Improving Performance And Programmer Productivity For I/o-intensive High Performance Computing Applications

Saba Sehrish
University of Central Florida

Improving Performance and Programmer Productivity for I/O-Intensive High Performance Computing Applications

by

Saba Sehrish
BS Computer Systems Engineering, G.I.K Institute of Engineering Sciences and Technology, Pakistan 2003
MS Computer Engineering, University of Central Florida, USA 2007

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2010

Major Professor:
Jun Wang

Abstract

Due to the explosive growth in the size of scientific data sets, data-intensive computing is an emerging trend in computational science. HPC applications are generating and processing large amount of data ranging from terabytes (TB) to petabytes (PB). This new trend of growth in data for HPC applications has imposed challenges as to what is an appropriate parallel programming framework to efficiently process large data sets. In this work, we study the applicability of two programming models (MPI/MPI-IO and MapReduce) to a variety of I/O-intensive HPC applications ranging from simulations to analytics. We identify several performance and programmer productivity related limitations of these existing programming models, if used for I/O-intensive applications. We propose new frameworks which will improve both performance and programmer productivity for the emerging I/O-intensive applications.

Message Passing Interface (MPI) is widely used for writing HPC applications. MPI/MPI-IO allows a fine-grained control of assigning data and task distribution. At the programming frameworks level, various optimizations have been proposed to improve the performance of MPI/MPI-IO function calls. These performance optimizations are provided as various function options to the programmers. In order to write an efficient code, they are required to know the exact usage of the optimization functions, hence programmer productivity is limited. We propose an abstraction called **Reduced Function Set Abstraction (RFSA)** for MPI-IO to reduce the number of I/O functions and provide methods to automate the selection of appropriate I/O function for writing HPC simulation applications. The purpose of RFSA is to hide the performance optimization functions from the application developer, and relieve the application developer from deciding on a specific function. The proposed set of functions relies on a selection algorithm to decide among the most common optimizations provided by MPI-IO.

Additionally, many application scientists are looking to integrate data-intensive computing into computational-intensive High Performance Computing facilities, particularly for data analytics. We have observed several scientific applications which must migrate their data from an HPC storage system to a data-intensive one. There is a gap between the data semantics of HPC storage and data-intensive system, hence, once migrated, the data must be further refined and reorganized. This reorganization must be performed before existing data-intensive tools such as MapReduce can be effectively used to analyze data. This reorganization requires at least two complete scans through the data set and then at least one MapReduce program to prepare the data before analyzing it. Running multiple MapReduce phases causes significant overhead for the application, in the form of excessive I/O operations. For every MapReduce application that must be run in order to complete the desired data analysis, a distributed read and write operation on the file system must be performed. Our contribution is to extend Map-Reduce to eliminate the multiple scans and also reduce the number of pre-processing MapReduce programs. We have added additional expressiveness to the MapReduce language in our novel framework called **MapReduce with Access Patterns (MRAP)**, which allows users to specify the logical semantics of their data such that 1) the data can be analyzed without running multiple data pre-processing MapReduce programs, and 2) the data can be simultaneously reorganized as it is migrated to the data-intensive file system. We also provide a scheduling mechanism to further improve the performance of these applications.

The main contributions of this thesis are, 1) We implement a selection algorithm for I/O functions like read/write, merge a set of functions for data types and file views and optimize the atomicity function by automating the locking mechanism in RFSA. By running different parallel I/O benchmarks on both medium-scale clusters and NERSC supercomputers, we show an improved programmer productivity (35.7% on average). This approach incurs an overhead of 2-5% for one particular optimization, and shows performance improvement of 17% when a combination of different optimizations is required by an application. 2) We provide an augmented Map-Reduce system (MRAP), which consist of an API and corresponding optimizations *i.e.* data restructuring and scheduling. We have demonstrated up to 33% throughput improvement in one real application (read-mapping in bioinformatics), and up to 70% in an I/O kernel of another application (halo catalogs analytics). Our scheduling scheme shows performance improvement of 18% for an I/O kernel of another application (QCD analytics).

To the best parents in the world; Siddique and Khalida,

To the most amazing sisters and brother; Farah, Urva and Usman

Acknowledgments

First of all I would like to thanks my committee members, Dr. Ronald DeMara, Dr. Mark Heinrich, and Dr. Rajeev Thakur for their valuable feedback and suggestions to improve my work. Thanks Dr. DeMara for your advice on course work and research, thanks Dr. Heinrich for being there during my first steps toward research and thanks Dr. Thakur for being a great help for my first conference publication.

Special thanks to my advisor, Dr. Jun Wang for his support all these years. I have learned so much during this time, which will definitely benefit me in my work life. All the motivation and constant encouragement kept me going and made me closer to my goals. His advice has always been very valuable. He has helped me establish work ties with the national labs collaborators and develop long-term networking, which will be beneficial in my career. I also appreciate the guidance and support (in the High Performance Computing area) from our collaborators at Argonne National Laboratory and Los Alamos National Laboratory. I am thankful to all of my Professors and the EECS staff at UCF.

This research work is supported in part by the US National Science Foundation under grants CNS-0646910, CNS-0646911, CCF-0621526, CCF-0811413, US Department of Energy Early Career Principal Investigator Award DE-FG02-07ER25747, and National Science Foundation Early Career Award 0953946.

All these years in the PhD program, we come across many people who share our work load, happy times and not so happy times. We find everlasting friendships in colleagues working in our research group and also with the ones we just share our work space with. I would like to say thanks to all of my friends and especially my lab mates Rosa, Rochelle, Grant, Christopher and Pengju for making it a pleasant journey for me. Thanks for sharing the late night work sittings and group meals. Many thanks to my dearest friend and roommate, Shafaq, for being the best roommate in the world and for keeping things more fun and memorable. I will always cherish the time I spent at UCF. I am grateful to the Women in EECS group for giving me the opportunity to work with them and their support.

I want to say special thanks to my family. I would like to express my gratitude for my parents for being so understanding and always being there for me. They have always encouraged me to strive for better. Thanks Dad for sharing the PhD passion with me, Thanks Mom for all your prayers and sweet talks to show willingness to do my work. Thanks to my sisters Farah and Urva and brother Usman for keeping everything alive for me, and for keeping me updated and sane. You guys rock!

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Current High Performance Computing (HPC) applications have seen tremendous growth in data sets in recent years. These applications are generating and processing large amounts of data ranging from terabytes (TB) to petabytes (PB). For example, state of the art simulations in astronomy generate over 200 GB per checkpoint [SWJ05]. This trend of increasing data shows that by the end of 2011, astronomers will utilize all 1.6 million CPU cores of the upcoming NCSA/IBM Blue Waters system [refb]. This will potentially generate volumes exceeding 10 PB of data per run and 10 to 200 TB per checkpoint, depending on the simulations. Similarly, the output from a single seismology simulation may be as large as 47 terabytes of data and 400,000 files. When the LSST was being designed, the detectors on the lens were simulated before manufacture. Each detector had to be able to calculate 10,000 light cones through each simulated universe. Each universe in the simulation was 30 TB of data and required heavy analysis in the LSST simulator [refd]. These applications are I/O-intensive because they read/write

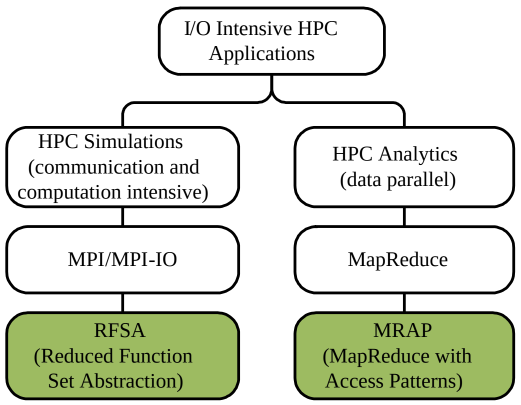

I/O Intensive HPC Applications
HPC Simulations (communication and computation intensive)
HPC Analytics (data parallel)
MPI/MPI-IO
MapReduce
RFSA (Reduced Function Set Abstraction)
MRAP (MapReduce with Access Patterns)

Figure 1.1: Research Work Overview

large data sets and also spend significant amount of time in performing I/O operations. I/O-intensive HPC applications are either simulation-based applications, which perform checkpoint writes or analytics-based applications, which process and analyze terabytes and petabytes of data.

Much attention has been given to harness the computational and storage capabilities of the supercomputing and cluster resources using various HPC parallel programming frameworks. This effort has resulted in the programming models that achieve high performance but become very complex for some applications. The complexity arises because of the fine grain control given to the application developer for translating their algorithms to parallel code and manually assigning data and task distribution e.g. in MPI (Message Passing Interface) [GTL99], [TGL99b], [ref97]. MPI is widely used for writing the simulation-based applications that are computation and communication intensive and is accompanied with parallel file systems *e.g.* PVFS2 [reft], GPFS [SH02], PanFS [WUA08], Lustre [refl], *etc* for high performance I/O operations. At the programming frameworks level, various optimizations have been proposed to improve the performance of MPI/MPI-IO function calls. For example, in MPI-IO collective I/O and data sieving are used to convert the small I/O accesses to large contiguous requests, thereby improving the performance [TGL99a].

In addition to the conventional programming abstractions, scientists are adopting many data-intensive frameworks, *e.g.* MapReduce [DG04, Dea06] to develop analytics-based HPC applications, which are data parallel. MapReduce is tightly coupled with a parallel and distributed file system *e.g.* GFS [GGL03], HDFS [Bor], *etc* and is optimized to read and process large volumes of data. Both MPI/MPI-IO and MapReduce cover a wide range of I/O-intensive applications consisting of simulation-based and analytics-based applications as we also show in the Figure 1.1.

In this dissertation work, we study the applicability of both of these programming models to a variety of I/O-intensive HPC applications. We identify several performance and programmer productivity related limitations of these existing programming models and propose ways to assist application developers without compromising performance. Figure 1.1 shows the proposed approaches as they are added to the current hierarchy of the I/O intensive HPC applications. We consider both performance and programmer productivity as we develop our approaches to build these new frameworks. We propose a **Reduced Function Set Abstraction** (RFSA) for MPI-IO, by reducing the number of I/O functions. We abstract the existing methods (*e.g.* for reading, writing, locking) via automating the selection of appropriate I/O function. Our approach improves programmer productivity without any performance penalties. Additionally, we also propose a novel MapReduce based framework to better support HPC analytics applications. MapReduce lacks direct ways to express various common HPC access patterns. Our approach, called **MapReduce with Access Patterns** (MRAP) provides extended classes, templates and corresponding optimizations to add expressiveness of HPC data access patterns in MapReduce. In the Sections 1.1 and 1.2, we introduce our approach for MPI/MPI-IO and MapReduce respectively.

1.1 Message Passing Interface

MPI/MPI-IO (in C and Fortran) is the most widely used parallel programming model. MPI not only provides the communication and computation functions but also I/O routines for transferring data to and from storage (MPI-IO). MPI-IO includes various I/O access methods in which multiple processes read/write a file collectively or independently, asynchronously or synchronously. It consists of a comprehensive set of functions with very fine-grained communication and I/O optimizations that allow programmers to work closely with the hardware for high performance. The programmers are required to know the exact usage of a communication or I/O optimization before they use it. For example MPI-IO provides blocking and non-blocking I/O operations. A blocking I/O operation

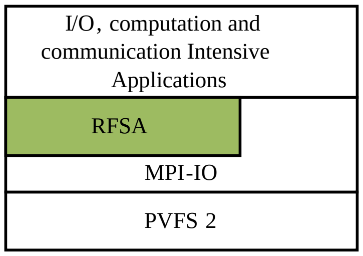


Figure 1.2: Reduced Function Set Abstraction

blocks the function call and no other functions can be performed in parallel with this I/O call. A programmer can not use these optimizations unless she/he is fully aware of the application behavior and I/O access patterns. This work is an effort to automate the selection of these optimizations instead of an application programmer deciding which specific function to use, the selection algorithm will decide appropriately. The idea is to reduce the number of language concepts and function calls without any performance penalty and is therefore named RFSA (Reduced Function Set Abstraction). We integrate our approach in the existing MPI-IO library (ROMIO) [TGL99b] as shown in the Figure 1.2. We demonstrate the RFSA concept by implementing a selection algorithm for blocking collective/non-collective reads and writes. Collective I/O operations are implemented as an optimization to the non-contiguous access patterns. It is shown in [TGL99a] that collective I/O out-performs independent I/O operations for a set of non-contiguous access patterns. We identify those cases and use this optimization without being specified by the application developer. We evaluate our design using two metrics; one for the *programmer productivity* and other for the *performance*. Programmer productivity in the parallel programming community is very challenging to quantify, and in this work we quantify it by measuring the writability of a parallel program, which depends on the number of optimizations that can be successfully hidden from the programmer. In our initial implementation, we are able to represent blocking reads and writes with a single function each.

We extend the same concept of abstracting the optimizations from application developers to atomicity semantics and the locking mechanism in MPI-IO. Atomicity semantics define the outcome of multiple concurrent I/O accesses, at least one of which is a write to a shared or overlapping region. With the advent of parallel I/O libraries, data can be accessed in various complex patterns by multiple processes. These access patterns can be contiguous or non-contiguous, overlapping or non-overlapping depending upon the application behavior. Locking mechanisms provided by the file system are used to ensure that during a concurrent I/O access, shared data is not being violated. File locks, byte range locks, list locks, and datatype locks are various locking mechanisms that are available to guarantee the atomicity semantics. Adapted from the POSIX semantics, parallel file systems such as GPFS [SH02] and Lustre [refl] provide a byte range locking mechanism. Byte range locks provide an option for guaranteeing atomicity of non-contiguous operations. By locking the entire region, changes can be made by using a read-modify-write sequence. However, this approach does not consider the actual non-contiguous access pattern that may occur in a byte range and introduces *false sharing.* This approach also limits the benefits of parallel I/O that can be gained, by unnecessarily serializing the accesses. To address these particular cases, [ACT06] [CLC07] propose to lock the exact non-contiguous regions within a byte range and maximize the concurrent I/O access.

An observation is that the locks are requested (e.g. file locks, byte-range locks, list locks, and datatype locks) even if there are no overlaps, when locks are not needed. This observation leads us to a very important question, whether it is possible to isolate the cases where atomicity is required and where it is not, and optimize the locking mechanism.

1.1.1 Contributions of RFSA

We propose a scheme to identify the conflicts at the application level, by identifying the concurrent access patterns and overlaps within an application. Our conflict detection algorithm, implemented at the MPI-IO level for independent I/O operations uses file views and datatype decoding to determine the overlaps. RFSA is presented in detail in Chapter 3. Our methods are designed to assist the application programmers by selecting appropriate I/O functions (read/write) and determining the atomicity requirements. Here, we list our main findings and the performance improvements gained by our approach.

- We have implemented a selection algorithm that transparently detects which blocking read/write function to use given a particular file view. The implementation is provided as a unified read/write function in MPI-IO. Our new functions `RFSA_READ` and `RFSA_WRITE` provide an alternative to 6 `MPI_File_read_xxx` and 6 `MPI_File_write_xxx` functions respectively, and show an improved programmer productivity by 35.7% on average for three different benchmarks [SW10].

- Our experiments with three MPI-IO bencmarks show that RFSA READ/WRITE provides comparable performance benefits for different read- /write scenarios by incuring an overhead of 2-5%, but showing an improvement of up to 17% for the benchmarks with write requests requiring a combination of optimization functions instead of one function [SW09, SW10].

- We have implemented a conflict detection algorithm that transparently identifies the cases where atomicity is required and thereby acquires and releases locks in MPI-IO. This functionality is provided as a part of file view function that is called by all processes before reading/writing. If there is a conflict detected, MPI_File_set_atomicity is used to ensure the atomicity semantics. Our results show that with our conflict detection algorithm performance improves by up to 59% if byte range locks are used, and up to 72% if file locks are used and are requested unnecessarily [SWT09]. The overhead of our conflict detection algorithm shows a performance penalty of up to 3.6%.

1.2 MapReduce

HPC architectures consist of a compute cluster/supercomputer for performing complex computations [refx, refw] and a storage cluster with parallel file systems [WUA08, SH02] to store the results of these computations. Additionally, the storage cluster is also used to store the data sets from other scientific equipments, for example telescopes, sensors, etc. Several types of analysis operations are performed on these large data sets generated from various sources.

Some of these applications access data in large chunks, matching well with the data processing framework of Hadoop [reff], and MapReduce [DG04, Dea06]. However, there are some other applications that access data in various patterns, with the strided access pattern being the common scientific application pattern [BGG09]. These patterns are derived by the nature of applications that generate the data sets, and the way this data is laid out in the file system. For example, there can be a two-dimensional data set, which is stored in row major order in the file system. Any column major access to this data set will result in a strided access pattern. Similarly, there are some other applications, like template matching on a large image data set that requires a tiled access pattern. Adapting data models like MapReduce, which are capable of processing large I/O requests for HPC data analytics, suffer from limited performance because of various access patterns. These various access patterns may result in small non-contiguous

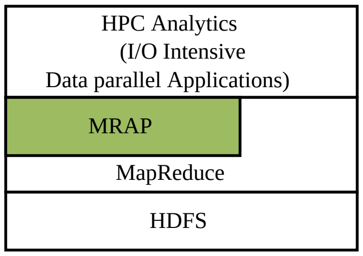


Figure 1.3: MapReduce with Access Patterns

I/Os, which negatively impacts the performance due to excessive disk seeks resulting in underutilized bandwidth.

If the access pattern is known in advance, then data can be laid out according to the way it will be accessed. Hence, it is possible to convert the non-contiguous (strided) access pattern to contiguous accesses. It can be achieved by storing all the contiguous stripes that are separated by strides all together. We provide a framework called MRAP - MapReduce with Access Patterns that consist of user API and templates to specify new file access patterns, particularly non-contiguous strided accesses. Providing the support for access patterns to the scientists for writing MapReduce applications is an attractive approach because scientists are already familiar with these patterns. Given an easy-to-use API and templates (that are configurable using a configuration file), they will be able to write their data-intensive HPC applications. Since the API and templates are built using MapReduce and HDFS as shown in the Figure 1.3, they are expected to have performance penalties because HDFS does not perform well for non-contiguous read operations (small I/O).

To mitigate this small I/O performance lag especially during an application run, we use a concept called *data restructuring*. It converts non-contiguous access patterns to contiguous accesses according to a specified access pattern to achieve high I/O performance. Data restructuring can be performed at two levels, i.e. when data is initially copied to the distributed file system (HDFS) using command *copyFromLocal*, or before an application is run. In either case, a user specified configuration file is used to describe

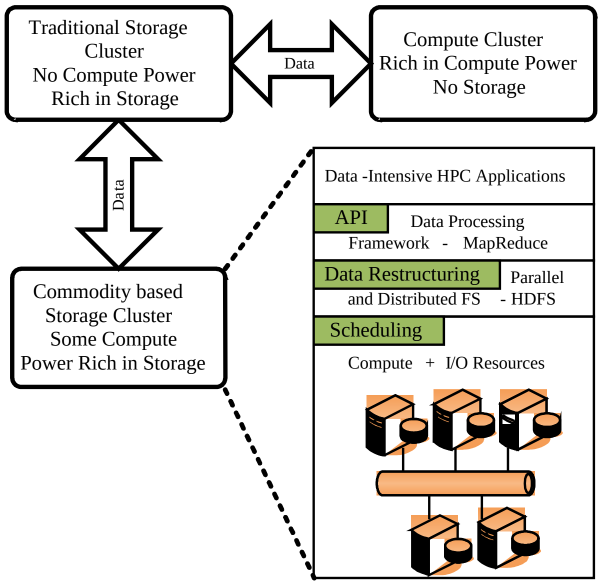

Traditional Storage Cluster No Compute Power Rich in Storage
Data
Compute Cluster Rich in Compute Power No Storage
Data
Commodity based Storage Cluster Some Compute Power Rich in Storage
Data -Intensive HPC Applications
API
Data Processing Framework - MapReduce
Data Restructuring
Parallel and Distributed FS - HDFS
Scheduling
Compute + I/O Resources

Figure 1.4: System Architecture with MRAP, and it's components

the access pattern. Our preliminary results for the implementation of latter level show the performance gain of 70% when data is restructured. This performance gain is with the assumption that the restructured file does not have to be restructured again when a new application is run on the same data set. These small I/O requests result in mapping to multiple DFS chunks and make it challenging to schedule map tasks (multiple DFS chunks are assigned to a map task). We propose an efficient scheduling mechanism to deal with these cases when access patterns result in mapping to multiple DFS chunks, and these chunks have to be processed in a group. In Figure 1.4, we show the overall architecture of the system where our framework, MRAP and its optimizations, will be deployed. Current HPC systems consist of a compute cluster and a storage cluster. It also consists of a setup to support offline HPC data analytics using commodity based cluster. The cluster runs a data parallel framework e.g. MapReduce and Hadoop Distributed File System with our MRAP framework. MRAP framework supports various HPC data access patterns and improves the performance of user applications by using data restructuring and scheduling.

1.2.1 Contributions of MRAP

Our approach reduces the overhead of writing multiple MapReduce programs to pre-process data before its analysis. The main contributions of our work are as follows:

- We provide extended classes and templates to specify the sequence matching and strided (non-contiguous) access patterns for reading HPC data sets, such that access patterns are directly specified in the map phase. We ran a real application from bioinformatics and an astrophysics I/O kernel by using MRAP API and templates respectively. Our results show a maximum throughput improvement up to 33% [SMW10].

- HDFS is not designed to perform well for small I/O requests, hence we proposed to use a scheme called **data restructuring** to improve the performance. When a file is stored on HDFS, it is stored according to the specified access pattern i.e. restructured. We also studied the performance penalties due to the non-contiguous accesses (small I/O requests) and implemented data restructuring to improve the performance. Data restructuring uses a user-defined configuration file and reorganizes data in a file such that all non-contiguous chunks are stored contiguously, and show a performance gain of up to 70% for the aforementioned applications [SMW10].

- We implemented an improved scheduling scheme which selects an optimal node for scheduling a map task which requires multiple chunks. We either create virtual splits or use a weighted set cover approach to minimize the remote I/O requests. Our initial results show a performance gain of 18% for data sets up to 85 GB distributed on a 45 node cluster setup.

CHAPTER 2
BACKGROUND

In this chapter, we will briefly describe the relevant concepts in MPI/MPI-IO *e.g.* datatypes, file views, collective I/O, *etc.* We will also discuss how MapReduce based framework works, various classes for writing MapReduce program and how they work with distributed file system.

2.1 MPI/MPI-IO Background

2.1.1 MPI File views and datatypes

MPI_datatypes are of two types, one is basic datatype *e.g.* MPI_INT, MPI_BYTE, *etc.* and the other is derived datatype *e.g.* MPI_TYPE_VECTOR, MPI_TYPE_INDEXED, MPI_TYPE_SUBARRAY, *etc.*

File views are used to describe the logical layout of a file. It is a collective operation called before a read/write call. File views are created by using either basic or derived datatypes. The functions provided by MPI-IO to create and retrieve file views are `MPI_File_set_view` and `MPI_File_get_view` respectively. When setting a file view following parameters are used, the start of the view is set to MPI_OFFSET disp; the type of data is set to MPI_datatype etype; the distribution of data to processes is set to MPI_datatype filetype; and the representation of data in the file is set to datarep. We use file views to retrieve the data access pattern to determine the overlapping regions for optimizing MPI_File_set_atomicity operation.

2.1.2 MPI I/O optimizations

I/O optimizations which are most commonly referred in this work are **data sieving** and **collective I/O**. These optimizations are used to improve the performance of small I/O accesses. Numerous studies of the I/O characteristics of parallel applications have shown that many applications need to access a large number of small, noncontiguous pieces of data from a file. For good I/O performance, however, the size of an I/O request must be large (In the order of megabytes). The I/O performance suffers considerably if applications access data by making many small I/O requests. To reduce the effect of high I/O latency, it is critical to make as few requests to the file system as possible. Instead of accessing each contiguous portion of the data separately, a single contiguous chunk of data starting from the first requested byte up to the last requested byte is read into a temporary buffer in memory in data sieving. Similarly, collective I/O also allows a process to read a large contiguous chunk of data but then using MPI's communication framework, it redistributes the data among multiple processes as required by them [TGL99a]. These two techniques convert multiple non-contiguous small I/O accesses into fewer large contiguous I/O requests.

2.1.3 MPI Atomicity

MPI-IO allows the programmer to define complex datatypes, hence creating complex file views and access patterns. Since ROMIO implementation uses I/O APIs supported by the file system, the guarantee of atomic mode indirectly relies on the file system. Some file systems support POSIX semantics that work for contiguous access patterns but have no control over non-contiguous access patterns. ROMIO uses fcntl locks for non-contiguous access, and locks the range of the bytes encompassing the non-contiguous access patterns. This approach serializes the accesses if there is no *true overlapping* of patterns. To deal with this case, list locking and datatype locking is used that provides lock to the exact non-contiguous regions with in a range. But if there is no overlapping at all even in non-contiguous access, locking can be eliminated. Other file systems like PVFS and PVFS do not provide atomicity and rely on the I/O library layer to provide these guarantees.

I/O atomicity is referred to as the outcome of the concurrent I/O access to overlapping regions, both in the file and process's memory. Atomicity semantics require that the outcome should be defined by only one of the processes participating in the concurrent write operation. MPI-IO provides a function, MPI_File_set_atomicity to guarantee atomic I/O operations. This function uses lock implementations provided by the parallel file systems e.g. ROMIO implementation uses fcntl locks (byte range locks) where provided by file systems, or does not support atomic mode for parallel file systems like PVFS and PVFS2. In short, MPI-IO 's atomicity semantics define the results of concurrent requests (atleast one is write request) to overlapping regions by multiple processes.

2.2 MapReduce Background

MapReduce framework provides a large-scale data processing engine tightly coupled with a distributed file system. In this section, we will briefly describe how a MapReduce program works, and how it interacts with the file system. We use an open source implementation of MapReduce and distributed filesystem *i.e.* Hadoop [reff]. Its architecture is shown in the Figure 2.1. Hadoop MapReduce is a software framework for writing data parallel applications that process terabytes of data on large clusters of commodity hardware. It consists of two major components, 1) a metadata server for the distributed file system called as *Name node*, 2) an execution engine to run MapReduce jobs called as *JobTracker*. Each node in the cluster runs a *Data node* and *task tracker* to serve individual map/reduce tasks. The main features offered by Hadoop framework are scalability,

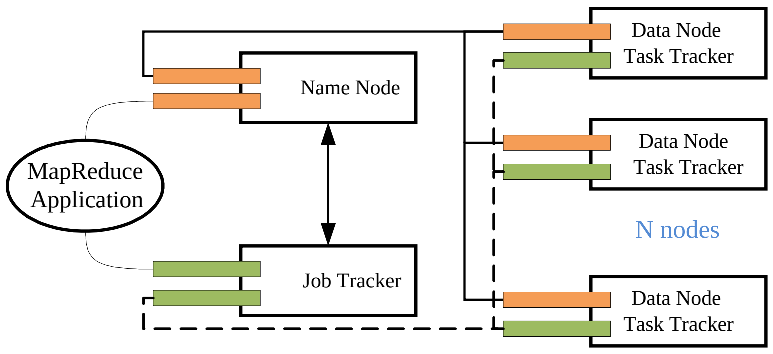


Figure 2.1: Hadoop Architecture [reff].

resiliency, reliability and high performance read operations due to software replication and data locality exploitation. Typically the compute nodes and the storage nodes are the same, that is, the Map/Reduce framework and the Hadoop Distributed File System are running on the same set of nodes. This configuration allows the framework to effectively schedule tasks on the nodes where data is already present, resulting in very high aggregate bandwidth across the cluster [reff].

2.2.1 MapReduce Programming Overview

Each MapReduce program consists of two phases as shown in Figure 2.2; a Map phase and a Reduce phase. During the map operation, one data parallel operation is performed and results are collected at the intermediate combine phase; and then another operation, reduce, is performed before the output data becomes persistent storage. The MapReduce framework works exclusively on <key,value> pairs. The map operation is expecting an input of <key,value> and subsequently outputs a set of <key,value> pairs for the reduce phase of the operation [reff]. From Figure 2.2 it is shown that all map and reduce operations are tasks run on the *tasktrackers* in the Hadoop cluster. These individual map and reduce tasks are monitored from inception to completion by the *jobtracker*. During the combine phase of the map reduce operation, intermediate output data from

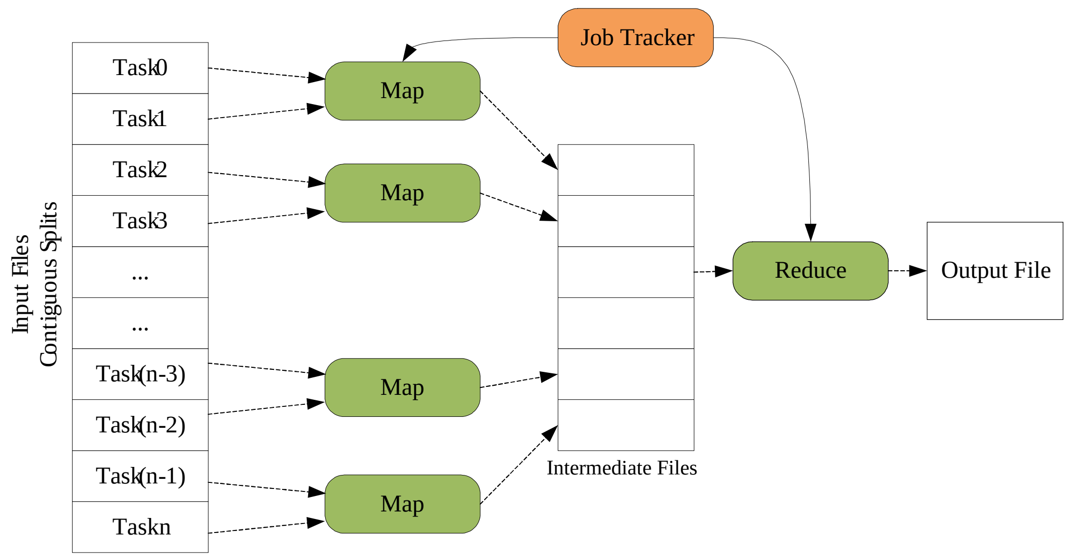


Figure 2.2: Flow of MapReduce operations [DG04].

all map tasks on an individual *tasktracker* is written to local storage for the reduce phase. This is a high-level view of the steps involved in a map reduce operation. There is no interprocess communication between any map task during the map phase, and likewise no communication between reducers. These two operations, map and reduce, allow for a large parallel dataset to be operated upon very quickly with the assurance of task resiliency.

2.2.2 Parallel and Distributed FileSystem - HDFS

HDFS consists of a metadata server, i.e. a Name node and a set of data nodes [reff]. It is based very closely on the Google file system [GGL03]. HDFS uses a scheme of three-way software replication to ensure that the files stored are always intact in three separate places across a Hadoop cluster. For example, two out of three replicas are stored on the same rack and third replica is stored on a different rack. This software replication approach allows for Hadoop to guarantee system resiliency. Figure 2.3 shows the conceptual model of HDFS. As it can be seen from the figure that a client application

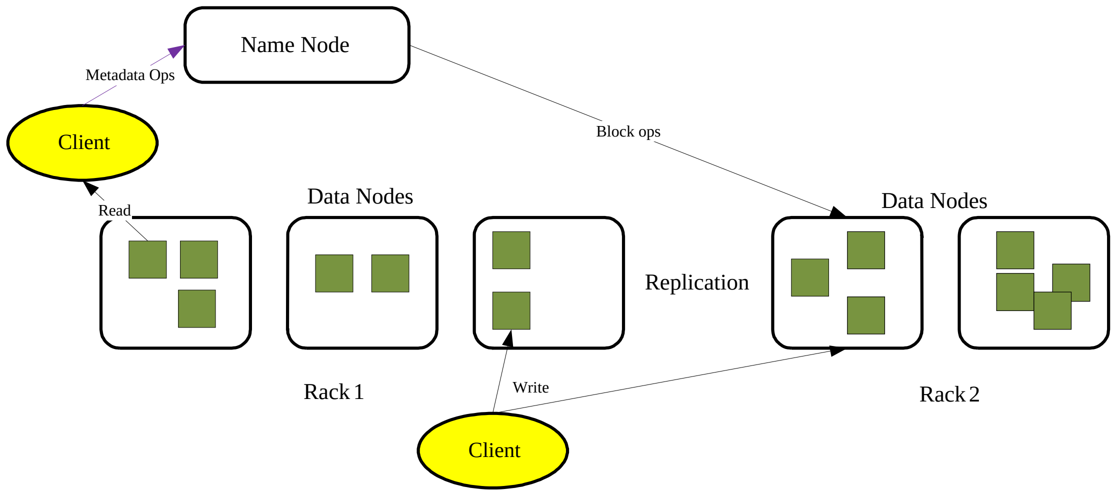


Figure 2.3: Hadoop Distributed File system (HDFS) [Bor].

is accessing the file system. It first directs the file queries to the *Namenode*, the *namenode* then directs the file request to the appropriate *datanode(s)* and the *datanode(s)* supply the client application with the data. Figure 2.3 also shows the replication of the file across servers in a rack and across server racks. When file chunks are written to *datanodes* across the HDFS, *namenode* tries to group at least one replicated chunk on the same server rack as the primary and then another chunk to an adjacent rack of *datanodes*, while also ensuring that no two replications of a chunk are stored to the same *datanode*. In the event of hardware failure of a server, the *namenode* takes an active role in re-establishing the health of the cluster without need for intervention by the user [reff]. This feature makes HDFS a suitable candidate for large-scale data processing applications.

CHAPTER 3

RFSA - AN MPI-IO BASED REDUCED FUNCTION SET ABSTRACTION TO SUPPORT I/O-INTENSIVE HPC SIMULATIONS

In this chapter, we provide methods to reduce the number of functions and abstract the corresponding performance optimizations in MPI-IO. The motivation is that if we keep the number of functions in MPI-IO closer to the number of I/O functions in any sequential language e.g. C/C++, we can achieve better programmer productivity and improved performance. These sequential languages provide four distinct functions for file I/O operations i.e. open, close, read and write. On the other hand, MPI-IO has 12 different flavors only for read operations. The programmer productivity is directly related to the writability of a programming language, which is defined by the number of functions in a programming language [Seb02].

A programming language is evaluated on the basis of following measures; readability, writability, reliability and cost [Seb02]. Programmer productivity is directly related to the writability of a programming language, as it is a measure of how easily a language can be used to create programs for a chosen problem domain. Furthermore, writability is driven by the simplicity and orthogonality, support for abstraction and expressivity. In the case of parallel programming languages, there is no direct way to measure simplicity, orthogonality and expressivity. The "support for abstraction" means the ability to define and use complicated operations by hiding the details. We can indirectly measure support for abstraction by using the number of function calls and the number of concepts concealed in those function calls.

Our approach reduces the number of functions by abstracting the performance optimization functions without impacting performance. Hence, instead of giving the programmers the ability to use any optimization, the proposed methods select the appropriate optimization. Our approach also avoids any performance penalties due to inappropriate use of these functions. In short, we propose methods to automate the decision of when to use MPI_File_write, MPI_File_write_all, MPI_File_set_atomicity, etc.

We represent MPI-IO as consisting of m different groups of function calls, where each group represents a specific operation like a open/close, read/write, etc. Each group has up to g number of options, some of them may be performance optimizations. In the following notation, a function call is represented as $F_1\_c_1$ that means a function F and option c. F_1 defines a group of functions e.g. read, write, etc and c_* specifies the choices, e.g. positioning, coordination, synchronization, etc. p_x represents the parameter list of a particular function call. Formally, MPI-IO is represented as follows:

$$MPI-IO : \begin{cases} F_1\_c_1(p_{11}, p_{12}, ..., p_{1j_1}) \\ F_1\_c_2(p_{21}, p_{22}, ..., p_{2j_2}) \\ \vdots \\ F_m\_c_1(p_{m1}, p_{m2}, ..., p_{mj_m}) \\ \vdots \\ F_m\_c_g(p_{m1}, p_{m2}, ..., p_{mj_m}) \end{cases}$$

Now, in RFSA we take MPI-IO and provide a list of functions that wrap up the comprehensive set of MPI-IO function calls and hide the optimizations from the application programmer. Hence, some of the optimizations can be utilized transparently to the application programmer. For example, in MPI-IO there are 6 different blocking write functions but only one of them is used to perform a write at any time. These blocking functions are differentiated on the basis of their positioning and coordination as shown in the Table 3.1. Essentially, we provide a select procedure that makes the choice of a specific function on behalf of the programmer.

Table 3.1: Characteristics of Blocking Read/write

Positioning	**Coordination**
Explicit Offset*(e)*	Collective*(c)*
Individual File Pointer*(i)*	Non-collective*(nc)*
Shared File Pointer*(s)*	

We take a group of functions covering different concepts and categorize them as one function RF_1 in RFSA. RF_1 will correspond to a group of functions, e.g. in MPI-IO blocking reads are encapsulated in one RFSA function RFSA_read.

$$RFSA : \begin{cases} RF_1(p_{11}, p_{12}, ..., p_{1k_1}) \\ RF_2(p_{21}, p_{22}, ..., p_{2k_2}) \\ \vdots \\ RF_n(p_{n1}, p_{n2}, ..., p_{nk_n}) \end{cases}$$

where $m >> n$ and normally $j_i <= k_i, (i = 1, 2, \cdots)$.

An important point to note here is that only the number of function calls within a group is reduced, not the number of groups. For example read/write are also a part of RFSA, but their 6 different options are represented as one function. The number of parameters either remains the same or increases depending on the function type.

3.1 RFSA Read/Write Operations

MPI-IO provides a comprehensive set of read/write functions with performance optimizations like collective, independent, blocking and non-blocking reads/writes. Only one read/write function is used to serve the required read/write request, hence we provide a selection procedure to make this choice transparent to the application programmer. In many parallel applications, each process may need to access several non-contiguous portions of a file, the requests of different processes are often interleaved and may together span large contiguous portions of the file. If the user provides the MPI-IO implementation with the entire access information of a group of processes, the implementation can improve I/O performance significantly by merging the requests of different processes and serving the merged request. Such optimization is broadly referred to as collective I/O [TGL99a]. However, it is left to the application developer to decide whether or not collective I/O is required. Options like using these optimizations make it extremely difficult to write an effective MPI/MPI-IO program. Our RFSA unified function automatically decides whether or not collective I/O is required. This method comes with an overhead of detecting the overall byte range being accessed by each participating process and is already implemented in the current implementation of collective read/write.

We use the contiguous/non-contiguous access pattern and the percentage of overlapping byte range provided as the selection criteria. *Byte range is defined as the region starting from the first file offset/process to the end offset/process.*, where end offset is the last byte accessed by a process i.e. last $(offset + blocklength)$. If the byte ranges of all processes depict a significant overlapping region with interleaving patterns, then the task can be divided among all of them for collective I/O. Otherwise individual I/O operations are used. The following steps are performed to make a decision.

1. Check if the file access pattern is non-contiguous.

2. Calculate the list of offsets and lengths in the file and determine the start and end offsets.

3. Communicate the start and end offsets to other processes. This enables each process to reach the same decision of either collective/non-collective.

4. Check if the accesses of different processes are overlapping. In case of overlapping byte ranges, it is useful to perform a collective I/O operation. Assuming $i-1$ and i to be the ranks of two consecutive processes, a simple byte range overlap check would be if ($start\_offset(i) < end\_offset(i-1)$), that means the accesses of these two processes lie in the same range.

We calculate the percentage of byte overlap between two consecutive processes to make sure that there is a significant overlap, the communication is minimal because each process uses the information of its neighboring processes. The condition to select the collective I/O is based on the non-contiguous access pattern in the overlapping byte range among all participating processes as shown in the algorithm 3.1.1. In algorithm 3.1.1, $OP(i, b, c)$ corresponds to a **c**ollective, **b**locking operation with **i**ndividual file pointers, and OP can be a read/write function. Similarly, **s** in the function corresponds to a **s**hared file pointer, and **e** represents an **e**xplicit offset. **nc** corresponds to a **n**on-**c**ollective I/O operation. It is also shown in the algorithm 3.1.1 how positioning options are used in the selection. If the offset parameter is not specified, it is assigned the value -1 and it follows either the individual or shared file pointer option. The individual file pointer routines have the same semantics as the data access with explicit offset routines except that offset is defined to be the individual file pointer maintained by MPI. Individual/shared file pointers are specified at the time file is opened. The routines with shared file pointers only use and update the shared file pointer maintained by MPI and individual file pointers are not used nor updated [ref97].

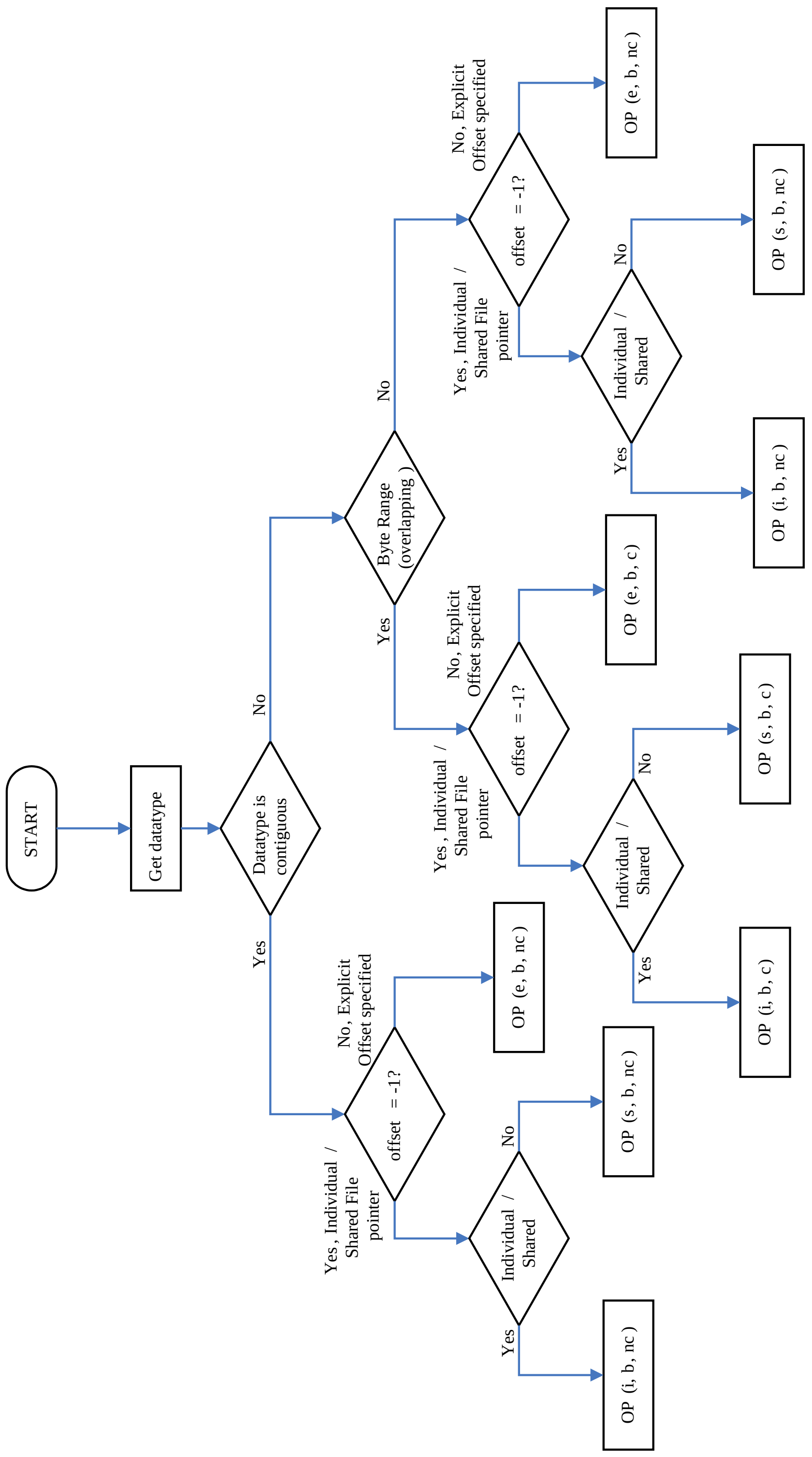


Figure 3.1: Flow Chart for Selection Algorithm

Algorithm 3.1.1 Selection Algorithm for Read/Write coordination

```
1: if Data access pattern is not contiguous (i.e. MPI_Datatype_iscontig) then
2:     if Byte ranges of participating processes overlap then
3:         if offset is -1 then
4:             if file pointer is individual then
5:                 return $OP(i, b, c)$
6:             else
7:                 return $OP(s, b, c)$
8:             end if
9:         else
10:            return $OP(e, b, c)$
11:        end if
12:    else
13:        if offset is -1 then
14:            if file pointer is individual then
15:                return $OP(i, b, nc)$
16:            else
17:                return $OP(s, b, nc)$
18:            end if
19:        else
20:            return $OP(e, b, nc)$
21:        end if
22:    end if
23: end if
```

It should be noted that it is impossible to provide a generic selection algorithm considering the complexity of non-blocking operations. Therefore, in our initial design we focus on blocking collective/independent operations. Also, if we want to leverage the same idea to MPI communication functions, a selection algorithm depending on number of source and destination processes should be developed. The main idea presented in this paper is to abstract the functions from the programmer that are performance optimizations.

3.1.1 Merge Some Functions

In this section, we briefly discuss some other functions that are not performance optimizations, but are potential candidates for inclusion in RFSA. We can merge some functions, and reduce the number of functions in the original set. For example, MPI_File_open is an explicit call to open a file, and has some additional checks to open file as read-only, write-only, etc. MPI_File_open is made a part of MPI_initialize, but this approach loses the flexibility of MPI_File_open, and it does not affect performance. There is no universal mode to set while opening a file. Similarly MPI_File_close is made a part of MPI_Finalize. Additional parameters for file name and file handler are required to open and close a file, respectively. The argument for programmer productivity in both approaches is that programmer will not have to worry about explicit open and close functions.

MPI-IO has a rich set of function calls to create data types, commit them and use in the file views as shown in Listing 1. We abstract the explicit commit function from the programmer and hide it inside MPI_Type_vector and other different data types. This approach also reduces number of function.

Listing 1 MPI-IO: Setting File view

```
//Create a datatype .e.g. vector in this example
MPI_Type_vector(count, array_of_blocklengths,
array_of_displacements, etype, newtype);

//Commit the new datatype
MPI_Type_commit(&newtype);

//Set the file view
MPI_File_set_view(fh, disp, etype, newtype,
"native", MPI_INFO_NULL);
```

3.2 RFSA Atomicity Operation

`MPI_File_set_atomicity` is used to ensure the atomicity semantics in the MPI-IO library. The locking mechanisms provided by the underlying file system are used to guarantee the atomicity specified by this function. The overhead of the locking mechanism appears in three forms; first is the communication overhead that is generated while acquiring and releasing the locks (sending the requests to lock server(s)), second is the storage space overhead that is caused by storing the locks during their acquire and release time (a data

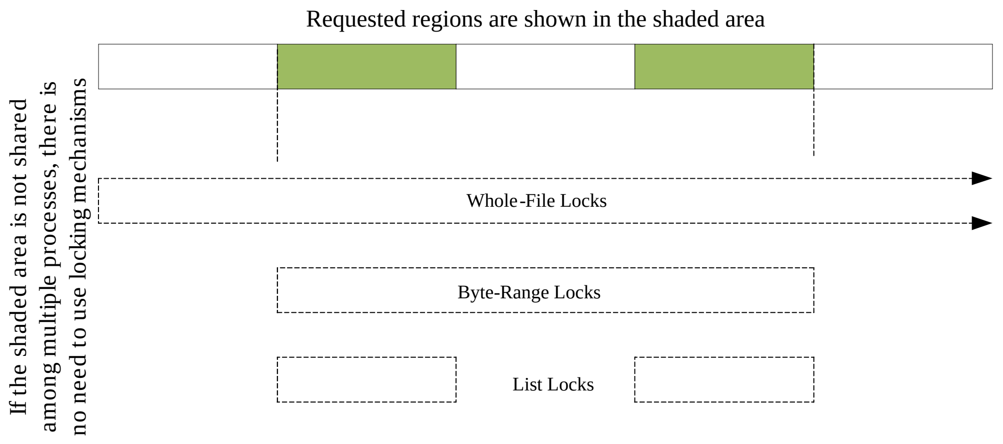

Requested regions are shown in the shaded area
If the shaded area is not shared among multiple processes, there is no need to use locking mechanisms
Whole-File Locks
Byte-Range Locks
List Locks

Figure 3.2: Different Locking mechanisms can be performed depending on the outcome of Conflict Detection.

structure is maintained to store the locks, and for fine-grained locks like list locks, this structure can grow very large.), and the third is computation overhead to assign new locks (i.e., the tree data structure is scanned to check if the same lock request is being held by a different process). These overheads can be reduced by making sure that unnecessary locking requests are not generated. An observation is that the locks are requested by using `MPI_File_set_atomicity` (e.g. file locks, byte-range locks, list locks and datatype locks) even if there are no overlaps and locks are not needed. This observation leads us to a very important question, whether it is possible to isolate the cases where atomicity is required and where it is not, and optimize the locking mechanism.

We propose a conflict detection algorithm that should be performed before any locking mechanism as shown in the Figure 3.2. Our goal is to optimize the lock acquiring process by providing an efficient conflict detection algorithm beforehand to identify the overlapping regions, thereby requesting the locks only if there are overlapping regions. The conflict detection algorithm presented in this paper is based on MPI datatypes and file views. Typically, a file read/write request in any MPI-IO program consists of following steps: 1) Create the Data types, 2) Create the File views, and 3) Read/Write Request. The conflict detection is performed when a file view is created using (`MPI_File_set_view`). Since it is a collective call, each process can exchange their file views and determine the overlapping regions by comparing offset/block length pairs. Each node acts as a conflict detector for itself.

The file view is created using `MPI_File_set_view`, and then each node decodes the supplied MPI datatype. Decoding a datatype is not straightforward and is a two step procedure using MPI-IO functions. The first step is getting the envelope of the datatype using `MPI_Type_get_envelope` which returns information such as number of displacements and block lengths used to create the datatype. The second step is getting the actual contents in the form of offset/block length using `MPI_Type_get_contents`. The decoded datatypes are exchanged using collective communication functions. The overhead of conflict detection is based on the complexity and size of the datatype. For some datatypes this overhead is as small as exchanging two long integer values, while for others it can consist of a long list of offset/length pairs.

We categorize the derived datatypes in to two broader categories based on their structure. The first category is a *regular datatype*; all the processes have the same block size but a different displacement e.g. `MPI_Type_vector`, `MPI_Type_subarray`. In a subarray, each process accesses a subarray that is defined by the number of dimensions and the starting and ending offsets in each dimension. For a regular datatype, we need to exchange the start and end offset in each dimension for each process. The second category is *irregular datatype*; all the processes may access different block sizes, different patterns and different displacements, e.g. `MPI_Type_hindexed`, where each process accesses a non-contiguous region defined by a list of offsets and corresponding block lengths.

Regular Datatypes: In independent write operations, when there are overlaps among different writes, only one process should perform the write at a time. For regular datatypes, we take the example of `MPI_Type_subarray`. A subarray datatype is defined by the number of dimensions, size of array in each dimension, sizes of subarrays in each dimension and the start positions for each subarray. The start of each subarray and the size in each dimension is used to identify the overlaps between any two consecutive tiles or subarrays as shown in the following equations. A conflicting region is specified by $CR(CO, CL)$, where CO is the conflicting offset and CL is the conflicting length. The displacement of the i^{th} process is specified by $disp_i(x, y)$ for a two-dimensional subarray and the corresponding block length is given by $blklen_i(x, y)$.

$$CO = max(disp_i(x, y), disp_j(x, y)) \tag{3.1}$$

If both the displacements i.e. $disp_i$ and $disp_j$ are the same, the CL is given by eq. 3.2, otherwise eq. 3.3 is used.

$$CL = min(blklen_i(x, y), blklen_j(x, y)) \tag{3.2}$$

$$\begin{aligned} CL = [&min(disp_i(x, y) + blklen_i(x, y), \\ &disp_j(x, y) + blklen_j(x, y))] - CO \end{aligned} \tag{3.3}$$

Irregular Data types: For irregular datatypes, the offset/length pairs are required because each non-contiguous region will be of a different size. Each process compares its own offset/length list against the others. A conflicting region $CR(CO, CL)$ is defined by the following equations, where CO is the starting offset, and CL is the length of the conflicting region. *disp(i)* is the displacement of the i^{th} process and *blklen(i)* is the corresponding block length.

$$CO = max(disp(i), disp(j)) \tag{3.4}$$

If both the displacements i.e. $disp(i)$ and $disp(j)$ are the same, the CL is given by eq. 3.5, otherwise eq 3.6 is used.

$$CL = min(blklen(i), blklen(j)) \tag{3.5}$$

$$\begin{aligned} CL = [&min(disp(i) + blklen(i), \\ &disp(j) + blklen(j))] - CO \end{aligned} \tag{3.6}$$

Since, the exact displacement and block length values are used, *false sharing* is eliminated completely. There are more complex datatypes that we categorize as multi-level datatypes, and MPI facilitates the creation of nested datatypes. For example in *non-contig* benchmark, `MPI_Type_contig`, `MPI_Type_vector` and `MPI_Type_struct` are used to create file views. In such cases, we perform multi-level decoding to determine the conflicts. The overhead incurred by conflict detection is evaluated in Section 3.4 in terms of communication and computation time. The communication overhead is determined by the collective communication calls to exchange datatypes. The computation overhead includes the time to generate and compare the datatypes. It depends on the datatype or the size of the list to be compared. For each process, if the size of the list is N, it will perform a linear compare of order N. The space required is equal to the size of the datatype or the offset/length list.

3.3 Implementation

We have implemented all the functions using the ROMIO [TGL99b] library, and provide these interfaces to the application developers. Table 3.2 summarizes these functions. The implementation details are described in the following subsections.

3.3.1 RFSA_File_Read/Write

RFSA_READ covers six blocking functions with combinations of offsets and collective/non-collective options as shown in the Table 3.2. The interface provided to the application developer is the same as a typical read_at function because it includes the same parameter list as read, read_all, etc but with an additional parameter i.e. an *offset*. If the offset is specified, it is treated as a explicit offset option for read/write, otherwise either individual or shared file pointer options are selected on the basis of file handler properties.

The decision between collective read and independent read is made on the basis of non-contiguous access pattern and the overlapping byte range, file pointer options do not make any difference to the decision of collective and non-collective operation. The I/O selection function is hidden in the implementation. The MPI-IO functions used in the selection algorithm implementation are `MPI_Datatype_iscontig` to determine the contiguous/non-contiguous access, and an existing byte range function to find the start and end offset pairs for each process. In case there is a non-contiguous access per process within same byte range, the set of processes can coordinate to perform a collective I/O operation. In that case, developer will not have to specify a collective function denoted by `_all` explicitly. `RFSA_WRITE` covers the corresponding write functions as `RFSA_READ` and maintains the same interface as `write_at`.

Table 3.2: Functions in MPI-IO and RFSA

MPI-IO Functions	RFSA
MPI_File_read MPI_File_read_all MPI_File_read_at MPI_File_read_at_all MPI_File_read_shared MPI_File_read_ordered	RFSA_read
MPI_File_write MPI_File_write_all MPI_File_write_at MPI_File_write_at_all MPI_File_write_shared MPI_File_write_ordered	RFSA_write

```
int RFSA_READ/WRITE (MPI_File fh, void *buf, int count, MPI_Offset offset,
          MPI_Datatype ftype, MPI_Status *status)
```

3.3.2 RFSA_File_set_view

We implement the self-detecting locking mechanism in ROMIO, by adding it to `MPI_File_set_view` as shown in the listing 2. We call the new function implementation `RFSA_File_set_view`. Listing 2 also shows how to use the conflict variable in the main program. The decoding process utilizes two function from MPI-IO library; `MPI_Type_get_envelope` and `MPI_Type_get_contents`. Our initial implementation provides conflict detection support for a few selected data types, `MPI_Type_vector, MPI_Type_subarray, MPI_Type_hindexed`. Each process exchanges the view information using the `MPI_Allgather` collective communication function. Once the data is ready at each node, it performs the comparison for the conflicts. Our current implementation is tested with PVFS2, which does not support locking. We have used the algorithms presented in [RLG05] for file locks and [TRL05] [PGK06] for byte range locks implementations with PVFS2. For byte range locks, we determine the start and end offsets of the byte range accessed by each process using an existing function in ROMIO i.e. `ADIOI_Calc_my_off_len`; it returns the start and end offsets used by `BR_Lock_acquire(br_lock, ..)`.

MPI-IO write functions can be performed collectively or independently. The collective write operations do not require conflict detection because conflicts cannot occur in collective operation. The independent write operations do not communicate with each other to optimize the non-contiguous access and require locking to protect the shared data regions. Our conflict detection implementation can be used with any locking mechanism and independent write function.

Listing 2 Pseudocode for Conflict Detection

```
//Pseudocode for Conflict Detection
int Conflict_Detection(MPI_Datatype ftype) {
//Get the datatype envelope,
MPI_Type_get_envelope(ftype, &num_ints, &num_adds, &num_dtypes, &combiner);

//Get the actual contents of the datatype
MPI_Type_get_contents(ftype, num_ints, num_adds, num_dtypes,
array_of_ints, array_of_adds, array_of_dtypes);

//Gather datatypes from all other nodes
MPI_Allgather(array_of_ints, num_ints, MPI_INT, ai_all,
num_ints, MPI_INT, MPI_COMM_WORLD);
...
//Compare datatypes
switch(combiner){
case MPI_COMBINER_SUBARRAY:
// Compare the elements of ai_all, that contains array of
   starts, and the block lengths are same for all blocks!
break;
case MPI_COMBINER_INDEXED:
case MPI_COMBINER_HINDEXED:
break;
...
}
return conflict;
}
```

3.4 Evaluating the Functions

In this section we will evaluate RFSA functions by quantifying programmer productivity and performance. The main objective of RFSA is to improve the programmer productivity of MPI-IO programmers without compromising the performance. The former is directly related to the complexity offered by a programming language. We discuss both the programmer productivity and the performance of MPI programs in the following subsections.

3.4.1 Programmer Productivity

Performance has always been the ultimate goal in HPC, and programmer productivity has always been overlooked. Conceptual programming effort and empirical data analysis have been used to measure programmer productivity as in [HCS05], [CYZ04], [PG08]. Programmer Productivity is a measure of ratio of output to input i.e. the rate at which required functionality is implemented. In this paper, we use lines of code as an output and the programmer's effort required to write a parallel program as the input. Programmer's effort is quantified as the number of functions a programmer has to consider before implementing a particular functionality. We decided to use this metric because all the benchmarks used in this paper have already been developed, and there is no record of person-hour/person-month efforts on these benchmarks.

We use a tool **SourceMonitor** to determine the lines of code and the number of distinct functions for MPI-IO and RFSA implementations for three different benchmarks. The purpose of using the number of distinct functions is to determine the number functions a programmer has to look at before implementing a particular functionality. For example, if there are 100 lines of code, 8 distinct functions, one for initialization, open, rank, size, read, write, close, finalize, then the programmer's effort using MPI-IO will be 18. And the programmer productivity will be 5.55. Programmer's effort using RFSA will be 8, and the programmer productivity will be 12.5. Using three different MPI-IO benchmarks, we determine the programmer productivity for MPI-IO and RFSA implementations. The results are presented in Table 3.3. It is shown that programmer productivity by RFSA is 35.7% better than MPI-IO on average for the three tested benchmarks.

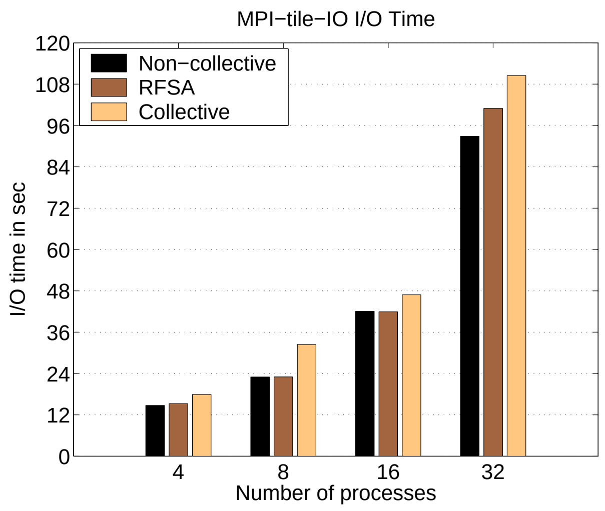
Table 3.3: Programmer Productivity: MPI-IO Versus RFSA

Benchmark	**Output**	**Input** MPI-IO	RFSA	**Productivity** MPI-IO	RFSA
Noncontig	1377	21	11	65.5	125.18
MPI-Tile-I/O	850	12	7	70.83	121.42
S3asim	6069	67	55	90.58	110.34

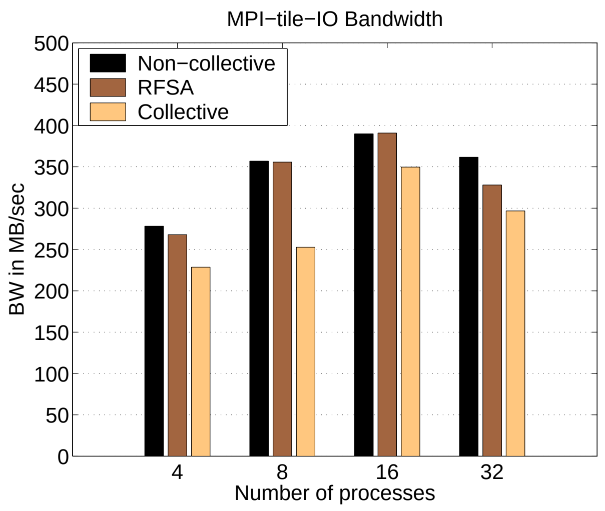
3.4.2 Performance

We have used three benchmarks, with contiguous and non-contiguous access patterns. The experimental setup consists of a 16 node cluster, with PVFS2. Each node is a Dell PowerEdge 2 CPU, dual core with 4GB memory and two 500 GB SAS hard drives. PVFS2 has been setup on all 16 nodes, so all nodes serve as I/O nodes and the compute nodes. The network connection is Intel Pro/1000 NIC, and the cluster network consist of Nortel BayStack 5510-48T GigaBit switch. We have used PVFS2 version 2.7.0 and MPICH2-1.0.7 in our experiments.

To show the efficacy of our selection algorithm in particular, we replaced the specific set of code as shown in listing 1 and listing 2 with our function and gathered I/O time and bandwidth results. The results are then compared with both choices (collective and non-collective) to see the effectiveness of our approach. We evaluate the conflict detection using three different benchmarks for both overlaps and no-overlaps in file access patterns. We compare the time to determine conflicts combined with and without locks and also show the overhead of conflict detection in terms of the communication and computation time. In our experiments, we use the best case of a concurrent write access i.e. when there are no locks, and all processes can perform a write operation concurrently. The worst case used in the experiments is the whole file locks [RLG05], when all writes by



(a) I/O time



(b) BW for overlapping regions

Figure 3.3: MPI-Tile-I/O: Comparing I/O time and BW

different processes become serial. Additionally, we have also used the byte range locks as implemented in [TRL05] [PGK06].

We have also performed some tests of the selection algorithm on Franklin, NERSC Cray XT4 system [refq]. Franklin is a massively parallel processing (MPP) system with 9,660 compute nodes. Each node has dual processor cores, and the entire system has a total of 19,320 processor cores available for scientific applications. Each of Franklin's compute nodes consists of a 2.6 GHz dual-core AMD Opteron processor with a theoretical peak performance of 5.2 $GFlop/sec$. Each compute node has 4 GB of memory, and each service node (e.g. login node) has 8 GB of memory. The full system consists of 102 cabinets with 39 TB of aggregate memory. The theoretical peak performance of Franklin is about 101.5 $TFlop/sec$. The Parallel file system on Franklin is provided by Lustre [refl] with approximately 350 TBytes of user disk space.

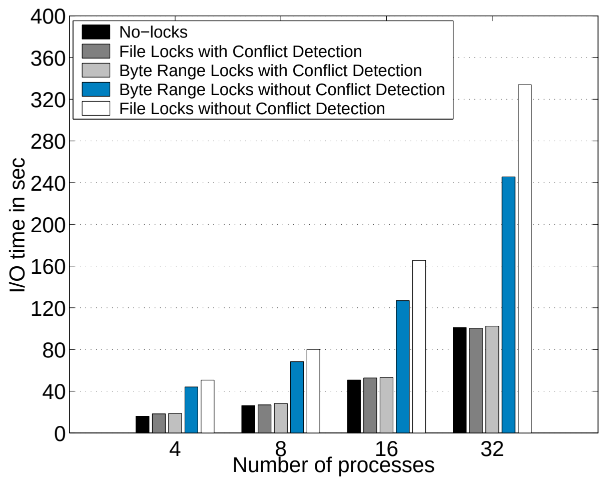
3.4.2.1 MPI-Tile-I/O

RFSA Read/Write MPI-Tile-I/O performs a write operation after creating MPI_Type_Subarray based file view, that logically divides a data file into a dense two dimensional set (x, y) of tiles. The writes are non-contiguous and whether the number of elements will overlap in either the x or y dimension is set on the command line. In our experiments, we considered no-overlapping among different tiles and replaced the code as shown in Listing 3 by our `MPI_RFSA_write` function and measured I/O time and bandwidth. The problem size consists of an array of 4096×8192 elements per process, where the size of each element is 32 *bytes*. Each process writes 1 GB of data. We have tested with 4, 8, 16 and 32 processes, where the processes are arranged in panels of $2X2$, $4X2$, $8X2$ and $16X2$.

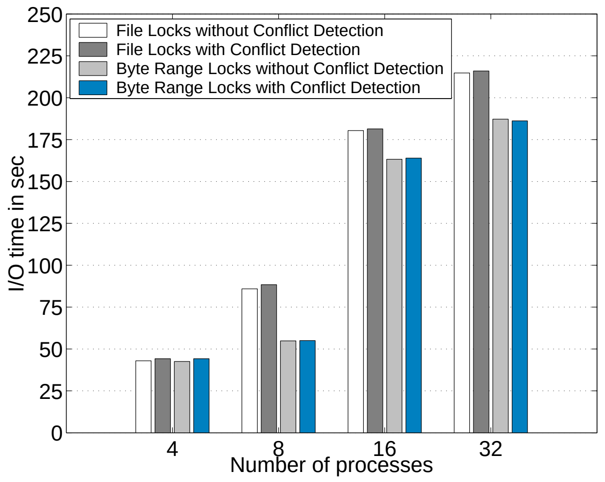
Listing 3 MPI-IO: Selecting Collective/Independent write

```
if(collective)
MPI_File_write_all(mpi_fh, buffer, count, offset,
dtype, &status);
else
MPI_File_write(mpi_fh, buffer, count, offset,
dtype, &status);
```

We have included the results for maximum write time and the bandwidth based on the maximum write time (worst case). It can be seen in Figure 3.3 that independent writes are performing better than collective writes and RFSA write function performs close to the independent writes. The reason is that although the file access pattern is non-contiguous, the byte ranges of all processes are not overlapping, and collective I/O converts to the independent I/O. The only difference in the three bars is due to the synchronization and communication in collective I/O and RFSA. Independent writes are performing the best because, they have no communication. This explains why independent and collective writes do not show significant differences in bandwidth and I/O time.



(a) I/O time for non-overlapping regions



(b) I/O time for overlapping regions

Figure 3.4: MPI-Tile-IO: Comparing I/O time for non-overlapping and overlapping regions with conflict detection and locks.

We also repeated the same procedure but specified the overlap parameter, so for 4096×8192 elements, 1024 was used as the overlap in x direction and 1024 in y direction. Similar trends were observed for the tile I/O overlap option.

Conflict Detection MPI-Tile IO [refp] is used to write non-overlapping and overlapping tiles. *Non-overlapping Access*: The number of dimensions for `MPI_Type_subarray` is 2 in MPI-Tile-IO, and the array of starts gives the x and y position for each tile. The array of sub-sizes returns the size of each tile in both dimensions. The problem size consists of an array of *4096 X 8192* per process, where each element size is 32 bytes. Each process writes 1GB of data, hence if there are 32 processes then the total amount of data written will be 32GB. The 4, 8, 16 and 32 processes are arranged in 2x2, 4x2, 8x2 and 16x2 panels. The I/O time results are shown in Figure 3.4. I/O time includes the time for `MPI_File_set_view` (also the time for conflict detection), `MPI_File_write` and waiting time if there are file locks or byte range locks.

There are five bars, the first bar shows the best case of no-locks, whereas the last bar shows the worst case. The second and third bar shows the case when conflict detection is performed, no conflicts are reported and the underlying file and byte range locking is disabled as a result. The fourth bar performs byte range locks without using conflict detection. We can see that file and byte range locks combined with conflict detection performs close to the no-locks, i.e. an ideal case for the non-overlapping I/O accesses. The overhead of conflict detection is minimal, because the datatypes exchanged in MPI-Tile-IO consist of start and end offset of each tile accessed by a process. This overhead increases with the number of processes, and the detailed overhead results are shown in Figure 3.7. File locks have the worst performance because they introduce sequential access and the I/O time increases with the increase in number of processes.

Overlapping Access: We used *overlap-x* and *overlap-y* options in MPI-Tile-IO to generate the overlapped I/O access patterns. The problem size consists of an array of 4096×8192 per process with an overlap of 512×1024 in x and y direction respectively, where each element size is 32 bytes. Each process writes 1GB of data with an overlapping data of 16MB per process. The atomicity semantics guarantee that the 16MB overlapping data will be defined by one process at a time and not contain any data from more than one processes. We compare the I/O time for file locks and byte range locks with and without conflict detection. No-locks results are not provided here, because the output in the overlapping region is not defined without locks.

I/O time for the other four cases is shown in the Figure 3.4. It should be noted that since there are overlaps, locks cannot be avoided, and the purpose of these results is to demonstrate that if a conflict detection is performed before any locking mechanism, the overhead is not significant. The major contribution of our work is for the cases when there are no overlaps and locking is eliminated completely. These results also show that conflict detection can be combined with any other locking mechanism.

3.4.2.2 S3asim

S3asim is a sequence similarity search algorithm simulator [CFL06], that uses a variety of parameters to adjust the database total fragments, sequence size, query count, etc. The search algorithm offers three different writing strategies. First, only the master process performs all write operations. Second, workers perform collective writes and third, workers perform non-collective writes. In our experiments we have used the aforementioned method where only worker processes write. Additionally, we add the new `MPI_RFSA_write` as a third option, where workers decide whether collective or independent writes should be performed. We measure I/O time of the worker that includes the time of all write operations to the output file.

Table 3.4: S3asim Parameters

Parameter	**Value**
Database minimum sequence length	6 bytes
Database maximum sequence length	10 MB
Database fragments	128
Query Count	20
Maximum data per query	150 MB

After each worker finishes its query processing of its fragment, it sends its ordered scores to the master process. The master process merges the ordered scores to its list and once all fragments of an input query have been processed, it sends the locations in the aggregate file to each worker to write the results. If the collective write option is set, all of the workers synchronize to write their results collectively to the corresponding locations sent by the master. The results are written to mutually exclusive locations in the file, so the data is interleaved but not overlapping. If the independent write option is set, then each worker writes the result data to the output file independently when it receives the location from the master. Since, in collective write all participating processes are blocked until synchronized, workers can not proceed onto the next query unlike independent writes. The overhead of inherent synchronization time when the data access patterns are

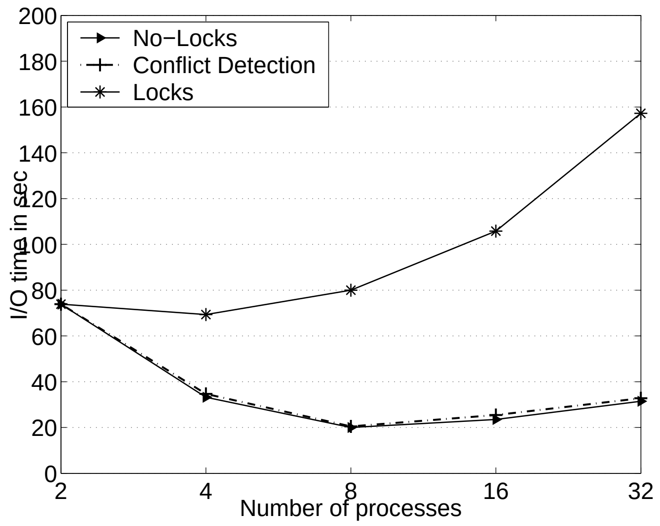
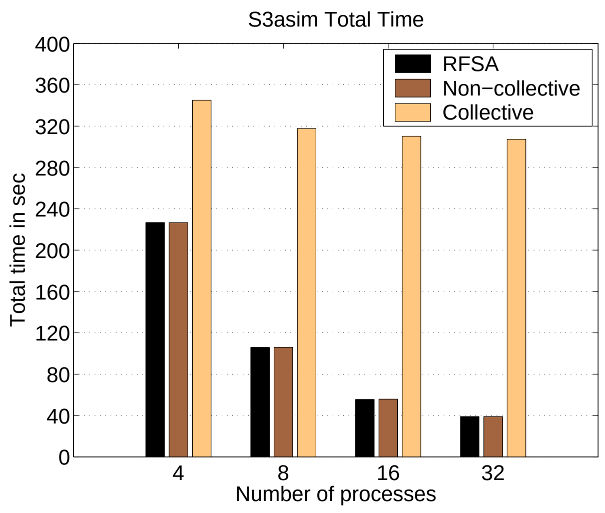


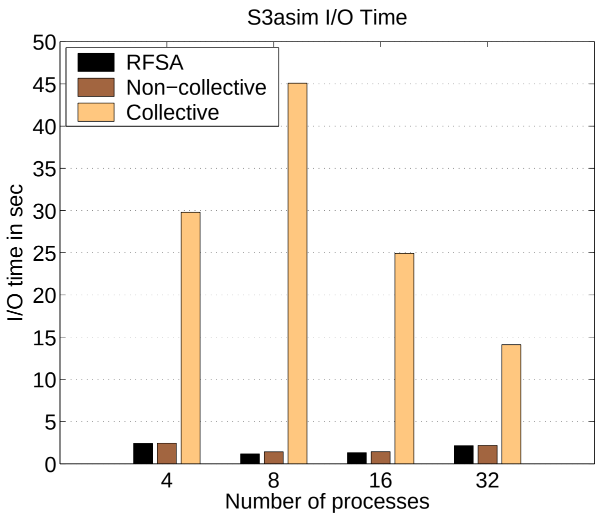
Figure 3.5: S3asim: Comparing I/O time when there are no-locks, conflict detection with file locks, and file locks.

interleaved but non-overlapping significantly impact the overall application performance as shown in Figure 3.6. This behavior also motivates us to automatically detect the best option for writes. We used the configuration shown in Table 3.4 for the test runs. It can be seen in Figure 3.6 that our RFSA performs better than non-collective writes by up to 17%. The reason is that there are 20 queries performing a write operation, and they can be either set to perform collective writes or independent writes. But our selection algorithm in RFSA leads to a combination of collective/independent write operations, hence every individual write is optimized resulting in better performance.

Conflict Detection The datatype used by workers is `MPI_Type_hindexed`, and is defined by an array of block lengths and displacements. All the workers process different segments of the database for query search, and a few of them write the results to a shared file. Figure 3.5 shows three cases; no-locks i.e. none of the locking mechanisms were applied, conflict detection i.e. conflict detection algorithm was run to determine overlaps, and finally file locks. The conflict detection is performed only for the processes that actually write, and on average it performs within 3.6% of the no-locks case. The file locks show an increase in I/O time with an increase in the number of processes, since



(a) S3asim: Comparing Total execution time



(b) S3asim: Comparing worker I/O time

Figure 3.6: S3asim: Total execution time and worker I/O time

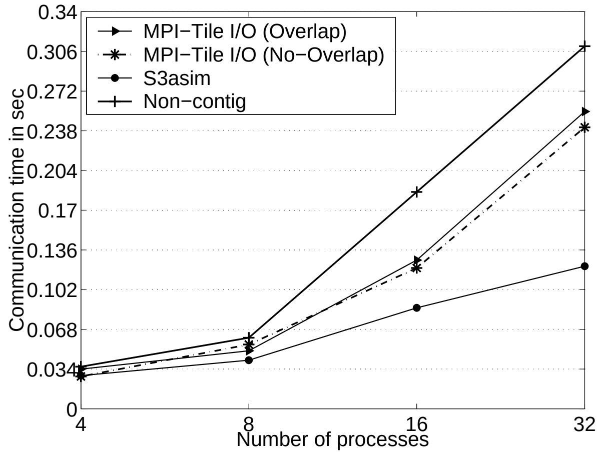
there are fewer writers as compared to the number of processes, and the line is not a steep curve. Our conflict detection algorithm returns no conflict for the S3asim benchmark, and this result is in accordance with [CFL06].

3.4.2.3 Conflict Detection Overhead:

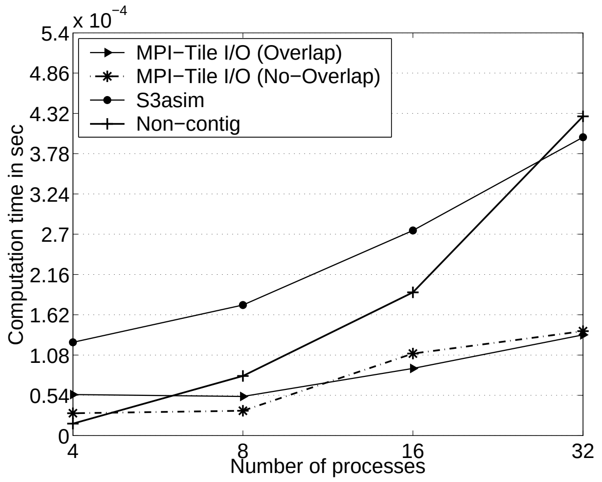
In this section, we present the overhead incurred by our conflict detection algorithm. Each process participates in two collective communication calls to gather the file view and the starting offset. We measure the communication overhead as the time spent in communicating the required information. This overhead depends on the datatypes and the number of processes that actually perform the write operation. In Figure 3.7, we show the communication overhead in different benchmarks. It is noted that in tile-io, each process writes a tile/subarray that may or may not have overlapping regions, but in S3asim only a few workers that find the match perform the write operation. This explains the less communication time for S3asim as compared with tile-io. Non-contig shows the maximum overhead because it has a multi-level datatype and, we need two collective calls to communicate the two levels of datatypes.

After communicating the file views, each process computes the conflicts, which are the results of the comparisons of its own file view with the received ones. The time spent in computing the conflicts is quantified as the computation overhead. This overhead depends on the datatypes and also the size of offset/length pairs generated from the file views. In Figure 3.7, we also show the computation overhead in various benchmarks. It can be seen that the overhead is minimal in tile-io, the reason being that tile-io writes data logically in sub-arrays, and to perform the conflict detection we do not generate the offset/length pairs and use the start and end offsets in each dimension to detect conflicts. For S3asim, the overhead is also minimal because the actual number of workers writing the results is less than the number of processes performing the search. For example, in a $32p$ run, for certain queries only 4 or 5 processes write in the end. S3asim has a few writers but its datatype is more complex and with the increase in the number of processes it has more writers, so S3asim has greater overhead with increased number of processes. Noncontiguous benchmark performs four different access patterns, i.e. contiguous/non-contiguous in memory and contiguous/non-contiguous in file. We only present the results when accesses are either non-contiguous in memory or in a file. A file size of 2GB is used but we keep per process file size constant, the vector length i.e. the number of elements in the vector datatype used is set to 32 and element count i.e. the number of elements in a contiguous chunk to 128. The first derived datatype is `MPI_Type_vector`, and the second derived datatype that is comprised of `MPI_Type_vector` is `MPI_Type_struct`. The overhead of conflict detection for non-contig is shown in Figure 3.7. Non-contig has a steady increase in computation overhead with the number of processes, it is a case of multi-level datatype.

Number of Lock Requests: We emphasize that with conflict detection, if there are no overlaps in the application access pattern, locking can be avoided. Otherwise, only the



(a) Communication Time Overhead



(b) Computation Time Overhead

Figure 3.7: Communication and Computation Overhead of Conflict Detection for different Benchmarks

overlapped region should be locked to guarantee the atomicity of concurrent operations. Finally, we investigate the utilization of our scheme with the existing locking mechanisms. Assuming that there are three locking implementations, i.e. whole-file, byte range and list locks, we compare the number of locks per client in each case with and without conflict detection. There is one lock per client for the file locks and the byte range locks,

Table 3.5: Reduction in number of locks

Approach	**Number of Locks/Client**
Whole-File Locks	1
Byte-Range Locks	1
List Locks	64
Locks with Conflict detection	*No-0verlaps:* 0 *Overlaps:* 1 (Whole-file), 1 (Byte-Range), Number of CR (List)

but these are coarse-grained locks. The non-contiguous access patterns observe false sharing with coarse-grained locks. We want to show that a locking mechanism combined with our approach is effective in reducing the number of lock requests that will be issued for any lock server that needs communication and space on the server to be stored. Many scientific applications have patterns that would require hundreds of thousands of

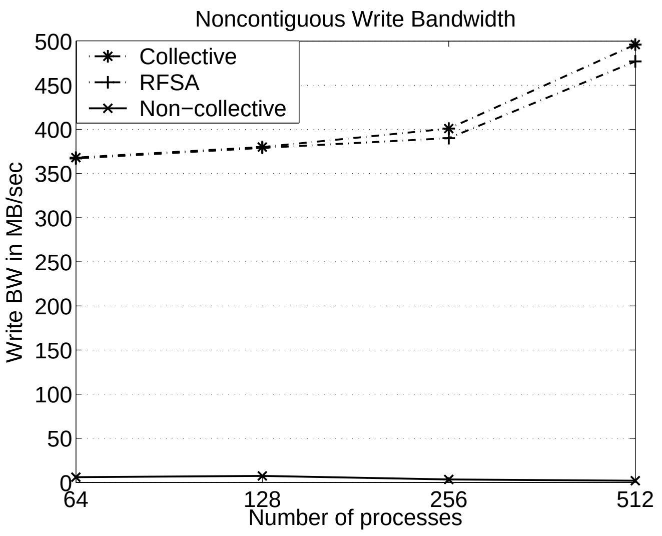


Figure 3.8: Noncontiguous Benchmark: Comparing Write Bandwidth on Franklin Supercomputer

locks [CLC07]. In Table 3.5, we show a simple comparison of the number of locks (whole file, byte-range and list) with and without using conflict detection. It should be noted that the list locks and the datatype locks are very fine-grained locks. The lock acquiring process is instigated by a client. The client first calculates which servers to access for the locks; the saving from conflict detection will come in the form of either none or a fewer number of requests. The implementation of conflict detection with list locks [ACT06] is left for future work.

3.4.2.4 Scalability Testing on Franklin Supercomputer

We also ran Noncontiguous benchmark on the Franklin machine. We experimented with 64-512 processes, by setting the number of elements in the vector to 4096 and the element count for a contiguous chunk to 1024, and the file size is same for all runs. In Figure 3.8, the bandwidth represents the cumulative bandwidth when the file access is noncontiguous. We can see that the RFSA selection algorithm is performing a little worse than collective write with the increase in number of processes. The reason is that

RFSA selection algorithm has an additional overhead of communicating its start and end offset pairs to all processes. The byte range locks of the Lustre file system affect the performance of individual writes, making the write accesses sequential. In the current version of our selection algorithm, there is no criteria for checking the communication overhead of redistribution of data as compared to performing individual I/O operations.

CHAPTER 4

MRAP - A MAPREDUCE BASED FRAMEWORK TO SUPPORT I/O-INTENSIVE HPC ANALYTICS WITH ACCESS PATTERNS

4.1 Introduction

In this chapter, we describe the MapReduce Framework for HPC Data Analytics with non-contiguous access patterns. It consists of MapReduce-based functions and templates that are provided for specifying the non-contiguous access patterns and various optimizations to improve the performance of the applications using theses methods.

Today's cutting-edge research deals with the increasing volume and complexity of data produced by ultra-scale simulations, high-resolution scientific equipment and experiments. These datasets are stored using parallel and distributed file systems and are frequently retrieved for analytics applications. There are two considerations regarding these datasets. First, the scale of these datasets [refd, SWJ05] affects the way they are stored (e.g. metadata management, indexing, file block sizes, etc). Second, in addition to being extremely large, these datasets are also immensely complex. They are capable of representing systems with high levels of dimensionality and various parameters that include length, time, temperature, etc. For example, the Solenoidal Tracker at the Relativistic Heavy Ion Collider (STAR; RHIC) experiment explores nuclear matter under extreme conditions and can collect seventy million pixels of information one hundred times per second [refv]. The data generated by some astrophysics applications consist of millions of objects in a three dimensional space where each object has its own unique properties [refa, SKT00]. Such extraordinarily rich data presents researchers with many challenges in representing, managing and processing (analyzing) it. Many data processing frameworks coupled with distributed and parallel file systems have emerged in recent years to cope with these datasets [reff, Dea06, DG04, GGL03].

However, the raw data obtained from the simulations/sensors needs to be stored to data-intensive file systems in a format useful for the subsequent analytics applications. There is an information gap because current HPC applications write data to these new file systems using their own file semantics, unaware of how the new file systems store data, which generate unoptimized writes to the file system. In HPC analytics, this information gap becomes critical because the source of data and commodity-based systems do not have the same data semantics. For example, in many simulations-based applications, data sets are generated using MPI datatypes [GTL99] and self describing data formats like netCDF [refr, LLC03] and HDF5 [refi, refs]. However, scientists are using frameworks like MapReduce for analysis [EPF08, KNG09, refk, MTF08, Sch09] and require datasets to be copied to the accompanied distributed and parallel file system e.g. HDFS [Bor]. The challenge is to identify the best way to retrieve data from the file system following the original semantics of the HPC data.

One way to approach this problem is to use a high-level programming abstraction for specifying the semantics and bridging the gap between the way data was written and the way it will be accessed. Currently, the way to access data in HDFS-like file systems is to use the MapReduce programing abstraction. Because MapReduce is not designed for semantics based HPC analytics, some of the existing analytics applications use multiple MapReduce programs to specify and analyze data [KNG09, Sch09]. We show the steps performed in writing these HPC analytics applications using MapReduce in Figure 4.1 a). In Figure 4.1 a), N MapReduce phases are being used; the first MapReduce program is used to filter the original data set. If the data access pattern is complex, then the second MapReduce program will perform another step on the data and extract another dataset. Otherwise, it will perform the first step of analysis and generate a new data

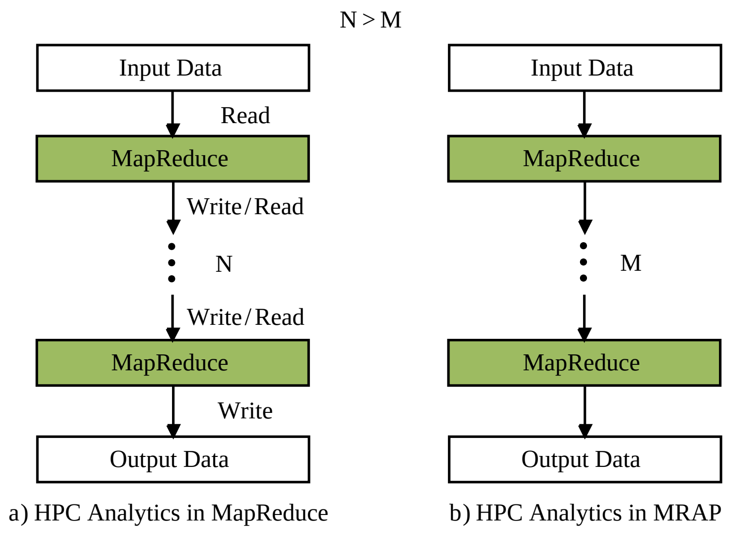


Figure 4.1: Steps performed in writing HPC analytics using both MapReduce and MRAP.

set. Similarly, depending on the data access pattern and the analysis algorithm, multiple phases of MapReduce are utilized.

For example, consider an application that needs to merge different data sets followed by extracting subsets of that data. Two MapReduce programs are used to implement this access pattern. That is, the first program will merge the data set and the second will extract the subsets. The overhead of this approach is quantified as 1) the effort to transform the data patterns in MapReduce programs, 2) number of lines of code required for MapReduce data pre-processing and 3) the performance penalties because of reading excessive datasets from disk in each MapReduce program.

We propose a framework based on MapReduce, which is capable of understanding data semantics, simplify the writing of analytics applications and potentially improve performance by reducing MapReduce phases. Our aim in this project is to utilize the scalability and fault tolerance benefits for MapReduce and combine them with scientific access patterns. *Our framework, called **MapReduce with Access Patterns (MRAP)**, is a unique combination of the data access semantics and the programming framework (MapReduce), which is used in implementing HPC analytics applications.* We have considered two different types of access patterns in the MRAP framework. The first pattern is for the matching, or similar analysis operations where the input data is required from two different data sets. These applications access data in smaller contiguous regions. The second pattern is for the other types of analysis that uses data in multiple smaller non-contiguous regions. The MRAP framework consists of two components to handle these two patterns; The MRAP API and MRAP data restructuring.

The MRAP API is used to specify both access patterns. It is flexible enough to specify the data pre-processing and analytics in fewer MapReduce programs than currently possible with traditional MapReduce as shown in the Figure 4.1 b). Figure 4.1 shows that MRAP is utilizing M phases, and $M < N$, where N is the number of corresponding phases in a MapReduce based implementation. With MRAP, $M < N$ is possible because it allows the users to describe data semantics (various access patterns based on different data formats) in a single MRAP application. As mentioned in the previous example, two MapReduce programs are used to describe an access pattern that requires merge and subset operations on data before analyzing it. In MRAP, only one MapReduce program is required, because data is read in the required merge pattern in the Map phase, and subsets are extracted in the reduce phase. MRAP data restructuring reorganizes data with the copy operation to mitigate the performance penalties resulting from non-contiguous small I/O requests (second access pattern). Our prototype of the MRAP framework includes basic implementation of functions that allow users to specify data semantics and data restructuring to improve the performance of our framework. Figure 4.2 shows the high-level system view with MRAP. Our results with a real application in bioinformatics and an I/O kernel in astrophysics, show up to 33% performance improvement by using the MRAP API, and up to 70% performance gain by using data restructuring.

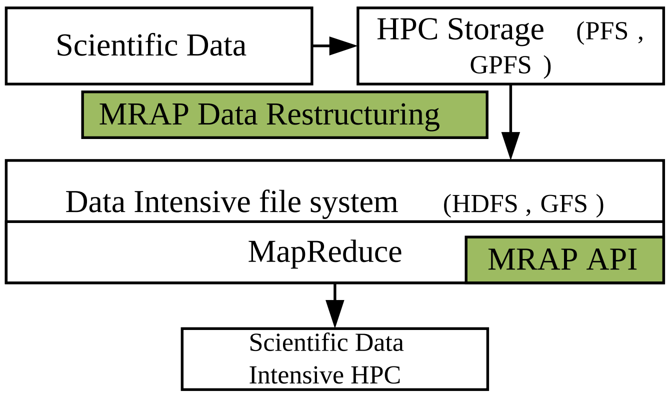
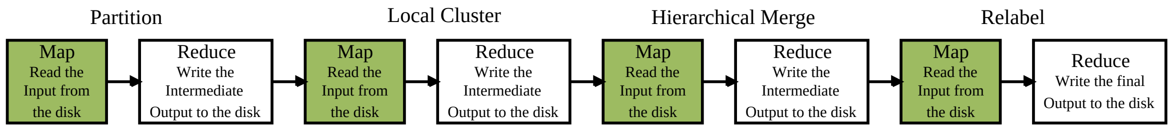

Scientific Data
HPC Storage (PFS , GPFS)
MRAP Data Restructuring
Data Intensive file system (HDFS , GFS)
MapReduce
MRAP API
Scientific Data Intensive HPC

Figure 4.2: High-Level System View with MRAP

4.2 Motivation for developing MRAP

In this section we describe the motivation for developing MRAP. Data for analysis comes in all types of formats, file semantics, and comes from many various sources; whether it be weather data to high energy physics simulations. Within these patterns, we also need to provide for efficient file accesses. Current approaches show that to implement these access patterns, a series of MapReduce programs is utilized [KNG09, Sch09], also shown in the Figure 4.1.

In a MapReduce program, the map phase always reads data in contiguous chunks, and generates data in the form of (key, value) pairs for the reduce phase. The reduce phase then combines all the *values* with the same *keys* to output the result. This approach works well when the order and sequence of inputs do not affect the output. On the other hand, there are algorithms that require data to be accessed in a given pattern and sequence. For example, in some image pattern/template matching algorithm, the input of the function must contain a part of the original image and a part of the reference image. Using MapReduce for this particular type of analysis would require one MapReduce program to read the inputs from different image files and then combine them in the reduce phase for further processing. A second MapReduce program would then analyze the results of the pattern matching. Similar behavior has been observed in two other applications as



Partition
Local Cluster
Hierarchical Merge
Relabel
Map
Read the Input from the disk
Reduce
Write the Intermediate Output to the disk
Map
Read the Input from the disk
Reduce
Write the Intermediate Output to the disk
Map
Read the Input from the disk
Reduce
Write the Intermediate Output to the disk
Map
Read the Input from the disk
Reduce
Write the final Output to the disk

Figure 4.3: Overview of the distributed FoF algorithm using 4 MapReduce phases [KNG09].

shown in the Figure 4.3 and Figure 4.8 a). Figure 4.3 describes a distributed *Friends-of-Friends* algorithm; the four phases are MapReduce programs [KNG09]. Figure 4.8 a) is discussed in the Section 4.4. Both these figures show the multi-stage MapReduce applications developed for scientific analytics.

Using this multi-stage approach, all the intermediate MapReduce programs write data back to the file system while subsequent applications read that outputted data as input for their task. These excessive read/write cycles impose performance penalties and can be avoided if initial data is read more flexibly as compared with traditional approach of contiguous chunks read. Our proposed effort, MRAP API, is designed to address these performance penalties and to provide an interface that allows these access patterns to be specified in fewer MapReduce phases. Hence, the goal of the MRAP API is to deliver greater I/O performance than the traditional MapReduce framework can provide to scientific analytics applications.

4.2.1 Small I/O

Additionally, some scientific access patterns generally result in accessing multiple non-contiguous small regions per task, rather than the large, contiguous data accesses seen in MapReduce. Due to the mismatch in sizes of distributed file system (DFS) blocks/chunks and the logical file requests of these applications, small I/O problem arises. The small I/O requests from various access patterns impact the performance by accessing excessive amount of data, when implemented using MapReduce. The default distributed file system chunk used in current setup is either 64/128 MB. Most of the scientific applications store data with each value comprising of a few KBs. Each small I/O region (a few KBs) in the file may map to large chunks (64/128 MB) in the file system. If each of the requested small I/O region is 64 KB, then a 64 MB chunk will be retrieved to process only 64 KB making it extremely expensive.

A possible approach can be to decrease the chunk size to reduce the amount of extra data accessed. These smaller chunks increase the size of metadata [refj]. Also, the small I/O accesses also result in a large number of I/O requests sent to the file system. In MapReduce framework, smaller chunks and large number of I/O requests become extremely challenging because there is a single metadata server (*NameNode*) that handles all the requests. These small I/O requires efficient data layout mechanisms because the patterns can be used to mitigate performance impact. Hence, providing semantics for various access patterns which generate small I/O requires optimizations at both the programming abstraction and the file system levels of MapReduce.

4.2.2 Data Locality

The MapReduce framework runs in a commodity-based cluster environment, and the compute and data nodes are co-located. That is the same set of nodes used for scheduling map/reduce tasks are used to provide the required data sets. In the traditional MapReduce setup, each map task is assigned a DFS block for processing. The existing scheduling mechanism is optimized for the case when either a single chunk or multiple chunks on the same *DataNode* are processed per map task. Hence, selecting nodes for a given map task is *locality-aware* and generally results in the least amount of data transfers.

Different access patterns result in accessing smaller regions of data as compared with a complete chunk. These smaller regions are likely to be mapped to multiple chunks and most likely multiple chunks per map tasks will be read. Hence, not only excess data will be read but also overlapping of chunks may occur. The above-mentioned cases violate data locality because now one chunk may be requested by multiple map tasks increasing demand of a particular chunk. Also, multiple chunks that are scattered across nodes may be requested by one map task. These cases require some ways to avoid data movements to satisfy the chunk demand.

4.3 Design of MRAP

The MRAP framework consists of two components; 1) MRAP API, that is provided to eliminate the multiple MapReduce phases used to specify data access patterns, and 2) MRAP data restructuring, that is provided to further improve the performance of the access patterns with small I/O problem. Before we describe each component of the MRAP framework, it is important to understand the steps that existing HPC MapReduce applications perform in order to complete an analysis job. These steps are listed as follows:

- Copy data from external resource (remote storage, sensors, etc) to HDFS/similar data-intensive file system with MapReduce for data processing.
- Write at least one data pre-processing application (in MapReduce) to prepare data for analysis. The number of MapReduce applications depends on the complexity of initial access pattern.
  - This data preparation can be a conversion from raw data to scientific data format or in general filtering of formatted data.

- Write at least one MapReduce application to analyze the prepared datasets. The number of MapReduce applications for analysis varies with the number of steps in the analytics algorithm.

As compared with this existing MapReduce setup, our MRAP framework will allow the following steps:

- Copy data from external resource to HDFS/similar file system (perform MRAP data restructuring if specified by the user).
- Write an MRAP application to specify the pattern and subsequent analysis operation.
  - If data was restructured at the copy time, then map/reduce phases will be written only for the analysis operation.

We discuss the two components, i.e. MRAP API and MRAP data restructuring in detail in the next subsections.

4.3.1 MRAP API

The purpose of adding this data awareness functionality is to reduce the number of MapReduce programs that are used to specify HPC data access patterns. We provide an interface for two types of data access patterns; 1) applications that perform match operations on the data sets, and 2) applications that perform other types of analysis by reading data non-contiguously in the files. At least one MapReduce program is used to implement either 1 or 2. *Note: There are some complex access patterns which consist of different combinations of both 1 and 2.* We first briefly describe how MapReduce programs work, and why they require more programming efforts when it comes to HPC access patterns.

A MapReduce program consists of two phases: a Map and a Reduce. The inputs and outputs to these phases are defined in the form of a **key** and a **value** pair. Each map task is assigned a split specified in *InputSplit* to perform data processing. A *FileSplit* and a *MultiFileSplit* are the two implementations of the interface *InputSplit*. The function *computeSplitSize* calculates the split size to be assigned to each map task. This approach guarantees that each map task will get a contiguous chunk of data. In the example with the merge and subset operations on data sets, two MapReduce programs will be used as shown in the following.

- MapReduce1:

  `(inputA, inputB, functionA, outAB)`

- functionA (merge)

  `map1(inputA, inputB, splitsizeA, splitsizeB,`

  `merge, outputAB)`

  where merge has different criteria, e.g. merge on exact or partial match, 1-1 match of inputA and inputB, 1-all match of inputA and inputB, etc.

  `reduce1(outputAB, sizeAB, merge_all, outAB)`

- MapReduce2:

  `(outAB, functionB, outC)`

- functionB (subset)

  `map2(outAB, splitsizeAB,subset)`

  where subset describes the offset/length pair for each split of size splitsizeAB.

  `reduce2(subset, outC)`

We now describe MRAP API, classes and functions, and how it allows the user to minimize the number of MapReduce phases for data pre-processing. Then, we describe the provided user templates that are pre-written MRAP programs. The user templates are useful in the cases when the access patterns are very regular and can be specified

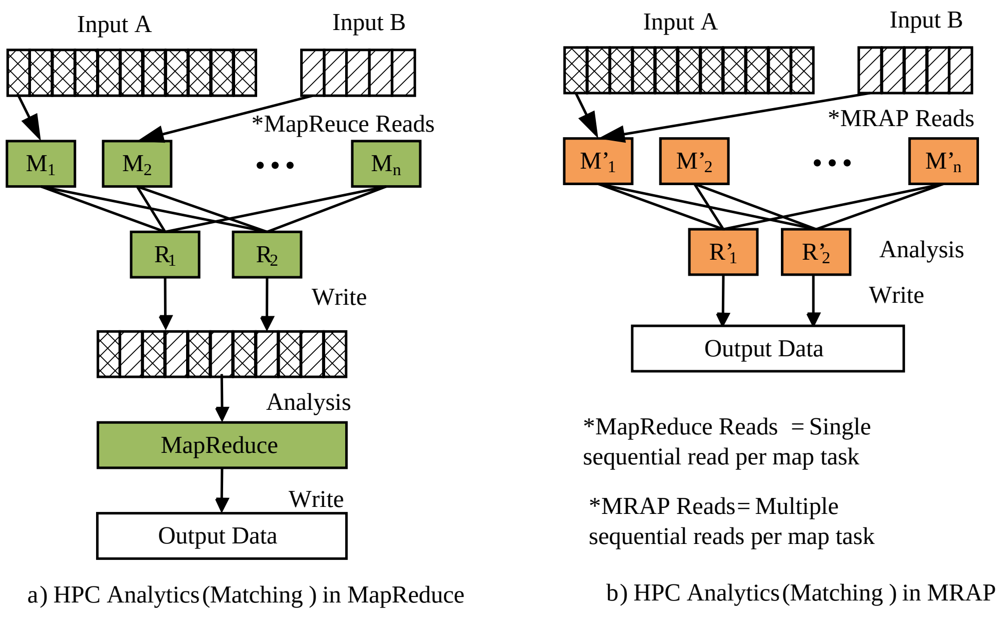

Input A
Input B
*MapReuce Reads
M_1
M_2
M_n
R_1
R_2
Write
Analysis
MapReduce
Write
Output Data
a) HPC Analytics (Matching) in MapReduce
Input A
Input B
*MRAP Reads
M'$_1$
M'$_2$
M'$_n$
R'$_1$
R'$_2$
Analysis
Write
Output Data
*MapReduce Reads = Single sequential read per map task
*MRAP Reads = Multiple sequential reads per map task
b) HPC Analytics (Matching) in MRAP

Figure 4.4: A detailed View of comparing MapReduce and MRAP for applications that perform matching.

by using different parameters in a configuration file *e.g, a vector access pattern can be described by the size of data set, size of a vector, number of elements to skip between two vectors, and number of vectors in the data set.*

In MRAP, we provide a set of two classes that implement customized InputSplits for the two aforementioned HPC access patterns. By using these classes, users can read data in one of these two specified patterns in the map phase and continue with the analysis in the subsequent reduce phase. The first pattern for the applications that perform matching and similar analysis is defined by $(inputA, inputB, splitSizeA, splitSizeB, function)$. $inputA/inputB$ represents both single or multi-file inputs. Each input can have a different split size, and then the *function* describes the sequence of operations to be performed on these two data sets. The new *SequenceMatching* class implements *getSplits()*, such that each map task reads the specified $splitSizeA$ from $inputA$ and $splitSizeB$ from $inputB$. This way of splitting data saves one MapReduce data pre-processing phase for the applications with this particular behavior as shown in the Figure 4.4.

The second access pattern is a non-contiguous access, where each map task needs to process multiple small non-contiguous regions. This pattern is defined as $(inputA, (list\_of\_offsets/maptaks, list\_of\_lengths/maptask), function)$ These patterns are more complex to implement, because these patterns are defined by multiple offset/length pairs. Each map task can either have the same non-contiguous pattern, or different pattern. Current abstraction for *getSplits()* either return an offset/length pair or a list of filenames/lengths to be processed by each map task. The new *AccessInputFormat* class with our new *getSplits()* method implements the above-mentioned functionality to the MapReduce API. These patterns are generally cumbersome to describe using MapReduce existing split methods because the existing splits are contiguous, and require the pattern to be transformed to the MapReduce (key, value) formulation.

The above mentioned access pattern that involved both merge and subset operations, is specified in one MRAP program, as follows:

- `(inputA, inputB, functionAB, outC)`
- functionAB (merge, subset)

  `map1(inputA, inputB, splitsizeA, splitsizeB,`

  `merge, outAB)`

  Read data from all sources in the required merge pattern

  `reduce1(outAB, sizeAB, subset, outC)` *Extract data of interest from the merged data*

We now explain the user templates. User templates are pre-written MRAP programs using new input split classes that read data according to the pattern specified in configuration file. We provide template programs for reading the astrophysics data for halo finding, and general strided access patterns. In the configuration file, we require type of application, e.g. any MPI simulation, or astrophysics to be specified. Each of these different categories generate data in different formats and access data in different patterns.

By providing a supported application type, MRAP can generate a template configuration file to the user. A sample configuration file for a strided access pattern is as follows.

```
<configuration>
Defining structure of the new file
<property>
<name>strided.nesting_level</name>
<value>1</value>
<description>It defines the nesting levels. < /description>
<property>
<name>strided.region_count</name>
<value>100</value>
<description>It defines the number of regions in a non-contiguous access pattern. < /description>
</property>
<property>
<name>strided.region_size</name>
<value>32</value>
<description>It defines the size in bytes of a region, i.e. a stripe size. < /description>
</property>
<property>
<name>strided.region_spacing</name>
<value>10000</value>
<description>It defines the size in bytes between two consecutive regions, i.e. a stride. <
/description>
</property>
</configuration>
```

We give a few examples of the access patterns that can be described in a configuration file. A *vector based* access pattern is defined as if each request having the same number of bytes (i.e. a vector) and is interleaved by a constant or variable number of bytes (i.e. a stride) and number of vectors (i.e. a count). A slightly complex vector pattern is a *nested* pattern, that is similar to a vector access pattern. However, rather than being composed of simple requests separated by regular strides in the file, it is composed of *strided segments* separated by regular strides in the file. That is, a nested pattern is defined by two or more strides instead of one as in a vector access pattern. A *tiled* access patterns is seen in datasets that are multidimensional, and are described by number of dimensions, number of tiles/arrays in each dimension, size of each tile, size of elements in each tile. An *indexed* access pattern is more complex, it is not a regular pattern and describe a pattern using a list of offsets and lengths. There are many more patterns used

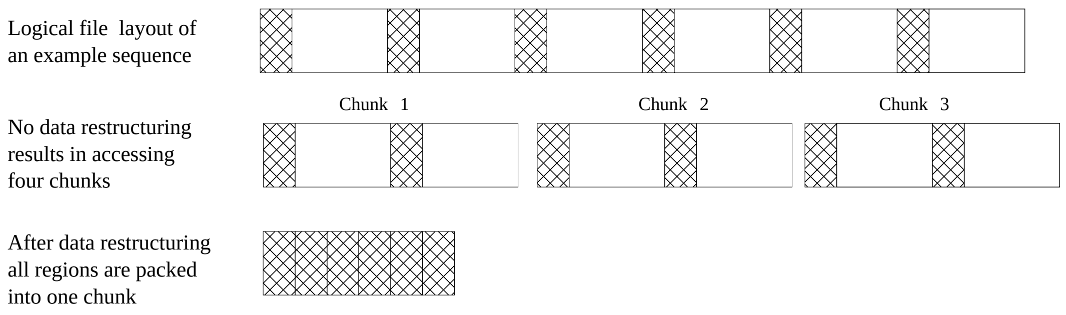

Logical file layout of an example sequence
Chunk 1
Chunk 2
Chunk 3
No data restructuring results in accessing four chunks
After data restructuring all regions are packed into one chunk

Figure 4.5: An example showing the reduction in number of chunks per map task after data restructuring

in HPC applications, but we have included the aforementioned patterns in this version of MRAP.

4.3.2 MRAP Data Restructuring

MRAP data restructuring is provided to improve the performance of access patterns which access data non-contiguously in small regions to perform analysis. Currently, there is a little support for connectivity of HDFS to resources that generate scientific data. In most cases, data must be copied to a different storage resource and then moved to HDFS. Hence, a copy operation is the first step performed to make data available to MapReduce applications. We utilize this copy phase to reorganize the data layout and convert small I/O accesses to large by packing small (non-contiguous) regions into large DFS chunks (also shown in the Figure 4.5). We implement data restructuring as an MRAP copy operation that reads data, reorganizes it and then writes it to the HDFS. It reorganizes the datasets based on user specification when data is copied to HDFS (for the subsequent MapReduce applications). The purpose of this data restructuring when copying data is to improve small I/O performance.

The MRAP copy operation is performed in one of two ways. The first method copies data from remote storage to the HDFS along with a configuration file, the second method does not have a configuration file. For the first method, in order to reorganize datasets, MRAP copy expects a configuration file, defined by the user, describing the logical layout of data to be copied. In this operation, the file to be transferred to HDFS and the configuration file are submitted to MRAP copy. As the file is written to HDFS, MRAP restructures the data into the new format defined by the user. When the copy operation is completed, the configuration file is stored with the restructured data.

This configuration file is stored with the file because of a very important sub-case of the MRAP copy operation, on-the-fly data restructuring during an MRAP application. In this case, a user runs an MRAP application on a file that was previously restructured by the MRAP copy function. In this job submission, another configuration file is submitted to define how the logical structure of the data in said file should look before the current MRAP application can begin. As shown in Figure 4.6, the configuration file stored with data during the initial restructuring is compared with the configuration file submitted with MRAP application. If the two configuration files match, that is, the logical data layout of the stored file matches what the MRAP application is expecting, then the MRAP operation begins. Else, data restructuring occurs again and the MRAP application runs once the file is restructured.

In the case of the second copy method, data is copied to HDFS from remote storage without any file modification. This option would be used when the user wants to maintain the original format in which the data was written. Hence, option two in MRAP will be performed as a standard HDFS copy. As discussed in the paragraph above, this option is amenable to use if the file in question is constantly being restructured. The approach used to optimize file access for this case is discussed later in the section.

As mentioned earlier that MRAP copy operation performs data restructuring, we now explain the data restructuring algorithm. The data restructuring converts small non-contiguous regions to large contiguous regions, and can be formulated as a *bin packing* problem where different smaller objects are packed in a minimal number of larger bins.

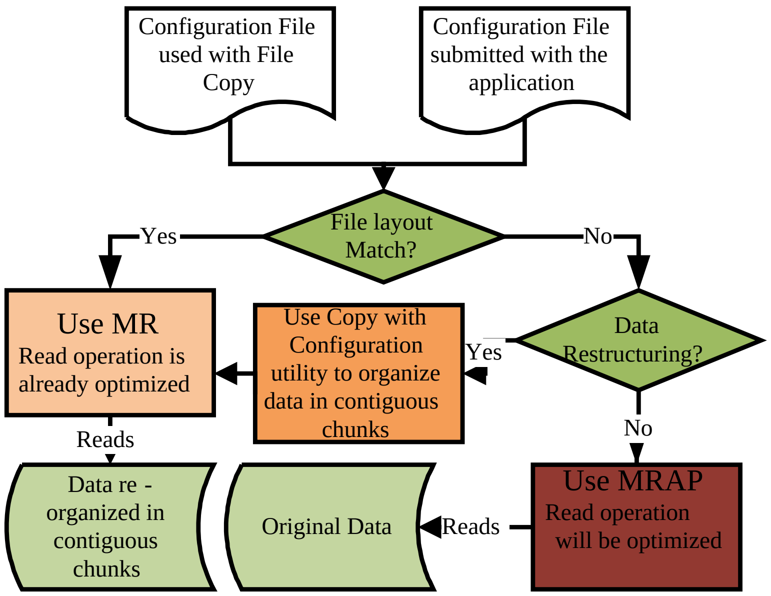

Configuration File used with File Copy
Configuration File submitted with the application
File layout Match?
Yes
No
Use MR
Read operation is already optimized
Use Copy with Configuration utility to organize data in contiguous chunks
Yes
Data Restructuring?
No
Reads
Data re - organized in contiguous chunks
Original Data
Reads
Use MRAP
Read operation will be optimized

Figure 4.6: Flow of operations with data restructuring

Definition: Given items with sizes $s_1, \ldots, s_n$, pack them into the fewest number of bins possible, where each bin is of size V.

This problem is a *combinatorial NP-hard* and there are many proposed heuristics to solve this. In data restructuring each item from bin packing problem corresponds to a smaller non-contiguous region, whereas each bin corresponds to a DFS chunk of size V. We use First-fit algorithm to pack the smaller regions into chunks. The algorithm is as follows:

Algorithm 4.3.1 Data Restructuring Algorithm

Input: A set U which consist of all smaller region size required by a map task in a MapReduce Application, $U = \{s_1, s_2, ..., s_m\}$, where s_i corresponds to size of i^{th} region, and m is the number of non-contiguous regions requested the task. A set C of empty chunks $C = \{c_1, c_2, ..., c_p\}$, where capacity of each c_x is V. $p = V/s_1 \times m$ when all s_i are of the same size else p is unknown.
Output: Find minimal p such that all smaller regions are packed into p number of chunks.
Steps:

```
for i is 1 to $m$, [Iterate through all the elements in set $U$]
$\forall c_j \in C$ = empty, $0 < j < p$
if $\sum_i s_i \leq V)$
Add $i^{th}$ element to $c_j$
else Add $c_j$ to $C$
increment $j$ i.e. start a new chunk
end for
$p = j$, since $j$ is keeping track of when a new chunk is added.
```

The time to perform data restructuring based on Algorithm 4.3.1 is determined by the following: number of regions in the access patterns that are needed to be combined into chunks is m, and size of each region is s_i. time to access each region T_{readS}, number of chunks after restructuring p where size of each chunk is V, time to write one region to the chunk is T_{chunk} and time to update metadata for the new chunk is T_{meta}. The time to perform data restructuring will be
$T_{dr} = (m \times T_{readS}) + (p \times (V/s_i) \times T_{chunk}) + (p \times T_{meta})$.
We can also determine the execution time of the application with M tasks, that will involve the time to read any access pattern $(M \times m \times T_{readS})$ and processing time T_p,
$T_{comp} = T_p + (M \times m \times T_{readS})$.
Data restructuring will be not beneficial if $T_{dr} > T_{comp}$. However, $T_{dr} < T_{comp}$ will result in contiguous accesses, significantly improving the performance.

The benefit of data restructuring is that when an application is run multiple times and requires only one data layout, the read operation has been highly optimized, making the total execution time of the operation much shorter. It minimizes the number of chunks required by a map task because after restructuring all smaller regions that are scattered among various chunks are packed together as shown in the Figure 4.5. However, if each application uses the file in a different way, that is, the file requires constant restructuring, then data restructuring will incur more overhead as compared to the performance benefits.

4.3.3 MRAP Scheduling

Scheduling is critical in our framework because some applications assign multiple chunks/-multiple small files to each map task. As a result, selecting a node to schedule map task with minimal network contention becomes challenging. We approach network contention from the angle that with appropriate scheduling schemes we can reduce the chunk transfers over the network. Our algorithm determines the nodes that are best for scheduling map tasks based on the *location of multiple chunks* and *minimal data transfer latencies* of these chunks.

In order to analyze the impact of data transfer latency on a map task, we quantify the execution time of a map phase in an application with N map tasks as $T = max(T_i)$, where $1 < i < N$, and T_i denotes the time taken by i^{th} map task and is given by $T_i = T_{computation} + T_{diskIO} + T_{network\ transfer\ time}$. $T_{computation}$ represents the computation time, T_{diskIO} represents the time needed for data transfer from the local disk, and $T_{network\ transfer\ time}$ represents the time to transfer data over the network from a remote host. Network latency can increase this time and will impact the performance. We try to minimize the $T_{network\ transfer\ time}$ by carefully selecting the nodes for task execution and data provision. If there are n chunks required by a map task, and m chunks are present on the node selected for that map task, then $m - n$ chunks will be transferred to the primary selected node. There can be at least one node providing these $m - n$ chunks. If there are multiple nodes, then selection of the nodes for map task execution and nodes for data transfer becomes challenging.

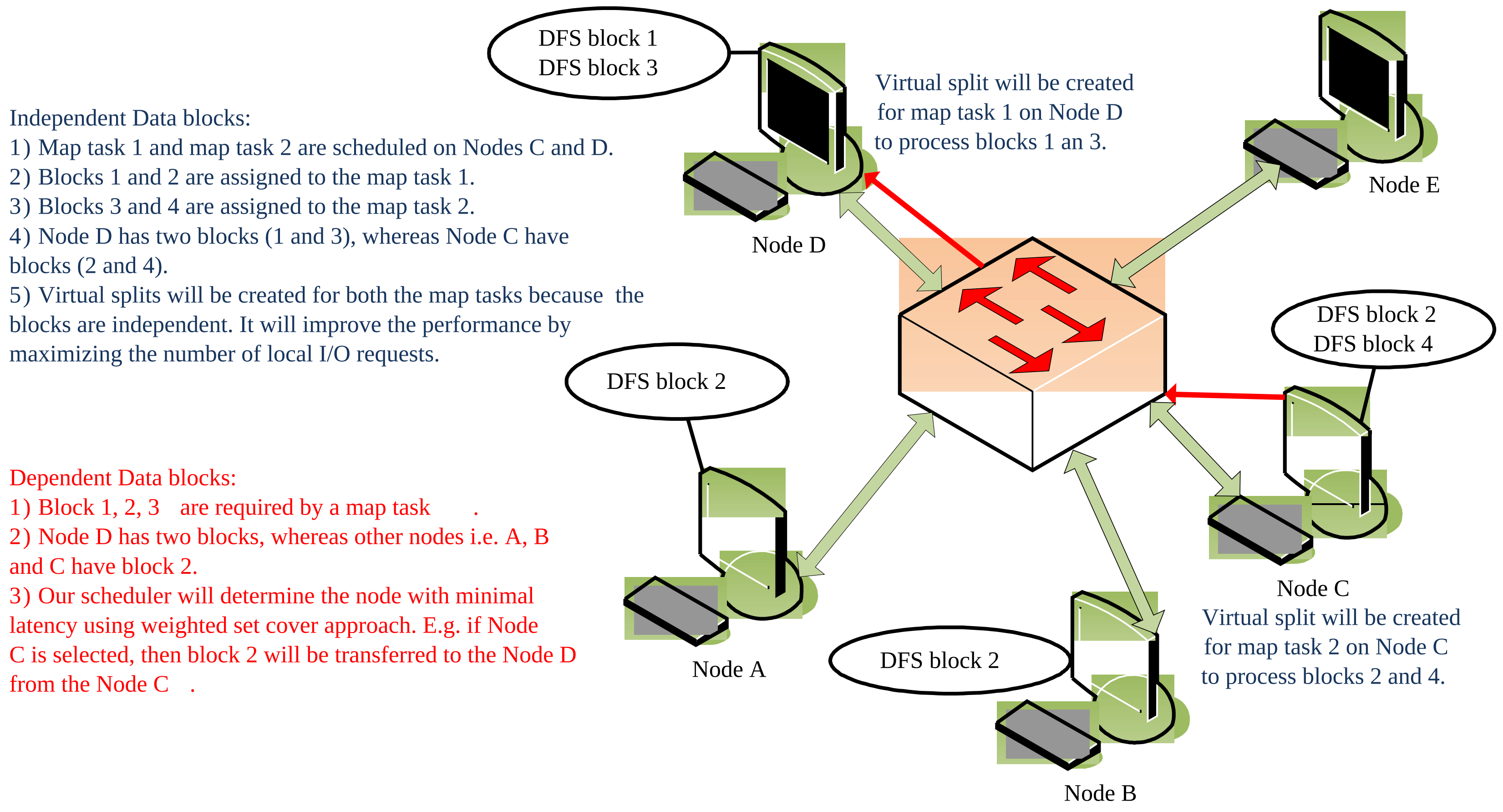
We identify two cases for our scheduling scheme as shown in the Figure 4.7. First, the multiple chunks/files assigned to a map task can be processed independently, hence we can create *virtual splits* for each map task. As shown in the figure, blocks 1 and 2 are assigned to map task 1, which was scheduled on Node D. Since, the blocks were independent, blocks 1 and 3 were combined into a virtual split to minimize the number of remote I/O requests.

Second, multiple chunks have to be processed together, hence tasks will be scheduled after considering the location of multiple chunks. As we can see in the Figure 4.7, blocks 1, 2 and 3 are assigned to a map task. It is scheduled on Node D because, Node D has maximum chunk/block contribution. Node D does not have any replicas of block 2, hence the node with minimal transfer latency will be used to provide the missing chunks (we used Node C as the one with minimal latency).

In the first case, the final outcome is not be affected by the order in which these chunks are processed. Our frame work intelligently creates map tasks depending on the chunks located on one node. Each task receives a list of splits. These splits are then compared with the local chunks. If local chunks are different than the requested chunks, we create a *virtual split* of local chunks. We keep the same number of map tasks and let each map task process the same number of chunks if and only if these chunks are marked as independent chunks in the initial configuration file.

We run a light weight scheduler with the MapReduce application. The purpose of this scheduler is to determine In the current implementation, we assume that there is only one copy of data set in the system, and also all the nodes are balanced with the data chunk distribution. [We can always run a balancer program to make sure that all nodes are evenly loaded.] In such a case, we have two options, either we launch a map task with dummy splits or we actually use a scheduler which would launch the map tasks after creating virtual splits.

In the second case, virtual splits can not be created because chunks have to be processed together. If the chunks which need to be processed together are located on a single node, then that particular node will be the best node to schedule. In the best case, there may exist some nodes that contain all the chunks required by a particular process. In the worst case, all the chunks are located across nodes. Hence, we need to determine the node for scheduling the map task and nodes for providing the required data sets. We take into account network topology and data transfer time from each node to select the best node. The problem can be formulated as:


Independent Data blocks:
1) Map task 1 and map task 2 are scheduled on Nodes C and D.
2) Blocks 1 and 2 are assigned to the map task 1.
3) Blocks 3 and 4 are assigned to the map task 2.
4) Node D has two blocks (1 and 3), whereas Node C have blocks (2 and 4).
5) Virtual splits will be created for both the map tasks because the blocks are independent. It will improve the performance by maximizing the number of local I/O requests.

Dependent Data blocks:
1) Block 1, 2, 3 are required by a map task .
2) Node D has two blocks, whereas other nodes i.e. A, B and C have block 2.
3) Our scheduler will determine the node with minimal latency using weighted set cover approach. E.g. if Node C is selected, then block 2 will be transferred to the Node D from the Node C .

Virtual split will be created for map task 1 on Node D to process blocks 1 an 3.

Virtual split will be created for map task 2 on Node C to process blocks 2 and 4.

Figure 4.7: Scheduling when multiple chunks are assigned to a map task.

Assume there are N nodes in our system denoted as a set $C = \{node_1, node_2, ..., node_N\}$. Each I/O node contains a set of chunks, and at any given time a MapReduce application requires to access chunks that will be a subset of chunks located on I/O nodes. If there are M map tasks launched for an application then, there is a set $U = \{s_1, s_2, ..., s_M\}$ such that s_i represents a data split consisting of multiple chunks and will be processed by the i^{th} map task. As it was mentioned earlier that a NameNode returns all the nodes containing a chunk (including the replica nodes). For example, if there are three replicas then three nodes will be returned for a specific chunk. In case of multiple chunks, the total number of nodes returned is equal to three times the number of chunks. We call this set K, and K_i will correspond to all the nodes that contain chunks from the split s_i. Each node in K_i will contain at least one data chunk in the split s_i. We want to find a set ($A \subset K$) of nodes, where one of the nodes is the *primary node* to host/schedule the map task, and others are the secondary nodes and will provide the missing data to the primary node. The cost of data transfer from the secondary nodes to the primary node should be minimum.

This nodes selection problem is similar to the **weighted set cover problem** [CSR01], which models many resource selection problems. A split and a node in scheduling problem exactly corresponds to an element and a set respectively in the weighted set cover problem. The only difference is that "a split also represents a set of chunks, not a single element". Each node contains at least one of the chunks from a split. We need to find the set of nodes for each split. The weighted set cover problem has been proven to be NP-hard so that a heuristic and iterative algorithm is generally used to solve it. Similar to the weighted set cover problem, we need to assign a weight to each of the candidate nodes. The weight w corresponding to each node is the latency or data transfer time of the chunks that it does not hold itself. For each node n_i, we define: $price(n_i) = \frac{w_i(n_i)}{|Chunk\ Contibution|}$. This price will determine which nodes are to be selected.

Algorithm 4.3.2 Pseudocode for scheduling using Weighted Set Cover Problem subsection

```
Input: A set $U$ which consist of splits required by map tasks in a MapReduce Application, $U = \{s_i, s_j, ..., s_k, ...|0 < i, j, k < M\}$. A set $C$ of nodes with the information that each node $n_i$ contains data chunks, $C = \{n_1, n_2, ..., n_N\}$.
Output: Find A for all splits in $U$ such that $A_i \subseteq C$, and $A_i$ has nodes that completely cover all the chunks in split $s_i$.
Steps:
$A = \{A_i, A_j, ..., A_k, ....\}$, $A$ and all $A_x$ are empty, we will start computing each $A_x$ and add it to the set $A$.
For i is 1 to $M$, [Iterate through all the elements in set $U$]
$A_i$ = empty
1) Compute $K_i$, This will be a list of nodes with required chunks. The chunks will be located on multiple nodes.
2) Compute $w$ for all the nodes in $K_i$. $w$ is the weight assigned to each chunk as explain earlier. Also, mark the nodes which will be used for the data transfer.
3) Find the Chunk contribution of each node in $K_i$.
4) Compute the price as $\frac{w}{|Chunk\ Contibution|}$ for each node in $K_i$.
5) Sort the values in ascending order, the first value will correspond to the primary node. Add the node to $A_i$.
6) Select the secondary nodes for a primary node based on the calculations in step 2. Add the nodes to $A_i$.
7) $A \leftarrow A \cup A_i$
```

Chunk contribution is simple to quantify, it is the number of the required chunks present on a node.

The algorithm starts by retrieving all the nodes containing the chunks from a split. It then starts the iteration with an empty set, A_i, which denotes a collection of the nodes selected until the last iteration. At each iteration, the algorithm selects a node n_i, and adds it to A_i. Nodes that are added to the set A_i are considered to be covering a part of split s_i. The algorithm finishes iteration when all the chunks in the s_i are covered, and then A_i gives a set of nodes, where the first one is used as a primary node and others as secondary nodes. This algorithm is used for all map tasks, and determines the optimal nodes for all map tasks belonging to one application.

4.4 Evaluation

In the experiments, we demonstrate how MRAP API performs for the two types of access patterns as described in Section 4.3.1. The first access pattern performs matching operation on the data sets, whereas the second access pattern deals with the non-contiguous accesses. We also show the performance improvement due to data restructuring for the second access pattern.

The most challenging part of this work is to fairly evaluate MRAP framework using various data access patterns against the same patterns used in the existing MapReduce implementations. Unfortunately, most HPC analytics applications which could enunciate the benefit from MRAP still need to be developed. Also, there are no established benchmarks currently available to test our design. We have used one application from the bioinformatics domain, an open source MapReduce implementation of the "read-mapping algorithm", to evaluate the MRAP framework. This application performs sequence matching and extension of these sequences based on the given criteria and fits well with the description of the first access pattern. For the second access pattern, we use both MRAP and MapReduce to read astrophysics data in tipsy binary format that is used in many applications for operations like halo finding. In the next subsection, we describe the testbed and the benchmark setup used to generate results.

4.4.1 Testbed and Benchmarks Description

There are 47 nodes in total with Hadoop 0.20.0 installed on it. The cluster nodes configurations are shown in the Table 4.1. In our setup, the cluster's master node is used as the *NameNode* and *JobTracker*, whereas the 45 worker nodes are configured to be the *DataNodes* and *TaskTrackers*.

Table 4.1: CASS Cluster Configuration

15 Compute Nodes and 1 Head Node	
Make& Model	Dell PowerEdge 1950
CPU	2 Intel Xeon 5140, Dual Core, 2.33 GHz
RAM	4.0 GB DDR2, PC2-5300, 667 MHz
Internal HD	2 SATA 500GB (7200 RPM) or 2 SAS 147GB (15K RPM)
Network Connection	Intel Pro/1000 NIC
Operating System	Rocks 5.0 (Cent OS 5.1), Kernel:2.6.18-53.1.14.e15
31 Compute Nodes	
Make& Model	Sun V20z
CPU	2x AMD Opteron 242 @ 1.6 GHz
RAM	2GB - registered DDR1/333 SDRAM
Internal HD	1x 146GB Ultra320 SCSI HD
Network Connection	1x 10/100/1000 Ethernet connection
Operating System	Rocks 5.0 (Cent OS 5.1), Kernel:2.6.18-53.1.14.e15
Cluster Network	
Switch Make & Model	Nortel Nortel BayStack 5510-48T Gigabit Switch

The first application, CloudBurst consists of one data format conversion phase and three MapReduce phases to perform read mapping of genome sequences as shown in the Figure 4.8 a). The data conversion phase takes an ".fa" file and generates a sequence file with a format following HDFS sequence input format. It breaks the read sequence into 64KB chunks and write sequence in the form of these pairs $(id, (sequence, start\_offset, ref/read))$. The input files consist of a reference sequence file and a read sequence file. During the conversion phase, these two files are read in to generate the pairs for ".br" file. After this data conversion phase, the first MapReduce program takes these pairs in the map phase and generates *mers* such that, the resulting (key, value) pairs are $(mers, (id, position, ref/read, left\_flank, right\_flank))$. The flanks are added to the pairs in the map phase to avoid random reads in HDFS. These key, value pairs are then used in the reduce phase to generate *SharedMers* as $(read\_id, (read\_position, ref\_id, ref\_position, read\_left\_flank, read\_right\_flank, ref\_left\_flank, ref\_right\_flank))$. The second MapReduce program generates Mers per read, and group the pairs generated in the first phase by $read\_id$. The Map phase does nothing and reduce phase groups by $read\_id$.

In MRAP, this whole setup is performed such that there is one map phase and one reduce phase as shown in the Figure 4.8 b). We do not modify the conversion phase in generating the .br file. The first phase reads from both reference and read sequence files to generate the SharedMers and they are coalesced and extended in the reduce phase. The only reason we can allow a the map phase to read chunks from multiple files is because, MRAP API allows for a list of splits per map task. Essentially, each mapper reads from two input splits and generate $(read\_id, (read\_position, ref\_id, ref\_position, read\_left\_flank, read\_right\_flank, ref\_left\_flank, ref\_right\_flank))$ for the shared mers. The reduce phase aligns and extends them, resulting in a file containing every alignment of every read with at most some defined number of differences.

In the second case, we perform a non-contiguous read operation on astrophysics data set used in halo finding application, followed by the grouping of given particles. There are two files in the downloaded data set: particles_name, which contains the positions, velocities and mass of the particles. In addition to the particles_name file, an input_data file summarizing cosmology, box-size etc and halo catalogs (ascii-files), containing: mass, position and velocity in different coordinates are also provided [refg, refh].

Finally, we used a micro benchmark to perform small I/O requests using MRAP to show the significance of data restructuring. We use three configurable parameters to describe a simple strided access pattern [CCC03]. These parameters are *stripe*, *stride* and *data set size*. We show the behavior of changing the stripe size with various data sizes, where the stride was dependent on the number of processes and stripe size. The stripe size is the most important parameter for these experiments because it determines the size and the number of read requests issued per process. We write a MapReduce program to perform the same patterned read operation. In the map phase each process

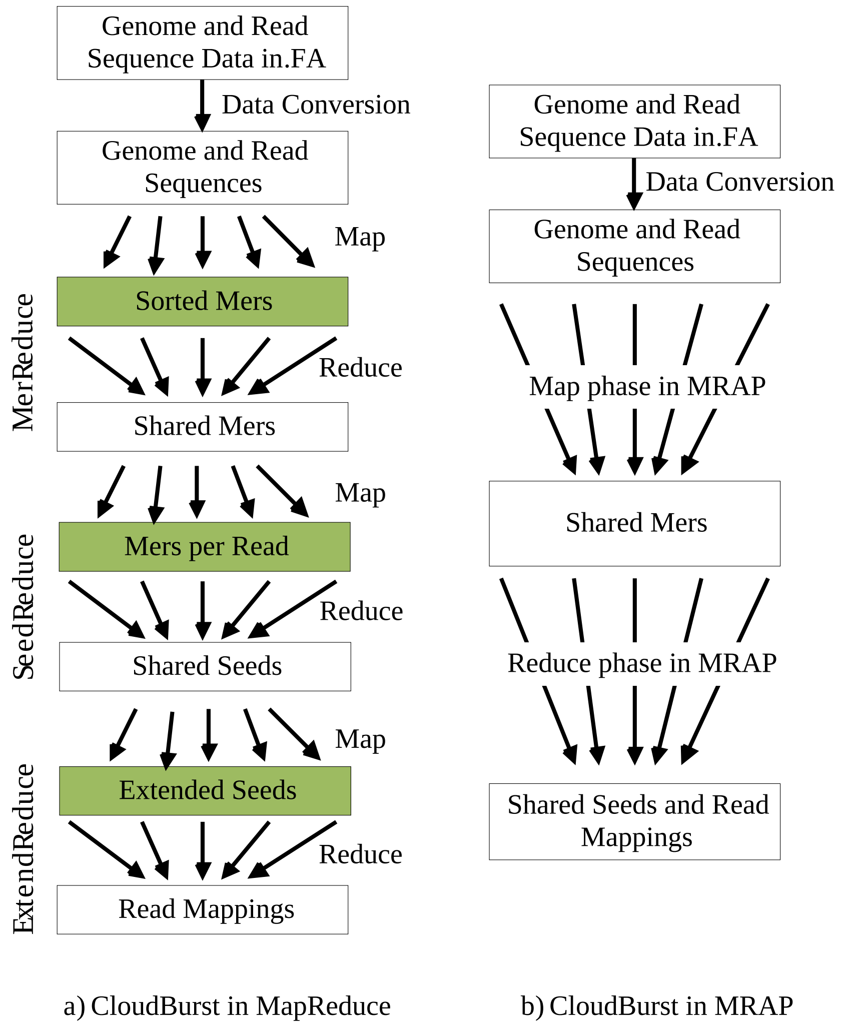


Figure 4.8: a) Overview of the Read-Mapping Algorithm using 3 MapReduce cycles. Intermediate files used internally by MapReduce are shaded [Sch09]. b) Overview of the Read-Mapping Algorithm using 1 MapReduce cycle in MRAP.

reads a contiguous chunk, and marks all the required stripes in that contiguous chunk. In the reduce phase, all the stripes required by a single process are combined together.

4.4.2 Demonstrating Performance for MRAP

Bioinformatics Sequencing Application: We compare the results of MRAP and the existing MapReduce implementation of the read-mapping algorithm. As shown in the Figure 4.8, the traditional MapReduce version of the CloudBurst algorithm requires three MapReduce applications in order to complete the required analysis as compared with one MapReduce phase required in MRAP. Because the MRAP implementation requires only one phase of I/O, we anticipate that it will significantly outperform the existing CloudBurst implementation. We first show the total number of bytes accessed by both MRAP and MapReduce implementations in the Figure 4.9. Each stacked bar shows the number of bytes read and written by MRAP and MapReduce implementation. The number of bytes read is more than the number of bytes written because reference data sequences are being merged at the end of reduce phases. The number of bytes accessed in the MRAP application are on average 47.36% less than the number of bytes accessed in the MapReduce implementation as shown in Figure 4.9. This reduced number of I/O accesses result in an overall performance improvement of upto 33%, as shown in the Figure 4.10.

We were also interested in looking at the map phase timings because each map phase in MRAP was reading its input from two different data sets. Further break down of the execution time showed that map task took $\approx 55sec$ in each phase to finish, and there were three map phases making this time equal to $2min, 45sec$. In MRAP, this time was $\approx 1min, 17sec$, because MRAP reads both read and reference sequence data in the map phase, as opposed to reading either read or reference sequence in the map phase. Hence, the time of a single map task in MRAP is greater than the time of a single map task in MapReduce, but the benefit comes from multiple stages in MapReduce based implementation.

The MRAP implementation of CloudBurst application accesses multiple chunks per map task. Multiple chunks per map task generate remote I/O requests if all the required chunks are not present on the scheduled node. In this test, $\approx 7 - 8\%$ of the total tasks launched caused remote I/O accesses, slowing down the MRAP application. Hence, supplemental performance enhancing methods, such as dynamic chunk replication or

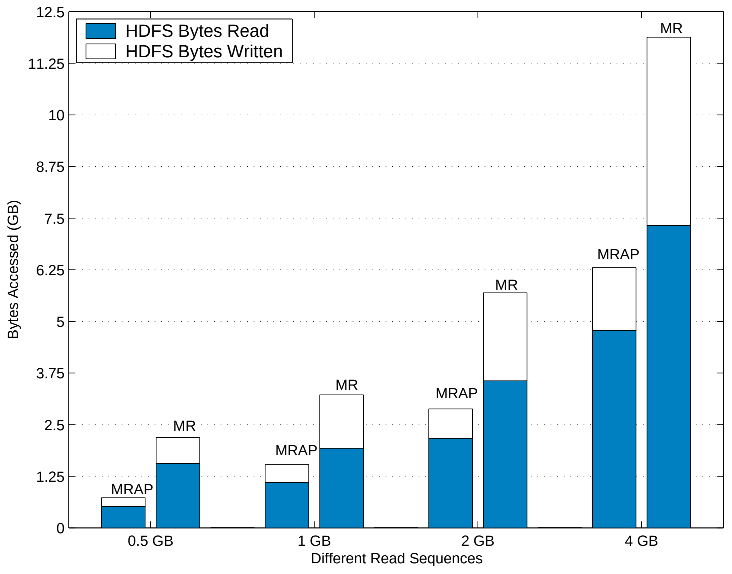


Figure 4.9: This graph compares the number of bytes accessed by the MapReduce and MRAP implementation of Read-mapping, and shows that MRAP accesses $\approx$ 47% less data.

scheduling multiple chunks, are required to be implemented with MRAP framework. Since, this particular access pattern does not access data in non-contiguous manner, data restructuring is not an appropriate optimization for it.

Astrophysics Data sets: We ran a set of experiments to demonstrate the performance of MRAP over MapReduce implementation for reading non-contiguous data sets. In this example, we use an astrophysics data set and show how MRAP deals with non-contiguous data accesses. As described earlier the halo catalog files contains 7 attributes, i.e. mass (m_p), position (x, y, z) and velocity (V_x, V_y and V_z) for different particles. In the setup, we require our test application to read these seven attributes for only one particle using the given catalogs, and scan through the values once to assign them a group based on a threshold value. Since MRAP only reads the required data sets, we consider this case where less data as compared with total data set is required by the application. The MapReduce application, reads through the data set and marks all the particles in the Map phase. The reduce phase filters out the required particle data. We assume that the data set has 5 different particles, and at each time step we have

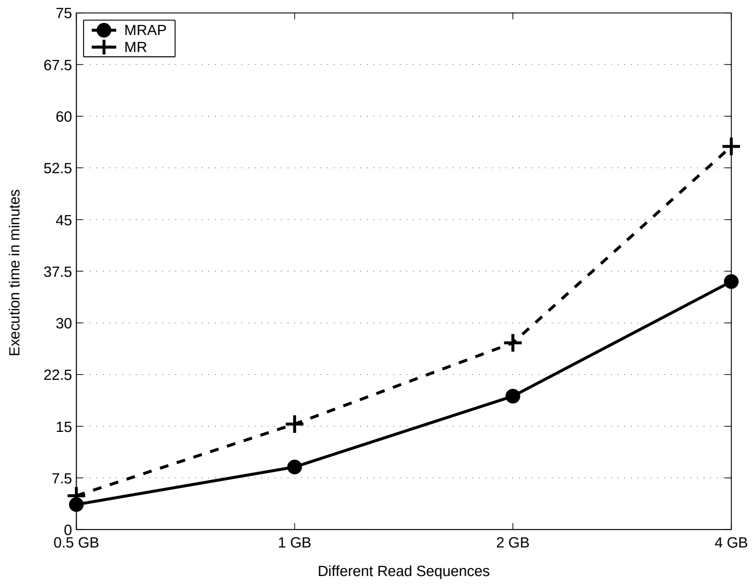


Figure 4.10: The graph compares the execution time of the Read-mapping algorithm using MRAP and MapReduce (MR).

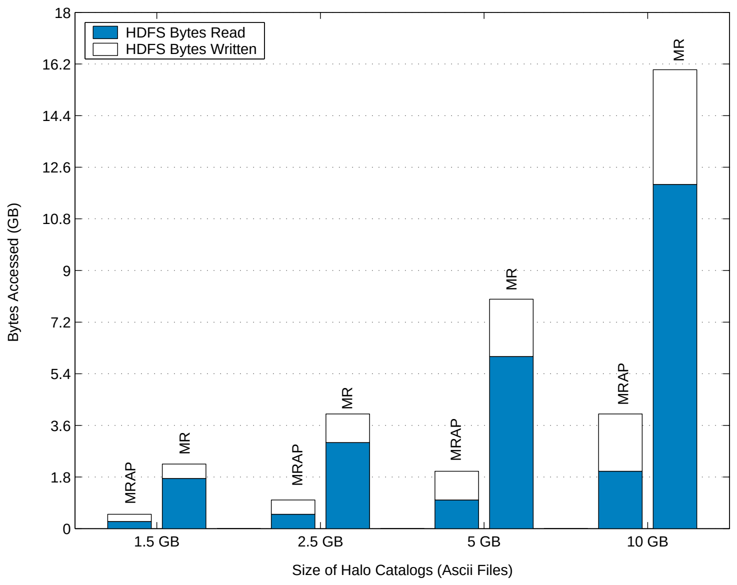


Figure 4.11: This graph shows the number of effective data bytes Read/Written using MRAP and MR. MRAP only reads the requested number of bytes from the system, as compared with MR, and shows ≈ 75% fewer bytes read in MRAP.

attributes of all the 5 particles. Essentially, the map phase reads entire data set, and the reduce phase writes only $1/5^{th}$ of the data sets. The second MapReduce application, scans through this filtered dataset and assigns the values based on the halo mass. The MRAP implementation, reads the required $1/5^{th}$ data in the map phase, and assigns the values in the reduce phase. We show the results in Figure 4.12 and 4.11 by testing this configuration on different data sets.

The application has access to data sets from $1.5 - 10$ GB in the form ≈ 6.3 MB files, and the data of interest is $\approx 0.3 - 2$ GB. Figure 4.11 shows that amount of data read and written in an MRAP application is $\approx 75\%$ less than the MapReduce implementation. The reason is that MRAP API allows to read the data in smaller regions, hence, instead of reading the full data set only data of interest is extracted in the map phase. This behavior anticipates significant improvement in the execution time of the MRAP application when compared with MapReduce. The results are shown in Figure 4.12, and they only show an improvement of $\approx 14\%$, because HDFS is not designed for small I/O requests. In the next subsection, we further elaborate on the small I/O problem, and show the results of proposed solution i.e. data restructuring.

4.4.3 Data Restructuring:

In the Section 4.4.2, we saw the performance benefits of MRAP for applications with different data access patterns, where it minimized the number of MapReduce phases. Some patterns e.g. non-contiguous accesses incur an overhead in the form of small I/O, as we demonstrate in the Figure 4.13. We used random text generated data sets of 15 GB, 30 GB, 45 GB and 60 GB in this experiment to show that small I/O degrades performance of read operations. We have used 15 map tasks, each map task reads 1, 2, 3 and 4 GB using small regions (stripe sizes) ranging from 1 KB to 1 MB. We choose this

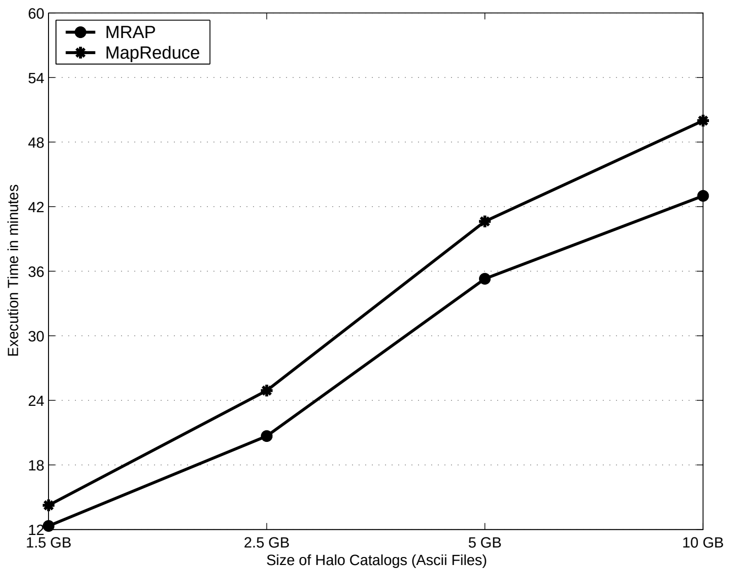


Figure 4.12: The graph compares the execution time of an I/O kernel that read Astrophysics data using MRAP and MapReduce. MRAP shows an improvement of up to $\approx 14\%$.

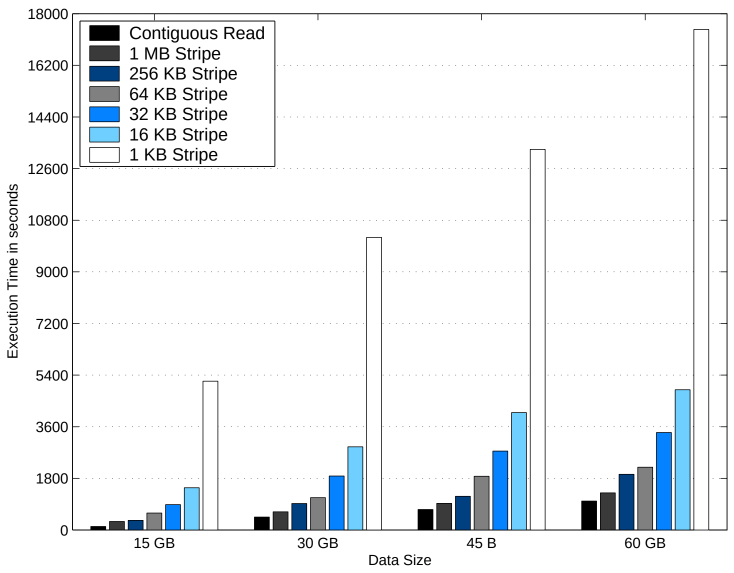


Figure 4.13: This figure shows the performance penalties due to small I/O by running a micro benchmark. The non-contiguous read operation with smaller stripe sizes has more performance penalties because of the amount of excessive data read and the number of I/O requests made.

range for stripe sizes because 1) there are many applications that store images which are as small as 1 KB [CLR09], 2) 64 KB is a very standard stripe size used in the MPI/IO applications running on PVFS2 [reft] and 3) 1 MB to 4 MB is the striping unit used in GPFS [SH02]. We have used the default chunk size of 64 MB in this experiment.

Figure 4.13 shows that smaller stripe sizes have larger performance penalties because of the number of read requests that are issued for striped accesses. 1 KB depicts the worst case scenario, where for 1 GB per map task will have 1,048,576 read calls which results in much more calls for larger data sets. Figure 4.13 also shows the time it takes to perform contiguous read for the same 1, 2, 3 and 4 GB per map task. Overall larger stripe sizes tend to perform well because as they approach the chunk size, they issue less read requests per chunk and they become contiguous within a chunk as the stripe size becomes equal to or greater than the chunk size.

A contiguous read of 1 GB with 64 MB chunks will result in reading 16 chunks. On the other hand, with 1 MB stripe size, there will be 1024 stripes in total for 1 GB set. The upper bound of the number of chunks that eventually provides these 1024 stripes is 1024. Similarly, for 1 KB stripe size, there are 65536 stripes that generate as many read requests, and may map to 65536 chunks in the worst case. In short, we could use some optimizations to improve this behavior, such as data restructuring, which are studied in this paper. We run a test by restructuring astrophysics data, and then read the data to find the groups in the given particle attributes. We restructure data such that the attributes at different time steps for each particle are stored together. In the example test case, we run the copy command and restructure data. The overhead of running copy command is shown in the Figure 4.14. After that we run the application to read the same amount of data as it was reading in Figure 4.12 and show the time it took to execute that operation. It should be noted that the amount of data read and written is the same after data restructuring. Data restructuring organizes data to minimize the number of I/O requests not the size of total requested data. In these tests, size of each request was

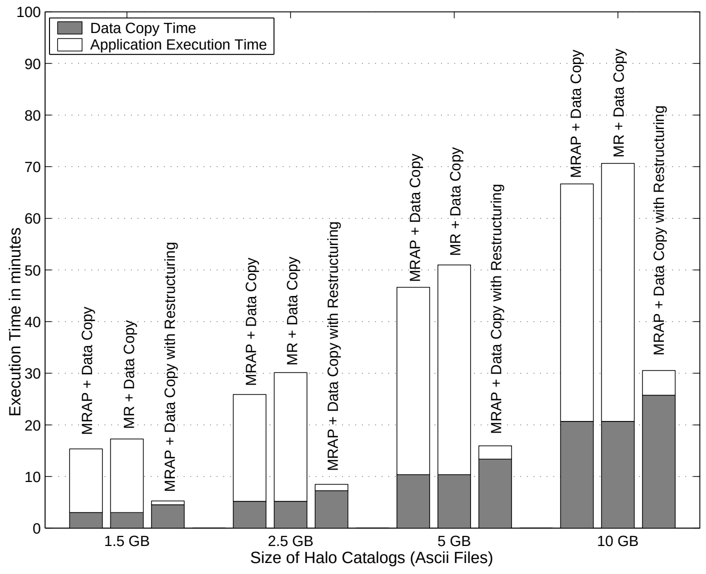


Figure 4.14: This figure shows the execution time of the I/O kernel for halo catalogs, with three implementations. MRAP API with data restructuring outperforms MR and MRAP API implementations.

$\approx 6.3MB$, and the number of I/O requests generated, for example for a $10GB$ data set is 1625. When data is restructured, 10 small regions each of 6.3 MB are packed into a single chunk of 64 MB, and reduce the number of I/O requests by 10 times. In the figure, we can see that data restructuring significantly improve the performance by up to $\approx 68.6\%$ as compared with MRAP without data restructuring and $\approx 70\%$ as compared with MapReduce. The overhead of data restructuring includes time to read the smaller regions, and put them into contiguous data chunks.

We would also like to describe that once restructured, subsequent runs with the same access patterns will perform contiguous I/O and have further performance improvement over non-restructured data. We present this case in the Figure 4.15, and show that data restructuring is useful for the applications with repeated access patterns. For the same configuration used in the Figure 4.14, we run the same application on 10 GB data set after data restructuring. It is evident from the graph, that even with the overhead as shown in Figure 4.14, data restructuring is giving promising results.

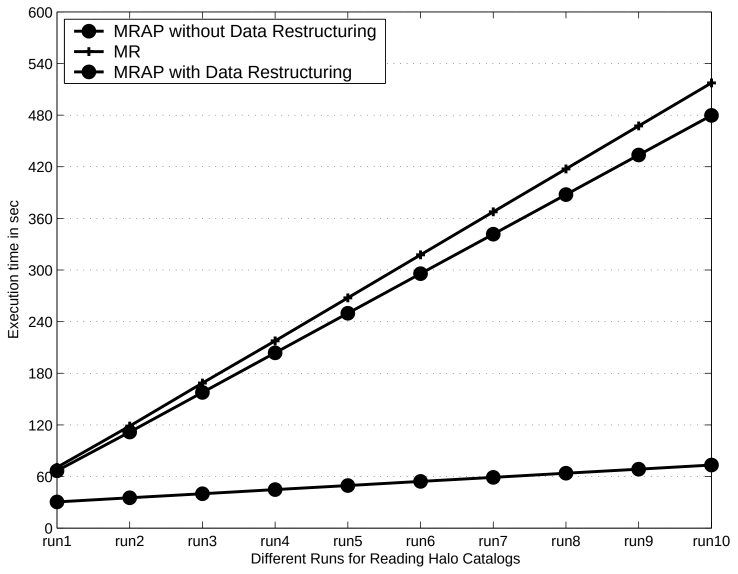


Figure 4.15: This figure shows the benefits of Data Restructuring in the long run. Same application with three different implementations (MR, MRAP, MRAP + data restructuring) is run over a period of time.

4.4.4 MRAP Scheduling

In this section, we present our initial results for scheduling. We show the % reduction in the number of remote I/O requests and % improvement in the I/O time for these two cases. We use two applications representing each of the independent and dependent chunks case. We use Adat to demonstrate the performance of creating *virtual splits*. Adat is package for carrying out post production analysis of the data generated by QCD simulations, and is written in C++ [refz]. The output of QCD (Quantum chromodynamics lattice field theory) program appears in the form of key, value pairs and are written to an XML file. ADAT is the analysis suite for the output of QCD simulation code and analyzes the XML files (in the form of key, value pairs). It provides a variety of analysis operations, we use one of these programs, i.e. the hadron_spec_strip analysis for our independent chunk workload. This analysis application analyzes multiple XML files (consisting of different Wilson Hadron measurements like forward propagation headers, forward propagation correlations, various currents, etc.) independently, making it an ap-

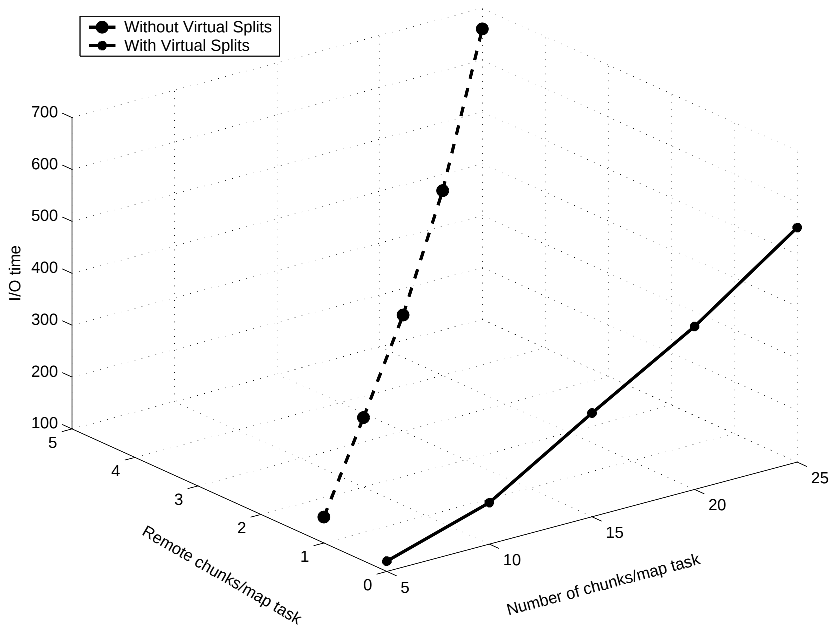


Figure 4.16: This figure shows the benefits of creating virtual splits.

propriate candidate to demonstrate the performance of virtual splits. It then generates a set of output files which are almost 1/4th of the size of input data for ADAT.

We implemented a MapReduce based I/O kernel of one of the analysis operations performed by adat suite, called *hadron_strip* in the code. The original code reads XML file(s) generated by QCD simulations, separates different measurements, analyzes them and writes specifications of each measurement in a separate file. Each XML file is read and processed sequentially and independently from each other. In the MapReduce version, we assign a set of files to each map task. Each map task then generates another set of files based on different measurements. The same measurements are combined together in the reduce phase. In the experiments, we use the I/O kernel with and without virtual splits. The maximum data set size used is $\approx$ 85 GB with a file size of $\approx$ 50 MB each. We kept the number of map tasks constant (58 map tasks, where one rack of cluster has 14 nodes and hosts 2 map tasks each, and the second rack has 30 nodes and hosts one map task each), but varied the number of chunks (files in our case) processed by each map task from 5 to 25.

Figure 4.16 shows the I/O time (in sec) with virtual splits. It can be seen that for smaller data size, starting with only 5 chunks per map task there is no significant difference in time *i.e.* only 30 sec which is very small. But as the assigned split size increases, more chunks are assigned to a map task. When more chunks are assigned to a map task, it means that there are more chances of remote I/O requests. Also for larger data sets, the chunks will be evenly distributed among all the nodes. For large size splits, we expect more remote I/O requests but overall ratio of remote I/O requests may be the same. For example, when 20 chunks are assigned to a map task, chances of all 20 chunks being on the same node is less likely as compared with the case when only 5 chunks are assigned to a map task.

As we can see in the Figure 4.16, the difference in the I/O time increases with the increase in the number of chunks assigned to a map task. It also causes more remote I/O requests. In our experiments, for 85 GB test 20% of the tasks were performing remote I/O. On average, each task was requesting $\approx 4 - 5$ chunks remotely. By creating virtual splits, we get 100% of local I/O tasks but there is an overhead to determine the virtual splits. It includes the cost of determining which blocks of a particular file(s) are residing locally on the node. On average, determining a chunk location and if it is a local chunk takes $\approx 2 - 4$ sec. Our experiments show that, on average for 25 chunks it took ≈ 85.8 sec. We also show the variation in the number of chunks transferred and I/O time with the total number of chunks assigned to a map task in the Figure 4.16. The best case is no transfer, which we achieve by using the virtual splits. This set of results show that virtual splits is an effective way of minimizing the data transfers, and it shows performance improvement of 18% for our experimental setup.

CHAPTER 5
RELATED WORK

In this chapter, we will discuss the research work done in parallel programming abstractions especially for I/O intensive HPC applications. We will describe the approaches used in programming models to improve performance and programmer productivity.

Performance and Programmer Productivity: In MPI-IO, several techniques to improve the performance of read/write and locking functions have been developed. MPI-IO application patterns often result in small I/O requests. To improve the small I/O problem, many approaches have been adopted in HPC community particularly for the applications using MPI/MPI-IO. These techniques are supported both at the file system and programming abstraction level. Data sieving allows the processes to read excessive contiguous data set in a given range instead of making small I/O requests to multiple non-contiguous chunks. The limitation of this approach is that each process reads excessive amount of data. Similarly, collective I/O also allows a process to read a contiguous chunk of data but then using MPI's communication framework, it redistributes the data among multiple processes as required by them [TGL99a]. Another implementation of collective I/O is View-based [BIS08], and communicates file views instead of the exact offset/length pairs and significantly reduces the communication cost. We do not propose a new optimization to improve performance but rather a way for an automated combination of the existing optimizations, if a read/write call is used repeatedly.

Many people feel that programming with MPI is too hard and they can prove it, while others believe MPI is fine and they can also prove it [SHP08]. Programmers use MPI because of its performance, completeness and ubiquity [refo]. Performance has always been the ultimate goal in HPC, and programmer productivity has always been overlooked. Conceptual programming effort and empirical data analysis have been used to measure programmer productivity as in [HCS05], [CYZ04], [PG08]. These approaches study and analyze the programmer productivity of parallel programming languages in HPC community and do not study the ways of improving the programmer productivity for HPC programmers.

MPI-IO Atomicity: Researchers have also contributed to provide atomicity semantics both at application and file system level. Non-contiguous access patterns and overlapping I/O patterns [CCC03, CCL02, LCC03] have been widely studied and the customized locking schemes, process rank ordering and handshaking have been proposed. List locks and datatype locks [ACT06] [CLC07] have maximum concurrency, but they acquire and maintain locks for all regions accessed by a process. We provide conflict detection to find the overlaps before lock requests are issued. The conflict check facilitates the locking mechanism by providing a decision where locks are necessary to guarantee atomicity.

Large scale data processing frameworks are being developed because of the information retrieval for web scale computing. Many systems like MapReduce [Dea06, DG04], Pig [ORS08], Hadoop [reff], Swift [RZD07, ZHC07], Dryad [IBY07, IY09, YIF08] and many more abstractions are there that allow large scale data processing. Dryad has been evaluated for HPC analytics applications in [EGF09]. However, our approach is based on MapReduce, which is well-suited for the data parallel applications where data dependence does not exist and applications run on a shared-nothing architecture.

In the Chapter 3, we present a case of using MapReduce for scientific analytics applications. Scientific applications use high level APIs like NetCDF [refr], HDF5 [refi] and their parallel variants [LLC03, refs] to describe the complex data formats. These APIs and libraries work as an abstraction with the existing most commonly used MPI framework by utilizing the *MPI File views* [ref97]. NetCDF (Network Common Data

Form) is a set of software libraries and machine-independent data formats that support the creation, access, and sharing of *array-oriented scientific data* [refr]. The data model represented by HDF5 support very complex data objects, metadata and a completely portable file format with no limit on the number or size of data objects in the collection [refi]. We develop ways of specifying access patterns similar to *MPI datatypes* and *MPI File views* within MapReduce.

Data Restructuring: To improve small I/O, in large-scale systems with thousand of processes, collective I/O with its two-phase is used for abstractions like MPI-IO [TGL99a]. Other approaches are for checkpointing applications like PLFS, which adds a layer between the application and the file systems and re-maps an application's write access pattern to be optimized for the underlying file system [BGG09]. DPFS provides striping mechanisms that divides a file into small pieces and distributes them across multiple storage devices for parallel data access [SC01]. Our approach of data restructuring is significantly different from these approaches because we re-organize data such that processes are not required to communicate with each other, and maintain shared-nothing architecture for scalability.

Scheduling: Job and task scheduling in distributed systems has been researched widely. The ultimate goal in all scheduling schemes is managing resources among multiple jobs, while maintaining the performance of individual tasks and overall jobs. In large scale distributed systems like Grids, the prominent scheduling schemes for data-intensive applications are *task-centric* and worker-centric. Task-centric approach shows improved performance by reusing the data on Grid computing sites but results in unbalanced tasks because it does not consider the load on each individual worker. Decoupling data and computation as proposed in [RF02], evaluates various task and data scheduling mechanisms. They show the best results are obtained when a task is scheduled to a site that has its input data already in place, combined with proactive replication of a popular input data set to a random/least-loaded site. Many other systems like Falkon aims to enable the rapid and efficient execution of many tasks on large compute clusters, and to improve application performance and scalability using novel data management techniques [RZD07]. They provide a scheduling mechanism to copy the data to the resources acquired dynamically for a particular application and have many variants of this approach with performance improvements [RZF08, RFZ09]. These ensure that data is cached/diffused to the sites which will be selected for the task execution.

There are other schemes as well that considers localities like in storage affinity [SCB04], spatial clustering [MAW06], dynamic scheduling [VVY04] etc. In worker-centric approach the workers are selected that have enough compute resources to execute a job and then data is transferred to the worker nodes. A worker-centric approach which leads to tremendous amount of data transfers for data-intensive applications because data has to be moved to the resources that are chosen for executing the application [KMG07]. Our approach is different because we are using clusters in a tightly coupled environment as compared to a Grid, and the required data stays in the cluster and will not be moved from the resources once an application finishes and resources are freed up.

In the systems where compute and data resources are co-located, these systems deal with two levels of resource sharing one is sharing within an application and other is sharing among applications [refe, refc, IPC09]. For a single application environment, it uses "data-locality" aware scheduling based on the location of a single chunk [reff]. Our scheduling schemes are derived from this architecture but are strongly impacted by the application behavior of data-intensive HPC applications *i.e.* we deal with complex access patterns that requires the scheduling of multiple chunks per map task. Improving MapReduce performance in heterogeneous clusters involves a scheduling mechanism, Longest Approximate Time to End, for speculative tasks that processes single chunks [ZKJ08]. Our focus is on the regular map tasks with multiple chunks, and their LATE approach can be added to our scheduling scheme to further improve the performance.

Porting MapReduce to HPC File systems for HPC analytics Some works include using MapReduce on existing parallel file systems, example systems include a PVFS2 shim layer [refu, refy], GPFS [refn, AGP09], MapReduce CGL [EPF08], and MapReduce/MPI [refm]. These systems decouple MapReduce from HDFS with an effort to provide the same functionality using different file systems. PVFS2 shim layer does provide for basic MapReduce functionality, but it does not provide the functionality of specifying access patterns on MapReduce. Another approach with GPFS modifies the file block size by providing meta blocks that are compatible with MapReduce processing block size [AGP09]. Both these approaches decouple MapReduce from HDFS, and try to snap a traditional parallel file system onto a data-intensive framework, whereas our work provides for a data-intensive file system for a compute-intensive framework.

Our approach is significantly different from this work, we provide data semantics and required optimizations to the MapReduce framework. Some of HPC data analytics match well with MapReduce programming style and using MapReduce with existing HPC file systems is an attractive approach but requires modification to the file system for supporting larger blocks, etc. There are other applications that need additional API support and optimizations to use MapReduce for high performance. Our goal is to support different HPC data analytics applications with data access patterns using enhanced APIs in MapReduce rather than porting MapReduce to existing HPC file system.

Some other approaches like CGL MapReduce [EPF08] also propose a solution to improve the performance of scientific data analysis applications developed in MapReduce. However, their approach is fundamentally different from our work. In CGL MapReduce they do not address decreasing the number of MapReduce phases, rather they mitigate the file read/write issue by providing an external mechanism to keep read file data persistent across multiple map reduce jobs. Their approach does not work with HDFS, and relies on a NFS mounted source.

MPI/MR implementation is based on C++ and allows for interprocess communication via MPI. By providing a C based MapReduce, MR-MPI allows for control of memory allocation in a MapReduce job [refm]. We provide support for MPI datatypes on MapReduce, and do not use MPI communication functions. Zazen [TRM10] provides a new data access method to overcome the I/O bottleneck for analytics applications after simulation data has been obtained. They use an analysis cluster, but cache a copy of simulation output files on local disks of analysis cluster and use a novel task-assignment protocol to co-locate data access with computation. Our approach also use an analysis cluster, exploits data locality but also supports HPC data access patterns.

CHAPTER 6
CONCLUSION AND FUTURE WORK

This proposed dissertation work is based on two parallel programming frameworks: MPI/MPI-IO and MapReduce. We have developed a reduced function set abstraction for MPI-IO and an HPC data access pattern aware MapReduce based framework. In the following subsections, we summarize the findings of our current work and discuss the future work.

6.1 RFSA - Conclusion

We have implemented a selection algorithm that transparently detects which blocking read/write function to use given a particular file view. The implementation is provided as a unified read/write function in MPI-IO. Our programmer productivity results show that reducing the number of functions provided to the application programmer is an effective way of improving the programmer productivity (35.7%). Our results show that for a selected set of benchmarks, which perform multiple read/write operations, the selection procedure performs 17% better than the function that performs best for a given particular case.

We have proposed a scheme to perform conflict detection using file views, and introduce lock free independent write operations if there are no conflicts. We have implemented our algorithm in ROMIO. In MPI-IO applications atomicity guarantees rely on the file system locks. Our Conflict detection algorithm is able to extract overlapping regions from the file views (for independent operations) created by MPI-IO application with a minimal overhead. It paves the way to the lock-free and scalable approaches of MPI-IO atomicity support.

6.2 RFSA - Future Work

- We have abstracted the blocking I/O calls, in the future we would like to extend this work to non-blocking function calls, and open/close function calls. We would also like to study the applicability of RFSA to the communication functions provided by MPI.
- We have tested the conflict detection algorithm on a cluster with 16 nodes, the scalability testing of this algorithm will be studied. The current algorithm relies on the MPI collective communication calls. The number of messages to be exchanged for detecting overlaps increases with the number of processes and file size. We need to extend the current algorithm to create different communication groups. As a result, we can distribute the conflict detection task among a group of nodes rather than using each node as a conflict detector for itself.

6.3 MRAP - Conclusion

We have developed an extended MapReduce framework to allow users to specify data semantics for HPC data analytics applications. Our approach reduces the overhead of writing multiple MapReduce programs to pre-process data before its analysis. We provide functions and templates to specify the sequence matching, and strided (non-contiguous) accesses in reading different data sets, such that access patterns are directly specified in the map phase. For experimentation, we ran a real application from bioinformatics and an astrophysics I/O kernel. Our results show a maximum throughput improvement up to 33%. We also studied the performance penalties due to the non-contiguous accesses (small I/O requests) and implemented data restructuring to improve the performance. Data restructuring uses a user-defined configuration file and reorganizes data in a file such that all non-contiguous chunks are stored contiguously, and show a performance gain of up to 70% for the astrophysics data set. These small I/O requests also map to multiple chunks that are assigned to a map task, and require schemes to improve performance by selecting optimal nodes for scheduling map tasks on the basis of multiple chunk locations. Our improved scheduling mechanism shows a performance improvement of up to 18%.

6.4 MRAP - Future Work

- We would like to implement more HPC analytics applications using MRAP. We will also add more access patterns and support of formats like netCDF, HDF5 and ADIOS like interfaces to MapReduce and make it more expressive for scientists.
- Data restructuring is implemented as a sequential HDFS copy operation. The parallel version of this copy operation is only available for HDFS to HDFS copy operation. We would like to implement a parallel copy operation with data restructuring to further improve the performance of this operation. (Parallel copy operation from a remote storage to HDFS is not available in the HDFS API.)
- We would also like to study the scalability related issues of MRAP, because the purpose of using MapReduce framework is to keep the scalability and resiliency offered by the framework. Our initial tests use a 45 node cluster, we will run some more experiments on large scale clusters in future.
- A multiuser environment with HPC analytics applications imposes more challenges for scheduling. Our first effort was to optimize single application, and we will extend our work to multiuser environments.

LIST OF REFERENCES

[ACT06] P. M. Aarestad, A. Ching, G. K. Thiruvathukal, and A. N. Choudhary. "Scalable Approaches for Supporting MPI-IO Atomicity." In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, pp. 35–42, Washington, DC, USA, 2006. IEEE Computer Society.

[AGP09] R. Ananthanarayanan, K. Gupta, P. Pandey, H. Pucha, P. Sarkar, M. Shah, and R. Tewari. "Cloud Analytics: Do We Really Need to Reinvent the Storage Stack?" In *HotCloud '09: Workshop on Hot Topics in Cloud Computing in conjunction with the 2009 USENIX Annual Technical Conference*, 2009.

[BGG09] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. "PLFS: A Checkpoint Filesystem for Parallel Applications." In *Supercomputing, 2009 ACM/IEEE Conference*, Nov. 2009.

[BIS08] J. G. Blas, F. Isaila, D. 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)*, pp. 409–416, Washington, DC, USA, 2008. IEEE Computer Society.

[Bor] D. Borthaku. "The Hadoop Distributed File System: Architecture and Design.".

[CCC03] A. Ching, A. Choudhary, K. Coloma, W. keng Liao, R. Ross, and W. Gropp. "Noncontiguous I/O Accesses Through MPI-IO." In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, p. 104, Washington, DC, USA, 2003. IEEE Computer Society.

[CCL02] A. Ching, A. Choudhary, W. keng Liao, R. Ross, and W. Gropp. "Noncontiguous I/O through PVFS." In *CLUSTER '02: Proceedings of the IEEE International Conference on Cluster Computing*, p. 405, Washington, DC, USA, 2002. IEEE Computer Society.

[CFL06] A. Ching, W. Feng, H. Lin, X. Ma, and A. Choudhary. "Exploring I/O Strategies for Parallel Sequence-Search Tools with S3aSim." *hpdc*, **0**:229–240, 2006.

[CLC07] A. Ching, W. keng Liao, A. Choudhary, R. Ross, and L. Ward. "Noncontiguous locking techniques for parallel file systems." In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pp. 1–12, New York, NY, USA, 2007. ACM.

[CLR09] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig. "Small-File Access in Parallel File Systems." In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, April 2009.

[CSR01] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[CYZ04] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi. "Productivity Analysis of the UPC Language." In *18th IEEE International Parallel and Distributed Processing Symposium (IPDPS04)*, 2004.

[Dea06] J. Dean. "Experiences with MapReduce, an abstraction for large-scale computation." In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pp. 1–1, New York, NY, USA, 2006. ACM.

[DG04] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters." In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pp. 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[EGF09] J. Ekanayake, T. Gunarathne, G. Fox, A. S. Balkir, C. Poulain, N. Araujo, and R. Barga. "DryadLINQ for Scientific Analyses." In *E-SCIENCE '09: Proceedings of the 2009 Fifth IEEE International Conference on e-Science*, pp. 329–336, Washington, DC, USA, 2009. IEEE Computer Society.

[EPF08] J. Ekanayake, S. Pallickara, and G. Fox. "MapReduce for Data Intensive Scientific Analyses." In *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pp. 277–284, 2008.

[GGL03] S. Ghemawat, H. Gobioff, and S.-T. Leung. "The Google File System." In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp. 29–43, New York, NY, USA, 2003. ACM.

[GTL99] W. Gropp, R. Thakur, and E. Lusk. *Using MPI-2: Advanced Features of the Message Passing Interface.* MIT Press, Cambridge, MA, USA, 1999.

[HCS05] L. Hochstein, J. Carver, F. Shull, S. Asgari, and V. Basili. "Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers." In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, p. 35, Washington, DC, USA, 2005. IEEE Computer Society.

[IBY07] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. "Dryad: distributed data-parallel programs from sequential building blocks." *EuroSys '07: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pp. 59–72, 2007.

[IPC09] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. "Quincy: Fair Scheduling for Distributed Computing Clusters." In *SOSP '09: Proceedings of 22nd ACM Symposium on Operating Systems Principles*. ACM, 2009.

[IY09] M. Isard and Y. Yu. "Distributed data-parallel computing using a high-level programming language." In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pp. 987–994, New York, NY, USA, 2009. ACM.

[KMG07] S. Y. Ko, R. Morales, and I. Gupta. "New worker-centric scheduling strategies for data-intensive grid applications." In *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pp. 121–142, New York, NY, USA, 2007. Springer-Verlag New York, Inc.

[KNG09] Y. Kwon1, D. Nunley2, J. P. Gardner3, M. Balazinska4, B. Howe5, and S. Loebman6. "Scalable clustering algorithm for N-body simulations in a shared-nothing cluster." Technical report, University of Washington, Seattle, WA, 2009.

[LCC03] W. keng Liao, A. Choudhary, K. Coloma, G. K. Thiruvathukal, L. Ward, E. Russell, and N. Pundit. "Scalable Implementations of MPI Atomicity for Concurrent Overlapping I/O." *Parallel Processing, International Conference on*, **0**:239, 2003.

[LLC03] J. Li, W. keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. "Parallel netCDF: A High-Performance Scientific I/O Interface." In *Supercomputing, 2003 ACM/IEEE Conference*, pp. 39–39, Nov. 2003.

[MAW06] L. Meyer, J. Annis, M. Wilde, M. Mattoso, and I. Foster. "Planning spatial workflows to optimize grid performance." In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pp. 786–790, New York, NY, USA, 2006. ACM.

[MTF08] A. Matsunaga, M. Tsugawa, and J. Fortes. "CloudBLAST: Combining MapReduce and Virtualization on Distributed Resources for Bioinformatics Applications." In *ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pp. 222–229, Washington, DC, USA, 2008. IEEE Computer Society.

[ORS08] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. "Pig latin: a not-so-foreign language for data processing." In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1099–1110, New York, NY, USA, 2008. ACM.

[PG08] I. Patel and J. R. Gilbert. "An Empirical Study of the Performance and Productivity of two Parallel Programming Models." In *IPDPS*, pp. 1–7, 2008.

[PGK06] S. Pervez, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. D. Gropp. "Formal Verification of Programs That Use MPI One-Sided Communication." In *PVM/MPI*, pp. 30–39, 2006.

[refa] "Advanced Particle Simulation for Computational Cosmology and Beam Physics: Cosmology. `http://t8web.lanl.gov/people/salman/icp/cos.html`.".

[refb] "Blue Waters, Sustained Petascale Computing. `http://www.ncsa.illinois.edu/BlueWaters/`.".

[refc] "Capacity Scheduler. `http://hadoop.apache.org/core/docs/r0.20.0/capacity_scheduler.html`.".

[refd] "Data- Driven Computational Science and Future Architectures at the Pittsburgh Supercomputing Center. `http://www.cisl.ucar.edu/dir/09Seminars/roskies20090130.ppt`.".

[refe] "Fair Scheduler. `http://hadoop.apache.org/core/docs/r0.20.0/fair_scheduler.html`.".

[reff] "HADOOP. `http://hadoop.apache.org/core/`.".

[refg] "Halo Catalogs. http://t8web.lanl.gov/people/heitmann/arxiv/codes.html.".

[refh] "Hashed Oct-Tree . http://t8web.lanl.gov/people/salman/icp/hot.html.".

[refi] "HDF5. http://www.hdfgroup.org/HDF5/.".

[refj] "HDFS Metadata. https://issues.apache.org/jira/browse/HADOOP-1687.".

[refk] "Implementing WebGIS on Hadoop: A Case Study of Improving Small File IO Performance on HDFS. http://www.cluster2009.org/47.pdf.".

[refl] "Lustre Filesystem. http://www.lustre.org/.".

[refm] "MapReduce-MPI Library. http://www.sandia.gov/~sjplimp/mapreduce.html.".

[refn] "MapReduce on GPFS. http://www.usenix.org/events/fast09/wips_posters/ananthanarayanan_wip.pdf.".

[refo] "MPI and High Productivity Programming. http://www.cs.uiuc.edu/homes/wgropp/bib/talks/tdata/2007/mpiandhpl-osu.pdf.".

[refp] "MPI-Tile IO. http://www.mcs.anl.gov/~thakur/pio-benchmarks.html.".

[refq] "National Energy Research Scientific Computing Center. http://www.nersc.gov/.".

[refr] "netCDF. http://www.unidata.ucar.edu/software/netcdf/.".

[refs] "Parallel HDF5. http://www.hdfgroup.org/HDF5/PHDF5/.".

[reft] "Parallel Virtual File System version 2. http://www.pvfs.org/.".

[refu] "PVFS2 Shim Layer. http://institute.lanl.gov/isti/irhpit/projects/hdfspvfs.pdf.".

[refv] "Relativistic Heavy Ion Collider. http://www.bnl.gov/rhic.".

[refw] "Roadrunner. http://lanl.gov/news/index.php/fuseaction/home.story/story_id/13602.".

[refx] "Roadrunner. http://www.lanl.gov/roadrunner/.".

[refy] "http://cmulargescalelunch.kyloo.net/files/hdfspvfs-pdlslides.ppt.".

[refz] "US Lattice Quantum Chromodynamics. http://www.usqcd.org/usqcd-software/.".

[ref97] "MPI-2: Extensions to the message-passing Interface. http://www.mpi-forum.org/docs/.", July 1997.

[RF02] K. Ranganathan and I. Foster. "Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications." In *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, p. 352, Washington, DC, USA, 2002. IEEE Computer Society.

[RFZ09] I. Raicu, I. T. Foster, Y. Zhao, P. Little, C. M. Moretti, A. Chaudhary, and D. Thain. "The quest for scalable support of data-intensive workloads in distributed systems." In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, pp. 207–216, New York, NY, USA, 2009. ACM.

[RLG05] R. B. Ross, R. Latham, W. Gropp, R. Thakur, and B. R. Toonen. "Implementing MPI-IO atomic mode without file system support." In *CCGRID*, pp. 1135–1142, 2005.

[RZD07] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. "Falkon: a Fast and Light-weight tasK executiON framework." In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pp. 1–12, New York, NY, USA, 2007. ACM.

[RZF08] I. Raicu, Y. Zhao, I. T. Foster, and A. Szalay. "Accelerating large-scale data exploration through data diffusion." In *DADC '08: Proceedings of the 2008 international workshop on Data-aware distributed computing*, pp. 9–18, New York, NY, USA, 2008. ACM.

[SC01] X. Shen and A. Choudhary. "DPFS: A Distributed Parallel File System." *Parallel Processing, International Conference on*, **0**:0533, 2001.

[SCB04] E. Santos-Neto, W. Cirne, F. Brasileiro, A. Lima, R. Lima, and C. Grande. "Exploiting Replication and Data Reuse to Efficiently Schedule Data-intensive Applications on Grids." In *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 210–232, 2004.

[Sch09] M. C. Schatz. "CloudBurst: Highly Sensitive Read Mapping with MapReduce." *Bioinformatics*, p. 236, April 2009.

[Seb02] R. W. Sebesta. *Concepts of Programming Languages (Fifth Edition)*. Addison-Wesley Publishing, 2002.

[SH02] F. Schmuck and R. Haskin. "GPFS: A Shared-Disk File System for Large Computing Clusters." In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, p. 19, Berkeley, CA, USA, 2002. USENIX Association.

[SHP08] S. Spetka, H. Hadzimujic, S. Peek, and C. Flynn. "High Productivity Languages for Parallel Programming Compared to MPI." *HPCMP Users Group Conference*, **0**:413–417, 2008.

[SKT00] A. S. Szalay, P. Z. Kunszt, A. Thakar, J. Gray, D. Slutz, and R. J. Brunner. "Designing and mining multi-terabyte astronomy archives: the Sloan Digital Sky Survey." *SIGMOD Rec.*, **29**(2):451–462, 2000.

[SMW10] S. Sehrish, G. Mackey, J. Wang, and J. Bent. "MRAP - A Novel MapReduce-based framework to support HPC Analytics Applications with Access Patterns. ." In *ACM High Performance Distributed Computing*, June 2010.

[SW09] S. Sehrish and J. Wang. "Smart Read/Write for MPI-IO." In *The 14th International Workshop on High-Level Parallel Programming Models and Supportive Environments(HIPS'09), in conjunction with the 23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009.

[SW10] S. Sehrish and J. Wang. "Reduced Function Set Abstraction (RFSA) for MPI-IO." In *Journal of Supercomputing*, 2010.

[SWJ05] V. Springel, S. D. M. White, A. Jenkins, C. S. Frenk, N. Yoshida, L. Gao, J. Navarro, R. Thacker, D. Croton, J. Helly, J. A. Peacock, S. Cole, P. Thomas, H. Couchman, A. Evrard, J. Colberg, and F. Pearce. "Simulations of the formation, evolution and clustering of galaxies and quasars." *Nature*, **435**(70422):629–636, June 2005.

[SWT09] S. Sehrish, J. Wang, and R. Thakur. "A Conflict Detection Algorithm to Minimize Locking for MPI-IO Atomicity." In *Proceedings of the EuroPVM/MPI 2009*, September 2009.

[TGL99a] R. Thakur, W. Gropp, and E. 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*, p. 182, Washington, DC, USA, 1999. IEEE Computer Society.

[TGL99b] R. Thakur, W. Gropp, and E. Lusk. "On implementing MPI-IO portably and with high performance." In *IOPADS '99: Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pp. 23–32, New York, NY, USA, 1999. ACM.

[TRL05] R. Thakur, R. B. Ross, and R. Latham. "Implementing Byte-Range Locks Using MPI One-Sided Communication." In *PVM/MPI*, pp. 119–128, 2005.

[TRM10] T. Tu, C. A. Rendleman, P. J. Miller, F. D. Sacerdoti, R. O. Dror, and D. E. Shaw. "Accelerating Parallel Analysis of Scientific Simulation Data via Zazen." In *USENIX conference on File and Storage Technologies*, pp. 129–142, 2010.

[VVY04] S. Viswanathan, B. Veeravalli, D. Yu, and T. G. Robertazzi. "Design and Analysis of a Dynamic Scheduling Strategy with Resource Estimation for Large-Scale Grid Systems." In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pp. 163–170, Washington, DC, USA, 2004. IEEE Computer Society.

[WUA08] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. "Scalable Performance of the Panasas Parallel File System." In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pp. 1–17, Berkeley, CA, USA, 2008. USENIX Association.

[YIF08] Y. Yu, M. Isard, D. Fetterly, M. Budiu, lfar Erlingsson, P. K. Gunda, and J. Currey. "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language." In R. Draves and R. van Renesse, editors, *OSDI*, pp. 1–14. USENIX Association, 2008.

[ZHC07] Y. Zhao, M. Hategan, B. Clifford, I. T. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. "Swift: Fast, Reliable, Loosely Coupled Parallel Computation." In *IEEE SCW*, pp. 199–206, 2007.

[ZKJ08] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. "Improving MapReduce Performance in Heterogeneous Environments." In *8th Symposium on Operating Systems Design and Implementation (OSDI'08)*, pp. 29–42, 2008.