ON SCALABLE COLLECTIVE I/O FOR HIGH PERFORMANCE COMPUTING

A Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Vishwanath Venkatesan

December 2013

ON SCALABLE COLLECTIVE I/O FOR HIGH PERFORMANCE COMPUTING

Vishwanath Venkatesan

APPROVED:

Edgar Gabriel, Chairman
Dept. of Computer Science

Barbara Chapman
Dept. of Computer Science

Jaspal Subhlok
Dept. of Computer Science

Shishir Shah
Dept. of Computer Science

Emmanouil Doxastakis
Dept. of Chemical and Biomolecular Engineering

Dean, College of Natural Sciences and Mathematics

Acknowledgments

First and foremost I would like to thank my advisor Dr. Edgar Gabriel for providing me with the opportunity and guidance to achieve this goal. Without his constant encouragement and support, completing this dissertation would have not been possible. I owe a major part of my success to him.

My committee members Jaspal Subhlok, Shishir Shah, Barbara Chapman and Manolis Doxastakis; I can best describe them as very friendly and easily approachable. Not only did they serve on my thesis committee but were also available whenever I needed their help.

I would like to thank Quincey Koziol, Mohamad Chaarawi from HDF5 and Bryon Nietzel from Intel for having given me an opportunity to contribute on the Exascale Fast Forward Project, which is now a part of this thesis. Thanks to Jerome, Peter, Johnathan and Frank for the interesting ping-pong sessions which was definitely a highlight of my summer at HDF Group.

I would also like to thank all my friends from Intel, for providing an excellent summer of 2012. Special thanks to Nitin, Nagesh, and Ravindra for giving me an opportunity to work in their exciting team which provided valuable experience.

I thank Sunil Thulasidasan from LANL, for not only providing an opportunity to work in his group but also providing letters of recommendations when needed. Special thanks to Lukas Kroc, Shiva Kashiviswanath and Christian Sommer for their words of wisdom, which really inspired me to take up PhD again.

It would not be out of place to thank my friends from Columbia, Nalini, Ravindra,

Karthik TJ, David, Noah, Jayanth, Baradhwaj, Chinmay, Rohit, Abhinav, Sambudhho, Sarfaraz, Raghu, Aaron, Ruchika, Sumit Sharma, Nipun, Tarun, Neethi, Ashwath, Srikanth for making my tough times at Columbia feel lighter.

My friends from Pratham made my time at UH fun and exciting. Special mention to Pranav, Charu, Kinjal, Darel, Abhi, Nirja, Jinal, Simer, Arshad, Radhika, Chintan, Joseph and Deepa. It was a lot of fun working with you all.

My cricket group at UH for the midnight cricket sessions which were a real stress buster. Thanks to Pankaj, Akshay, Satish, Soham, Nishant, Lakhan and others.

My lab mates at Parallel Software Technologies Lab, who made my everyday fun-filled and eventful. Special thanks to Kshitij Mehta, for listening to my talk and providing invaluable advise. Thanks to Peggy Lindner, for the words of encouragement before the talk. All my lab mates Shailesh, Shwetha, Youcef, Hadi, Mohamad, Saber, Sarat, Jyoti. Thank you for all your support.

I would also like to thank Prof. N. Venkateswaran for igniting the idea of doing a PhD and my friends from WARFT, Karthik Ganesan, Haswath Narayanan and Viswanath Krishnamurthy for all the fun-filled conversations.

My family who have always been with me in my lows and highs. Savithri Venkatesan, P. K. Venkatesan, Badrinath Venkatesan and Viswadhara Meenakshi, thank you for being there for me. This would not be possible without you guys!

I would like to dedicate this dissertation to my Mom and Dad: Mrs. and Mr. Venkatesan.

Humility, unostentatiousness, harmlessness, forbearance, uprightness, service to the guru, purity, steadiness and self-control - all this is called knowledge.

- Bhagavan Sri Krishna, Bhagavat Gita - Jnana Yoga

ON SCALABLE COLLECTIVE I/O FOR HIGH PERFORMANCE COMPUTING

An Abstract of a Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Vishwanath Venkatesan

December 2013

Abstract

The increasing number of cores per node has propelled the performance of leadership-scale systems from teraflops to petaflops. On the other hand, bandwidth of I/O subsystems have almost been stagnant. This has created a huge gap between the computation and I/O time, making I/O a major bottleneck. Furthermore, the realized I/O bandwidth in such systems is in general far lower compared to the theoretical peak bandwidth. The Message Passing Interface (MPI) has been the *de facto* standard for parallel computing in the past couple of decades. MPI-I/O, which is a part of the MPI specification, not only offers a clean approach to access the file system from the application but also acts as a middle-ware between the application and the file system to specify a variety of enhancements. Specifically, collective I/O has proven to be very effective for I/O in large scale systems and helps to bridge the gap between the theoretical and sustained I/O bandwidth. This dissertation aims at developing approaches to improve parallel I/O at this level. In particular, this dissertation provides methods to utilize data-layout-aware rank assignment to improve I/O performance, overlap collective I/O with computation and finally use the principles of collective I/O on staging based I/O architectures.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A supercomputer is a device for turning compute-bound problems into I/O-bound problems

– Ken Batcher

Emeritus Professor at Kent State University

IEEE Seymour Cray Computer Engineering Award Recipient

In the past two decades the main focus of scientific computing has been to improve the computational capability of HPC systems. This has lead to the development of faster processors, memory and increased disk capacity. But the rate at which the disk drives can read and write has not improved with the same pace. For example, the CPU speeds have been increasing 50-100% every year in comparison to disk access time which has decreased to about one third in ten years [7] [8]. This has created a huge gap between the CPU performance and the I/O access time and is in general

termed as the I/O bottleneck. Also, new application areas such as visualization, image processing and grand challenge problems, are creating ever-increasing demands for I/O. Some examples of such applications are, high performance aircraft simulation, computational fluid and combustion dynamics, simulation of human proteins and their folding and air quality modeling [9]. Such applications will have significant slow-downs despite the use of cutting-edge processors due to the limitations of the I/O subsystem.

Furthermore, with petascale systems in existence and exascale systems to come in future, complexity of systems are bound to increase dramatically thereby decreasing the average time to failure, for example, a hundred-thousand-nodes Blue Gene (BGŁ) supercomputer has a failure once in every 8 hrs [10]. This necessitates frequent application check-pointing. Moreover, with increasing number of nodes the amount of data that needs to be written for every checkpoint can also increase significantly. This makes I/O a major concern. In fact, authors in [11] argue that the performance of supercomputers should ultimately be measured by how fast they can move data both within the system and across a network, rather than using floating point computation rate. Since Amdahl's law stipulates that scalability of a parallel application is limited by its least scalable section, its fair to say that improving I/O can no longer be treated as an afterthought [12].

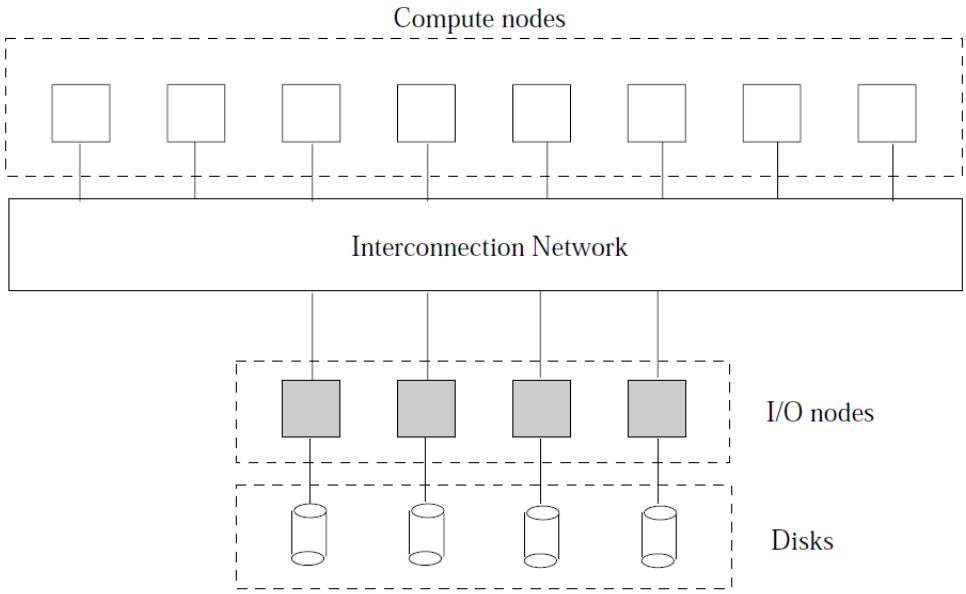

Figure 1.1: Parallel I/O architecture for a distributed memory system

1.1 Parallel I/O Infrastructure

Parallel and distributed architectures with the help of some hardware and software enable servicing of concurrent I/O requests to the file system. In particular, parallel I/O on large scale systems largely depends on the utilization of hundreds of I/O servers with tens of thousands of magnetic hard disks [13]. These I/O nodes serve as servers for the *parallel file system* also holding the meta-data (bookkeeping for each file) information. A parallel file system typically provides a global namespace and stripes files across multiple I/O nodes, disks or RAIDs. A file is generally split into multiple smaller units called striping units, where the striping units are distributed across disks in a round-robin fashion. Striping provides higher bandwidth and allows multiple processes to access distinct portions of a file concurrently. Some of the

popular file systems are PVFS [14], Lustre [15], and GPFS [16]. Many leadership-scale installations use similar settings for their parallel file systems. For example, Japanese supercomputer, K-computer has a Lustre based file system with 864 Object Storage Targets [17], the Jaguar system from Oak Ridge National Labs also uses Lustre filesystem with 672 OSTs [18], and the Chinese supercomputer Tianhae-2 [19, 20] has upto 128 Object Storage Servers (OSS 's). In general, it is advised to have I/O servers dedicated for I/O but it is not mandatory. Figure 1.1 [21] shows an example of a typical parallel I/O architecture in a distributed memory system.

1.2 Parallel I/O and Challenges

Utilization of a parallel file system from the application can be done using different approaches. We will discuss each of them and look at the advantages and disadvantage of using each approach in a parallel system in this section.

1.2.1 Sequential I/O

Sequential I/O refers to the uniprocess file I/O operations in applications. Applications written in C or Fortran have traditionally used sequential I/O for file operations, for example fwrite, fread, pwritev, preadv. POSIX (Portable Operating System Interface) [22] provides a standard API for serial file access by a single process. There is no explicit support for specifying parallelism using POSIX I/O API. The primary advantage of using this approach is that it is relatively simple to use and a lot of legacy codes could be used directly. But, if only one process performs I/O in a parallel system it will lead to serious load imbalance issues and the process involved in

the file operations can become a huge performance bottleneck. This is a fundamental limitation and has triggered a lot of research activity to overcome limitations of POSIX I/O with parallel systems [23].

1.2.2 Parallel I/O

Parallel I/O as discussed earlier can be performed in two different scenarios:

- **Individual File Parallel I/O :** One approach to have parallel I/O using POSIX I/O is by making each individual process access its own file. The primary advantage of this approach is that it is simple, easy and will possibly result in good bandwidth. One problem with this approach it requires complex and expensive pre-processing and post-processing steps for splitting and merging the files. If we do not use any such processing, then this approach
  - loses fault tolerance (in case of node failure we need the same amount of processes are needed to restart the application)
  - will not be able support varying the number of processes between runs, for example if the result of an `n` process run is the input for an `m` process run, where m $\neq$ n.

  Moreover, having one file for each process is not really a scalable approach as it can overwhelm the metadata server, which contains information about data on I/O nodes, very soon.
- **Shared File Parallel I/O :** In this approach, multiple processes can access the same file simultaneously and efficiently. Unlike individual I/O which is sequential I/O performed independently by a number of processes, this represents true

parallel I/O. In other words, the I/O should be parallel from the applications perspective [21]. This approach overcomes almost all the problems associated with individual file based parallel I/O approach. Since this approach necessitates coordination between processes accessing the same file to ensure consistent result, there are various other factors that come into picture which make it hard to extract the same I/O bandwidth as the individual I/O method. We will look at the challenges posed by this approach in detail in the next section. In the rest of the document all references to shared file parallel I/O are simply referred to as parallel I/O.

1.2.3 Challenges

Shared file parallel I/O, as discussed in the previous section, is the ideal approach to perform I/O in a parallel system. But making shared file parallel I/O performant can be challenging and we will discuss some of those challenges in this section.

- **Application Programming Interface (API):**

  Most parallel file systems have evolved out of uniprocessor file systems [21] and retain the same API, namely the POSIX I/O API. As discussed earlier, the POSIX I/O API is not an appropriate API for parallel I/O for two main reasons.

  1. *No support for non-contiguous file accesses.*

     Typically in a parallel system, applications have a tendency to access large amounts of small, non-contiguous chunks of data. Most parallel file systems provide higher performance with contiguous access patterns in comparison to non contiguous access patterns [2] [24] [25]. But it is not uncommon to

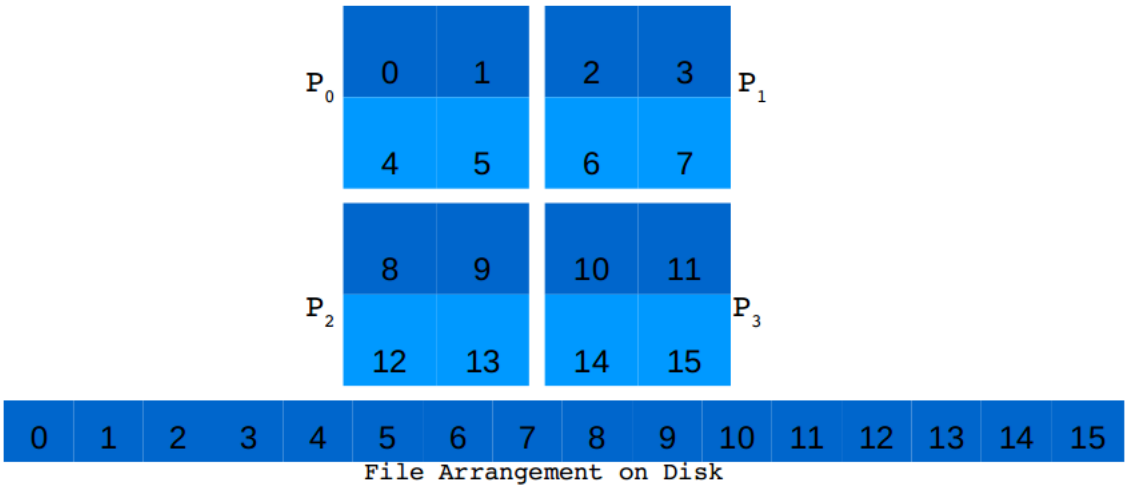

Figure 1.2: Non-contiguous access and its mapping to disk

have such access patterns. For example, consider a two-dimensional matrix stored in a row-major storage format as shown in fig 1.2 [23]. Assuming a two-dimensional data decomposition, each process will have to read/write a 2D sub-block of the matrix. An interface like that provided by POSIX, there is no way to specify the global access pattern to the I/O system. In such a case each process must seek to a particular location in a file, read or write a small contiguous piece, then seek to the next location and repeat the same steps. This will result in numerous number of small I/O requests which can potentially result in poor performance. This necessitates that the API have the potential to specify access pattern of all the processes.

2. *No support for collective I/O*

   When there are non-contiguous requests from multiple processes as in figure 1.2, it can result in a large number of small requests arriving in any

order from multiple processes resulting in poor performance. This is not only because the I/O latency for these accesses is high but also due to the poor utilization of the file system cache. So it is necessary for the processes to work in collaboration. If an I/O API allows the user to specify the entire access information of each process as well as the fact that the processes need access to the file simultaneously, the implementation can read the entire file and the send the right pieces of data to the right processes [21]. This type of optimization is called as collective I/O. In case of write operations, the contiguous offsets could be grouped by data exchange through communication and the write operation could be carried out to the according offsets. Such optimizations can provide significant performance improvements.

3. *Portability*

   Some file system vendors support variations of the POSIX API, and, therefore programs written in these API are not portable. Having a API that has the potential to abstract this information can increase portability of applications

Thus, API plays a critical role in performing I/O efficiently.

- **Network:**

  Network plays an important role in deciding the performance of the I/O operations. A typical parallel system can contain different types of network used for different purposes, namely, the message passing network between the compute nodes and the I/O network which connects to the I/O nodes. Some popular

networks available are InfiniBand [26], Myrinet [27], Quadrics [28], Dolphin [29]. Both, message passing and I/O networks have a significant impact on the performance of I/O operations. With multi-core processors gaining popularity, the intra-node network also starts to play a major impact. When a collective I/O operation is performed it is very important that the data-exchange is performed on the network which has the highest bandwidth and the lowest latency. Such an optimization can in turn result in significant improvements to the I/O performance.

- **Interference:**

  Accessing shared files in parallel I/O also poses other forms of challenges. When many processes attempt to write to a same file simultaneously they tend to fill up the write caches quickly thereby stalling the application. This is called internal interference [18]. Collective I/O helps us to overcome such interferences by providing the concept of aggregators (representative processes accessing a shared file) which reduces contention at the file system.

- **Scalability & Irregular data:**

  With increasing size and complexity of high-performance systems, periodic checkpointing of data to handle failures has gained importance. Most of such checkpointing data is dynamically created and irregularly partitioned. To handle such scenarios, independent I/O becomes crucial. In addition, the strict data consistency semantics adopted by filesystems constricts the scalability of I/O subsystems. In such scenarios it is seen that using a forwarding layer to aggregate and delegate I/O requests to storage systems has proven to be beneficial [30–33]

This dissertation will focus on addressing some of these challenges through collective I/O.

1.3 Parallel I/O in Parallel Programming Models

In the past couple of decades there have been a number of parallel programming models introduced. Parallel programming models/languages could be broadly classified into three different categories [3] namely, message passing models, shared memory models and partitioned global address space languages. We will look at some of them in detail.

Shared Memory Models

In this section we discuss some of the popularly used shared memory programming models and their support for I/O.

- OpenMP (Open multi-processing) [34] [35] is an Application Programming Interface (API) that has been developed to enable portable and scalable shared memory application. It supports parallelization with a specified set of compiler directives, environment variables, and library routines. OpenMP can be used for parallelizing applications for platforms ranging from desktops to supercomputers. OpenMP based on [36] provides I/O support for parallel applications in shared memory platforms. This library provides interfaces which have the capability to perform collective I/O with multiple threads. It does not provide any asynchronous interfaces. Since this is based on a shared memory model, there is a fundamental scalability limitation with this library as more number

of threads writing to the file system can saturate the network bandwidth. The OpenMP I/O is not a part of the OpenMP standard.

- POSIX threads (pthreads) [37] is a standard API for creating and manipulating with threads created in 1995. Although more complex in comparison to OpenMP, pthreads are still used with certain applications and are present in a lot of legacy applications. But pthreads does not provide any explicit support for parallel I/O.
- Threading Building Blocks (TBB) [38] is a library which supports parallel programming using standard C++ templates. Although a lot restricted compared to OpenMP, TBB is good for specifying parallelism with tasks instead of raw threads. Also TBB support complete compatibility with OpenMP. TBB does not provide any explicit support for parallel I/O, but some work on performing I/O using overlapping with parallel_for construct has been performed in [39].

Message Passing Models

Message Passing Interface (MPI) which is the most widely used message passing model and contains a specification for parallel-I/O in version 2.0. MPI-I/O will be discussed in detail in the upcoming sections. The other message passing models such as PVM [40] or P4 [41] do not provide any specification for parallel I/O.

Partition Global Address Space Models

PGAS (Partition Global Address Space) models provide a global view of the logical address space where a portion of it is local to each processor. It combines the programming convenience of shared memory with the locality and performance control of message passing [42]. Two popular languages in this model are:

- *Unified Parallel C* [43] is an extension to the C programming language for high performance computing on large scale machines. It supports both shared and distributed memory platforms. UPC provides an API for parallel I/O. UPC I/O [44] offers very similar functionalities like MPI I/O such as collective I/O, blocking, non-blocking I/O and shared file access. The main difference is that UPC I/O does not have any concept of file-view or derived data-types.
- *Co-array Fortran* [45] is an extension to Fortran 95. It uses Fortran like notation to express data decomposition as in message passing models. Co-Array Fortran I/O [46] is only a minor extension to Fortran 95 I/O. It provide flexibility to be implemented using either thread-based images or process-based images. It also gets rid of the portability problems associated with Fortran 95 I/O on multiple images. A detail list of the extensions has been shown in [46]. Coarray fortran at present does not provide support for parallel I/O, although Eachampati, et al., discuss different approaches [47] that could be used to provide parallel I/O in co-array fortran [48] as a part of the OpenUH complier [49]. In particular, the authors discuss about using global arrays as I/O buffers and asynchronously manage the data transfer between the buffer and the storage.

1.4 Preamble to MPI - I/O

Message Passing Interface (MPI) [50] is a *message-passing library interface specification*. It has almost been *de facto* standard for distributed and parallel programming in the last couple of decades. MPI provides a variety of standard interfaces for message passing which are efficient, flexible and portable. MPI itself is a specification [50] and not an implementation. There are multiple implementations available for MPI, some of the popular implementations are Open MPI [51] [52], MPICH [53] and Intel MPI [54]. MPI-I/O was introduced in MPI specification version 2 [55] to handle files in parallel and distributed systems which leverages concepts from MPI specification version 1 [56] [3]. It provides a rich set of parallel I/O interfaces that allows application developers to express complex I/O access patterns with a single I/O request. Also, MPI-I/O provides the implementation the potential to optimize accesses to the underlying file system. In this section we will look at some of the interesting features that MPI-I/O provides along with an introduction to the existing I/O implementations.

1.4.1 MPI File View

A file view maps the relationship between the regions of a file that a process will access to the way those regions are laid out in the disk [23]. In MPI-I/O each process has its own file view. A process can access only regions present in its file view. A simple example for a file view is shown in fig 1.3. In this example, each process has a non-overlapping repeating file view. File views of different processes can be overlapping but that is restricted to read accesses. Setting a file view in MPI is a

collective operation, which means that all processes that share accesses to a given file must participate in this operation. Having the file view set beforehand gives the MPI-I/O implementation valuable information about the offsets for subsequent accesses from processes which could be used to significantly improve the I/O performance. For example, by using pre-calculated offsets, the underlying implementation could potentially pre-fetch data items for read operations.

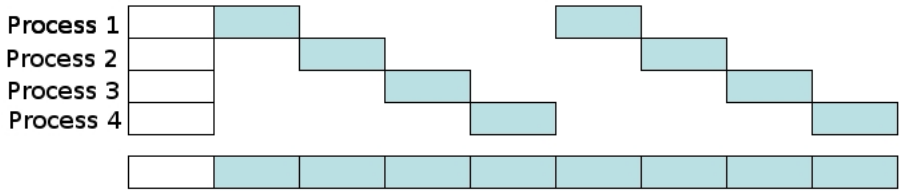

Figure 1.3: Example of 4 Processes defining a non-overlappable file view

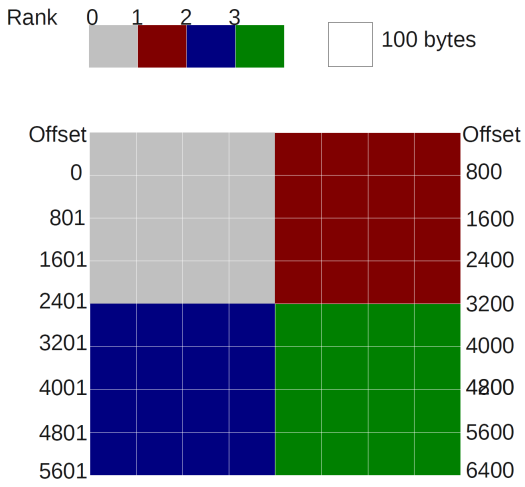

Figure 1.4: Example of 4 Processes accessing a 2D matrix

1.4.2 Collective I/O Operations

MPI I/O provides both individual and collective I/O operations. Individual I/O is similar to POSIX I/O in the sense that each process performs I/O for its own portion of the file view. Unlike individual I/O operations, collective I/O operations require the participation of all the processes in the operation. This provides a huge potential to improve I/O performance of an application, such as merging I/O requests from multiple processes and choosing the aggregators(writers/readers). Such optimizations can provide significant improvement in performance. For example, consider the following scenario shown in fig 1.4. In this scenario, four processes access the same input file which holds a two-dimensional matrix stored in a row major format. Assuming a 2D data decomposition, each process will work on a subset of the matrix. This means each process would require a large number of small I/O requests to be satisfied which in turn would lead to moving disk head back and forth to access data of all processes. This could be optimized with collective I/O where a large number of these small I/O requests could be merged to create few contiguous requests. This reduces disk latency and avoids unnecessary rewinds of the disk head. In this dissertation the focus would be mostly on optimizing collective I/O operations.

1.4.3 Asynchronous I/O operation

Asynchronous operations allows the user to overlap the computation with I/O such that I/O cost could be hidden behind computation. The application can introduce additional function calls to check or wait for completion of the I/O operations. This can be advantageous in comparison to the blocking version where the processes have

to wait until the I/O operation is complete which might leave system resources idle. MPI offers non-blocking variants for individual I/O operations but not for collective I/O operations. This dissertation will introduce a non-blocking variant for collective I/O and discuss in detail the challenges involved in designing them.

1.5 MPI I/O Implementations

1.5.1 ROMIO

The first and the most popularly used MPI implementation is ROMIO [2]. ROMIO has been implemented as a part of MPICH [53] and also is basis for many I/O libraries used in commercial MPI implementations. ROMIO's portability comes from the the layer called ADIO (Abstract Device Interface for Parallel I/O) [57] upon which ROMIO implements the MPI-IO interface. ADIO abstracts file-system features from the I/O API's and provides support for popular file systems like PVFS [14], Lustre [15], GPFS [16] and PanasasFS [58].

ROMIO offers two key optimizations

- **Data Sieving :**

  Data Sieving [2] [59] is a technique that ROMIO uses to handle non-contiguous requests from one process. It is known fact that it is critical to make as few requests as possible to reduce the effect of high I/O latency. To avoid this scenario, ROMIO tries to read single contiguous chunk of data from the first requested byte to the last requested byte. It then updates the user buffer with the according parts from the temporary buffer. The main disadvantage with this

approach is that if the size of the temporary buffer has to be matched with that of the user's request. But the extent of the user's request can be quite large. So to avoid this ROMIO specifies a maximum amount of contiguous data that could be read. The same technique could also be used in the case of writing data, but a read-modify-write must be performed to avoid overwriting the data already present in the holes between the contiguous data segments. Furthermore the portion of the file being modified must also be locked during the read-modify-write to prevent concurrent updates by other processes.

- **Collective I/O :**

  ROMIO offers a client-side collective I/O implementation based on the two-phase [60] [61] I/O strategy. This algorithm is used to handle non-contiguous requests from multiple processes. In the first phase the processes distribute the view information between each other based on which determine the extent of the users request and perform access with one large contiguous chunk. In the second phase this chunk is redistributed across the processes in their according offsets. A detailed discussion of two-phase I/O has been done in the upcoming chapters.

The selection criteria which ADIO module shall be used as of today is based on the file system only [3]. But, storage solutions demand that the I/O library allows for the easy deployment of a site specific configuration file, allowing system administrators to pre-tune modules and provide the optimal parameters for modules. This requires ability to switch between modules without having to recompile the entire library or

the application [3]. So a new flexible architecture called OMPIO [1] was developed. We will look at OMPIO in detail in the upcoming section.

1.5.2 OMPIO

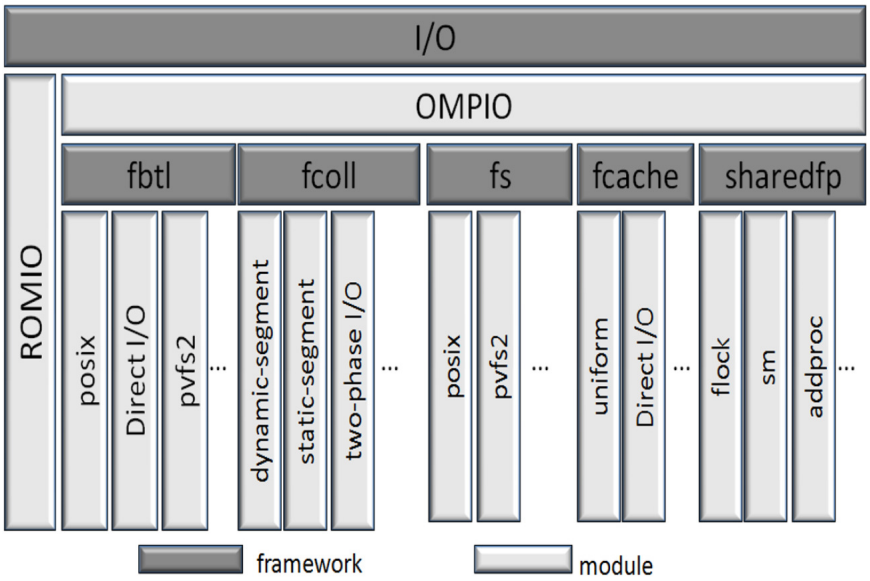

Figure 1.5: Overview of the OMPIO component and its frameworks [1]

The OMPIO [1] module is an implementation of the I/O framework of Open MPI [52]. This implementation coexists with ROMIO, which is the parallel I/O library used in all published versions of . There are several reasons that we have chosen to utilize OMPIO in this research, including:

- It increases the modularity of the parallel I/O library by separating functionality into distinct sub-frameworks.
- It allows frameworks to utilize different run-time decision algorithms to determine which module to use in a particular scenario, enabling non-file system

specific decisions

- It improves the integration of parallel I/O functions with other components of Open MPI, most notably the derived data type engine and the progress engine

The integration with the Open MPI progress engine allows for seamless progress of non-blocking I/O operations. The integration with the derived data type engine has multiple advantages, most notably faster decoding of derived data types and the usage of optimized data type to data type copy operations [1]. The architecture of OMPIO with its various components and frameworks has been shown in fig 1.5. The different frameworks in OMPIO are [3]

- *file system framework (fs):*

  Different file systems have different file manipulation functions that are specific and optimized for a corresponding file system. For example, though PVFS supports standard POSIX I/O operations, it has PVFS specific functions which are optimized for the file system. For example, `MPI_File_open` a collective call in PVFS is more optimal when one process resolves the filename and broadcasts it to the other processes. This framework abstracts out these file system specific manipulation functions like open, close and delete.

- *file byte transfer layer(fs):*

  The fbtl framework provides implementation for individual read and write operations. Even this framework allows to exploit filesystem specific functionality. It abstracts the byte-level read/write operations such that it could be adapted based on the underlying file-system.

- *collective I/O framework(fcoll):*

This framework provides interfaces for the collective file I/O operations. The interfaces of this framework provides a collection of different collective I/O read-/write algorithms. This framework allows the deployment of site-specific collective I/O algorithm, without having to modify any aspect of the collective file operation.

- *shared file pointer framework (sharedfp)*

  This framework provides the functionality required to manage the shared file pointer, allowing for generic and architecture specific optimizations.

1.5.3 The Adaptable IO System (ADIOS)

ADaptable IO System (ADIOS) [62], is an I/O API, that provides a simple POSIX IO like API and an external XML configuration file. Since I/O performance on HPC machines strongly rely on machine characteristics and configuration, it is important to tune I/O libraries and make good use of appropriate library APIs. Efficient code execution for scientific codes running on different platforms depend primarily on three factors namely [63],

- *Select the appropriate I/O method:*

  The external XML file allows the users to specify I/O description and configuration. The I/O description from the XML file helps in runtime selection of potentially different I/O methods suitable for the underlying hardware with minimal effort from the user.

- *Stored in desired file formats:*

A new file format BP is introduced by authors in [62] [63] designed for minimal required coordination, compact metadata storage, and resilience in cases of failures. It can also be converted at low cost to standard file formats like NetCDF [64] or HDF5 [65].

- *Identify data for analysis*

  With the file format BP are associated multiple efficient methods for data characterization. These compute attributes that can be use to identify data sets without having to analyze the entire data for large files and helps users to handle flood of data generated.

The authors in [63] demonstrate sufficient performance improvements with ADIOS and claim that the level of abstractions that ADIOS provides are quite important for extracting good performance from scientific codes running on different HPC machines.

1.5.4 Data Format Libraries

In addition to MPI I/O there are other higher level data format APIs available which are used widely in the scientific community. Libraries like HDF [65] and NetCDF [64] have been developed to provide higher level I/O support to applications. For example, they can directly read/write grids and meshes.

HDF5

Hierarchical Data Format (HDF5) [65] is a portable file format library developed by the HDF Group for storing, retrieving and analyzing data. HDF5 can handle

up to 32 dimensional data and saves the information along with the metadata in a file, which helps in the portability of data. The library also supports hierarchical file structures enabling the users to specify a variety of datatypes. As shown in fig 1.6 [66], dataset is mapped to file and its memory description and layout is stored in the metadata information. This helps the user to organize data in a variety of different ways including hierarchical formats wherein the actual mapping is taken care of by the library. HDF5 also supports parallel file access with the help of MPI I/O underneath the hood [67].

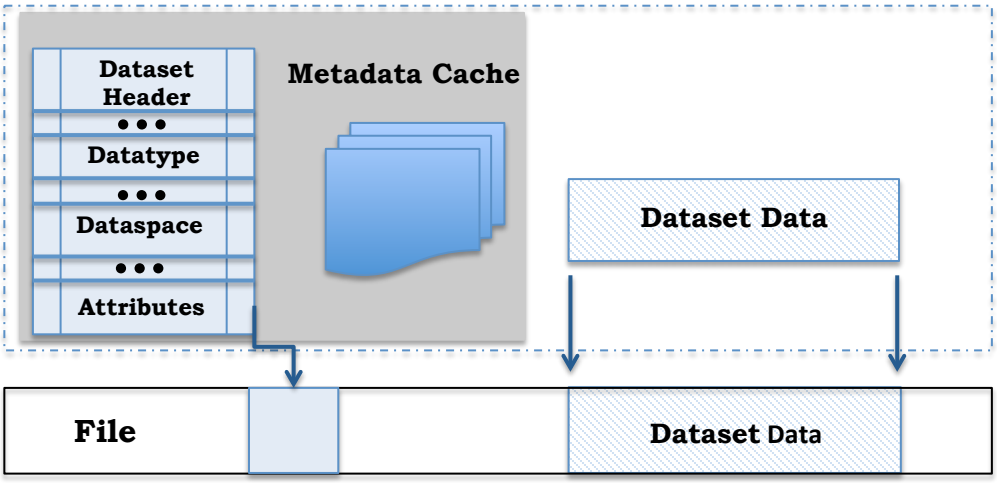

Figure 1.6: Illustration of data organization in HDF5 to improve flexibility

NetCDF

The Network Common Data Format (NetCDF) [64] was initially developed to support the requirements for Unidata applications (atmospheric and meteriological applications). This format similar to HDF supports *Self-describing* and *portable* way for

representing scientific data. NetCDF uses direct access to data rather than sequential access. This can prove to be efficient when the order in which data is accessed is different (i.e., reading and writing data in different order). NetCDF-4 still uses HDF file-format to support multiple dimensionality, although the overhead of this is comparatively very high. Like HDF5, NetCDF also provides support for parallel I/O with MPI I/O [68].

Both these libraries try to maintain the same format as that of the serial access and still provide parallel I/O. But these libraries unlike ADIOS are not yet self-optimizing, in the sense, they do not have the ability to chose the best method based on the underlying file-system and hardware characteristics.

1.6 Scalability Challenges

One of the main challenges facing I/O researchers today is that the improvements/optimizations developed should not only provide maximum I/O throughput but also show scalability with larger number of processes. A typical I/O stack of current HPC systems look like as shown in figure 1.7. It consists of serial and parallel high-level I/O libraries (eg: HDF5, Parallel HDF5, NetCDF, Parallel NetCDF); MPI I/O (eg: ROMIO or OMPIO) and POSIX implementations in the next level; parallel file systems which finally write the actual data to the storage infrastructure [31]. To satisfy the scalability challenges imposed by the increasing number of compute nodes, we need to choose the right layer in the stack to improve.

- File systems, can be a good target for improvements, but most file systems on

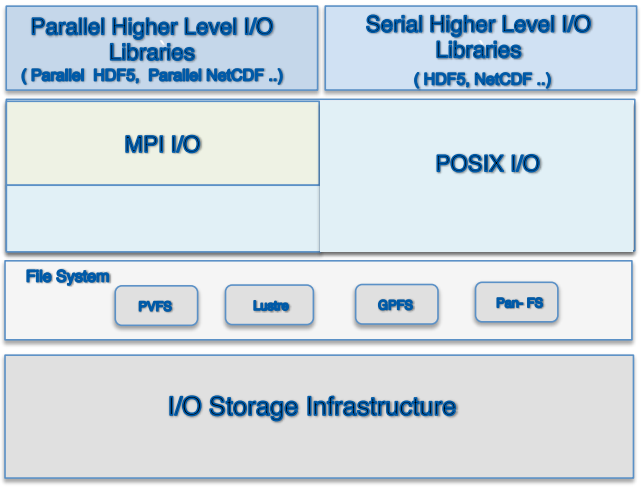

Figure 1.7: Today's typical IO Software Stack of a HPC system

leadership class machines like PanasasFS [58], Lustre [15], PVFS [14], GPFS [16] were not designed envisioning scalability challenges in the order of hundreds of thousands of processes[31, 69, 70]. In addition, different HPC machines use different file systems and optimizing one file system would not completely address the problem.

- In the layers above file systems, there are programming models such as MPI-I/O, POSIX I/O or parallel HDF and NetCDF interfaces. Making improvements to any one interface, might not address all the applications.
- In addition, some HPC systems use stripped down operating systems (minimalistic) to reduce system noise which can have adverse impact on the performance of applications [71, 72]. General features of operating systems such as multitasking, process preemption and context switching, ensure faster response time for applications, but they also introduce significant noises in the form of context switches, cache poisoning and interrupts [31]. For a system to be scalable its

important to have minimal OS noise. There has been a number of efforts to develop and deploy lightweight Linux kernels [73–75]. Also file system components contribute to this noise. File system components can initiate asynchronous communication to handle dirty data or cache locks which can hamper sychronicity of applications. Parallel machines like Blue Gene/P use a stripped down version of the OS kernel without multiprocessing and POSIX I/O system calls on compute nodes to eliminate operating system noise [76]

Recent research suggests that having a staging area or an I/O forwarding layer can help mitigate the above discussed challenges imposed by scalability requirements. To incorporate this, the software stack architecture from figure 1.7 changes to the stack as shown in figure 1.8. Having a forwarding layer or a staging area, aids the application in using asynchronous approaches which in-turn facilitate to reduce I/O overheads on applications' total processing time.

Although this is a fairly recent development, there have been many approaches and frameworks designed in recent times to incorporate forwarding in I/O subsystems. Abbasi, et al., discuss about a project called DataStager [77] which provides scalable data staging services with ADIOS [78] framework. ZOID [30] is another project from Iskra, et al., which also provides a architecture design with support for I/O forwarding. This project also provides a request scheduler where optimizations can be performed on requests from the same handle at the staging I/O nodes. There is also another work from Nisar, et al., which discusses about a delegation based I/O mechanism which was built on top of ROMIO [79] by assigning a set of compute

nodes as I/O staging/server nodes [80]. This work uses a plugin to intercept I/O calls and send them to the I/O nodes for request handling. One of the most recent projects was the Exascale Fast Forward I/O [5] project which is part of the Exascale Fast Forward initiative of the DOE. This project focuses on the development of the ideal stack for I/O at the Exascale level. Chapters 2, 5 will discuss about these projects in detail.

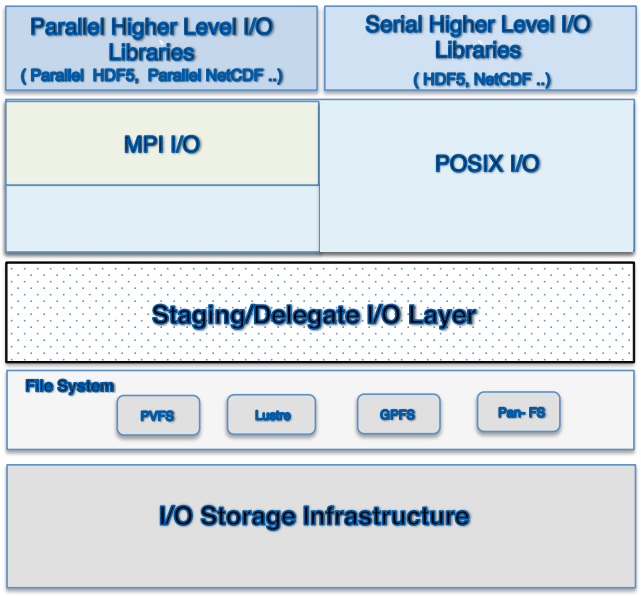

Figure 1.8: Recent Developments lead to the inclusion of an I/O forwarding layer to handle scalability challenges of I/O in HPC systems

1.7 Goals

From the discussions in the previous sections, its clear that to get performance improvements in the current petascale era and the future exascale era, there is a need to reduce the number of accesses to the storage structures and also try to hide the I/O

cost from applications as much as possible. To accomplish this both collective I/O and asynchronous I/O style optimizations are needed. This dissertation focuses on exploring optimizations using both collective I/O and asynchronous I/O to mitigate challenges of the current and future parallel applications.

In specific, this dissertation looks at achieving the following:

- Develop data-layout aware ranking strategy to optimize collective I/O operations
- Develop non-blocking collective I/O operations
- Develop a framework in a I/O staging node to use collective I/O principles for optimization

The first goal of this dissertation focuses on optimizing collective algorithm by using the data layout information in resource mapping. In this approach, the knowledge about the algorithms communication pattern for a given data layout is used to decide the resource mapping of MPI processes. A new module in the resource mapping engine of Open MPI is designed for this purpose, which will be able to use this information and the communication bandwidth(s) of the underlying network link(s) in the system to create an optimal mapping. The work will be evaluated with benchmarks such as Tile I/O [81] and BT I/O [82]

The second goal of this dissertation will look at developing a approach to overlap computation with I/O using non-blocking collective I/O operations. This work focuses on combining the benefits of collective I/O with asynchronous I/O. This is the first effort to create such operations and we are not aware of any other programming model offering such operations. Since it does not also exist in the MPI specification, this work will be done as a part of libNBC [4]. This work will be evaluated on

our PVFS [14] with a micro-benchmark and a real-world application to measure the potential for overlap.

The final part of the dissertation focuses on designing an optimization framework for a staging I/O node (or) I/O server capable of performing collective I/O style optimizations. This work will be done as a part of the Exascale Fast Forward project. The idea of the work will be to focus on reducing the number of I/O accesses to the underlying file system. Optimizations will be explored which are specific to I/O staging node style architectures. The work will be evaluated with synthetic benchmarks and applications scenarios.

The following chapters in the dissertation are divided as follows. Chapter 2 discusses about the relevant background and the related work done in the literature. Chapter 3 explains the design of the approach to process placement for collective I/O operations and evaluates its performance. Chapter 4 explains in detail about the design and evaluation of non-blocking collective I/O operations. Chapter 5 talks about the work done with optimizing I/O operations at I/O staging nodes using collective I/O principles as a part of the Exascale Fast Forward project. Finally Chapter 6 summarizes the work and presents the conclusion.

Chapter 2

Background and Related Work

Chapter 1 discussed about the importance of Collective I/O and Non-blocking I/O to improve performance of I/O operations in HPC applications. Collective I/O helps to reduce contention at the file systems, provides an abstraction to application users for a well-optimized machine specific implementation, and further increases overall I/O bandwidth obtained from the I/O subsystem. Non-blocking I/O, on the other hand, helps the application to hide the time consumed in I/O operations from the application. Both these approaches have been widely researched in the literature. This section will focus on discussing some of the background works done in the past, which are relevant and related to this dissertation.

2.1 Collective I/O

The concept of collective I/O was derived from the usage of collective communication operations, to facilitate group based I/O operations. Blocking collective I/O

operations currently defined in MPI, provide a higher level abstraction to the user. This provides important advantages such as programmability, safety (with regards to programming errors) and performance [4]. This also insulates the user from implementation details and provides the MPI developer with the freedom to provide optimizations.

2.1.1 Two Phase I/O

ROMIO [2] which is one of the widely used MPI I/O implementations, provides a client-side collective I/O implementation based on the two-phase algorithm. The algorithm assumes that file systems handle large contiguous requests much better than small non-contiguous ones. As the name suggests the algorithm is divided into two-phases [2], for example, in a collective read operation:

- In the first phase, processes are prompted to make a single, large, contiguous access assuming a distribution in memory.
- In the second phase, processes redistribute data among themselves to the desired data distribution in the file. This phase will add an overhead on communication between processes, but this overhead is small compared to the time saved in I/O [2].

Figure 2.1 shows a simple example of a two-phase I/O based read algorithm. Initially the entire information about the data distribution is communicated with all the processes. Then the entire access region is divided into non-overlapping, contiguous sub-regions denoted as file domains and each file domain is assigned to a unique

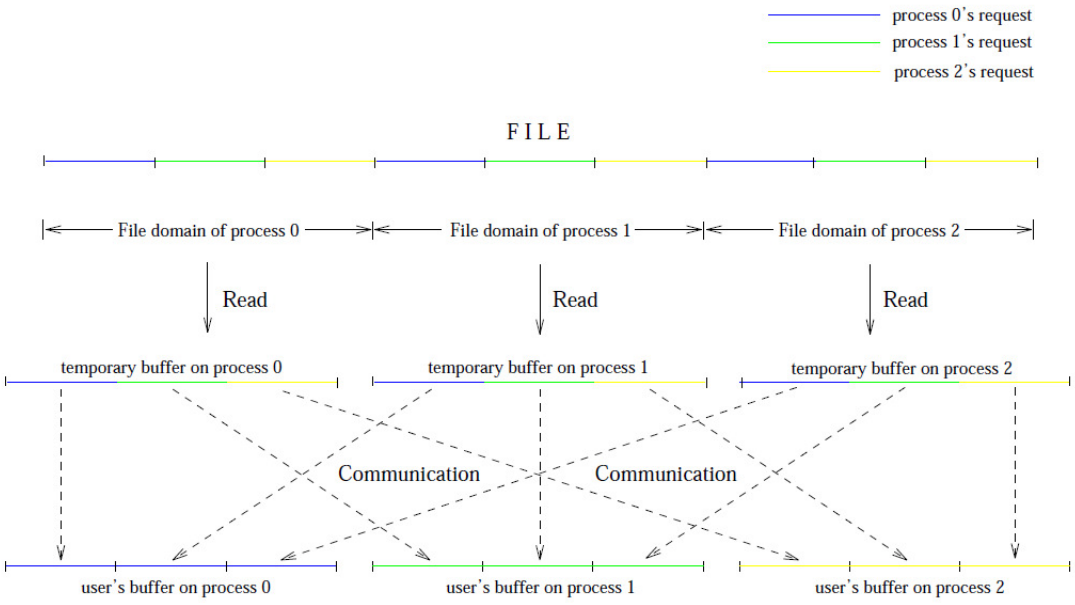

Figure 2.1: Simple Example of Two Phase I/O read figure source [2]

process. Each process creates a lists of offsets, lengths that they hold and that they require from other processes and broadcast the information to all the processes. One process makes the read/write operations on behalf of all processes for the requests located in its file domain. These representative processes are called aggregators. Each aggregator also does actual I/O operations in multiple cycles based on a value called the *cycle buffer size*. The parameters `number of aggregators` and `cycle buffer size` are provided to be configurable. Two-phase I/O was implemented as a part of ROMIO and is the default choice for collective I/O. Over the years there were many approaches that were developed to optimize collective I/O operations in ROMIO. We will look at a couple of them here.

2.1.2 Optimizations for Collective I/O Operations in ROMIO

- **View-based Collective I/O**

  The View-based collective I/O [83] provides a file system independent collective I/O implementation within ROMIO. This approach reduces the access overheads by using a "*declare once, use several times*" strategy. In other words the mapping is created between the view (contiguous memory region) and the file layout. This mapping can be used to boost performance, when the access pattern and the actual physical distribution matches. The results in [83] show that view based I/O can be useful in improving execution time of data-intensive applications.

- **File Domain Partitioning based on file system locking**

  The authors in [84] propose and discuss three new file domain partitioning techniques to support the two-phase I/O algorithm. Locking in parallel file systems has a greater impact on the I/O performance obtained. Different file systems have different locking protocols. GPFS [16] uses a distributed-token based locking whereas Lustre [15] uses a distributed-server based locking. In the former a token handler has the authority of authorizing locks for the neighboring byte ranges and in the latter I/O servers take care of locking for the stripes of files it stores. But both these file systems use an extent-based locking scheme. This means, the file system provides the lock to the entire file for the first process that accesses it (for example, a write call). When a second write arrives from another process, the first process will relinquish a part of the file for the requesting

process. The main advantage of this locking protocol is if a process has successive requests within a region where it has been already granted a lock, then there is no lock request needed. In their work, Liao and Choudhary [84] use this knowledge, and partition the file-domains generated in the two-phase algorithm such that domain sizes match the nearest lock boundaries. This minimizes the re-acquisition of locks due to ill-aligned partitioning of file-domains. They use the striping factor of the file system to determine the lock boundaries.

- **Resonance I/O**

  In the work done in Resonance I/O [85] the authors use the striping information provided by parallel file systems and rearrange I/O requests from processes to match the data layout on the disks of the I/O nodes in order to turn non-sequential access of disk data into sequential ones. The authors also implement this in ROMIO and see a significant performance improvement with this approach.

- **List I/O**

  Thakur et al [86], in their work on making MPI I/O portable argue that the standard POSIX I/O interface are not suitable for non-contiguous access and provide a new I/O interface called the list I/O interface. This work was extended and there was support added to these interfaces in ROMIO by developing a driver for PVFS [14] which improved the performance of these interfaces significantly [87]. But, the performance benefit starts to disappear as the size of the list become larger.

- **Listless I/O**

  Another work [88] focused on the overheads of List I/O for non-contiguous file access, including creation of lists, reading through the list before every copy operation. An on-the-fly flattening approach was suggested in this work to handle non-contiguous file access, which identifies and copies large chunks of evenly spaces, non-contiguous data using scatter-gather operations. The performance benefits were dependent on the actual access pattern. Since vector machines natively support scatter/gather operations, best performance benefits were observed here.

The main drawback with two-phase is that there is an all-to-all style communication in the beginning where all the processes exchange information with each other. Moreover each process can have a chance to communicate with all the aggregators. This can become performance prohibitive as the number of processes scales up. In addition, in one of the recent works Sehrish et al., [89] state that with increasing there is a change in behavior of the two-phase algorithm, wherein there is upto 60% of time spent on communication and the rest on I/O. In their work they try to mitigate this problem by pipelining non-blocking communication requests and I/O requests, with the use of double buffering.

To overcome this limitation of two-phase algorithm two other algorithms were introduced in [90] as a part of OMPIO [1]. The first algorithm, called dynamic segmentation, is based on the two-phase collective I/O algorithm described above. The

fundamental difference between the two algorithms is that dynamic segmentation algorithm has the ability to group processes internally thereby varying the number of processes participating in communication. The static segmentation algorithm is similar to dynamic segmentation except that it optimizes the communication occurring during the shuffle operation by keeping it uniform.

2.1.3 Dynamic Segmentation Algorithm

This algorithm follows mostly the two-phase collective I/O algorithm described above. The primary goal of this algorithm is to combine data from multiple processes in order to minimize the number of I/O operations presented to the file system and avoid rewinds on the disk if possible. The first step is similar to the two-phase algorithm, wherein all the processes perform an `MPI_Allgather` operation to share the list of file offsets and number of elements to be written with each other within the given collective operation. This enables every process to have the knowledge of the operations to be performed by every other process. All processes can sort these lists in ascending order and divide them into cycles of operation, wherein there are fixed number of bytes written in each cycle.

There can be more than one aggregator process depending the number of aggregators specified and like in two-phase each aggregator will be responsible to handle I/O for a certain number of processes. The aggregator and its group of processes are partitioned into a subgroup and communication occurs only within this group. The groups of processes is decided based on 1) number of aggregators and 2) data

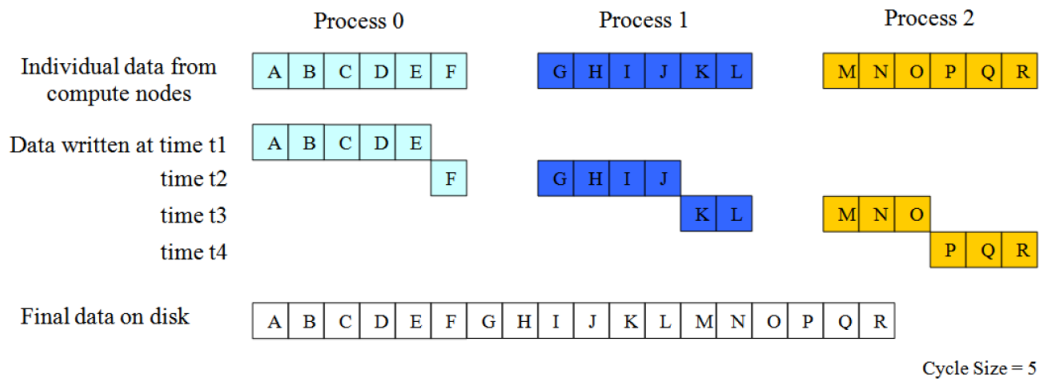

Figure 2.2: A sketch of dynamic segmentation I/O write (figure source [3]).

layout. The process groups are created such that each aggregator could have as large a contiguous chunk as possible. This is the fundamental difference between the two-phase and the dynamic segmentation algorithm. The advantage of partitioning the processes into subgroups is that we could reduce the amount of communication involved within the collective operation without compromising too much on the contiguity of the data. This algorithm can prove to be effective as we scale the number of processes as the communication costs can go up significantly in those cases. A sketch of the dynamic segmentation for a single writer has been shown in figure 2.2. Here three processes are writing collectively with a cycle size of five bytes using a single writer process, namely rank zero.

The dynamic segmentation algorithm has been used for the prototype implementation of nonblocking collective I/O operations.

2.1.4 Static Segmentation Algorithm

The static segmentation algorithm mostly follows the dynamic segmentation algorithm except that the data is always written in fixed size chunks. Since the algorithm ensures that each process contributes the same amount of data in every cycle, it makes a better use of the communication resources

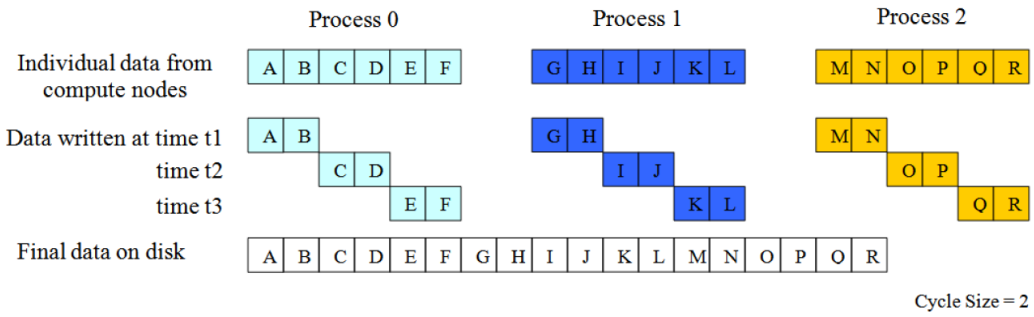

Figure 2.3: A sketch of static segmentation I/O write (figure source [3]).

Figure 2.3 shows a scenario where three processes are writing collectively with cycle buffer size of two bytes [90]. The algorithm currently supports multiple writers and multiple cycles similar to the dynamic segmentation algorithm.

This algorithm focuses on keeping the communication pattern uniform for effective use of communication resources and completely neglects irregular access pattern to the file system. This is counter-intuitive to the common knowledge which states, that file access operations are the most time consuming part of collective operations. However many large scale installations provide huge caches on the I/O nodes, which effectively decouples the compute cluster from the storage devices and thus show -

from the applications perspective - virtually no sensitivity to irregular or strided file access patterns [91]. In addition, solid state drives which have gained significant popularity in recent times have a distinct feature that they are not sensitive to random accesses in file. For these two scenarios, static segmentation focuses on optimizing the second most time consuming part.

2.1.5 Process Placement Discussion

Form the discussions in the previous sections it is clear that communication between processes in collective I/O algorithms has become a major concern. But any optimization we perform from the S/W end does not benefit if the processes themselves are connected through an interconnect with relatively poor performance. In the past, when applications had a certain pattern of communication, mapping algorithms were used to map processes (ranks) to processors (physical nodes/cores) such that the pattern of communication used in the applications always happen with the maximum bandwidth that could be achieved from that system. This is done by considering both the communication characteristics of the application and underlying network topology. For instance, two processes which communicate frequently should be mapped close to each other, thereby minimizing the communication cost, hence reducing the overall execution time of an application. In a heterogeneous environment, requirement of an efficient process placement strategy becomes even more important. Some of the related work in the area is discussed in this section

MPIPP [92] is a library designed to map parallel processes to processors such that

the total communication cost is minimized. In this framework, first the Application communication graph is obtained from the traces collected by Intel Trace Collector or Intel MPI library [54]. System topology graph is generated using a simple ping pong benchmark. In order to optimize the communication cost, MPIPP adopts a heuristic approach based on the k-way graph partitioning algorithm suggested in [93]. The algorithm starts with a randomized mapping of processes to the nodes, changes iteratively the assignment of one pair of processes and calculates the estimated performance improvement recursively. The algorithm stops when no further improvement could be obtained. MPIPP deals as of today with Symmetric Multi Processor (SMPs) and clusters, and does not consider at this point replicated MPI processes.

Mercier, et al., [94] propose a new strategy to process placement. Similar to MPIPP, they also use topology and application information to generate the optimal mapping. They propose to use a topology discovery mechanism based on the PM2 runtime system [95], which provides information about the CPU architecture, but does not provide any information regarding the underlying network topology. Further they modify the MPI implementation to collect information about the communication characteristics of the application. Now the actual mapping itself is done using the SCOTCH software [96], which implements a dual recursive bi-partitioning algorithm to compute the static mapping for the graphs.

The authors have extended their work in [97] by using the TREEMATCH algorithm, which calculate a near-optimal mapping of processes to resources on NUMA architectures. The hardware information in this work is gathered using the HWLOC library [98]. Application communication characteristics are expressed as a communication matrix where the global amount of data exchanged between each pair of processes is stored. The TREEMATCH algorithm is a graph algorithm which works recursively on each level of the memory hierarchy and groups processes such that the cost of remaining communication is minimized. Since it works with multiple levels of hierarchy, the algorithm is also well suited for heterogeneous architectures.

In his work Traff [99] formulates the topology optimization problem as a graph embedding problem and shows that topology optimizations for a hierarchical network, could be handled as a graph partitioning problem. In their work Hoefler, et al., [100] show how topology mapping gains importance with the increase in size of the network and also show how their proposed mapping strategies are capable of improving performance for special kinds of networks like fat-tree and torus. They also show that these approaches minimize network congestion. There have also been similar work done focused on specific networks like mesh and torus in [101, 102]

All the approaches discussed above, focus on optimizing the execution time using the application's communication pattern. But none of the above mentioned approaches focus on using I/O access pattern used in the application. This could be critical as for data-intensive applications can consume the maximum amount of time.

Chapter 3 discusses how data access pattern could be used for process placement and how collective I/O algorithms can benefit from that.

2.2 Non-blocking I/O operations

Non-blocking I/O was introduced in the MPI specification v2.0. The main idea of these routines was to provide a non-blocking variant of the regular write and read operations which can return immediately and could be progressed later with an `MPI_Test` or `MPI_Wait`. This provides an opportunity to hide the I/O cost, in the event of having an equally large computation part. Let us assume an example program which

- Has access to non-blocking I/O operations
- Produces data iteratively
- Does not perform write operations on the data for a while

In the ideal scenario, the non-blocking version of the write should be able to hide the I/O operation completely as shown in figure 2.4. In their work Buettner, et

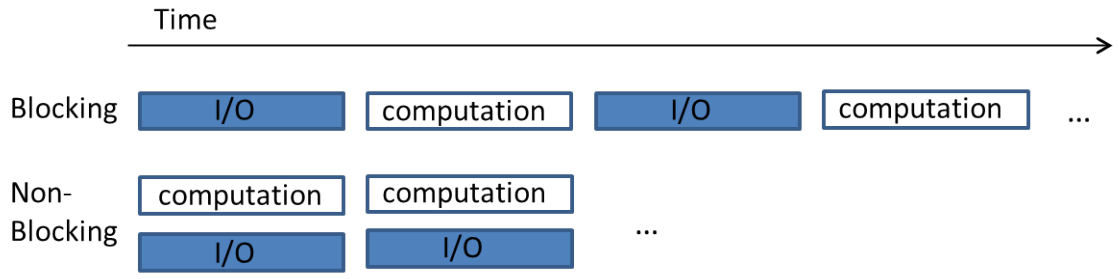

Figure 2.4: Comparison between non-blocking I/O and blocking I/O in an ideal case

al. [103] show that non-blocking I/O has the potential to reduce execution times of

applications in high performance computing.

Currently the MPI specification only supports non-blocking operations only with individual I/O. Although this is beneficial, it lacks the benefits that could be obtained by using collective I/O.

However, MPI specification offers to express collective MPI file operations as split collective interface. But, split collective interfaces have a couple of limitations:

- There must only be a single split collective active on a file handle at any time
- No other collective file I/O operations can be issued on a file handle when a split collective is active.

The first limitation prevents optimization techniques such as pipelined communications for communication/communication overlap [104] and the second limitation reduces programmability. The MPI-2.2 standard also allows to perform a global synchronization in the begin call of a split collective. This limits certain usage patterns.

Non-blocking collective operations did not exist even for communication operations also until Hoefler, et al. [4] introduced the specification for non-blocking collective communication operations. We will discuss about this in the following section.

2.2.1 Non-blocking Collective Communication Operations

Many applications benefit from overlapping communication and computation using non-blocking MPI point-to-point operations. Non-blocking collective operations to overlap communication and computation are not supported in the MPI standard

version 2. It combines the advantages provided of both non-blocking communication [105] and collective communication [106]. The approach used to emulate this functionality is to perform the blocking collective operation in a separate thread on a duplicate communicator. This approach would require an MPI implementation with the MPI_THREAD_MULTIPLE support without a "*big lock*" and there are not many implementations that offer this feature. In addition, managing threads is a very complex task for programmers. To overcome these issues a specification for non-blocking collective operations were introduced [4]. These collective operations offer the possibility of overlapping communication and computation for collective operations and also reduce the effect of pseudo synchronization. Pseudo synchronization occurs due to data dependencies in the communication pattern (receivers have to wait for senders). A non-blocking collective operation helps to move the pseudo-synchronization to the background. This provides some tolerance to process skew from the applications perspective.

A portable prototype implementation for non-blocking collective communication operations has been provided in LibNBC [107]. LibNBC is based on MPI-1 and written in ANSI C to enable high portability to many different parallel systems. This library uses so called collective schedules to save the necessary operations to complete a MPI collective communication. These schedules are built with helper functions and executed by the scheduler to perform the operation.

From the discussions above it is logical to say that combining non-blocking

I/O and collective I/O will prove to be beneficial. Chapter 4 of this dissertation presents the design and implementation of non-blocking collective I/O operations using LibNBC and evaluates the merits of using them in HPC applications.

2.3 I/O Delegation Approaches

Section 1.6 discussed the importance of ensuring scalability with increasing number of compute nodes. In the recent years, there has been significant effort invested in creating delegation based I/O architectures. Some of these approaches were introduced and discussed in chapter 1. The basic idea of these approaches is to reduce the overhead at the I/O servers by adding an additional layer in-between the compute nodes and the file-system's I/O servers called the forwarding layer to handle the stream of requests coming in from the various compute nodes. Although this kind of I/O architecture inherently provides support for asynchronous progress, but lack the benefits obtained from collective I/O optimizations, that client side parallel I/O functions benefit from. This section discusses some of the recent works which focus on applying collective I/O style optimization at staging I/O nodes/ forwarding layers.

DataStager [77] was one of the earliest efforts to provide support for staging I/O with ADIOS. This attempt focused on providing complete asynchronous I/O support by ensuring the completion of transfer when the function call returns. The work focused on increasing the I/O bandwidth by using RDMA operations to buffers in the staging nodes. Requests in the staging node are handled using different types of schedulers including a rate limiting scheduler, which also facilitates processing more

than one request simultaneously. This work does not use any collective I/O style optimizations at the I/O forwarding layer as the requests received are already in POSIX I/O format

ZOID [30] is another project providing an architecture with I/O forwarding layer support. This project also had similar kind of architecture as that of datastager with scheduling policies to handle requests, but has support for MPI-I/O request at the staging nodes, in contrary to having just POSIX I/O requests. In addition, this project also provides support for merging requests from different compute nodes going to the same file at the staging node by matching file handles and grouping requests in a queue [31].

Another project which uses delegation based [108] architecture for I/O is built on top of the MPI I/O implementation ROMIO [79]. In this architecture a handful of nodes are selected from the compute nodes and made exclusively as staging nodes. I/O requests are intercepted by a plugin implemented in ROMIO and sent to IO delegate nodes. These nodes further optimize I/O requests into a smaller number of large and contiguous requests. To perform such optimizations the authors use a complete cache page management system at the I/O delegate nodes. This is based on the client side caching work described in [109, 110]. The upper bound on the memory size was kept at 1GB, beyond which page eviction is done to cache other incoming requests.

One of the very recent initiatives to create a I/O software stack which incorporates

I/O delegation style architectures is the Exascale Fast Forward project [5]. This project uses multiple layers, aiming to support scalability requirements of the future without compromising on the I/O bandwidth. Chapter 5 of this dissertation talks about the EFF stack in detail and also introduces a optimization framework called *compactor* which has been designed for providing collective I/O style optimizations in this stack.

Chapter 3

Data Locality Aware Process Placement for Collective I/O Operations

The complexity of modern micro-processors and the utilization of hierarchical networks makes process placement, i.e., the decision on where to place each MPI process, an increasingly difficult but important task. Various projects aim to map the communication pattern of the application to the underlying hardware such that process pairs with high communication volume are close to each other from the hardware perspective [92, 97], often focusing on specific network topologies such as torus or mesh networks [101, 102], hierarchical networks [99] or by minimizing network congestion [100]. A detail description of these approaches has also been provided in section 2.1.5.

An often overlooked component in the process placement research is the I/O

occurring in the application. The time spent in I/O operations dominates the overall execution time for an increasing number of data intensive applications since the communication and computational components of high-end systems evolve at a faster rate than the storage components. MPI I/O as discussed in section 1.4 has been shown to be beneficial for the I/O performance of many application due to features such as the fileview – which allows registering the I/O access pattern of processes in advance – and collective I/O operations, which represent the counterpart of group communication operations for file I/O. In most applications the logical organization of the processes within the fileview, i.e., the order in which processes access the file based on their offset into the file, is unique, since it is tightly coupled to the data distribution strategy used by the application.

This section presents an approach to optimize the process placement of a parallel application based on their I/O access pattern, specifically focusing on optimizing the communication occurring in collective I/O operations. This section also presents the *SetMatch* algorithm which calculates a near-optimal process placement at minimal cost that minimizes communication time in collective I/O operations. This work makes a significant contribution towards the solution of the general problem, and also presents a simplified approach for commonly occurring data access patterns such as 2-D and 3-D data distributions and process topologies.

Two collective I/O algorithms were considered in this work, two-phase [2] and dynamic segmentation [90]. Both these algorithms have been explained in detail in sections 2.1.1, 2.1.3 respectively.

3.1 Design and Implementation

For the subsequent discussion, the process placement problem can be formally described as follows. Given an architecture matrix P, where each element of the matrix P_{ij} is the bandwidth capacity between processors i and j; and an application matrix R, where each element of the matrix R_{ij} is the amount of data communicated between processes i and j. The goal is to find a mapping of processes onto processors which optimizes the total communication costs, where the costs between a pair of processes i, j is R_{ij}/P_{ij}.

From the technical perspective, the problem can be broken down into three components:

- Generating the architecture matrix
- Generating a description of the communication pattern to create the application matrix
- Map application processes to underlying node architecture such that communication cost is minimized. In the following, we discuss the most relevant aspects of our work in more details.

3.1.1 Architecture Matrix

The architecture matrix is generated by providing a description of the hardware used for the application and is based, within the context of this work, on bandwidth values between pairs of processors. The bandwidth values are determined by using a ping-pong benchmark between cores on the same node, and cores on different nodes. Based on the bandwidth the value for the architecture matrix is determined. Larger

the bandwidth, smaller is the value chosen for the matrix.

3.1.2 Application Matrix

The application matrix contains the amount of data communicated between each pair of processes. The fundamental assumption in the work is, that MPI processes reading/writing neighboring portions of a file communicate with the same aggregator processes during the collective I/O operations.

To support arbitrary access patterns, the order in which processes access the file based on the offset into the file has to be recorded. This can be done during `MPI_File_set_view` and written into a separate file, that can be used when re-executing the same problem/application. For applications not setting the file view, the offsets of each file access can be recorded during the `MPI_File_read/write` operations, although this is not supported in the current implementation. To minimize the size of the output file during the record operation, a compressed row storage (CRS) format is used to record the matrix. In the following, we illustrate how the application matrix is constructed based on the file I/O access pattern.

Consider an application in which the file view is set such that processes access the file in the following order:

$$0; 4; 1; 0; 4; 1; 5; 2; 3; 4; 1; 3; 2; 5; 4; 2; 5; 4; 0; 4; 1$$

with each number representing the rank of the process accessing the next portion of the file. Every time two processes have a neighboring region in the file, i.e. they are adjacent in the list shown above, the value representing the amount of communication

between those processes is increased by one in the application matrix. For example, processes 0 and 4 have four neighboring file regions in the sequence shown above and consequently have a value of 4 in the matrix, while the processes 1 and 5 only have one common boundary. This results in the following matrix:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 4 & 0 \\ 1 & 0 & 0 & 1 & 4 & 1 \\ 0 & 0 & 0 & 2 & 1 & 3 \\ 0 & 1 & 2 & 0 & 1 & 0 \\ 4 & 4 & 1 & 1 & 0 & 2 \\ 0 & 1 & 3 & 0 & 2 & 0 \end{pmatrix}$$

While creating the application matrix by recording the file view works for arbitrary access patterns, it requires running the application to build the application matrix. Note, that the same problem occurs in many other projects working on process placement problems [92, 97], and is therefore not specific to our approach. However, we also developed a significantly simplified methodology to generate the application matrix for certain common data distributions. The most important scenario covered by this simplified approach are applications using a 2-D data distribution and a 2-D cartesian process topology. In this particular scenario, the vast majority of the communication in the collective I/O operation will occur between processes having the same coordinate in the outermost dimension of the cartesian process topology, assuming that one process per row will act as an aggregator in the collective I/O operation. collective I/O operation. This feature has been already exploited when forming the groups for the dynamic segmentation algorithm [111]. It is however also correct for the two-phase I/O algorithm, assuming that

- processes with the same coordinate in the outermost dimension of the cartesian process topology are located on the same node
- each node has (at least) one aggregator, a strategy used for example by ROMIO [2].

The application matrix can be created in this case by assigning process pairs with the same coordinate in the outermost dimension of the cartesian grid a larger value than between process pairs in different rows of the cartesian process topology. This scenario will be referred to as the *special case* for the remainder of this section. For example, the application matrix could look as follows for a 2x4 topology:

$$\begin{pmatrix} 4 & 4 & 4 & 4 & 0 & 0 & 0 & 0 \\ 4 & 4 & 4 & 4 & 0 & 0 & 0 & 0 \\ 4 & 4 & 4 & 4 & 0 & 0 & 0 & 0 \\ 4 & 4 & 4 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 4 & 4 & 4 \\ 0 & 0 & 0 & 0 & 4 & 4 & 4 & 4 \\ 0 & 0 & 0 & 0 & 4 & 4 & 4 & 4 \\ 0 & 0 & 0 & 0 & 4 & 4 & 4 & 4 \end{pmatrix}$$

3.1.3 Mapping Algorithm

Generally speaking, once the architecture and the application matrix are available, any graph mapping algorithm from literature could be used for the mapping step. The initial focus was on the algorithm used by the MPIPP toolset [92].

This algorithm uses the heuristic k-way graph partitioning algorithm [93] as the basis with a new objective function. This algorithm uses a random mapping of processes to available cores as a starting point, swaps a pair of processes and recalculates

an objective function for each new mapping. In case the new configuration processes shows benefits, i.e. a lower value of the objective function, the modified configuration is kept, otherwise it is undone. The algorithm stops if no improvement can be made over multiple iterations.

The main drawback of this algorithm is the time it takes to run to completion for large process counts. In experiments conducted during the course of this study we found that for the 256 processes test case the algorithm can take up to 90 minutes on a typical desktop system to compute the mapping with two passes, and still produces suboptimal mapping due to the low number of passes.

Therefore, a much simpler algorithm called *SetMatch* was developed which is a simplified version of the *Treematch* [97] algorithm for these type of communication and architecture matrices. Similarly to Treematch, this algorithm first partitions the application and architecture matrix into smaller, independent sets. For the architecture matrix this can be achieved by grouping all processors/cores on the same node. For the application matrix, this is typically achieved by grouping all processes with high communication volumes. In the second step of the algorithm, each subgroup of processes that resulted from subdividing the application matrix is mapped to subgroups of the architecture matrix.

There are two main differences between the *SetMatch* and the Treematch algorithm: first, *SetMatch* only uses one level of hierarchy; second, insignificant values were chosen to be ignored in the application matrices by replacing them them with a value of 0. Specifically, any value that falls within a certain percentage of the

maximum value found (e.g. 10%) in the application matrix is used for creating independent application sets while other values are ignored. This reduces the cost involved in creating the independent groups of the application matrix, which is the main cost of the *Treematch* [97] algorithm. Pseudo-code for the *SetMatch* algorithm is provided in Algorithm 1.

Algorithm 1 ***SetMatch***

Input: num_procs, comm_matrix, arch_matrix, min_value
Output: ranklist

```
1: procedure MAP_SETMATCH
2:     make_sets                          ▷ app_sets
3:     merge_sets                         ▷ remove interleaving
4:     make_sets                          ▷ Independent arch_sets
5:     match_sets                         ▷ generate ranklist
```

The algorithm subdivides the available resources from the architecture matrix as well as the application processes into independent sets. The decision to create a set of processes is based on communication volumes for the application matrix, and on communication bandwidth for the architecture matrix. In case of the architecture matrix, group of processes that have intra-node communication cost can form an independent architecture set.

In case of the of the generalized application matrix, the min_value is chosen to be 10% from the maximum number of contiguous accesses in the matrix. For the specialized application matrix described earlier, the min_value is chosen as the product of the outer dimensions of the cartesian topology. In case of the architecture matrix the 'min_value' is selected to be the value representing intra-node communication

Algorithm 2 : Creating Initial Sets

Input: num_procs, matrix, matrix_type, min_value
Output: num_sets, sets

```
 1: function MAKE_SETS
 2:     availList[1..—N];Boolean, numsets, sets
 3:     Initialize availList
 4:     for i=0 to N do
 5:         if availList then
 6:             continue
 7:         numsets++
 8:         Allocate new set of size N and initialize
 9:         for j=0 to N do
10:             if matrix_type = APP_MATRIX then
11:                 if matrix[i][j] ≤ min_value then
12:                     Add process to the current app_set
13:                     Update Availability List
14:             if matrix_type = ARCH_MATRIX then
15:                 if matrix[i][j] ≤ min_value then
16:                     Add this process to the current arch_set
17:                     Update Availability List
```

Algorithm 3 : Merging Interleaved Sets

Input: set, numSets, numProcs
Output: finalSets, final_set

```
 1: function MERGE_SETS
 2:     Allocate and initialize final_set and finalSets
 3:     copy (final_set[0], set[0])
 4:     num_final_sets++
 5:     for i ← 1 to numSets do
 6:         setFound ← false
 7:         for j ← 0 to finalSets do
 8:             if groups_interleave(final_set[j], set[i]) then
 9:                 copy (final_set[j], set[i])
10:                 setFound ← true
11:                 break
12:         if ¬setFound then
13:             finalSets++
14:             copy (final_set[finalSets - 1], set[i])
```

Algorithm 4 : Match application and architecture sets

Input: app_set, arch_set, archSets, appSets, numProcs
Output: ranklist

```
1: function MATCH_SETS
2:     while mappedProcs ≠ numProcs do
3:         for i ← 0 to appSets do
4:             archFragment ← false
5:             for j ← 0 to archSets do
6:                 if (arch_set[j].nprocs is higher) then
7:                     archFragment ← true
8:                     map(ranklist, archSet[i], appSet[j])
9:             if ¬archFragment then
10:                map(ranklist, archSet[j], appSet[i])
11:            sort(appSet)
12:            sort(archSet)
```

cost. This has been shown in algorithm 2.

Once the initial sets are created, they are checked for interleaving, to ensure that they are completely independent. In case of the specialized application matrix this approach will result in independent sets. This cannot be guaranteed however for the generalized application matrix scenario, and necessitates a merging of interleaved sets. This is shown in algorithm 3. Once all sets are independent, each application set has to be matched to an architecture set. The goal is always to fit an entire application set into an architecture set.

To accomplish this, the algorithm locates the largest possible architecture set to match the application set. Once the architecture processes are matched, the existing architecture set is fragmented to be used for another application set. If there

are no architecture sets found that could match the application sets, the algorithm fragments the application matrix and maps it to the next biggest architecture set available. This process continues until all application sets are mapped to architecture sets. This is shown in algorithm 4.

3.1.4 Implementation

This work has been implemented in OMPIO [1], a parallel I/O framework in Open MPI [51]. While the implementation is specific to OMPIO, the conceptual aspects of this work can easily be extended and transferred to other implementations. A new component of the rank mapping framework (rmaps) has been created, the Data Locality Aware Mapping (DLA) component. In general, an rmaps component retrieves information about the allocated resources, the number of processes requested, and creates a mapping for them. The output is an array of ranks which is used to associate application processes to actual nodes. The new dla rmaps component takes an additional input file, which allows to specify either the cartesian topology or the actual application matrix generated from the fileview as an input, or direct list of ranks once calculated. The module is available for free download at `http://www2.cs.uh.edu/~venkates/OpenMPI-dla.tar.gz`, and will be contributed to the Open MPI source code in the near future.

3.2 Evaluation

The efficiency of the approach discussed in previous section is evaluated for several scenarios. The platform used is the *crill* cluster at the University of Houston which consists of 16 nodes with four 12-core AMD Opteron (Magny Cours) processors each (48 cores per node, 768 cores total), 64 GB of main memory and two dual-port InfiniBand HCAs per node. The cluster has a PVFS2 (v2.8.2) parallel file system with 15 I/O servers and a stripe size of 1 MB. The file system is mounted onto the compute nodes over the second InfiniBand network interconnect of the cluster. The cluster utilizes slurm as a resource manager.

The second platform is the MEGWARE PC Farm Atlas at TU Dresden. This cluster has a total of 92 nodes, each node having two AMD Opteron 6274 *Bulldozer* multicore chips with 64 cores total, and between 12GB and 512GB of main memory per node. A QDR InfiniBand connection is provided for the communication and the I/O infrastructure. The Lustre filesystem utilized on the cluster has a total 11 Object Storage Targets(OSTs).

Six different mapping approaches were taken into consideration for evaluation

1. `Byslot`: Linear mapping on available slots
2. `Bynode`: Round robin based mapping
3. `MPIPP`: MPIPP with cartesian topology
4. `MPIPPG`: Generalized MPIPP using the application matrix generated by recording the fileview
5. `SetMatch`: *SetMatch* algorithm with cartesian topology

6. `SetMatchG`: *SetMatch* algorithm using the application matrix generated by recording the fileview

A pre-release version of OpenMPI v1.7 was used for the measurements, including support for KNEM [112] to extract higher intra-node bandwidth. A simple point-to-point benchmark from the OSU micro-benchmark suite [113] was used to determine the intra-node and inter-node bandwidth for the architecture matrix. The intra-node bandwidth obtained with these tests in the *crill* system were peaking at around 5800MB/s – although the bandwidth dropped for larger message lengths to around 3500MB/s. The inter-node bandwidth using the DDR InfiniBand network interconnect is up to 2100 MB/s. The bandwidth for the *Atlas* system, was calculated similarly using the OSU benchmark. But the Opteron 6274 cpus in the Atlas cluster have a peculiar problem, wherein selecting neighboring cores can lead to suboptimal intra-node bandwidth as they share the same L1 instruction cache [114]. This means, in case two processes get spawned on adjacent cores, there would be a lot of cache misses, leading to suboptimal performance. To mitigate his problem, processes need to be mapped in a way such that two processes do not share the same L1 instruction cache. For the default mapping strategies such as `Bynode` and `Byslot` this was done using the Locality Aware Mapping Algorithms (LAMA) [115] developed for Open MPI [116]. For all the other approaches, an additional feature was added to the DLA component which could be triggered by an *mca* parameter. The inter-node bandwidth using the QDR InfiniBand network interconnect network was around 2100 MB/s and the intra-node bandwidth was a little above 5000MB/s for both the LAMA and the DLA based approaches. The chart 3.1 shows the bandwidth

across different message length on the systems discussed above.

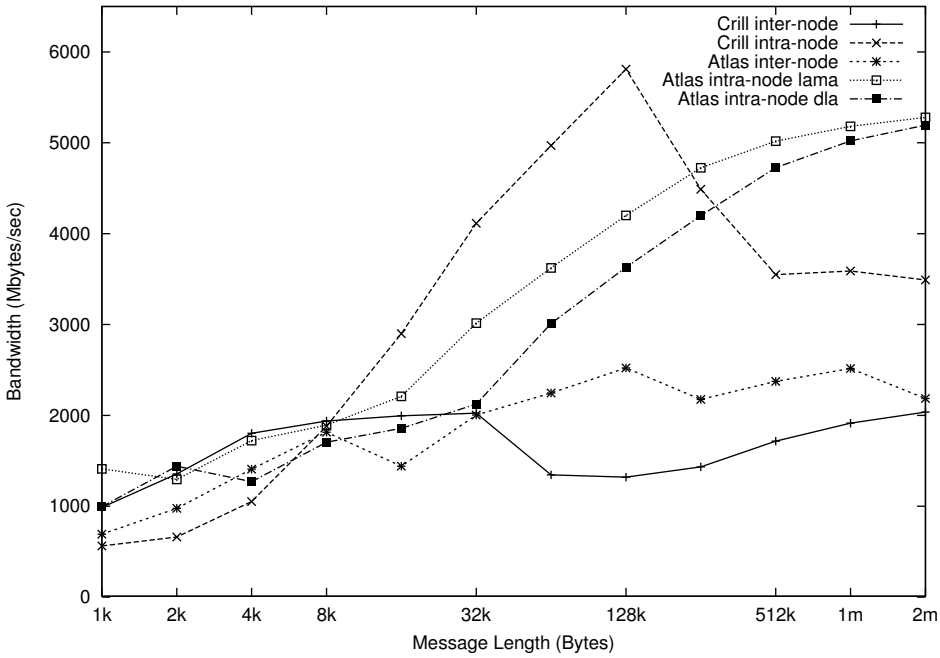

Figure 3.1: Inter and Intra node bandwidth for systems used in study.

3.2.1 Tile I/O

The MPI Tile I/O benchmark is widely used to study performance of MPI-IO for tiled accesses to a two-dimensional dense dataset [117]. The benchmark uses a cartesian communicator to access the file which makes it an ideal benchmark to study the effectiveness of our mapping strategies. For the measurements used in this chapter, the number of tiles in the x and y dimension was selected based on the number of processes. Two different tile sizes (1KB, 1MB) were used with (768x800, 40x15) elements respectively. All tests were executed three times and the maximum achieved bandwidth across those runs is selected.

Tests were executed using 128 processes on four nodes and 256 processes on eight

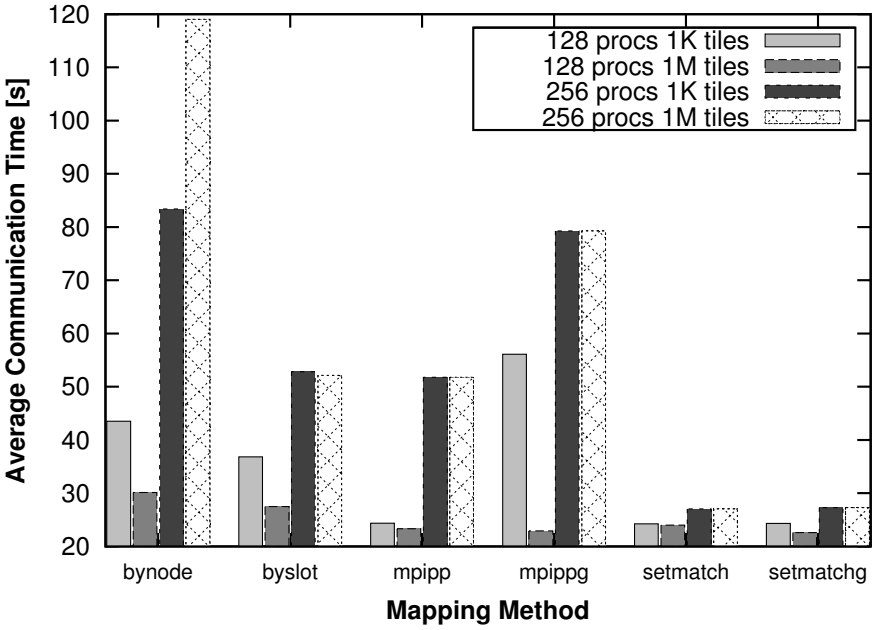

Figure 3.2: Communication time for different mapping strategies in the dynamic segmentation algorithm on crill.

nodes on the *crill* cluster. Note, that 128 processes could be executed on three nodes of this cluster, and 256 processes on six nodes. However, a slightly larger number of nodes were allocated due to performance considerations of the parallel file system used in the *crill* cluster. This leads in most scenarios to a higher number of cores being available for the process placement algorithm than actually requested by mpirun. In case of the *Atlas* cluster tests were performed using 128 processes.In addition, the number of aggregators used in the collective I/O algorithms was kept constant and same as the number of nodes used for all the tests. Figures 3.2 and 3.4 show the

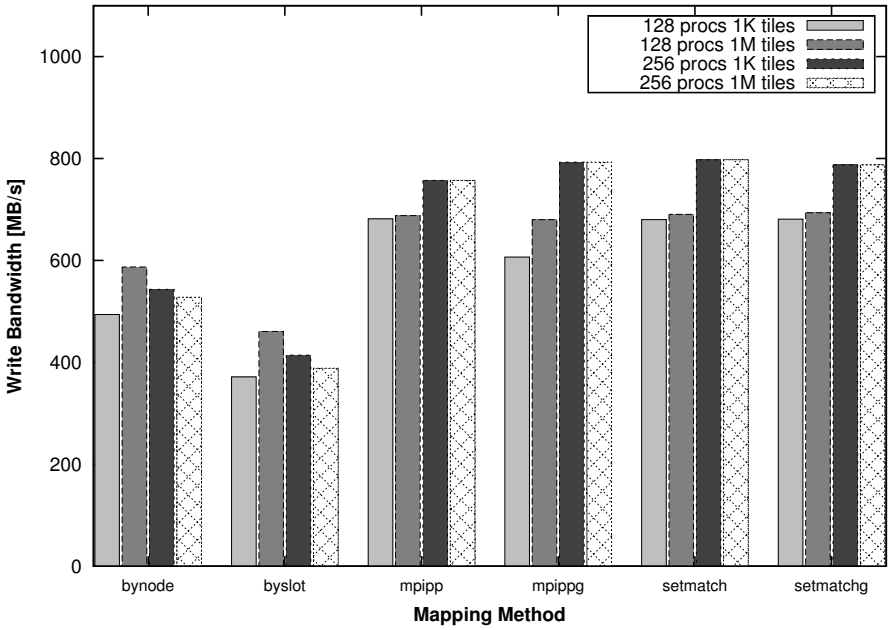

Figure 3.3: Bandwidth comparison for different mapping strategies using the dynamic segmentation algorithm on crill.

average communication time in the communication step of the dynamic segmentation and the two-phase I/O algorithm with the different mapping strategies, since the variance in these measurements was negligible.

The mapping methods developed in this dissertation show significant reduction in communication times compared to the standard bynode and byslot mapping strategies of Open MPI with the dynamic segmentation algorithm, except for the 256 processes case with the MPIPP algorithm. Analysis of the degraded performance for this scenario revealed a suboptimal mapping by the MPIPP algorithm, since the number of internal iterations had to be limited to two due to the exceeding amount of time spent in the mapping algorithm. The *SetMatch* algorithm provides the best

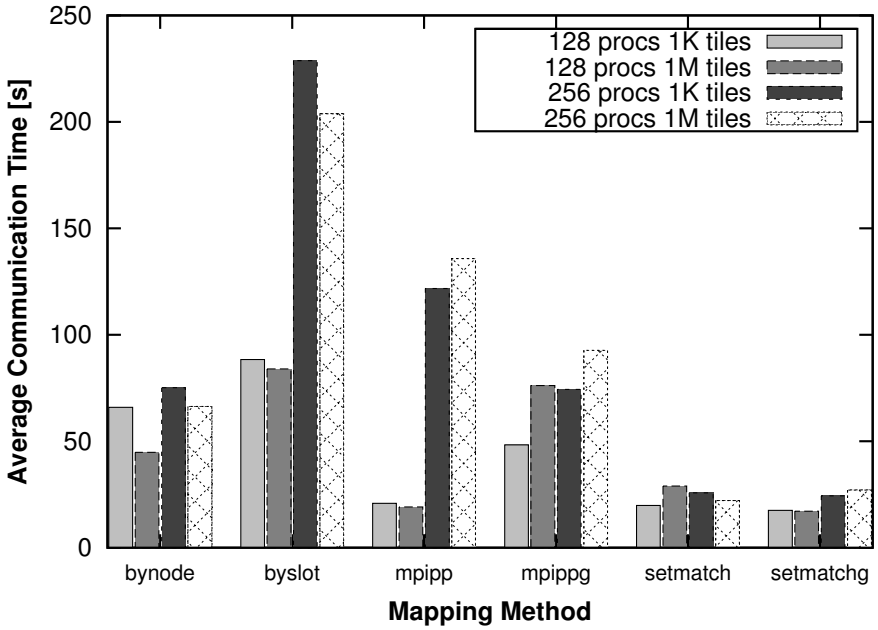

Figure 3.4: Communication time for different mapping strategies in the two-phase I/O algorithm on crill.

performance in comparison to all other methods at significantly lower costs for the mapping step: calculating the process placement for the 256 process scenario took around 100 ms on this platform. Fig 3.3 shows the resulting I/O bandwidth for Tile I/O benchmark using the dynamic segmentation algorithm. The results indicate a significant improvement in the write bandwidth along the lines of what is expected, based on the savings in the communication time. Trends are mostly similar with the two-phase I/O algorithm, with some interesting deviations as shown in figure 3.4. The first difference is evident when comparing the communication times obtained with the default bynode and byslot mappings. Based on the analysis of the data access and communication pattern of the benchmark, the byslot mapping should

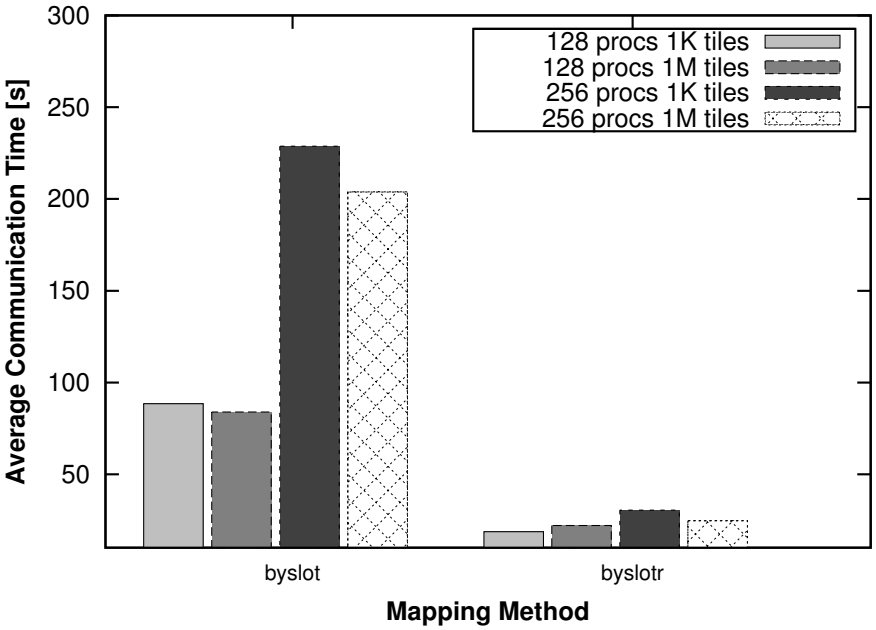

Figure 3.5: Average communication time for 48 and 32 processes per node using the byslot mapping with the two-phase I/O algorithm.

lead to lower communication times compared to the bynode mapping, similar to the results obtained using dynamic segmentation algorithm. This is however not the case in this test. The reason for this behavior is that the byslot mapping in Open MPI will fill up all the 48 cores on an individual node before assigning processes to another node, which increases the memory pressure on each node dramatically and negates the performance benefits of the intra-node communication.

Fig. 3.5 shows a slightly modified version of the same test, in which comparison is done between the default byslot mapping of Open MPI to a byslot mapping restricting the maximum number of processes per node to 32, which is the number of MPI

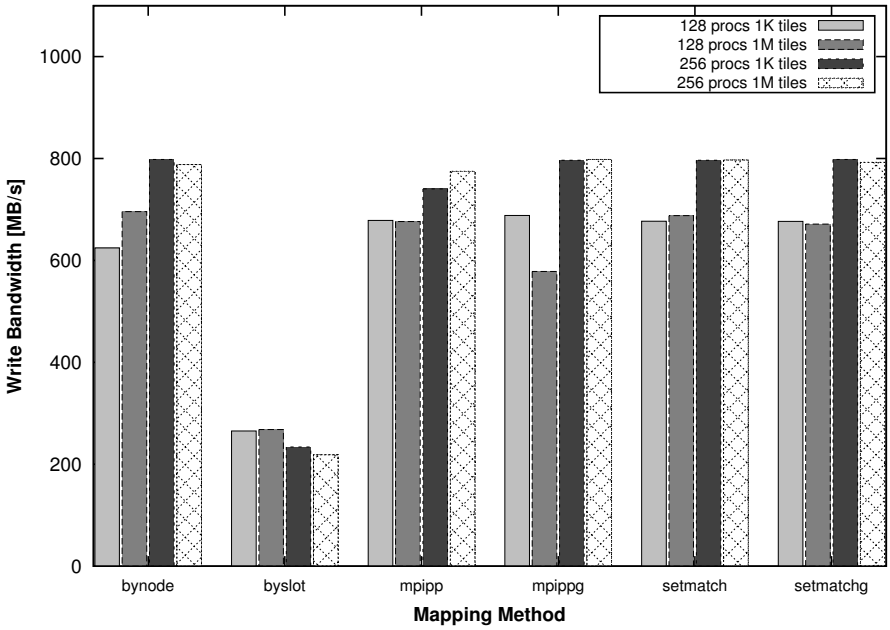

Figure 3.6: Bandwidth comparison for different mapping strategies using the two-phase algorithm on crill.

processes per node in the bynode scenario. In this case, byslot clearly outperforms bynode for the two-phase I/O algorithm as expected.

Second, there is no significant difference in the bandwidth observed using the two-phase I/O algorithm for any mapping approach as shown in figure 3.6.

Two reasons have been identified for contributing to this behavior. First, the overall bandwidth achieved was in the range of 800 MB/s, which is close to the peak bandwidth supported by this parallel file system. Second, an additional file synchronization call was added using `MPI_File_sync` to the benchmark to avoid

caching effects, which was originally not part of the Tile I/O benchmark. Removing this call led to bandwidth improvements up to 45% in the write bandwidth, although the improvement can be attributed mostly to caching effects on the server side. Ultimately, it is clear that the results observed in this test are an artifact of the parallel file system in *crill*, since expected improvements in the communication time using the two-phase I/O algorithm was observed as well.

3.2.2 Modified Tile I/O

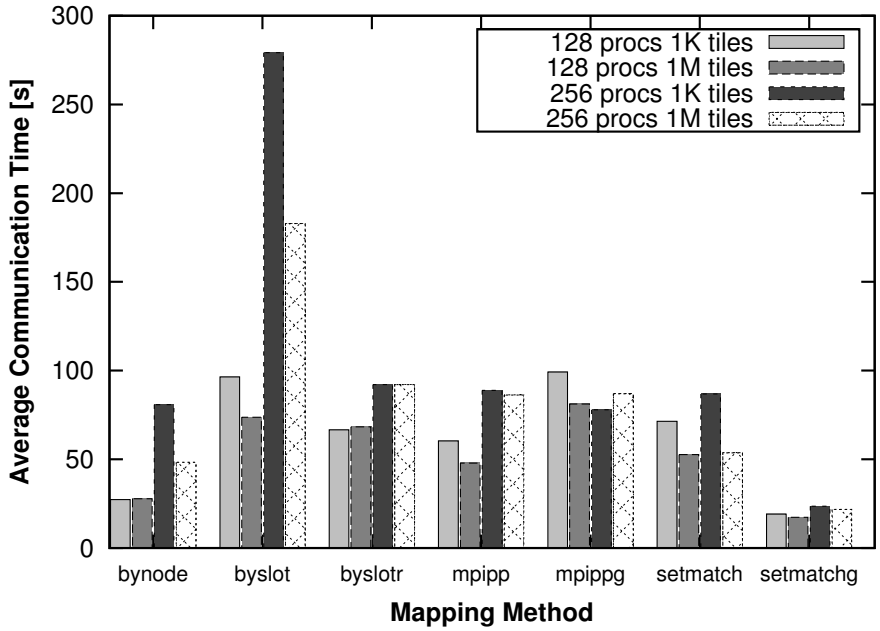

Figure 3.7: Communication time for different mapping strategies using the two-phase algorithm for the modified Tile I/O test.

In addition to the normal Tile-IO benchmark a new version version was also created where the communicator used in the benchmark was modified to reorder

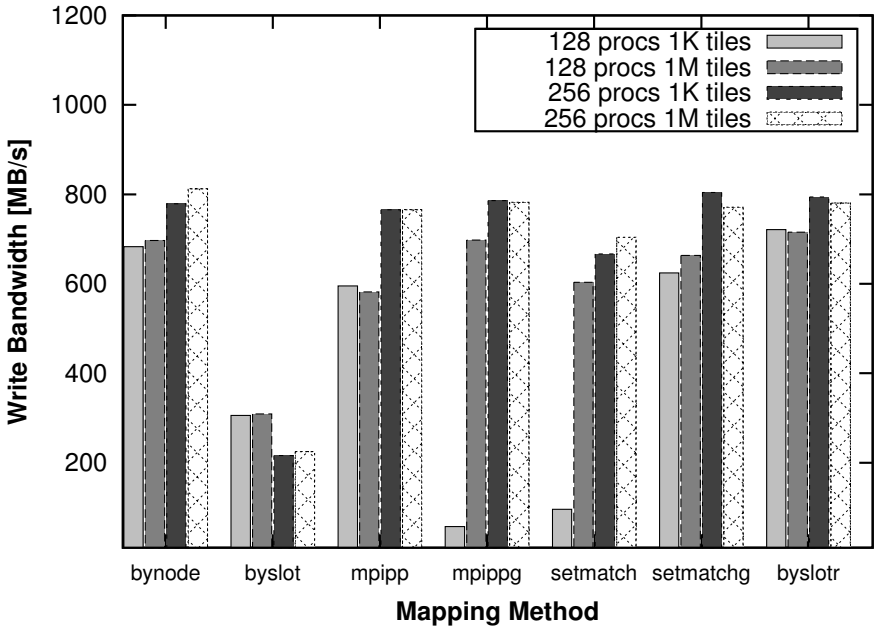

Figure 3.8: Bandwidth for different mapping strategies using the two-phase algorithm for the modified Tile I/O test.

ranks internally and thus create a more irregular scenario. In the previous tests, the byslot mapping would have provided 'accidentally' a correct mapping, except for the fact that it overloads each node with processes. For the modified Tile I/O test, neither byslot nor bynode provides the correct mapping. The goal therefore is to demonstrate that the *SetMatch* algorithm provides the optimal mapping also for this irregular scenario. Fig. 3.7 shows that only the *SetMatch* algorithm with the generalized application matrix is able to provide consistently low communication times across all scenarios which reflects in the write bandwidth observed also as shown in figure 3.8. The improvement is not as significant as the improvement obtained with the communication time due to an artifact of the parallel file system used as

discussed in the previous section.

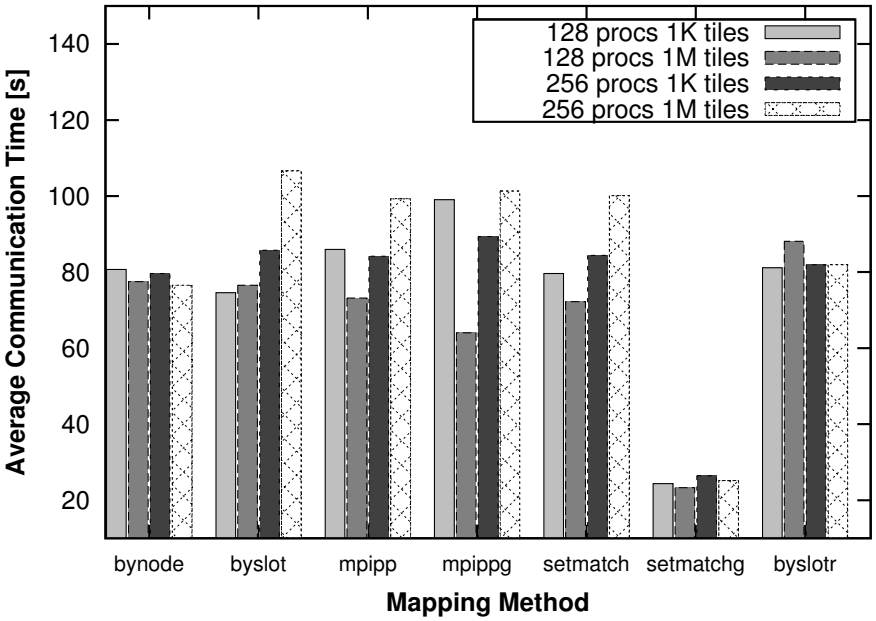

Figure 3.9: Communication time for different mapping strategies using the dynamic segmentation algorithm for the modified Tile I/O test.

The version of the MPIPP and *SetMatch* using the cartesian topology as an input can not correctly map in this scenario, since the cartesian topology is not created from `MPI_COMM_WORLD`, but from a different communicator. The generalized MPIPP algorithm can correctly describe the application matrix, but still produces a suboptimal mapping, especially for 256 processes. Also shown in this graph are numbers for a restricted byslot mapping, i.e. the mapping which limits the number of processes per node to 32. Neither bynode nor any version of the byslot produces a configuration leading to similar improvements in the communication time as the

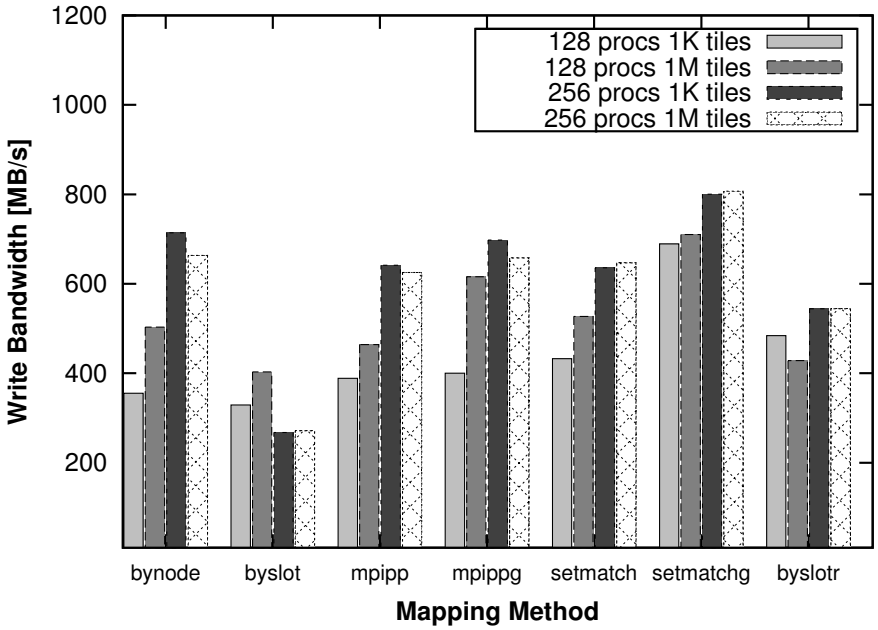

Figure 3.10: Bandwidth for different mapping strategies using the dynamic segmentation algorithm for the modified Tile I/O test.

generalized *SetMatch* algorithm. Similar performance has been observed in our experiments with even the dynamic segmentation algorithm as shown in figure 3.9. Similar improvements have also been observed in the bandwidth obtained as shown in figure 3.10.

The graph in figure 3.2.2 shows the communication times of the different approaches discussed on the Lustre file system at the Atlas cluster. It is seen that in the vast majority of cases the setmatch algorithm with the generalized matrix performs better than all the other approaches. This result is presented to show the benefit of our approaches in a different machine. The inconsistency in the results can

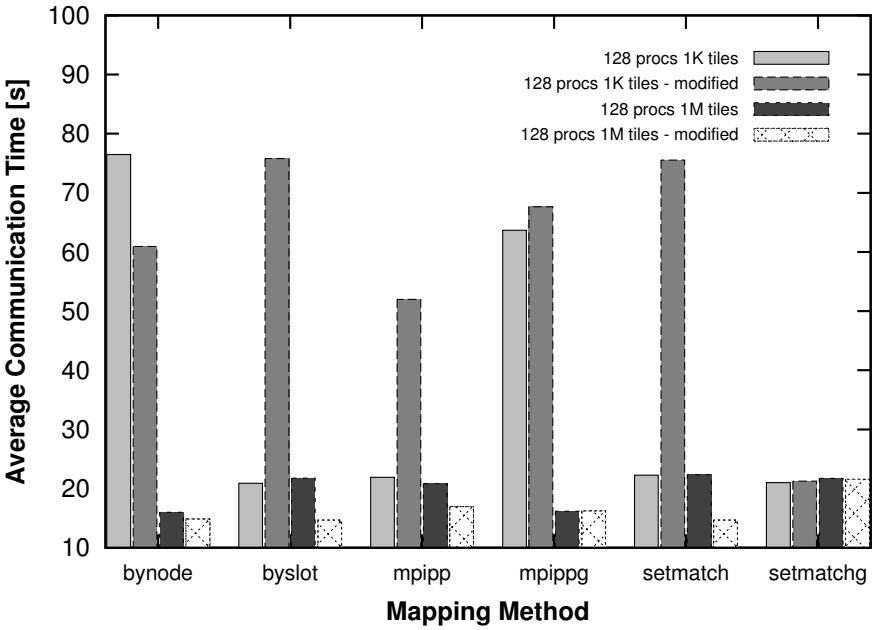

Figure 3.11: Avg communication time comparison for different mapping strategies on dynamic segmentation algorithm on Atlas

be attributed to the fact that the *Atlas* system was a shared resource and measurements were performed without exclusive access unlike the *Crill* system. Between runs, the systems state could change or additional noise could be introduced into the system through other jobs running in the same system. This could have a big impact on the measurements. This was more pronounced with the two-phase algorithm, were the results had huge variance between runs. Further investigation on this system was not performed due to its unavailability.

3.2.3 BT-I/O

BT-I/O is a part of the NAS parallel benchmarks (NPB) suite [118]. It has been developed based on one of the kernels of the BT computational kernels. The class D of the BT-I/O benchmark used in this study writes 135 GB data over 250 iterations, i.e. around 600MB of data per-iteration. Due to the smaller amount of data written per function call, the communication times per iteration will also be low, which fundamentally limits the improvement per-iteration that can be achieved. Nevertheless, the results shown in Table 3.2.3 for the two-phase I/O algorithm indicate that using the setmatchg algorithm leads up to 30% reduction in I/O time in the benchmark and 17% reduction in total application time. Similar performance was obtained using the dynamic segmentation algorithm as shown in Table 3.2.3.

Mapping	No.of Processes	Avg Total Time (s)	Avg I/O Time (s)
Bynode	144	838.91	264.09
	256	549.27	193.58
Byslot	144	1100.88	407.3
	256	1020.7	637.75
mpipp	144	769.49	223.46
	256	461.28	134.38
mpippg	144	819.25	228.01
	256	493.84	151.96
setmatch	144	805.25	243.67
	256	477.36	135.13
setmatchg	144	792.25	232.96
	256	458.42	134.98

Table 3.1: I/O time and total execution time of BT I/O using two-phase I/O algorithm.

In all, the results demonstrate significant improvements in the communication time of collective I/O operations and the application scenarios overall due to the

Mapping	No.of Processes	Avg Total Time (s)	Avg I/O Time (s)
Bynode	144	920.83	335.54
	256	587.95	257.28
Byslot	144	1287.45	580.95
	256	901.79	433.94
mpipp	144	739.93	229.87
	256	478.74	167.54
mpippg	144	878.37	297.99
	256	526.61	211.16
setmatch	144	776.17	238.36
	256	504.06	165.94
setmatchg	144	750.14	224.66
	256	463.03	163.23

Table 3.2: I/O time and total execution time of BT I/O using dynamic segmentation algorithm.

mapping strategy developed in this work. When applicable, the simplified approach for 2-D and 3-D data distributions shows significant benefits without requiring to record the file view of an application first. For generic and irregular scenarios, the generalized *SetMatch* algorithm that was presented in this chapter was able to provide adequate performance for all scenarios used in this study. This work can be expanded in multiple directions, including more and larger application scenarios and platforms. It is also possible to support process placement approaches for cases without fileview information. In addition, the special case that was demonstrated for the 2-D pattern can be extended to various other regular process distributions.

Chapter 4

Non-Blocking Collective I/O Operations

Overlapping computation and communication is a standard technique to optimize the performance of parallel applications. This technique allows to hide latencies and improve bandwidth of data transfers to remote processes. This functionality is offered to the user through a special nonblocking interface, which allows to start operations and check for completions later. Benefits of nonblocking operations have been demonstrated for point-to-point [119, 120] and nonblocking collective [4, 121] operations. The Message Passing Interface (MPI) standard specifies so called "immediate" versions of some operations. MPI-2.2 offers immediate versions of all point-to-point communication calls, individual file operations and MPI-3.0 has immediate versions of all collective communication functions. These special functions return with a handle before the operation is completed. The handle can be used to test and wait for

completion of the associated operations. Although non-blocking collective I/O operations are an obvious extension, there has been no significant work on this feature. This chapter discusses about challenges associated with developing non-blocking collective I/O operations, in-order to help hiding the costs of I/O operations.

4.1 LibNBC

The Non-Blocking Collective library (LibNBC) is a portable implementation of the non-blocking collective communication operations defined in the MPI-3.0 [122] specification. LibNBC was written based on MPI-1.0 and using ANSI C to keep it portable [107] across all platforms. Currently, LibNBC has been with integrated most of the popular MPI implementations such as Open MPI [52] and MPICH2 [53]. LibNBC supports for immediate versions of all collective communication operations defined in MPI-1 and easily extensible with new operations. The library uses collective schedules to save the necessary operations to complete a MPI collective operation.

4.1.1 Collective Schedule

A collective schedule is a process specific *execution plan* for a collective operation. It consists of all necessary information to perform the operation These are designed to support any collective communication scheme with arbitrary data dependencies. There can be multiple rounds used in the schedule to model the data dependencies. Operations in round r may depend on operations in round i $\leq$ r and are not executed by the scheduler before all operations in rounds j $<$ r have been finished. Each

collective operation can be represented as a series of point-to-point operations with one operation per round. Some operations are independent of each other, and some depend on previous operations, Independent operations can be in the same round and dependent operations have to be in the right order in different rounds.Operations are building blocks of a collective schedule. It is used to progress collective operations. Operations are grouped in rounds and executed by the scheduler. An example of a collective schedule for a tree-based `MPI_BCAST` has been shown in figure 4.1 [4]. The left side of the figure 4.1 shows the broadcast tree and the right side shows the pseudo code for the schedule. The vertices of the graph in figure 4.1 shows the rank of the processes in the communicator and the edge numbers represent the ranks. The schedule for rank 1 contains three rounds. In the first round there is only one receive operations followed by the barrier which marks the completion of the operation. This needs to be done to ensure correctness of the broadcast operation. The next two rounds with a `send` each could be combined as one round, although the second send is started only after the completion of the first send to reduce congestion in the network. The schedule itself is stored as a linear-array in memory to ensure cache-efficiency.

A handle is used to identify running collective operations, which are called instances (*an instance of a collective operation is a currently running operation which has not been completed by Test or Wait*). The handle identifies the current state, the schedule and all necessary information to progress the collective operation. Each handle is linked to a user communicator. Each communicator gets duplicated at the first use into a so-called shadow communicator. All communications done by

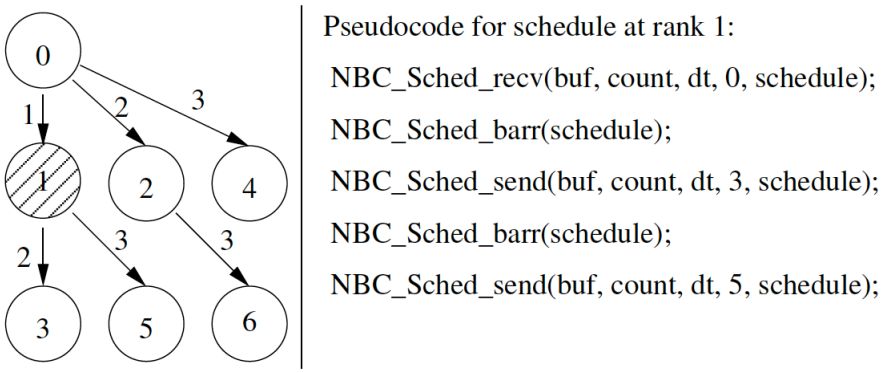

Figure 4.1: Example of collective schedule `MPI_Bcast` (figure source [4])

LibNBC is performed on the shadow communicator to prevent tag collisions on the user communicator. The handle also holds all information related to outstanding requests needed by the collective routines.

4.1.2 Progressing Non-blocking Instance

LibNBC currently uses two different kinds of progress. The first one is done inside the library, for example MPI progress (either done synchronously with `MPI_TEST` calls or asynchronously depending on the implementation). The second progress is the transition between the different rounds. The current implementation of LibNBC provides only a synchronous option for this progress, which means that users should call `NBC_TEST` for the operation to progress in the background.

4.2 Nonblocking Collective I/O Operation

A major difference between collective communication and collective I/O operations stems from the fact, that each process is allowed to provide different volumes of data to a collective read or write operation, without having knowledge on the data volumes provided by other processes. This is not the case for collective communication operations, where either each process provides exactly the same amount of data (e.g. Bcast, Reduce, Allreduce, Gather, Scatter, Allgather, Alltoall etc.) or in case of the vector version of the operations a process knows the communication volumes of all processes communicating with him (Gatherv, Scatterv, Allgatherv, Alltoallv, Alltoallw). This information is, however, essential to determine how much data a process has to contribute within a cycle of the collective I/O operation.

Thus, the first step in most collective I/O algorithms is an Allgather(v) step which determines the overall amount of data each process is contributing along with the according offsets into the file. In the case of the dynamic segmentation algorithm, this communication operation is within each group of an aggregator. This allows every process to determine how much data it has to contribute in every cycle of the algorithm. For nonblocking operations, the challenge is, that upon calling `MPI_File_iwrite_all` the according Allgather(v) operations can not be finished, since this would result in a blocking communication operation when initiating the nonblocking write-all. This is however not possible, since it could lead to a deadlocks.

Thus, the solution developed here consists of a two-step approach: while initiating the nonblocking collective read/write operation, we generate first a schedule which executes the nonblocking Allgather(v) communication step. Note, that the

operation is not exactly an `MPI_Allgatherv`, but consists of multiple Gather(v) operations executed on disjoint groups of processes in the same communicator. The last step of the Allgather(v) schedule will be executed when the Allgather(v) operation is finished, and creates a new schedule which executes the actual collective I/O operations. This second schedule contains the data gathering at the aggregator processes, the sorting based on the offsets into the file, and the asynchronous writing to the file.

Associated with that are two further challenges: first, no temporary buffers used within the collective I/O algorithm can be allocated upfront when posting the operation, since the overall amount of data and many of the according buffers are only known at the end of the Allgather(v) step. Therefore, we extended the set of operations supported by the progress engine of LibNBC in addition to nonblocking read and write by dynamic memory management functions, which allow to allocate and free buffers as part of the LibNBC schedule. Second, due to the fact that the asynchronous I/O operation are implemented using `aio_read` and `aio_write` operations which have their own data structure to identify pending operations, the LibNBC progress engine has been extended with the ability to progress multiple, different handles simultaneously, e.g. `MPI_Requests` for communication operations and the internal aio-handles for asynchronous I/O operations.

4.2.1 Schedule Caching

One of the distinct features of LibNBC is its ability to cache a schedule of a collective operation. This allows to speed up execution of operations which are posted repeatedly by an application. I/O operations generally fit the repetitive pattern required for caching a schedule, e.g. in case an application writes periodic checkpoint files. In this scenario an application has two options. The first option is to append the most recent data that has to be written to the end of an existing file. The second option would use a different file for every checkpoint. Both approaches post unique challenges for caching a schedule.

For the first option, the challenge comes from the fact that every collective I/O operation which appends data to an already existing file will lead to new offset values into the file. Moreover, the MPI standard also allows for a process to mix individual and collective I/O calls, which makes predicting the current position of the file pointer of a process impossible. Since the order in which data has to be written to the file depends on the file view and the current position of the individual file pointer, the actual amount of data that a process has to contribute to a particular cycle of the collective I/O is not necessarily repetitive, even if the arguments passed to the MPI function are identical to the previous instance. Thus, caching the schedule would not help in this scenario. The second scenario where a separate file is used for every checkpoint is equally challenging, due to the fact the schedules would be cached on a per file handle basis. This is in equivalence to the collective communication operations, where the caching is being done on a per communicator basis, although the MPI specification does not providing attribute caching functions on files as of today.

Transferring a schedule from one file handle to another file handle can theoretically be done, the challenge being however how to keep a file handle around once a file has been closed, without creating an unnecessary memory overhead.

4.3 Experimental Evaluation

The following section evaluates the impact of the nonblocking collective I/O operations. Evaluation results were obtained with a micro-benchmark and a parallel image processing application.

The system used in these tests is the *crill-cluster* at the University of Houston, which consists of 16 nodes with four 12-core AMD Opteron (Magny Cours) processor cores each (48 cores per node, 768 cores total), 64 GB of main memory and two dual-port InfiniBand HCAs per node. The parallel file system used is PVFS2 with 16 I/O servers and a stripe size of 64 KB. The file system is mounted onto the compute nodes over the Gigabit Ethernet network interconnect of the cluster. The current implementation of nonblocking I/O collective operations is tied to OpenMPI and its new parallel I/O framework (OMPIO), mostly for retrieving and maintaining file handle related aspects and for decoding derived data types and the file view. The version of Open MPI executed is equivalent to the Open MPI trunk revision 24640. In the following analysis focuses on write operations only, for the sake of simplicity.

The first test executed is using the Latency-IO micro-benchmark developed as part of the latency test suite [123], which is a micro-benchmark executing either individual or collective I/O operations. Initially, comparison is made with the performance obtained with the blocking version of the dynamic segmentation algorithm

vs. a sequence of `NBC_File_iwrite_all` followed by `NBC_Wait`. Table 4.1 presents the bandwidth achieved in both scenarios for 64 and 128 MPI processes when using 32 aggregator processes and a 4 MB cycle buffer size. The overall file size written were 63 GB and 125 GB respectively (1000MB per process). All tests have been executed three times, and the average bandwidth obtained over all three runs is presented. Note that the variation in the individual performance numbers between different runs very fairly small. The results indicate a small overhead for the 64 processes test case of the nonblocking implementation, which achieved 94% of the bandwidth obtained with the blocking version. For 128 processes the nonblocking version slightly outperformed the blocking version, which is attributed however to measurement jitter. All-in-all, the conclusion drawn from this analysis is that nonblocking implementation does not impose a significant, fundamental overhead compared to the blocking version.

Table 4.1: Performance comparison of blocking vs. nonblocking collective I/O algorithm.

No. of processes	Blocking Bandwidth	Nonblocking Bandwidth
64	703 MB/s	660 MB/s
128	574 MB/s	577 MB/s

The second test evaluates the ability to overlap collective I/O operations with compute operations. For this, the same benchmark is executing a compute function after posting the nonblocking collective write operation. The compute operation is configured to take the equal amount of time as the I/O operation. Thus, the expectation is, to observe an overall execution time equal or larger than the time

required to perform the I/O operation only for the according scenario, with the upper bound being twice the amount of time required for the same test without the compute operation in case I/O and computation cannot be overlapped. Table 4.2 presents the results achieved for the same test cases as outlined above, the first column being the time spent in the I/O test without overlap, the second column representing the time spent in writing the same amount of data and performing an equally expensive compute operation, and the third column showing the time spent in the compute operation for the overlap test.

Table 4.2: Evaluating the overlap potential of nonblocking collective I/O operations.

No. of processes	I/O only time	Overlapping time	Time spent in computation
64	85.69 sec	85.80 sec	85.69 sec
128	205.39 sec	205.91 sec	205.39 sec

The results in this section indicate the ability to entirely hide the I/O operation under optimal circumstances. These optimal circumstances are represented by the ability of libNBC to progress the operation either through a progress thread or through inserting regularly `NBC_Test` function calls into the compute operation. Within the context of this analysis, the second approach chosen. Moreover, it was also identified that the frequency and number of calls to `NBC_Test` had a tremendous influence on the overlap performance: calling it too often will introduce an additional overhead, if there are to few calls to this function, the library will not be able to progress the function. In our experimental results, the time required to execute one cycle in the dynamic segmentation algorithm was identified as the optimal interval

between two subsequent calls to `NBC_Test`.

4.3.1 An Application Scenario

Further tests have been executed with a parallel image processing application. This application is used to analyze smear sample from fine needle aspiration cytology, with the overall goal being to assist medical doctors in identifying cancer cells [124]. The challenge imposed by this application is due to the high resolution of the microscopes and the fact that images are captured at various wave-length to identify different chemical properties of the cells. For a $1cm \times 1cm$ sample with 31 spectral channels the image can contain overall up to 50GB of raw data. The MPI version of the code has furthermore the option to write the texture data into output files to facilitate future processing steps in realizing a complete computer aided diagnosis (CAD) solution. This makes the application compute and I/O intensive.

The following tests, focus on the code section which writes the texture data into files. This code sequence contains a loop in which texture data for each of the twelve Gabor filters is calculated and then written to a separate file. The computational part within this loop consists of two parallel fast-fourier transforms (FFTs), which are implemented using the FFTW library [125] version 2.1.5, and a convolution operation. For the version using the non-blocking collective I/O functions, writing the texture data in one iteration is overlapped with the execution of the FFTs and the convolution of the next iteration. Progress of the non-blocking collective I/O operation is implemented in two ways. The first one uses `NBC_Test` function calls in-between each FFT and the convolution operation. The second code version uses

a patched version of the FFTW library which contains further function calls to `NBC_Test`. Note, that the initial reading of the image and final writing of the cluster assignments have not been modified and still use the blocking collective MPI I/O version.

For evaluation purposes two separate images were used. The first image has 8192 $\times$ 8192 pixels and 21 spectral channels, writing 12 times 256 MB of texture data (3 GB total). The second image has 12281 $\times$ 12281 pixels and also 21 spectral channels, writing 12 times 576 MB of data (6.75 GB total). Tests have been executed with 64 and 96 processes on the same cluster and file system as in the previous section. Figure 4.2 show the times spent in I/O operations for the 8k image size. Figure 4.3 shows the time spent in I/O operation for the 12k image size. The average obtained over three separate runs is presented in this result also. It was also ensured that both blocking and non-blocking collective I/O operation use the same algorithm, with the same number of aggregator processes and the same cycle buffer size.

The results indicate that the version of the code which uses the FFTW library as a 'black box', i.e. without any `NBC_Test` function calls inserted, offers only little benefit compared to the original version of the code which uses blocking, collective MPI I/O operation. The main problem is the limited ability to progress the non-blocking operations without a progress thread and with a very small number of calls to `NBC_Test`. On the other hand, using the patched version of the FFTW library ensures more progress and demonstrates significant benefits of the non-blocking collective I/O operations. The benefit is more obvious for the 64 processes test cases compared to the 96 processes test cases due to the increased execution time of the

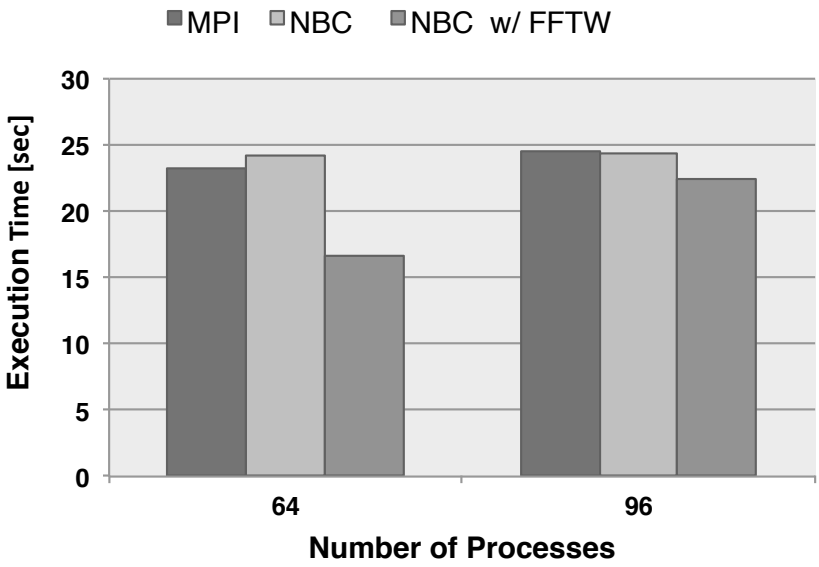

Figure 4.2: I/O times for 8k × 8k image for 64 and 96 MPI processes.

FFTs and the convolution for the 64 process test cases, which offer therefore more potential for overlapping computation and I/O operations. Hiding the entire costs of the I/O operations for a real application is however very difficult, since :

- the application has to have compute intensive sections that can be used for overlapping computation and I/O operations
- the timespan between two subsequent calls to `NBC_Test` can not be controlled in the similar manner as for the micro-benchmark.

Nevertheless, with additional `NBC_Test` calls inserted into the FFTW library we were able to reduce the time spent in I/O operation by up to 35% – which can be highly significant for large scale applications.

Non-blocking Collective I/O operations has been recommended to MPI Forum to

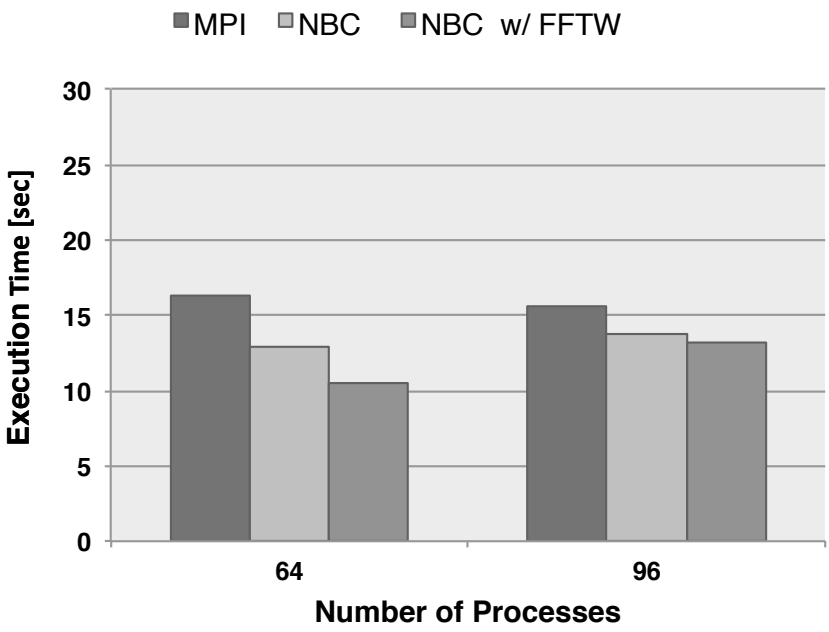

Figure 4.3: I/O times for 12k × 12k image (right) for 64 and 96 MPI processes.

be added to the MPI Specification. The approach discussed in this dissertation has been provided as a prototype to support this proposal.

It is clear from section 2.2.1 that collective I/O can have issues with pseudo synchronization. This can get more pronounced with increasing number of processes. Since I/O can be the most expensive operation for data-intensive applications, it is important to reduce the time consumed as much as possible. Non-blocking collective I/O operations tries to hide the cost involved in I/O operations as much as possible. But the way progress is made restricts the amount of overlap that could be obtained. An ideal approach should be able hide the cost involved in I/O operations completely. Chapter 5 discusses about a solution which aims at achieving this goal.

Chapter 5

Compactor: Collective I/O Optimizations using I/O Delegates

I/O Delegate/staging node based I/O architectures have gained popularity in recent years. Many large system installations at leadership scale facilities like the Blue Gene series of supercomputers [126] are adapting staging areas for the following reasons.

- To reduce workload on compute nodes (i.e to reduce resource sharing)
- To reduce OS noise on compute nodes as discussed in detail in section 1.6.
- To provide true asynchronous I/O behavior for applications at compute nodes.

There has been some work done using staging node based architectures in [30, 31, 77, 80]. Section 1.6 discusses in detail about the benefits and some of the existing approaches that use this type of architecture.

From the discussions in chapters 3, 4 so far, one could broadly summarize the principles of collective I/O as follows:

- **Combining contiguous requests:** One of the key principles of collective I/O is to merge neighboring accesses to files to a huge contiguous access. This ensures that the requests are file system friendly.
- **Optimizing accesses to the file system:** Another important feature is to have reduced number of accesses to the underlying file system. This reduces the contention at the file system servers. Choosing the optimal number of accesses becomes critical to ensure good performance.

Though, staging based I/O architectures provide benefits by shipping the I/O operation to a staging node, these architectures lack the benefits that could be obtained with collective I/O. This chapter discusses about how staging I/O nodes can benefit from collective I/O style optimizations. There has been some relevant work done in this area which has been discussed in detail in section 2.3. This work has been done as a part of the Exascale Fast Forward (EFF) [5] I/O project.

5.1 Exascale Fast Forward I/O

The Exascale Fast Forward (EFF) [5] initiative is a program sponsored by Department of Energy (DOE) for providing solutions to the next generation demands of computing. Exascale Fast Forward I/O project aims at providing an I/O stack for scalable and high performance I/O. The EFF I/O storage software stack contains several components essential to the proper functioning and performance of the application I/O. Figure 5.1 shows the entire architecture and software stack proposed in the EFF I/O project.

Applications will run on the compute nodes and will use HDF5 [65] for their I/O

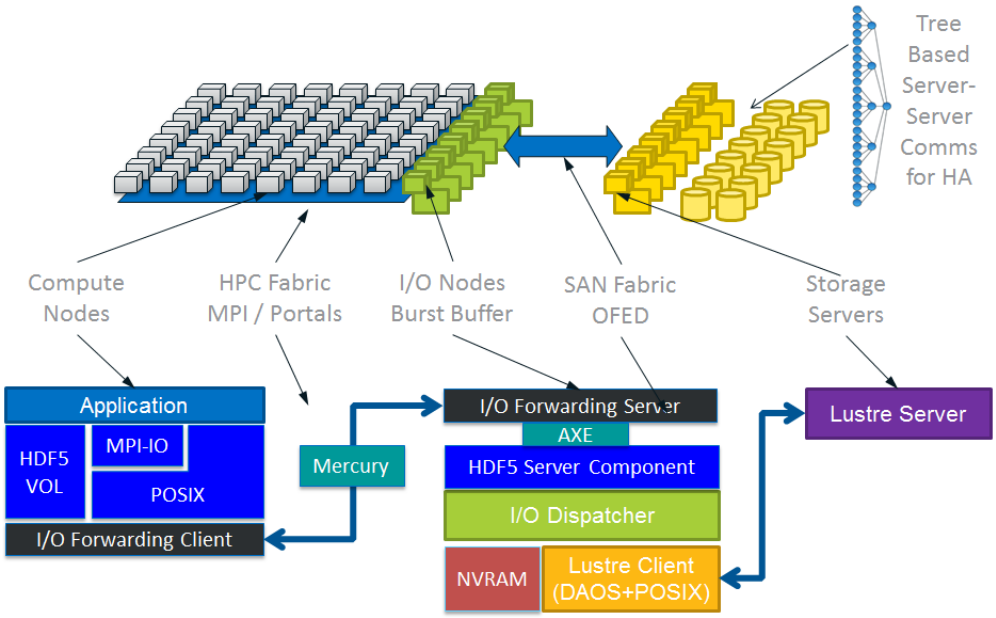

Figure 5.1: Exascale I/O Storage Software Stack [5]

needs. The HDF5 library will forward all data access operations to the I/O nodes (IONs) using Mercury [127], a fast Remote Procedure Call (RPC) mechanism. The HDF5 library asynchronously ships all operations to the server and tracks dependencies between operations. This allows for a completely asynchronous behavior at the client side. The Mercury server(s), running on the IONs, will insert the operations received into an Asynchronous Execution Engine (AXE), (explained later), with the required dependencies. When AXE schedules to execute those tasks , the HDF5 server module will translate each HDF5 call into I/O Dispatcher (IOD) call(s). IOD is built on top of PLFS [128] (Parallel Log File system), which stores data in a burst buffer (Non-Volatile RAM) and eventually migrates them to Distributed Application Object Storage (DAOS) [129] which is a next generation file system based on Lustre [130].

The work done in this chapter is limited to the HDF5 server components running on the IONs, i.e. Mercury, AXE, and HDF5-FF server. The components of the EFF I/O stack relevant to the work done in this chapter have been elaborated in the following sections

5.1.1 The Virtual Object Layer (VOL)

The Virtual Object Layer (VOL) [131] is an abstraction layer implemented just below the public HDF5 API, which intercepts HDF5 API calls that access objects in the file and forwards those calls to a plugin object driver. The plugins could actually store the objects in variety of ways. A plugin could, for example, have objects distributed remotely over different platforms, provide a raw mapping of the model to the file system, or even store the data in other file formats (like native netCDF or HDF4 format). The user still gets the same data model where access is done to a single HDF5 container; however the plugin object driver translates from what the user sees to how the data is actually stored. Figure 5.2 shows the general architecture of the HDF5 library after the VOL is inserted.

For the EFF I/O stack, the application running at the Compute Nodes (CNs) uses the HDF5 API for I/O and selects the IOD VOL Plugin for storing its data. The VOL layer captures the HDF5 API calls that access objects in file and routes them through the IOD plugin. These VOL operations are forwarded to the IONs with the help of the function shipper (mercury).

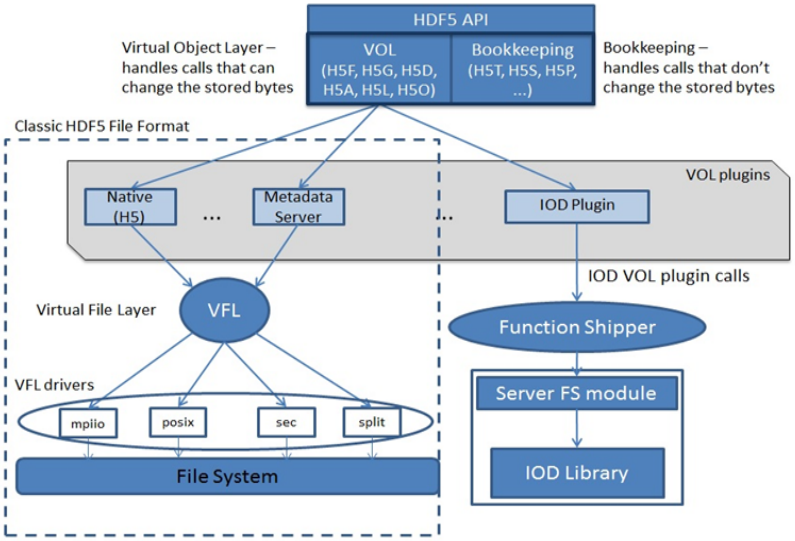

Figure 5.2: General Architecture of HDF5 library with VOL [6]

5.1.2 Mercury: Function Shipper

Mercury [127] is an Remote Procedure Call (RPC) mechanism that forwards the VOL calls made at Compute Nodes (CNs) on the client side to the IONs. Mercury is generic so it can handle various types of operations and also has a framework for extending support to other operations. Calling functions in mercury with relatively small arguments results in using the short messaging mechanism exposed by the network abstraction layer, whereas functions containing large bulk data arguments, additionally use the RMA mechanism [127]. The API for mercury is completely asynchronous. The asynchronous request objects received from mercury, are stored and tracked in the IOD VOL plugin described above.

A server side mercury module operates at the IONs to receive VOL operations from clients. VOL operations within the server side module initiate tasks that are

inserted into an asynchronous execution engine running on the ION, possibly with a dependency on another VOL operation, as indicated by the client-side VOL plugin. Each task in the asynchronous execution engine maps a VOL operation into one or more IOD API calls.

5.1.3 Asynchronous Execution Engine

The Asynchronous Execution Engine (AXE) supports asynchronous execution of tasks, which can have execution order dependencies. The AXE has several useful features:

- It provides a rich and intuitive interface for specifying functions and their dependency relationships
- An engine that asynchronously executes a function constrained to all its dependencies.
- Also provides a mechanism to monitor the status and results.
- Provides ways to define data structures to be passed around and shared across multiple function calls.

This is different from a typical non-blocking operation in the aspect that there is no need for progressing on any of the operations. The asynchronous engine maintains a thread pool from which it selects threads to progress tasks. AXE uses Directed Acyclic Graphs (DAG) to represent the dependencies and tasks, wherein the nodes of the DAG represent the tasks and the links between them represent a dependency. An example of an AXE task graph is shown in figure 5.3. The dependency in figure 5.3 should be read as follows:

- Task T2 cannot be scheduled without the completion of task T1.
- Tasks T4 and T5 can be scheduled simultaneously after the the completion of task T3.
- Task T11 can only be scheduled after either task T8 or T9 complete.
- BT13 (barrier-task) cannot be scheduled without the completion of tasks T1-T12 and so on.

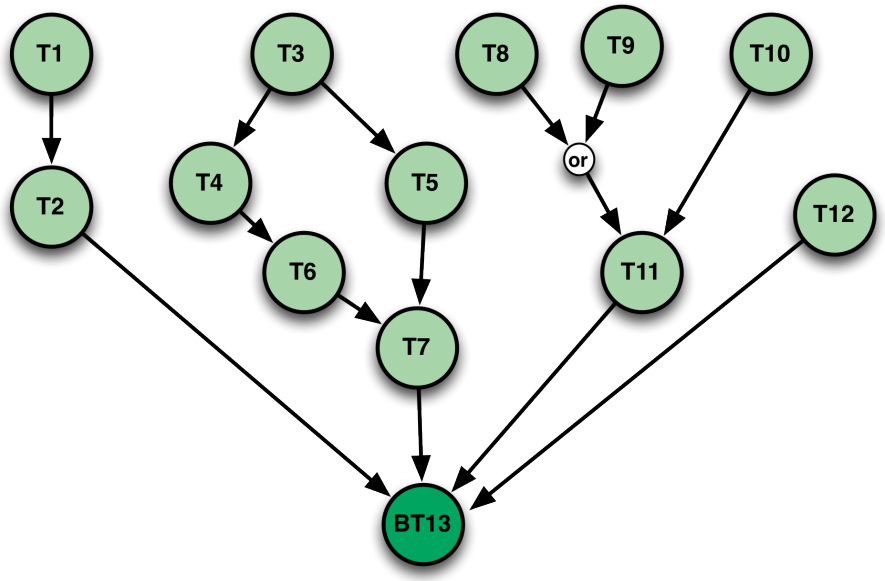

Figure 5.3: An example of an AXE task graph with a barrier task

5.2 Compactor: Design and Implementation

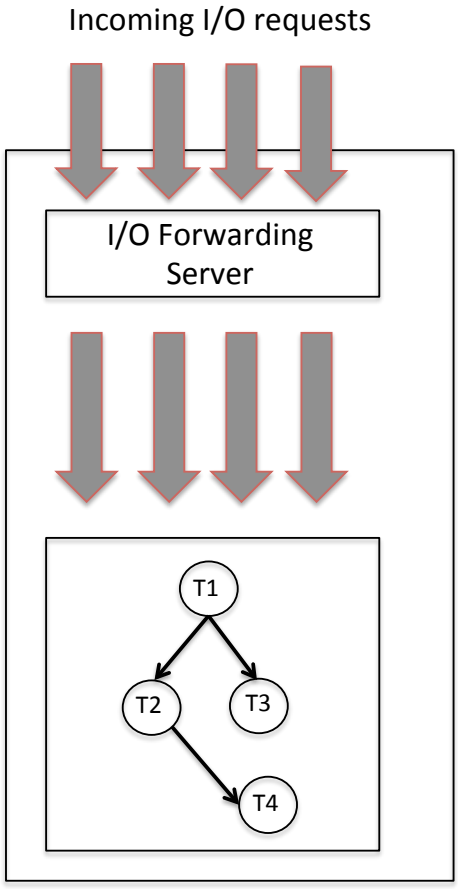

Figure 5.4: Original State of the I/O Forwarding Server

Since this work focuses on improving I/O performance at the I/O staging nodes, the focus is primarily at the I/O Forwarding Server of the EFF stack as shown in figure 5.1. As discussed in the previous sections, the primary purpose of this module is to insert the shipped VOL operations into an asynchronous engine (AXE) running on the ION, with possible dependencies on other VOL operations. These operations, when they are assigned a thread and scheduled by the AXE to execute, translate the

VOL operations into I/O Dispatcher (IOD) API calls. At this point, a feature called *"Compactor"* is developed to intercept I/O requests, before they are inserted into the AXE and look of opportunities to optimize the same. Figure 5.4 shows how I/O requests are handled originally by the I/O Forwarding Server, and figure 5.5 shows the modification to incorporate the compactor feature.

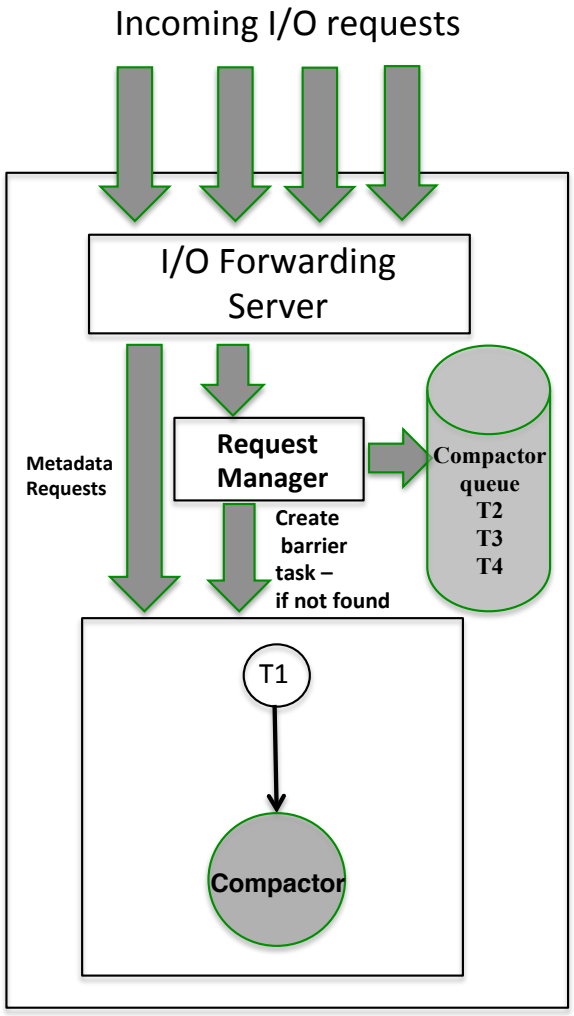

Figure 5.5: Modified I/O Forwarding Server with the Compactor

Applying optimizations to I/O requests mandates the need for accumulating as many I/O requests as possible. To accomplish this, the raw I/O requests are delayed before the AXE schedules them. This will give enough time to accumulate requests, possibly from multiple compute nodes, and determine if there could be any possibility for optimizations. Fortunately, high-performance use of the EFF storage stack already entails asynchronous execution of the raw data I/O tasks, and adding a small bit of additional delay to the execution of those tasks should have minimal effect on the application, while potentially having a large overall performance boost.

To create a delay in scheduling raw data I/O tasks, the following algorithm is used. Whenever there is a raw data I/O request:

- The request is added to the compactor queue (this is where the optimizations are applied).
- The request manager checks to see if there is a compactor task currently in the AXE. If not, a compactor task (barrier-task) is created, with dependencies on all the currently executing tasks in the AXE.
- When the compactor task is executed by the AXE, the requests in the compactor queue are examined for optimizations (merging, short-circuiting, etc.) and the resulting set of raw data I/O operations are executed by the compactor task.
- Then compactor queue is then cleaned out, and the compactor task is rescheduled in the AXE, with dependencies on the raw data I/O operations just initiated

An important point to be noted is that the compactor currently focuses only on raw I/O requests. It does not handle metadata operations/synchronous I/O requests.

The compactor task can accumulate raw asynchronous I/O requests in the queue until the AXE schedules the compactor task. Alternatively, instead of depending on I/O tasks in the AXE, this compactor task could depend on a delay task (one that just sleeps for a certain amount of time), or a task that schedules only after a few AXE tasks are queued (the optimal time-stamp/number-of-requests can be left as a configurable parameter, passed in from the client side as a hint). Unfortunately, both of these approaches have downsides:

- Having a delay task in the AXE would waste an otherwise useful thread
- Changing the AXE to schedule tasks in a different way would entail large changes to its architecture.

In this work the compactor task is delayed depending on I/O tasks in the AXE.

5.3 Optimization Approaches

This project primarily focused on three different kinds of optimizations.

- Collective Buffering : Merging I/O requests from different compute nodes
- Write Morphing : Overlapping merge of I/O requests from one client
- Write Stealing : Stealing read data from writes in the same queue

5.3.1 Collective Buffering

This is the most typical form optimization used in collective I/O algorithms. In the case of the EFF stack, the I/O requests from multiple clients enter the server. In general, number of clients associated with each I/O server, can vary from hundreds

to thousands. So there is a huge opportunity for merging requests from multiple clients, and this optimization takes advantage of this. Figure 5.6 shows an example for a typical collective buffering style optimization scenario.

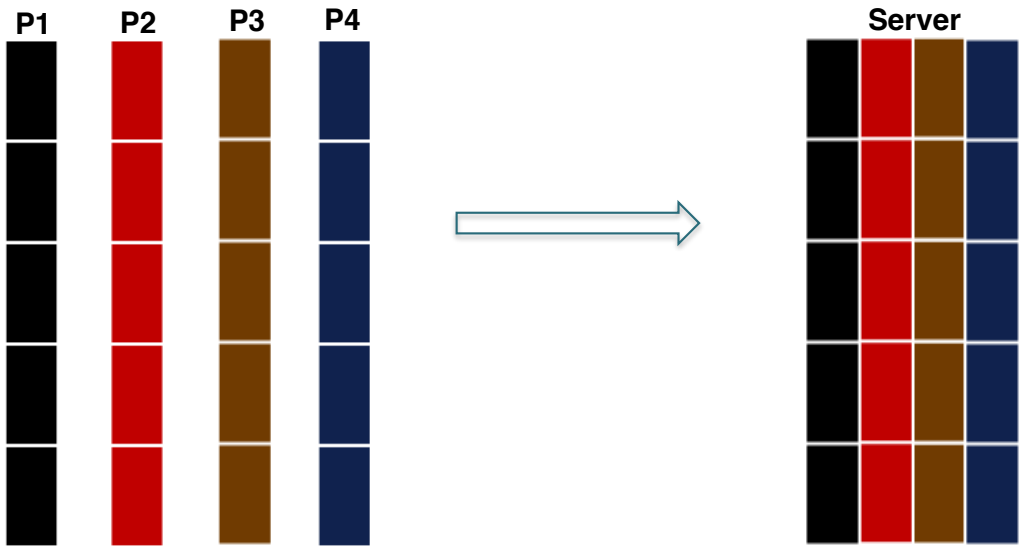

Figure 5.6: Example scenario for collective buffering

To accomplish this within the context of the compactor, the following steps have to be employed. Once the I/O requests are collected in the compactor queue, it is important to find out whether the I/O requests under consideration overlap.

- If the raw I/O requests overlap, then they are addressed as individual requests. Merging with overlapping cannot be kept consistent if there is no temporal information available. Since with the current state of the stack, it is not possible to provide temporal information from multiple clients, this scenario cannot be handled.
- If requests do not overlap, which is the scenario with most collective style I/O calls, these requests can be merged.

The main benefit of this optimization is to improve performance by reducing the contention to the underlying layers of the stack or the file system. To perform the merging, the HDF5 selection(filetype) information is extracted from each individual I/O request. To merge the filetype the in-built HDF5 routines are used. These routines work constructing *span-trees* for each filetype selection, where merging multiple selections would mean modifying the node count or adding a new node to an existing tree. This allows HDF5 to support logical operations across two selections. The main challenge is to match the memory buffers to the merged selections. The following algorithms was adopted to accomplish this:

- Each selection is broken down into pairs of offsets and lengths
- The memory buffers are also converted into <offset, length> pairs, matching each entry in the filetype <offset, length> list.
- The file type of the merged selection is also translated into <offset, length> pairs.
- Both the unmerged and merged <offset, length> pair lists are compared such that the memory address and length for each entry in the merged selection is obtained.
- With a merged selection and the list of memory addresses and lengths, both raw data HDF5 I/O or IOD I/O operations can be performed.

To establish this concept, the current version of the compactor implements this optimization only for write I/O requests.

5.3.2 Write Morphing

Overlapping writes from multiple clients cannot be overlapped because of the lack of availability of temporal information. But in case of overlapping writes from the same client we will have both temporal and spatial information. In this case, writes can be allowed to overlap. This provides interesting prospects for optimizations. An example scenario where this can be useful is a multithreaded application trying to write a file in an iterative loop, where subsequent writes overlap with the other. In such scenarios, writes could be grouped as much as possible and the write buffers could be *morphed* together to form one write. This kind of optimization will greatly reduce the number of accesses to underlying file system/layers of the stack and in-addition, reduce the redundancy in writes.

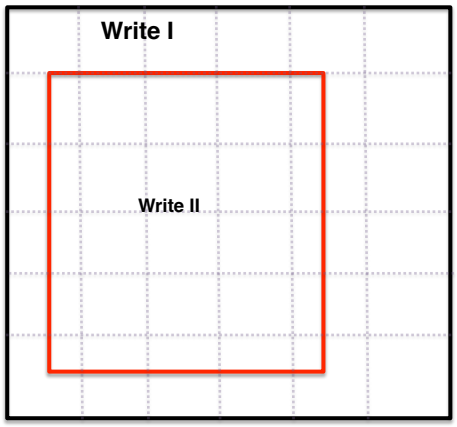

Figure 5.7: Example scenario for write morphing

Let us consider the scenario as shown in Figure 5.7. Here *Write I* happens first and *Write II* follows next. When these two write operations are merged, constructing a merged file-type is straightforward and can be done with a similar approach as

discussed in section 5.3.1, but the challenge is in *morphing* the memory buffers with data from the *latest* writes. Let us assume that the two writes depicted in figure 5.7 roughly translates to these offsets shown in Table 5.3.2. As you can see, when

Write I	Write II	Merged
0 - 32	16 - 64	0 - 16 (I)
48 - 96	16 - 64	16 - 64 (II)
96 - 128	112 - 160	64 - 80 (I)
160 - 256	176 - 208	80 - 96 (II)
		96 - 112 (I)
		112 - 160 (II)
		160 - 176 (I)
		176 - 208 (II)
		208 - 256 (I)

Table 5.1: Rough translation of figure 5.7 to offset ranges and expected merged offset ranges

we try to merge the offsets from the two writes, the memory buffer data needs to come from different writes for different offset ranges. Furthermore, to add to the complexity, there is also additional trouble with dynamically increasing size of the list of offsets. For example, if there is an overlap between offset-length [1]: 0 $\rightarrow$ 64 and offset-length [8]: 16 $\rightarrow$ 32, then offset-length[1] will be broken into 0 $\rightarrow$ 16, 32 $\rightarrow$ 64. which increases the size of the list and offset-length[8] will now be pushed offset-length[9].

5.3.3 Write Stealing

The compactor is capable of intercepting both read and write requests. This presents an unique opportunity for optimization. With read requests, its possible to compare it with the existing writes. In a scenario, where matching read and the write requests

are in the same compactor queue, we can theoretically get the read data from the write buffers directly without touching the file system.

If this optimization is applied, then there are three different ways a read I/O request can run to completion.

- Read and write dont overlap : Read entirely from the file system (case I)
- Read and write overlap completely : buffers can be entirely copied (case II)
- Read and write partially overlap : Read buffer are partially copied from writes and partially read from the file system (case III).

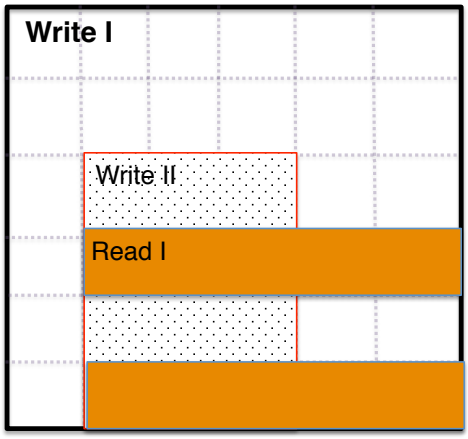

Figure 5.8: A Scenario where we have a partial overlap with writes

An example of the third scenario has been shown in figure 5.8. To support all three scenarios discussed, all the selections (filetypes) were translated to <offset, length> pairs. With this approach, the compactor can support multiple dimensions. The reads and writes are compared to check for overlaps. The parts of the filetype which overlapped were copied directly from the write operations and the remainder is serviced from the file system. Logical operations between filetypes were facilitated

by the HDF5 feature which was used in both collective buffering and write morphing optimizations.

This optimization can be highly beneficial in case of visualization applications, where one process writes to a file and the other process reads to visualize the data. In such scenarios, if both the processes send their I/O requests to the same I/O staging node, write-stealing can be performed to ensure faster response to the visualization application.

Although this optimization can be very beneficial, there are couple of constraints with this optimization

1. Writes should happen before reads
2. In case where there is a need for fault-tolerance, reads have to get committed after writes. This can reduce the performance benefits.

5.4 Evaluation

The efficiency of the optimizations discussed in section 5.3 were evaluated on the *crill* cluster at the University of Houston which consists of 16 nodes with four 12-core AMD Opteron (Magny Cours) processors each (48 cores per node, 768 cores total), 64 GB of main memory and two dual-port InfiniBand HCAs per node. The cluster has a PVFS2 (v2.8.2) parallel file system with 15 I/O servers and a stripe size of 1 MB. The file system is mounted onto the compute nodes over the second InfiniBand network interconnect of the cluster. The cluster utilizes slurm as a resource manager. MPICH2 (v3.0.2) was used with multi threading support

Since the EFF stack is still under development and the lower layers of the stack have not been completely developed, native HDF5 I/O operations were used from the HDF5 server component to complete I/O requests. Although this approach has some limitations, this was the only way to access real files from the stack in its current state. Since the motivation for this evaluation was to study the effectiveness of the Compactor, only one server was used. Moreover, modified usage of the stack means that relative performance improvements in measurements have more meaning than the actual performance values themselves. Measurements were performed using a micro benchmark developed to test the newly designed optimization and an application benchmark called Flash I/O.

5.4.1 FFBench

Fast Forward Benchmark (FFBench) suite consists of synthetically designed scenarios to evaluate the performance and efficiency of the compactor. The benchmark takes as input a configuration file to select between different benchmarks and input sizes.

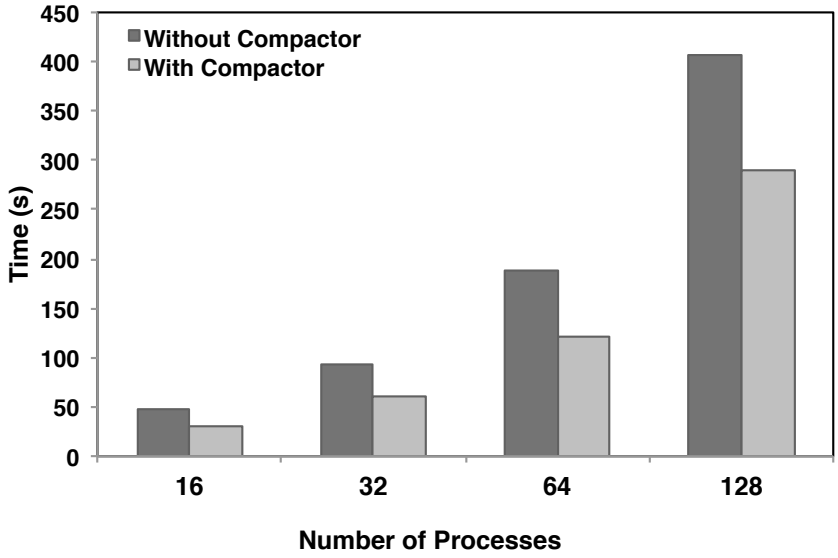

Figure 5.9: FFBench test for collective buffering

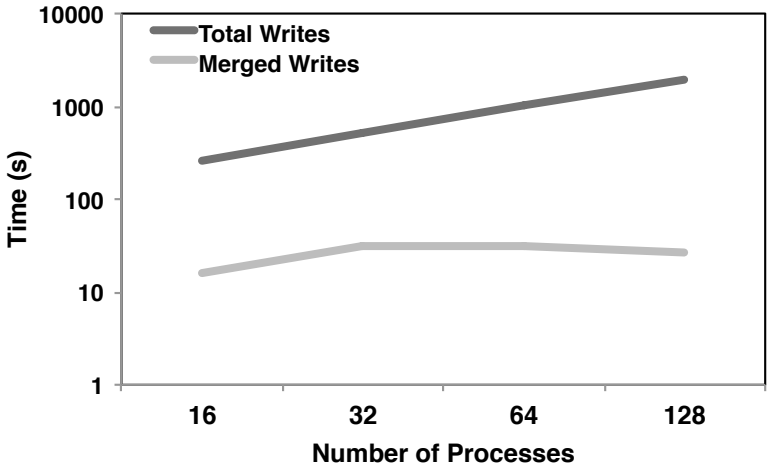

Figure 5.10: Total Write I/O requests (vs) Requests Merged

In the first test, the collective buffering optimization of the compactor was evaluated. In this test, each process writes to different rows of a 2 dimensional matrix.

The size of the matrix was kept at $(np * 512) \times 65536$. Measurements were made with 16, 32, 64, 128 processes. The graph in figure 5.9 shows that the compactor is able to accumulate write requests, and we see that there is upto 30% performance improvement in comparison to the scenario without the compactor.

The performance improvement obtained comes from the requests that were merged.

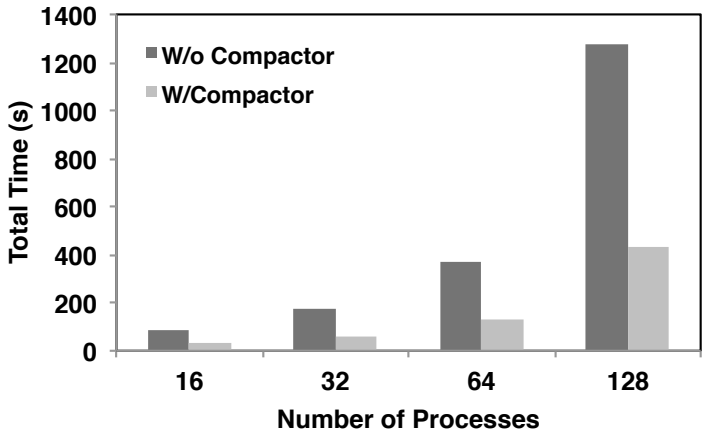

Figure 5.11: Write Morphing: Write to overlapping regions from one client

As shown in figure 5.10, irrespective of the total number of requests, the merged requests minimizes the number of writes between 16-32.

The second test focused on evaluating the write morphing optimization of the compactor by creating multiple overlapping writes to a file. The writes were created from one process as write morphing is not allowed across multiple clients. The size of the 2-D matrix was varied to see benefits. As shown in figure 5.11 we can see there is significant improvements obtained by using this approach. In the best case we see close to 40% performance improvement obtained with this optimization.

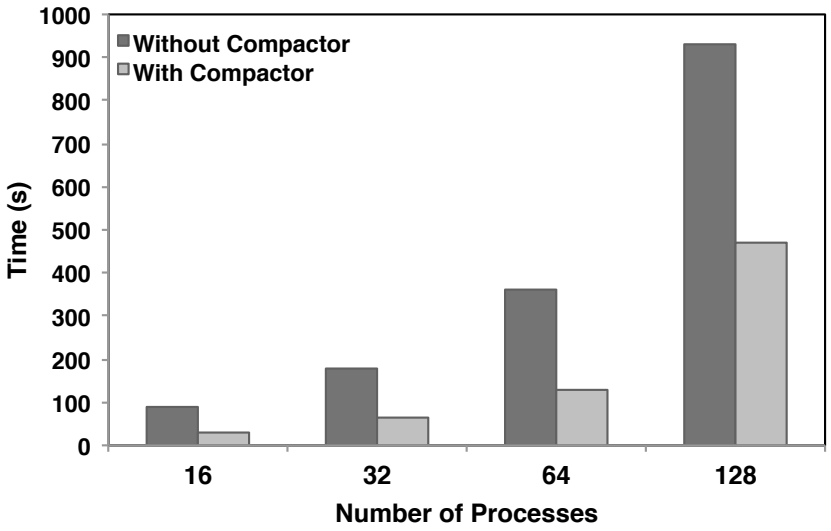

Figure 5.12: Write Stealing: Reads to overlapping write regions from multiple clients

The write-stealing optimization was evaluated by extending the benchmark used for collective buffering with reads following the writes using similar selections (file-types). The expectation is, if the reads and writes both land on the same compactor queue, then ideally we should see no time spent on reads. The figure 5.12 shows time spent in both the read and write operations. In majority of the cases the improvement obtained is close to 70%. This is because all the time spent in the read operations have been completely saved by the write stealing feature. This is more evident in the figure 5.13 which shows the percentage of time spent in the read/write operations. It is clear that with the compactor all the time spent is only for the write operation and the read time is completely amortized.

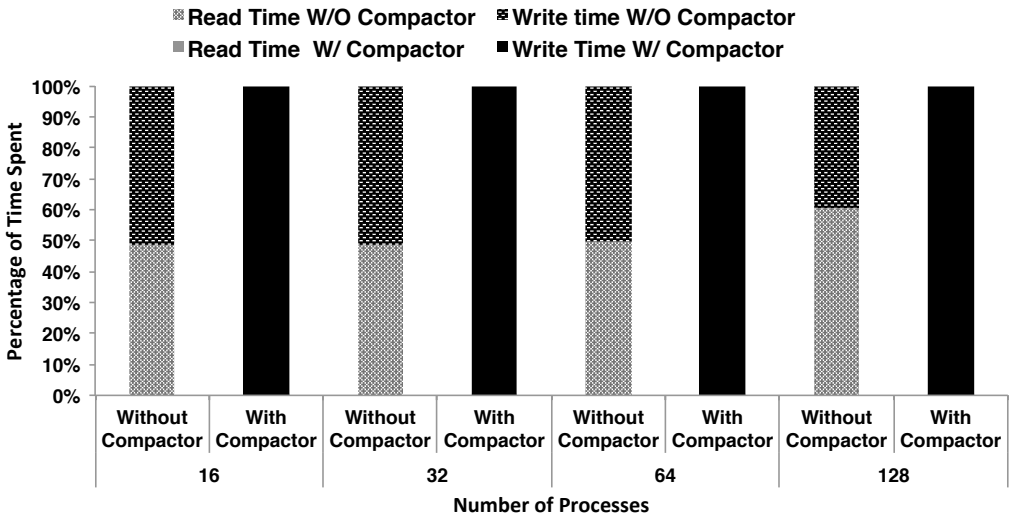

Figure 5.13: Write Stealing: Percentage of time spent in read/write for the test 5.12

5.4.2 Flash I/O

The FLASH I/O benchmark suite [132] is an extracted I/O kernel from the FLASH [133] application. The FLASH application is a lock-structured adaptive mesh hydrodynamics code that solves fully compressible, reactive hydrodynamic equations, developed mainly for the study of nuclear flashes on neutron stars and white dwarfs [133] [108]. The benchmark produces a checkpoint file, a plotfile with centered data, and a plotfile with corner data. The plotfiles have single precision data.

The results of this benchmark has been shown in 5.14. We see that there is significant improvement in the performance obtained with the compactor. All these benefits are mostly from the collective-buffering approach as there are no scenarios where write-morphing or write-stealing can benefit. Despite that, we see that there is close to 42% reduction in the time consumed. The amount of merging accomplished by the compactor can be seen in the chart 5.15.

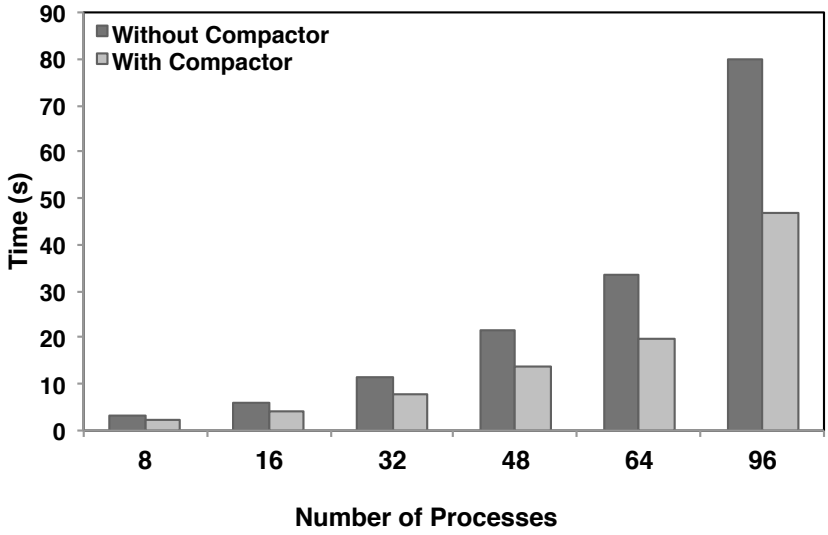

Figure 5.14: Flash I/O results for writing checkpoint files

The results for generating plot files with corners has been shown in figure 5.16 and the results for generating plot files without corners has been shown in figure 5.17. Both these consume only a fraction of the time consumed by the check-pointing operation. But we can see almost 50-55% decrease in time consumption with optimizations from the compactor. Overall from the results we can conclude that having a feature like the compactor can prove to be very beneficial for I/O stacks. It also provides a couple of new optimizations which can improve I/O performance of applications in the future.

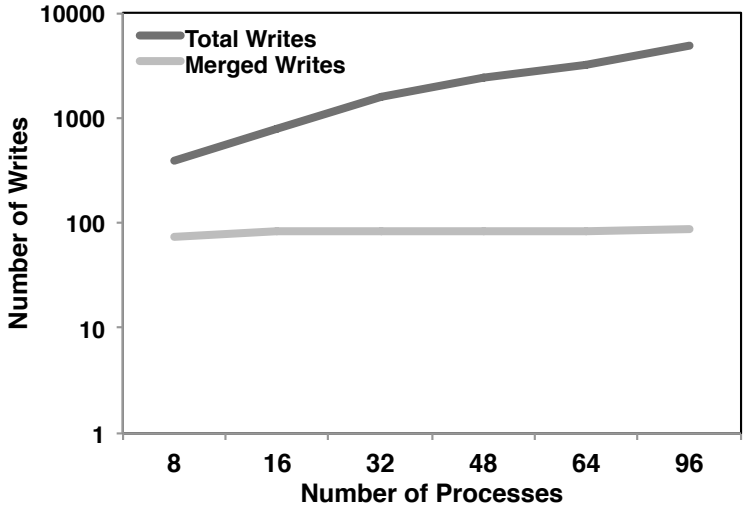

Figure 5.15: Flash I/O: Merged vs Non-merged writes

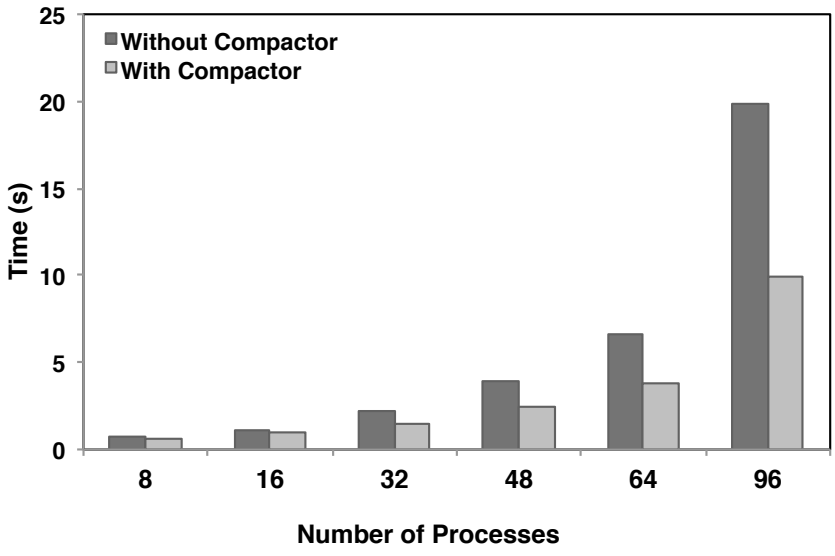

Figure 5.16: Flash I/O results for writing Plot files - with corners

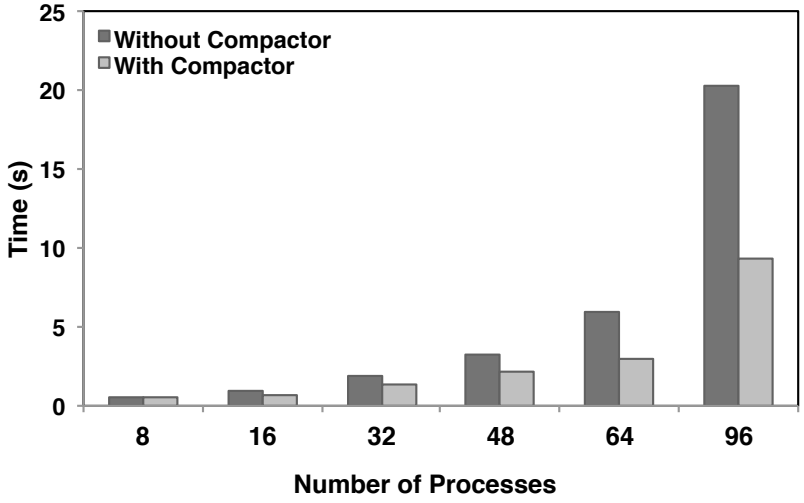

Figure 5.17: Flash I/O results for writing Plot files - without corners

Chapter 6

Summary and Future Work

This chapter summarizes the contributions of this dissertation towards improving collective I/O for high performance computing. Future work and research perspectives of this work are also outlined and elaborated.

6.1 Summary of Contributions

Increasing compute power of high performance systems further broadens the gap between the I/O time and the compute time. In addition, the use of larger number of nodes introduce significant scalability challenges such as frequent hardware crashes. I/O becomes an important component in such scenarios fault tolerance issues nee to be addressed through checkpointing. To mitigate I/O problems, various approaches were developed in the past among which one of the most widely used approach was MPI I/O. MPI I/O was introduced in $v2.0$ of the MPI specification. Amongst other interesting features, MPI I/O offers features like collective I/O and asynchronous I/O.

Although these features were able to address challenges of the previous generation, they are not sufficient for the next generation. This dissertation contributes by designing approaches to improve collective I/O and asynchronous I/O to address challenges of the future.

In the first contribution, a novel approach to improve collective I/O through process placement was introduced. With this approach the data access pattern of an application was used to create an ideal mapping of processes on a cluster, such that communication time spent in a collective I/O algorithm is minimized. This is critical to scalability because, as we increase the number of processes used for an application, the time spent in communication in a collective I/O algorithm becomes more than the time spent in the I/O operations. A new algorithm was developed to generate faster mapping and two different approaches were employed to provide data layout information. Majority of the results obtained showed that the approaches developed resulted in significant reduction of communication time. In certain cases there was close to 50% reduction in the time spent in communication in collective I/O algorithms which can result in huge improvements in the overall I/O performance.

To further reduce the time spent in collective I/O operations, it is important to overlap the time spent in I/O behind actual computation in the application. The second contribution of this dissertation focuses on making the collective I/O operations non-blocking. Non-blocking I/O operations existed in MPI specification $v2.0$ and non-blocking collective communication was standardized into the MPI specification $v3.0$. But non-blocking collective I/O operations, did not exist prior to this work. This dissertation presents the design of the first non-blocking collective I/O

operation. The evaluation shows that there was significant potential to overlap computation and I/O and there was up to 35% performance improvement in a real life application given that there could be enough progress made.

The final contribution of this dissertation focused on an approach to provide complete asynchronous support for applications to perform I/O and still benefit from collective I/O style optimizations. In particular, this work focused on developing a feature called *"Compactor"* on the Exascale Fast Forward I/O stack to provide benefits of collective I/O like contention reduction and I/O request compaction. Two new optimizations were also introduced; namely, write morphing, which is the optimization to merge overlapping write requests and write stealing, which is the optimization where reads are serviced directly from write rather than from the file system. The results showed that having a feature like the *"Compactor"* can result in significant improvement in performance. With optimizations like collective buffering we see between 30% to 50% performance improvement. In case of the write stealing optimization, we see close to 70% improvement in performance, which can be significant in case of visualization applications.

6.2 Research Perspective and Future Work

In general, there are other possible approaches with which collective I/O operations could be optimized. Collective I/O algorithms as we saw from chapters 4, 3 can be very susceptible to changes in parameters (eg, aggregators, cycle buffer size) associated with it. An optimal parameter combination can be different for each system. To decide this, a dynamic tuning technique which capable of selecting the

best algorithm from the collection of algorithms for a given system can be critical. The selection of the algorithm can also vary based on the pattern of data-access in the file. Designing a generic solution, which is capable of selecting the right algorithm for a given set of parameter can be an interesting future direction to optimize collective I/O operations. In addition, the dynamic segmentation algorithm currently works with an assumption of one aggreagator per group. But this might not be optimal. So having multiple aggregators per-group would also be an interesting extension. This also introduces additional research opportunities in determining the right size and right number of aggregators for a group in the collective I/O operation. Parallel I/O users sometimes require the use multiple fileviews usage within the same application. But setting fileviews often can be performance prohibitive, and furthermore much of the collective I/O optimizations depend on the knowledge obtained from fileviews. To help address this problem, list collective I/O operations could be introduced and this can also lead to interesting future research directions.

Apart from that, the work done in this dissertation can also be extended in several ways. Some of them are discussed in this section.

Process placement approaches for collective I/O have proven to be very beneficial. There are multiple ways in which this work could be extended. The current solution provided, looks at one data-layout pattern/per application. But in reality, there can be more than one data-layout for each application. To handle this a cumulative data layout matrix could be generated and provided as input to the mapping algorithms. This might have interesting effects and is a potential future direction. Another possible future direction is to provide support for collective I/O operations without

fileview information. In addition, this work currently only support 2D scenarios, it can also be extended to support multi-dimensional scenarios.

A good extension to the non-blocking collective I/O work would be to alter the progress approach with the Asynchronous Execution Engine (AXE) to have completely asynchronous progress. This can have implications to the overall performance obtained from the operation and is an interesting future direction for this work.

The work done with optimizing I/O requests at staging nodes can be extended in multiple ways. Unlike traditional client side collective I/O operations, optimizations at staging node really depend on how many requests can be accumulated at the server. Currently the existing approaches and approach discussed in this dissertation accomplish this with the help of a timed request manager. But this might not be ideal at all circumstances and it would be beneficial to have deterministic request accumulation at the server end. If the user understands that certain requests could be compacted, it can be set as a hint to the library such that, requests can be accumulated based on number of requests rather than time. In addition, optimizations at staging nodes largely depend on the request being asynchronous, but even if the user uses synchronous I/O requests there are opportunities to optimize those, which could be a extension of this work. For synchronous raw data read I/O calls, the *write stealing* optimization could be applied by matching them with pending asynchronous writes. For synchronous raw data write I/O calls, could be merged with an already en-queued asynchronous write I/O operation. Although, aggregating synchronous operations with pending asynchronous operations must be carefully performed, because this aggregation could lead to degradation performance of the

synchronous operation in cases of a small synchronous I/O request aggregated with a large asynchronous I/O request.

Although, the implementation for the work described in chapter 5 of this dissertation is specific to HDF5, it can easily be made generic to other I/O APIs such as MPI I/O or netCDF, as the algorithms are dependent only on fundamental concepts of I/O libraries. Trying to apply the algorithms discussed in chapter 5 to an I/O library like OMPIO can lead to interesting prospects for future research.

Bibliography

[1] Mohamad Chaarawi, Edgar Gabriel, Rainer Keller, Richard L. Graham, George Bosilca, and Jack J. Dongarra. Ompio: a modular software architecture for mpi i/o. In *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, EuroMPI'11, pages 81–89, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] Rajeev Thakur, William Gropp, and Ewing Lusk. Data Sieving and Collective I/O in ROMIO. In *FRONTIERS 99: Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, page 182. IEEE Computer Society, 1999.

[3] Mohamad Chaarawi. *Optimizing Performance Of Parallel I/O Operations For High Performance Computing*. PhD thesis, The University of Houston, 2011.

[4] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *Proc. of the 2007 Intl. Conf. on High Perf. Comp., Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, Nov. 2007.

[5] Eric Barton. Exascale Fast Forward I/O Project: https://users.soe.ucsc.edu/ ivo//blog/2013/04/07/the-ff-stack/, 2012.

[6] M. Chaarawi. Hdf5 virtual object layer. `https://wiki.hpdd.intel.com/download/attachments/12127153/Milestone%203.1%20Design%20HDF%20IOD%20VOL.pdf?version=1&modificationDate=1364837670178&api=v2`.

[7] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitive Approach*. Morgan Kaufmann Publishers, 2003.

[8] David Kotz and Ravi Jain. I/O in parallel and distributed systems. In Allen Kent and James G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 40, pages 141–154. Marcel Dekker, Inc., 1999. Supplement 25.

[9] J.M. del Rosario and A.N. Choudhary. High-performance I/O for massively parallel computers: problems and prospects. *Computer*, 27(3):59 –68, march 1994.

[10] Robert Louis Cloud. Problems in modern high performance parallel i/o systems. *CoRR*, abs/1109.0742, 2011.

[11] J.E. Smith, W.-C. Hsu, and C. Hsiung. Future general purpose supercomputer architectures. In *Supercomputing '90. Proceedings of*, pages 796 –804, nov 1990.

[12] John M. May. *Parallel I/O for high performance computing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[13] Rajeev Thakur, Ewing Lusk, and William Gropp. I/O in parallel applications: The weakest link. *The International Journal of High Performance Computing Applications*, 12(4):389–395, Winter 1998.

[14] PVFS2 webpage. *Parallel Virtual File System*. http://www.pvfs.org, Last accessed on April, 2011.

[15] Lustre webpage. *http://www.lustre.org*, Last accessed on April, 2011.

[16] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *In Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 231–244, 2002.

[17] Shinji Sumimoto. An overview of fujitsus lustre based file system. Technical Report Technical Report, Fujistu, 2011.

[18] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew Wolf. Managing variability in the io performance of petascale storage systems. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–12. IEEE Computer Society, Washington, DC, USA, 2010.

[19] Lu Yotong. Hpc system at nudt, 2013.

[20] China's tianhae-2 caps top 10 supercomputer, 2013.

[21] Rajeev Thakur. Parallel i/o.

[22] The Open Group Base Specifications Issue 7. `IEEE Std 1003. l-2008`.

[23] Jeremy Logan. *Improving Parallel I/O Performance Using Interval I/O*. PhD thesis, The University of Maine, 2010.

[24] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael L. Best. File-Access Characteristics of Parallel Scientific Workloads. *IEEE Trans. Parallel Distrib. Syst.*, 7(10):1075–1089, 1996.

[25] Sandra Johnson Baylor and C. Eric Wu. Parallel I/O Workload Characteristics Using Vesta. In *IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 16–29, April 1995.

[26] OpenFabrics Alliance. OpenFabrics webpage http://www.openib.org, Last accessed on April, 2011.

[27] Myricom. Myrinet webpage http://www.myri.com/myrinet/overview/, Last accessed on May, 2009.

[28] Quadrics. Quadrics webpage http://www.quadrics.com/, Last accessed on May, 2009.

[29] Dolphin. Dolphin webpage http://www.dolphinics.com/, Last accessed on May, 2009.

[30] Kamil Iskra, John W. Romein, Kazutomo Yoshii, and Pete Beckman. Zoid: I/o-forwarding infrastructure for petascale architectures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 153–162, New York, NY, USA, 2008. ACM.

[31] Nawab Ali, Philip H. Carns, Kamil Iskra, Dries Kimpe, Samuel Lang, Robert Latham, Robert B. Ross, Lee Ward, and P. Sadayappan. Scalable i/o forwarding framework for high-performance computing systems. In *CLUSTER*, pages 1–10. IEEE, 2009.

[32] Kazuki Ohta, Dries Kimpe, Jason Cope, Kamil Iskra, Robert Ross, and Yutaka Ishikawa. Optimization techniques at the i/o forwarding layer. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing*, CLUSTER '10, pages 312–321, Washington, DC, USA, 2010. IEEE Computer Society.

[33] Hasan Abbasi, Jay Lofstead, Fang Zheng, Scott Klasky, Karsten Schwan, and Matthew Wolf. Extending i/o through high performance data services. In *IN CLUSTER COMPUTING*. IEEE International, 2007.

[34] OpenMP Application Review Board. *OpenMP Application Program Interface, Ver. 2.5*, May 2005.

[35] Barbara Chapman, Gabriele Jost, Ruud van der Pas, and Foreword by David J. Kuck. *Using OpenMP, Portable Shared Memory Parallel Programming*. MIP Press, Oct 2007.

[36] Kshitij Mehta, Edgar Gabriel, and Barbara Chapman. Specification and performance evaluation of parallel i/o interfaces for openmp. In *Proceedings of the 8th international conference on OpenMP in a Heterogeneous World*, IWOMP'12, pages 1–14, Berlin, Heidelberg, 2012. Springer-Verlag.

[37] W. Richard Stevens and Stephen A. Rago. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 2005.

[38] Intel(R) Threading Building Blocks. Product Review: Intel Threading Building Blocks, Dec 2006. http://www.devx.com/go-parallel/Article/33270/1763.

[39] Overlapping i/o and processing in a pipeline. `http://software.intel.com/en-us/blogs/2007/08/23/overlapping-io-and-processing-in-a-pipeline/`, 2007.

[40] PVM. *Parallel Virtual Machine*. http://www.csm.ornl.gov/pvm/pvm_home.html, Last accessed on May, 2009.

[41] P4. *Parallel Programming System*. http://www.netlib.org/p4/, Last accessed on May, 2009.

[42] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, PASCO '07, pages 24–32, New York, NY, USA, 2007. ACM.

[43] UPC consortium. UPC language specification v1.2, 2005. Lawrence Berkeley National Lab Tech Report LBNL-59208.

[44] Tarek El-ghazawi, Franois Cantonnet, Proshanta Saha, Rajeev Thakur, Rob Ross, and Dan Bonachea. UPC-IO: A Parallel I/O API for UPC V1.0pre10.

[45] R. Numrich and J. Reid. Co-Array Fortran for Parallel Programming. In *ACM Fortran Forum 17(2)*, pages 1–31, 1998. `http://citeseer.ist.psu.edu/numrich98coarray.html`.

[46] Co-array Fortran I/O webpage. http://www.co-array.org/caf_io.htm/, Last accessed on January, 2011.

[47] Deepak Eachempati, Alan Richardson, Terrence Liao, Henri Calandra, and Barbara Chapman. A coarray fortran implementation to support data-intensive application development. In *The International Workshop on Data-Intensive Scalable Computing Systems (DISCS), in conjunction with SC'12*, November 2012.

[48] Deepak Eachempati, Hyoung Joon Jun, and Barbara Chapman. An open-source compiler and runtime implementation for coarray fortran. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 13:1–13:8, New York, NY, USA, 2010. ACM.

[49] Barbara Chapman, Deepak Eachempati, and Oscar Hernandez. Experiences developing the openuh compiler and runtime infrastructure. *International Journal of Parallel Programming*, pages 1–30, 2012.

[50] Message Passsing Interface Forum. MPI: A Message-Passing Interface Standard, Version 2.2.

[51] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design

of a Next Generation MPI Implementation. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

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

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

[54] Intel. Intel MPI Implementation. http://www.intel.com/cd/software/products/asmona/eng/cluster/mpi/index.htm, Last accessed on May, 2009.

[55] Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface*. http://www.mpi-forum.org/, Last accessed on April, 2011.

[56] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, June 1995. `http://www.mpi-forum.org`.

[57] Ewing Lusk Rajeev Thakur, William Gropp. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *Proceedings of The 6th Symposium on the Frontiers of Massively Parallel Computation* , pages 180–187. IEEE Computer Society Press, October 1996.

[58] PanFS webpage. *Panasas File System*, 2013. http://www.panasas.com/.

[59] Rajeev Thakur, William Gropp, and Ewing Lusk. A case for using mpi's derived datatypes to improve i/o performance. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–10, Washington, DC, USA, 1998. IEEE Computer Society.

[60] Rajeev Thakur and Alok Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Sci. Program.*, 5(4):301–317, 1996.

[61] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel i/o via a two-phase run-time access strategy. *SIGARCH Comput. Archit. News*, 21(5):31–38, 1993.

[62] Jay F. Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, CLADE '08, pages 15–24, New York, NY, USA, 2008. ACM.

[63] Jay Lofstead, Fang Zheng, Scott Klasky, and Karsten Schwan. Adaptable, Metadata Rich IO Methods for Portable High Performance IO. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[64] S. A. Brown, M. Folk, G. Goucher, and R. Rew. Software for Portable Scientific Data Management. *Computers in Physics*, 7(3):304–308, May/June 1993.

[65] Hierarchical Data Format Group. *HDF5 Reference Manual*, September 2004. Release 1.6.3, National Center for Supercomputing Application (NCSA), University of Illinois at Urbana-Champaing.

[66] Jerome Sougmane. *An In-situ Visualization Approach for Parallel Coupling and Steering of Simulations through Distributed Shared Memory Files*. PhD thesis, University of Bordeaux, France, 2012.

[67] Hierarchical Data Format Group. *Parallel HDF5*. http://www.hdfgroup.org/HDF5/PHDF5/.

[68] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netcdf: A high-performance scientific i/o interface. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 39–, New York, NY, USA, 2003. ACM.

[69] PANASAS. *PANASAS ActivStor Parallel Storage Clusters* . http://www.panasas.com/activestor.

[70] James H. Laros, Lee Ward, Ruth Klundt, Sue Kelly, James L. Tomkins, and Brian R. Kellogg. Red storm io performance analysis. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, CLUSTER '07, pages 50–57, Washington, DC, USA, 2007. IEEE Computer Society.

[71] Pete Beckman, Kamil Iskra, Kazutomo Yoshii, and Susan Coghlan. Operating system issues for petascale systems. *SIGOPS Oper. Syst. Rev.*, 40(2):29–33, April 2006.

[72] Pete Beckman, Kamil Iskra, Kazutomo Yoshii, Susan Coghlan, and Aroon Nataraj. Benchmarking the effects of operating system interference on extreme-scale parallel machines. *Cluster Computing*, 11(1):3–16, March 2008.

[73] Suzanne M. Kelly and Ron Brightwell. Software architecture of the light weight kernel, catamount. In *In Cray User Group*, pages 16–19, 2005.

[74] José Moreira, Michael Brutman, José Castaños, Thomas Engelsiepen, Mark Giampapa, Tom Gooding, Roger Haskin, Todd Inglett, Derek Lieber, Pat McCarthy, Mike Mundy, Jeff Parker, and Brian Wallenfelt. Designing a highly-scalable operating system: the blue gene/l story. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[75] Rolf Riesen, Ron Brightwell, Patrick G. Bridges, Trammell Hudson, Arthur B. Maccabe, Patrick M. Widener, and Kurt Ferreira. Designing and implementing lightweight kernels for capability computing. *Concurr. Comput. : Pract.*

Exper., 21(6):793–817, April 2009.

[76] IBM journal of Research and Development staff. Overview of the ibm blue gene/p project. *IBM J. Res. Dev.*, 52(1/2):199–220, January 2008.

[77] Hasan Abbasi, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, and Fang Zheng. Datastager: scalable data staging services for petascale applications. In *Procoeedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, pages 39–48, New York, NY, USA, 2009. ACM.

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

[79] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 23–32, 1999.

[80] Arifa Nisar, Wei-keng Liao, and Alok Choudhary. Delegation-based i/o mechanism for high performance computing systems. *IEEE Trans. Parallel Distrib. Syst.*, 23(2):271–279, February 2012.

[81] R. Ross. *Parallel I/O Benchmarking Consortium.* `http://www-unix.mcs.anl.gov/rross/pio-benchmark/html/`.

[82] P. Wong and R. F. Van der Wijngaart. *NAS Parallel Benchmarks I/O Version 3.0.* Technical Report NAS-03-002, Computer Sciences Corporation, NASA Advanced Supercomputing (NAS) Division.

[83] Javier García Blas, Florin Isaila, David E. Singh, and J. Carretero. View-Based Collective I/O for MPI-IO. In *CCGRID '08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 409–416, Washington, DC, USA, 2008. IEEE Computer Society.

[84] Wei-keng Liao and Alok Choudhary. Dynamically Adapting File Domain Partioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols. In *Proceedings of the Supercomputing Conference*, 2008.

[85] Xuechen Zhang, Song Jiang, and Kei Davis. Making Resonance a Common Case: A High-Performance Implementation of Collective I/O on Parallel File Systems. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[86] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing mpi-io portably and with high performance. In *Proceedings of the sixth workshop on*

I/O in parallel and distributed systems, IOPADS '99, pages 23–32, New York, NY, USA, 1999. ACM.

[87] Avery Ching, Alok Choudhary, Kenin Coloma, Weikeng Liao, Robert Ross, and William Gropp. Noncontiguous I/O Accesses Through MPI-IO. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 104, Washington, DC, USA, 2003. IEEE Computer Society.

[88] Joachim Worringen, Jesper Larsson Traff, and Hubert Ritzdorf. Fast Parallel Non-Contiguous File Access. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 60, Washington, DC, USA, 2003. IEEE Computer Society.

[89] Saba Sehrish, Seung Woo Son, Wei-keng Liao, Alok Choudhary, and Karen Schuchardt. Improving collective i/o performance by pipelining request aggregation and file access. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 37–42, New York, NY, USA, 2013. ACM.

[90] Mohamad Chaarawi, Suneet Chandok, and Edgar Gabriel. Performance Evaluation of Collective Write Algorithms in MPI I/O. In *Proceedings of the International Conference on Computational Science (ICCS)*, volume 5544, pages 185–194, Baton Rouge, USA, 2009.

[91] Doug Balog Stephen C. Simms, Gregory G. Pike. Wide Area Filesystem Performance using Lustre on the TeraGrid. In *Teragrid Conference*, 2007. `http://datacapacitor.researchtechnologies.uits.iu.edu/lustre_wan_tg07.pdf`.

[92] Hu Chen, Wenguang Chen, Jian Huang, Bob Robert, and H. Kuhn. Mpipp: an automatic profile-guided parallel process placement toolset for smp clusters and multiclusters. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 353–360, New York, NY, USA, 2006. ACM.

[93] C.H. Lee, M. Kim, and C.I. Park. An efficient k-way graph partitioning algorithm for task allocation in parallel computing systems. In *Systems Integration, 1990. Systems Integration '90., Proceedings of the First International Conference on*, pages 748–751, 1990.

[94] Guillaume Mercier and Jérôme Clet-Ortega. Towards an efficient process placement policy for mpi applications in multicore environments. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Pass ing Interface*, pages 104–115, Berlin, Heidelberg, 2009. Springer-Verlag.

[95] Gabriel Antoniu, Luc Boug, and Raymond Namyst. Dsm-pm2: a generic, multi-protocol dsm layer for the pm2 multithreaded runtime system.

[96] Franois Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs.

In Heather M. Liddell, Adrian Colbrook, Louis O. Hertzberger, and Peter M. A. Sloot, editors, *HPCN Europe*, volume 1067 of *Lecture Notes in Computer Science*, pages 493–498. Springer, 1996.

[97] Guillaume Mercier and Emmanuel Jeannot. Improving mpi applications performance on multicore clusters with rank reordering. In *Recent Advances in the Message Passing Interface*, EuroMPI'11, pages 39–49. Springer-Verlag, 2011.

[98] Francois Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 180–186, Washington, DC, USA, 2010. IEEE Computer Society.

[99] J.L. Traff. Implementing the MPI process topology mechanism. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 28–28, 2002.

[100] T. Hoefler and M. Snir. Generic Topology Mapping Strategies for Large-scale Parallel Architectures. In *Proceedings of the 2011 ACM International Conference on Supercomputing (ICS'11)*, pages 75–85. ACM, Jun. 2011.

[101] Hao Yu, I-Hsin Chung, and Jose Moreira. Topology mapping for blue gene/l supercomputer. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[102] Abhinav Bhatele, Laxmikant V. Kale, and Sameer Kumar. Dynamic topology aware load balancing algorithms for molecular dynamics applications. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 110–116, New York, NY, USA, 2009. ACM.

[103] David Buettner, Julian Kunkel, and Thomas Ludwig. Using non-blocking I/O operations in high performance computing to reduce execution times. In *To be Published in Proc. of the 16th European PVM/MPI User's Group Meeting (Euro PVM/MPI 2009)*, September 2009.

[104] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[105] Ron Brightwell and Keith D. Underwood. An analysis of the impact of mpi overlap and independent progress. In *Proceedings of the 18th annual international conference on Supercomputing*, ICS '04, pages 298–305, New York, NY, USA, 2004. ACM.

[106] Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Trans. Program. Lang. Syst.*, 26(1):47–56, January 2004.

[107] T. Hoefler and A. Lumsdaine. Design, Implementation, and Usage of LibNBC. Technical report, Open Systems Lab, Indiana University, Aug. 2006.

[108] Arifa Nisar, Wei-keng Liao, and Alok Choudhary. Scaling parallel i/o performance through i/o delegate and caching system. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 9:1–9:12, Piscataway, NJ, USA, 2008. IEEE Press.

[109] W. K. Liao, A. Ching, K. Coloma, Alok Choudhary, and L. Ward. An implementation and evaluation of client-side file caching for MPI-IO. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10, 2007.

[110] Wei-Keng Liao, Kenin Coloma, Alok Choudhary, and Lee Ward. Cooperative Client-Side File Caching for MPI Applications. *Int. J. High Perform. Comput. Appl.*, 21(2):144–154, 2007.

[111] Mohamad Chaarawi and Edgar Gabriel. Automatically Selecting the Number of Aggregators for Collective I/O Operations. In *Workshop on Interfaces and Abstractions for Scientific Data Storage, IEEE Cluster*, pages 428–437, September 2011.

[112] Brice Goglin and Stéphanie Moreaud. KNEM: a Generic and Scalable Kernel-Assisted Intra-node MPI Communication Framework.

[113] OSU benchmark homepage. http://mvapich.cse.ohio-state.edu/benchmarks/, 2002.

[114] AMD. Shared level-1 instruction-cache performance on amd family 15h cpus. `http://developer.amd.com/wordpress/media/2012/10/SharedL1InstructionCacheonAMD15hCPU.pdf`.

[115] Joshua Hursey, Jeffrey M. Squyres, and Terry Dontje. Locality-aware parallel process mapping for multi-core HPC systems. In *IEEE International Conference on Cluster Computing*, Austin, TX, September 2011. (Poster).

[116] Joshua Hursey and Jeffrey M. Squyres. Advancing application process affinity experimentation: open mpi's lama-based affinity interface. In *EuroMPI*, pages 163–168, 2013.

[117] R. Ross. *Parallel I/O Benchmarking Consortium.* http://www-unix.mcs.anl.gov/ross/pio-benchmark.html, Last accessed on April, 2011.

[118] P. Wong and R. F. Van der Wijngaart. *NAS Parallel Benchmarks I/O Version 2.4. Technical Report.* NAS-03-002, Computer Sciences Corporation, NASA Advanced Supercomputing (NAS) Division.

[119] Ron Brightwell and Keith D. Underwood. An analysis of the impact of MPI overlap and independent progress. In *ICS '04: Proceedings of the 18th annual*

international conference on Supercomputing, pages 298–305, New York, NY, USA, 2004. ACM Press.

[120] Francoise Baude, Denis Caromel, Nathalie Furmento, and David Sagnol. Optimizing metacomputing with communication-computation overlap. In *PaCT '01: Proceedings of the 6th International Conference on Parallel Computing Technologies*, pages 190–204, London, UK, 2001. Springer-Verlag.

[121] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm. Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations. *Elsevier Journal of Parallel Computing (PARCO)*, 33(9):624–633, 9 2007.

[122] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, 09 2012. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[123] Edgar Gabriel, Graham E. Fagg, and Jack J. Dongarra. Evaluating dynamic communicators and one-sided operations for current MPI libraries. *International Journal of High Performance Computing Applications*, 19(1):67–79, 2005.

[124] Edgar Gabriel, Vishwanath Venkatesan, and Shishir Shah. Towards High Performance Cell Segmentation in Multispectral Fine Needle Aspiration Cytology of Thyroid Lesions (accepted for publication). *Computational Methods and Programs in Biomedicine*, page t.b.d., 2009.

[125] Matteo Frigo and Steven G. Johnson. The Design and Implementation of FFTW3. *Proceedings of IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[126] H. Yu, R.K. Sahoo, C. Howson, G. Almasi, J.G. Castanos, M. Gupta, J.E. Moreira, J.J. Parker, T.E. Engelsiepen, R.B. Ross, R. Thakur, R. Latham, and W.D. Gropp. High performance file i/o for the blue gene/l supercomputer. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 187–196, 2006.

[127] *Mercury: Enabling Remote Procedure Call for High-Performance Computing*, Indianapolis, IN, 2013.

[128] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. Plfs: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 21:1–21:12, New York, NY, USA, 2009. ACM.

[129] Eric Barton. Lustre - Fast Forward to Exascale. `http://www.opensfs.org/wp-content/uploads/2013/04/LUG-2013-Lustre-Fast-Forward-to-Exascale.pdf`.

[130] J. Lombardi. Lustre restructuring. https://wiki.hpdd.intel.com/download/attachments/12127153/Milestone%202.3%20Solution%20Architecture%20-%20Lustre%20Restructuring%202013-01-07.pdf?version=1&modificationDate=1364839103314&api=v2.

[131] Mohamad Chaarawi. Virtual Object Layer. https://conuence.hdfgroup.uiuc.edu/display/VOL/Virtual+Object+Layer.

[132] M. Zingale. Flash I/O Benchmark Routine parallel HDF5. http://www.ucolick.org/~zingale/flash_benchmark_io/.

[133] B. Fryxell, K. Olson, P. Ricker, and et. al. Flash: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. In *The Astrophysical Journal Supplement Series*, pages 273–334, 2000.