MASS HDFS: Multi-Agent Spatial Simulation Hadoop Distributed File System

Yun-Ming Shih

A Capstone Project
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2018

Committee:

Munehiro Fukuda, Committee Chair

Michael Stiber, Committee Member

Johnny Lin, Committee Member

Program Authorized to Offer Degree:
Department of Computing and Software Systems

University of Washington

Abstract

MASS HDFS: Multi-Agent Spatial Simulation Hadoop Distributed File System

Yun-Ming Shih

Chair of the Supervisory Committee:
Professor Munehiro Fukuda
Computing & Software Systems

Parallel computing has been widely used for processing big data, but most systems only handle simple structured data types and do not support complex structured data. As many scientific data are highly structured, such as climatological data, the requirements have to be addressed. UWCA is a web-serviced climate analysis application that uses Multi-Agent Spatial Simulation (MASS), a parallelization library created by Distributed Systems Laboratory at the University of Washington Bothell's Computing & Software Systems Division (DSLab), for its computations. Without a proper file handling system, the master node becomes a bottleneck, causing slow performances. Frameworks have been developed to handle structured data in parallel. ROMIO gives precise control to structured data, but the extensive features make the system complicated. SciHadoop gives a simple computing model, but requires converting science data to text type for processing. This inspired DSLab to develop the MASS Parallel I/O layer for parallel file reading.

Our proposed system, MASS HDFS, is a MASS Parallel I/O layer that uses HDFS for file distribution and has additional write functionality. It aims to provide the capability to handle structured data while maintaining simplicity of usage. MASS HDFS can read data into distributed arrays without introducing a single point of bottleneck or data conversion. The performance evaluation shows that for a 200MB file with up to eight computing nodes, MASS HDFS spends 0.6 seconds on file open and 0.09 seconds on file read. Opening a 50MB

file using 50 million Place elements takes 4.5 seconds. Using 12.5 million Place elements to open a 50MB file takes about 1.5 times longer. Reading data with 50 million Place elements can be done within 15 seconds, and 18 times faster using 12.5 million Place elements. Our project made parallel I/O possible, as well as demonstrating the potentiality of processing data on a per-data-item scale using four computing nodes.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank the chair of my capstone committee, Prof. Fukuda, for helping me throughout the capstone process. This project would not have been possible without his guidance, assistance, and hard work.

I would also like to thank the other members of my project committee, Prof. Stiber and Prof. Lin, for their meticulous work in reviewing my progress reports.

Finally and most importantly, I would like to thank my parents and sister for their full support and always believing in me. I would not have been able to complete the program without them.

Chapter 1

INTRODUCTION

An increasing need for data processing is pushing development of parallelized big data analysis. Most applications deal with simple structured data like CSV and SQL. Scientific data, such as climate analysis data, has a complex structure which is not well supported. To expand the use of parallelized big data analysis within scientific fields, the DSLab research group proposed a multi-agent-based method that can process multi-dimensional NetCDF data. They demonstrated its practicality by incorporating the University of Washington Climate Analysis (UWCA) web application, which uses NetCDF software with the MASS Java Library.

1.1 MASS Java Library

MASS is an agent-based parallel-computing library that simulates multi-entity interactions in physics problems, neural networks, artificial societies, battle games, etc. The parallelization is supported by multiple commodity servers within a cluster. As shown in Figure 1.1, MASS consists of two main components: Places and Agents. A Places object, mapped to threads, is a matrix of Place elements that are dynamically allocated over a cluster. Each Place has the ability to exchange information with one another. An Agents object, mapped to processes, is a set of execution instances that reside and migrate from one Place to another. Each Agent instance can interact with other Agent instances and Place elements.

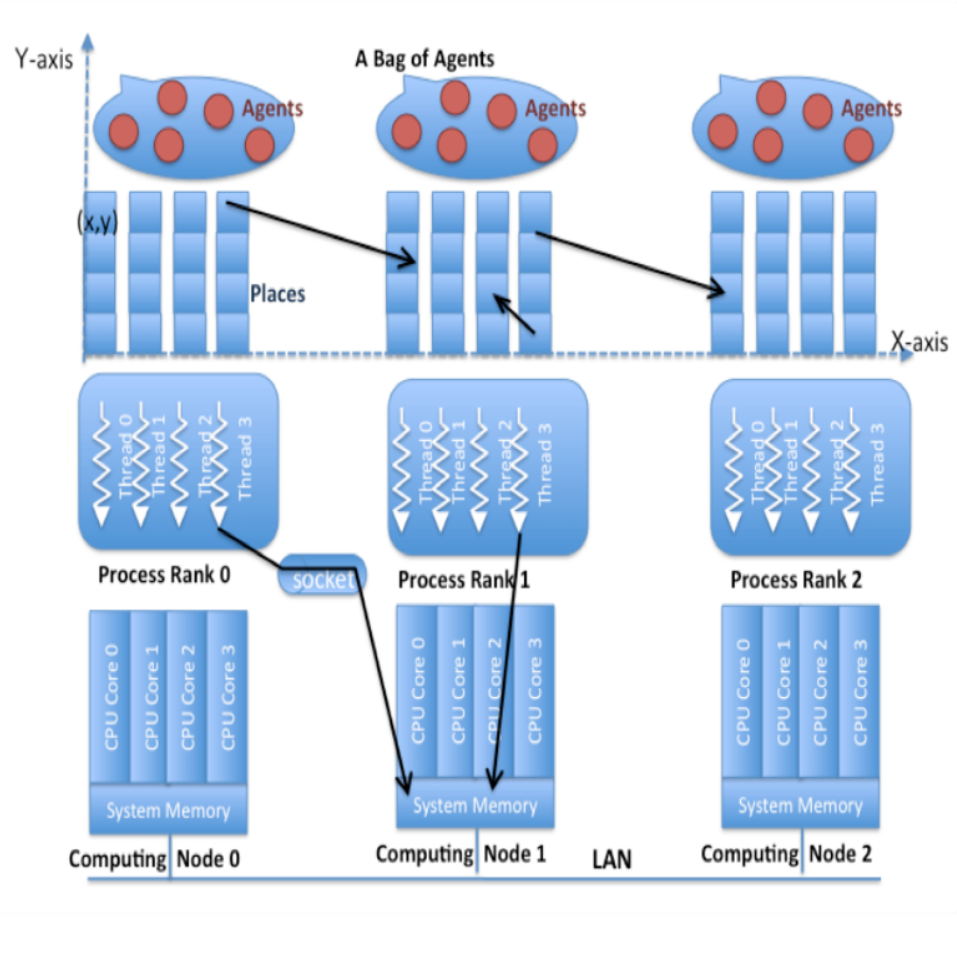

Figure 1.1: MASS Places and Agents

1.2 Problem Definition

The University of Washington Climate Analysis (UWCA) is a web application that uses NetCDF software and MASS Java to calculate the Time of Emergence (ToE) in parallel. ToE is the time that the climate change signal emerges from the noise of natural climate variability, which is the key to climate predictions and risk assessments.

UWCA [24] was designed with one master server reads all the data from storage and sends the data to the slave servers for processing. The amount of time spent in reading large

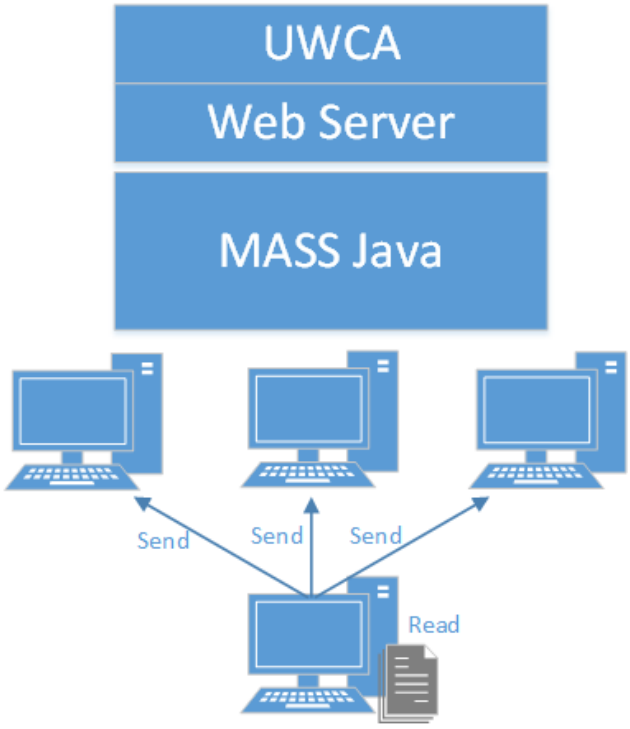

Figure 1.2: UWCA Read - Bottleneck at Master Node

files (22 GB) is tremendously slow. The slow reading performance is a result of having one master server read and transfer data to slaves intensively (Figure 1.2). Adding an I/O layer in UWCA decreases the code readability, and the file processing improvement was far from meeting the expectation. This inspired DSLab to add a Parallel I/O layer on top of MASS to handle files for the MASS applications.

Michael O'Keefe, a former DSLab undergraduate student, conducted parallel reading tests on UWCA and found that the performance issue resides in the UWCA implementation. O'Keefe later proposed a solution [14] to improve the UWCA performance by adding a Parallel I/O Read to the MASS Java library. MASS Parallel I/O is a MASS Java layer that allows efficient file reading from each node to Places. This version does not handle file transfer from master to slave nodes. The implementation made opening, reading, and closing

files possible at each slave server with the assumption of a file's existence.

1.2.1 Proposed Solutions

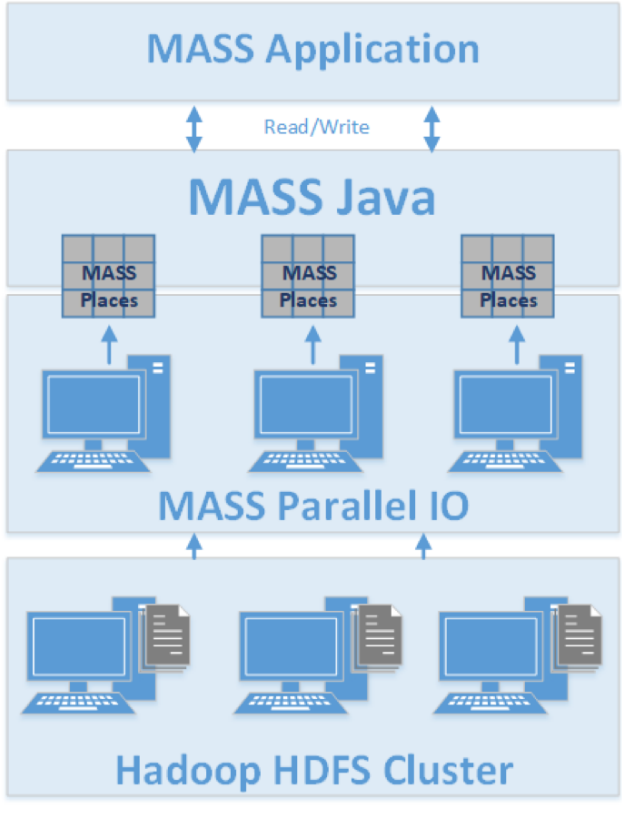

Figure 1.3: MASS HDFS

O'Keefe's MASS Parallel I/O held the assumption of a file's existence over the cluster and only provided the read functionality. To further support parallel climate analysis applications and other possible MASS Java applications, MASS Parallel I/O should handle file storing, distributing, and writing. We propose integrating MASS with Hadoop Distributed File System: MASS HDFS (Figure 1.3). This is a complete MASS Parallel I/O layer that uses HDFS for data storage and distribution over a cluster. Upon completion of the project, MASS HDFS will present the MASS Parallel I/O's write functionality and the ability to

demonstrate processing data at the smallest unit. Our three project goals are listed as follows:

1. Use HDFS in MASS Parallel I/O for file storage and distribution.

2. Complete Parallel I/O Write API.

3. Demonstrate processing data at a per-data-item unit.

This paper is structured as follows: Chapter 2 describes the background of the components used in this project. Chapter 3 covers the related work. Chapter 4 details the development methods including Hadoop setup, change in the development approach, Parallel I/O read and write flows and mechanisms, and the implementation of a separate program we developed to collect files from the MASS cluster. The performance evaluation methods and results are presented in Chapter 5. Finally, Chapter 6 concludes the project and presents possible future work.

Chapter 2

BACKGROUND

In this chapter, we cover some details of the important components used in this project. We explain what MASS Parallel I/O is, how UWCA works, basic NetCDF structures the user should be aware of before performing file operations, and three main Hadoop components: MapReduce, YARN, and HDFS.

2.1 MASS Parallel I/O

MASS does not have any intrinsic parallel I/O. All read and write operations must go through the master node, which creates a bottleneck. MASS Parallel I/O (Figure 2.1), developed

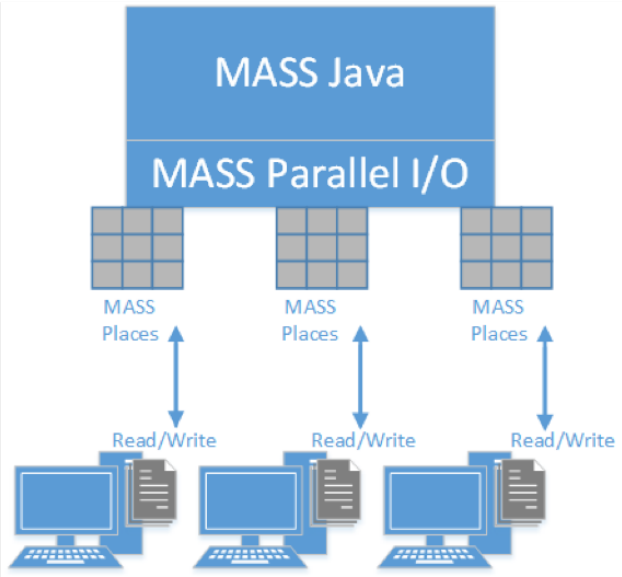

Figure 2.1: MASS Parallel I/O

in 2015, is an additional layer introduced to the MASS Java Library which resolves the bottleneck issue at the master node. It allows both file read and write to be done in parallel by executing file open, read, write, and close on every node. For file read operation, the Parallel I/O layer reads the data to Places allowing the execution entities to interact with the data. The Parallel I/O write operation has the ability to write large files into a cluster efficiently. Introducing such a layer resolves UWCA's performance issue and benefits other MASS applications that could use parallel I/O capabilities.

2.2 UWCA and NetCDF

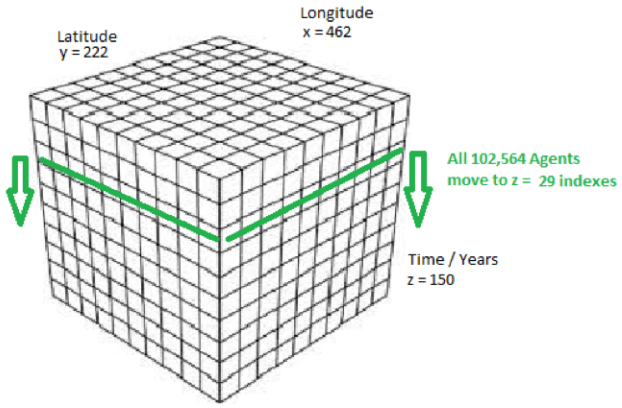

Figure 2.2: NetCDF file used in UWCA

UWCA is a web application module which accepts requests to read NetCDF files. UWCA uses version 0.8.2 of MASS Java. With this version, MASS does not support Parallel I/O to read the file. A separate jar is used to read data from file to MASS Places. NetCDF (Network Common Data Form) is a set of libraries and platform-independent, self-describing data formats that provide array-oriented data access [16]. It is commonly used in climatology, meteorology, and oceanography applications. A NetCDF file has a header describing the structure of the data arrays and the file metadata. The structure describes the shape and attributes of a data variable. Each variable is composed of one or more dimensions that form a

shape. The user should know the variable name and structure to perform file operations [17]. In the case of UWCA, a climatological NetCDF file has three dimensions: longitude (x), latitude (y), and time (z). UWCA uses Agents to travel through the z-coordinate to look at the simulated data and tries to predict ToE. The mapping algorithm maps the data to Places with the correct dimensions, but not quite one-to-one because it compresses the time unit from day to year. Figure 2.2 shows the structure of a UWCA NetCDF file and how Agent instances traverse the time dimension for data-to-Place mapping.

2.3 Hadoop

Hadoop is a distributed system that can process big data in parallel over clusters of commodity servers. The commonly known components include MapReduce for large data processing and HDFS for data storing [1]. MapReduce splits the data sets into chunks and passes each chunk to the mapper. The framework sorts the outputs of the map tasks and passes the outputs to the reduce tasks. Typically, the MapReduce and HDFS components reside over the same cluster for efficient high-speed file transfer [7]. Yet Another Resource Negotiator (YARN) was introduced in Hadoop 2.0 for resource management, job scheduling, and monitoring. The YARN APIs are complex low-level infrastructure APIs. Although it is possible to write a separate YARN-based application, the APIs will increase the development difficulty [23].

HDFS is the Hadoop distributed file system [18] that uses Linux-like commands. It consists of a namespace and blocks storage service. A namespace manages the user-requested file operations while the blocks storage service handles block operations and replications. By default, HDFS makes three replications for each block. The block replica placement policy is based on the factors of reliability, availability, and network bandwidth utilization. Blocks are distributed over the cluster, and more will be created if one of them becomes unavailable. Higher number of replication increases the file system reliable and available. However, it will require higher network bandwidth usage, which causes lower writing efficiency. Reducing the number of replications decreases the system reliability and availability. On the other hand, less network bandwidth is required which boosts the system write performance. In

this project, we excluded MapReduce because it requires converting data to plain text when processing structured data. We also excluded the resource manager, YARN, to avoid increasing the development difficulty as it has complex low-level API.

Chapter 3

RELATED WORK

In this chapter, we review three related works that are comparable to the MASS HDFS components and explain how MASS HDFS differs from these systems. First, we look at SciHadoop, which is a plugin that supports processing array-based structured data in parallel using HDFS. Second, we compare PVFS, an authentic file system that can handle scientific data efficiently, to HDFS. Third, we discuss a complex portable MPI-IO system, ROMIO, that aims to handle scientific data using PVFS.

3.1 SciHadoop

Instead of using an agent-based method like MASS, SciHadoop (Buck et al. 2011) introduced a Hadoop plug-in that uses MapReduce to process large scientific data. MapReduce runs on top of the distributed file system and processes blocks stored locally for efficient network bandwidth usage [7]. Typically, MapReduce handles unstructured data such as log-processing in a low-level byte stream form. Instead of byte stream data models, scientific data is usually array-based. To allow mappers to access high-level structured data, SciHadoop uses the scientific file format libraries such as NetCDF and HDF5 ([22] and [21]). These file format libraries translate high-level data models into low-level byte stream data models. This means partitioning has to be done at the logical level as opposed to the physical level. Partition technique is crucial in the design of SciHadoop development as it affects the result in reducing data transfers, remote reads, and unnecessary reads. Implementation details to achieving these goals can be found in [5].

3.2 PVFS

Parallel Virtual File System (PVFS) is a parallel file system that runs on the Linux clusters developed by Haddad and Ibrahim (2000) [11]. It focuses on general parallel file I/O and is used for HPC applications [12]. To present a stable, full-featured parallel file system that meets the requirements for the Linux clusters, PVFS aims to:

1. Provide high speed multiple concurrent processes/threads I/O

2. Support UNIX/POSIX I/O API [15], and MPI-IO [10, 13]

3. Work with common UNIX commands

4. Provide PVFS file access for UNIX I/O API applications

5. Be robust and scalable

6. Demonstrate ease of use/install

Similar to HDFS, PVFS is open source and has cluster managers that handle file requests and metadata operations. Both systems allow the client and processing units to run on separate clusters. Although performance is better on the same clusters, both HDFS and PVFS divide a file into chunks (or stripes) to store over the cluster. For data reliability and availability, PVFS adapts RAID erasure coded redundancy versus the replications method used in Hadoop 2.0, which has higher overhead. Note that all Hadoop 3.0 releases also employ RAID erasure coded redundancy (See Section 4.1.1). Below is a comparison summary conducted by Tantisiriroj et al. (2011).

1. HDFS and the processing components (MapReduce) are over the same cluster. PVFS separates the processing component for scalability.

2. HDFS supports one writer per file. PVFS supports concurrent writes.

3. HDFS exposes its block location to data nodes for Hadoop applications. PVFS does not expose its stripe mapping to applications.

4. HDFS provides user-based custom API and semantics. PVFS is fixed with UNIX API and mostly POSIX semantics.

5. HDFS uses replication for reliability. PVFS uses RAID erasure coding redundancy. (Note, HDFS also uses RAID in version 3.0)

6. Neither system is optimized for small file handling.

For this project, our team selected HDFS over PVFS because of the simplicity and our familiarity with the system.

3.3 ROMIO

ROMIO is a portable MPI-IO that supports common file operations across clusters of various file systems [20]. It is built on top of an abstract-device interface for portable parallel I/O implementation [19]. ROMIO supports basic portable file operations and implements a small portion of non-portable functions for different file systems and machines. Different read/write functions are invoked by ROMIO's read/write process depending on the file system. For example, _cread/_cwrite are used on Intel PFS, pread(64)/pwrite(64) on SGI XFS, and read/write on UNIX and POSIX. This Parallel I/O also supports shared file operators by storing the pointer value in a file which can be synchronously acquired by processes using file locks.

While ROMIO supports shared file operators over clusters of various file systems, MASS Parallel I/O supports global data access at a unit as small as per-data-item through basic portable calls. Unlike ROMIO, which segregates different files types on different machines in a cluster, MASS HDFS stores different file types across the cluster servers. ROMIO provides precise control to scientific data; however, the system is complex which lowers the

usage simplicity. The current MASS Parallel I/O is designed to read/write highly structured NetCDF files in parallel to facilitate the climate analysis community. It also supports file read/write on unstructured files for other suitable applications that require parallel I/O. We are able to demonstrate loading NetCDF data in a one-to-one fashion to allow access from the processing units. In the next chapter we will discuss our methods and implementations in detail.

Chapter 4

DEVELOPMENT METHODS

Due to a few major issues, a change in the development approach was delivered during the development phase. This chapter will cover the setup of HDFS, the cause and effect of the change in approach, and the design and implementation of Parallel I/O Write and Jsch file collections.

4.1 Hadoop Setup and Development Approach

4.1.1 Hadoop Setup

There were three Hadoop releases available at the starting period of the development phase: Hadoop 2.7.3, Hadoop 2.8.0, and Hadoop 3.0.0-alpha2. While 2.8.0 has a couple major HDFS improvements (See 2.8.0 Release Notes [2]) over the 2.7 release line, 2.7.3 was built upon a stable release. The major HDFS change in Hadoop 3.0 is replacing the data replication method with the RAID erasure coding method for durable storing data. This change reduces the overhead in storage space, network, and other resources from 200% with a default three-replication method, to 50% using the RAID erasure coding [4]. The alpha version of Hadoop is meant for making frequent and rapid changes to user feedback. It has no guarantees of quality or API stability and is not recommended for production use.

As a result, we initially installed with Hadoop 2.7.3, but later switched to Hadoop 3.0.0-alpha4 due to the Jsch Maven dependency issue described in Section 4.1.3. Although an alpha version has a low stability, Hadoop 3.0.0-alpha4 was the only public release version that handled the Jsch issue at the time. This part will be covered under the Dependency Issue section.

4.1.2 Original Approach

One of the goals to accomplish in this project is distributing a file for MASS Parallel I/O Read. The original design is to store files in HDFS. When MASS Parallel I/O Read requests a file, the MASS Parallel I/O layer uses the Hadoop Core library to retrieve the byte data directly from HDFS to the MASS Places as shown in Figure 4.1. In the following section, we review the three major issues which occurred while adopting this approach.

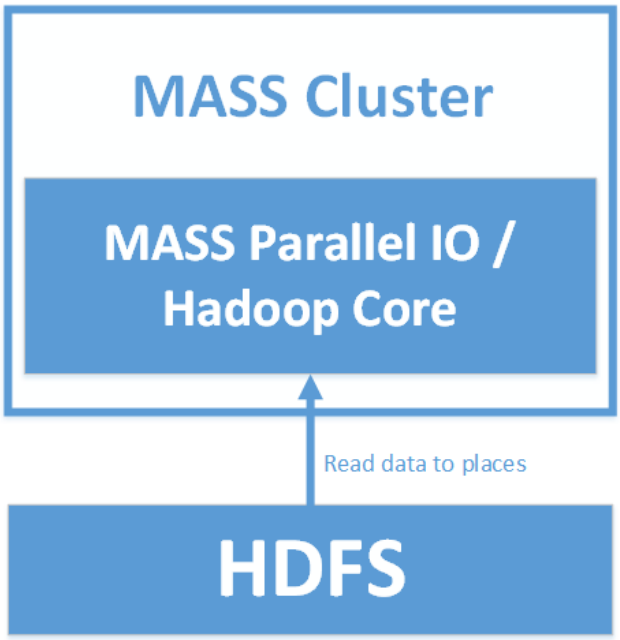

Figure 4.1: Original Development Approach

4.1.3 Issue I - Dependency

The older version of Jsch does not support the encryption method that UW1-320 lab machines require for authentication. While MASS Jsch was upgraded to the latest version (0.1.54) for compatibility with the lab environment, Hadoop 2.7.3 still uses version 0.1.51 Jsch. This became an issue since Maven's dependency lookup method follows the Nearest

Definition First rule [6, Chapter 3.6]. The older version of Jsch was picked up by Maven causing authentication failure in MASS while launching remote nodes.

The Hadoop community upgraded the latest version of Jsch to include the fix of an old version vulnerability. The upgrade was done in Hadoop 2.7.4, 2.8.2, 2.9.0, and 3.0.0-alpha4. Although Hadoop 3.0.0-alpha4 had no guarantee of API stability and quality, it was the only version with the Jsch upgrade that was released to the public by July 2017 [3]. By replacing Hadoop 2.7.3 which was installed at the early development stage with version 3.0.0-alpha4, the dependency issue was resolved.

4.1.4 Issue II - Classpath

In order to correctly access HDFS, the Hadoop Core Library needs the classpath to the Hadoop home directory. Unfortunately, the MASS master node launches MProcess, remote daemon processes each running at a different node, by overwriting the classpath of the remote processes at runtime. The classpath is overwritten to MASS home, which resides in the application jar. This causes the library to search the application jar instead of Hadoop home for dependencies and configurations when the it tries to establish an HDFS connection. As a result, HDFS home directory becomes the local home directory instead of "hdfs://<host>/<directory>", which causes a runtime error while performing HDFS operations. This issue was resolved by adding the Hadoop home path while setting the MASS home path at runtime.

4.1.5 Issue III - Logging

The third issue we encountered before adopting a new approach was Hadoop logging. Instead of using a dedicated channel, MASS uses console I/O for communications between the master and remote nodes. Meanwhile, Hadoop also logs INFO and ERROR level messages to the console, causing Stream Corruption Exception thrown in MASS. The logging scheme can be modified in log4j.properties, which is a file that defines what Hadoop logs and where.

However, due to the time constraint, we modified the development approach instead of addressing this issue.

4.1.6 Modified Approach

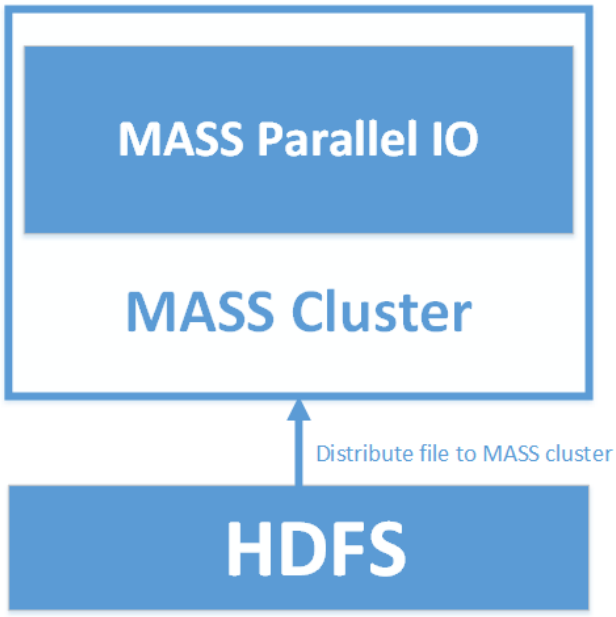

Figure 4.2: Modified Development Approach

The modified approach, as shown in Figure 4.2, distributes the file to the /tmp directory over the MASS cluster instead of reading the data directly from HDFS to Places. MASS launches a separate HDFS client program to distribute the requested file. To retrieve the data, MASS will go to the /tmp directory and get the data using regular file system operations. In the case of a NetCDF file being structured, MASS Parallel I/O uses the NetCDF API to handle the file. Regardless of the change, NetCDF files must be distributed to the cluster before performing any file operations. For this reason, this new approach only affects text file read.

4.2 Parallel I/O Read

4.2.1 Flow and Mechanism

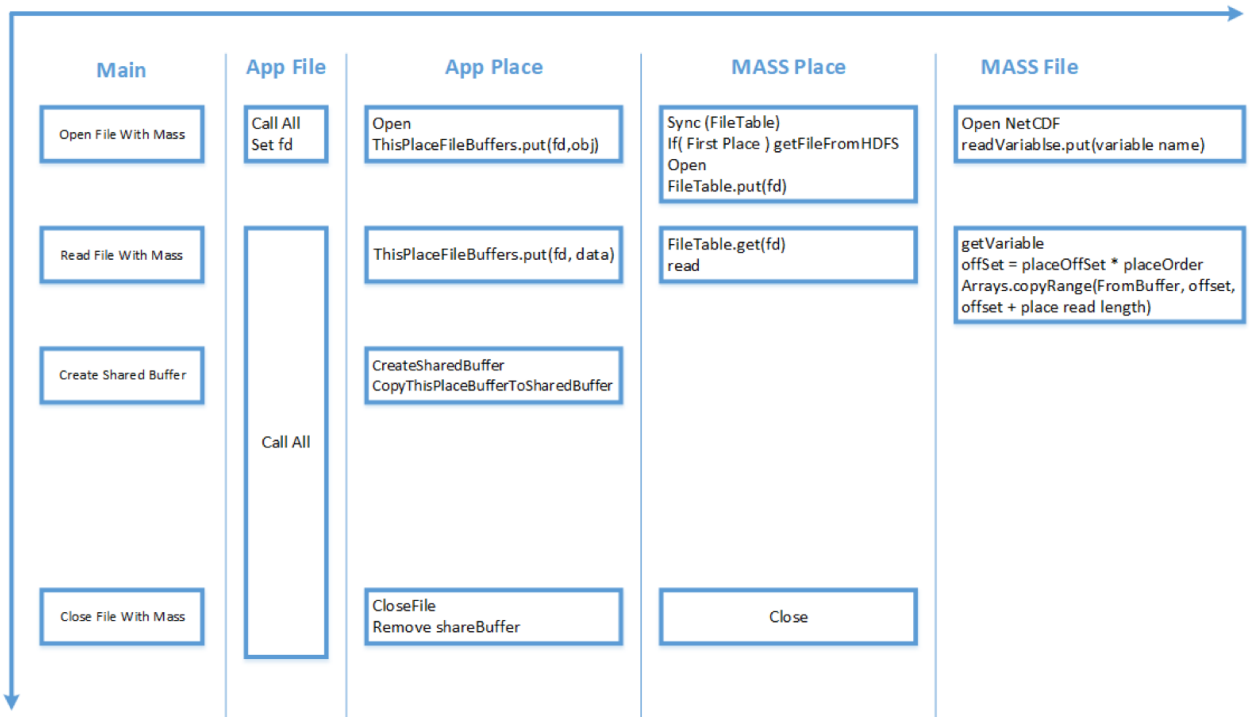

Figure 4.3: Parallel I/O Read Interaction Chart

The usage of MASS Parallel I/O Read Application Main method, shown in Figure 4.3, flows vertically from top to bottom. There are four actions required to complete the read operation:

1. User **opens** the file with MASS.

2. User **reads** the file from memory to Places.

3. User **creates** a shared buffer for global access (assertion purpose in Parallel I/O Read).

4. User **closes** the file with MASS.

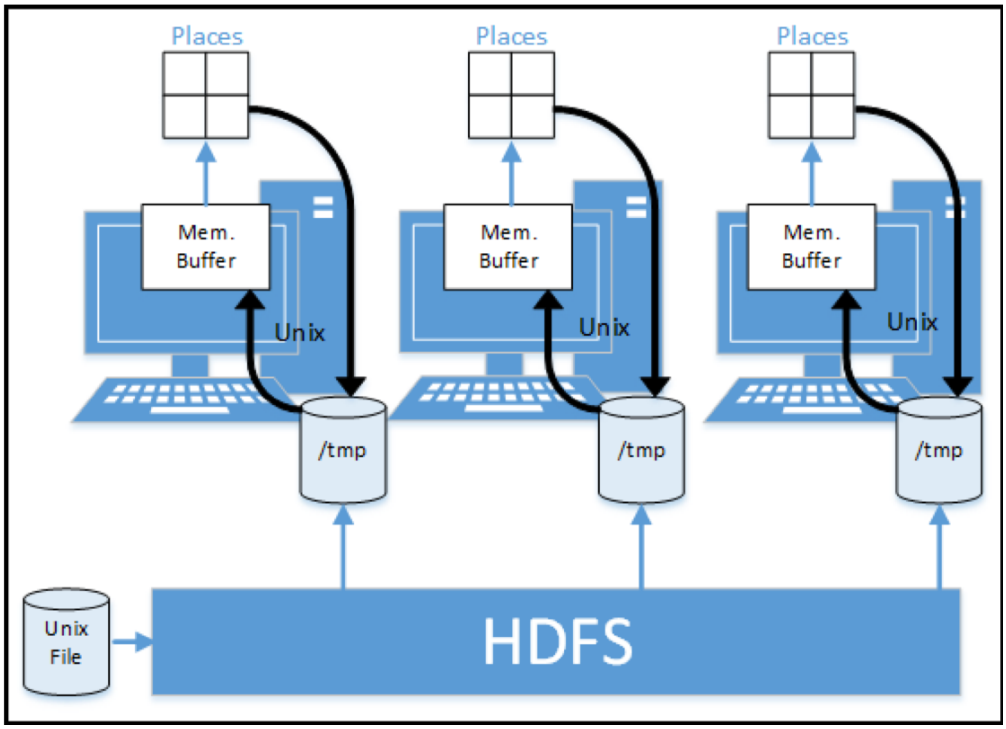

Figure 4.4: Parallel I/O Flow Chart

Each action call goes through the application layer to the MASS layer horizontally. Open File with MASS invokes the CallAll in the application layer File object, which signals each Place to perform the task. The application layer Place, which extends from the MASS layer Place, wraps the arguments for the MASS layer. The first Place of each node retrieves the file from HDFS to its local directory and reads the file to a memory buffer using regular file system operation calls. The data is then ready in the memory buffer for all Places to read.

The original MASS Parallel I/O file read functionality assumes that files exist on each server. We handled this assumption by adding the getFileFromHDFS method in the MASS Place layer before opening the requested file in MASS. The getFileFromHDFS method executes a script command to launch a separate HDFS client program as mentioned in the modified approach. (See Appendix A for HDFS Client example and the invoking script.) This client program supports multiple HDFS operations, but only the read operation is used

in this project for local file retrieval.

In Figure 4.4, the first Place of each node retrieves the file from HDFS to the local /tmp in the open operation. The first Place then reads a portion of the data from /tmp to a memory buffer using regular file operation calls. The portion to read is determined by the number of nodes in the cluster and the order of each node. This is done by calculating the nodeOffset, nodeReadLength, and readOffset:

```
// Read To memory buffer offset algorithm
int nodeOffset = totalDataItemSize / totalNodes
int nodeReadLength =
   (myNodeId is not lastNodeInCluster) ? nodeOffSet :
        nodeOffSet + (totalDataItemSize % totalNodes)
int readOffset = myNodeId * nodeOffset;
```

When the process reaches the main read method, each Place on the same node reads a portion of the data to itself. This data portion is determined by the number of Places on this node and the Place order:

```
// Read to Place offset algorithm
int placeOffset = dataItemSizeOnThisNode / totalPlaces;
   int placeReadLength =
      (placeOrder is not lastPlaceOnNode) ? placeOffSet :
           placeOffSet + (dataItemSizeOnThisNode % totalPlaces)
   int readOffset = placeOffset * placeOrder;
```

4.3 Parallel I/O Write

4.3.1 Flow and Mechanism

One of the main goals of this project is to complete MASS Parallel I/O Write. The flow of the NetCDF Write goes as follows:

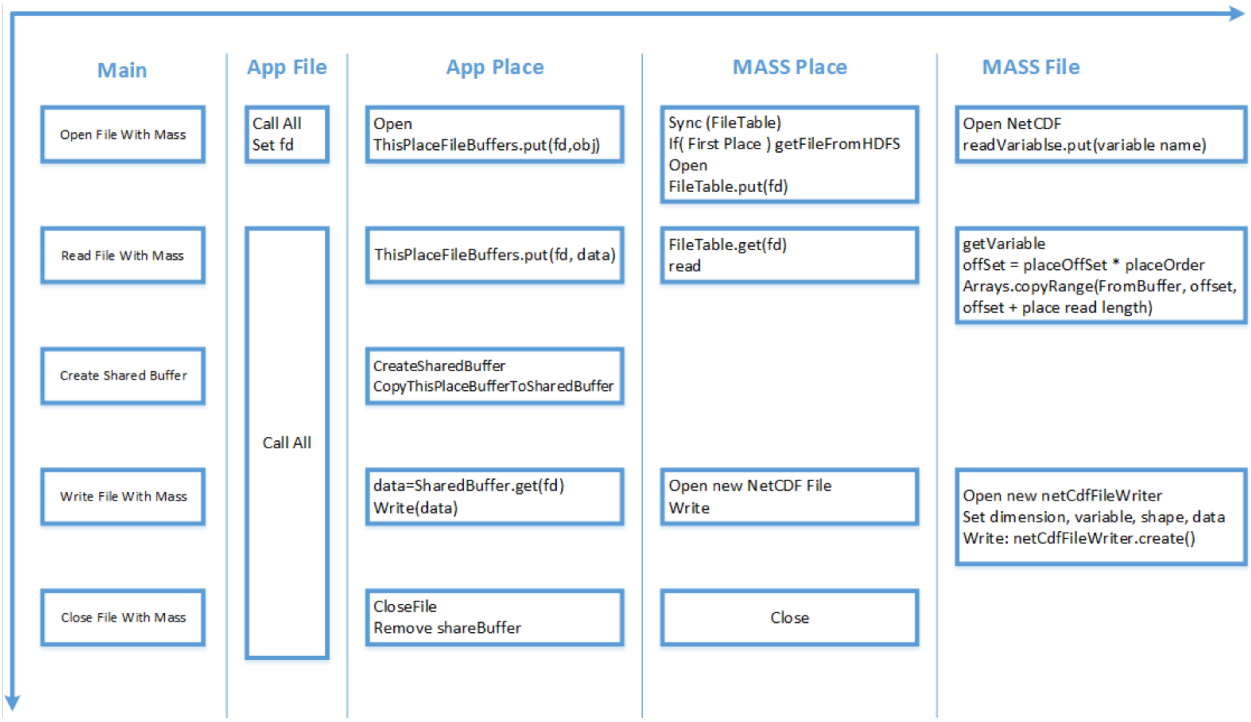

Figure 4.5: Parallel I/O Write Interaction Chart

1. User **opens** the file with MASS.

2. User **reads** the file from memory to Places.

3. User **creates** a shared buffer for global access.

4. User **writes** the data in the shared buffer from Places to /tmp.

5. User **closes** the file with MASS.

In Figure 4.5, the user first opens and reads the file to a shared buffer, then calls WriteFile with MASS in the application main. Next, the App-layer File object invokes the CallAll to signal all places to write the data from the shared buffer to a new file. Similar to MASS Parallel I/O Read, the application layer Place that extends from the MASS layer Place wraps the arguments for the MASS layer. The first Place on each node prepares a file to write and the last Place of each node flushes the data to the file and writes the file to the local /tmp.

As shown in Figure 4.4, the write function gets access to the data in the shared buffer and is able to perform data write. The first Place of a node creates a file writer and sets up a new file structure if handling NetCDF file type. After the file is set up by the first Place, each Place fills the write buffer with an assigned data portion and the last Place flushes it to the local /tmp. Data portion is determined by the number of places on this node and the place order:

```
// Write to file offset algorithm
int placeOffset = dataItemSizeOnThisNode / totalPlaces;
int placeReadLength = (placeOrder is not lastPlaceOnNode) ? placeOffSet :
      placeOffSet + (dataItemSizeOnThisNode % totalPlaces)
int offset = placeOffset * placeOrder;
for(int i = offset; i < (offset+placeReadLength); i++) {
    dataOut.put(i, dataToWrite[i]);
}
```

4.3.2 Writing File to MASS Nodes Mechanism

The actual write algorithm lies in MASS Place and MASS File. MASS reads the data into a shared buffer and writes only a portion of data back into each node. This portion is determined by the total count of data items and the number of nodes in the cluster. For instance, a 60MB text file with 60 million data items is being written to a three-node MASS cluster. Each node will create a new file containing 20MB of data (20 millions of one-byte data items).

Text file type is easier to handle because it can read and write through byte stream. A text file for MASS Parallel I/O write can be created with a portion of the original size. More specifically, a text file can be created with the size being the original file size divided by the number of nodes.

A NetCDF file is structured and has to be created with a matching structure. The reason for keeping the structure is because of the variable shape. A size four 2-D variable could have the shape of 2×2, 1×4, or 4×1. Each node handles a size of two when dealing with two nodes, but the shape is still unknown as it could be 2×1 or 1×2. Since each variable has a specific shape associated to itself, it is difficult to create a new NetCDF file with only a partial shape. When data is read to a shared buffer, the shape is destructed from n-D to 1-D. In order to write a NetCDF file with the correct shape, we keep the shape information from file-read to create a new NetCDF file on each node. Therefore, every newly created NetCDF file for write has the same structure as the original file, while only a portion of the data will be filled.

4.3.3 MASS Place Layer and MASS File Layer Implementation

Text File

As mentioned in the previous section, MASS reads the data into a shared buffer and writes a portion of data into each node. Each Place fills the buffer with byte data and only the last Place writes it out. The counter numberOfPreparedPlace tracks the number of Places that

have filled the data into the buffer. When the numberOfPreparedPlace reaches the total count of Place elements, the last Place will flush the data out using the fileChannel object.

```
// Text Write Implementation
public boolean write(byte[] dataToWrite, int placeOrder)
        throws IOException, InvalidNumberOfPlacesException {
    synchronized (dataOut) {
        fillBufferWithByteData(dataToWrite, placeOrder);
        numberOfPreparedPlace++;
        if(numberOfPreparedPlace == myTotalPlaces) {
            byte[] b = dataOut.array();
            dataOut.rewind();
            fileChannel.write(dataOut);
            return true;
        }
    }
    return false;
}
```

NetCDF File

For NetCDF file write, the NetcdfFileWriter object is created first to prepare the file structure with dimensions, variables, and shapes. The prepareAndCreateNetCDFWriteFile method below demonstrates the steps of constructing a NetCDF file with a defined structure.

```
// prepare new NetCDF file
private void prepareAndCreateNetCDFWriteFile(String variableName, int[] shape) {
    this.shape = shape;
    fileSize = getFileSizeFromShape(shape);
    List<Dimension> dimensions = new ArrayList<>();
    for (int dim = 0; dim < shape.length; dim++) {
```

```
        dimensions.add(netcdfFileWriter.addDimension
                            (null, "Dim" + dim, shape[dim]));
    }
    dataVariable = netcdfFileWriter.addVariable(
                null, variableName, DataType.FLOAT, dimensions);
    netcdfFileWriter.create();
}
```

After the file structure has been created, each place then fills the write buffer using an Index object from the NetCDF API to translate the data from 1-D to n-D. The data is filled based on the Place offset, which is determined by the Place order:

```
// Fill write buffer with data
private void fillBufferWithFloatData(float[] dataToWrite, int placeOrder) {
    Index index = Index.factory(shape);
    int placeOffset = getPlaceReadOffset(dataToWrite.length);
    int placeReadLength = getCurrentPlaceReadLength(
                    dataToWrite.length, placeOffset, placeOrder);
    int offset = placeOffset * placeOrder;
    for(int i = 0; i < (offset + placeReadLength); i++) {
        if(i >= offset) {
            dataOut.setFloat(index, dataToWrite[i]);
        }
        index.incr();}}
```

The Write method synchronizes on the write buffer. Each Place increases the value of the counter that monitors the number of Place elements that have filled the write buffer. This counter tracks if all Place elements have filled the write buffer. Once the counter reaches the total number of Place elements, the last Place has finished filling the write buffer, and will write the data to the /tmp directory.

```
// NetCDF Write Implementation
public boolean write(float[] dataToWrite, String variableName,
                                        int[] shape, int placeOrder){
        synchronized (dataOut) {
            fillBufferWithFloatData(dataToWrite, placeOrder);
            numberOfPreparedPlace++;
            if(numberOfPreparedPlace == myTotalPlaces) {
                netcdfFileWriter.write(dataVariable, dataOut);
                return true;
            }
        }
    return false;
}
```

4.4 User-Level Example Code

All MASS Applications need to initialize the MASS cluster through initMASS. Applications that require using the MASS cluster can only run after the cluster is set up. At the end of the computation, the user should always call MASS.finish to clean up the instantiated objects and to disconnect all the connections established for the session.

```
public static void main(String[] args) {
    mainArgs = args;
    if (mainArgs.length != 9) {
        printExpectedArguementsOrder();
    } else {
        initMASS();
        runAppShellToCreateAndTestFilesWithMass();
        MASS.finish();
```

```
    }
}
```

In the initMASS method, the cluster gets set up based on the user-provided parameters. Basic authentication parameters are the username, password, machine file path, and port. The machine file gives the domain names of the machines that the user wishes to use for the cluster. The user has to provide the number of processes, threads, and the Places dimension for the cluster. MASS.init establishes the cluster, and after the cluster has been set up, the user can instantiate the Places object for the application. Below is an example of the initMASS method. Please refer to the MASS Java Manual [9] for more information.

```
private static void initMASS() {
    String[] massArgs = new String[4];
    massArgs[0] = mainArgs[USERNAME];
    massArgs[1] = mainArgs[PASSWORD];
    massArgs[2] = mainArgs[MACHINE_FILEPATH];
    massArgs[3] = mainArgs[PORT];

    // set up MASS cluster based on user param
    MASS.init(massArgs, Integer.parseInt(mainArgs[NUM_PROCESSES]),
    Integer.parseInt(mainArgs[NUM_THREADS]));

    int x = Integer.parseInt(mainArgs[X_PLACES]);
    int y = Integer.parseInt(mainArgs[Y_PLACES]);
    int z = Integer.parseInt(mainArgs[Z_PLACES]);
     // instantiate Places object as needed
    NETCDF_TEST_PLACES = new Places(1, "NetcdfTestPlace", null, x, y, z);
    TXT_TEST_PLACES = new Places(2, "TxtTestPlace", null, x, y, z);
}
```

The code to run MASS Parallel I/O Read and Write are similar. As described in Section 4.3.1, file write is done by opening the file, reading data to MASS, creating a shared buffer to access data in all Places, writing data to /tmp, then closing the file object using MASS. Below is an example of the MASSParallelInputForWrite method that demonstrates the MASS file write procedure.

```
private static void testMASSparallelInputForWrite(TestFile fileTester) {
        fileTester.openFileWithMASS();
        fileTester.readFileWithMASS();
        fileTester.openReadAndCloseFileWithoutMASS();
        fileTester.createSharedFileBuffer();
        fileTester.writeFileWithMASS();
        fileTester.closeFileWithMASS();
}
```

For each of the function calls above, the user has to wrap up the arguments and use CallAll to call the user-defined method. Each Place executes the method with the provided arguments once the method name and the arguments have been sent to the remote Places. In the example below, *writeFileIntoPlaceBuffer_* is the user-defined method that will be invoked at each Place for the NetCDF writeFileWithMASS operation.

```
public void writeFileWithMASS() {
    Object netcdfWriteArgsWrapper;
    Object[] netcdfWriteArgs = new Object[5];
    netcdfWriteArgs[0] = fileDescriptor;
    netcdfWriteArgs[1] = variableName;
    netcdfWriteArgs[2] = shape;
    netcdfWriteArgs[3] = filepath;
    netcdfWriteArgsWrapper = netcdfWriteArgs;
    places.callAll(
```

```
        ParallelInputTestPlace.writeFileIntoPlaceBuffer_
                                ,netcdfWriteArgsWrapper);
}
```

As shown in Figure 4.5, the CallAlls are called in the App File layer which invokes the user-defined methods in the App Place layer. More specifically, the writeFileWithMASS method above calls the *writeFileIntoPlaceBuffer_* method in the App Place layer. The snippet below shows that the writeFileIntoPlaceBuffer method gets the data from the shared buffer and calls the writeFileIntoThisPlaceBuffer method to unwrap the arguments and invoke the write function in the MASS Place layer to complete the operation.

```
public void writeFileIntoPlaceBuffer(Object arg)
throws Exception {
    Object[] writeArgs = (Object[]) arg;
    if (sharedFileBuffers.containsKey(fileDescriptor)) {
        // get data from shared buffer
        Object writeDataBuffer = sharedFileBuffers.get(fileDescriptor);
        writeArgs[writeArgs.length - 1] = writeDataBuffer;
        writeFileIntoThisPlaceBuffer(writeArgs);
    }
}

public void writeFileIntoThisPlaceBuffer(Object[] writeArgs)
throws Exception {
    // unwraps arguments
    int fileDescriptor = (int) writeArgs[0];
    String variableName = (String) writeArgs[1];
    int[] shape = (int[]) writeArgs[2];
    String filePath = (String) writeArgs[3];
    float[] writeDataBuffer = (float[])writeArgs[4];
```

```
    if(writeDataBuffer instanceof float[]) {
        openForWrite(filePath, variableName, shape);
        // MASS Place layer wirte
        write(fileDescriptor, writeDataBuffer, variableName, shape);
    }
}
```

4.5 File Collection Jsch

4.5.1 Design

MASS Parallel I/O Write first reads a portion of a file based on the number of nodes in the cluster. Then each node writes the data to its local /tmp directory. Next, File Collection Jsch collects the files from each node and reassemble the files to ensure its quality. This is done by establishing a Jsch session to collect the files. The file data are transferred through ByteArrayOutputStream and are reassembled using different methods based on file type.

4.5.2 Implementation

As mentioned in the previous section, a newly created text file size is equal to the original size divided by the number of nodes. In Figure 4.6, an N-node MASS cluster divides a text file into n parts and writes each part to each of the MASS nodes. The user then collects the files to the local directory. The text files are collected from the remote nodes using ByteArrayOutputStream. Since the order of data is known, the files are reassembled by appending each byte array one after another.

On the other hand, the NetCDF file stores data in a variable that is composed of multiple dimensions. The shape of a variable cannot be reconstructed once it is destructed. Therefore, each NetCDF file in the cluster contains the original variable structure, but only fills with a certain portion of data items. In Figure 4.7, each file gets collected to a local machine. At the end of a Jsch session, the local machine will have as many NetCDF files as the number

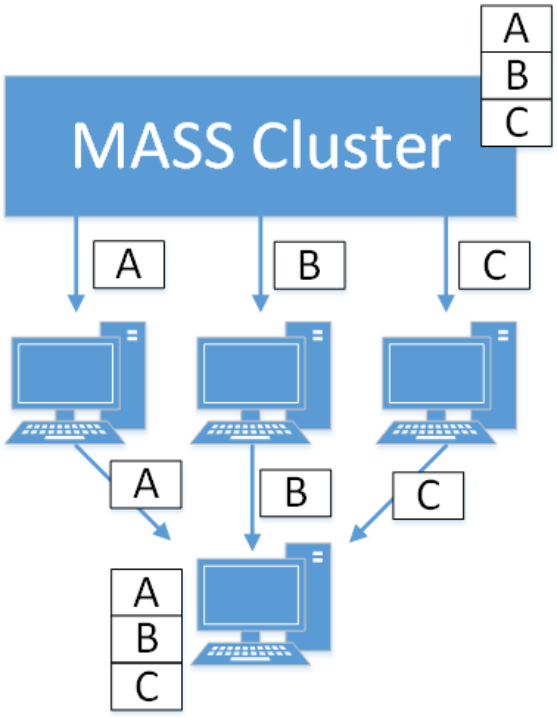

Figure 4.6: Text File Collection Using Jsch

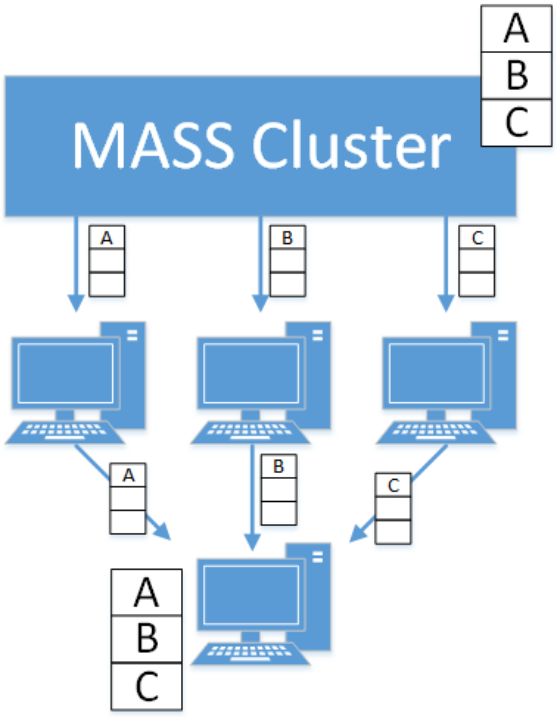

Figure 4.7: NetCDF File Collection Using Jsch

of nodes in the MASS cluster. The local machine will create another NetCDF file, then open and read each collected file to put the data into the newly created file. See Appendix B for examples of implementations.

4.5.3 Usage Example

To start the File Collection Jsch jar, the user will provide two arguments: username and password. In the Main method example below, the user will be prompted to enter the type and additional information, such as dimension sizes, of the file to be retrieved.

- For text file:

  <filename> <fileSizeInBytes>

- For NetCDF file:

  <filename> <variableName> <xDimSize> <yDimSize> <zDimSize>

```
public static void main(String[] args) {
   if(args.length == 2 ) {
       username = args[0];
       password = args[1];
       printApplicationStartMessage();
       promptParseAndExecuteUserInput();

   } else {
       System.out.println("Expected Arguments: <username>, <password>");
       System.exit(1);
   }
}
```

Next, the system will ask the user to specify the machines in the correct order. The machine order should be numbered from small to large, except the first machine of the list, which should always be the master node. File Collection Jsch supports up to 16 machines and the argument format should be:

- <master> <node1> <node2> (Ordered by machine number)

Suppose the machines uw1-320-01, uw1-320-02, and uw1-320-03 are used for MASS HDFS having uw1-320-03 being the master node. The argument order used in File Collection Jsch would be ”03 01 02”.

Chapter 5
PERFORMANCE EVALUATION

We conducted two extreme cases for the performance evaluation: the lowest Place extreme and the highest Place extreme. The first extreme case aims to optimize the CPU usage by using one Place per core. Since each of the lab servers has four CPU cores, we conducted the test using four Place elements per CPU. The second extreme case focuses on dedicating one Place per data item to demonstrate the ability of processing data at the smallest unit.

5.1 Extreme Case 1 Test Method

We conducted the performance evaluation on file open and read operations. The performance is not impacted by the measurement because it is done by taking time-stamps before, between, and after the operations. The open performance can be affected by disk caching as it keeps the recently used files. To maintain performance consistency, we performed four tests on each file and alternated file types between each test. In this first extreme-case scenario, we used three text files and three NetCDF files of size 50MB, 100MB, and 200MB. We conducted the test using 1, 2, 4 and 8 nodes with 2 and 4 threads, and 4 Place elements on each node. We summarized the result by taking the first open/read test and the average open/read time from the 2nd, 3rd, and 4th tests to avoid disk caching. Although we cannot guarantee that the first open performance is not affected by disk caching, the average open performance is guaranteed to be using the disk caching.

5.2 Extreme Case 2 Test Method

The purpose of this second extreme-case scenario is to complete our third goal of demonstrating the potential for extracting the data at a per-data-item scale using MASS Parallel I/O. This

means the data can be mapped, one-to-one, to a distributed parallel processing framework waiting for the processing units to execute. A MASS Place creation takes about 0.032ms/Place. The maximum number of Place elements that can be created before disk swap occurs is under 24 million (Table 5.1).

Table 5.1: Place Creation Time Table

Places (million)	Approx. MASS init time (min)
15	2
20	2.5
22.5	4.2
24	4.4
24.5	15+ (CPU uses swap memory)

Most data used in parallel processing are large. For this demonstrative performance test, we tested 50MB text and NetCDF files. The performance is measured by taking the time-stamps before, between, and after file open and read operations, so the performance is not impacted by the measurement. We used 50 million Place elements for the text file (1 byte per character) and around 12.5 million Place elements for the NetCDF file (4 bytes per float data). This test is done on 4 nodes with 4 threads to keep the number of Place elements under 24 million.

5.3 Results

5.3.1 Extreme Case 1 Result

In this section, we compare the performance in 2 threads and 4 threads. The results are grouped by file type and operation, with each group having the first-time performance and the average performance results.

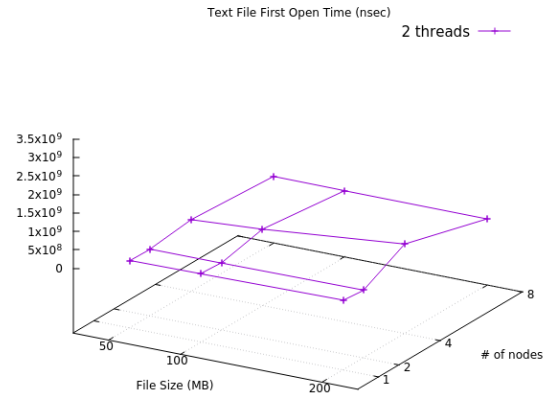

Figure 5.1: First Time Text File Open Using 2 Threads

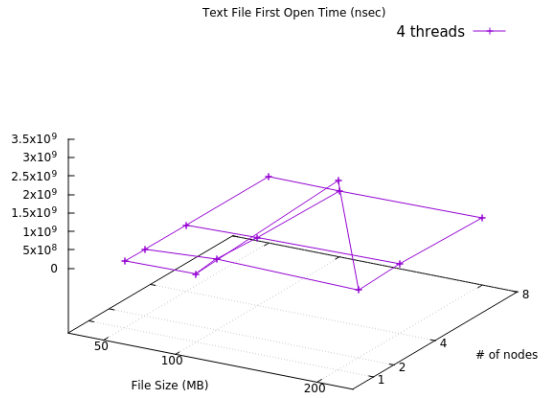

Figure 5.2: First Time Text File Open Using 4 Threads

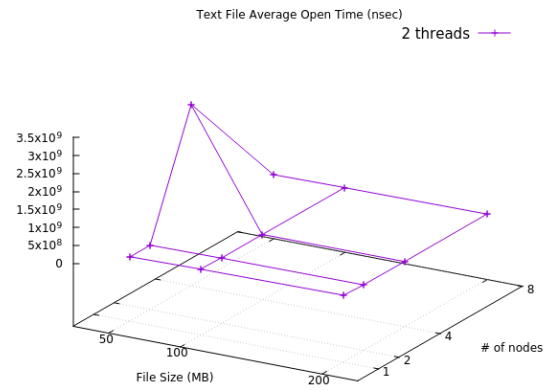

Figure 5.3: Average Time Text File Open Using 2 Threads

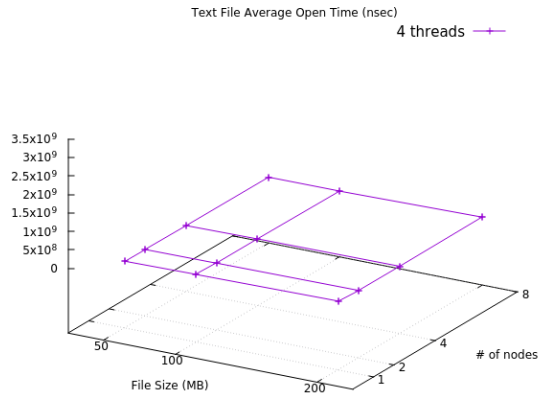

Figure 5.4: Average Time Text File Open Using 4 Threads

Text File Open

The first-time text file open takes on average 0.07 seconds with 50MB, 0.11 seconds with 100MB, and 0.58 seconds with 200MB on 2 and 4 threads. For average text file open, 0.45 seconds with 50MB, 0.06 seconds with 100MB, and 0.08 with 200MB. The performance is slower on 200MB first-time file open and 50MB average time file open as shown in Figure

5.2 and Figure 5.3. We suspect the outlier results are due to system workload. As our lab is available for all CS department students, it is likely the performance evaluation was interrupted by other processes. We calculated the time performance without the spikes and results were 0.04 seconds on 50MB on average file open and 0.18 seconds on 200MB on first-time file open. This result without the spikes better match our expectations.

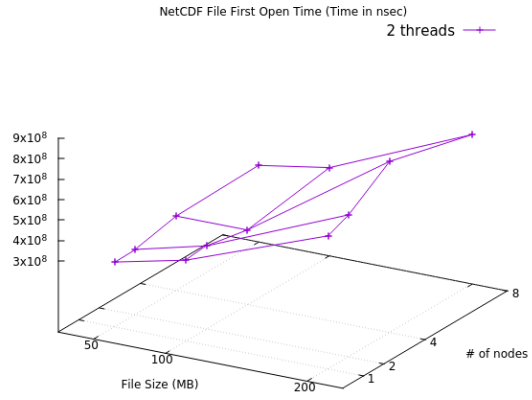

Figure 5.5: First Time NetCDF File Open Using 2 Threads

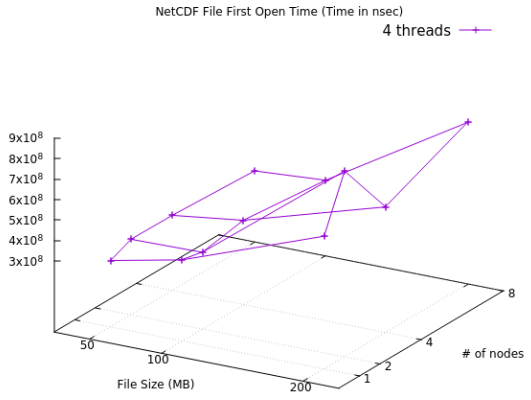

Figure 5.6: First Time NetCDF File Open Using 4 Threads

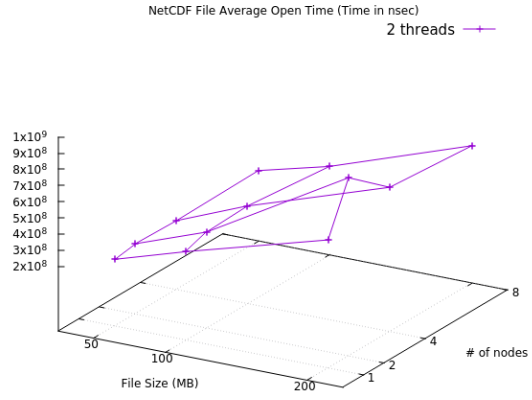

Figure 5.7: Average Time NetCDF File Open Using 2 Threads

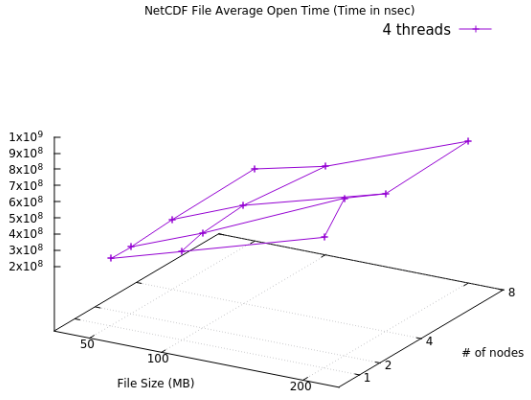

Figure 5.8: Average Time NetCDF File Open Using 4 Threads

NetCDF File Open

A summary of the first-time file open performance average on both 2 and 4 threads is as follows: 0.30 seconds with 50MB, 0.35 seconds with 100MB, and 0.69 seconds with 200MB. For average file open, the system takes 0.23 seconds on 50MB, 0.37 on 100MB, and 0.69 seconds on 200MB. NetCDF file open performance is fairly consistent: the time spent increases as the file size and the number of nodes increase. From Figures 5.5 to 5.8, the file open time increases as the number of computing nodes increases. This is because the first Place of each node uses the NetCDF API to open the requested file and loads the entire data to a memory buffer.

Text File Read

Text file read has the best performance result throughout the performance evaluation. The worst case was completed within 0.2 seconds. The average of using 2 and 4 threads on first-time file read summarizes to 0.02 seconds on 50MB, 0.03 seconds on 100MB, and 0.09 seconds on 200MB. Average file read time shows 0.02 seconds on 50MB, 0.04 seconds on 100MB, and 0.07 seconds on 200MB. Figures 5.9 and 5.12 show a clear decreasing behavior on 200MB file using 2, 4, and 8 nodes. Figures 5.10 and 5.11 show a time decrease on all files between 4 nodes and 8 nodes. However, because of the file sizes are relatively small, the data does not demonstrate a clear consistent trend of decrease in file reading time from Figures 5.9 to 5.12.

NetCDF File Read

NetCDF file read performance was completed under 0.16 seconds for both first-time reading and average reading scenarios. Our average of using 2 and 4 threads summary for first-time NetCDF file read shows 0.03 seconds on 50MB, 0.05 seconds on 100MB, and 0.11 seconds on 200MB. For average time NetCDF file read, our summary shows 0.03 seconds on 50MB, 0.04 on 100MB, and 0.16 on 200MB. Since the system loads the data to a memory buffer for file

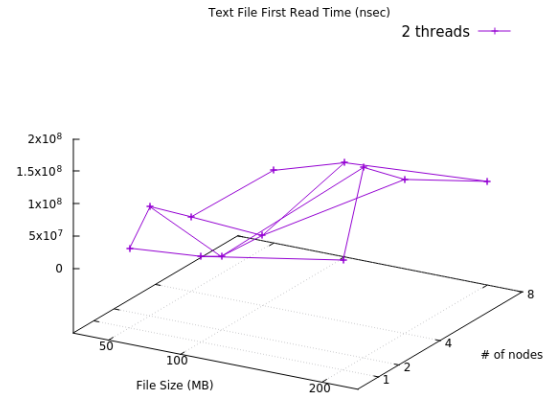

Figure 5.9: First Time Text File Read Using 2 Threads

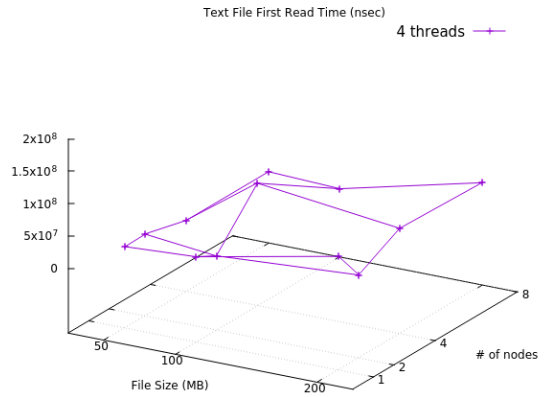

Figure 5.10: First Time Text File Read Using 4 Threads

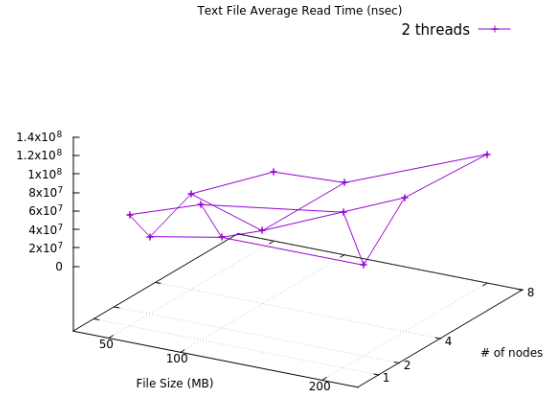

Figure 5.11: Average Time Text File Read Using 2 Threads

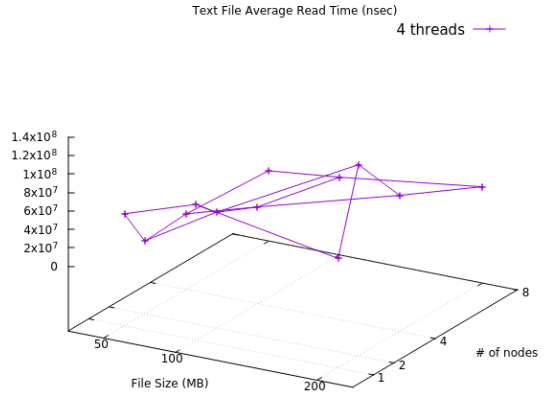

Figure 5.12: Average Time Text File Read Using 4 Threads

read, we can see the first-time result and the average result are similar. Figure 5.13 shows a clear performance improvement from 1 node to 8 nodes. The improvement can also be found in Figure 5.14 on 50MB and 100MB. Three out of four diagrams have the best case using 2 computing nodes. An unexpected outlier result of 0.44 seconds was found in the 4-thread average case on 200MB file read using 4 nodes. We believe this was also due to the process interruption caused by other users. Without the outlier data, everything was completed in

under 0.2 seconds.

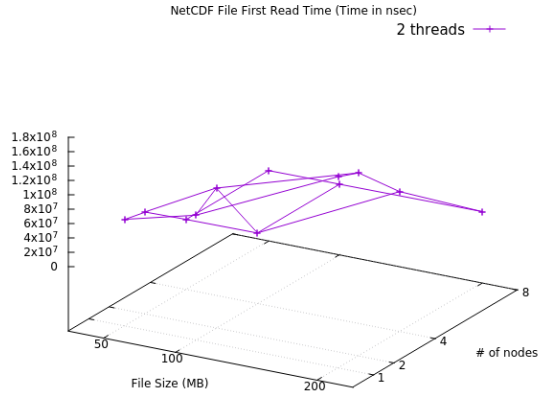

Figure 5.13: First Time NetCDF File Read Using 2 Threads

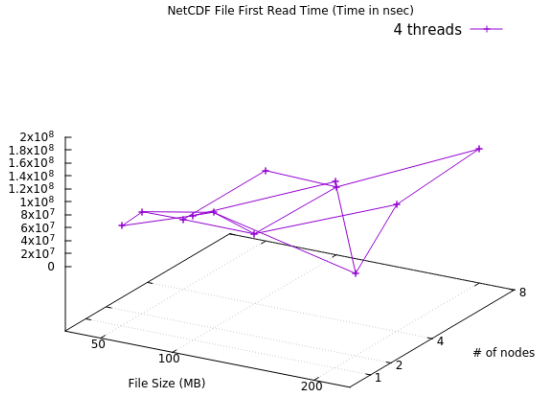

Figure 5.14: First Time NetCDF File Read Using 4 Threads

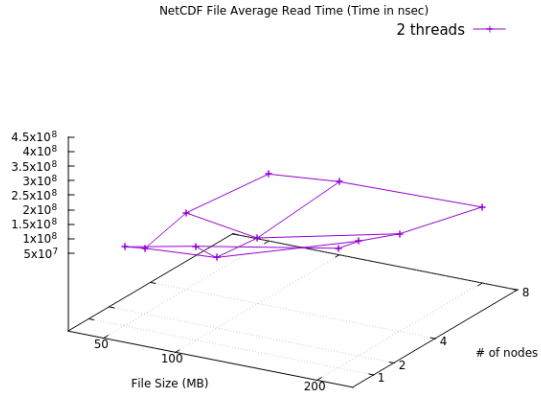

Figure 5.15: Average NetCDF File Read Using 2 Threads

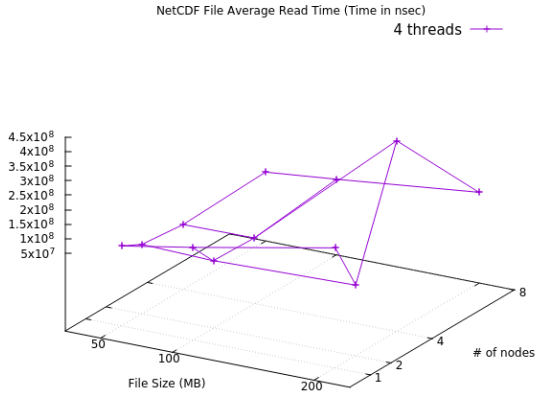

Figure 5.16: Average NetCDF File Read Using 4 Threads

In summary, the MASS HDFS file open time increases as the number of MASS nodes increases, which could be a result of the HDFS file read operation. Since the 3.0.0-alpha4 version of HDFS uses the erasure coding method, we have no guarantee of which datanodes the MASS clients would query for the data. It is possible that multiple MASS clients would be requesting data from the same datanode, and in this case, the HDFS read operation becomes

sequential. The MASS HDFS read performances do not show obvious improvements either. We suspect this is because file read is a memory operation rather than a disk operation; therefore, it becomes difficult to see the improvements with the performance time being relatively small.

5.3.2 Extreme Case 2 Result

The second extreme case result better meets our expectations: the NetCDF file needs more time on file open and less time on file read compared to the text file (Figure 5.17). The numbers show that it takes approximately 4.5 seconds to open and 14.1 seconds to read using 50 million Places, and 6.3 seconds to open and 0.7 seconds to read using 12.5 million Places. Opening the NetCDF file with 12.5 million Place elements takes about 1.5 times longer than opening the text file using 50 million Place elements. This is an expected result, as each NetCDF file has to be read entirely using NetCDF API before loading to the memory buffer by the first Place.

On the other hand, reading the text file using 50 million Place elements takes 18 times longer than reading the NetCDF file using 12.5 million Place elements. Notice that there are two Places objects created after the MASS cluster is set up successfully in the initMASS method in Section 4.4. When the user specifies using 12.5 million Place elements, the application creates 12.5 million Places for NetCDF and 12.5 million Places for text. This means when the user specifies using 50 million Place elements, a total of 100 million Places will be created, 50 million Places for each type. Although we expected text file read to be more time consuming, we did not expect the result to be more than 4 times longer. We believe with such noticeable memory usage difference, the memory overhead is the main cause of the reading time being 4 times longer when applying 50 million Places.

As each character is 1 byte and each float data is 4 bytes, float data requires fewer Place elements to hold data items in files of the same size. The total time in processing the text file is significantly greater than the NetCDF file. This evaluation not only demonstrates the ability to process data at the smallest unit but also suggests this extreme case is better with

numerical data when dealing with the same size of files.

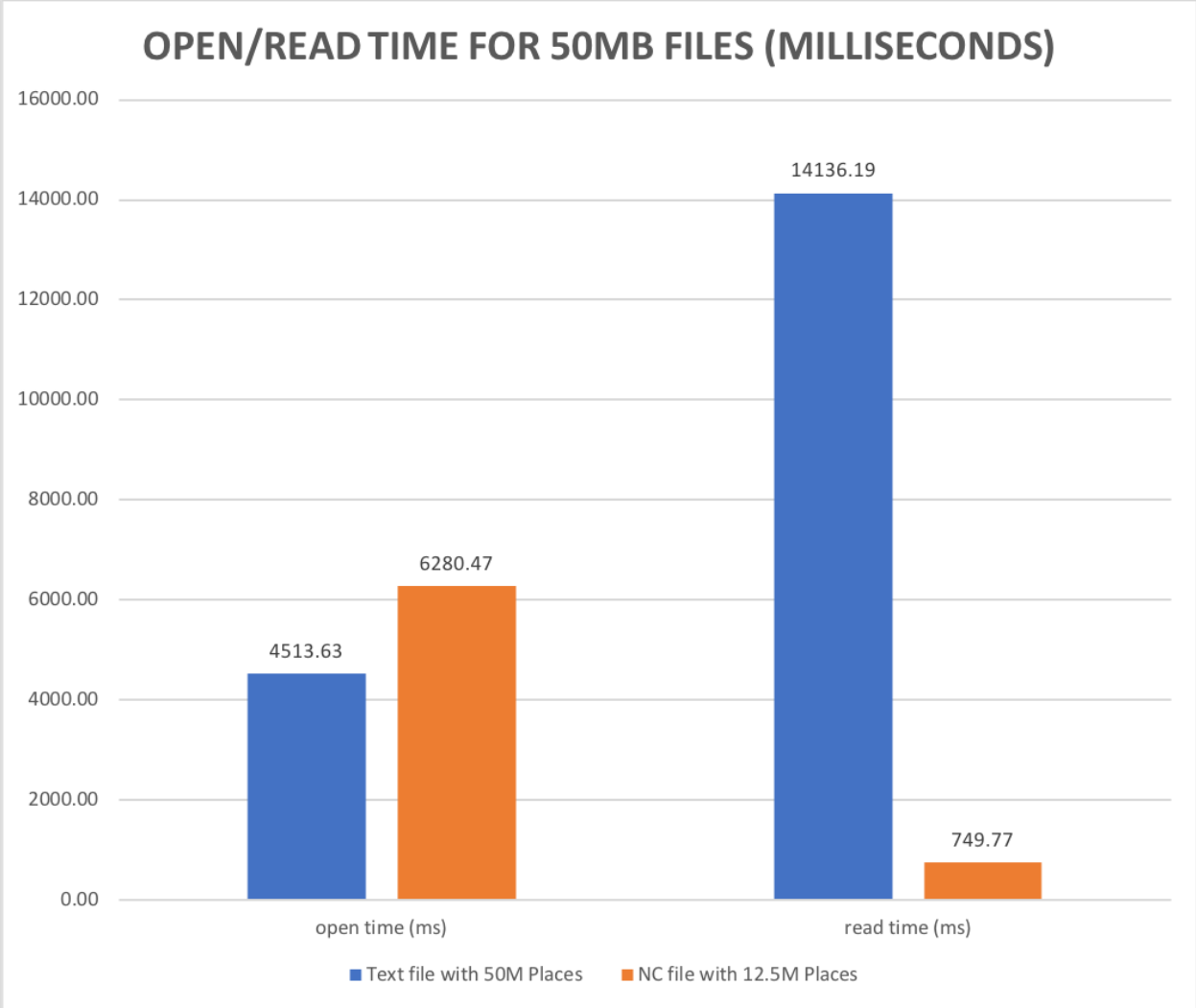

Figure 5.17: Open and Read Time for 50MB Files (milliseconds)

Chapter 6

CONCLUSION

6.1 Summary

In this project, we developed a simple-to-use I/O layer that handles parallel file operations to support scientific parallel computing applications. More specifically, we provided the capability to handle structured data while maintaining simplicity of usage. We also allowed MASS HDFS to read data into distributed arrays without introducing a single point of bottleneck or data conversion. As a solution to further support scientific analysis applications, we proposed integrating HDFS with MASS to handle parallel file distribution and completing the MASS Parallel I/O write functionality. Our initial development method was to use the Hadoop Core Library to read the byte stream data directly from the distributed file system to MASS Places. Due to time constraint, we could not further investigate the issues which occurred. We altered the development approach by using a separate HDFS client program to distribute the file from HDFS to the MASS cluster, then apply regular file system calls to retrieve data from HDFS to MASS Places.

In Chapter 1 and 2, we provided introduction and background of MASS HDFS. We focused on providing information about MASS, MASS Parallel I/O, UWCA, and Hadoop basic components.

In Chapter 3, we discussed related work that has been done on similar topics. SciHadoop differs from MASS HDFS by MapReduce, which requires data conversion. ROMIO efficiently uses PVFS, but the system is complex. Our MASS HDFS maintains simplicity of usage and does not require data conversion.

In Chapter 4, we explained the issues we encountered. We detailed how the developmental approach detoured from reading byte data directly from HDFS to Places, to distributing

files to the /tmp directory over the cluster, then reading the data to Places using UNIX file operations.

In Chapter 5, we described the two extreme cases presented in the performance evaluation. The first case optimizes the CPU usage having four Place elements on each node in the cluster. The second case focuses on demonstrating the ability of processing data at the smallest unit by dedicating one Place per data item. We discussed the first extreme case result in Text Open, NetCDF Open, Text Read, and NetCDF Read sections. However, the results do not show clear performance improvements, with the performance time being relatively small. Our second extreme case of evaluation successfully demonstrated the potentiality of processing data at a per-data-item scale and indicates that this extreme case is more efficient with numerical data, as there are less data items.

6.2 Future Work

MASS HDFS currently supports text data and float data. A possible area of improvements includes adding additional data type supports in the read/write features. Another area of research is stress testing MASS HDFS to explore the number of nodes that optimizes the cluster and the number of Place elements required to efficiently handle a file based on its size and data type. Other manners in which this research could be extended are to investigate the issues discussed in Chapter 4 and complete MASS HDFS with the original method of approach, or to integrate MASS HDFS with a stable version of Hadoop with the Jsch upgrade instead of the current alpha version.

BIBLIOGRAPHY

[1] Apache Hadoop Project. Apache Hadoop. http://hadoop.apache.org/.

[2] Apache Hadoop Project. Apache Hadoop 2.8.0. http://hadoop.apache.org/docs/r2.8.0/index.html.

[3] Apache Hadoop Project. Apache Hadoop Releases. http://hadoop.apache.org/releases.html.

[4] Apache Hadoop Project. HDFS Erasure Coding. http://hadoop.apache.org/docs/r3.0.0-alpha2/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html.

[5] Joe B Buck, Noah Watkins, Jeff LeFevre, Kleoni Ioannidou, Carlos Maltzahn, Neoklis Polyzotis, and Scott Brandt. Scihadoop: array-based query processing in hadoop. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–11. IEEE, 2011.

[6] John Casey, Vincent Massol, Brett Porter, and Carlos Sanchez. Better builds with maven. *The How-to Guide for Maven*, 2:p65, 2008.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[8] DSLab. MASS Java Developer Guide. http://depts.washington.edu/dslab/MASS/index.html.

[9] DSLab. MASS Java Manual. http://depts.washington.edu/dslab/MASS/index.html.

[10] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced features of the message-passing interface*. MIT press, 1999.

[11] Ibrahim F Haddad. Pvfs: A parallel virtual file system for linux clusters. *Linux Journal*, 2000(80es):5, 2000.

[12] Samuel Lang, Philip Carns, Robert Latham, Robert Ross, Kevin Harms, and William Allcock. I/o performance challenges at leadership scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 40. ACM, 2009.

[13] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997. http://www.mpi-forum.org/docs/docs.html.

[14] Michael O'Keefe. CSS499 Winter 2016 Term Report. http://dept.washington.edu/dslab/MASS/.

[15] B Pollak. Portable operating system interface (posix)-part 1x: Real-time distributed systems communication application program interface (api). *IEEE Standard P*, 1003.

[16] Russ Rew and Glenn Davis. Netcdf: an interface for scientific data access. *IEEE computer graphics and applications*, 10(4):76–82, 1990.

[17] Russ Rew, Glenn Davis, Steve Emmerson, and Harvey Davies. Netcdf user's guide, 1993.

[18] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. IEEE, 2010.

[19] Rajeev Thakur, William Gropp, and Ewing Lusk. An abstract-device interface for implementing portable parallel-i/o interfaces. In *Frontiers of Massively Parallel Computing, 1996. Proceedings Frontiers' 96., Sixth Symposium on the*, pages 180–187. IEEE, 1996.

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

[21] The HDF Group. HDF5. https://support.hdfgroup.org/HDF5/.

[22] UCAR Community Programs. unidata NetCDF. https://www.unidata.ucar.edu/software/netcdf/.

[23] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[24] Jason Woodring, Matthew Sell, Munehiro Fukuda, Hazeline Asuncion, and Eric Salathe. A multi-agent parallel approach to analyzing large climate data sets. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pages 1639–1648. IEEE, 2017.

Appendix A

HDFS CLIENT PROGRAM AND SCRIPTS

A.1 HDFS Client Script

The following script is invoked in the getFileFromHDFS method in the MASS Place layer (See Figure 4.3.1).

```
#!/bin/sh
task="$1"
file="$2"
cd /CSSDIV/research/dslab/jas/HDFSClient
~/hadoop-3.0.0-alpha4/bin/hadoop jar
hdfs-client-1.0-SNAPSHOT-jar-with-dependencies.jar $task $file
```

As shown in the following snippet, the command includes 1. the directory where the script resides, 2. operation the user wishes to execute, and 3. the HDFS file path of the file. The operation "read" and the HDFS file path are mapped to task and file variables in the script respectively.

Listing A.1: getFileFromHDFS

```
// getFileFromHDFS in MASS Place Layer
 private void getFileFromHDFS(String filename) throws IOException,
 InterruptedException {
   String[] command = { MYSCRIPT_DIRCTORY, "read", HDFS_USERFOLDER + filename};
   Process process = Runtime.getRuntime().exec(command);

   logFormattedDebug("Retrieving " + filename + " from HDFS ...");
```

```
   process.waitFor();
}
```

A.2 HDFS Client Main

The script in A.1 executes the Main method of the HDFS Client program. As mentioned in the previous sections, the script passes two arguments to the Main method, task and file. Task is a user provided operation to be executed and file is the HDFS path to locate the requested file. The Main method example below demonstrates the HDFS Client usage of "read" and "mkdir" operations.

```
public static void main(String[] args) throws IOException {
   if (args.length < 1) {
       printUsage();
       System.exit(1);
   }
   HDFSClient client = new HDFSClient();
   if (args[0].equals("read")) {
       if (args.length < 2) {
           System.out.println("Usage: hdfsclient read <hdfs_path>");
           System.exit(1);
       }
       client.readFile(args[1]);

   } else if (args[0].equals("mkdir")) {
       if (args.length < 2) {
           System.out.println("Usage: hdfsclient mkdir <hdfs_path>");
           System.exit(1);
       }
```

```
        client.mkdir(args[1]);
    } else {
        printUsage();
        System.exit(1);
    }

    System.out.println("Done!");
}
```

A.3 HDFS Client Read

Section A.2 demonstrated the "read" and "mkdir" usage; this section details the readFile method which is used for the "read" operation. The connection to HDFS has to be set up before performing any file operations. The Configuration instance locates the Hadoop directory and retrieves the defined configuration files for setting up HDFS connection. In Section 4.1.4, the issue occurred because the classpath was overwritten, which causes the Configuration instance not being able to locate the Hadoop directory. After the connection is established, all file operations will be done at the HDFS level. As shown in the example below, readFile checks the file's existence in HDFS, and uses FSDataInputStream and OutputStream to retrieve to file from HDFS to the regular file system.

```
// HDFS Client Read
public void readFile(String file) throws IOException {
    // Establish HDFS connection
    Configuration conf = new Configuration();
    FileSystem fileSystem = FileSystem.get(conf);

    Path path = new Path(file);
    if (!fileSystem.exists(path)) {
        System.out.println("File " + file + " does not exists");
```

```
        return;
    }

    FSDataInputStream in = fileSystem.open(path);

    String filename = file.substring(file.lastIndexOf('/') + 1,
            file.length());
    filename = TMP_DIRECTORY + filename;

    OutputStream out = new BufferedOutputStream(new FileOutputStream(
            new File(filename)));

    byte[] b = new byte[1024];
    int numBytes = 0;
    while ((numBytes = in.read(b)) > 0) {
        out.write(b, 0, numBytes);
    }

    in.close();
    out.close();
    fileSystem.close();
}
```

Appendix B

FILE COLLECTION - JSCH SELECTED METHODS

```
 private static void createAndAddTextTestFile(String[] inputSections) {
       try {
           String fileName = inputSections[0];
           int fileSize = Integer.parseInt(inputSections[1]);

           TextFileAssembler textFile = new TextFileAssembler(fileName, fileSize);
           promtpForMachineNumbers();
           boolean forNCFile = false;
           ArrayList<byte[]> data = downloadFile(textFile.getFilename(), forNCFile);
           textFile.createFile(data);
       } catch (IOException e) {
           System.out.println(
               "ERROR - could not create Text file: "+ e.getMessage());
       }
   }

private static void createAndAddNetcdfTestFile(String[] inputSections) {
    try {
        String fileName = inputSections[0];
        String variableName = inputSections[1];
        int[] dims = new int[inputSections.length - 2];
        for (int i = 2; i < inputSections.length; i++) {
            int dim = Integer.parseInt(inputSections[i]);
```

```
            dims[i - 2] = dim;
        }
        NetcdfFileAssembler netcdfFile =
            new NetcdfFileAssembler(fileName, variableName, dims);
        promtpForMachineNumbers();
        netcdfFile.setNumberOfMachines(numberOfMachines);
        boolean forNCFile = true;
        downloadFile(netcdfFile.getFilename(), forNCFile);
        netcdfFile.createFile(machineList);
    } catch (Exception e) {
        System.out.println(
                "Invalid input, could not create the test NetCDF file: "
                + e.getMessage());
    }
}

private static ArrayList<byte[]> downloadFile(String fileName, boolean isNCFile) {
    ArrayList<byte[]> allData = new ArrayList<byte[]>(numberOfMachines);
    for(String machineNumber:machineList) {
        try {
            JSch jsch = new JSch();
            Session session = jsch.getSession(username,
                    "uw1-320-"+machineNumber+".uwb.edu", PORT);
            session.setPassword(password);
            java.util.Properties config = new java.util.Properties();
            config.put("StrictHostKeyChecking", "no");
            session.setConfig(config);
            session.connect();
            Channel channel = session.openChannel("sftp");
```

```
            channel.connect();
            ChannelSftp channelSftp = (ChannelSftp) channel;
            channelSftp.cd(TMP_DIRECTORY);

            ByteArrayOutputStream os = new ByteArrayOutputStream();
            channelSftp.get(fileName, os);


            if(!isNCFile) { // for TxtFile
                allData.add(os.toByteArray());
            } else { // for NC File
                String toFileName = machineNumber + "_" + fileName;
                FileChannel outFileChannel =
                        new FileOutputStream(toFileName).getChannel();
                ByteBuffer buffer = ByteBuffer.wrap(os.toByteArray());
                outFileChannel.write(buffer);
                fileChannels.add(outFileChannel);
            }

         channelSftp.exit();
         session.disconnect();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
    return allData;
}

public void createFile(ArrayList<byte[]> dataList) throws IOException {
```

```
    //  createFile in TextFileAssembler.java
    FileOutputStream outFile = new FileOutputStream("__"+filename);
    for(byte[] data : dataList) {
        outFile.write(data);
    }
    outFile.close();

}

public void createFile(String[] machineList) throws InvalidArgument,
IOException,InvalidRangeException, InvalidNumberOfNodesException {
    // createFile in NetCDFFileAssembler.java
    if(filename.isEmpty()) {
        throw new InvalidArgument();
    }

    if(netcdfFileWriter == null) {
        netcdfFileWriter = NetcdfFileWriter.createNew(
            NetcdfFileWriter.Version.netcdf3, filename,null);
    } else {
        throw new RuntimeException();
    }


    Variable dataVariable = prepareAndCreateNetCDFWriteFile(machineList);
    dataOut = new ArrayFloat(shape);

    for(int nodeOrder = 0; nodeOrder < machineList.length; nodeOrder++) {
        try {
```

```
            String machineNumber = machineList[nodeOrder];
            //NetcdfFile ncFile = NetcdfFile.open();
            //faster, but limited to 2GB
            NetcdfFile ncFile = NetcdfFile.openInMemory(
                        machineNumber + "_" + filename); max
            Array data = null;
            for (Variable var : ncFile.getVariables()) {
                data = var.read(new int[]{0, 0, 0}, var.getShape());
            }
            float[] dataToWrite = (float[]) data.copyTo1DJavaArray();
            System.out.println(filename + " = " + Arrays.toString(dataToWrite));
            fillBufferWithFloatData(dataToWrite, nodeOrder);
            ncFile.close();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InvalidRangeException e) {
            e.printStackTrace();
        }
    }
    netcdfFileWriter.write(dataVariable, dataOut);
    netcdfFileWriter.close();

}
```

Appendix C
MASS HDFS APPLICATION

Appendix C focuses on the procedures to run MASS HDFS Application and the implementation details of the application program.

C.1 Execution Procedure

Similar to other MASS Java applications, the MASS HDFS application jar with dependencies should be created and placed into a directory of user's choice. The application is ran using the jar command with the following parameters: username, password, machine XML file, port number, number of processes (nodes) for the cluster, number of threads per node, and the Place dimension of x, y, and z. The script presented below is an example of executing the MASS HDFS test application on port 5555, using 2 nodes over the cluster, 4 threads on each node, and having a $8 \times 1 \times 1$ Place.

Listing C.1: MASS HDFS Application Script

```
java -Xmx20g -jar MASS_Parallel_Input_Tests.jar
user_name ********** ./machinefile.xml 5555 2 4 8 1 1
```

The machine XML file should include the following tags: master node indicator, hostname, javahome, username, MASS home path, and the private key location for SSH into different nodes. Please refer to the MASS Java Developer Guide [8] for further information regarding to creating, compiling, running, and debugging MASS application.

Listing C.2: Machinefile XML

```
<nodes>
    <node>
```

```
        <master>true</master>
        <hostname>uw1-320-03.uwb.edu</hostname>
        <javahome>/usr/bin</javahome>
        <username>dslab</username>
        <masshome>/CSSDIV/research/dslab/jas/ParallelIOTest</masshome>
        <privatekey>/CSSDIV/research/dslab/.ssh/id_rsa</privatekey>
    </node>
    <node>
        <master>false</master>
        <hostname>uw1-320-00.uwb.edu</hostname>
        <javahome>/usr/bin</javahome>
        <username>dslab</username>
        <masshome>/CSSDIV/research/dslab/jas/ParallelIOTest</masshome>
        <privatekey>/CSSDIV/research/dslab/.ssh/id_rsa</privatekey>
    </node>
</nodes>
```

C.2 Output Examples

After running the Listing C.1 script, the MASS cluster will be set up and the user will be prompted to specify the file to test in the format shown in Listing C.3.

Listing C.3: Start of MASS HDFS Application

```
java -Xmx20g -jar MA SS_Parallel_Input_Tests.jar user_name ***********
./machinefile.xml 5555 2 4 8 1 1 MProcess on uw1-320-00.uwb.edu run with
command: /usr/bin/java -Xmx20g -cp /CSSD
IV/research/dslab/jas/ParallelIOTest/*.jar
edu.uw.bothell.css.dsl.MASS.MProcess uw1-320-00.uwb.edu 1 2 4 3400
/CSSDIV/research/dslab/jas/ParallelIOTest MASS.init: done
Please specify file to test using the following format:
```

```
For txt file: <filename> <fileSizeInBytes>
For nc file: <filename> <variableName> <xDimensionSize> <yDimensionSize>
<zDimen sionSize>
Can have up to 3 dimensions in a nc file (must have at least one)
Note that the filename must end in either .nc or .txt
If the number of places per node is less than the file size then exceptions will
be thrown (expected)
You may not add a file to the same test period more than once - may cause
assertion errors if size differs
You can add any number of test files to each test period
Enter "read" to test reading each file using MASS Parallel IO
Enter "write" to test writing each file using MASS Parallel IO
Enter "exit" to end the application
```

The following example demonstrates the NetCDF file read, write, and exit calls. For file write, the user can verify the data integrity in the /tmp/TestFiles/ directory upon completion of the task.

Listing C.4: NetCDF Example Output

```
-> 50MB_ncfile.nc data 232 232 232
Retrieving 50MB_ncfile.nc from HDFS ...
**********************************************************************
SUCCESS retrieving test NetCDF file: /tmp/TestFiles/50MB_ncfile.nc
**********************************************************************
/tmp/TestFiles/50MB_ncfile.nc added to test files.
-> read
open time = 300959575
read time = 8782015
total time = 309741749
**********************************************************************
```

```
SUCCESS using MASS to open, and read /tmp/TestFiles/50MB_ncfile.nc
*****************************************************************
-> write
*****************************************************************
SUCCESS using MASS to open, and write /tmp/TestFiles/50MB_ncfile.nc
*****************************************************************
-> exit
```

Similar procedures are used for testing text files as they are for testing NetCDF files. Instead of providing variable name and the shape dimensions, the user will specify the file name and the file size in bytes for testing text file:

Listing C.5: Text Example Output

```
-> 50MB_file.txt 50000000
Retrieving 50MB_ncfile.nc from HDFS ...
*****************************************************************
SUCCESS retrieving test TXT file: /tmp/TestFiles/50MB_file.txt
*****************************************************************
/tmp/TestFiles/50MB_file.txt added to test files.
-> read
open time = 30147319
read time = 6766655
total time = 36914371
*****************************************************************
SUCCESS using MASS to open, and read /tmp/TestFiles/50MB_file.txt
*****************************************************************
-> write
*****************************************************************
SUCCESS using MASS to open, and write /tmp/TestFiles/50MB_file.txt
*****************************************************************
```

```
-> exit
```

C.3 Application Code

This section we detailed the implementation of the application code. The application starts with the application main (Listing C.6). The 9 arguments (mainArgs) are the parameters presented in Listing C.1. If the arguments are correctly provided, the program is executed in the following order: 1. Setting up the MASS cluster through initMASS, 2. Run the application in runAppShellToCreateAndTestFilesWithMass method, 3. Cleanup and exit the application using MASS.finish(). We suggest our readers to locate the initMASS explanation under Section 4.4, and we also suggest our readers to refer to the MASS Java Manual [9] for more information on the MASS.init and MASS.finish. This section will only focus on the implementation of the test application.

Listing C.6: Main: Main and initMASS

```
public static void main(String[] args) {
    mainArgs = args;
    if (mainArgs.length != 9) {
        printExpectedArguementsOrder();
    } else {
        initMASS();
        runAppShellToCreateAndTestFilesWithMass();
        MASS.finish();
    }
}

private static void initMASS() {
    String[] massArgs = new String[4];
    massArgs[0] = mainArgs[USERNAME];
```

```
    massArgs[1] = mainArgs[PASSWORD];
    massArgs[2] = mainArgs[MACHINE_FILEPATH];
    massArgs[3] = mainArgs[PORT];

    MASS.init(massArgs, Integer.parseInt(mainArgs[NUM_PROCESSES]),
    Integer.parseInt(mainArgs[NUM_THREADS]));

    int x = Integer.parseInt(mainArgs[X_PLACES]);
    int y = Integer.parseInt(mainArgs[Y_PLACES]);
    int z = Integer.parseInt(mainArgs[Z_PLACES]);

    NETCDF_TEST_PLACES = new Places(1, "NetcdfTestPlace", null, x, y, z);
    TXT_TEST_PLACES = new Places(2, "TxtTestPlace", null, x, y, z);
}
```

RunAppShellToCreateAndTestFilesWithMass starts out by printing an application start message. This is the message shown in Listing C.3. The user has the choice to clear the test files directory at the start or the end of the process. Both CLEAR_FILES_ON_START and CLEAR_FILES_ON_FINISH are both global boolean variables in the application level. Based on our testing purpose, both variables are set to false. The createTestFilesDirectory method simply calls the FILE_IO class the create the test file directory if the directory does not exist. See Listing C.8 for promptParseAndExecuteUserInput.

Listing C.7: Main: Test Application Method

```
private static final boolean CLEAR_FILES_ON_START = false;
private static final boolean CLEAR_FILES_ON_FINISH = false;

private static void runAppShellToCreateAndTestFilesWithMass() {
    printApplicationStartMessage();
```

```
    if (CLEAR_FILES_ON_START) {
        clearTestFilesDirectory();
    }

    createTestFilesDirectory();
    promptParseAndExecuteUserInput();
    if (CLEAR_FILES_ON_FINISH) {
        clearTestFilesDirectory();
    }
}

private static void createTestFilesDirectory() {
    try {
        FILE_IO.createTestFilesDirectory();
    } catch (Exception e) {
        logger.error(e.toString());
    }
}
```

As the name described, promptParseAndExecuteUserInput parses the user inputs and executes the requests. As shown in Listing C.5, the user first specifies the file information followed by a single word command: read, write, or exit. These inputs are passed to the parseAndRespondToUserInput method as a String array, inputSections. The method determines the command and the file type based on the input length.

Listing C.8: Main: Parses and Executes User Input

```
private static void promptParseAndExecuteUserInput() {
    Scanner scanner = new Scanner(System.in);
    while (true) {
        System.out.print("-> ");
```

```
        String input = scanner.nextLine();
        if (checkToExit(input)) {
            break;
        }
        parseAndRespondToUserInput(input);
    }
}

private static void parseAndRespondToUserInput(String input) {
    String[] inputSections = input.split(" ");
    if (inputSections.length == 1) {
        checkToExecuteTests(inputSections[0]);
    } else if (inputSections.length == 2) {
        createAndAddTestFile(inputSections); // this is txt file
    } else if (inputSections.length > 2 && inputSections.length <= 11) {
        createAndAddNetcdfTestFile(inputSections); // this is netcdf file
    } else {
        System.out.println("Invalid input.");
    }
}
```

We will first look at createAndAddTestFile and createAndAddNetcdfTestFile methods. Both methods call the TestFile.factory which returns either a TxtTestFile or a NetcdfTestFile based on the file extension provided (Listing C.9 TestFile factory). Once the TestFile object is created, it will be placed into the file table (addTestFileToTestFileStorage) for later use.

Listing C.9: Main: Create and Add Test File

```
// In Main class; for Text
private static void createAndAddTestFile(String[] inputSections) {
    try {
```

```
        String fileName = inputSections[0];
        int fileSize = Integer.parseInt(inputSections[1]);
        TestFile testFile = TestFile.factory(fileName, fileSize);
        addTestFileToTestFileStorage(testFile);
    } catch (Exception e) {
        System.out.println("Invalid input, could not create the test Txt file: "
        + e.getMessage());
    }
}

// In Main class; for NetCDF
private static void createAndAddNetcdfTestFile(String[] inputSections) {
    try {
        String fileName = inputSections[0];
        String variableName = inputSections[1];
        int[] dims = new int[inputSections.length - 2];
        for (int i = 2; i < inputSections.length; i++) {
            int dim = Integer.parseInt(inputSections[i]);
            dims[i - 2] = dim;
        }
        TestFile testNcFile = new NetcdfTestFile(fileName, variableName, dims);
        addTestFileToTestFileStorage(testNcFile);
    } catch (Exception e) {
        System.out.println("Invalid input, could not create the test NetCDF file: "
        + e.getMessage());
    }
}

// In Main class
```

```
private static void addTestFileToTestFileStorage(TestFile testFile) {
    if (!testFiles.containsKey(testFile.getFilepath())) {
        testFiles.put(testFile.getFilepath(), testFile);
    } else {
        System.out.println("Cannot add the same file more than once.");
    }
}

// In TestFile class (App File Layer)
public static TestFile factory(String filename, int fileSize) throws Exception {
    if (fileSize < 0) {
        throw new RuntimeException("File size is less than zero.");
    }
    if (filename.toLowerCase().endsWith(".nc")) {
        return new NetcdfTestFile(filename, "var", new int[] { fileSize });
    }
    if (filename.toLowerCase().endsWith(".txt")) {
        return new TxtTestFile(filename, fileSize);
    }
    throw new RuntimeException("File type not supported.");
}
```

Listing C.10 and C.11 present the constructors of TxtTestFile class and NetcdfTestFile class. Notice, these classes are in the App File layer of Figure 4.5. The constructors call the getXXXFileFromHDFS method to retrieve the files from HDFS to the master node only. This step not only asserts the existence of the requesting file, but also serves to create a holder for the TestFile object. Methods, getTxtFileFromHDFS and getNetcdfFileFromHDFS, are identical to the getFileFromHDFS method mentioned in Listing A.1.

Listing C.10: TestFile: TxtTestFile Constructors

```
// TxtTestFile constructor
public TxtTestFile(String filename, int fileSize) throws Exception {
    super(filename, fileSize, Main.TXT_TEST_PLACES);
    if (!filename.toLowerCase().endsWith(".txt")) {
        throw new RuntimeException("Txt file must end with .txt");
    }

    getTxtFileFromHDFS(filename);
    if (!FILE_IO.fileExists(filepath)) {
        throw new FileNotFoundException(
            "The given file to open does not exist: " + filepath);

    } else {
        Main.printAndLogDebug(String.format(
            "*****************************************"));
        Main.printAndLogDebug(String.format(
            "SUCCESS writing test TXT file: %s", filepath));
        Main.printAndLogDebug(String.format(
            "*****************************************"));
    }

    Main.printAndLogDebug(filepath + " added to test files.");
}
private void getTxtFileFromHDFS(String filename)
    throws IOException,InterruptedException {
    String[] command = { "/tmp/myscript", "read ",
                         Main.HDFS_USERFOLDER + filename};
    Main.printAndLogDebug("Retrieving " + filename + " from HDFS ...");
```

```
    Process process = Runtime.getRuntime().exec(command);
    process.waitFor();
}
```

Listing C.11: TestFile: NetcdfTestFile Constructors

```
// NetcdfTestFile constructor
public NetcdfTestFile(String filename, String variableName, int[] shape)
    throws Exception {
    super(filename, Main.NETCDF_TEST_PLACES);
    if (!filename.toLowerCase().endsWith(".nc")) {
        throw new RuntimeException("Netcdf file must end with .nc");
    }
    this.variableName = variableName;
    this.shape = shape;
    this.fileSize = getFileSizeFromShape(shape);

    getNetcdfFileFromHDFS(filename);
    if (!FILE_IO.fileExists(filepath)) {
        Main.printAndLogDebug("File doesn't exist :: " + filepath);
        throw new FileNotFoundException(
            "The given file to open does not exist: " + filepath);
    } else {
    Main.printAndLogDebug(String.format(
        "************************************************"));
    Main.printAndLogDebug(String.format(
        "SUCCESS retrieving test NetCDF file: %s", filepath));
    Main.printAndLogDebug(String.format(
        "************************************************"));
    }
```

```
    Main.printAndLogDebug(filepath + " added to test files.");

}
private void getNetcdfFileFromHDFS(String filename)
    throws IOException, InterruptedException {
    String[] command = { Main.MYSCRIPT_DIRCTORY, "read ",
                        Main.HDFS_USERFOLDER + filename};
    Process process = Runtime.getRuntime().exec(command);

    Main.printAndLogDebug("Retrieving " + filename + " from HDFS ...");
    process.waitFor();
}
```

Now, the test file object has been created and stored successfully, we will then look at the checkToExecuteTests method appeared in Listing C.8. This method determines the I/O type and calls executeTest to process the operation. In method executeParallelInputFileTests, the stored test file object will be retrieved, and the operation will be performed based on the I/O type. Note, all the codes presented at this point resides in the Main layer of Figure 4.5. We cover the testMASSparallelInputForRead and the testMASSparallelInputForWrite in the next section, C.3.1.

Listing C.12: Main: Execute Tests

```
private static void checkToExecuteTests(String inputSection) {
    boolean error = false;
    if (inputSection.equalsIgnoreCase("read")) {
        ioType = READ;
    } else if(inputSection.equalsIgnoreCase("write")) {
        ioType = WRITE;
    } else {
```

```
            System.out.println("Invalid input.");
            error = true;
        }
        if(!error) {
            if (ioType == READ || ioType == WRITE) {
                try {
                    executeTest();
                } catch (Exception e) {
                    System.out.println(e.getMessage());
                    testFiles.clear();
                }
            }
        }
    }

    private static void executeTest() {
        try {
            executeParallelInputFileTests();
        } catch (Exception e) {
            printAndLogDebug(String.format("An exception occurred: %s", e.toString()));
        }
    }

    private static void executeParallelInputFileTests() throws Exception {
        for (String testFilePath : testFiles.keySet()) {
            TestFile testFile = testFiles.get(testFilePath);
            if(ioType == READ) {
                testMASSparallelInputForRead(testFile);
            } else if(ioType == WRITE) {
```

```
            testMASSparallelInputForWrite(testFile);
        }
        testFile.closeFileWithMASS();
    }
    testFiles.clear();
}
```

C.3.1 Test MASS Parallel Input For Read and Write

Methods, testMASSparallelInputForRead and testMASSparallelInputForWrite, are invoked by method executeParallelInputFileTests in Listing C.12. For the demonstration purposes, we commented out the time measuring code for higher readability. TestFile is polymorphic; each of the TestFile function calls may invoke the method in TxtTestFile or NetcdfTestFile. We examine the functions in TxtTestFile and NetcdfTestFile in the next sections.

Listing C.13: Main: Test Parallel Input For Read

```
private static void testMASSparallelInputForRead(TestFile fileTester)
throws Exception {
   //long openstartTime = System.nanoTime();
   fileTester.openFileWithMASS();
   //long openendTime = System.nanoTime();
   //long readstartTime = System.nanoTime();
   fileTester.readFileWithMASS();
   //long readendTime = System.nanoTime();

   //divide by 1000000 to get milliseconds.
   //long openduration = (openendTime - openstartTime);
   //long readduration = (readendTime - readstartTime);
   //long openandreadduration = (readendTime - openstartTime);
```

```
    //printAndLogDebug("open time = " + openduration);
    //printAndLogDebug("read time = " + readduration);
    //printAndLogDebug("total time = " + openandreadduration);

    fileTester.openReadAndCloseFileWithoutMASS();
    fileTester.createSharedFileBuffer();
    fileTester.assertReadWithMASSequalsReadWithoutMASS();

    printAndLogDebug("*********************************************************");
    printAndLogDebug(String.format("SUCCESS using MASS to open, and read %s",
    printAndLogDebug("*********************************************************");
}

private static void testMASSparallelInputForWrite(TestFile fileTester)
throws Exception {
    fileTester.openFileWithMASS();
    fileTester.readFileWithMASS();
    fileTester.openReadAndCloseFileWithoutMASS();
    fileTester.createSharedFileBuffer();
    fileTester.assertReadWithMASSequalsReadWithoutMASS();
    fileTester.writeFileWithMASS();
    printAndLogDebug("*********************************************************");
    printAndLogDebug(String.format("SUCCESS using MASS to open, and write %s",
    printAndLogDebug("*********************************************************");
}
```

C.3.2 App File Layer: TxtTestFile

TxtTestFile class resides in the App File layer (Figure 4.5). The constructor was shown in Listing C.10 and the rest of the class is implemented with override methods that are used in Listing C.13. As shown in the code below, each method call invokes places.callAll. This was mentioned at the end of Section 4.4. The callAll method takes 2 parameters: 1. user defined method and 2. argument wrapper.

Listing C.14: App File Layer: TxtTestFile

```
@Override
public void assertReadWithMASSequalsReadWithoutMASS() {
    Object readTestWrapper;
    Object[] readTestArgs = new Object[2];
    readTestArgs[0] = fileDescriptor;
    readTestArgs[1] = txtBufferReadWithoutMASS;
    readTestWrapper = readTestArgs;
    places.callAll(ParallelInputTestPlace.testExpectedVsSharedFileBuffer_,
        readTestWrapper);
}

@Override
public void readFileWithMASS() {
    Object txtReadArgsWrapper;
    Object[] txtReadArgs = new Object[1];
    txtReadArgs[0] = fileDescriptor;
    txtReadArgsWrapper = txtReadArgs;
    places.callAll(ParallelInputTestPlace.readFileIntoPlaceBuffer_,
        txtReadArgsWrapper);
}
```

```
@Override
public void writeFileWithMASS() {
    Object txtWriteArgsWrapper;
    Object[] txtfWriteArgs = new Object[2];
    txtfWriteArgs[0] = filepath;
    txtWriteArgsWrapper = txtfWriteArgs;
    places.callAll(ParallelInputTestPlace.writeFileIntoPlaceBuffer_,
        txtWriteArgsWrapper);
}

@Override
public void openReadAndCloseFileWithoutMASS() throws IOException {
    FileChannel fileChannel;
    fileChannel = FileChannel.open(Paths.get(filepath), OPEN_OPTIONS[0]);
    ByteBuffer buffer = ByteBuffer.allocate((int) fileChannel.size());
    fileChannel.read(buffer);
    txtBufferReadWithoutMASS = buffer.array();
    fileChannel.close();
}

@Override
public void createSharedFileBuffer() {
    Object sharedTxtArgsWrapper;
    Object[] sharedTxtArgs = new Object[2];
    sharedTxtArgs[0] = fileDescriptor;
    sharedTxtArgs[1] = fileSize;
    sharedTxtArgsWrapper = sharedTxtArgs;
    places.callAll(ParallelInputTestPlace.createSharedFileBuffer_,
        sharedTxtArgsWrapper);
```

```
}
```

Each user-defined method in Listing C.14 are implemented in ParallelInputTestPlace and TxtTestPlace. Both classes are in the App Place layer. Although, App Place layer and MASS Place layer are separated in Figure 4.5, the two layers represent the same object from the class perspective. The TxtTestPlace extends from ParallelInputTestPlace, and the ParallelInputTestPlace is an extension from the MASS Place. In other words, the App Place is a MASS Place because the App Place extends from the MASS Place. This implementation design may be complex but efficiently reduces code redundancy. The user-defined methods can be found in ParallelInputTestPlace class executeCallMethod method. Please refer to Section 4.2 and 4.3 for the purpose of each method. For now, we will focus on Listing C.15 and C.16 to see how each user-defined method is implemented.

- openFileInMemoryForAllPlacesToAccess
  Calls the openFileInMemoryForAllPlacesToAccess() method in ParallelInputTestPlace class, then calls the MASS Place open method and stores the returned file descriptor.

- readFileIntoPlaceBuffer
  Calls the readFileIntoPlaceBuffer() method that calls the override method, readFileIntoThisPlaceBuffer, in TxtTestPlace. This method, which can be found in Listing C.16, then calls the MASS Place read method to proceed the read operation.

- createSharedFileBuffer
  Calls the createSharedFileBuffer() method, which invokes the override method, copyThisPlaceBufferToSharedPlaceBuffer. This method that can be found in the TxtTestPlace class focuses on copying the data from Places buffer to a globally shared buffer. This operation does not carry to the MASS Place layer as shown in Figure 4.5.

- testExpectedVsSharedFileBuffer
  Calls the testRead() method. This command is used for assertion and development

purposes. The user can find the testRead method implementation in Listing C.15.

- writeFileIntoPlaceBuffer

  Calls the writeFileIntoPlaceBuffer() method. In writeFileIntoPlaceBuffer(), the method first retrieves the data to be written from the shared buffer and then invokes the override method, writeFileIntoThisPlaceBuffer, in TxtTestPlace (Listing C.16). This method then calls the MASS Place write method to proceed the write operation.

- closeFile

  Call the closeFile() method. This method removes the buffers and invokes MASS Place close to complete the close operation.

Please refer to Section 4.2 and 4.3 for MASS Place layer implementation.

Listing C.15: App Place Layer: ParallelInputTestPlace

```
public abstract class ParallelInputTestPlace extends Place {
    // Called by callAll()
    public Object callMethod(int functionId, Object arg) {
        try {
            executeCallMethod(functionId, arg);
        } catch (Exception e) {
            String errorMessage = String.format(
                "An exception occurred in place %s: %s", this, e.toString());
            if (isMasterNode()) {
                Main.printAndLogError(errorMessage);
            } else {
                logger.error(errorMessage);
            }
        }
        return null;
    }
```

```
private void executeCallMethod(int functionId, Object arg)
    throws Exception {
    switch (functionId) {
        case openFileInMemoryForAllPlacesToAccess_:
            openFileInMemoryForAllPlacesToAccess(arg);
            break;
        case readFileIntoPlaceBuffer_:
            readFileIntoPlaceBuffer(arg);
            break;
        case createSharedFileBuffer_:
            createSharedFileBuffer(arg);
            break;
        case testExpectedVsSharedFileBuffer_:
            testRead(arg);
            break;
        case writeFileIntoPlaceBuffer_:
            writeFileIntoPlaceBuffer(arg);
            break;
        case closeFile_:
            closeFile(arg);
            break;
    }
}

public void openFileInMemoryForAllPlacesToAccess(Object arg)
    throws Exception {
    String filepath = (String) arg;
    int fileDescriptor = open(filepath, 0); // Place open
```

```
    if (thisPlacesFileBuffers.containsKey(fileDescriptor)) {
        throw new RuntimeException(this + "
            file descriptor is not unique: " + fileDescriptor);
    }
    thisPlacesFileBuffers.put(fileDescriptor, new Object());
    this.fileDescriptor = fileDescriptor;
}


public void readFileIntoPlaceBuffer(Object arg) throws Exception {
    Object[] readArgs = (Object[]) arg;
    readFileIntoThisPlaceBuffer(readArgs);
}

public abstract void readFileIntoThisPlaceBuffer(
 Object[] readArgs) throws Exception;


public void writeFileIntoPlaceBuffer(Object arg) throws Exception {
    Object[] writeArgs = (Object[]) arg;

    if (sharedFileBuffers.containsKey(fileDescriptor)) {
        Object writeDataBuffer = sharedFileBuffers.get(fileDescriptor);
        writeArgs[writeArgs.length - 1] = writeDataBuffer;
        writeFileIntoThisPlaceBuffer(writeArgs);
    }

}
```

```
public abstract void writeFileIntoThisPlaceBuffer(
    Object[] writeArgs) throws Exception;

public void createSharedFileBuffer(Object arg) {
    Object[] sharedArgs = (Object[]) arg;
    int fileDescriptor = (int) sharedArgs[0];
    long fileBufferSize = (long) sharedArgs[1];

    if (!thisPlacesFileBuffers.containsKey(fileDescriptor)) {
        throw new RuntimeException("File descriptor " +
            fileDescriptor + " must be read by each place before
                creating the shared buffer.");
    }
    synchronized (sharedFileBuffers) {
        initSharedBuffer(fileDescriptor, fileBufferSize);
        Object sharedFileBuffer =
            sharedFileBuffers.get(fileDescriptor);
        Object thisPlacesFileBuffer =
            thisPlacesFileBuffers.get(fileDescriptor);
        copyThisPlaceBufferToSharedPlaceBuffer(
            thisPlacesFileBuffer, sharedFileBuffer);
    }
}

public abstract void copyThisPlaceBufferToSharedPlaceBuffer(
    Object thisPlacesFileBuffer, Object sharedFileBuffer);

protected int getTotalPlaces() {
    return MASSBase.getCurrentPlacesBase().getNumberOfPlacesOnCurrentNode();
```

```
}

protected int getPlaceReadLength(int bufferSize) {
    return bufferSize / getTotalPlaces();
}

public int getCurrentFileDescriptor() {
    return fileDescriptor;
}

private void initSharedBuffer(int fileDescriptor, long bufferSize) {
    if (!sharedFileBuffers.containsKey(fileDescriptor)) {
        Object sharedBuffer =
        createSharedBuffer(getBufferSizeBasedOnNumberOfNodes(bufferSize));
        sharedFileBuffers.put(fileDescriptor, sharedBuffer);
    }
}

public abstract Object createSharedBuffer(int bufferSize);

private int getBufferSizeBasedOnNumberOfNodes(long bufferSize) {
    int totalNodes = MASSBase.getSystemSize();
    int sharedBufferSize = (int) (bufferSize / totalNodes);
    if (MASSBase.getMyPid() == totalNodes - 1) {
        sharedBufferSize += bufferSize % totalNodes;
    }
    return sharedBufferSize;
}
```

```
public void testRead(Object arg) throws Exception {
    Object[] testReadArgs = (Object[]) arg;
    int fileDescriptor = (int) testReadArgs[0];
    Object expectedData = testReadArgs[1];
    setThisNodesExpectedFileBuffer(fileDescriptor, expectedData);
    testThisNodesBufferData(fileDescriptor);
}

private void setThisNodesExpectedFileBuffer(
    int fileDescriptor, Object expectedFileBuffer) {
    synchronized (expectedSharedFileBuffers) {
        if (!expectedSharedFileBuffers.containsKey(fileDescriptor)) {
            Object thisNodesExpectedNetcdfFileBuffer =
            createExpectedSharedBufferForThisNode(expectedFileBuffer);
            expectedSharedFileBuffers.put(fileDescriptor,
            thisNodesExpectedNetcdfFileBuffer);
        }
    }
}

public abstract Object createExpectedSharedBufferForThisNode(
    Object expectedSharedBuffer);

private void testThisNodesBufferData(int fileDescriptor)
        throws Exception {
    synchronized (sharedFileBuffers) {
        if (sharedFileBuffers.containsKey(fileDescriptor) &&
            expectedSharedFileBuffers.containsKey(fileDescriptor)) {
            Object actualFileBuffer =
```

```
                sharedFileBuffers.get(fileDescriptor);
            Object expectedFileBuffer =
                expectedSharedFileBuffers.get(fileDescriptor);

            try {
                assertExpectedEqualsActualBufferData(
                    expectedFileBuffer, actualFileBuffer);
            } finally {
            }
        }
    }
}

public abstract void assertExpectedEqualsActualBufferData(
    Object expectedSharedBufferData, Object actualSharedBufferData)
        throws Exception;

public void closeFile(Object arg) throws IOException {
    int fileDescriptor = (int) arg;
    sharedFileBuffers.remove(fileDescriptor);
    expectedSharedFileBuffers.remove(fileDescriptor);

    if (!close(fileDescriptor)) { // Place close
        logger.error(this + "
        failed to close the file with the file descriptor "
            + fileDescriptor);
    }
}
```

```
    protected int getExpectedFileSizeForThisNode(int entireFileSize) {
        int sizePerNode = getExpectedFileSizeOffset(entireFileSize);
        int totalNodes = MASSBase.getSystemSize();
        if (MASSBase.getMyPid() == totalNodes - 1) {
            sizePerNode += entireFileSize % totalNodes;
        }
        return sizePerNode;
    }

    protected int getExpectedFileSizeOffset(int entireFileSize) {
        int totalNodes = MASSBase.getSystemSize();
        return entireFileSize / totalNodes;
    }

    private boolean isMasterNode() {
        return MASSBase.getMyPid() == 0;
    }
}
```

Listing C.16: App Place Layer: TxtTestPlace

```
@Override
public void readFileIntoThisPlaceBuffer(Object[] readArgs) throws Exception {
    if (readArgs.length != 1) {
        throw new IllegalArgumentException(
        "One argument expected for Txt Read. Number of args: "
            + readArgs.length);
    }

    int fileDescriptor = (int) readArgs[0];
```

```
    if (!thisPlacesFileBuffers.containsKey(fileDescriptor)) {
        throw new RuntimeException(this +
            " file descriptor to read has not been opened: "
                + fileDescriptor);
    }
    byte[] thisPlacestxtReadData = read(fileDescriptor); // Place Read
    thisPlacesFileBuffers.put(fileDescriptor, thisPlacestxtReadData);
}

@Override
public byte[] createSharedBuffer(int bufferSize) {
    return new byte[bufferSize];
}

@Override
public void copyThisPlaceBufferToSharedPlaceBuffer(
        Object thisPlacesFileBuffer, Object sharedFileBuffer) {
    if (sharedFileBuffer instanceof byte[]
        && thisPlacesFileBuffer instanceof byte[]) {
        byte[] txtSharedFileBuffer = (byte[]) sharedFileBuffer;
        byte[] txtThisPlacesFileBuffer = (byte[]) thisPlacesFileBuffer;

        int placeReadLength = getPlaceReadLength(
            txtSharedFileBuffer.length);

        for (int myIndex = 0, sharedIndex =
        getPlaceOrderPerNode() * placeReadLength;
        myIndex < txtThisPlacesFileBuffer.length;
            myIndex++, sharedIndex++) {
```

```
            txtSharedFileBuffer[sharedIndex] =
                txtThisPlacesFileBuffer[myIndex];
        }

    } else {
        throw new RuntimeException(
        "txt buffers for file descriptor are not a supported type.");
    }
}

@Override
public Object createExpectedSharedBufferForThisNode(Object
    expectedSharedBuffer) {
    if (expectedSharedBuffer instanceof byte[]) {
        byte[] txtExpectedSharedBuffer = (byte[]) expectedSharedBuffer;
        int fileOffset =
            getExpectedFileSizeOffset(txtExpectedSharedBuffer.length);
        int fileSize =
            getExpectedFileSizeForThisNode(
                txtExpectedSharedBuffer.length);
        return Arrays.copyOfRange(txtExpectedSharedBuffer,
            fileOffset * MASSBase.getMyPid(),
            (fileOffset * MASSBase.getMyPid()) + fileSize);
    } else {
        throw new RuntimeException(
            "txt expected shared buffer is an invalid type.");
    }
}
```

```
@Override
public void assertExpectedEqualsActualBufferData(
    Object expectedSharedBufferData,
    Object actualSharedBufferData) throws Exception {
    if (expectedSharedBufferData instanceof byte[] &&
        actualSharedBufferData instanceof byte[]) {
        byte[] txtExpectedSharedBufferData =
            (byte[]) expectedSharedBufferData;
        byte[] txtActualSharedBufferData =
            (byte[]) actualSharedBufferData;
        try {
            Assert.assertArrayEquals(txtExpectedSharedBufferData,
                txtActualSharedBufferData);
        } catch (AssertionError e) {
            throw new RuntimeException("ASSERTION FAILED");
        }
    } else {
        throw new RuntimeException(
            "Cannot assert buffer data: txt buffer type is invalid.");
    }
}

@Override
public void writeFileIntoThisPlaceBuffer(Object[] writeArgs)
    throws Exception {
    if (writeArgs.length != 2) {
        throw new IllegalArgumentException(
        "Two argument expected for Txt Write. Number of args: "
            + writeArgs.length);
```

```
    }
    String filePath = (String) writeArgs[0];
    byte[] writeDataBuffer = (byte[])writeArgs[1];
    if(writeDataBuffer instanceof byte[]) {
        openForWrite(filePath, writeDataBuffer.length);
        write(fileDescriptor, writeDataBuffer); // Place Write
    }
}
```

C.3.3 App File Layer: NetcdfTestFile

Similar to TxtTestFile, the NetcdfTestFile class also resides in the App File layer and its constructor was shown in Listing C.11. The rest of the class is implemented with override methods that are used in Listing C.13. As shown in the code below, each method call invokes places.callAll and the explanation can be found at the end of Section 4.4. The callAll method takes a user defined method and an argument wrapper.

Listing C.17: App File Layer: NetcdfTestFile

```
@Override
public void assertReadWithMASSequalsReadWithoutMASS() {
    Object assertTestWrapper;
    Object[] assertTestArgs = new Object[2];
    assertTestArgs[0] = fileDescriptor;
    assertTestArgs[1] = netcdfBufferReadWithoutMASS;
    assertTestWrapper = assertTestArgs;
    places.callAll(ParallelInputTestPlace.testExpectedVsSharedFileBuffer_,
        assertTestWrapper);
}
```

```
@Override
public void createSharedFileBuffer() {
    Object sharedNetcdfArgsWrapper;
    Object[] sharedNetcdfArgs = new Object[2];
    sharedNetcdfArgs[0] = fileDescriptor;
    sharedNetcdfArgs[1] = fileSize;
    sharedNetcdfArgsWrapper = sharedNetcdfArgs;
    places.callAll(ParallelInputTestPlace.createSharedFileBuffer_,
        sharedNetcdfArgsWrapper);
}


@Override
public void readFileWithMASS() {
    Object netcdfReadArgsWrapper;
    Object[] netcdfReadArgs = new Object[2];
    netcdfReadArgs[0] = fileDescriptor;
    netcdfReadArgs[1] = variableName;
    netcdfReadArgsWrapper = netcdfReadArgs;
    places.callAll(ParallelInputTestPlace.readFileIntoPlaceBuffer_,
        netcdfReadArgsWrapper);
}

@Override
public void writeFileWithMASS() {
    Object netcdfWriteArgsWrapper;
    Object[] netcdfWriteArgs = new Object[5];
    netcdfWriteArgs[0] = fileDescriptor;
    netcdfWriteArgs[1] = variableName;
```

```
    netcdfWriteArgs[2] = shape;
    netcdfWriteArgs[3] = filepath;
    netcdfWriteArgsWrapper = netcdfWriteArgs;
    places.callAll(ParallelInputTestPlace.writeFileIntoPlaceBuffer_,
        netcdfWriteArgsWrapper);
}

@Override
public void openReadAndCloseFileWithoutMASS()
    throws IOException, InvalidRangeException {
    NetcdfFile ncFile = NetcdfFile.open(filepath);
    // openInMemory limited to 2GB max
    //NetcdfFile ncFile = NetcdfFile.openInMemory(filepath);
    Array data = null;
    for (Variable var : ncFile.getVariables()) {
        data = var.read(new int[]{0, 0, 0}, var.getShape());
    }
    netcdfBufferReadWithoutMASS = (float[]) data.copyTo1DJavaArray();
    ncFile.close();
}

@Override
public void setFileDescriptor() {
    fileDescriptor = ++fileDescriptorIndex;
}

private int getFileSizeFromShape(int[] shape) {
    int size = 1;
    for (int i : shape) {
```

```
        size *= i;
    }
    return size;
}
```

Each user-defined method in Listing C.17 are implemented in ParallelInputTestPlace and NetcdfTestPlace. As mentioned in Section C.3.2, both classes are in the App Place layer. The NetcdfTestPlace extends from the ParallelInputTestPlace, and the ParallelInputTestPlace is itself an extension of the MASS Place. The user-defined methods can be found in ParallelInputTestPlace class executeCallMethod method (See Listing C.15). The readers may refer to Section 4.2 and 4.3 for the purpose of each method. We will focus on Listing C.15 and C.18 to see how each user-defined method is implemented.

- openFileInMemoryForAllPlacesToAccess
  Calls the openFileInMemoryForAllPlacesToAccess() method in ParallelInputTestPlace class, then calls the MASS Place open method and stores the returned file descriptor.

- readFileIntoPlaceBuffer
  Calls the readFileIntoPlaceBuffer() method. This method then calls the override method, readFileIntoThisPlaceBuffer, in NetcdfTestPlace. This method, can be found in Listing C.18, then calls the MASS Place read method to proceed the read operation.

- createSharedFileBuffer
  Calls the createSharedFileBuffer() method, which invokes the override method, copyThisPlaceBufferToSharedPlaceBuffer. This method that can be found in the NetcdfTestPlace class focuses on copying the data from Places buffer to a globally shared buffer. This operation does not carry to the MASS Place layer as shown in Figure 4.5.

- testExpectedVsSharedFileBuffer

Calls the testRead() method. This command is used for assertion and development purposes. The user can find the testRead method implementation in Listing C.15.

- writeFileIntoPlaceBuffer

  Calls the writeFileIntoPlaceBuffer() method. In writeFileIntoPlaceBuffer(), the method first retrieves the data to be written from the shared buffer and then invokes the override method, writeFileIntoThisPlaceBuffer, in NetcdfTestPlace (Listing C.18). This method then calls the MASS Place openForWrite and write methods to proceed the write operation. The openForWrite method was described at the end of Section 4.4.

- closeFile

  Call the closeFile() method. This method removes the buffers and invokes MASS Place close to complete the close operation.

Further information of MASS Place layer can be found in Section 4.2 and 4.3.

Listing C.18: App Place Layer: NetcdfTestPlace

```
@Override
public void readFileIntoThisPlaceBuffer(Object[] readArgs) throws Exception {
    if (readArgs.length != 2) {
        throw new IllegalArgumentException(
        "Two arguments expected for Netcdf Read. Number of args: "
        + readArgs.length);
    }
    int fileDescriptor = (int) readArgs[0];
    String variableName = (String) readArgs[1];

    if (!thisPlacesFileBuffers.containsKey(fileDescriptor)) {
        throw new RuntimeException(this +
        " file descriptor to read has not been opened: " + fileDescriptor);
```

```
    }

    float[] thisPlacesNetcdfReadData =
        (float[]) read(fileDescriptor, variableName); // Place read
    thisPlacesFileBuffers.put(fileDescriptor, thisPlacesNetcdfReadData);
}


@Override
public float[] createSharedBuffer(int bufferSize) {
    return new float[bufferSize];
}

@Override
public void copyThisPlaceBufferToSharedPlaceBuffer(
    Object thisPlacesFileBuffer, Object sharedFileBuffer) {
    if (sharedFileBuffer instanceof float[] &&
        thisPlacesFileBuffer instanceof float[]) {
        float[] netcdfSharedFileBuffer = (float[]) sharedFileBuffer;
        float[] netcdfThisPlacesFileBuffer = (float[]) thisPlacesFileBuffer;

        int placeReadLength = getPlaceReadLength(netcdfSharedFileBuffer.length);

        for (int myIndex = 0;
            sharedIndex = getPlaceOrderPerNode() * placeReadLength; myIndex <
            netcdfThisPlacesFileBuffer.length;
                myIndex++, sharedIndex++) {
            netcdfSharedFileBuffer[sharedIndex] = netcdfThisPlacesFileBuffer[myIndex];
        }
```

```
    } else {
        throw new RuntimeException(
            "Netcdf buffers for file descriptor are not a supported type.");
    }
}

@Override
public Object createExpectedSharedBufferForThisNode(Object expectedSharedBuffer) {
    if (expectedSharedBuffer instanceof float[]) {
        float[] netcdfExpectedSharedBuffer = (float[]) expectedSharedBuffer;
        int fileOffset =
            getExpectedFileSizeOffset(netcdfExpectedSharedBuffer.length);
        int fileSize =
            getExpectedFileSizeForThisNode(netcdfExpectedSharedBuffer.length);
        return Arrays.copyOfRange(
            netcdfExpectedSharedBuffer, fileOffset * MASSBase.getMyPid(),
                (fileOffset * MASSBase.getMyPid()) + fileSize);
    } else {
        throw new RuntimeException(
            "Netcdf expected shared buffer is an invalid type - " +
            expectedSharedBuffer.getClass());
    }
}

@Override
public void assertExpectedEqualsActualBufferData(
    Object expectedSharedBufferData, Object actualSharedBufferData) throws Exception {
    if (expectedSharedBufferData instanceof float[] &&
        actualSharedBufferData instanceof float[]) {
```

```
        float[] netcdfExpectedSharedBufferData =
            (float[]) expectedSharedBufferData;
        float[] netcdfActualSharedBufferData =
            (float[]) actualSharedBufferData;
        try {
            Assert.assertArrayEquals(netcdfExpectedSharedBufferData,
            netcdfActualSharedBufferData, 0);
        } catch (AssertionError e) {
            throw new RuntimeException("ASSERTION FAILED");
        }
    } else {
        throw new RuntimeException(
        "Cannot assert buffer data: Netcdf buffer type is invalid.");
    }
}

@Override
public void writeFileIntoThisPlaceBuffer(Object[] writeArgs) throws Exception {
    if (writeArgs.length != 5) {
        throw new IllegalArgumentException(
            "Five arguments expected for Netcdf Write. Number of args: "
                + writeArgs.length);
    }
    int fileDescriptor = (int) writeArgs[0];
    String variableName = (String) writeArgs[1];
    int[] shape = (int[]) writeArgs[2];
    String filePath = (String) writeArgs[3];
    float[] writeDataBuffer = (float[])writeArgs[4];
    if(writeDataBuffer instanceof float[]) {
```

```
        openForWrite(filePath, variableName, shape);
        write(fileDescriptor, writeDataBuffer, variableName, shape); // Place Write
    }
}
```