ECE454 - Assignment 2 Part B - July 5th, 2015

Aaron Morais - aemorais
Fasih Awan - faawan
Victor Lai - v6lai

Algorithm

```
for each smallNode do:
    smallNeighbours = smallNode.getBiggerNeighboursSorted()
    for each mediumNode in smallNeighbours do:
        mediumNeighbours = mediumNode.getBiggerNeighboursSorted()

        if node exists in both smallNeighbours and mediumNeighbours do:
           add new Triangle(smallNode.Id, mediumNode.id, element)
        endif
    endfor
endfor
```

From an adjacency list consisting of (vertex, list of neighbour vertices), keep only the neighbour vertices with values larger than the root vertex, and sort the list of neighbour vertices.

For every neighbour vertex, get the neighbour's list of neighbours, and iterate through both the root vertex's and the neighbour vertex's list of neighbours incrementally using 2 iterators. If the iterator's values are equal, then both the root vertex and neighbour vertex share a common neighbour, forming the triangle (root vertex, neighbour vertex, common neighbour vertex).

The main speed up occurs from the efficient accesses using 2 iterators. Instead of requiring nested for-loops to check for matching elements, only 1 iteration of twice the size occurs.

Multi-Threaded Optimizations

Each thread will parse a different line and insert at a fixed index in the main array. Because each line represents a different vertex, threads will never access the same array index at the same time, allowing parallelism without synchronized data structures.

To divide the triangle count work, each thread has a reference to the neighbours list generated from parsing the input, along with a starting index in the range of [0, numCores) and a step variable to increment the index by. Thus, each thread will access a different vertex to avoid duplicate work, and because only accesses are allowed, there is no need for synchronization.