-process data transfer within the collective I/O algorithm [38]. A similar approach has also been adopted by Tsujita et al. in implementing a multi-threaded two-phase I/O algorithm for Lustre file system [58]. On addition, the authors also proposed a solution for the management of CPU core affinity [56]. Ultimately, none of the projects deploy multi-threading to the full internal operations of the parallel I/O library however.

2.5 Lustre File System

The Lustre file system is an object-based parallel file system deployed in multiple supercomputers of the top500 list. The files are striped into multiple chunks of data called stripes. Each stripe is stored as an object containing the raw data along with multiple other information describing it. The Lustre architecture consists of two main component: 1) the Object Storage Server (OSS) that constitutes the I/O servers that manages all file I/O operations including the storage of data into the object storage targets (OSTs) and 2) the Metadata Server (MDS) that manages the file metadata as a standalone service.

Lustre is a POSIX (not completely) compliant parallel file system. It offers the clients concurrent and coherent I/O access to *resources*, i.e., files, through a unified *namespace* using standard POSIX semantics [36]. In order to maintain data consistency and I/O cache coherency for concurrent access of multiple clients, Lustre follows a locking protocol implemented through the Lustre Distributed Lock Manager (LDLM) [60].

2.5.1 Lustre Locking Protocol

The biggest challenge is to achieve a strong I/O consistency in a parallel distributed environment without affecting the performance. Let us assume we have two consecutive I/O operations, 1 write then 1 read, on a same region of the file. Once the first operation finishes in one client side, the second operation should have an access to a "clean", newly updated version of the data. While it seems very straightforward, maintaining such a consistency in a modern parallel file systems might be problematic due to data caching.

To alleviate the impact of disk latency in I/O operations, operating systems tends to use page caching. Instead of writing the data to the physical storage device, the data is temporally cached in the main memory, to be executed on the background asynchronously while the main application proceeds with the next instructions. In this case, if the read() function in our example requires an access to the same data before the write operation has fully completed, it needs to read it directly from the page cache. However, if the clients issuing these two I/O operations are in separate nodes, hence do not share the same page cache, the file system has to flush the whole cached data to the disk before proceeding with the read operations. Therefore, the Lustre locking protocol imposes that only one client can acquire a "write lock" on a sub-region of the file at once.

Furthermore, even if a client requests a lock on only 1 stripe, the OST holding that part of the file returns a larger lock covering all the stripes it is holding. As a result, only one client can acquire a "write lock" on each OST at once, and hence the reason behind assigning 1 aggregator to each OST in the MPI collective I/O implementations. This locking behavior is called *lock expansion*.

2.5.2 Lustre Locking Problem

In this section, we discuss the locking behavior of Lustre with multiple writers to a single shared file. Let us use the same example illustrated by Fig. 5 in which an aggregator is a distinct Lustre client.

Lock Expansion Enabled (default)

Client 1 starts by requesting a first write operation on stripe 0 from OST 1. Since there is no LDLM lock on that stripe, the OST will automatically grant a lock of mode *PW*, standing for *Protective Write Mode*, to the client. Because of the lock expansion feature of Lustre, whenever a client request a PW lock on a region of the file, the OST handling that region will attributes the largest possible lock. In this case, as initially there are no locks, OST 1 expand the lock to cover all the stripes it handles, i.e., stripes 0, 2, .. 14.

The main reason for adopting the lock expansion strategy is to reduce the number of lock

requests, and hence the latency of a Remote Procedure Call (RPC). In fact, Lustre assumes only 1 writer per OST. Therefore, as it is the case in our example, the same client 1 continues the writing of the remaining stripes to OST 1. As it has a PW lock on all the stripes, no subsequent lock requests are required. This explains the choice of the static-cyclic partitioning method for collective I/O in Lustre.

Generally speaking, increasing the number of writers/readers accessing an I/O node should result into higher bandwidth and hence better I/O performance. Unfortunately, this is not the case in Lustre as it will cause a lock conflict between the clients.

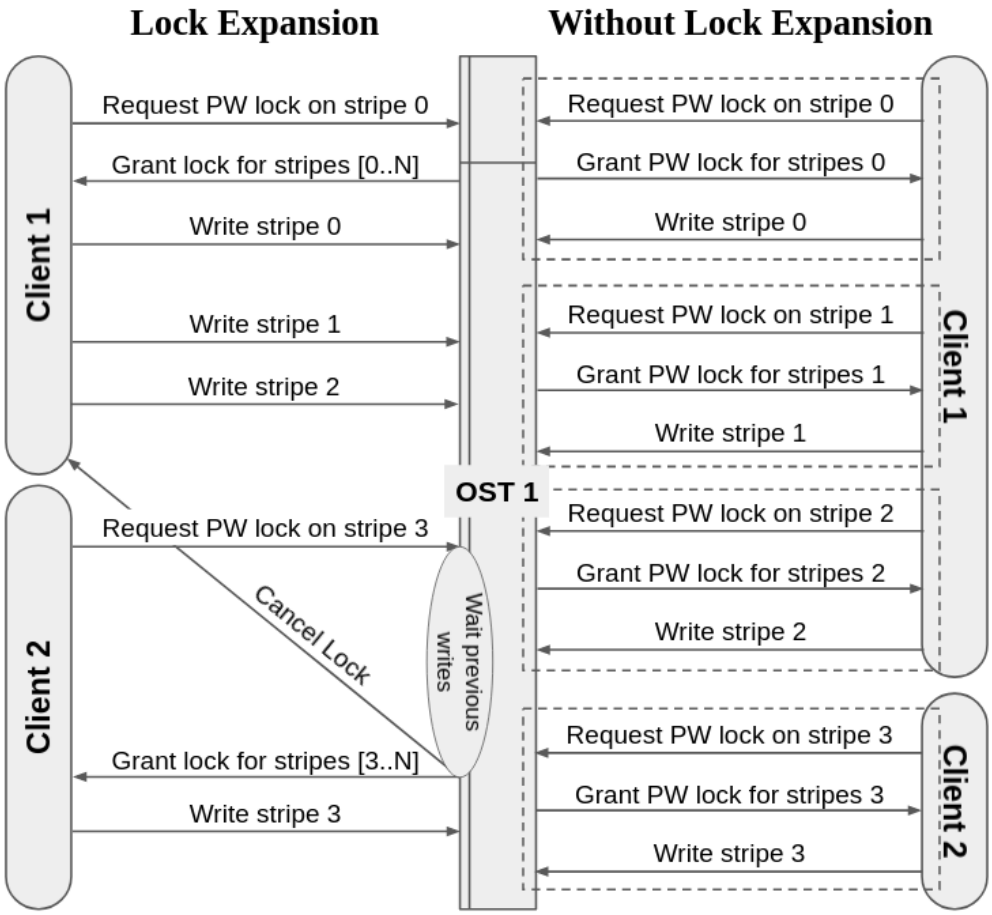

Figure 7: Lustre locking behavior in case of concurrent write operations when the lock expansion is enabled (default) and disabled.

When a second client needs to write a stripe on the same OST, it has to request a PW lock that conflicts with the existing lock attributed to client 1. As illustrated in Fig. 7, the OST has to wait first for the completion of the existing write operation as the PW lock is blocking. Afterwards, it checks the next I/O operation. If it is not conflicting with the existing locks, e.g., same client, it permits the operation without sending any new RPC. Otherwise, it has to cancel the previous lock, notify its previous holder of the cancellation and grant a new lock over all available stripes to the new client. Consequently, only one client at once can perform I/O operation while the others wait for it completion, hence the I/O serialization problem.

Lock Expansion Disabled

An intuitive solution for the serialization problem is to disable the default lock expansion feature. By doing so, the OST will grant the client a PW lock only for the requested stripe. When a second client requests a lock on a different stripe, it will be automatically granted since there is no existing lock on it. Unfortunately, although it enables parallel I/O access of multiple clients, this solution has shown even worse performance [13].

In fact, by disabling lock expansion, every call for a write operation has to be preceded by a round trip of RPCs exchanged between client and server. As shown in Fig. 7, even the same client needs to request a PW lock for each atomic I/O operation and wait for the OST response before accessing the file. This would become an important source of overhead unless it is combined with Lockahead requests.

2.5.3 Lustre Lockahead

Starting from the Lustre version 1.11, a new feature has been introduced to solve the serialization problem of shared file I/O in Lustre called **Lustre Lockahead** [42] (LLA). As its name would suggest, Lockahead allows a client to request a list of LDLM locks prior to I/O calls. Primarily designed for MPI-I/O library, Moore et al. from *Cray, Inc* has implemented a support of the Lustre Lockahead feature for ROMIO [55], the MPI I/O implementation of MPICH [19]. However,

they only explored the benefits of this feature for collective I/O implementation with a static-cyclic partitioning scheme.

The first step in this new locking mechanism is to disable the lock expansion to allow concurrent access of multiple clients to a single OST. Then, already aware of its access pattern, an I/O process which is a Lustre client can create an array of Lockahead advises defining the future I/O operations. An advise is defined by it lock mode (read or write) and the bytes range of the lock in term of file offsets (start and end). Lustre implements an interface called *"ladvise"* which is comparable to the Linux system call *fadvise()*. It allows the user to pass specific advises, i.e., hints, about the file access from a Lustre client to the server side. Based on these hints, the Lustre server applies appropriate read-ahead and caching techniques to optimize file I/O. Both lock expansion and Lockahead features are requested trough this interface using the *"llapi_ladvise()"* function.

Those advises might be issued asynchronously through an existing asynchronous lock request functionality of Lustre. The major advantage of this functionality is that no thread was created to wait for the LDLM lock but it is rather handled by the internal "ptlrpc" daemon thread. As a results, the delay caused by the multiple RPCs round-trips will be roughly substituted by the cost of one. The only downside of this solution is that asynchronous lock requests must be non-blocking, i.e., if an existing lock already exists, the request will be denied. In that case, when the actual write operation is issued for that range of data, the OST has to cancel the old lock and grant a new lock to the writer. This is the case of multiple collective I/O calls to an already opened file. In fact, Lustre imposes a restriction that a lock request has to be either asynchronous and non-blocking or synchronous and blocking [14]. Therefore, setting the Lockahead advises as blocking will nullify the benefits of asynchronous requests. Nevertheless, if we issue larger few Lockahead requests for each aggregator, the overhead of the exchanged RPCs would no longer exists.

3 Overlapping Collective I/O

In this chapter, we target the communication overhead within collective I/O operations. The contribution of this work is two-fold. First, we present and evaluate various design options for overlapping two internal cycles of the two-phase I/O algorithm, using both, the communication as well as the file access phase. Second, we explore the benefits of using different data transfer primitives for the shuffle phase, such as non-blocking two-sided communication, i.e., `Isend/Irecv` and multiple versions of one-sided communication (`Put`). In both instances, the focus of this work is on collective write operations. In Section 3.1, we detail the design and implementation of the different overlap methodologies and data transfer primitives. Then, we present in Section 3.2 our performance evaluation along with a full analysis of the results.

3.1 Design and Implementation

In the following, we introduce first four different design options for overlapping two internal cycles of the two-phase I/O algorithm, focusing on collective write operations. Second, we explore using different data transfer primitives for the shuffle phase, including non-blocking two-sided communication and multiple versions of one-sided communication.

3.1.1 Overlap Algorithms

In the original implementation of the two-phase I/O algorithm, all processes have to send their local data to the aggregators via inter-process communication. Each aggregator holds data from multiple processes in a temporary buffer. During the file access phase, the aggregators issue I/O requests to the file system to write the content of the temporary buffer onto disk. The next shuffle phase cannot be initiated until the temporary buffer is completely flushed and ready to receive the next chunks of data. These two phases are carried out in multiple cycles until the entire data is written.

In the overlapped two-phase I/O, the temporary buffer is divided into two separate sub-buffers

such that the size of each one would be equal to the half of the buffer size in the original implementation. By doing so, we can run different operations in each sub-buffer, hence overlap the shuffle phase and the I/O phase.

Two different operations are performed in each sub-buffer: 1) Inter-process communication (send/receive operations), and 2) I/O operations. Both operations might be called using either blocking or non-blocking implementations. The use of non-blocking functions over two collective sub-buffers is the main key to perform overlapping. However, there are multiple ways to implement the overlapping technique depending on the choice of the asynchronous functions and the phases that will be overlapped. Hence, we propose four different approaches.

In the following, we describe the various overlap algorithms. A brief note on the terminology used: whenever an algorithm utilizes a non-blocking or asynchronous code section, we use the the postfixes `_init` and `_wait` to indicate the start and the end of the non-blocking section, while code sections not having these postfixes indicate blocking implementations. Hence, `shuffle` and `write` represent blocking versions of the shuffle and I/O phase respectively, while, e.g., `shuffle_init` and `shuffle_wait` indicate the start and the completion of the non-blocking version of the shuffle operation.

Communication Overlap

In this first approach, non-blocking send/receive operations are being used while maintaining a blocking version of the write operation. Algorithm 1 provides a high level overview of this approach. A shuffle phase is initiated on the first sub-buffer p_1 (`shuffle_init`) before the start of the internal cycles. In each cycle, a shuffle phase is started, followed by a wait operation of the previous shuffle operation (`shuffle_wait`). The completion of the first shuffle operation means that all required data has been received in the sub-buffer and the buffer is ready to be written. While a write operation is ongoing, the other shuffle operation continues running in the background. Since the I/O operation is synchronous, the next cycle cannot be started until the file I/O phase is completed on the aggregator. Then, pointers representing the two sub-buffers are being swapped

(`swap_buffer_pointers`), ensuring that buffer used in the last cycle in the shuffle operation is being used in the write step next, and vice versa.

Algorithm 1 Communication Overlap

Require: $tempbuf_1$, $tempbuf_2$, $NumberOfCycles$

```
1: p1 ← tempbuf1
2: p2 ← tempbuf2
3: shuffle_init (p1)
4: for i=1 to NumberOfCycles do
5:     shuffle_init (p2)
6:     shuffle_wait (p1)
7:     write (p1)
8:     swap_buffer_pointers (p1, p2)
9: shuffle_wait (p1)
10: write (p1)
```

Some notes on the nomenclature used in this and all subsequent algorithms: Function names ending with `_init` represent a code section that initiates communication and/or file I/O operations and will not block during execution, while functions ending with `_wait` represent code sections enforcing the completion of previously initiated operations. A function without either of those endings represent a blocking code section, which could however be implemented using a sequence of initiation and completion, but doesn't necessarily have to.

Non-blocking shuffle operations can be implemented using non-blocking point-to-point data transfer operations in MPI, such as `MPI_Isend` and `MPI_Irecv`, and `MPI_Wait` for the completion. An important performance consideration for this code version will however be the ability of the MPI library to make progress after initiating the communication operations. MPI libraries provide progress for pending non-blocking data transfer operations either when invoking an MPI function, or more recently also through a specific progress thread [25].

Writes Overlap

This algorithm represents the counterpart to the Communication-Overlap version discussed above, using however blocking shuffle steps and asynchronous write operations. Asynchronous

write operations can be implemented for example using the `aio_write()` functionality (and although slightly beyond the scope of this paper, it should be noted however that quality of the support for this function is dependent on the file system used for the file I/O operations). Within the scope of this paper, the assumption is that using `MPI_File_iwrite` will provide the method best suited on the given file system.

Algorithm 2 Write Overlap

Require: $tempbuf_1$, $tempbuf_2$, $NumberOfCycles$
1: $p_1 \leftarrow tempbuf_1$
2: $p_2 \leftarrow tempbuf_2$
3: `shuffle` (p_1)
4: `write_init` (p_1)
5: **for** i=1 to NumberOfCycles **do**
6: `shuffle` (p_2)
7: `write_init` (p_2)
8: `write_wait` (p_1)
9: `swap_buffer_pointers` (p_1, p_2)
10: `write_wait` (p_2)

Whether the Communication or the Write Overlap algorithm is expected to lead to better performance depends on multiple factors, most notably: i) costs of the shuffle phase vs. the file I/O phase and ii) whether the MPI library or the operating system is better at ensuring progress of communication operations or I/O requests in the background. In our experience, the file I/O phase is on most clusters more expensive than the shuffle phase. Furthermore, since `aio_write` operations are often execute by an OS thread, progress of non-blocking write operations will often be better than progress of communication operations when not using a separate progress thread.

Write-Communication Overlap

The third approach uses a non-blocking implementation of the write operation as well as for the shuffle phase. In each cycle, the algorithm initiation an asynchronous write operation on the first sub-buffer followed by initiating a shuffle stage on the second sub-buffer. Algorithm 3 shows this approach.

Algorithm 3 Write-Communication Overlap

Require: $tempbuf_1$, $tempbuf_2$, $NumberOfCycles$

```
1: p1 ← tempbuf1
2: p2 ← tempbuf2
3: shuffle (p1)
4: for i=1 to NumberOfCycles do
5:     write_init (p1)
6:     shuffle_init (p2)
7:     wait_all (p1, p2)
8:     swap_buffer_pointers (p1, p2)
```

Write-Communication-2 Overlap

A slightly revised version of the approach shown in Algorithm 3 is given by avoiding that the non-blocking shuffle and non-blocking write operation finish approximately at the same time. Instead, this version defined by Algorithm 4 follows more closely a data-flow model, in that the completion of any non-blocking operation is immediately followed by posting the follow-up operation first. Unlike the other versions, this algorithm handles two shuffles and two writes operations in each iteration which correspond to 2 cycles.

Algorithm 4 Write-Communication-2 Overlap

Require: $tempbuf_1$, $tempbuf_2$, $NumberOfCycles$

```
1: p1 ← tempbuf1
2: p2 ← tempbuf2
3: shuffle (p1)
4: write_init (p1)
5: for i=1 to NumberOfCycles do
6:     write_wait (p2)
7:     shuffle_init (p2)
8:     shuffle_wait (p1)
9:     write_init (p1)
10:    shuffle_wait (p2)
11:    write_init (p1)
12:    write_wait (p1)
13:    shuffle_init (p1)
```

3.1.2 Data Transfer Primitives

Whereas we focused in the previous section on identifying the different possible algorithms to implement the overlapping technique, we will discuss in this section two possible communication models that can be used for the shuffle phase: Two sided communication (send/receive) and one sided communication (Put/Get).

Two-sided Communication

The current implementations of the two-phase algorithm uses a two-sided communication model. During the shuffle phase, MPI processes send the data from their local buffer to the corresponding aggregator using non-blocking communication (`MPI_Isend, MPI_Irecv`). Internally, MPI libraries typically use different protocols for short and long messages, namely an *eager* and a *rendezvous* protocol. For short message, the eager protocol sends the data to the receiving process, independent of whether the receiver is ready to receive the data item or not. If the receiver has not yet posted the matching receive operation, the data will be buffered in an *unexpected message* queue on the receiver process. Consequently, a receive operation has to check the unexpected message upon posting the operation, which can be costly if the queue contains many messages.

MPI libraries utilize a rendezvous protocol for long messages, which requires a hand-shake between sender and receiver process. This ensure that large messages will not end up in the unexpected message queue of the receiver processes, limiting the additional memory requirement on that processes. However, the rendezvous protocol prevents a sender from continuing execution until the receiver process is ready to receive the data. For the MPI library and network interconnect used in the evaluation Section 3.2 (Open MPI master using UCX 1.6.1 on an InfiniBand network), the rendezvous protocol is used for messages starting from 512 KBytes in size.

In collective I/O operations, a large number of processes are communicating with a few aggregator processes. Since aggregator processes have significantly higher workload than non-aggregators – due to the file I/O access operations that they have to perform –, two-sided communication will typically result in many entries in the unexpected message queue of an aggregator, or, if the

messages are too large, the rendezvous protocol will enforce that non-aggregators will have to 'slow down' to the speed of the aggregator processes.

One-sided Communication

One sided communication, also known as Remote Memory Access (RMA), is a communication model added to the MPI specification starting from version 2. This model does not impose a synchronization of sender and receiver during communication: only one process is required for the data transfer, by either `Put`-ting or `Get`-ting data from the remote process. Even though the target process does not contribute to the communication operation in itself, it has to define and expose a region of its main memory for the operation, a so-called *window*.

One-sided communication is often considered faster and more light-weight compared to two-sided communication, due to the fact that there is no message matching required on the receiver side, and there is no unexpected message queue that needs to be parsed for every receive operation.

Within the scope of this work, which focuses on overlapping internal cycles of the two-phase collective I/O operation, we allocate two separate windows analogous to the two collective sub-buffers, using `MPI_Win_allocate`, the size of the windows being the size of the sub-buffers for aggregators and zero for non-aggregators.

One-sided operations also include methods that allow a process to control when to grant access to a memory window, or more generally speaking, a synchronization method between the processes. MPI provides two synchronization models: active-target and passive-target synchronization. Implementations have been developed for both methods.

1. **Active-target RMA:** It is a collective synchronization model. The simplest version of this synchronization method requires a call to `MPI_Win_fence` to start and end an exposure epoch. When closing an exposure epoch, the standard requires that all outstanding RMA operations on that window have completed.

   In our implementation, an `MPI_Win_fence` function is used at the start of the `shuffle_init` operation, and a second one whenever we need to ensure the completion of the data transfer.

This provides both, the origin and target processes (the aggregators), the information required to continue the next step and/or cycle in the algorithm. However, `MPI_Win_fence` is known to be an expensive operation.

2. **Passive-target RMA:** It provides more flexibility than active-target RMA, since the target process is not directly involved in the synchronization operation itself. In this model, the origin process has to acquire a lock on the remote window of the target process using `MPI_Win_lock`, execute its RMA operations, and release the lock using `MPI_Win_unlock`. The completion of the RMA operation is guaranteed for that particular window in the origin side. However, the target process does not know when the data transfer operation to his local buffer has finished.

   There are two challenges using this model in the two-phase I/O algorithm. First, the MPI specification offers a choice between two lock types: `MPI_LOCK_SHARED` or `MPI_LOCK_EXCLUSIVE`. The second option only allows one process to write into a window at a time, which will serialize the shuffle phase and thus harm the performance. `MPI_LOCK_SHARED` allows concurrent access to a window, and is usually used for read operations. However, since we can guarantee that different process will not overwrite each others data during the shuffle phase, we decided to use this version in our code.

   The second issue concerns the target processes. On one hand, aggregator processes need to know when all data-transfer operations on a sub-buffer have finished in order to initiate the I/O operations. On the other hand, the origin processes must not execute any `MPI_Put` operation on any sub-buffer before the aggregator has not finished writing its content to file. To meet these two requirements, `MPI_Barrier` synchronize had to be introduced to ensure correct semantics of the operation.

3.2 Performance Evaluation

In the following section we evaluate the different overlap methodologies introduced in Section 3.1.1 as well as the different data transfer primitives described in Section 3.1.2.

For our tests we used two platforms: the *Crill* cluster at University of Houston, and the *Ibex* cluster at the KAUST Supercomputing Laboratory. On the *Crill* cluster we used a partition consisting of 16 nodes with four 2.2 GHz 12-core AMD Opteron processor (48 cores per node, 768 cores total) and 64 GB memory per node. The cluster uses a QDR InfiniBand network interconnect, using UCX v 1.6.1 as the underlying communication library for the MPI library. A BeeGFS parallel file system v7.0 distributed over all 16 nodes has been used for our tests, with a stripe size of 1MB. The *ibex* cluster is a heterogeneous cluster with different families of AMD and Intel CPUs. In our tests we used the Skylake CPU family partition consisting of 108 nodes with 2.6 GHz 40-core Intel Xeon Gold 6148 Processor and 376 GB memory per node. Similarly to Crill, the cluster also uses a QDR InfiniBand network interconnect using UCX v 1.6.1 as the underlying communication library. The experiments were performed under a BeeGFS filesystem consisting of 3.6 PB of storage on which we set 16 storage targets and a stripe size of 1MB.

All of the algorithms and versions described in the previous sections have been implemented using the OMPIO [8] parallel I/O framework in Open MPI [17], by modifying the existing *vulcan* fcoll component. For all subsequent tests, the default OMPIO parameters and settings have been used, e.g., a collective buffer size of 32MB and the automatic runtime aggregator selection algorithm [7]. In order to evaluate the different approaches, three different benchmarks have been used.

1. **IOR**: The Interleaved or Random (IOR) [52] benchmark is a synthetic parallel I/O benchmark used for testing performance of parallel file systems using different access patterns through different interfaces, e.g., POSIX I/O and MPI-I/O. It has several high-level parameters that can be used to define the I/O pattern, e.g., transfer size, block size, and segment count. For our tests, we mimicked a 1-D data distribution among processes by setting the transfer size and the block size both to 1GB and the segment count to 1. Tests have been executed for 10 different process counts between up to 704 processes, creating files that range between 16 and 704 GB in size.

2. **Tile I/O**: MPI-TILE-IO [47] is a synthetic benchmark to test parallel I/O operations on a

two dimensional dense dataset. In our measurements, the number of tiles is set according to the number of processes ensuring that each dimension is equal to the square root of number of processes. Our tests used two different configuration: a tile size of 256 bytes with 2048×1024 elements per process, and tile size of 1 MByte and 32×16 elements per process. The Tile I/O benchmark was executed for process counts ranging from 16 to up to 729 processes.

3. **Flash I/O**: The FLASH I/O benchmark [65] suite is an extracted I/O kernel from the FLASH application. The benchmark is based on a lock-structured adaptive mesh hydrodynamics code that solves fully compressible, reactive hydrodynamic equations, developed mainly for the study of nuclear flashes in astrophysics. The benchmark produces a checkpoint file, a plotfile with centered data, and a plotfile with corner data. We focus in our study on the checkpoint file, since it is the largest of three output files generated. Tests have been executed for various configuration using between 16 and 704 processes.

For each benchmark test case, we run between 3 and 9 measurements using each of the previously described algorithms and for each process count. For the summary statistics as shown in Table 1, all measurements series were used. When comparing individual data points, e.g., in Fig. 8, we used the minimum execution time across all measurements within a series for that benchmark, process count and algorithm.

Table 1: Number of runs an overlap algorithm resulted in best performance

Benchmark	No Overlap	Comm Overlap	Write Overlap	Write-Comm Overlap	Write-Comm 2 Overlap
IOR	21	11	32	28	15
Tile I/O 256	17	13	18	31	26
Tile I/O 1M	10	6	18	20	17
Flash I/O	11	12	11	16	19
Total:	59	42	79	95	77

3.2.1 Overlap Methodology

First, we evaluate the performance of the different overlap algorithms described in Section 3.1.1. For this, we ran the benchmarks described above with all four overlap algorithms, as well as a

version of the two-phase collective I/O algorithm that does not overlap shuffle and file I/O access phases at all. Table 1 presents for each overlap algorithm the total number of test runs in which the corresponding algorithm provided the best performance, across all benchmarks, platforms, process counts, and problem sizes.

The foremost conclusion of Table 1 is that there is no clear winner or best approach across all test cases and platforms. In fact, even the version of the code not performing any overlap between the shuffle and the file I/O phase – which was originally only included to provide a base-line number – lead in 59 out of the 352 test series executed as part of this study with the lowest execution time, i.e., in approx. 16% of the test cases. The results however also indicate, that in 251 out of the 352 test series (71%) an overlap algorithm that used asynchronous write operations outperformed other approaches, showing a clear overall benefit of using asynchronous file I/O in the file access phase on both clusters.

Although the technical specifications of the clusters seem very similar, the overall performance numbers obtained are very different. First, despite of the fact that both clusters use a QDR InfiniBand network interconnect, the maximum bandwidth between two nodes on the Ibex cluster was higher than on the Crill cluster, due to the slightly older AMD processors (Magny Cours) used on Crill (approx. $3,400$ MB/s vs. $2,600$ MB/s). Furthermore, the BeeGFS storage on Ibex provided significantly higher write bandwidth compared to the BeeGFS file system on Crill. The parallel file system on Crill is based on using two additional hard drives in each of the 16 compute nodes, while Ibex uses a large scale parallel storage system. However, the Crill cluster has been used in a dedicated mode for these measurements, resulting in significantly less variance for the data obtained, while the Ibex cluster was shared with other users during the tests, resulting in larger performance variations.

Fig. 8 shows the differences obtained with the two clusters for the Tile I/O benchmark using 1M tile size for 256 and 576 processes.

The Crill cluster shows no significant performance benefit for using any overlap algorithm for 256 processes, and approx. 6% performance improvement by overlapping shuffle and I/O phase

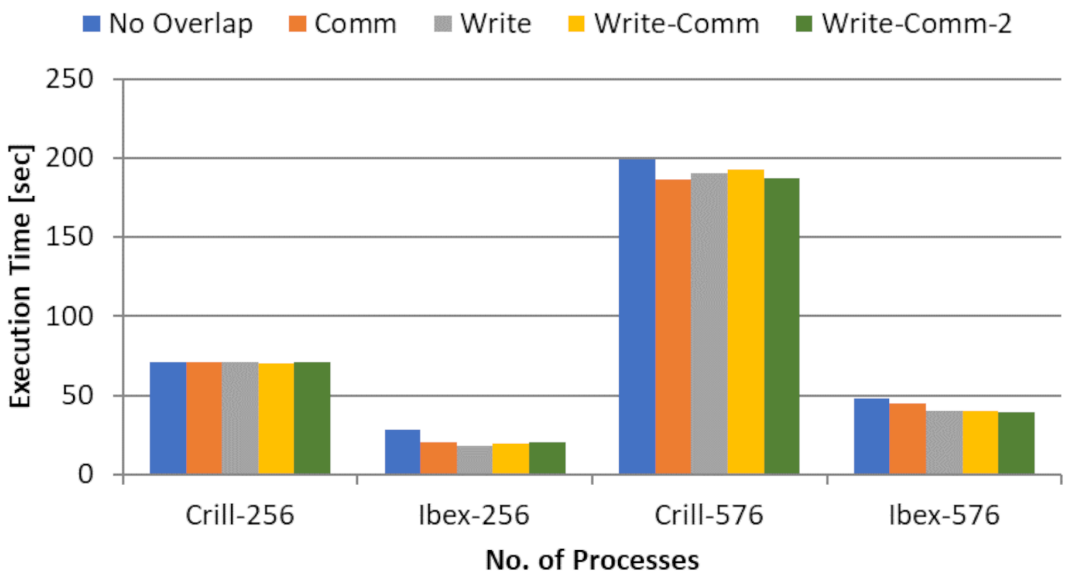

Figure 8: Execution time of the Tile I/O benchmarks for 1M tile size using 256 and 576 processes.

for 576 processes. The Ibex cluster other hand showed performance improvements in both cases, approx. 34% for 256 processes, and 17% for 576 processes. A detailed analysis of communication time and I/O time was performed using the no-overlap version of the code, in order to get a break down of how much time is spend in the shuffle vs. the file I/O access phase. For the 576 process test case, the collective I/O operation spends 93% of its time in file I/O access phase, and only approx. 7% in I/O operation on the Crill cluster, while on Ibex it spends approx. 23% of the overall time in communication, offering therefore a much larger window for improvements. Thus, despite of the fact that nearly the entire communication time could be hidden behind I/O operations for this test case on Crill, it resulted only in a limited 6% performance improvement overall on this platform, and a significantly larger improvement on Ibex.

This behavior is also confirmed by analyzing the average improvement obtained with each overlap algorithm and benchmark case for the Crill (Fig. 9) and the Ibex (Fig. 10) clusters. The average values shown in these graphs are determined by calculating the relative improvement in the execution using an overlap algorithm over the *no overlap* version, excluding however data points

in which *no overlap* was faster than the overlap version, i.e., negative improvements. The values therefore represent the average improvement per overlap algorithm and benchmark **if** a performance improvement over the *no overlap* version was observed.

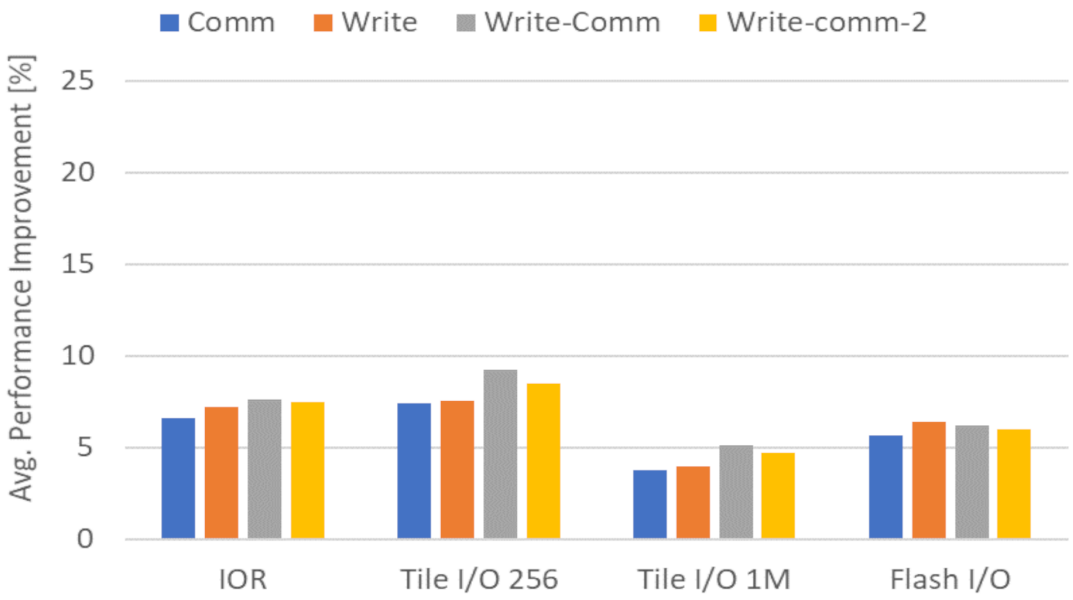

Figure 9: Average relative performance improvement on the Crill cluster for each overlap algorithm and benchmark.

The average improvement on the Crill cluster was between 3.7% and 9.2%, with overlap algorithms using an asynchronous write operation outperforming the communication overlap version in all instances. The same holds for the Ibex cluster, the average improvement was however higher, ranging between 8.6% and up to 22.3%.

3.2.2 Data Transfer Primitives

In the second part of the evaluation, we focus on the data transfer primitives used by the two-phase I/O implementation. As discussed in Section 3.1.2, different implementations of the shuffle step have been developed based on the *Write-Communication-2* overlap algorithm, using non-blocking two-sided communication, one-sided communication using `MPI_Put` and `MPI_Win_fence`

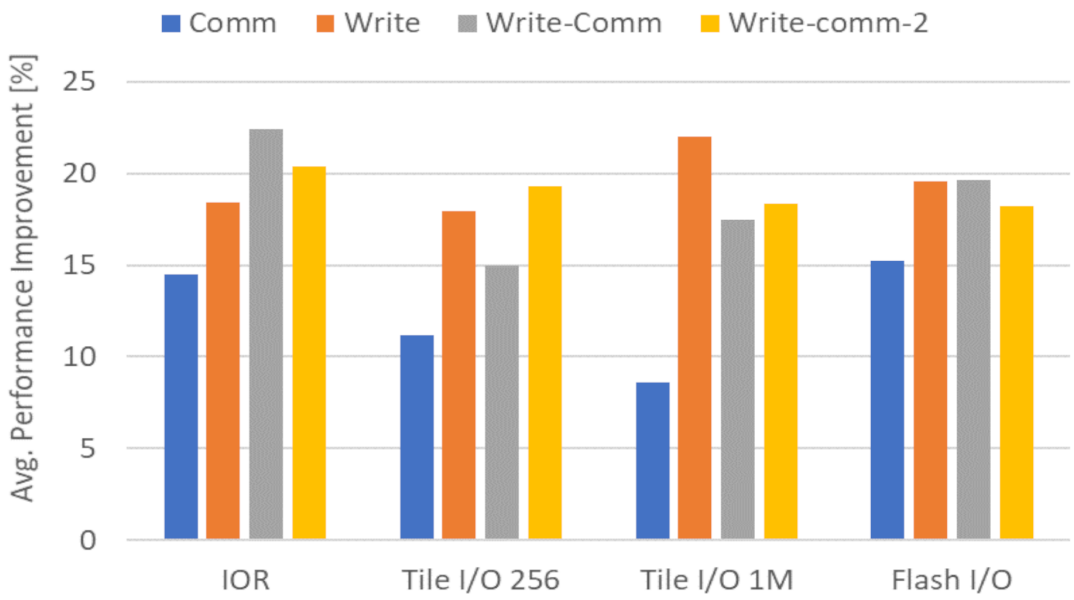

Figure 10: Average relative performance improvement on the ibex cluster for each overlap algorithm and benchmark.

for synchronization, and one-sided communication using `MPI_Put` and `MPI_Win_lock/unlock`. Tests have been executed on the Crill as well as the Ibex cluster using the IOR, and the Tile I/O benchmarks for both 256 and 1M tile sizes.

Fig. 11 summarizes the results of the analysis. The graph shows for each benchmark test case the number of times a particular implementation resulted in the best performance overall. The results indicate that in the overwhelming number of test-cases (75%) two-sided data communication resulted in the best performance, and outperformed either of the two versions using one-sided communication. The results where consistent across both clusters. Ultimately, the synchronization costs introduced by the `MPI_Win_fence` and/or `MPI_Barrier` required for the one-sided operations to ensure correct output data out-weight the performance benefits of an `MPI_Put` operation over `Isend/Irecv` communication.

The only notable deviation of the general trend is for the Tile I/O benchmark when using the small 256 Byte tiles. In this scenario, the one-sided communication using `MPI_Win_fence` for

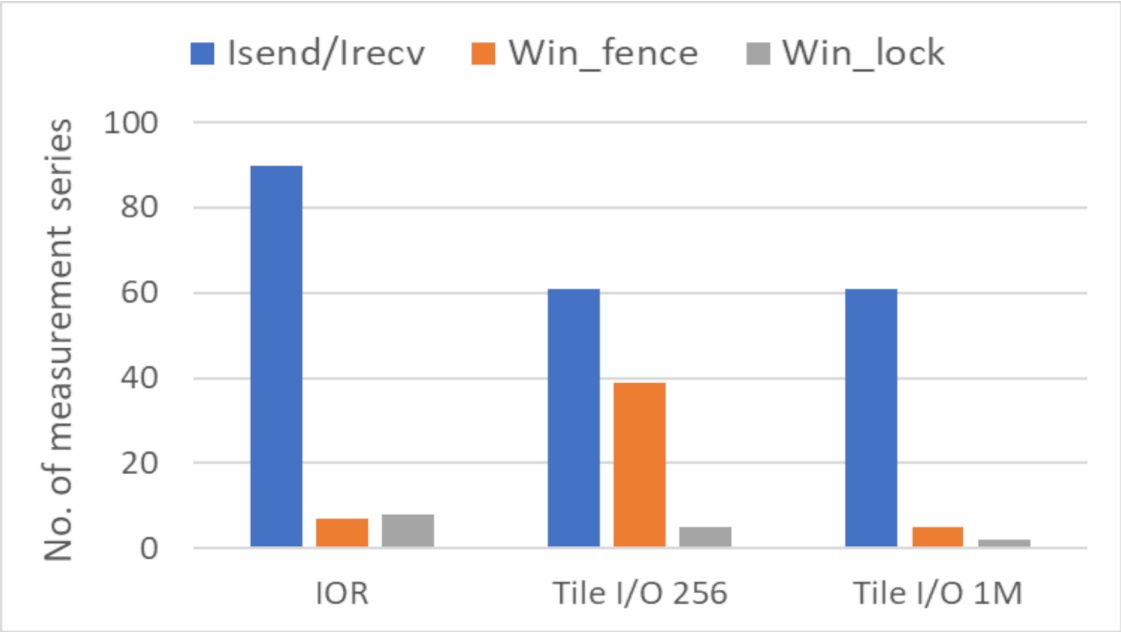

Figure 11: Summary statistics comparing the number of times each of the three different data transfer primitives resulted in the best performance for each benchmark.

synchronization achieved in approx. 37% of the test cases the best performance. On average, the performance gain over two-sided communication was approx. 27% on Crill, and 30% on the Ibex cluster in these scenarios. It will require some further analysis to fully understand reasons for the difference in the performance behavior between this benchmark case and the IOR respectivel the Tile I/O benchmark for 1MB tile sizes. It should be noted however that both of the other two benchmarks operate on significantly larger, contiguous memory regions than the Tile I/O 256 benchmark does, which contains many, smaller, dis-contiguous data elements.

There was furthermore another interesting trend in this analysis. The benefits of using one-sided communication increased for larger process counts on the Crill cluster. Only one measurement series out of 80 test cases showed benefits when using one-sided communication for tests using less than 256 processes on Crill. For tests using 256 processes or more, 35 out of 84 test cases lead better performance when using one-sided communication over the two-sided counterpart.

4 Multi-threaded Optimizations for Individual MPI I/O

In this chapter, we explore the benefits of multi-threading in MPI I/O, representing a major step towards developing a unified MPI+threads based file I/O model. We demonstrates benefits of using multiple threads for both phases of MPI I/O operations, namely the internal processing part performing the file offset calculations on each process, and the actual read/write operations. The main focus of this work is on individual I/O operations. We evaluate the performance of the redesigned functions using multiple benchmarks and on multiple platforms, demonstrating significant performance improvements for many scenarios over the original, single-threaded version. In Section 4.1, we present our design and implementation of the multi-threaded File I/O solution by focusing on each I/O phase separately. Then, we present our performance evaluation and analysis in Section 4.2.

4.1 Design and Implementation

In this section we discuss the technical challenges faced while incorporating support for multiple threads into the MPI I/O library functions, and the solutions derived during this process. We will be discussing the Build IO and the File Access phase separately.

4.1.1 Build IO Phase

As discussed in Section 2.3, the overall goal of the Build IO phase is to generate the list of triplets of (memory address, file offset, length) used subsequently as arguments during File Access phase by the actual read/write operations. Since the Build IO phase and the File Access phase are performed in multiple cycles, the main challenge in deriving parallel version of the code is the data dependence between one cycle and its predecessors: both file view and buffer view in cycle i will continue where the previous cycle $i-1$ has finished.

Furthermore, the number of elements in the *io_array*, i.e., the array containing the triplets described previously, is not known upfront. This is due to the fact that a data block from the buffer view might have to be split into two or more entries in the *io_array* based on the file view.

The original sequential code reallocates the *io_array* and increases its size whenever necessary, which imposes potentially significant costs for large arrays and frequent re-allocations.

Buffer Management

When dealing with large contiguous chunks of data, memory reallocation of the *io_array* does not cause any performance problems. In fact, each element in the file view and the buffer view represent the largest possible contiguous chunk of data that, i.e., there is no case in which two elements of the same list can be merged into one bigger element. As an example, let us assume that both file view and buffer view are contiguous. In this case a single I/O operation will be issued in which 1) the length is equal to the total size of data, 2) the memory address corresponds to the first byte of the local buffer and 3) the offset in the file view is set to the initial displacement value set by the user when specifying the file view. However, if the data is small and scattered in the main memory representing a nearly random access pattern – as is the case in some real parallel applications from seismic imaging and machine learning, – the size of the *io_array* might have to be adjusted multiple times.

Although the number of elements of the *io_array* cannot be determined before the completion of the Build IO phase in a given cycle, we derived a solution to calculate the upper bound for the size of the *io_array* and thus avoid reallocating the memory of the *io_array* multiple times.

To do so, we first need to identify the worst case scenario that would result in the largest possible number of elements. When an element from the buffer view is compared to an element of the file view and the two elements have two different lengths, the larger element has to be split into two sub-elements. The first sub-element will have the same data length as the smaller of the two elements, while the second sub-element has the remaining data length, and will be treated as a new separate element subsequently. Therefore, the worst case scenario is given whenever an element of the file view or the buffer view has to be split into two new elements during the Build I/O phase. This is also illustrated in the example shown by Fig. 6 in Section 2.3.

In order to derive the formula for the upper bound of the size of the *io_array* we define the

following variables:

- N, M: Number of elements of the file view and buffer view, respectively.
- X_i Y_i: Element at index i in the file view and buffer view, respectively.
- $S(E)$: Size of element E.
- N_{io}: Total number of elements in *io_array*.

For each element X_i, there exist L_i ($L_i \geq 1$) elements in the buffer view that satisfy the inequality defined by (1).

$$\sum_{j=l_{i-1}}^{j=l_i-1} S(Y_j) < S(X_i) < \sum_{j=l_{i-1}}^{j=l_i} S(Y_j) \tag{1}$$

We define l_{i-1} as the index of the first element in the buffer view used by X_i, and l_i as the last element. As we explained in the example, the element Y_{l_i} is always split between the two element X_i and X_{i+1}. Hence, an element X_i is being matched with L_i elements from the buffer view as described by (2) such that l_{i-1} and l_i are respectively the indexes of the first and last elements used.

$$X(i) \longmapsto L_i = l_i - l_{i-1} + 1, \forall i \in [1, N] \tag{2}$$

The total number of the elements N_{io} in *io_array* is then equal to the sum of all L_i elements. Furthermore, since the indexes of the first and last elements of the whole buffer view are known, i.e., $l_0 = 1$ and $l_N = M$, one can derive the formula for the upper bound in the number of elements of *io_array* as defined in (3).

Proof:

$$\begin{aligned} N_{io} &= \sum_{i=1}^{N} L_i \\ &= \sum_{i=1}^{N} (l_i - l_{i-1} + 1) \\ &= N + \sum_{i=1}^{N} (l_i - l_{i-1}) \\ &= N + l_N - l_0 \end{aligned}$$

■

or in its final formulation:

$$N_{io} = N + M - 1 \tag{3}$$

Formula (3) allows to allocate the *io_array* such that no re-allocation will be necessary during execution of the Build IO phase, assuming that the values for N and M are known.

4.1.2 Parallelization of Build IO Phase

As discussed earlier, the Build IO phase calculates the elements of the *io_array* in an iterative manner, with each loop iteration (cycle) processing elements whose lengths add up to a fixed number of bytes. From the high level perspective, the parallelization should occur on the loop level, i.e., each thread is responsible for calculating the elements of the *io_array* for a subset of the loop iterations. The challenge however is that cycle i is dependent on the result of cycle $i-1$, hence there is a dependency between the loop iterations.

To break this dependency, our solution breaks the code of the loop body into two parts: a sequential pre-calculation phase, and a parallel version of loop performing the calculations of the *io_array* without the data dependency. Algorithm 5 depicts the high level organization of the revamped algorithm.

Algorithm 5 Multi-Threaded Build IO

Require: $NbThreads$, $NbCycles$, fh_info, bf_info

Pre-Calculation Step

```
1: for index=0 to NbCycles do
2:     fh_info ← calc_args(file_view, index)
3:     bf_info ← calc_args(buffer_view, index)
4:     set_buildio_args (fh_info, bf_info, index)
```

Parallel Build IO step

```
5: function THREADED_BUILD_IO(threadId)
6:     for index=threadId to NbCycles, step=NbThreads do
7:         args ← get_buildio_args (index)
8:         build_io_array (args)
```

The pre-calculation step determines for each cycle the essential information to calculate the elements of *io_array* in that iteration. More specifically, the pre-calculation step calculates the following items for each of the file view and buffer view:

1. *Element index*: Index of the first element to be processed in this cycle.
2. *Element offset*: Length of data that was already processed within the first element in the previous cycle.
3. *Total offset*: Total length of data that was already processed in all previous cycles.
4. *Number of elements*: Total number of elements to be processed in the current cycle. This parameter corresponds to either the value of N or M introduced in (3).

Note that the items listed above can be calculated without having to perform the full matching of elements in the buffer view to the elements of the file view, making it computationally significantly cheaper than the subsequent loop. Two arrays of size equivalent to the number of cycles are being allocated for the buffer view (depicted as **bf_info** in Algorithm 5), and the file view (**fh_info**), with each element holding the information described above for that cycle.

The main compute loop of the Build IO phase has been modified to use the information provided in those two arrays as a starting point. A thread will execute a certain number of cycles of the

Build IO phase, retrieve the parameters determined during the pre-calculation phase for this cycle – including the parameters N and M required to allocate the *io_array* using (3).

The number of threads can be controlled currently through an Open MPI runtime parameter (*io_ompio_num_threads*). By default, multi-threading is not enabled unless the parameter is set to a value greater than 1. The maximum number of threads allowed is defined as the number of cycles. The threads are currently created upon opening the file or during the execution of the file I/O operation itself. Our prototype implementation also assigns threads to separate cores, using the gcc extensions for thread affinity manipulations. The long term goal however would be to be able re-use the threads available for the computational part of the code.

4.1.3 File I/O Phase

The final step in an individual read/write operation is to perform the actual file I/O itself. Within the OMPIO architecture, file access is abstracted out into the file byte transfer layer (fbtl) [8]. The *io_array* calculated during the Build IO phase is stored as part of the OMPIO file handle structure. Hence, to support multiple threads invoking the fbtl component simultaneously, the OMPIO file handle is replicated on each thread, with the only difference being the *io_array* attached to each structure.

Note furthermore, that the fbtl read/write interfaces operate on absolute offsets into the file, e.g., using `pread/pwrite`, ensuring that the data written by different threads is not interfering with each other. Since the POSIX file handle is however the same, data of different threads might be combined on the operating system level.

4.2 Performance Evaluation

In this section we evaluate the performance of the multi-threaded code versions introduced in this paper. The impact of multi-threading to both the Build IO phase and the File Access phase are discussed separately.

4.2.1 Build IO Benchmarks

Section 4.1.1 described the two components necessary to fully exploit multi-threading during the Build IO phase, namely (i) overhaul of the *io_array* buffer management to avoid re-allocations, and (ii) restructuring of the code constructing the *io_array* into a sequential pre-calculation part, and a parallel multi-threaded part. Generally speaking, the impact of the Build IO phase is larger in use-cases in which both the file view and the buffer view have a large number of elements, and hence result in a large number of operations required to construct the *io_array*.

In order to evaluate the impact of the proposed improvements described in Section 4.1, we designed a benchmark that allows to control the number of elements in these arrays. Specifically, the buffer view is defined as a derived datatype that is based on a strided memory pattern: each element in the buffer is separated from the next element by four bytes. The size of an element can be controlled as an input parameter of the benchmark. In order to minimize the impact of the file I/O operation in these tests, the file view is defined as a single, contiguous block of data. Hence, the number of elements in the resulting *io_array* are fully controlled by the buffer view. The benchmark currently supports individual write operations (`MPI_File_write(_at)`).

Evaluating the impact of the revised buffer management strategy does not necessarily require a parallel platform, and we therefore use a single server for this part of the evaluation. Tests have been executed on the *Eureka* server at the University of Houston, which has two Intel Xeon E5-2640 CPUs, with 6 cores and 12 logical threads each, and 64 GB of main memory. Furthermore, since the main focus of the evaluation at this stage is the optimizations introduced to the Build IO phase, an in-memory **tmpfs** file system has been used to minimize the impact of the file system performance on the data.

4.2.2 Buffer Management

The first set of tests are designed to isolate and evaluate the impact of the changes made to the buffer management, i.e., avoiding re-allocations during the Build IO phase. To evaluate this aspect of the work, a separate version of the Build IO algorithm has been implemented which includes the

pre-calculation step described previously (which is necessary to determine the parameters of the new buffer management code), but executes the rest of the code in a single-threaded mode. We will refer to this version as *Without-reallocation* (or *WR*). Performance differences observed between this version of the code and the original, sequential, version can mostly be attributed to the change in the buffer management strategy and avoiding of re-allocations during Build IO phase.

The benchmark described above was executed on the *Eureka* server using a single process for element sizes of 1, 4, 8 and 16 Bytes. The performance obtained with this version of the code is compared to the execution times obtained for the same benchmark using the original code. The overall size of the output file was 2 GBytes.

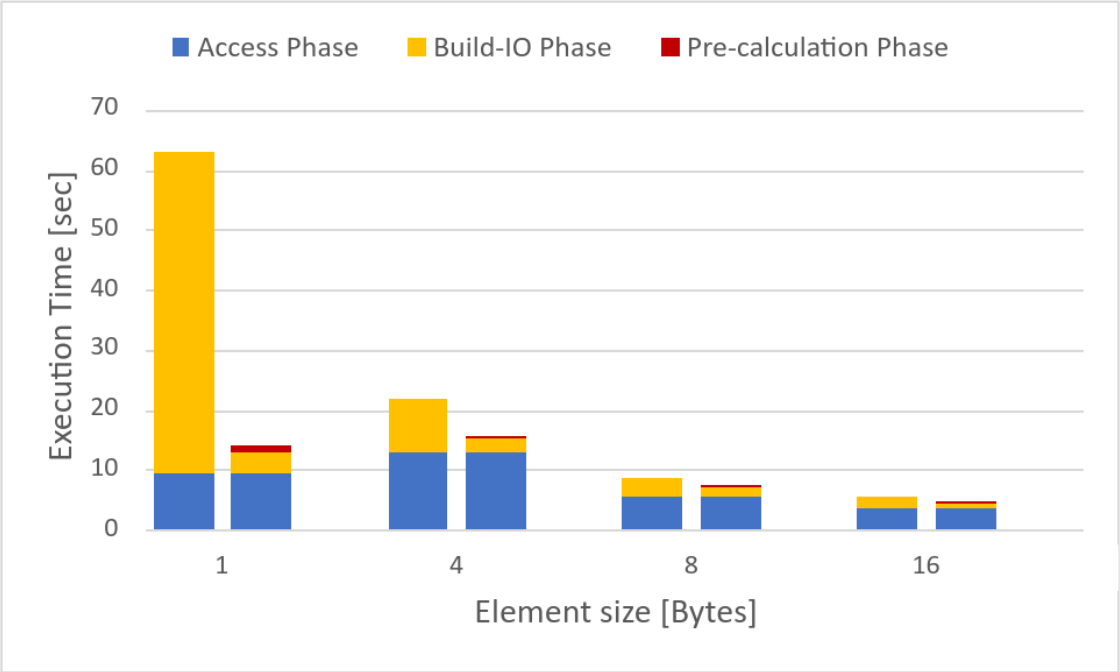

Figure 12: Breakdown of the execution time of individual I/O write operation. The first stacked bar corresponds to the original implementation while the second corresponds to *"Without-reallocation"* version.

The results illustrated in Fig. 12 show significant improvement in the performance of the Build IO phase when using the new buffer management approach, with a performance improvements ranging from 36% to 90%. As expected, the impact of this code change is higher for larger number

of elements in the *io_array*, which for a fixed file size is reciprocal to the size of an individual element in this benchmark.

4.2.3 Multi-threaded Build IO

The second part of the evaluation targets the parallel version of the Build IO phase described in Section 4.1.2. We implemented the full version of the Algorithm 5 in OMPIO [8], the parallel I/O framework used by Open MPI starting from the v2.0 release. Using only a single thread to perform the Build I/O phase results in the *Without-reallocation* version used in the previous subsection. The benchmark was executed using 48 different parameter combinations for different number of threads, resulting in total of 576 runs. For the sake of brevity, we will only present a subset of the results. The benchmark parameters are:

- *Element size:* 1, 4, 8 and 16 Bytes.
- *Cycle buffer size:* 512KB, 1MB, 8MB and 32MB.
- *File size:* 1GB, 2GB and 4GB.

Fig. 13 summarizes the impact of the number of threads on the performance of the Build IO phase. These tests used an element size of 16 Bytes in the buffer view, and a cycle buffer size, i.e., the number of bytes written during a single internal iteration of the `MPI_File_write` operation, of 32MB. The results represent the relative improvement over the original implementation. The performance obtained with the original code version is denoted as having 0 threads in the following graphs.

The results illustrated by Fig. 13 provide three important insights. First, comparing the execution time using one thread to the original code version, i.e., depicted as zero threads in the graph, shows a performance improvement of around 38%, which can be attributed to the change in the buffer management discussed previously.

Second, the execution time of the Build IO phase benefits from using multiple threads and shows significant improvements between two and ten threads. The relative improvement compared

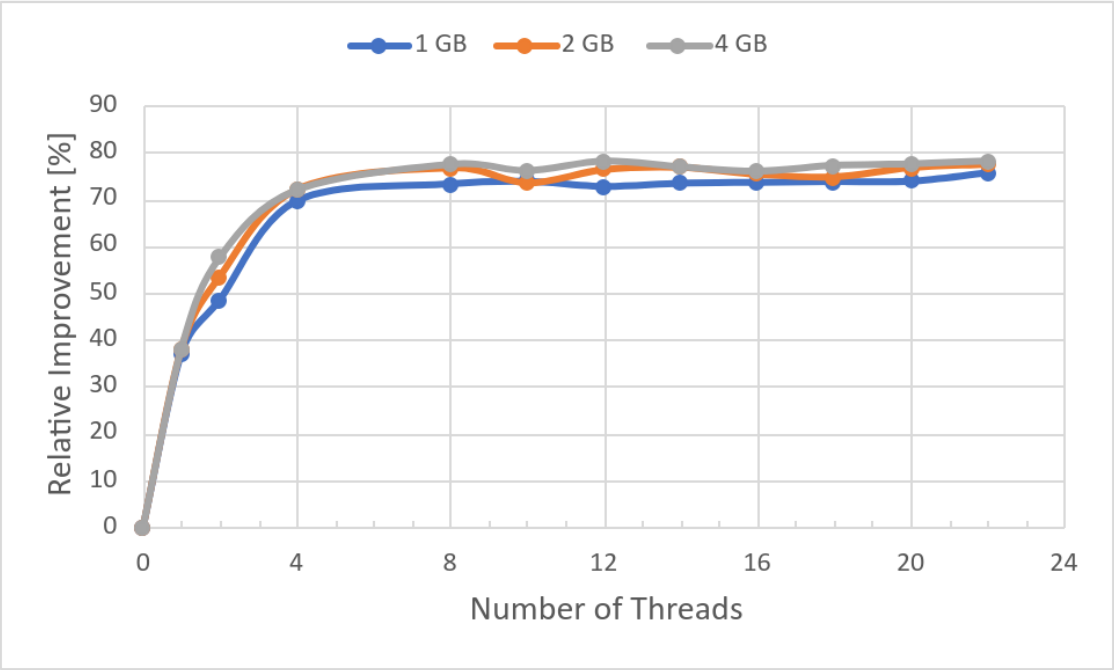

Figure 13: Relative performance improvement of the multi-threaded Build IO version for different number of threads and different file sizes.

to the original version ranges from 48% for two threads up to and 73% for ten threads for a file size of 1GB. Further increasing the number of threads does not improve the performance of the Build IO phase however. This is due to the fact we cannot assign a distinct physical core for each thread anymore. Since the Eureka server has 12 physical cores and 24 logical threads, any additional thread will have to share a physical core with another thread, assuming two threads are reserved for the Operating System and the main thread.

Finally, while increasing the overall file size also leads to larger execution times overall, the relative performance improvement remains nearly constant across the different file sizes.

Fig. 14 shows the execution time of a subset of the tests described for 1, 2, 4 and 8, threads, as well as the execution time of the Build IO phase in the original code, demonstrating the benefits and scalability of the new multi-threaded code described in this paper.

Although our multi-threaded Build IO code has shown a significant improvement in most test

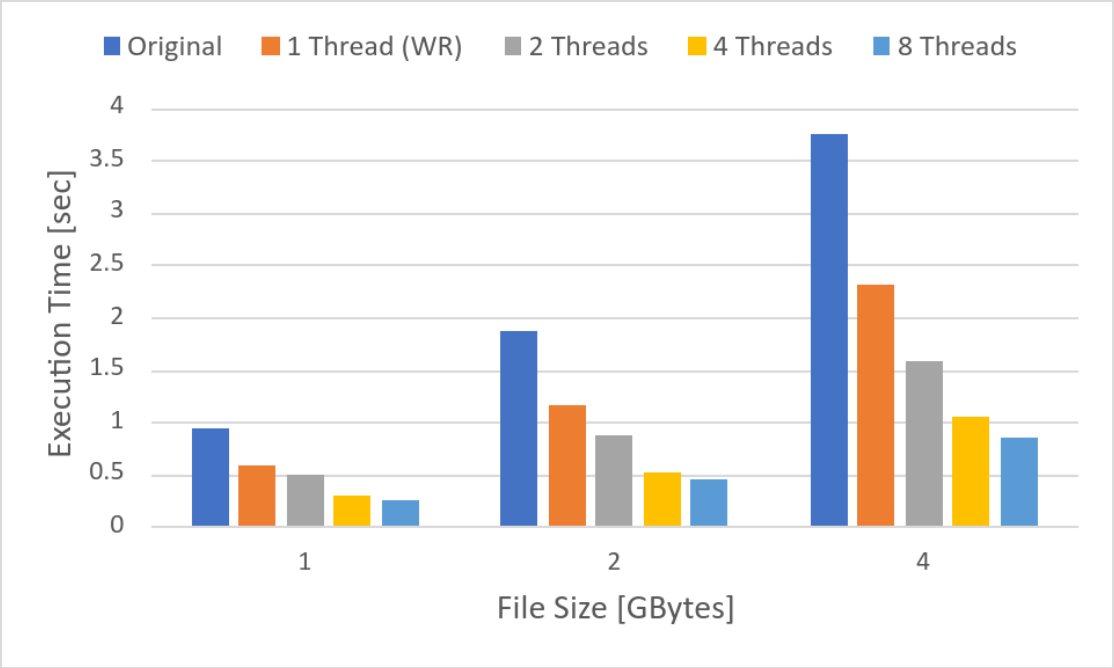

Figure 14: Execution time of the Build IO phase using the original version and the multi-threaded Build IO version for different file sizes.

cases, there were some scenarios which showed a performance degradation when using the multi-threaded code. Theoretically, if the cost of the pre-calculation phase is higher then the gain from the multi-threaded execution of the Build IO loop, the overall performance of the multi-threaded code will most likely suffer. Fig. 15 summarizes the performance improvement compared to the original version as a function of the length of the *io_array* described earlier. The results indicate that there is in fact a lower bound of the number of elements in this array for the multi-threaded code version to show performance benefits.

For lower number of I/O elements per cycle, the average relative improvement is at its lowest values for all numbers of threads. Then, within a certain range of values, approximately between 0.25 and 0.5 million elements, the performance starts to improve exponentially until reaching the peak performance after which we enter a steady state phase.

The graph line describing the *Without-reallocation* version (1 Thread (WR)) shows a huge drop

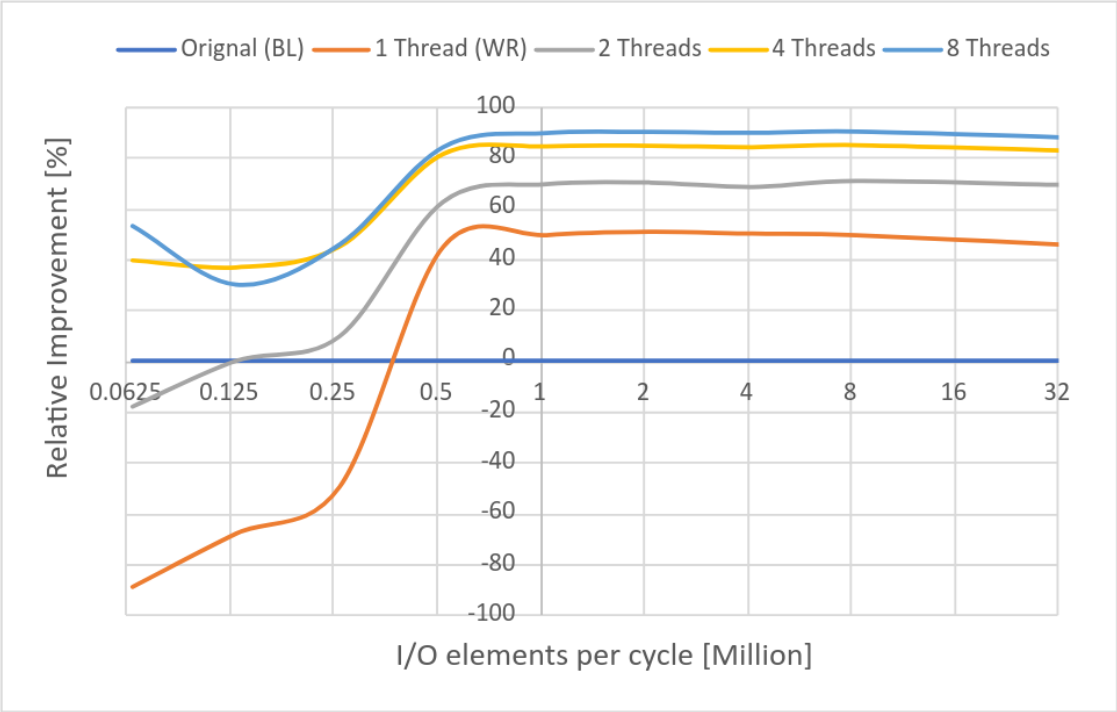

Figure 15: Average relative performance improvement of the multi-threaded Build IO version in function of the number of I/O elements created in each cycle for different number of threads. The baseline (BL) is the original implementation.

in performance for lower number of I/O elements which in fact corresponds to the pre-calculation step overhead. Nevertheless, we notice that for the same range of values, the relative improvement of the same test cases proportionally increases with higher number of threads. The benefits of multithreading improved the Build-IO phase to an extent where the pre-calculation overhead has been completely overshadowed and the overall performance becomes better than the original version in all cases.

For all the 576 test cases, we calculated the total number of I/O elements created in each cycle and measured the relative performance improvement for each data point compared to the original implementation. In case we have multiple data points with the same number of I/O elements, we measured the average value.

4.2.4 Multi-threaded File I/O

In this section, we evaluate the full multi-threaded File I/O version described in Section 4.1.3 that includes the new buffer management algorithm and the parallel, multi-threaded versions of the Build IO and File Access phase.

Fig. 16 presents the results obtained on the Eureka server using the same benchmark as in the previous subsections, including however also the File Access time. For this tests, an element size of 16 bytes, a cycle buffer size of 32MB, and an overall file size of 4GB file has been used.

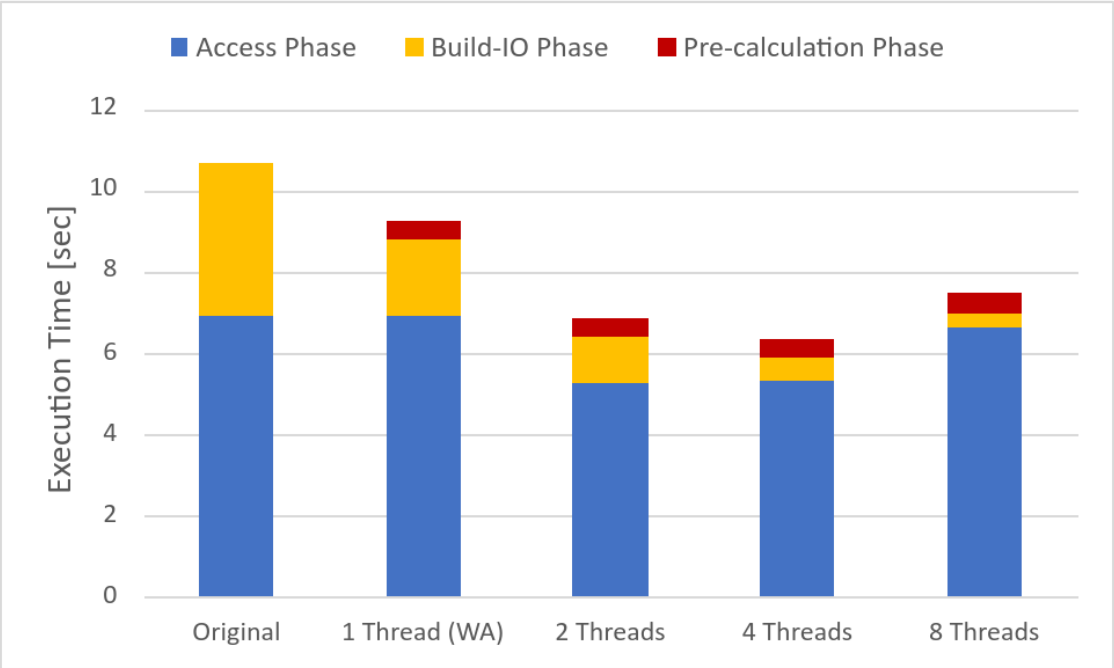

Figure 16: Breakdown of the execution time of individual I/O write operation for different numbers of threads using the original version and the multi-threaded file I/O version on the *Eureka* server.

Fig. 16 shows the results obtained. There is an interesting distinction between the number of threads leading to the best performance of the Build IO phase and for the File Access phase. The best performance for the Build IO phase was achieved using 8 threads, with an improvement of 77.5% compared to the original version. However, for the best performance for the overall benchmark and the File Access phase was achieved using 4 threads. The overall performance

improvement is 40.62% for this scenario. Unlike computation, going up to 8 threads for I/O operations produced a performance degradation that negated the improvement in the Build IO phase.

In the final set of measurements we would like to evaluate the benefits of the new multi-threaded individual MPI I/O functions using a standard MPI I/O benchmark on parallel file system. For this, we used the Interleaved or Random (IOR) [52] benchmark. The tests executed mimicked a 1-D data distribution by using a fixed size of 1GB for both transfer size and block size parameters, and setting the segment count to 1.

The first set of measurements were performed on the *Crill* cluster at the University of Houston using a Lustre parallel file system with 16 OSTs and a stripe size of 1MB. Tests were executed using the original as well as the multi-threaded version of the MPI I/O library using between 1 and 8 threads in each test case, for process counts of 14, 28, and 56 processes - which translate to 1, 2, and 4 processes per node respectively.

The results are summarized in Fig. 17, and show a performance improvement for the multi-threaded version for 14 and 28 processes. This improvement can be attributed to the fact that one or two single-thread processes can not saturate the bandwidth of the file system, and having more threads per process therefore improves the performance obtained with the IOR benchmark. The situation is different starting from the 56 processes test case. In this scenario 56 MPI processes are distributed over 14 nodes, i.e., 4 processes per node, each node using up to 8 threads or a total of 32 threads per node. While this is still lower than the core count of a compute node on the *Crill* cluster, the file system in this scenario is quickly saturated by the I/O operations performed by the various process, and therefore does not show significant benefits for the multi-threaded version. This is consistent and reproducible also for tests using larger process counts on this system, which we omitted for the sake of brevity. Ultimately, since the overall bandwidth of the file system is limited on this machine, only smaller test cases are able to obtain a performance improvement by using a multi-threaded `MPI_File_write` operation.

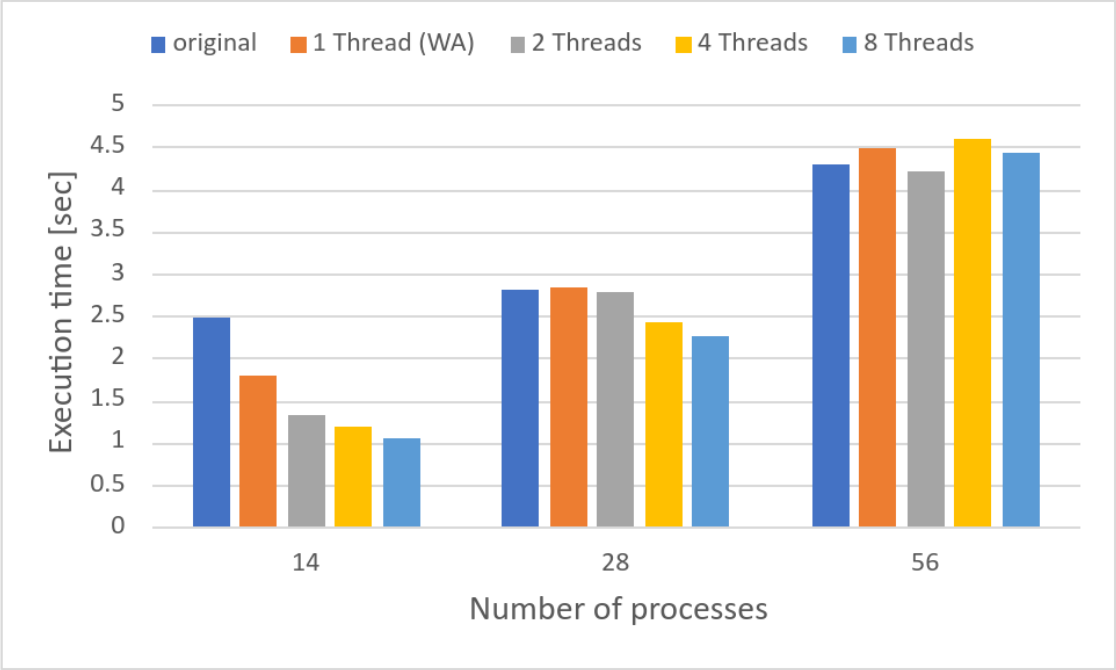

Figure 17: Execution times obtained with the IOR benchmark using individual MPI I/O write operations for different number of threads on the *Crill* cluster.

To extend the analysis performed on the *Crill* cluster, we executed similar tests on a much larger machine with a high-performance, high-bandwidth storage system. The *Shaheen II* cluster at KAUST Supercomputing Centre is a Cray XC40 system delivering over 7.2 Pflop/s of theoretical peak performance. It consist of $6,174$ dual socket compute nodes with 2.3GHz 16-core Intel Haswell processors, and 128GB of DDR4 memory (total of 197,568 cores and 790TB of aggregate memory). Tests have been performed on the *Shaheen II* Lustre file system (v2.12) deployed on a Cray Sonexion 2000 storage system with 17.6PB of total disk space and 144 OSTs. consisting of 12 cabinets containing a total of 5988 4TB SAS disk drives. The cabinets are interconnected by an FDR Infiniband Fabric with Fine Grained Routing, where optimal pathways are used to transport data between compute nodes and OSSes (Lustre Object Storage Service). Our measurements were performed using 32 OSTs and a stripe size of 1MB. The results presented use a fixed number of processes but vary the number of nodes and therefore the number of processes per node.

The results shown in Fig. 18 demonstrate significant performance improvements for the multi-threaded version over the original, single-threaded, sequential implementation of the MPI I/O operation. Consider for example the scenario using 64 processes on 64 compute nodes. This configuration of having only one process per node would be unusual for an MPI only code, but is very typical for the hybrid MPI + threads applications that we are targeting with this work. In this case, using up to 8 threads for the MPI process leads to a performance improvement of 69% over the single-threaded, original version, demonstrating the benefits and the potential of this approach.

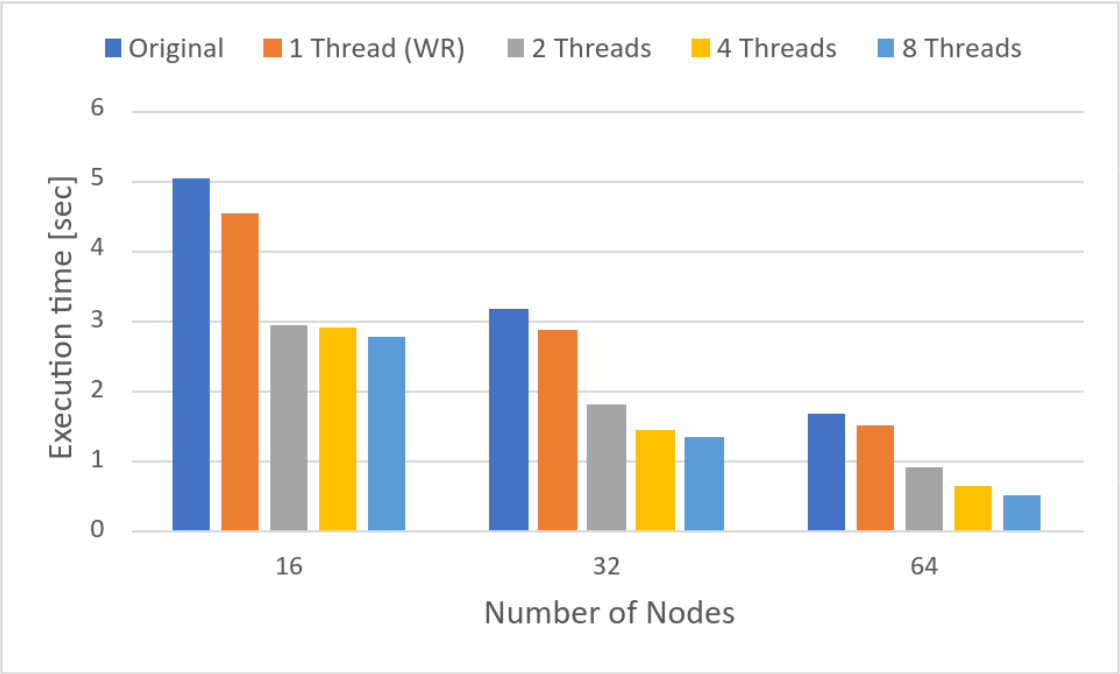

Figure 18: Execution time of IOR with individual I/O write operation using the original version and the multi-threaded file I/O version for 64 processes on the *Shaheen II* platform.

We extended the analysis to use up to 32 threads per process. The multi-threaded version is able to improve the performance of the I/O operations as long as the total number of threads on a node does not exceed the number of physical cores. Having more threads than cores ultimately leads to no further performance improvement or even a slight performance degradation.

Fig. 19 illustrates this observation more clearly. Even though we are using the same number

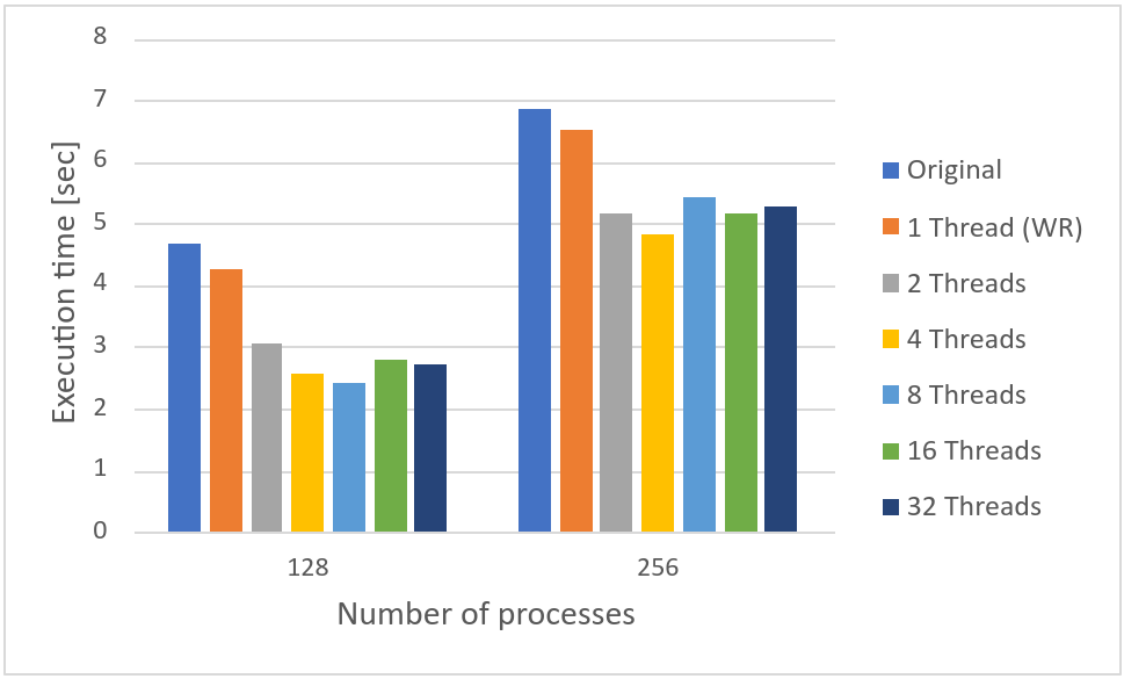

Figure 19: Execution time of IOR with individual I/O write operation using the original version and the multi-threaded file I/O version on the Shaheen platform with 32 Nodes.

of nodes, the peak performance for 128 and 256 processes are not attained for the same number of threads. This is basically related to the number of physical cores attributed to each process. If we consider the case of 128 processes, the number of processes per node is 4. Since each node has 32 physical cores, the number of cores per process is $32/4 = 8$ cores. Similarly, for 256 processes, the number of cores per process is 4. In the two cases, this number is exactly equal to the number of thread per process with which we achieve the best performance. This actually confirms our prediction that oversubscribing the nodes, i.e., using more threads than cores available on a node, lead to a performance degradation in our tests.

Overall, the IOR tests executed on the `Shaheen II` confirm however the benefits using the multi-threaded optimizations in the MPI I/O library, but also the necessity to carefully balance the number of processes and the number of threads per process on a given system.

5 MPI I/O Optimization on Lustre Parallel File System

In this chapter, we focus on another potential source of performance degradation in parallel I/O on the Lustre parallel file system, i.e., file locking. Lustre has introduced a new locking mechanism to enhance the performance of I/O operations. We incorporated this new feature into the MPI I/O implementation by investigating the different possible approaches in supporting it for both collective and individual I/O operations. By doing so, we discovered new further optimizations that can reduce the communication overhead of the current collective I/O implementation for Lustre by adopting a different file domain partitioning. We evaluate the performance of the new MPI I/O implementation for Lustre using different benchmarks demonstrating significant performance improvements for both collective I/O and individual I/O operations.

First, we explain the main issues affecting the performance of collective I/O operation on Lustre and the proposed solutions. Then, we present our design and implementation of the different approaches we adopted of the support of Lustre Lockahead feature for MPI I/O functions in Section 5.1. Finally, we provide in Section 5.2 experimental support of the efficiency of the proposed solutions.

5.1 Design and Implementation

In this section, we start by discussing the major issues of the current collective I/O implementation on Lustre, even with Lockahead, and the proposed solutions. Then, we present the general design of Lustre Lockahead support in MPI I/O and describe the different approaches devised for both collective and individual I/O operations. The description will be based on the OMPIO [8] parallel I/O frameworks of Open MPI [17].

5.1.1 Issues of Collective I/O on Lustre and Solutions

The previous research investigating the cause behind the mediocre performance of collective I/O on Lustre mainly lay the blame on the Lustre locking problem explained in Section 2.5.2. Therefore, current MPI I/O implementations has adopted the static-cyclic partitioning scheme in implementing

the two-phase I/O algorithm while restricting the number of aggregators to match the number of I/O servers. With the introduction of Lustre Lockahead [42], Moore et al. lifted this restriction while preserving the same file domain partitioning method. These optimizations had dramatically improved the performance of the file access phase of the two-phase I/O algorithm. Nevertheless, we argue that maintaining the same static-cyclic partitioning method is no longer beneficial as it will lead to a unnecessary communication overhead that might narrow the overall I/O performance gain.

Fig. 20 provides an example that allows a better understanding of the impact of file domain partitioning on the shuffle phase. Let assume we have a 1-D data distribution over 8 processes from which process 0 and 4 are selected as aggregators.

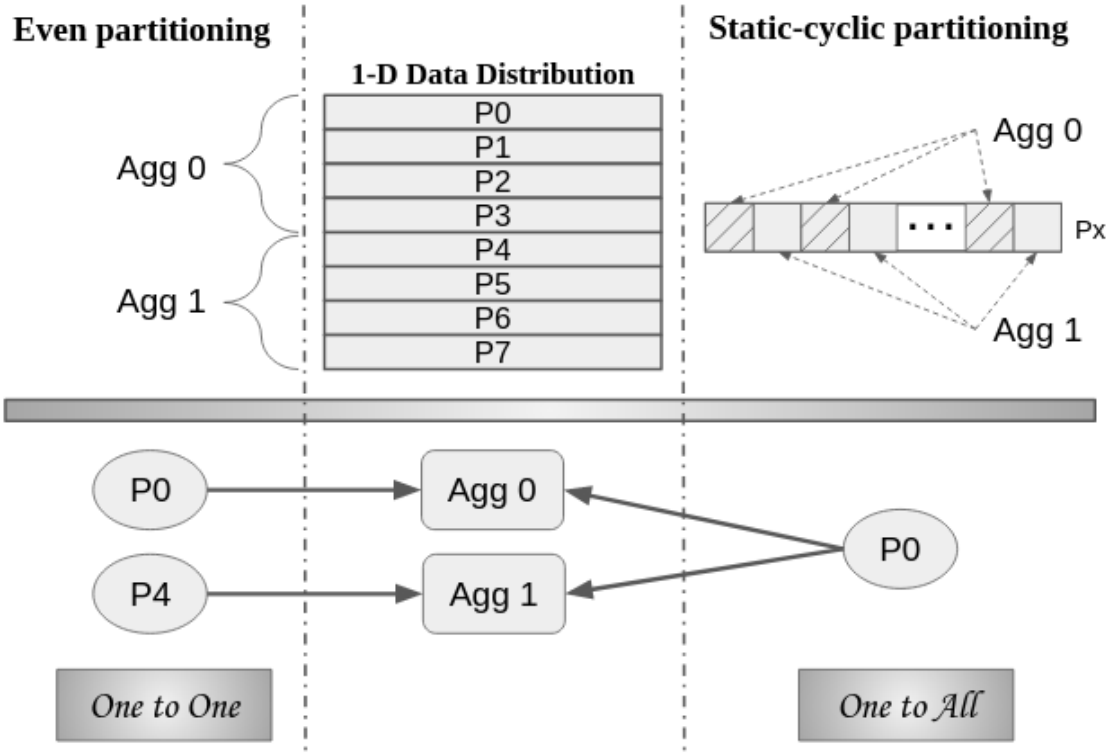

Figure 20: Illustrative Example of the shuffle phase using two file domain partitioning schemes on a 1-D data distribution in a single cycle. The upper part describes the data distribution over the aggregators while the other shows the communication type applied in each case.

In an even partitioning method, each aggregator will be assigned one large contiguous chunk of data. Consequently, Agg 0 (Agg 1) receives its data from the processes P0, P1, P2 and P3 (respectively P4, P5, P6 and P7). In one cycle, an aggregator need to receive a chunk of data that exactly fits its collective buffer. Since its size is usually smaller then the data held by one process, only one process is involved in the send operation resulting in a *one to one* communication.

On the other hand, in a static-cyclic partitioning scheme, the data is assigned to the aggregators following a strided access pattern. The data size in one process is large enough to fill the collective buffer of both aggregators for several cycles. Therefore, in a single cycle, one process has to send its data to all aggregators resulting in a *one to all* communication.

With higher number of aggregators and/or larger data size per process, the communication cost will exponentially grow in case of the static-cyclic partitioning while remaining linearly reasonable for the even partitioning method. Therefore, we postulate that we can avoid the file locking overhead and exploit the full bandwidth of Lustre I/O servers using Lustre Lockahead while profiting from lower communication cost by adopting an even partitioning scheme.

5.1.2 Lustre Lockahead in MPI I/O

In general, a parallel application starts by collectively opening a file for all the processes in a communicator group using `MPI_File_open`. The file might be already existing. If a Lustre client had previously performed I/O operations on the file, the client-side locks would have been saved to the LRU cached locks queue which can cause a lock conflict with the subsequent Lockahead request.

For this reason, we start by clearing all existing locks by locking and immediately unlocking the file descriptor using *Lustre Group Locking*. It is a Lustre locking mechanism that enables a group of processes to disable LDLM locking on a file through the *ioctl()* system call. Afterwards, we disable the lock expansion feature by passing the LU_LADVISE_LOCKNOEXPAND advise which internally sets the appropriate LDLM flag on each subsequent lock request.

Within the OMPIO architecture, file management operations, e.g., opening and closing a file,

are abstracted out into "file system" (fs) framework. The framework implements a separate module called "Lustre" to handle the specificity of the file system such as setting stripe count and stripe size. Since these preliminary steps are required for all processes independently of the I/O operation, we chose to incorporate it into the "fs" open interface for Lustre.

In the following, we propose the different approaches we devised in implementing Lustre Lockahead for both collective I/O and individual I/O operations.

Lockahead in Collective I/O

We investigated three different approaches in implementing the Lockahead support for collective I/O operations. Each implementation uses a different technique in defining the Lockahead advises and issuing the locking requests.

1. *Global file domain request*: In an even partitioning method, each aggregator handles one large contiguous chunk of data defined by its size called "Domain Size" and its offset in the file. Therefore, we can create for each aggregator a single Lockahead advise defining the file domain and issue one global request, and hence the method name.

2. *Request per stripe*: In this method, we create one Lockahead advise for each Lustre stripe. Each aggregator will hence have an array of advises covering all the stripes defining its file sub-domain. Then, we issue a request for the whole array.

3. *Request per cycle*: In the other two methods, we invoke all the Lockahead requests before starting the two-phase I/O algorithm. However, in this implementation, we define and issue one Lockahead advise within the access phase for each cycle. In fact, the size of an I/O operation in a cycle is limited to the size of the collective buffer allocated for each aggregator. Therefore, each request has to match the exact size of the collective buffer.

Lockahead in Individual I/O

The performance of individual I/O in Lustre, in particular the write operations, is strictly dependent on the OSTs capabilities and the maximum number of clients they can handle. Increasing

the number of processes, and hence of writers, would be even more harmful to I/O performance then collective I/O with an even partitioning method. Therefore, we extended our work to support Lustre Lockahead for individual I/O operation.

Unlike collective I/O operations, the data does not necessary follow a uniform access pattern like the strided data of the static cyclic partitioning method or the contiguous data of the even partitioning method. For this reason, we had to find the appropriate way in defining the Lockahead requests for non-uniform access pattern. We designed two different solutions to approach this problem:

1. *Request per I/O element*: In the *Build IO* phase, each element of the created *io_array* represents an atomic I/O operation of a single contiguous block of data. For each cycle, we create an array of Lockahead advises that exactly matches the I/O elements then issue a global request for the whole array before starting the I/O operations in the *File Access* phase.

2. *Request per file view element*: The MPI derived datatypes describing the file view are internally decoded into one array of file view elements. Each element typically provides relative displacement to the beginning of the file for each contiguous data block. Therefore, we can create a Lockahead advise for each element and issue all the requests within the `MPI_File_set_view` function.

5.1.3 Data Alignment

A Lustre client can request multiple non-aligned Lockahead advises of different sizes. Nevertheless, the Lustre server would still handle them on a stripe basis by granting a lock of size equal to the smallest multiple of a stripe size that is greater than or equal to the lock size.

On the one hand, the data handled by each client, e.g., the domain size of an aggregator in collective I/O, has to be aligned to the Lustre stripe size. Otherwise, when two adjacent non-aligned file sub-domain are overlapped at one stripe, it will result into a concurrent access to a same stripe by two different clients, and hence conflicting lock requests. In fact, This would have a huge impact

on the I/O performance that negates the benefits of Lustre lockhead.

On the other hand, two consecutive I/O operations at the level of one aggregator with a non-aligned collective buffer, might also overlap at one stripe for each two consecutive cycles. While it does not have a significant impact on the locking mechanism as it occurs within the same Lustre client, it will cause unnecessary extra I/O operation. The same fact stands for the local individual I/O buffer.

To solve these problems, we adjusted the calculation of the domain size as well as the setting of the collective buffer size and the individual buffer size parameters to be aligned the Lustre stripe size .

5.2 Performance Evaluation

In this section, we evaluate the performance of the new MPI I/O design supporting Lustre Lockahead for both individual and collective I/O operations.

Within the OMPIO architecture, collective I/O is abstracted out into the file collective layer (fcoll) [8]. The default fcoll component selected for Lustre parallel file system, called dynamic_gen2, implements a static-cyclic partitioning method while the new implementation is actually based on another component, i.e., "vulcan" as it uses an even partitioning method.

The experiments were performed on the *Shaheen II* supercomputer at KAUST Supercomputing Centre. To avoid any potential source of differences in performance, all measurements were performed using 64 nodes and 32 OSTs with a stripe size of 1 MiBytes.

In order to evaluate the different approaches, we mainly used the Interleaved or Random (IOR) [52] benchmark. Unless mentioned, we set the data size to 256 MiBytes per process. In addition, we also used the MPI-TILE-IO [47] benchmark to test parallel I/O operations on two dimensional dense data sets. The number of tiles is equal to the square root of number of processes. Each tile consists of 64×16 elements of 1 MiBytes resulting into 1 GiBytes of data per process. We performed each measurement described in this section at least 3 times then we extracted the mean value.

5.2.1 Impact of File Locking and Partitioning on Collective I/O

In Section 5.1.1, we investigated the primary reasons for the poor performance of the current MPI collective I/O implementation on the Lustre parallel file system. We believe that adopting an even partitioning method instead of the original static-cyclic partitioning scheme would lead to optimal I/O performance if blended with the benefits of Lustre Lockahead.

To support our theoretical analysis, we evaluated the impact of the two file partitioning method on the cost of each phase of the two-phase I/O algorithm and explored the benefits of Lustre Lockahead with an even partitioning method. To perform these measurements, we modified the source code of the "dynamic_gen2" and "vulcan" fcoll components by removing the internal overlapping method [15] and setting a single collective buffer. By doing so, we are able to measure the cost of each phase separately and eliminate an external factor affecting the overall performance of the algorithm, i.e., the overlapping technique. Then, we implemented a third version based on the same source code of the "vulcan" component that implements an even partitioning method but with Lustre Lockahead support. Finally, we run the IOR benchmark using the three adjusted components and extracted the execution time of each phase on a per process basis and calculated the mean values. Note that the Lockahead implementation uses the same partitioning method as the vulcan component which implies an identical communication cost.

In the first set of measurements, we evaluated the effect of the number of aggregators on the communication cost for each file partitioning method.

The results shown in Fig. 21 confirm our theoretical predictions. In case of the static-cyclic method, increasing the number of aggregators puts more load on the shuffle phase since each process has to transfer its local data to a higher number of processes in each cycle which causes a higher execution time. However in an even partitioning method, the time spent on the communication phase keeps decreasing until hitting a "close to zero" value which is the case of setting all processes as aggregators. Basically, during the whole collective I/O operation, an aggregator receives the data only from N_{agg}/P processes where N_{agg} is the number of aggregators and P is the number of processes. Consequently, when all the processes are aggregators, each process will actually perform

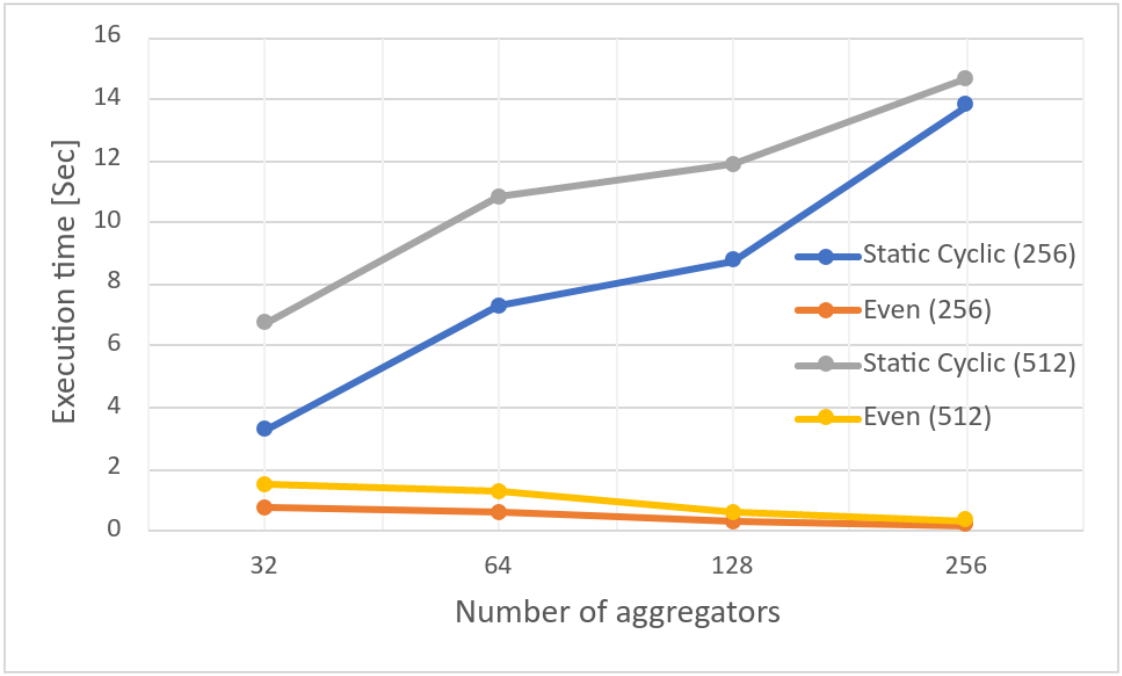

Figure 21: Execution time of the shuffle phase within an MPI I/O collective write operation for different number of aggregators using the "Static Cyclic" and the "Even" partitioning schemes for 256 and 512 processes.

a send-to-self communication, i.e., a memory copy of its local data to its local collective buffer.

In the second set of measurement, we run the IOR benchmark with 1024 processes using 32 aggregators matching the exact number of OSTs while increasing the data size per process.

From Fig. 22, we notice that the original two-phase I/O implementation for Lustre, using the static-cyclic partitioning method, shows the lowest performance of all versions except for the smallest output file. In fact, the execution time of the shuffle phase in this version is abruptly increasing from 0.6 seconds to 30.86 seconds with larger data sizes going from 16 to 256 GiBytes. Although it was supposed to be the most efficient implementation, the communication overhead caused a huge drop in overall performance of the collective I/O operation.

Basically, the goal from adopting a static-cyclic partitioning method is to reduce the Lustre file locking overhead as explained in Section 2.2.2. The results confirm this statement as the execution time of the file access phase with the original version is always the lowest compared

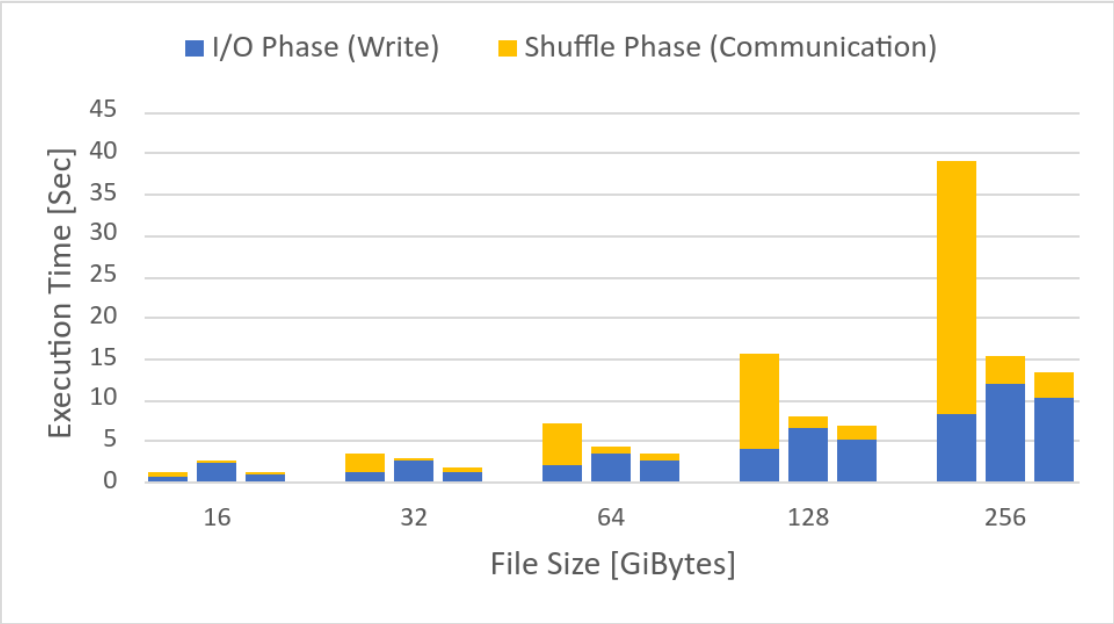

Figure 22: Breakdown of the execution time of collective I/O write operation using 32 aggregators. The stacked bars respectively correspond to : 1) static-cyclic partitioning, 2) even partitioning and 3) even partitioning+Lockahead (Global request version).

to the other versions. When using the component implementing the even partitioning method, multiple aggregators are concurrently accessing the I/O server causing a file locking overhead. This overhead increases the I/O phase cost by at least 70% as in the case of 64, 128 and 256 GiBytes file sizes. For a file size of 16 GiBytes, it even multiplied it by an approximate factor of 4×: the I/O phase execution time went from 0.54 seconds using the static-cyclic method to 2.35 seconds using an even partitioning method.

Nevertheless, with the introduction of Lustre Lockahead, we benefit from the lower communication cost of an even partitioning method while reducing the locking overhead to the minimum. The results indicate that the execution time of I/O phase using this partitioning method combined with the Lustre Lockahead support is slightly higher than the original version by approximately 24% in most cases. Yet, it achieves the best performance in term of the overall execution time of the collective I/O operation.

5.2.2 Lockahead in Colletive I/O

In this section, we evaluate in more depth our design for supporting Lustre Lockahead in collective I/O operations. We implemented the three Lockahead versions proposed in Section 5.1.2 along with the necessary data alignment optimizations described in Section 5.1.3. Then, we run the IOR benchmark for each version using both asynchronous and synchronous Lockahead requests.

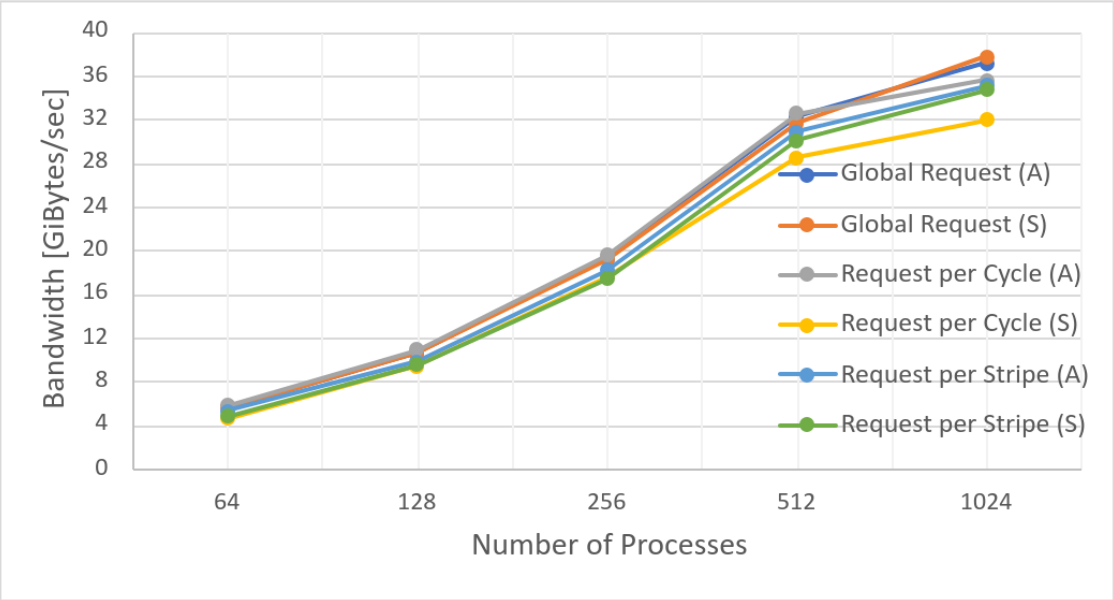

Figure 23: Comparison between the three proposed versions of collective write operations with Lockahead using both (A) asynchronous and (S) synchronous requests.

Fig. 23 presents the bandwidth achieved by each version. At first sight, we can easily identify the *"global request"* version i.e the version that uses the even partition method in which each aggregator issue one large Lockahead advise, as the best approach. For 64 processes, all the versions have similar write bandwidth of approximately 5.6 GiBytes/sec. Nonetheless, the differences become more apparent with higher number of processes. For 1024 processes, the difference in bandwidth between the best approach and the second, i.e., *"request per stripe"*, is around 2 GiBytes/sec. In the "request per stripe" version, each aggregator have an array of Lockahead advises that covers its file sub-domain such that each advise is associated to one stripe. The only difference between

these two versions is hence the number of Lockahead advises which explain the slight edge of the global request version.

Furthermore, we notice that the requests synchronicity did not have any impact on the performance of these two versions. While they have different number of Lockahead advises, the requests are issued at once using a single call of the *"llapi_ladvise()"* function. This is not the case with the *"request per cycle"* version in which the creation and issuance of the Lockahead advises are executed in each cycle. We can hence assume that asynchronous Lockahead requests are only beneficial for this version. As a consequence, one can argue that the global request approach provides more flexibility to the users as they can freely choose between the benefits of blocking and non-blocking Lustre requests, explained in Section 2.5.3, without harming the overall I/O performance. For the rest of the evaluation, we only used this Lockahead version.

Lustre Lockahead enables parallel I/O access of multiple aggregators to the same OST without affecting the I/O performance. Therefore, we believe that the default setting of the number of aggregators is no longer optimal as the only limiting factor would be the upper-bound performance of a single OST. A complete study of the impact of Lockahead on the number of aggregators is beyond the scope of this paper. Nevertheless, we performed some preliminary measurements to support our claim. We run the IOR benchmark with 1024 processes using the collective I/O implementation of the even partitioning method with and without the Lockahead support while increasing the number of aggregators.

Fig. 24 shows similar behavior for both versions with two peaks at 256 and 512 aggregators. However, the bandwidths at the two peaks of the implementation without Lockahead were almost identical with approximately 52 GiBytes/sec. On the other hand, the optimal performance of the Lockahead version were achieved in case of 512 aggregators, 97 GiBytes/sec of bandwidth compared to only 80.29 GiBytes/sec in the case of 256 aggregators. The results indicate that using Lockahead, the number of aggregators in collective I/O operations can be increased beyond the stripe count of the Lustre file system.

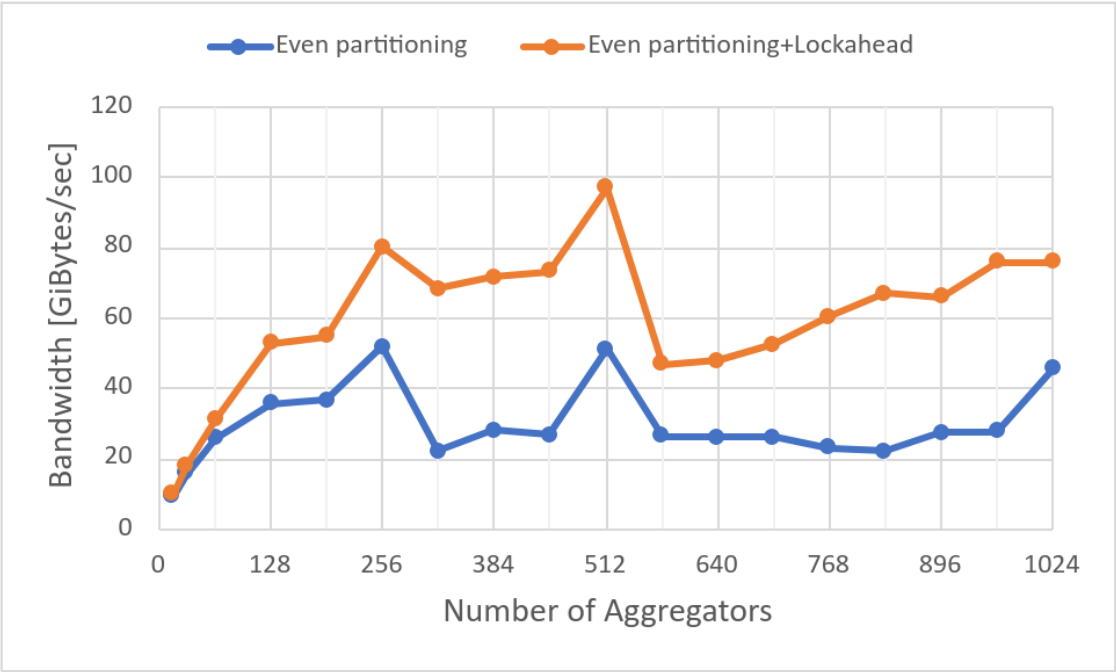

Figure 24: Impact of Lockahead on the default setting of number of aggregators for 1024 processes.

As a final step, we evaluate the global request version and compare it to the original OMPIO component for Lustre that uses a static-cyclic partitioning method and the default component implementing an even partitioning method without Lockahead using both IOR and Tile I/O benchmarks. For the Lockahead version, we set the number of aggregators to the half of the number of processes while maintaining the default configuration for the other versions.

The new Lockahead implementation shows a significant performance improvement in all instances for both benchmarks. In the case of the 1-D data decomposition presented by Fig. 25, the Lockahead version achieves a speedup ranging between 1.63× up to 3.63× compared to the same implementation without Lockahead corresponding to the performance gain from the removal of file locking overhead and the increased number of aggregators within the file access phase. When compared to the original implementation, i.e., the default Lustre component, the performance gap keeps increasing with the number of processes with a minimum speedup of 2.69× for 64 processes and up to 12.9× for 1024 processes. These outstanding results are attributed to two factors which

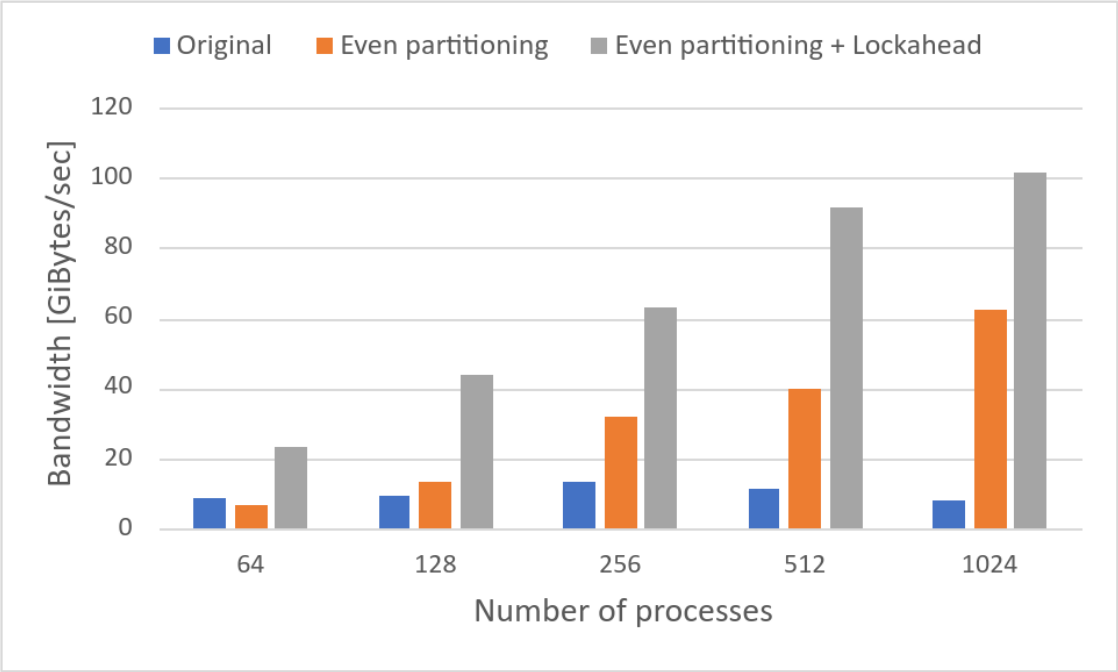

Figure 25: Performance comparison of the three collective I/O implementations with a 1-D data decomposition.

are the removal of the communication overhead and the parallel access of multiple writers to each OST.

The results depicted in Fig. 26 shows similar results with a 2-D data decomposition. Because of the communication overhead, the execution time of the original implementation is exponentially increasing in function of the number of processes. Starting from 576 processes, the original design fails due to long workloads resulting in a time out of internal requests. This proves that the new Lockhead version is more scalable in terms of number of processes and data size. Actually, its execution time follows a polynomial trend with a small variation compared to the one without Lockahead support that is linearly increasing with a slope of 5.26 seconds.

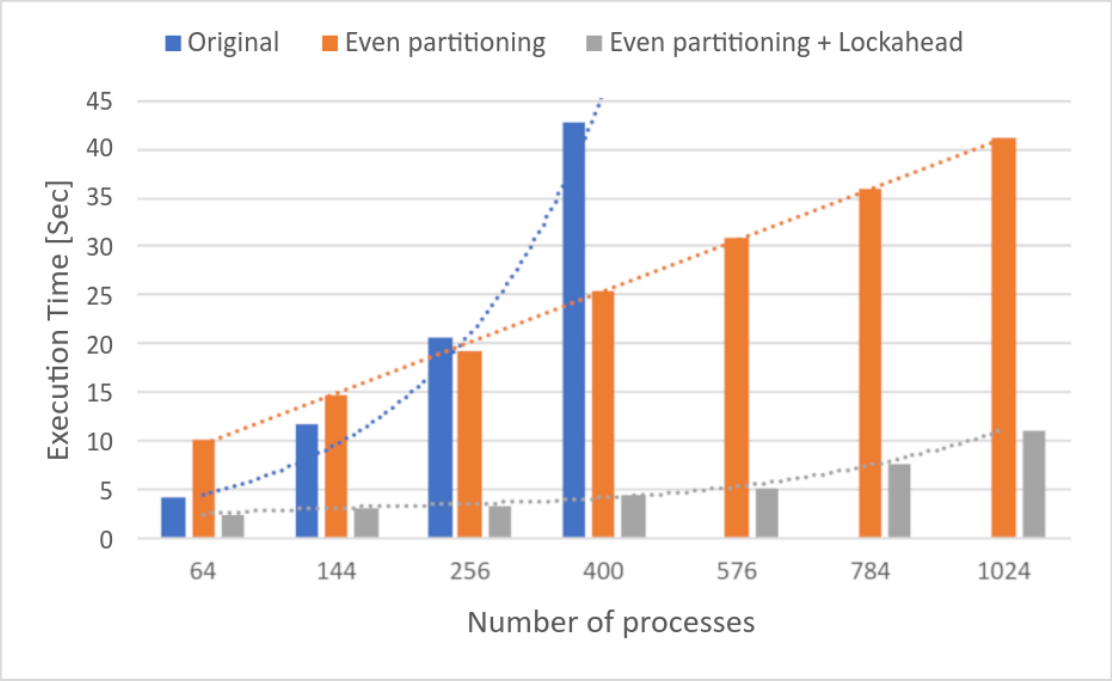

Figure 26: Execution time of the three collective I/O implementations with a 2-D data decomposition.

5.2.3 Lockahead in Individual I/O

The final part of the evaluation targets individual I/O operations. First, we run the IOR benchmark with the original implementation and the two Lockahead versions introduced in Section 5.1.2 using both asynchronous ans synchronous Lockahead requests.

Fig. 27 provides two important insights. First, the two Lockahead versions achieve similar performance improvement except for the case of the "Request per I/O" version with synchronous Lockahead requests. This is very similar to the case of the "Request per Cycle" version in collective I/O in which the requests are not created and issued at once like in the other versions but for each single cycle. Second, we notice that the relative performance improvement keeps decreasing with higher number of processes with a maximum speedup of 3.4× for 64 processes and minimum of 1.76× for 1024 processes.

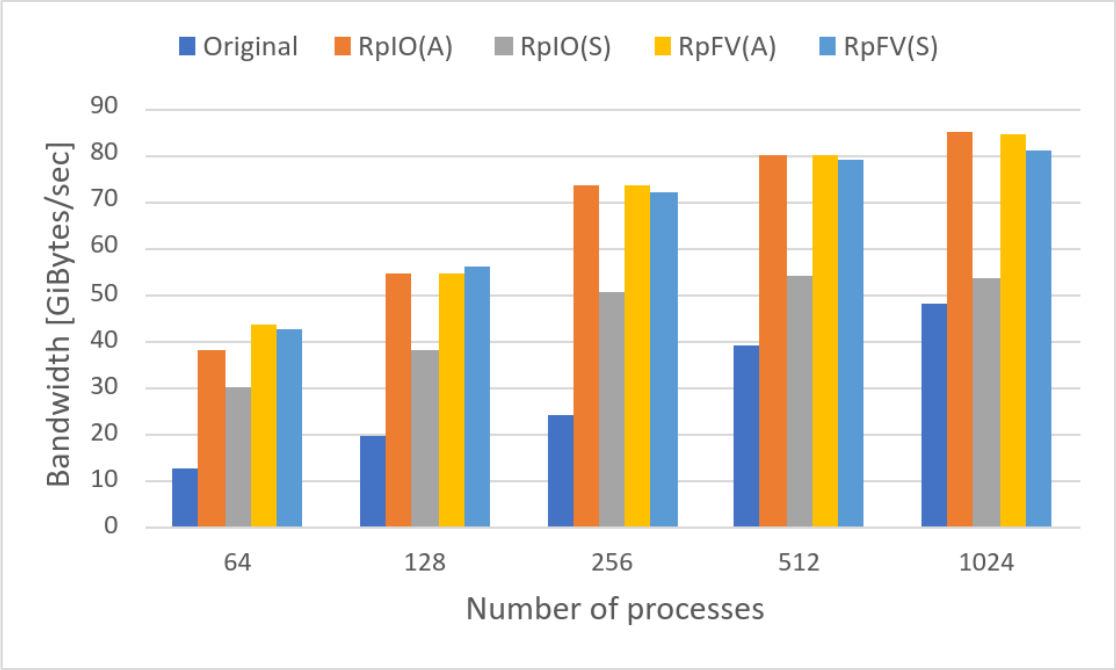

Figure 27: Performance comparison between the original individual I/O implementation and the two proposed Lockahead versions (RpIO) "Request per I/O element" and (RpFV) "Request per file view element" using both (A) asynchronous and (S) synchronous requests.

The same behavior was observed in Fig. 28 using the tile-io benchmark but with higher performance gain. This is explained by the huge overhead caused by the Lustre file locking problem which actually has more impact on individual I/O operations with 2-D data decomposition.

In fact, a file within the tile-io benchmark is seen as a 2-D matrix in which each process holds a non-contiguous tile of data. Therefore, a process cannot issue one global file system write operation as with 1-D data but rather split it into multiple I/O requests on contiguous chunks of data matching the rows of the tile. Consequently, the number of lock cancellations rises causing more locking overhead in the original implementation. With the introduction of Lustre Lockahead, the file locking overhead is fully removed which explains the substantial speedups ranging between 2.62× and up to 33.41×.

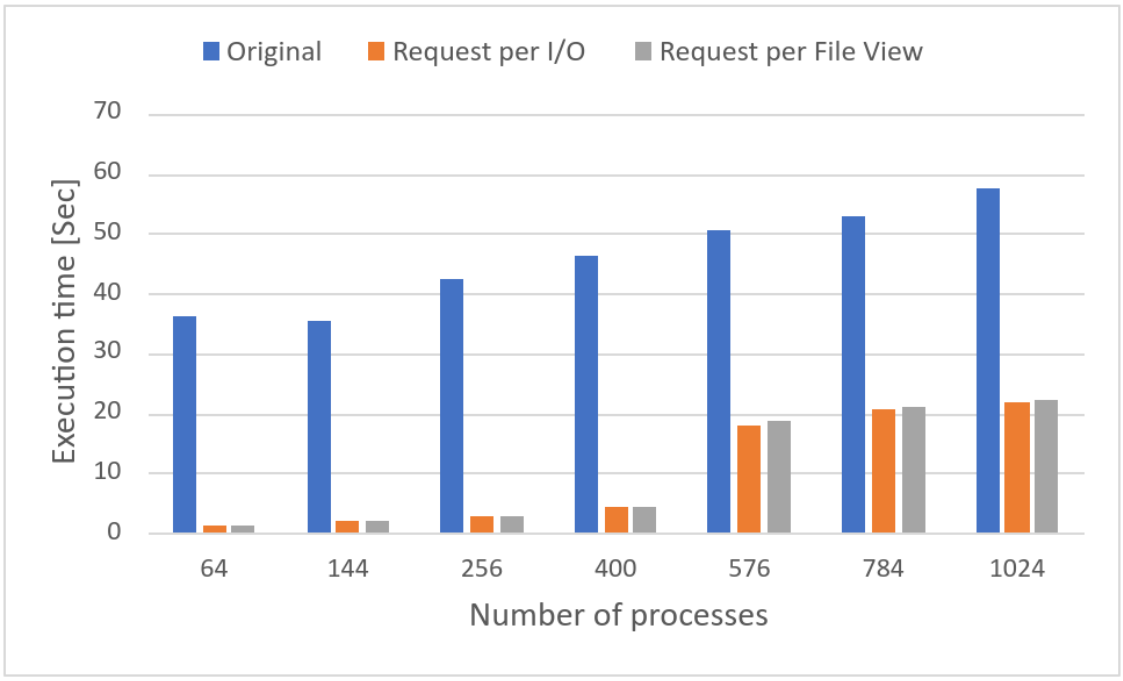

Figure 28: Execution time of individual I/O using the original implementation and the two proposed Lockahead versions with a 2-D data decomposition.

6 Summary and Future Work

With the increasing need for massively parallel computation in different fields, the HPC systems have evolved at an incredible rate. Nevertheless, the development of storage systems could not keep pace with it. Therefore, I/O overhead has been always the primary cause of performance degradation in parallel applications. Historically, file I/O was limited by the latency and bandwidth of magnetic hard drives based storage devices. In recent years, developments in storage technology lead however to a dramatic shift: solid state disks, burst buffers, and non-volatile memory (NVME) such as Intel Optane Technology that have had significant impact on the storage field, and have reduced the discrepancy between file I/O and network I/O latency.

These revolutionary changes has reshaped the current perspective of parallel I/O problems. While the actual file I/O operations, i.e., read/write were the main source of performance degradation because of the high latency of the traditional storage systems, the introduction of these new technologies has revealed other potential sources of overhead within the different layers of the I/O software stack.

In this dissertation, we targeted three specific aspects that have a significant impact on the performance of parallel I/O in Low latency storage systems. In the following, we summarize the contribution of each aspect. Then, we present the potential future works that can enrich the proposed solutions.

6.1 Summary of Contributions

The first research topic discussed in Section 3 targeted the inter processes communication overhead. We presented and evaluated various design options for overlapping two internal cycles of the two-phase I/O algorithm. We also explored two communication models using different data transfer primitives for the shuffle phase, namely non-blocking two-sided communication, one-sided communication using active-target synchronization, and one-sided communication using passive-target synchronization. Our results indicate that overlap algorithms incorporating asynchronous

I/O operations outperform overlapping approaches that only rely on non-blocking communication, and offer significant performance benefits of up to 22% compared to two-phase I/O algorithm not overlapping any internal operations. In the vast majority of the testcases however, using one-sided communication for the shuffle step did not lead to performance improvements compared to two-sided communication.

In the second topic explained in Section 4, we focused on the cost of the internal computation phase of individual I/O operations. We first presented the redesign of individual MPI I/O operations to incorporate utilization of multiple threads. Then, we described the changes necessary to the both the Build IO and the File Access phase and we evaluated every enhancement separately on different platforms. The multi-threaded Build IO code shows benefits between 38% and up to 90% for very fragmented memory and file access patterns that result in many, small, discontinuous data blocks. The multi-threaded File Access phase shows improvements of up to 69% on high performance parallel file system and storage environments. Overall, the key finding of this work is that the performance of individual MPI I/O operations can be enhanced through multi-threading. Not only can we reduce the overhead caused by the computation phase, but also unlock optimal performance for actual I/O operations.

In the last topic, we shed the light on another potential aspect that might harm parallel I/O performance which is File locking. In Section 5, we presented a high performance implementation of MPI I/O on the Lustre parallel file system based on the new Lockahead feature. Although the previous research has shown that the best approach in implementing collective I/O operation for Lustre has to follow a static-cyclic partitioning method, we demonstrate that such partitioning scheme might result into a communication overhead. We proposed a new approach in which we combine the benefits of Lustre Lockahead, i.e., improving the I/O operations and the low communication cost of the even partitioning method to design a more efficient solution for collective I/O. The results has shown outstanding performance improvement compared to the original implementation with a speedup ranging between 2.69× and 12.9×. We also implemented a Lustre Lockahead support for individual I/O operations. By doing so, we managed to fully remove the

file locking overhead which explains the substantial achieved speedups ranging between 2.62× and up to 33.41×.

6.2 Research Perspective and Future Work

In this dissertation, each research topic has targeted a different instance of the software latency exposed in low latency storage systems. Nevertheless, we can combine the benefits of all proposed solution in the process of creating a high performance solution of parallel I/O. This work can be extended in multiple directions by solitary enhancing each solution.

In case of the overlapping collective I/O solution, we can study in more depth the key factors that might affect the performance of the solution when applied on different parallel file systems as shown by some preliminary tests performed on the Lustre parallel file system. These tests showed very different results compared to the ones obtained on the BeeGFS file systems used in this study. Furthermore, we can also explore the cases in which the benefits of using the different data transfer primitives might differ.

Within the multi-threaded individual I/O solution, the key goal was to take steps towards a file access model that supports the MPI + threads programming model. As an important enhancement to this work, we can incorporate access to an existing thread pool, instead of creating threads for each file separately. This requires a deeper interaction between MPI and multi-threaded programming model such as OpenMP, a topic that is also actively discussed in the MPI Forum. Furthermore, there are additional algorithmic optimizations that can be explored, such as incorporating the newly added sequential pre-calculation step into the multi-threaded implementation and dynamically adjusting the number of threads being used.

In the last topic, the new support of the Lustre Lockahead feature has unlocked the gate for further optimizations to the MPI I/O functions, in particular collective I/O operations. First, the preliminary evaluation has shown that the default setting of the number of aggregators on Lustre is no longer optimal as a single I/O server can support higher number of concurrent clients with Lustre Lockahead. Furthermore, we can explore different application scenarios that might involve

irregular data layouts or very small file accesses.

Last but not least, each aspect presented in this dissertation has enhanced the overall parallel I/O performance under specific circumstances and configurations. The ultimate goal would be a comprehensive parallel I/O solution that embraces all instances of I/O optimizations. Yet, this work remains an important step toward this goal.

Bibliography

[1] Ali, N., Carns, P., Iskra, K., Kimpe, D., Lang, S., Latham, R., Ross, R., Ward, L., and Sadayappan, P. Scalable I/O forwarding framework for high-performance computing systems. In *2009 IEEE International Conference on Cluster Computing and Workshops* (2009), IEEE, pp. 1–10.

[2] Blas, J. G., Isaila, F., Singh, D. E., and Carretero, J. View-based collective I/O for MPI-IO. In *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)* (2008), IEEE, pp. 409–416.

[3] Board, I. S. *IEEE standard for information technology: POSIX Ada language interfaces — Part 1: Binding for system application program interface (API).* IEEE Computer Society Press, June 1992. Std 1003.5-1992.

[4] Bonnie, M. M. D., Ligon, B., Marshall, M., Ligon, W., Mills, N., Sampson, E. Q. S., Yang, S., and Wilson, B. Orangefs: Advancing pvfs. *FAST poster session* (2011).

[5] Brown, S. A., Folk, M., Goucher, G., and Rew, R. Software for portable scientific data management. *Computers in Physics 7*, 3 (May/June 1993), 304–308.

[6] Buttlar, D., Nichols, B., and Farrell, J. P. *Pthreads Programming.* O'Reilly & Associates, Inc., 1996.

[7] Chaarawi, M., and Gabriel, E. Automatically selecting the number of aggregators for collective I/O operations. In *Workshop on Interfaces and Abstractions for Scientific Data Storage, IEEE Cluster 2011 conference* (Austin, Texas, USA, 2011), p. t.b.d.

[8] Chaarawi, M., Gabriel, E., Keller, R., Graham, R. L., Bosilca, G., and Dongarra, J. J. OMPIO: A modular software architecture for MPI I/O. In *in Y. Cotronis, A. Danalis, D. S. Nikolopoulus, J. Dongarra (Eds.) 'Recent Advances in the Message Passing Interface', Lecture Notes in Computer Science, vol. 6960* (2011), Springer, pp. 81–90.

[9] Coloma, K., Ching, A., Choudhary, A., Liao, W.-k., Ross, R., Thakur, R., and Ward, L. A new flexible MPI collective I/O implementation. In *2006 IEEE International Conference on Cluster Computing* (2006), IEEE, pp. 1–10.

[10] Corbetty, P., Feitelsony, D., Fineberg, S., Hsuy, Y., Nitzberg, B., Prosty, J., Sniry, M., Traversat, B., and Wong, P. Overview of the MPI-IO parallel I/O interface. *Input/output in Parallel and Distributed Computer Systems 362* (1996), 127–146.

[11] Del Rosario, J. M., Bordawekar, R., and Choudhary, A. Improved parallel I/O via a two-phase run-time access strategy. *ACM SIGARCH Computer Architecture News 21*, 5 (1993), 31–38.

[12] Dickens, P. M., and Thakur, R. Improving collective I/O performance using threads. In *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999* (1999), IEEE, pp. 38–45.

[13] Farrell, P. Shared file performance improvements: Ldlm lock ahead. https://wiki.lustre.org/images/f/f9/Shared-File-Performance-in-Lustre_Farrell.pdf, 2016. Cray Inc. Accessed: 2020-07-20.

[14] Farrell, P. Ladvise lock ahead design. https://github.com/lustre/lustre-release/blob/master/Documentation/ladvise_lockahead.txt, 2017. Lustre-release project documentation. Accessed: 2021-10-22.

[15] Feki, R., and Gabriel, E. On overlapping communication and file I/O in collective write operation. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2020), IEEE, pp. 1–8.

[16] Feki, R., and Gabriel, E. Design and evaluation of multi-threaded optimizations for individual MPI I/O operations. In *2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)* (2022), IEEE, pp. 122–126.

[17] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting* (2004), Springer, pp. 97–104.

[18] Geist, A., Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Saphir, W., Skjellum, T., and Snir, M. MPI-2: Extending the message-passing interface. In *European Conference on Parallel Processing* (1996), Springer, pp. 128–135.

[19] Gropp, W., Lusk, E., Doss, N., and Skjellum, A. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing 22*, 6 (1996), 789–828.

[20] Group, H. D. F. *HDF5 Reference Manual*, September 2004. Release 1.6.3, National Center for Supercomputing Application (NCSA), University of Illinois at Urbana-Champaing.

[21] Hady, F. Faster access to more data: A deep exploration of drastically reduced latency to data at the memory, device, and computing system level. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-technology/faster-access-to-more-data-article-brief.html. Intel inc. Accessed: 2020-10-31.

[22] Hady, F. Restoring the balance between bandwidth and latency. https://www.intel.com/content/dam/www/public/us/en/documents/guides/balancing-bandwidth-and-latency-article-brief.pdf. Intel inc. Accessed: 2020-10-31.

[23] Herold, F., Breuner, S., and Heichler, J. An introduction to BeeGFS. https://www.beegfs.io/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf, 2014. ThinkParQ, Tech. Rep., Jun. 2018. Accessed: 2019-11-2.

[24] Hoefler, T., and Gottlieb, S. Parallel zero-copy algorithms for fast Fourier transform and conjugate gradient using MPI datatypes. In *European MPI Users' Group Meeting* (2010), Springer, pp. 132–141.

[25] Hoefler, T., and Lumsdaine, A. Optimizing non-blocking collective operations for infiniband. In *Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium (IPDPS)* (04 2008).

[26] Howard, J. H., et al. *An Overview of the Andrew File System*, vol. 17. Carnegie Mellon University, Information Technology Center, 1988.

[27] Intel. Intel MPI implementation. https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html, 2005. Accessed: 2020-02-05.

[28] Koziol, Q., and Matzke, R. Hdf5–a new generation of hdf: Reference manual and user guide. *National Center for Supercomputing Applications, Champaign, Illinois, USA, http://hdf. ncsa. uiuc. edu/nra/HDF5* (1998).

[29] Levy, E., and Silberschatz, A. Distributed file systems: Concepts and examples. *ACM Computing Surveys (CSUR) 22*, 4 (1990), 321–374.

[30] Li, J., Liao, W.-k., Choudhary, A., Ross, R., Thakur, R., Gropp, W., Latham, R., Siegel, A., Gallagher, B., and Zingale, M. Parallel netcdf: A high-performance scientific I/O interface. In *SC'03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing* (2003), IEEE, pp. 39–39.

[31] Liao, W.-k., and Choudhary, A. Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols. In *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (2008), IEEE, pp. 1–12.

[32] Liu, W., Zhou, J., and Guo, M. Topology-aware strategy for MPI-IO operations in clusters. *Journal of Optimization 2018* (2018).

[33] Lofstead, J., Zheng, F., Klasky, S., and Schwan, K. Adaptable, metadata rich IO methods for portable high performance IO. In *In Proceedings of IPDPS'09, May 25-29, Rome, Italy* (2009).

[34] Lofstead, J. F., Klasky, S., Schwan, K., Podhorszki, N., and Jin, C. 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* (2008), pp. 15–24.

[35] Lu, Q., Wu, J., Panda, D., and Sadayappan, P. Applying MPI derived datatypes to the NAS benchmarks: A case study. In *Workshops on Mobile and Wireless Networking/High Performance Scientific, Engineering Computing/Network Design and Architecture/Optical Networks Control and Management/Ad Hoc and Sensor Networks/Compil* (2004), IEEE, pp. 538–545.

[36] Lustre developers. *Lustre* Software Release 2.x: Operations Manual.* Oracle/Intel, "https://doc.lustre.org/lustre_manual.pdf".

[37] Ma, X., Winslett, M., Lee, J., and Yu, S. Faster collective output through active buffering. In *Proceedings 16th International Parallel and Distributed Processing Symposium* (2001), IEEE, pp. 8–pp.

[38] Ma, X., Winslett, M., Lee, J., and Yu, S. Improving MPI-IO output performance with active buffering plus threads. In *Proceedings International Parallel and Distributed Processing Symposium* (2003), IEEE, pp. 10–pp.

[39] May, J. M. *Parallel I/O for High Performance Computing*. Morgan Kaufmann, 2001.

[40] Mehta, K., and Gabriel, E. Multi-threaded parallel I/O for OpenMP applications. *Int. J. Parallel Program. 43*, 2 (Apr. 2015), 286–309.

[41] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard Version 3.1*, June 2015. http://www.mpi-forum.org.

[42] Moore, M., Farrell, P., and Cernohous, B. Lustre Lockahead: Early experience and performance using optimized locking. *Concurrency and Computation: Practice and Experience 30*, 1 (2018), e4332.

[43] Nitzberg, B., and Lo, V. Collective buffering: Improving parallel I/O performance. In *Proceedings. The Sixth IEEE International Symposium on High Performance Distributed Computing (Cat. No. 97TB100183)* (1997), IEEE, pp. 148–157.

[44] Ohta, K., Kimpe, D., Cope, J., Iskra, K., Ross, R., and Ishikawa, Y. Optimization techniques at the I/O forwarding layer. In *2010 IEEE International Conference on Cluster Computing* (2010), IEEE, pp. 312–321.

[45] Open MPI: Open Source High Performance Computing. *http://www.open-mpi.org/*.

[46] OpenMP Application Review Board. *OpenMP Application Program Interface, Draft 3.0*, October 2007.

[47] Ros, R. Parallel I/O benchmarking consortium. https://www.mcs.anl.gov/research/projects/pio-benchmark/. Accessed: 2018-09-30.

[48] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. Design and implementation of the sun network filesystem. In *Proceedings of the Summer USENIX conference* (1985), pp. 119–130.

[49] Schmuck, F., and Haskin, R. {GPFS}: A {Shared-Disk} file system for large computing clusters. In *Conference on File and Storage Technologies (FAST 02)* (2002).

[50] Schneider, T., Gerstenberger, R., and Hoefler, T. Micro-applications for communication data access patterns and MPI datatypes. In *European MPI Users' Group Meeting* (2012), Springer, pp. 121–131.

[51] Sehrish, S., Son, S. W., Liao, W.-k., Choudhary, A., and Schuchardt, K. Improving collective I/O performance by pipelining request aggregation and file access. In *Proceedings of the 20th European MPI Users' Group Meeting* (2013), ACM, pp. 37–42.

[52] Shan, H., and Shalf, J. Using IOR to analyze the I/O performance for HPC platforms. Tech. rep., Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US), 2007.

[53] Smith, J. E., Hsu, W.-C., and Hsiung, C. Future general purpose supercomputer architectures. In *Supercomputing'90: Proceedings of the 1990 ACM/IEEE Conference on Supercomputing* (1990), IEEE, pp. 796–804.

[54] Thakur, R., Gropp, W., and Lusk, E. A case for using MPI's derived datatypes to improve I/O performance. In *SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing* (1998), IEEE, pp. 1–1.

[55] Thakur, R., Gropp, W., and Lusk, E. On implementing MPI-IO portably and with high performance. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems* (1999), pp. 23–32.

[56] Tsujita, Y., Hori, A., and Ishikawa, Y. Affinity-aware optimization of multithreaded two-phase I/O for high throughput collective I/O. In *2014 International Conference on High Performance Computing & Simulation (HPCS)* (2014), IEEE, pp. 210–217.

[57] Tsujita, Y., Hori, A., Kameyama, T., Uno, A., Shoji, F., and Ishikawa, Y. Improving collective MPI-IO using topology-aware stepwise data aggregation with I/O throttling. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region* (2018), pp. 12–23.

[58] Tsujita, Y., Yoshinaga, K., Hori, A., Sato, M., Namiki, M., and Ishikawa, Y. Multithreaded two-phase I/O: Improving collective MPI-IO performance on a Lustre file system. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (2014), IEEE, pp. 232–235.

[59] Vilayannur, M., Lang, S., Ross, R., Klundt, R., Ward, L., et al. Extending the POSIX I/O interface: a parallel file system perspective. Tech. rep., Argonne National Lab.(ANL), Argonne, IL (United States), 2008.

[60] Wang, F., Oral, S., Shipman, G., Drokin, O., Wang, T., and Huang, I. Understanding Lustre filesystem internals. *Oak Ridge National Laboratory, National Center for Computational Sciences, Tech. Rep* (2009).

[61] Wang, Y., Davis, K., Xu, Y., and Jiang, S. iharmonizer: Improving the disk efficiency of I/O-intensive multithreaded codes. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium* (2012), IEEE, pp. 921–932.

[62] Wang, Z., Shi, X., Jin, H., Wu, S., and Chen, Y. Iteration based collective I/O strategy for parallel I/O systems. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (2014), IEEE, pp. 287–294.

[63] Welch, B. Posix IO extensions for HPC. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)* (2005).

[64] Zhang, X., Jiang, S., and Davis, K. Making resonance a common case: A high-performance implementation of collective I/O on parallel file systems. In *2009 IEEE International Symposium on Parallel & Distributed Processing* (2009), IEEE, pp. 1–12.

[65] Zingale, M. Flash I/O benchmark routine parallel HDF5. https://www.ucolick.org/~zingale/flash_benchmark_io/. Accessed: 2018-10-2.