**CSC3021 Concurrent Programming Assignment 2**

**Question 1: Analysis**

**CSR**

By looking through the code for my current CSR implementation, CSR will have issues with race conditions. This is due to the way that **outdeg** variable is calculated. In CSR, we calculate the **outdeg** variable by doing from the current source vertex to the next source vertex. This could lead to a situation where we suddenly jump to a completely different source vertex which would change the value that is calculated. To ensure that there would be no data races in the CSR implementation we would need to add some form of mutual exclusion to the iterate() method around the main calculation. By implementing some form of mutual exclusion, we could ensure that only one thread is able to access the critical section for CSR which should prevent data races from happening. The following code that would go into the critical section is;

for (int j = source[i]; j < source[i + 1]; j++) {  
 outdeg = source[i+1] - source[i];  
 out[destination[j]] += a \* (in[i] / outdeg);

}

This code would need placed in the critical section because if two threads where trying to calculate the outdeg, there is a chance that one of the outdeg values may be calculated incorrectly which would lead to an incorrect value at the end.

**COO**

By analysing COO, we can also see that race conditions can exist in the iterate() method. This is because of the traversal method of COO. CSR and CSC both use vertex traversal, which is ordered, COO uses edge to edge traversal. A COO graph represents a source and destination on a single line, which represents an edge. Due to the nature if this graph, it is unsorted. This could lead to some potential race conditions where 2 threads might operate on the same edge twice which would lead to a miscalculation. To try and get a COO graph to work concurrently we could try ordering the graph to ensure that threads will not operate on the same edge. We could also incorporate mutual exclusion around the critical section of the iterate() of COO to stop multiple threads from entering the critical section at the same which would stop race conditions. This critical section for COO is;

out[source[i]] += a \* (in[destination[i]] / outdeg[destination[i]]);

This code would need placed in the critical section because if 2 threads where to enter the critical section, there is a chance the same edge could be read twice, which would cause a miscalculation.

**CSC**

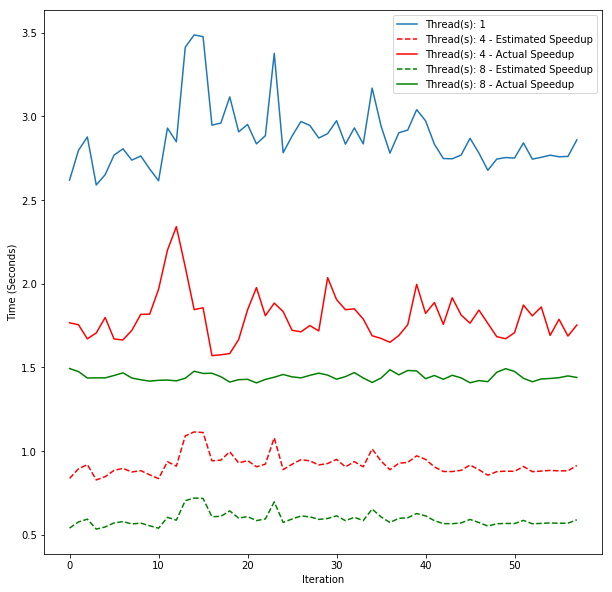
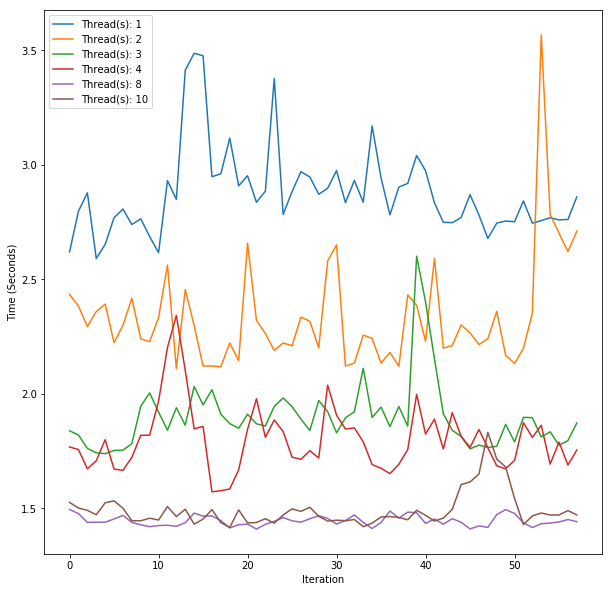
CSC should be relatively free from data races. The outdegree is calculated outside of the iterate() method unlink CSR and unlike COO, it shares a similar ordered traversal method to CSR, a list of vertex, first one is destination and the rest in the line are sources from the destination vertex. Since this is ordered, we will not run into the same issues that we would face with an unordered list. Because of the above point, this would make CSC the best graph to make concurrent

**Execution time and Amdahl law**

|  |  |  |  |
| --- | --- | --- | --- |
| **Matrix Type** | **Original Execution Time** | **Amdahl’s Law (4 Threads)** | **Amdahl’s Law (8 Threads)** |
| COO | 340.68440489599993 seconds | 3.057368466533027x speedup | 4.652783462893651x speedup |
| CSR | 87.492303608 seconds | 2.7482995771604593x speedup | 3.8784005981187826x speedup |
| CSC | 146.10971834400004 seconds | 3.12615545618925x speedup | 4.841942552161391x speedup |

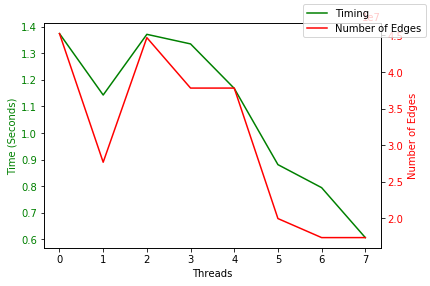
**Question 2: First Implementation**

The computer I ran my PageRank Implementation has a 4-core processor and 8 threads. Below are graphs showing execution time for using threads 1, 2, 3, 4, 8, 10 and when using 4 and 8 threads against the speedup estimated from Amdahl Law



As we can see from the above graphs, we can see that by using concurrency, we can get a significant speedup over our sequential implementation. In the first graph we can see random spikes like thread 2 taking a longer time to execute one iteration than 1 thread. This may be due to some process running in the background that were not happening during the 1 thread execution. When we look at using 8 threads compared to the 1 thread results, we can see that the time at least 2x as fast as the 1 thread results. When we look at the estimated speedup versus the actual speedup, we see that the estimated results are 2x faster than actual results. These results show that there are still optimisations that I could make to try and get closer to the speedup that I estimated using Amdahl’s law.

**Question 3: Measuring Workload Balance**

****

As we can see from the above graph, there does seem to be a correlation between time taken for a thread to finish execution of the iterate() method and number of edges that a thread need to compute. Thread 0 has the most edges that needs to be computed and it takes the longest amount of time to finish execution. Thread 7 has the least amount of edges that need to be computed and also fastest to finish execution. Apart from an abnormality with thread 1, more edges are calculated with the first threads as opposed to the last threads. This shows that there is an imbalance in the program where starting threads compute more than the last threads. To solve this problem, I will need to find a way to balance the threads among all the processors