Michael O'Keefe
Term Paper - Fall Quarter 2015

MASS Parallel Input/Output

Motivation

The main issue with the University of Washington Climate Analysis (UWCA) application is in the amount of time it takes to read the Netcdf files where the climate data is stored. During Summer Quarter 2015, I conducted various parallel file read tests in order to test the parallel reading speed of text and Netcdf files in hopes of revealing why UWCA's file reading performance is so low, or at least narrowing down what the issue is. The results from the parallel file reading tests were as expected: both text files and Netcdf files are read faster when using multiple computing nodes in parallel, as opposed to reading the files sequentially. That means that UWCA's read performance issue is in UWCA's implementation and not due to the Netcdf files themselves.

During Autumn Quarter 2015, I spent most of my time looking through and trying to understand UWCA's code, while looking for the performance issue (thinking that it would be found in the usage of the Netcdf files). At first I thought I caught some efficiency errors in how the files were being read, but performance tests showed that fixing the efficiency errors were minor, since they did not increase the read speed. Progress was not being made quickly, partly since UWCA was difficult to understand because of lack of documentation (a good lesson because now I fully understand the importance of detailed comments), and the initial developer working on UWCA did not have a lot of time to work with me this quarter to explain what certain parts of the code are supposed to do. This is important since I am able to figure out what sections of the code do by going through it, but it is difficult to tell if the code is doing what it is supposed to do without documentation or an explanation. Thus, without proper help, I was at a loss with how to make progress on UWCA's read performance.

It later came to our attention, mainly Doctor Fukuda's, that UWCA uses the Multi-Agent Parallel Simulation (MASS) library to analyze the Netcdf formatted climate data, but it cannot use the MASS library for reading the Netcdf files because MASS does not have any parallel Input/Output. Without parallel I/O, users must read and/or write files via the master node, (i.e., the main program), which will become a bottleneck between the disk and all slave nodes participating in the same computation. That means that reading Netcdf files using the MASS library can only be done on one node (the master node), thus there will always be single node reading performance even when using multiple nodes for the rest of the computation. This issue creates an urgency for parallel I/O to be implemented in the MASS library. This would solve UWCA's performance issue, and simultaneously benefit other MASS applications that could use parallel I/O capabilities and increase application performance.

Specification

The goal of the parallel I/O project is to allow each place to open, read, write, and close the same file in parallel. All functions are to be implemented inside the Plass.java class.

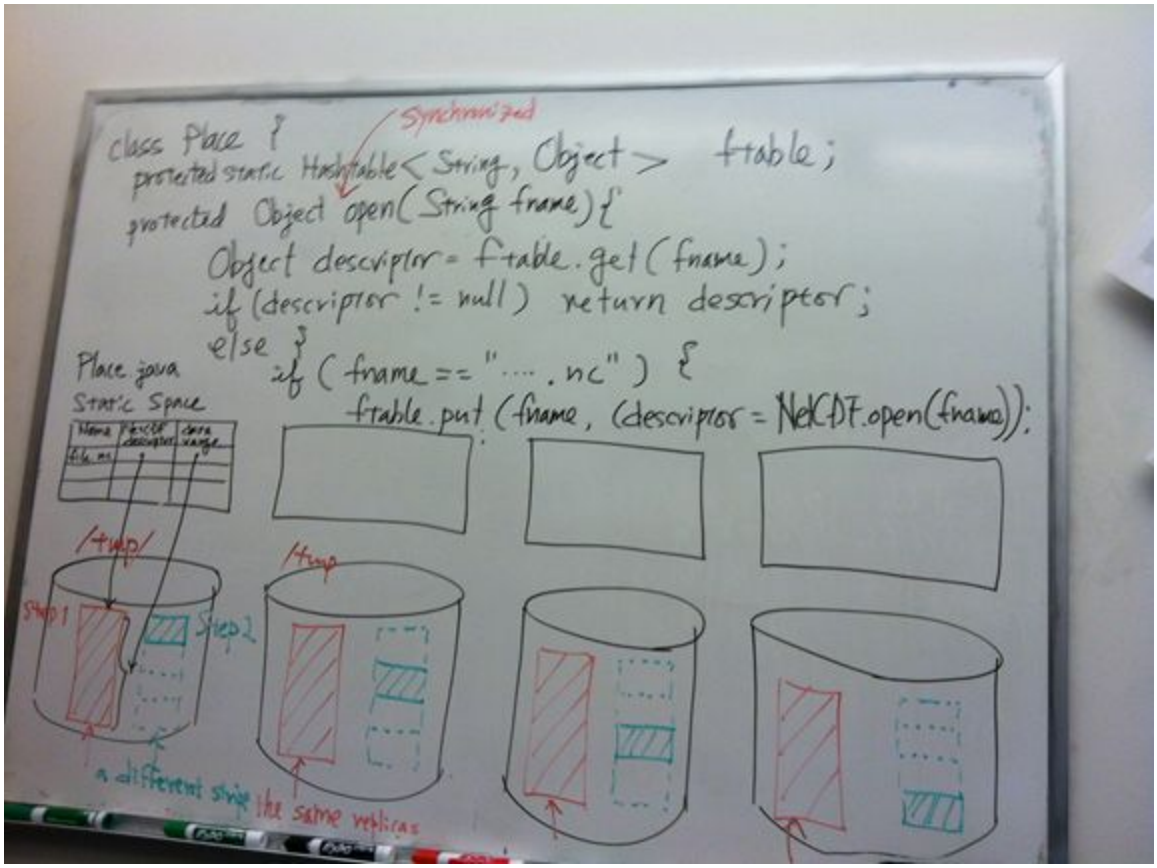
1. Open: the open function will use the given string parameter that contains the name of the file to be opened to open the file, and returns the opened file object. The function must be able to open the specified file depending on what the file type is. For example, if the file type is Netcdf, then the open function must open the Netcdf file using Netcdf's open syntax. File types that will be implemented are Netcdf and text. The code must be implemented so that it is easy to add additional files that can be opened. The open function must be synchronized and only be used by the starting place in each computing node.
2. Read: Use the FileChannel class to read the entire file into memory upon the very first read( ) so that the following reads won't go to the local disk.
3. Write: Write data into memory but not into the disk quickly, and postpone all writes to disk until the very last write( ) is issued from a place.
4. Close: the close function will use the given object parameter that contains the file descriptor (the type of file) to close the file, and returns a boolean (true for successful close, and false otherwise). The function must be able to close the file object depending on what type of file it is.The code must be implemented so that it is easy to add additional files that can be closed. The close function must be synchronized and only be used by the last place in each computing node.

Note: The Read and Write specification will be further developed next quarter once Doctor Fukuda and I are able to meet and discuss.

At first I will assume that each computing node has a replica of the same file at the /tmp directory. I will work under this assumption until the four functions above are working properly. Next, I will assume that each computing node has a different portion of the same file at the /tmp directory.
For this part of the project, I will need to develop a file partitioning and distribution tool. The file and distribution should work for all of the file types implemented in the four functions listed above. The tool should be able to evenly partition a given file based on the number of computing nodes that will be used in the computation. I will also need to develop a collecting and merging tool that will put the file that was used for computation back together. This tool should know where each portion of the file is for collection and the order of each file portion for merging.

Design

1. Open:



2. Read:

(Pseudo code until discussed with Doctor Fukuda)

Return: a boolean true for success

Parameter(s): the object to read.

Function design:

Execute on very first Read() following reads won't go to disk

Check size of file for a more efficient read:

If large (need to determine what large is):

Use FileChannel's map() function, region of a file may be mapped directly into memory; for large files this is often much more efficient than invoking the usual read or write methods

Else:

Read into memory using FileChannel's read function

3. Write:

(Pseudo code until discussed with Doctor Fukuda)

Return: a boolean true for success

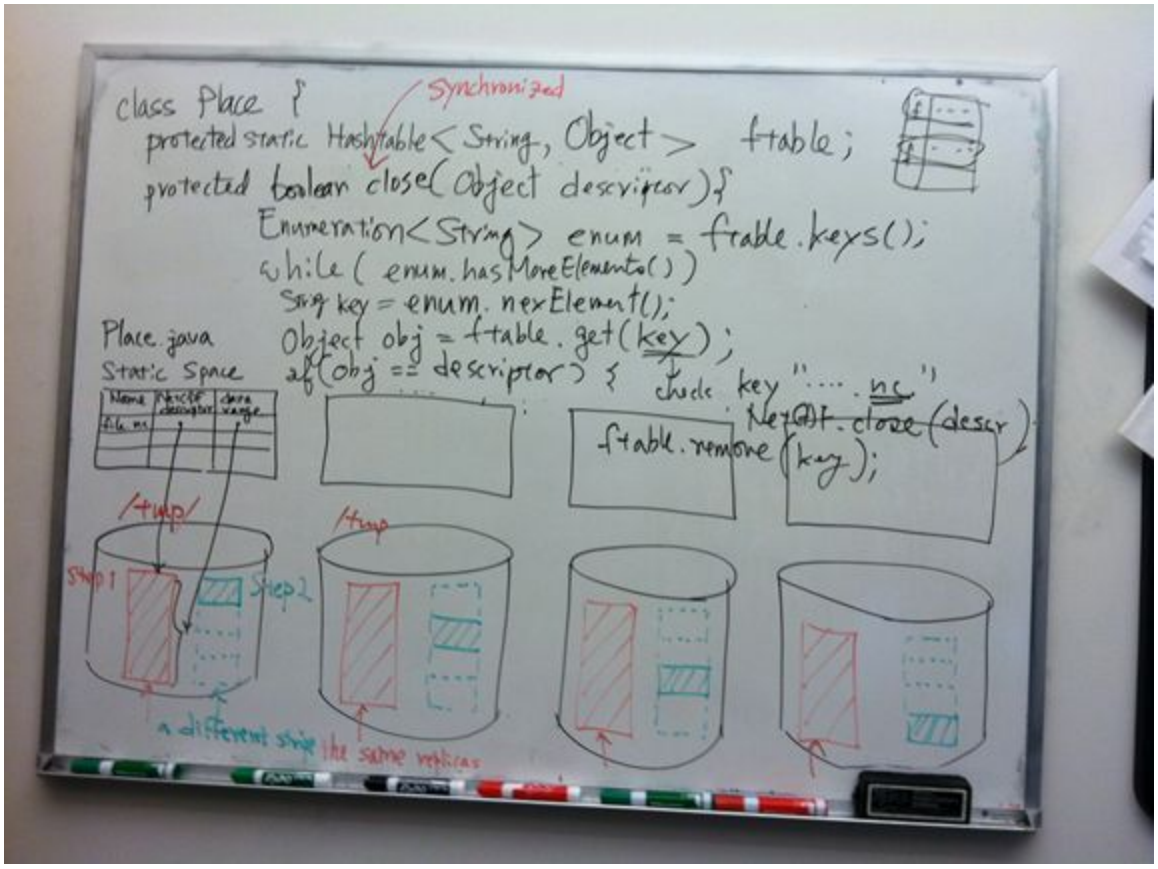
Parameter(s): the object to write

Function design:

Use FileChannel write() function to write the file into memory

Postpone writing to the disk until the final write issued from a place

4: Close:



Technical challenges:

- Learning various aspects distributed programing - how to make sure each place does what I want it to.
- Using Java Synchronized to stop threads during function execution.
- Working with FileChannel and understanding all functionality available.
- Creating a file collector and distributor.
- Working with the Linux OS.

Schedule

Based on our friday meetings:

Week (Date)	Finish with:
1 (Jan. 8 - 15)	Open implementation and unit test
2 (Jan. 15 - 22)	Close implementation and unit test
3 (Jan. 22 - 29)	Read implementation and unit test
4 (Jan. 29 - Feb. 5)	Write implementation and unit test
5 (Feb. 5 - 12)	Execution performance testing
6 (Feb. 12 - 19)	File partitioning and distribution tool
7 (Feb. 26 - Mar. 4)	Execution performance testing & verification
8 (Mar. 4 - 11)	File collecting and merging tool
9 (Mar. 11 - 18)	- Open week for any additions or setbacks -
10 (Mar. 18 - Finals)	Term report / guide