Michael O’Keefe

CSS 499 – Professor Fukuda

**Summer 2015 Report**

My task this past summer was to continue on with Jason Woodring’s research, UWCA, a climate change analysis application that utilizes Multi-Agent Spatial Simulation. The main performance issue with UWCA is currently the Netcdf file reading process. The slow performance could be caused by a wide range of issues from the UW1-320 machines themselves, to Netcdf software, or to poor programing. In order to narrow down and determine what is causing the performance problem, I was asked to create programs that test the reading speed of text files and Netcdf files, using individual and multiple computing nodes.

**What I Did / Results – Text File Testing**

The first task I was asked to perform was to test the speed of reading a large text file (15GB) on one computing node, and compare it to the speed of reading a 1GB text file on 15 computing nodes in parallel. It is expected that using 15 computing nodes should make the file reading process about 15 times as fast, but UWCA seemed to be read it’s files (which are complex Netcdf files) faster on one computing node. This test was mainly to check if there was an issue with performing parallel file reads on the UW1-320 machines.

In order to test both single and multiple node file reads, I needed to create both 1GB and 15GB text files. I used the command: “base64 /dev/urandom | head –c <size> > <filename>.txt” to create the text files. Note that these text files are filled with random ascii values.

This is a linux terminal command so I created the text files while in UW1-320-00 /opt/jwoodring/. The file names are “fileRead.txt” (1GB) and “fileRead15GB.txt”. I then created a bash script called “fileTransfer” to store “fileRead.txt” on all 15 working UW1-320 machines in their own /opt/jwoodring/ local directory (UW1-320-12 was down at the time) since the 1GB text file needed to be read by each machine.

The program that I created to perform the file read is called “Read.java.” The reading process is very simple. A scanner is created over the text file to be read, and that scanner will read and store each line of the text file. In order to test the reading process on individual and on multiple nodes, the program takes two arguments when run. The first argument can be either the String "one" or "two". When the first argument is "one", that specifies a single node file read, thus the program will be reading the file fileRead15GB.txt. If the first argument is "two", that specifies a 15 node file read, thus reading the file fileRead.txt (which is 1GB). The second argument specifies how the file being read is stored. The purpose of the second argument was to see how storage will affect the reading time (it has little to do with comparing the performance of single and multiple node file reads). The second argument can be either the String “single”, "eighth", "quarter", "half", or "full". If the second argument is “single”, then one line, of the text file being read, will be stored at one time. If the second argument is “eighth”, then one-eighth of the lines, in the text file being read, will be stored at one time. If the second argument is “quarter”, then one-quarter of the lines, in the text file being read, will be stored at one time.

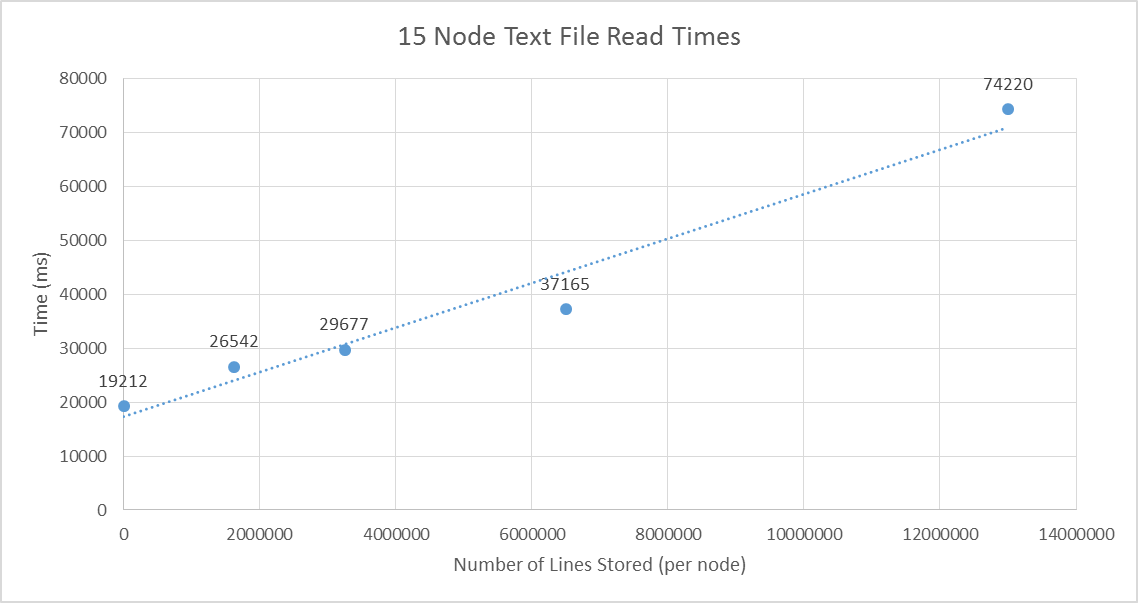
If the second argument is “half”, then one-half of the lines, in the text file being read, will be stored at one time. If the second argument is “full”, then all lines, in the text file being read, will be stored. Do not confuse this with only reading part of the text file. For example, the argument, “eighth,” does not mean that only one-eighth of the text file is read. It actually means that one-eighth of the text file is read and stored, then the next eighth is read and is stored where the previous-eighth was stored (overwriting the previously stored lines). To figure out how many lines are in a file you can use the Linux command: “wc –l <filename>.txt”.

The compiled version of Read.java, Read.class, also needed to be stored on each working UW1-320, so I created a bash script called, “updateRead”, that stores Read.class in each working UW1-320’s /opt/jwoodring/ directory. Note: updateRead will store the most recently compiled version of Read.java stored in UW1-320-00’s /opt/jwoodring/ directory to the other machines. In order to compile Read.java use the command: “javac Read.java” while in UW1-320-00 directory /opt/jwoodring/.

To make it easier to perform each type of parallel read test, I created bash scripts for each time of storage option. Each is stored in UW1-320-00’s /opt/jwoodring/ directory. The names of each bash script correlates to how they store the text file: “readTwoSingle” stores one line of the text file at a time, “readTwoEighth” stores one-eighth of the lines in the text file at a time, “readTwoQuarter” stores one-quarter of the lines in the text file at a time, “readTwoHalf” stores one-half of the lines in the text file at a time, and “readTwoFull” stores every line in the text file. Each of these bash scripts, their contents and how to run them are described in more detail in the bellow in the section: “How to Preform Parallel Read Tests with Bash Scripts”.

I was ready to begin my testing once all files were in their correct directories on each working UW1-320 machine. I performed each test a total of 10 times and computed to average in order to get more accurate results. The recorded results for 15 node read times are shown in Graph 1. Note that times may fluctuate throughout the day due to the number of users on the UW1-320 machines.

Graph 1: 15 Node Text File Read Times



The results show a linear trend when it comes to the storage method used. This is expected and due to the fact that memory allocation takes time and storing 15GB is a large overhead. Only one storage test is important to get for the 15GB single node read. The time it took to read all 15GB and store only one line at a time (arguments “one” and “single”) took an average of 324,243 (ms). That is equivalent to 5.40405 minutes. The other storage tests were not possible on one computing node considering that even storing one-eighth of the lines required storing 2.5 million strings (8 times). The 324,243 average read time is all we need to compare the parallel file read to the single node read. The parallel file read should be about 21,616.2 (ms) in order to be 15 times faster than the single node read (324,243 / 15 = 21,616.2). The parallel read time was 19,212 for the single line storage test, thus parallel text file reads are working as expected on the UW1-320 machines. The next step in the testing process is to test reading Netcdf files in parallel.

**What I Did / Results – Netcdf File Testing**

UWCA reads from 5 Netcdf files, a total of 22GB in size. Each of the files vary in size, but they are roughly 4GB each (note that one file is significantly larger, at 8GB).I have created two Java programs in order to narrow down the possibilities of what is causing the performance issue.

Before creating either of the programs I had to understand what Netcdf files are and how they work. Netcdf stands for Network Common Data Form and was created by Unidata, which is a part of the University Corporation 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. 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/>.

The first program, “WriteNetcdf.java”, creates a simple Netcdf file that has traits similar to the Netcdf files used in UWCA, although, the test file is much less complex. The test Netcdf file created is called, “testNetcdf.nc”, it is 4GB in size, it has three-dimensions (x, y, and z), each dimension has a size of 1000, and it has one three-dimensional variable called “data” that contains random float values. Meaning, testNetcdf.nc contains 1,000,000,000 random floats values (floats are 32-bits thus the file is 4GB in size), with one float per index in the three-dimensional variable “data”.

The second program, “NRead.java”, reads the simple Netcdf file, “testNetcdf.nc”, and stores the float data based on the arguments passed to the program. The first argument passed to the program must be the String “two” – this does not add any functionality to the program as of yet, just know that it is needed to start the reading process (I plan on adding more functionality to the first argument later on, so bear with me). The second argument passed to the program can be any of the following Strings: “single” for storing only one float at a time, “eighth” for storing one-eighth of the Netcdf’s float values at a time, “quarter” for storing one-quarter of the Netcdf’s float values at a time, “half” for storing one-half of the Netcdf’s float values at a time, and “full” for storing all of the Netcdf’s float values at one time. Note that passing an argument, such as “eighth,” does not mean that only one-eighth of the testNetcdf.nc file is read. It actually means that one-eighth of the file is read and stored, then the next eighth is read and is stored where the previous-eighth was stored (overwriting it). Imagine it like a circular array - once all the space is filled up in the array, start adding (and replacing) values at the first index again.

Once the two programs were finished, I needed to test whether or not a parallel read of a Netcdf file had better performance than a read using a single machine. To test the parallel read performance, I needed to distribute all files necessary to run the test program to each working UW1-320 machine (UW1-320-12 was not working at the time). First, I added both NRead.java and WriteNetcdf.java to UW1-320-00 using WinSCP (a file transfer program). The jar file, “netcdfAll-4.6.jar,” must be included in both programs classpath in order to compile them (an example of how to compile the programs can be found below). Second, I created the testNetcdf.nc file by compiling WriteNetcdf.java and running WriteNetcdf.class on UW1-320-00. Then I created a bash script called “updateNetcdfFile” that stored testNetcdf.nc on each working UW1-320 machine using the command “scp”. Thirdy, I added NRead.java to UW1-320-00 and compiled it to create NRead.class. Then I created a bash script that added NRead.class to each working UW1-320 machine using the same “scp” command. The last two requirements for running NRead.class on each machine were two jar files: “netcdfAll-4.6.jar” (the same one needed to compile), and “slfj-nop-1.7.12.jar” (needed to avoid a logger error). The two bash files used to add these to each working UW1-320 machine are “addNetcdfJar” and “addSLFjar”.

Now that all working UW1-320 machines can properly run NRead.class, I created bash scripts for all the different ways to store the float data (all of the bash scripts can be found below). By using linux’s “time” command I was able to run and time each type of read on all 15 machines. For example, using the command: “time sh nreadTwoFull”, NRead.class would run on all 15 Nodes and once the process finished, Terminal displays the real time, user time, and system time. Real time shows the amount of time it took to complete the run on all 15 nodes, and the user time shows how long it took to complete the run on the machine you are logged into (UW1-320-00, where all the bash scripts are stored). The first tests I performed were to check if the parallel read had loss in performance from the individual machine read. To do that I ran NRead Two Full on UW1-320-00 and recorded the real time. Then I ran NRead Two Full on all 15 machines and recorded the user time. The user time of the parallel run and the real time of the individual run were nearly the same every time meaning that there is no performance loss when performing parallel runs. Next I needed to test all types of storage options on all 15 machines.

To test the parallel runs, I ran each run using their bash script. I recorded 10 run times for each storage method and used the average time. The results can be found in Graph 2. Note that run times may fluctuate during the day due to the number of users on the UW1-320 machines. I found that either late at night or early in the morning were the best times to test.

Graph 2: 60GB Netcdf Read (15 Nodes, 4GB Each)

The results show that using a one-eighth or one-half storage method may be slightly faster than the one-half or full, although those storage methods may not be possible for UWCA (I am not sure). The results also show that there is nearly no slowdown when using full rather than one-half. Actually, storing all 4GB tended to be faster than storing only 2GB at a time. The most important piece of information is that reading and storing all 60GB (15 nodes x 4 GB per node) took less than 9 seconds for each case using a sequential read. The next step in the testing process is to make the Netcdf file, created by WriteNetcdf.java, more complex and similar to the Netcdf file used in UWCA. By doing that, it will be possible to see where any new slowdowns are (if there are any). I look forward to further testing.

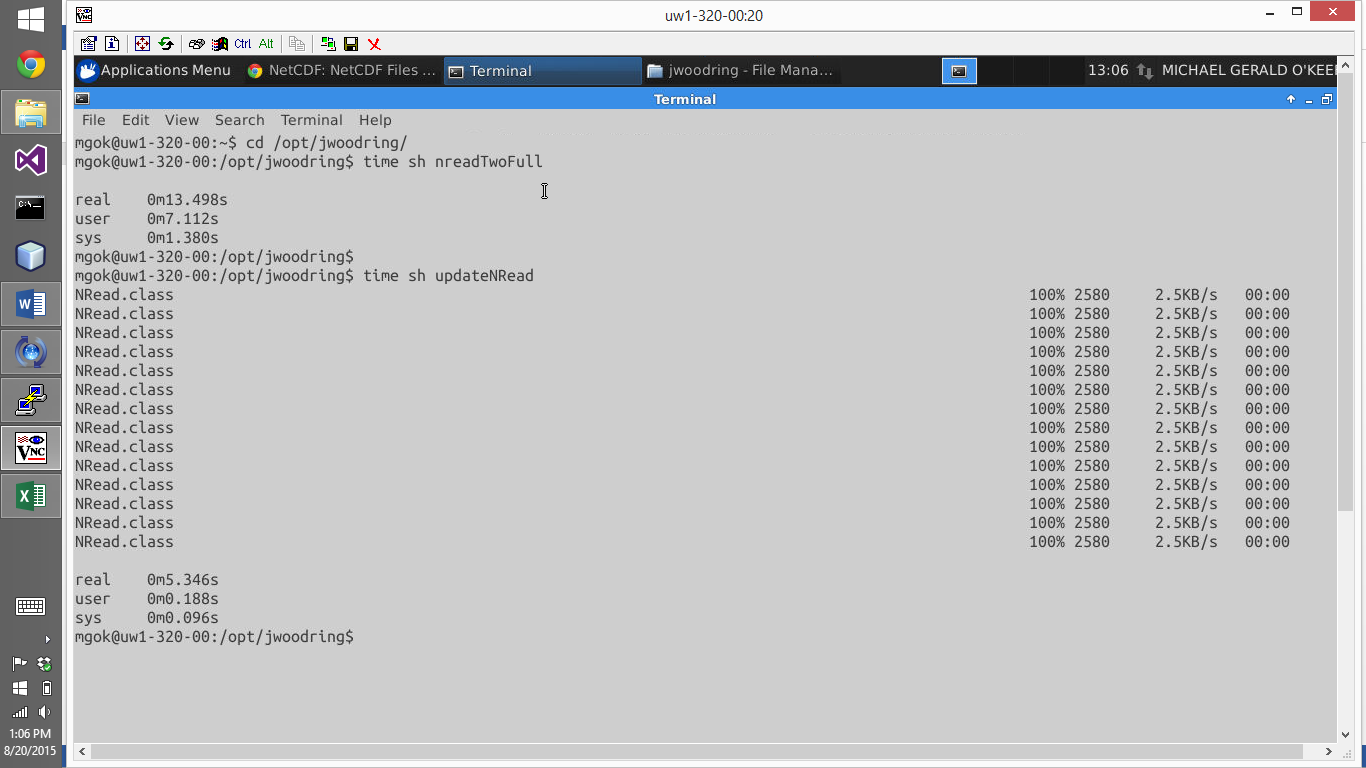
**How to Perform Parallel Read Tests with Bash Scripts**

**(Both Text and Netcdf)**

* All programs, jar files, bash scripts, and netcdf files needed are stored in UW1-320-00 in the directory /opt/jwoodring/
  + In order to run and change the files in any way you will need write access to this directory; otherwise, only read access is available
  + /opt/jwoodring is a local directory, meaning that every computer in UW1-320 has different contents in /opt/jwoodring/
    - Example: adding testNetcdf.nc to UW1-320-00 in /opt/jwoodring/, does not add it to any of the other UW1-320 machines
  + If any of the files are missing from UW1-320-00 /opt/jwoodring/ - the files should be uploaded to BitBucket
* UW1-320-12 is the only computer that is not set up to run the tests since it was down while performing my own tests (there are 16 computers in UW1-320, they go from UW1-320-00 to UW1-320-15). Thus the bash scripts only ssh into to 15 computers
* Use the linux command: time, before running the bash scripts in order to time each run
  + Example: to time reading testNetcdf with full storage on all 15 nodes use the command: time sh nreadTwoFull
* Bash scripts are required to perform all of the parallel file reading tests, but there are also bash scripts written for individual node reads (for convenience only). A bash script is a text file with the line, “#!/bin/sh” at the top of the file. To run any of the following bash scripts from terminal, log into UW1-320-00, change directory to /opt/jwoodring/ (command: cd /opt/jwoodring/) then use the command: sh <bashScriptName>. All requirements should already be completed but they are listed in case there is ever an issue. All bash scripts:
  + readOneSingle
    - runs Read.class, using single line storage, on UW1-320-00
      * requirements: UW1-320-00 must have Read.class, and fileRead15GB.txt in the directory /opt/jwooding
  + readOneEighth
    - runs Read.class, storing one-eighth of the lines at a time, on UW1-320-00
      * requirements: UW1-320-00 must have Read.class, and fileRead15GB.txt in the directory /opt/jwooding
  + readOneQuarter
    - runs Read.class, storing one-quarter of the lines at a time, on UW1-320-00
      * requirements: UW1-320-00 must have Read.class, and fileRead15GB.txt in the directory /opt/jwooding
  + readOneHalf
    - runs Read.class, storing one-half of the lines at a time, on UW1-320-00
      * requirements: UW1-320-00 must have Read.class, and fileRead15GB.txt in the directory /opt/jwooding
  + readOneFull
    - runs Read.class, storing all lines, on UW1-320-00
      * requirements: UW1-320-00 must have Read.class, and fileRead15GB.txt in the directory /opt/jwooding
  + readTwoSingle
    - runs Read.class, using single line storage, on each of the 15 working UW1-320 machines
      * requirements: all 15 nodes must have Read.class, and fileRead.txt (1GB) in their local directory /opt/jwooding
  + readTwoEighth
    - runs Read.class, storing one-eighth of the lines at a time, on each of the 15 working UW1-320 machines
      * requirements: all 15 nodes must have Read.class, and fileRead.txt (1GB) in their local directory /opt/jwooding

readTwoQuarter

* + - runs Read.class, storing one-quarter of the lines at a time, on each of the 15 working UW1-320 machines
      * requirements: all 15 nodes must have Read.class, and fileRead.txt (1GB) in their local directory /opt/jwooding
  + readTwoHalf
    - runs Read.class, storing one-half of the lines at a time, on each of the 15 working UW1-320 machines
      * requirements: all 15 nodes must have Read.class, and fileRead.txt (1GB) in their local directory /opt/jwooding
  + readTwoFull
    - runs Read.class, storing all lines, on each of the 15 working UW1-320 machines
      * requirements: all 15 nodes must have Read.class, and fileRead.txt (1GB) in their local directory /opt/jwooding
  + nReadTwoSingle
    - runs NRead.class, using single variable storage, on each of the 15 UW1-320 computing nodes
      * requirements: all 15 nodes must have NRead.class, netcdfAll-4.6.jar, slfj-nop-1.7.12.jar, and testNetcdf.nc stored in their local directory /opt/jwoodring/
  + nReadTwoEighth
    - runs NRead.class, using one-eighth variable storage, on each of the 15 UW1-320 computing nodes
      * requirements: all 15 nodes must have NRead.class, netcdfAll-4.6.jar, slfj-nop-1.7.12.jar, and testNetcdf.nc stored in their local directory /opt/jwoodring/
  + nReadTwoQuarter
    - runs NRead.class, using one-quarter variable storage, on each of the 15 UW1-320 computing nodes
      * requirements: all 15 nodes must have NRead.class, netcdfAll-4.6.jar, slfj-nop-1.7.12.jar, and testNetcdf.nc stored in their local directory /opt/jwoodring/
  + nReadTwoHalf
    - runs NRead.class, using one-half variable storage, on each of the 15 UW1-320 computing nodes
      * requirements: all 15 nodes must have NRead.class, netcdfAll-4.6.jar, slfj-nop-1.7.12.jar, and testNetcdf.nc stored in their local directory /opt/jwoodring/
  + nReadTwoFull
    - runs NRead.class, using full variable storage, on each of the 15 UW1-320 computing nodes
      * requirements: all 15 nodes must have NRead.class, netcdfAll-4.6.jar, slfj-nop-1.7.12.jar, and testNetcdf.nc stored in their local directory /opt/jwoodring/
  + addNetcdfJar
    - adds netcdfAll-4.6.jar to the directory /opt/jwoodring/ in UW1-320-01 through UW1-320-15 (except UW1-320-12)
      * requirements: UW1-320-00 /opt/jwoodring/ contains netcdfAll-4.6.jar
  + addSLFjar
    - adds slfj-nop-1.7.12.jar to the directory /opt/jwoodring/ in UW1-320-01 through UW1-320-15 (except UW1-320-12)
    - requirements: UW1-320-00 /opt/jwoodring/ contains slfj-nop-1.7.12.jar
  + updateNetcdfFile
    - adds testNetcdf.nc to the directory /opt/jwoodring/ in UW1-320-01 through UW1-320-15 (except UW1-320-12)
    - requirements: UW1-320-00 /opt/jwoodring/ contains testNetcdf.nc
      * If you want to change testNetcdf.nc – update it in UW1-320-00 and then call this bash script
  + updateNRead
    - adds NRead.class to the directory /opt/jwoodring/ in UW1-320-01 through UW1-320-15 (except UW1-320-12)
    - requirements: UW1-320-00 /opt/jwoodring/ contains NRead.class
      * If you want to change NRead.class – update NRead.java on UW1-320-00, compile NRead.java (command: javac NRead.java) and then call this bash script
* Visual example of how to run a bash script:



* + Log into UW1-320-00
  + Go to directory /opt/jwoodring
  + Use time and sh commands on the bash scripts provided

**Logic behind Commands Used**

* Compile java programs through Linux Terminal – log into the machine and directory where the java file is stored:
  + Use the command: javac <programName.java>. This will create a .class file of the program in the same directory
* Run java programs through Linux Terminal – log into the machine and directory where the .class file of your program is stored:
  + Use the command: java <programName> to run the program
* Run java programs on other UW1-320 computing nodes from UW1-320-00:
  + Use the command: ssh uw1-320-<computerNumber> ‘cd /opt/jwoodring/; java <programRunSpecifications>’ &
    - “&” is used to only if you want to run this process in the background
    - “;” is used to separate commands (commands to the left of “;” execute before commands to the right
    - <programRunSpecifications> can be found below for each program in the section “Program Compile and Run Specifications”
* Recording run time:
  + Place the command “time” before any command and real, user, and system times will be displayed once the full command has finished executing
    - Real time: the time it took for the entire process to finish
    - User time: amount of time the computing node you are on took in its own process (useful for parallel reads so you can tell the time one node took compared to all 15)
* Executing commands in parallel:
  + A “&” is used at the end of a command if you want to run it as a background process
    - Meaning that other commands can be executed while the background command is executing (jobs are being executed in parallel)
  + The command “wait” finishes executing once all background processes are finished
    - That is why “wait” is found at the end of each bash script and the time is recorded once all processes are finished
* Run Arguments – an argument is a String that is written after the program name. For example: To run a java program named “foo” with three arguments “x” and “y” and “z”, use the command: java foo x y z (where x is the first argument, y is the second, and z is the third)
* Specify a programs classpath – The classpath tells the Java compiler or Java Virtual Machine where to look for user-defined packages classes and packages when compiling and running.
  + When compiling or running a program, use the command: -cp <directory>
    - This command goes after *java* or *javac* and before the <programName> when running or compiling
    - To specify the directory:
      * “.” Refers to current directory
      * “~” Refers to home directory
      * “:” is used to separate multiple directories (if more than one is needed)
      * “/” is used before and after a directories name
      * Example:
        + ./thisPackage.jar/

Refers to thisPackage.jar in the current directory

* Specify maximum heap size – when your program needs a lot of memory to run (such as our write and read programs!) it is necessary to raise the maximum heap size, otherwise an error will occur saying that there is not enough heap space
  + - When running a java program use the command: java –Xmx<amountOfSpace> <programRunSpecifications>
      * <amountOfSpace> examples:
        + 20g = 20 gigabytes
        + 5m = 5 megabytes
        + It is fine to set <amountOfSpace> to a larger size than needed. It is only a flag that stops the program if the flag is reached.
      * Doing this increases the limit on the amount of heap space your program can take up. You still may get a garbage collector error, that means that your program is basically stalled and 98% of the work being done is cleaning (your program is getting nothing done).

**Program Compile and Run Specifications**

* Compile Read.java
  + Command: javac Read.java
* Run Read.Class - increase maximum heap, and pass two valid arguments:
  + Command: java –Xmx<size> Read <arg1> <arg2>
    - <size>: depends on size of the file being read and storage method
      * Use <size> = 100g, if unsure
    - <arg1>: can be either the String "one" or "two"
      * "one", specifies a single node file read, thus the program will be reading the file fileRead15GB.txt
      * "two", that specifies a 15 node file read, thus reading the file fileRead.txt (which is 1GB)
    - <arg2>: can be either the String “single”, "eighth", "quarter", "half", or "full" – specifies storage method (Note that fileRead.txt has approximately 13 million lines, and that fileRead15GB.txt has approximately 200 million lines)
      * “single”, for storing one line of the text file being read at a time
      * “eighth”, for storing one-eighth of the lines, in the text file being read at one time
      * “quarter”, for storing one-quarter of the lines in the text file being read at one time
      * “half”, for storing one-half of the lines in the text file being read at one time
      * “full”, for storing all of the lines in the text file being read at one time
* Compile NRead.java – specify classpath
  + Use command: javac –cp ‘./netcdfAll-4.6.jar/’ NRead.java
    - The jar file netcdfAll-4.6.jar is needed in /opt/jwoodring/ (should already be there)
* Run NRead.class – specify classpath, increase maximum heap, and pass two valid arguments:
  + Use command: java –Xmx<size> -cp ./netcdfAll-4.6.jar/:./slfj-nop-1.7.jar/:. NRead <arg1> <arg2>
    - The jar file netcdfAll-4.6.jar is needed in /opt/jwoodring/ of each machine for the use of netcdf in java programs (should already be there)
    - The jar file slfj-nop-1.7.12.jar is needed in /opt/jwoodring of each machine to avoid a logger error while running NRead.class (should already be there).
    - <size>: depends on storage method being used (arg2)
      * Use <size> = 20g, if unsure
    - <arg1>: must be the String “two”
    - <arg2>: can either be the String “single”, “eighth”, “quarter”, “half”, or “full” to specify how the data is stored
    - “single” for storing only one float at a time
      * Only one record of the data variable is stored at one time. There are 1000 x 1000 x 1000 (1 million) records of the variable so this happens to be a very time consuming process
    - “eighth” for storing one-eighth of the Netcdf’s float values at a time
      * NRead will break up the 1000 x 1000 x 1000 array into eight parts along the x axis. All data records are read but only one-eighth are stored at once
    - “quarter” for storing one-quarter of the Netcdf’s float values at a time
      * NRead will break up the 1000 x 1000 x 1000 array into four parts along the x axis. All data records are read but only one-fourth are stored at once
    - “half” for storing one-half of the Netcdf’s float values at a time
      * NRead will break up the 1000 x 1000 x 1000 array into 2 parts along the x axis. All data records are read but only one-half are stored at once
    - “full” for storing all of the Netcdf’s float values at one time
* Compile WriteNetcdf.java – specify classpath:
  + Use command: javac –cp ‘./netcdfAll-4.6.jar/’ WriteNetcdf.java
    - The jar file netcdfAll-4.6.jar is needed in /opt/jwoodring/ (should already be there)
* Run WriteNetcdf.class – specify classpath, and increase maximum heap:
  + Use command: java –Xmx<size> -cp ./netcdfAll-4.6.jar/:. WriteNetcdf
    - The jar file netcdfAll-4.6.jar is needed in /opt/jwoodring/ of each machine for the use of netcdf in java programs (should already be there)
    - <size>: depends on storage method being used (arg2)
      * Use <size> = 20g, if unsure
  + The newly written Netcdf file should be in the same directory that the program was ran in