**Code Instructions**:

**Compile-**

To compile the code, run these commands in bash, with the current directory set to the folder in which the files reside.

Here are the compile commands for my serial implementation:

g++ -o3 -w lcovingtAssignment1Serial.cpp -lpthread -o lcovingtAssignment1Serial

Here are the compile commands for my multithreaded implementation:

g++ -o3 -w lcovingtAssignment1.cpp -lpthread -o lcovingtAssignment1

IE the terminal will look something like this

user@computer:~/folder/Assignment1$ g++ -o3 -w lcovingtAssignment1.cpp -lpthread -o lcovingtAssignment1

**Run-**

Next, to run the code utilize the following format for the serial program:

./lcovingtAssignment1Serial graphFilePath k

For the multithreaded code, utilize the following format:

./lcovingtAssignment1 graphFilePath K numberOfThreads

**Example Commands -**

Here are a couple example commands for ease of use:

./lcovingtAssignment1 networkDatasets/graphV1000.txt 3 4

./lcovingtAssignment1Serial networkDatasets/graphV1000.txt 3

**How the code works:**

The code starts by initializing a two dimensional vector to store all the friendships as 1’s for when they are friends and 0’s to represent that they aren’t friends. Next we initialize another two dimensional array of the exact same size and populate it with -2’s. This array is called commonFriendCounts. CommonFriendCounts stores the number of common friends a person has with another. Next there are a couple of special values stored in commonFriendCounts. The initial values are all -2. That value means that a common friend count has not yet been calculated. The next important value stored in commonFriendCounts is -1, that value means that the two people are already friends. Later on I do a max search using common friend counts to build recommendation lists, -1 insures that they will be towards the bottom of that max list. I check to see if a possible recommendation’s common friends with an individual of interest is -1, if that is the case, they are not added to the recommendation lists we end up printing. The next big global data structure used is also a two dimensional array. It is called sortedRecommendationList. The first index into the list takes you to another list where a person’s recommendations will reside. It is accessed using that person’s/node’s value as an index. Now with the major data structures explained, let’s move on to the helper functions used to accomplish the task. Now after we have initialized and set up our datastructures to avoid throwing exceptions, the next course of action is to calculated everyone’s common friend counts with eachother. This is accomplished by calling a two layered nested for loop that iterates over all the node counts twice. In the multithreaded version, we iterate over our assigned segment for the first loop, and to ensure we cover every cell, all threads iterate over all nodes for the second interior loop. However un-optimal, if a common friend count has already been calculated, ie the cell is not equal to -2, the thread is to immediately return and try it’s next calculation. The method inside that double layered for loop is calcAndSaveNumberOfSharedCommonFriends. It takes two integers, which are used to represent nodes. The details will be spared of the function, as it can be quite complicated, but after it has been iterated for all cells, all common friend counts should be greater than the initial set values of -2. The next function to mention is the create recommendation list function. It takes an integer representing a node. Then it constructs a list of candidates, all of the other nodes in the graph. It then does a max search checking common friend counts with all of it’s candidates. When it chooses the max candidate it push\_back the new recommendation to it’s sortedRecommendationList. It then removes that recommendation from the candidate list, and repeats this process k times. Once the recommendation lists are all constructed, we simply print out all of our freshly made recommendation lists at the end of the program. That is the general gist of how the program works. Somethings to note, with only the common friend counts being parallelized the speed up gains of the program for larger graph sizes don’t quite match expectations. I believe there are some optimizations that could be made to the regular code, yet the creation of recommendation lists sure could be parallelized as well yielding better speedup.

**Final Notes:**

If there are any problems with the code compiling or running please email [leighton.covington@ndsu.edu](mailto:leighton.covington@ndsu.edu). I have spent a tremendous amount of time on this code, and I have gotten it to run and work with the example command formats provided.