Michael O'Keefe
CSS 499 – Professor Fukuda, MASS

Winter 2016 Report – Parallel I/O

Table of Contents

Preface

During Winter Quarter 2016, it was my task to implement parallel input and output within the MASS Library. The main reason for parallel I/O in the MASS Library is to increase the Netcdf file read performance within the University of Washington Climate Analysis (UWCA) application, while simultaneously enhancing the I/O capabilities of the MASS Library, which could be benefit many MASS applications. If interested, you may read more about our motivation, specification, and initial design for parallel I/O in the MASS Library in my term report for Autumn Quarter 2015 - http://depts.washington.edu/dslab/MASS/reports/MichaelOKeefe_au15.pdf. It is worth noting that the initial design for the project has changed a lot over the course of the quarter.

Before learning about the current implementation of the MASS Library's parallel I/O, it is important to know what Netcdf files are since they are the main reason for parallelization. Netcdf stands for Network Common Data Form, and they were created by Unidata, which is a component of the University for Atmospheric Research (UCAR). The Unidata website describes Netcdf as, "a set of software libraries and self-describing, machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data." In the simplest of terms a Netcdf file consists of three parts: dimensions, variables, and attributes. The dimensions of a Netcdf file have a name and a length. For example, UWCA has three-dimensions: time, longitude, and latitude. The variables of a Netcdf file have a name, a type, and a shape. For example, there is a variable in UWCA called "tasmax" that is of type float and has a three-dimensional shape over all three dimensions (that means there is a tasmax float record for every unique instance of time, longitude, and latitude). Imagine each variable as a mutli-dimensional array of data. Finally, the attributes for a Netcdf file hold metadata about files and data. For example, UWCA has a one-dimensional variable called longitude and one of its attributes is its "units", which is "degrees west". Knowing these basic aspects of a Netcdf file should be adequate for the rest of this report, but if you wish to learn more about Netcdf files, you may visit the following website - http://www.unidata.ucar.edu/software/netcdf/docs/.

Current Design

In order to implement parallel I/O within the MASS Library, each place must be able to open, read, write, and close the same file in parallel. To make these tasks possible, all of the current implementation has been done within the Place.java class (within the MASS Library). Currently, parallel I/O is being designed for Netcdf and Text files, but the code is open to the addition of more file types.

Place Fields Added for Parallel I/O

Picture 1: Place Fields Added for Parallel I/O, shows the fields that were added to Place.java for parallel I/O.

Picture 1: Place Fields Added for Parallel I/O

```
// stores each file and its attributes
protected static Hashtable< Integer, FileAttributes > fileTable = new Hashtable< >( );

// open options, 0 for READ, 1 for WRITE
private static final OpenOption[] OpenOperations = new OpenOption[ ] { READ, WRITE };

// place memory storage
private byte[] data;

// counts the number of files open
private static int count = 0;
```

The protected static field, *fileTable*, is a hash table that stores an integer file descriptor as a key and corresponding file attributes as a value. When any file is opened, it is given a unique integer as a file descriptor, its file attributes are set, and the newly opened file is added to the fileTable. Please note that the file name of the opened file is not the key for a file in the *fileTable* since that would not allow the same file to be opened more than once if the user wants to do so (this was a feature specified by Doctor Fukuda). I will go into more detail about what the file attributes are later on, but for now just know that they are a private class used for storing all attributes of a file that are needed for Parallel I/O. A hash table data structure was used for the *fileTable* because it allows for O(1) time retrievals when reading, writing, and closing files that have been opened. The *fileTable* is static because each file that has been opened will be read and written by all Place objects, thus each needs access to the *fileTable.*

The private static final field, *OpenOperations,* is an array of length 2 that specifies what operation is to be done to a file that has been opened. This field is specifically for text files, which use a FileChannel object to read or write the file to a buffer depending on the operation type. When a FileChannel is first opened, it must be specified what it is being opened for. Thus, we can specify a read operation by declaring *OpenOperations[0]* or a write operation by declaring *OpenOperations[1]* in the FileChannel's constructor.

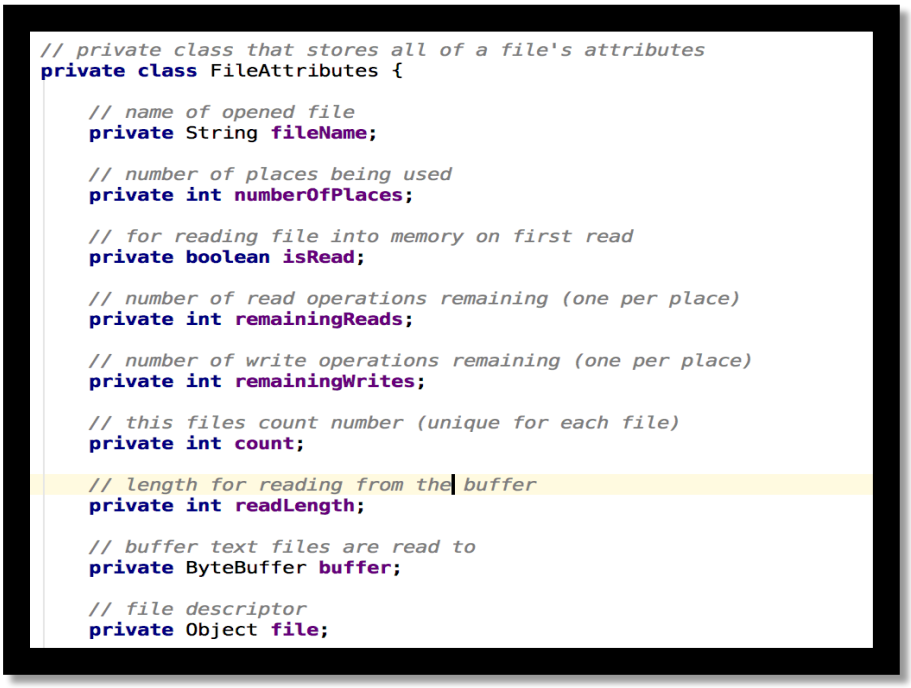
The private field, *data*, is a byte array that stores data that has been read an individual Place object (for a read operation), or that stores data that is to be written by an individual place object (for a write operation).

The private static field, *count,* is used to give each file in the *fileTable* their own unique file descriptor (a unique key). A file's descriptor will be equal to the number of files that have been added to the *fileTable* before itself. For example, the file descriptor of the first file added to the *fileTable* will have a file descriptor equal to 0, and the next file will have a file descriptor equal to 1. Each file descriptor will follow this pattern, even when a file is closed and removed from the *fileTable,* the *count* never decrements.

FileAttributes – Private Class within Place

FileAttributes is a private class within Place.java that is used for storing a file's attributes (a file's information that is needed while preforming parallel I/O). Picture 2: FileAttributes Fields, show the private fields that are stored in the FileAttributes class. The use for most fields are self explanatory from the commenting alone. The fields that may be confusing are *numberOfPlaces*, *remainingReads*, and *remainingWrites*. Each of these three fields store integer values and are initialized to the total number of place objects being used. After each Place performs a read or write operation, the *remainingReads* and *remainingWrites* are decremented accordingly.

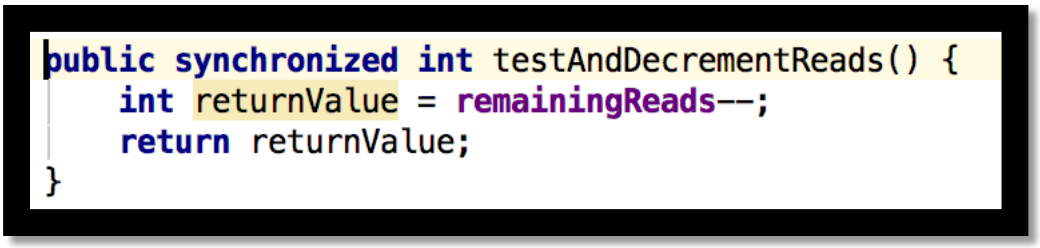
Picture 2: FileAttributes Fields



```
// private class that stores all of a file's attributes
private class FileAttributes {

    // name of opened file
    private String fileName;

    // number of places being used
    private int numberOfPlaces;

    // for reading file into memory on first read
    private boolean isRead;

    // number of read operations remaining (one per place)
    private int remainingReads;

    // number of write operations remaining (one per place)
    private int remainingWrites;

    // this files count number (unique for each file)
    private int count;

    // length for reading from the buffer
    private int readLength;

    // buffer text files are read to
    private ByteBuffer buffer;

    // file descriptor
    private Object file;
```

The methods within the FileAttributes class just consist of getters and setters for each field, as well as some test and set methods that are used to avoid race conditions while changing field values such as *remainingReads*. Race conditions must be taken into account when running any program using multiple threads. This is especially important to avoid while reading a file (since that is when multiple threads are used). Picture 3: Test and Decrement Reads, shows an example of how test and set is used.

Picture 3: Test and Decrement Reads



```
public synchronized int testAndDecrementReads() {
    int returnValue = remainingReads--;
    return returnValue;
}
```

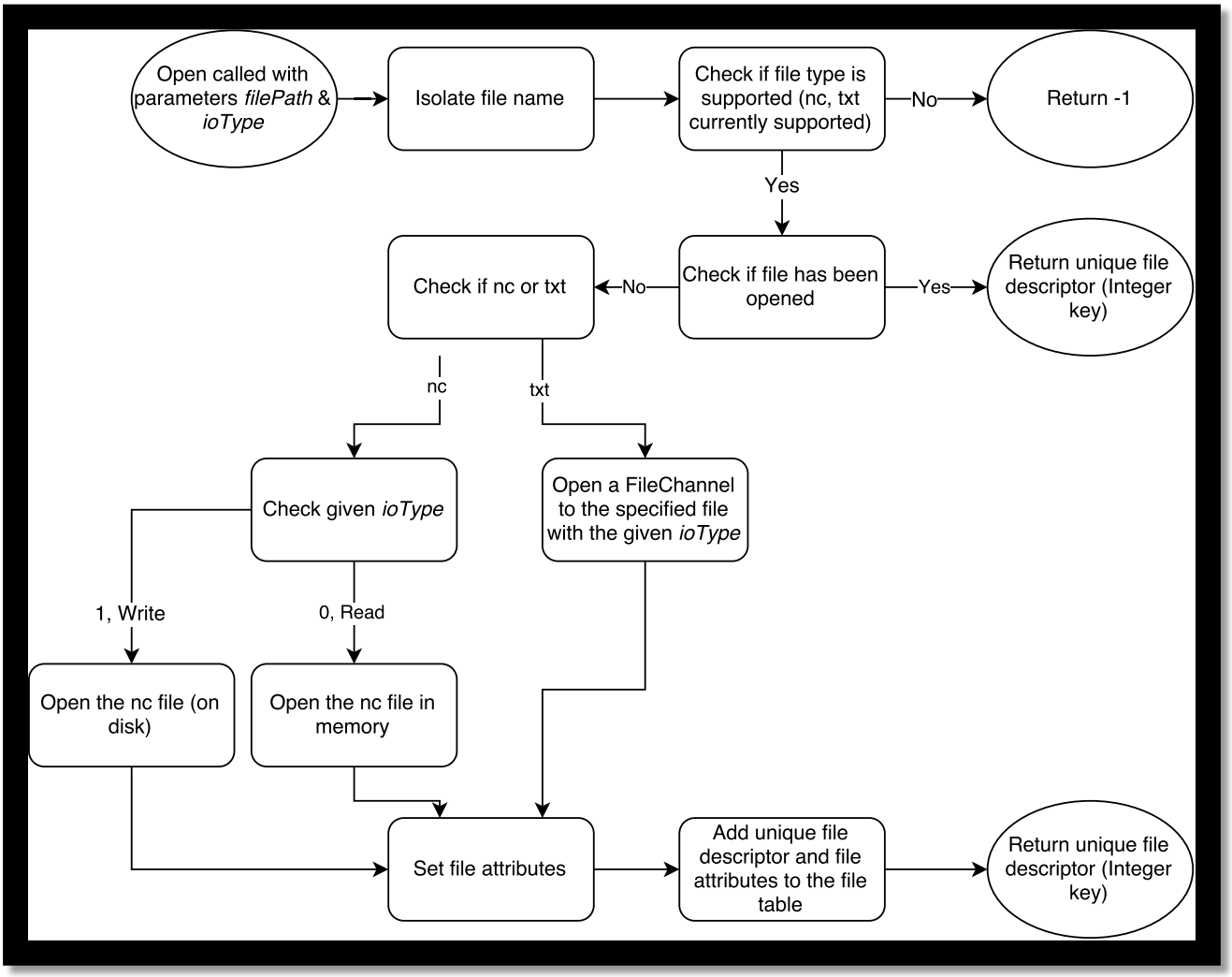
Open Method

Picture 4: Open Signature, shows the current signature for the *open* method. The method takes the given *filePath*, determines the file's name and whether the file's type is supported (currently only Netcdf and Text files are supported). If the given file is not supported, then -1 is returned, and the file is not opened. If the file type is supported and the file is not already in the *fileTable* then the file is opened accordingly based on the type of file, its file attributes are set, the file and its attributes are stored in the *fileTable* and the unique file descriptor is returned. If the file has already been opened, then nothing happens and the unique file descriptor is returned (all Place objects will try to open the file, but it only needs to be opened once). As mentioned previously, the same file can be opened and stored in the *fileTable* more than once, but only if the user calls the *open* method on the same *filePath* more than once. The method is synchronized to ensure that only one Place opens the the file and and puts the file into the *fileTable*. The parameter *filePath* should include the path to the file that is to be opened. The parameter *ioType* should either be a 0 for a read, and or a 1 for a write (determines how the file is opened). Look at Picture 5: Open Process, to see the process described visually. The *open* method is complete and works for both Netcdf and Text files.

Picture 4: Open Signature

```
protected synchronized int open( String filePath, int ioType ) throws IOException {
```

Picture 5: Open Process



Please note that I currently have the *read* and *write* method as a part of the *open* method, which does not make sense because *open* is synchronized and *read* shouldn't be. This mistake was due to miscommunication and will be fixed as soon as possible. The above design is correct and it is how the *open* method will be implemented.

The one thing I want to add to the *open* method is determining and storing the structure of a Netcdf file in its file attributes when it is first opened (I currently have been doing that in the *read* method, which is redundant to do for each Place).

Opening Text Files

If the Text file is being opened for a read operation then a new FileChannel for reading is connected to the Text file. If the Text file is being opened for a write operation then a new FileChannel for writing is connected to the Text file. The FileChannel is able to either read from or write to the Text file in later I/O operations.

Opening Netcdf Files

If the Netcdf file is being opened for a read operation then the file is opened in memory using the method NetcdfFile.openInMemory(<filePath>). If the Netcdf file is being opened for a write operation, then the Netcdf file is opened on the disk using the usual open method.

I performed a few performance tests to compare speed of reading a Netcdf file in memory vs in disk. Tests were performed on Netcdf file sizes varying from 500MB to 2GB. Reading from in memory is consistently 20% faster than reading from disk during the first read, and reading from in memory is consistently 40% faster than reading from disk during all previous reads. Picture 6: Netcdf Read In Memory VS Disk, shows one of the test results (time is measured in milliseconds). Please note that Netcdf files larger than 2GB cannot be opened in memory.

Picture 6: Netcdf Read in Memory VS Disk

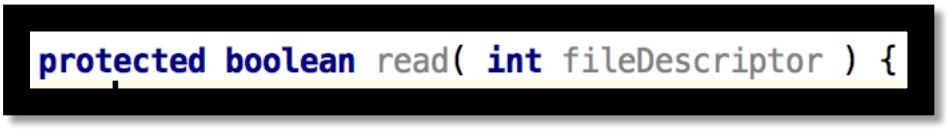
```
mgok@juno:/opt/mgok/NetcdfReadTests-Winter2016$ du -h testNetcdf.nc
2.0G    testNetcdf.nc
mgok@juno:/opt/mgok/NetcdfReadTests-Winter2016$ sh runReadFromDisk
4291
mgok@juno:/opt/mgok/NetcdfReadTests-Winter2016$ sh runReadFromMemory.txt
3445
mgok@juno:/opt/mgok/NetcdfReadTests-Winter2016$ sh runReadFromMemory.txt
2666
mgok@juno:/opt/mgok/NetcdfReadTests-Winter2016$ sh runReadFromMemory.txt
2573
mgok@juno:/opt/mgok/NetcdfReadTests-Winter2016$ sh runReadFromMemory.txt
2574
mgok@juno:/opt/mgok/NetcdfReadTests-Winter2016$ sh runReadFromDisk.txt
sh: 0: Can't open runReadFromDisk.txt
mgok@juno:/opt/mgok/NetcdfReadTests-Winter2016$ sh runReadFromDisk
4321
mgok@juno:/opt/mgok/NetcdfReadTests-Winter2016$ sh runReadFromMemory.txt
2605
```

Read Method

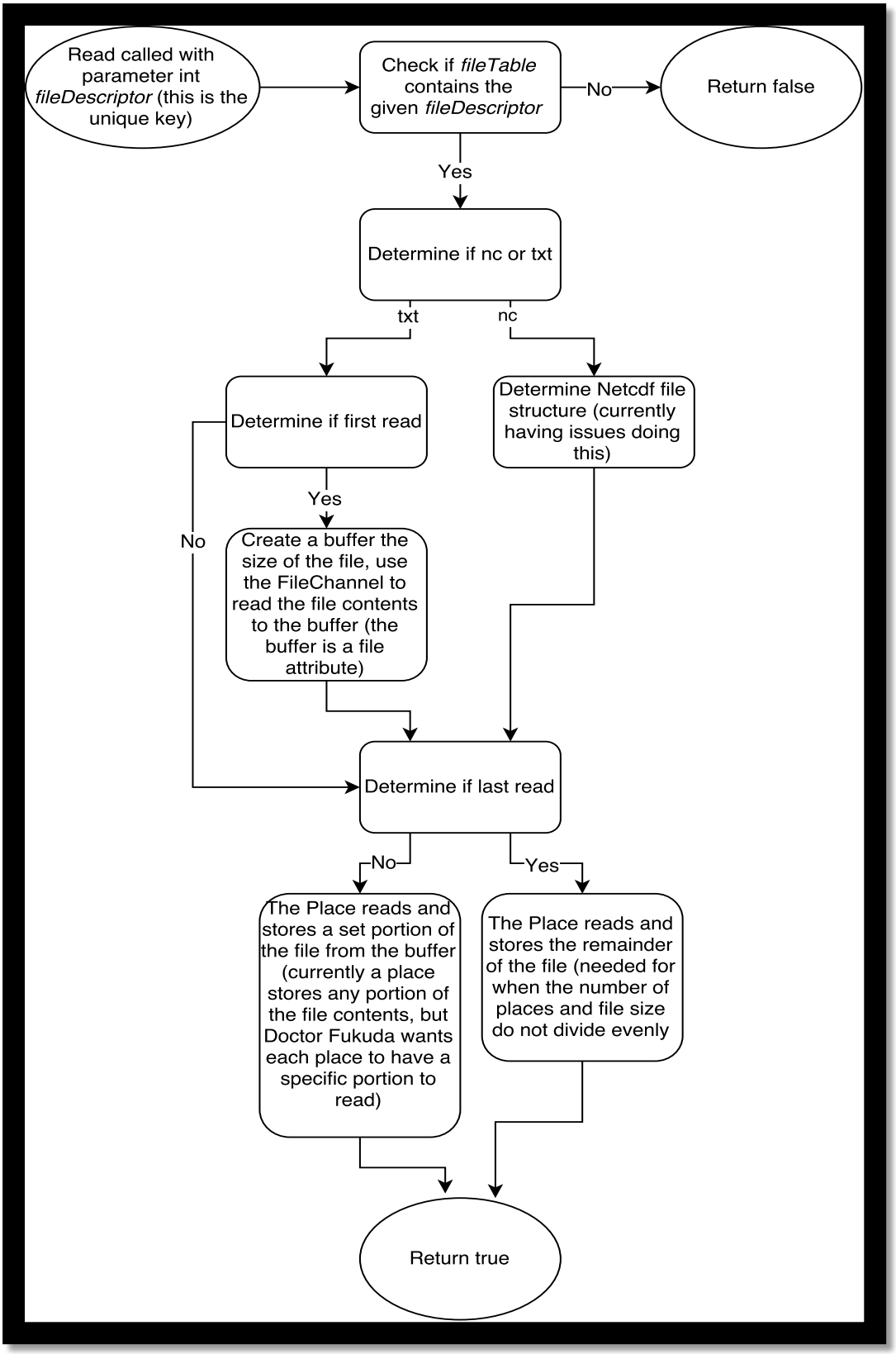
The signature of the *read* method can be found in Picture 6: Read Signature. The *read* method checks if the *fileTable* contains the given *fileDescriptor*. If it does, each Place reads and stores a predefined length of the file; otherwise, false is returned and nothing is read if the file does not

exist in the *fileTable.* The data read by a Place is stored in the Place's *data* field (recall the byte array *data*). The length that each Place reads is determined by the size of the file divided by the number of Places (thus each Place reads and stores the same amount of data). The last Place to read is responsible for reading the rest of the file, which could be longer than the specified length if the size of the file and the number of Places did not divide evenly (in case there is a remainder). Picture 7: Read Process, shows the process visually. Currently *read* works only for Text files and not Netcdf files. I am having issues using Netcdf functions that allow the user to determine the structure of any Netcdf file. For more information visit - http://www.unidata.ucar.edu/software/netcdf/docs/reading_unknown.html.

Picture 7: Read Signature



Picture 7: Read Process



Note that Doctor Fukuda has asked me to implement a way for each Place to have a specific portion of the file to read from. For example, if there 5 Places performing the *read* method, then Place1 would read the 1st portion of the file, Place2 would read the 2nd portion of the file, and so on. The current implementation allows which ever Place executes *read* first to read the 1st portion and the next Place to execute *read* reads the 2nd portion. Expect this to be a feature of *read* before the end of March 2016.

Reading Text Files

The first Place uses the FileChannel to read the entire Text file into a buffer (the buffer is apart of the Text file's file attributes so every Place has access to the buffer). Then each Place reads a set portion of the file from the buffer and stores it. The last place reads what ever has not been read from the buffer.

Reading Netcdf Files

Each Place reads a set portion of the file from the buffer and stores it. The last place reads what ever has not been read from the buffer.

Write Method

The basic design feature of the *write* method will be to have each Place write to the buffer, and then have the last Place write to the disk. Each Place will also have a specific portion of the file to write (just like the new feature for the *read* method).

The *write* method itself has not been implemented because I still have questions myself about how we want the user to be able to specify what he or she wants to write. I will be working with Doctor Fukuda as soon as possible in order to design the method so that I can begin implementation.

Close Method

Picture 8: Close Signature, shows the signature for the *close* method. The close method takes the given *fd* (file descriptor) and properly closes the corresponding file (depending on the type of file) and removes the file from the *fileTable*. If the *fd* was in the *fileTable* and the file was closed properly, then true is returned; otherwise, false.

Picture 8: Close Signature

Closing Text Files

Close the FileChannel associated to the Text file.

Closing Netcdf Files

Close the NetcdfFile associated to the Netcdf file.

Unit Testing

PlaceTest.java has been created in order to test the methods that have been implemented for parallel I/O within Place.java. PlaceTest.java can be found in the test folder of mass_core. The unit test methods created in PlaceTest.java test each parallel I/O method in isolation. That means that no other components of the MASS Library are involved in the testing process. The unit tests are also run when the MASS Library is built, which gives code assurance when new changes are made. Picture 9: Text File Read Unit Test, is an example of a unit test that has been created and tests the opening, reading, and closing of a Text file using multiple Places.

Picture 9: Open and Read Unit Test

```
@TestSubject
private Place place1 = new Place(); // create a place object to test with

@TestSubject
private Place place2 = new Place();


@Test
public void testOpenReadCloseWithTxt() throws Exception {
    String filePath = "/Users/Michael/mass_java_core/testTxt.txt";
    String filePath2 = "/Users/Michael/mass_java_core/testTxt2.txt";

    int descriptor;
    int descriptor2;
    int descriptor3;

    descriptor = place1.open(filePath, 0);

    descriptor = place2.open(filePath, 0);

    descriptor2 = place1.open(filePath2, 0);

    descriptor2 = place2.open(filePath2, 0);

    descriptor3 = place1.open(filePath, 0);

    descriptor3 = place2.open(filePath, 0);

    place1.close(descriptor);
    place2.close(descriptor);
    place1.close(descriptor);
}
```

Next Steps

UWCA needs to have its Netcdf file read performance increased before the end of April for its paper resubmission, so my goal is to finish these next steps before then. Luckily, I am taking Spring Quarter off so that I can take an internship over summer and graduate after Fall Quarter 2016. That means I can devote a lot of time to ensure that these next steps are finalized before the end of April.

Open Method

For Netcdf files, determine the file structure when the Netcdf file is first opened and store the structure within the file's attributes.

Read Method

Implement a way for each Place to have a specific portion of the file to read from. For example, if there are 5 Places performing the *read* method, then Place1 would read the 1st portion of the file, Place2 would read the 2nd portion of the file, and so on.

Write Method

Work with Doctor Fukuda and define the method's signature and design. Specifically determine how the user should specify what he or she wants to write.

File Partitioning and Distribution Tool

Work with Doctor Fukuda to design and implement a tool that will split a file depending on how many computing nodes that are being used and distributes each file piece to the designated node in the local /tmp directory.

File Collecting and Merging Tool

Work with Doctor Fukuda to design and implement a tool that will collect the file pieces from each computing node after the I/O operations have been preformed, and merges pieces back to one file.

Test on UWCA

Work with Jason Woodring to include MASS Parallel I/O functionality within UWCA, and test the Netcdf file reading performance.