ECE 454: Assignment 2B
Simarpreet Singh: s244sing@uwaterloo.ca | 20434209
Caitlin Pinn: cpinn@uwaterloo.ca | 20368217

The implementation used ensured both reading and computation are multithreaded, meaning each thread reads a different chunk of the problem. The reads of the byte[] input were parallelized by selectively reading chunks of the given input and skipping over the rest in each thread. This way we were able to add all the required data from the input byte[] array into the concurrent hash map that stored the input as an adjacency list. For computing the triangles we implemented a similar strategy; read only selective portions of the adjacency list. We partitioned the graph input data among threads and treated each input of the graph array as a separate problem. In case of the single thread implementation, all work is done by the main thread.

The input file is read using one thread if it is single threaded or read using multiple threads if there are multiple cores, one thread for each core. If reading is done in parallel, each thread reads every other line of the file. Through the reading of the file, an adjacency list graph representation can be obtained. The adjacent list is created using a concurrenthashmap where each vertex is used as the key and the value is a HashSet of the adjacent connected vertices. The file is read using a strictly greater than formula where only neighbours whose value is greater than the current vertex are added to the hash set of each vertex. This way we are able to obtain the sparse representation of the graph and our complexity for both time and memory considerably drops down as there are many vertices in the input files where there are duplicated neighbours. This way we are able to ensure that only unique nodes get added and also the input size of the graph list is not more than what we need. We also looked into adding nodes into the graph representation in order as this would result in some of the if-else conditions checks getting reduced in our worker thread but ultimately, this optimization seemed to have next to negligible gains.

In our algorithm, we also used the idea of intersecting the adjacency lists of different vertices and computing the common neighbours among them. This helped our further shorten out the adjacency lists to iterate over in our third for-loop as we would only iterate selectively over the nodes that could possible be forming a triangle. This reduced our computation time by quite a bit as this approach favors temporal and spatial locality of the caches. Caches are not trashed with new data and most of the cache accesses are hits since the data to iterate over already exists in them. We further looked into the idea of deleting the vertices that were either 1: already part of another triangle and had no other neighbours and 2: vertices that were empty to begin with as they did not met the “strictly greater” condition of the forward algorithm. This however didn’t result in much of an improvement as seen in our tests.

The implementation for multithreading utilized a fixed thread pool. The fixed thread pool uses a fixed number of threads to operate, in our case this being equal to the ncores parameter. We also looked into implementing a scheduled thread pool executor, but in our case the worker threads were built in a way to be mutually exclusive of each other and therefore we did not need the features that the scheduled thread pool executor had to offer.