Christopher DeRoche – Assignment 1

* Observations
  + Fully Tested/Created on IU Silo.
  + The biggest issue with this lab was timing of nodes with the main. Looking back, I would potentially use an asynchronous communication format and counter. It is something I have not messed with a lot but using something like events would have simplified some of the issues I have when creating this.
  + I think using sockets and having to keep live connections between everything probably ended up using more system resources overall and could have complicated things, but overall provided a TCP connection to guarantee data between main and sub nodes.
  + Inverted index is designed using a 1 document to 1 mapper layout so there is the assumption of a one-to-one format there. The program is currently written for a max of 2 files for inverted index and a single word count file. Any number of reducers can be used with either mode. Word count is also not limited by mappers in terms of how many you can spawn. **Inverted index must take 2 files to run.**
  + There are default values for every single argument passed into the program so almost every time, if data is there, the program can run.
  + The program does not generally use a lot of data storage and relies mostly on memory, and I think it is quite fast, this potentially created issues with timing so there are sleep functions called in some places. The only files used for storage are server/main\_node side. The data retrieved back from the nodes written to a file and read by another thread in the main\_node. This does not break the nodes accessing each other’s data requirement for the assignment. All data is only accessed locally by the main\_node and it is accessing its own data. The only way nodes and main can communicate is through sockets.
  + All data is sent between main\_node and other nodes using the Pickle Python object serialization package. This simplified sending data around to nodes and back immensely, but also required less communication and parsing on each side. Everything could be either sent as a Python list or dictionary then un-pickled then processed. This helped achieve a lot in terms of less sending and individual parsing of data.
* Main Node:
  + The main\_node.py is considered the master node that controls all the operations. This is the main node that spawns the mapper and reducer nodes as well. Everything is done in this application as threads to act as if they are running on separate machines. The main node creates a thread for each node to run inside of. This implementation of MapReduce uses sockets to communicate between main and sub nodes. Since this required a multi-threaded server-side application as well, there are threads for each server-side connection to process multiple nodes sends at once. Every major step of the process is logged using print statements along with the final output is printed at the end of the program.
  + main()
    - The main method processes the command-line arguments that will give the application the configuration required. This method also facilitates all the operations within the main node in sequential order of MapReduce. After all the functions are run and data is MapReduce there is a block of code to print the final MapReduce data. It reads from the temporary server-side file with the reduced data then destroys any temporary files used for operation. All temporary files are server-side only. The stdout is flushed and os.\_exit(0) is called for successful operation and to kill anything else running.
    - Possible Arguments:
      * Wordcount -w 🡪 Boolean to run wordcount
      * Invertedindex -I 🡪Boolean to run inverted index
      * Address -a 🡪String format host name
      * Port -p 🡪Integer port number
      * Mappers -m 🡪Integer number of mappers
      * Reducers -r 🡪Integer number of reducers
      * Input\_file -f 🡪String input file
      * Input\_file2 -f2 🡪 String input file two
    - Operations in Order:
      * Process command-line arguments
      * Spawn and start the main node thread
      * Initialize the cluster nodes
      * Read the input files data and separate it into chunks for mappers
      * Send the data to the mappers
      * Shuffle the returned data from the mappers
      * Send the data to the reducers
      * Print out the final reduced data
  + init\_main\_node(address, port, word\_count, inverted\_index, mappers, reducers):
    - Creates the server-side socket and binds it to the given address and port.
    - Creates a thread and class that will listen to each node individually for processed data.
  + MapperConnection class 🡪node\_listener(conn, addr)
    - Created by the init\_main\_node method.
    - Sets up the loop and recv functions to listen for data from the mappers.
    - Reads in the processed data and un-pickle it.
    - Appends the data to a temporary server-side only file with the mapped data.
    - Once a signal of “WORDCOUNTED” or “INVERTEDINDEXED” is given, the file is closed and saved.
    - The node then kills itself because the mapper is no longer needed, so the connection is also terminated.
  + ReducerConnection class 🡪 node\_listener(conn, addr)
    - Created by the init\_main\_node method.
    - Sets up loop and recv functions to listen for data from the reducers.
    - Reads in the processed data and un-pickle it.
    - Appends the data to a temporary server-side only file with the reduced data.
    - Once a signal of “WORDCOUNTED” or “INVERTEDINDEXED” is given, the file is closed and saved.
    - The node then kills itself because the reducer is no longer needed, so the connection is also terminated.
  + init\_cluster(mappers, reducers, address, port)
    - Takes in the number of mappers and reducers.
    - Takes in the host and port.
    - Creates the mapper nodes with a loop for the number of mappers there will be. The address and port are passed to the threads.
    - Creates the reducers nodes with a loop for the number of reducers there will be. The address and port are passed to the threads.
  + read\_and\_split\_data(input\_file, mappers, invertedindex)
    - Opens the file from the passed string of file name.
    - Reads in the data and closes the file.
    - Splits all the words into a list separated by spaces.
    - Gets the length of the data.
    - If we are doing an inverted index, just return the list of words from there.
    - Else continue
    - Separate the data into evenly spaced chunks. This is split by the number of mapper nodes, so each mapper gets some data.
    - Return the chunks of data back to the main
  + send\_to\_mappers(chunks, mappers, wordcount, invertedindex)
    - If wordcount
      * Loop through the chunks of data.
      * Create a msg to send to the mapper, but first pickle the data because we can send an entire list object packed together.
      * Iterate through the nodes available and send it.
      * When complete sending to mappers, send a wordcount signal to the mappers to begin the processing at the nodes.
    - If invertedindex
      * Package up the chunks of data into a message object using pickle.
      * Send to the passed mappers variable.
      * Send another message to the mapper to run inverted index on the data.
  + shuffle\_data(mappers, reducers)
    - While loop
      * Loop through each mapper and see if the connection has been killed. This will tell us if all the data is done. Once the mappers are all disconnected then break out of the loop to shuffle data.
    - Open the temporary server-side file with all the mappers data.
    - Split the data into a list to then be sorted.
    - Sort the data alphabetically.
    - Separate the data into chunks but grouped by word. So, there is a list of lists. The main list holds sub lists of the same word together.
  + send\_to\_reducers(chunks, reducers, wordcount, invertedindex)
    - For each chunk of chunks of data
      * Use pickle to package list and create a message.
      * Send the message at the index reducer connection.
    - For each node of reducers
      * Send the signal to either process wordcount or inverted index.
  + wait\_for\_reducers(reducers, wordcount, invertedindex)
    - This holds up the main until reducing is done so data can be finalized and printed out.
    - This replicates the same loop used earlier to wait for the mapper nodes.
    - While loop
      * For each node send a signal to check connection status.
        + If there is success, then go to the next node.
        + If there is an error then mark complete, destroy node in array
* Mapper Node:
  + init\_mapper()
    - Created in a thread from the main\_node with passed address and port
    - While loop
      * Get the data from the main\_node and un-pickle it.
      * Add the data to the list
      * When signal given, process either word count or inverted index
  + word\_count(mapper, data\_arr)
    - for each word in the data arr
      * Remove any periods or commas
      * Add the “, 1” mapping value
      * Append to the final array
    - Pickle the message to send
    - Send it to the main\_node
    - Send complete signal to main\_node
  + inverted\_index(mapper, data\_arr, doc)
    - for each word in the data arr
      * Remove any periods or commas
      * Add the “, 1” mapping value
      * Append to the cleaned-up array
    - Sort the array
    - Put the data into lists of lists sorted by alphabetical words
    - For each word in data
      * Count the number of values in the sub list
      * Get the word itself
      * Append the result to the mapped array. This includes the word, document number, and number of occurrences.
    - Pickle the mapped list
    - Send to the main\_node
    - Pick and send final signal to main\_node
* Reducer Node:
  + init\_reducer()
    - Created in a thread from the main\_node with passed address and port
    - While loop
      * Get the data from the main\_node and un-pickle it.
      * Add the data to the list
      * When signal given, process either word count or inverted index
  + word\_count(reducer, data\_arr)
    - for i in the range of the length of the data\_arr
      * Get each word
      * Remove the “,1” mapping
      * Put the word back into the data array at the same position
      * This will make it easier for processing
    - Create a map of the data. This will have each word with the number of times it occurs because a for loop counts the number of times.
    - Pickle the finalized data and send it back to the main\_node
    - Pickle and send the final signal to the main\_node
  + inverted\_index(reducer, data\_arr)
    - Sort the data passed
    - Create a list with sub lists of like words while also removing the “,1” mapping from it.
    - Loop through each sub list
      * If the length is greater than or equal to 2
        + Get the word as the key and values will be a string with the document number and number of occurrences.
      * Else if the length is one
        + Get the word, document, and number of occurrences.
    - Pickle the data and send it back to the main\_node
    - Pickle and send the final signal back to the main\_node