CSC3021 Assignment 3

Question 1 - Part A

Amdahl's Law is used to calculate the overall speedup of an algorithm. It depends on the parallel fraction of the algorithm and the number of threads that will be used to execute it.

To find suitable values for the Amdahl's Law equation, the parallel fraction for the COO, CSC, and CSR sparse matrices had to be calculated. I added code to print out the time taken to complete the edgemap function (parallel section) and then this time was divided by the total time taken to execute that particular power iteration. This was then repeated for every power iteration (58 times) then the average parallel fraction was calculated for that particular sparse matrix. This was then repeated for each of the three matrices.

Below is a table showcasing the average parallel fractions calculated from the three matrices:

COO	CSC	CSR
0.990000 (6sf)	0.977525 (6sf)	0.953450 (6sf)

Below is the speedup equation for Amdahl's Law:

$$Speedup = \frac{1}{(1-p) + \frac{p}{n}}$$

Where p is the parallel fraction and n is the number of threads. We can use Amdahl's Law (from above) with the parallel fraction values obtained for the COO, CSC, and CSR sparse matrices to calculate the speedup that may be achieved on both 4 threads and 1000 threads, as shown below:

COO – 4 threads

$$Speedup = \frac{1}{(1-0.990000) + \frac{0.990000}{4}}$$

$$Speedup = 3.88\ (3sf)$$

COO – 1000 threads

$$Speedup = \frac{1}{(1-0.990000) + \frac{0.990000}{1000}}$$

$$Speedup = 91.0\ (3sf)$$

CSC – 4 threads

$$Speedup = \frac{1}{(1 - 0.977525) + \frac{0.977525}{4}}$$

$$Speedup = 3.75\ (3sf)$$

CSC – 1000 threads

$$Speedup = \frac{1}{(1 - 0.977525) + \frac{0.977525}{1000}}$$

$$Speedup = 42.6\ (3sf)$$

CSR – 4 threads

$$Speedup = \frac{1}{(1 - 0.953450) + \frac{0.953450}{4}}$$

$$Speedup = 3.51\ (3sf)$$

CSR – 1000 threads

$$Speedup = \frac{1}{(1 - 0.953450) + \frac{0.953450}{1000}}$$

$$Speedup = 21.1\ (3sf)$$

As shown by the results, the general trend is that the sparse matrices with the higher parallel fraction result in a higher speedup compared to the lower parallel fraction matrices, for the same number of threads.

COO has the highest parallel fraction and hence has the greatest potential speedup out of the three formats whereas CSR has the lowest parallel fraction and hence has the lowest potential speedup out of the three formats. This is due to the nature of the mathematical equation, where the speedup depends on the size of the parallel fraction.

The edgemap function is the (future) parallel section that I'm measuring. The calculation of the PageRank calculation is a very important/long part of the power iteration steps so the edgemap section takes a significantly long time to complete and hence all three sparse matrices have a very high parallel fraction (all above 95%).

The longer it takes a matrix to complete all the power iteration steps, the longer it takes the edgemap to complete in comparison to the rest of the sequential section and hence the higher the parallel fraction (COO: 340sec, CSC: 161sec, CSR: 79sec).

Question 1 - Part B

To observe and describe possible race conditions in the parallel edgemap function, we must first define a race condition as the behaviour of a program where the outcome of the program depends on the ordering or timing of the events (and hence the interleavings). Where there is a shared resource between separate threads, operations done on that resource must be atomic operations or they must achieve atomicity by design eg. a critical section.

Starting at the edgemap function call within the three sparse matrices (COO, CSC, and CSR) it can be observed that COO does not have an inner for loop for the relax call whereas CSR and CSC do have an inner for loop. As part of the inner for loop, an integer j variable is declared, but this is thread-safe because it is declared within each thread and is not shared between them.

In the edgemap function in the three sparse matrices, there are accesses to three different arrays in total – the source, destination, and index arrays. Since as part of edgemap these arrays are not altered and are only accessed from memory, there is no possibility for a race condition here so this is thread-safe.

All three sparse matrices call the PageRank relax method so it needs to be analysed for race conditions. Within the relax method, the outdeg and x arrays are accessed but don't have any stores so they are race-condition-free and hence thread-safe.

The y array, however, is accessed (at position dst) and incremented by the value w * x[src]. This is a problem, as although this is only one line in the program, when broken down it will be made up of multiple atomic operations as shown below:

$$R = y[dst];$$

$$R = R + value;$$

$$y[dst] = R;$$

As shown above, there are multiple atomic operations as part of the y[dst] line and there is no critical section, locks, semaphores, monitors, or any other form of synchronisation to prevent interleavings of these atomic operations between the threads running edgemap. This is a huge problem as this means that depending on the execution and interleavings of the different threads, the loads and stores of the y array incrementation functionality could be overwritten by each other and hence cause incorrect values and erroneous behaviour.

Therefore, this current implementation of the relax method has the possibility of race condition (more specifically a data race). **It is extremely key to note that this race condition will only have the possibility of occurring if multiple threads are trying to perform the incrementation operation on the same position (dst) in the y array.**

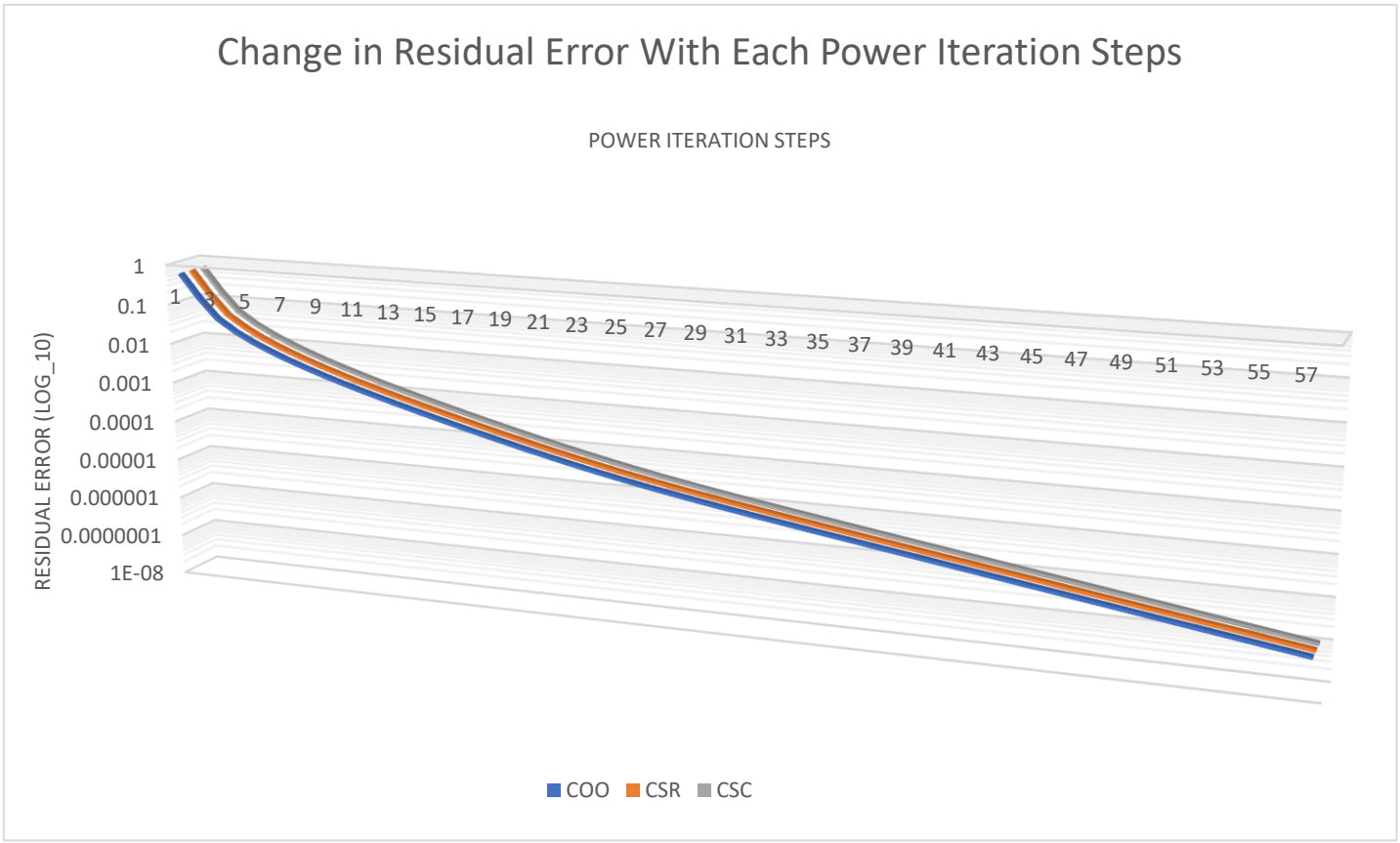
This means that each of the sparse matrices must be checked to see whether their dst (destination) value passed to the relax method could be the same only multiple threads. Both COO and CSR have the possibility of calling the relax method with the same destination (dst) value on different threads and hence have the possibility of a race condition when using relax. This is explained further by the fact that on multiple different iterations of the outer for loop in edgemap, there could be the same dst value passed to the relax method. Since the outer for loop iterations are being shared between

the threads, there is no way to guarantee which threads call certain iterations, and hence both CSR and COO have the possibility of race conditions and are not thread-safe.

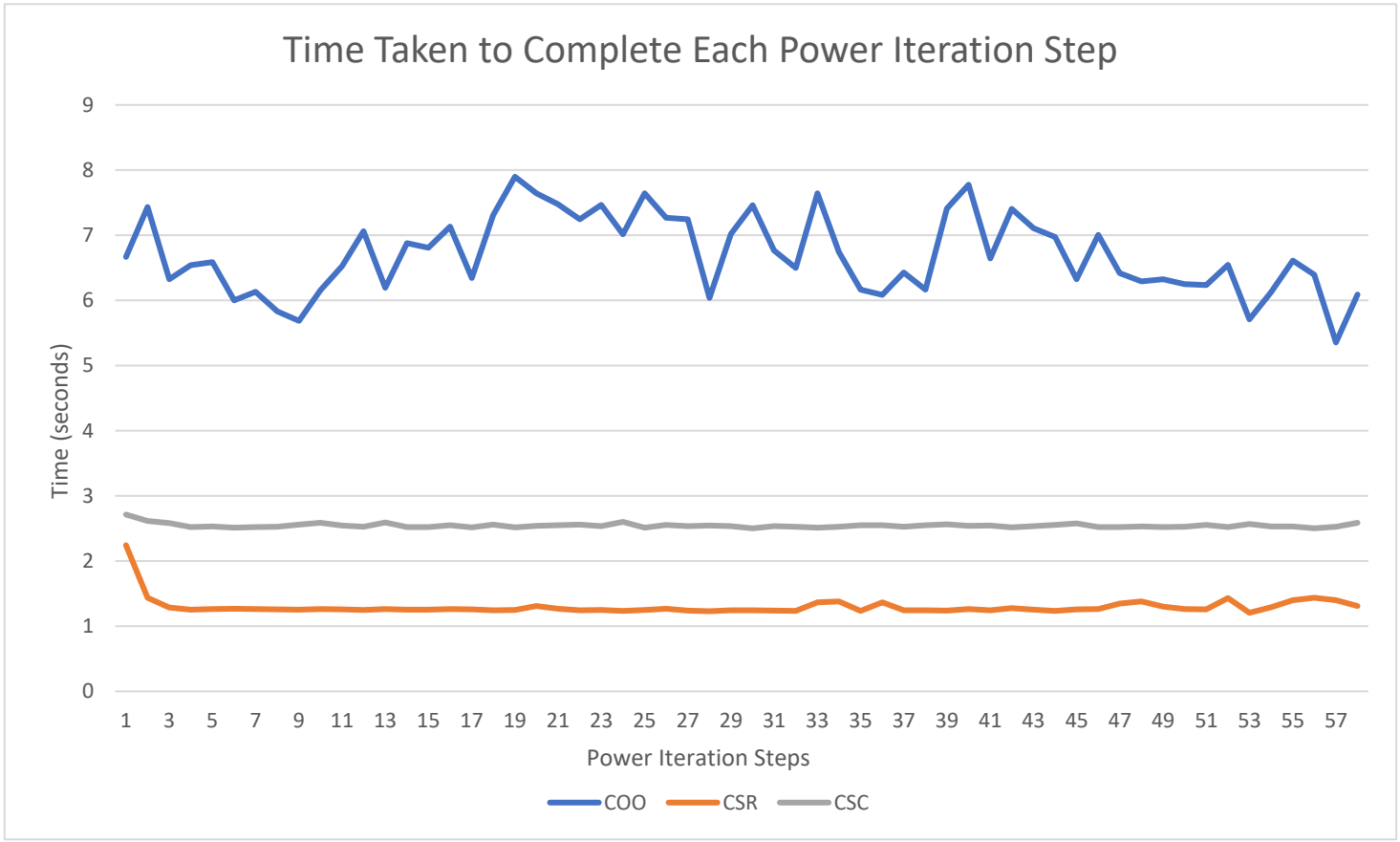
CSC on the other hand can be analysed to see that in edgemap, the dst value passed on every inner for loop iteration is equal to the counter of the outer for loop. This means that every iteration of the outer for loop is going to pass a different dst value as a parameter to the relax method call. This means in turn that no two threads are going to be trying to do the y[dst] operation using the same dst value as each other and hence, there is no possibility of there ever being interleavings between the different threads for the atomic operations executed as part of the y[dst] incrementation, hence there is no possibility for a race-condition. The entire edgemap operation for the CSC sparse matrix is free of race conditions and hence is thread-safe. This is due to the nature of how CSC works and how it is stored, with each row in the txt file containing all the in-vertices.

In summary, COO and CSR have the possibility of facilitating race conditions between multiple threads due to the relax method and the dst values passed to that relax method. On the other hand, CSC is free of any possible race conditions due to the reasons explained above.

Question 1 - Part C



As shown above, it is clear from Assignment 1 that the residual error is exactly the same for all three sparse matrices across the 58 power iterations. The values obtained and hence the graph lines are exactly the same for all three sparse matrices as they all use the same power iteration formula method. This is as expected and hence the residual error can be ruled out as a way of determining which sparse matrix would be more appropriate.



The line graph, as shown above, shows the time taken at each power iteration step for each of the three different sparse matrices: COO power iterations take an average of 6.7 seconds, CSC take an average of 2.5 seconds, and CSR take an average of 1.3 seconds.

It is clear from the graph that the CSR and CSC formats are much more efficient than the COO sparse matrix for storage and usage with this graph dataset. This is due to the nature of the storage of the sparse matrices as discussed in detail previously eg. outdegree of CSR being much more efficient too. The average time taken at each power iteration step is an important point to consider and will be factored into the overall discussions shortly.

	COO	**CSC**	**CSR**
Average time taken to complete 1 power iteration step	6.7s	2.5s	1.28s
Parallel fraction	0.990000 (6sf)	0.977525 (6sf)	0.953450 (6sf)
Speedup (4 threads)	3.88	3.75	3.51
Speedup (1000 threads)	91.0	42.6	21.1
edgemap() method is currently thread-safe	no	yes	no

The table, as shown above, contains all the necessary factors needed to decide which sparse matrix type is best for a concurrent program.

The speedup at 1000 threads should be discounted from this comparison due to two reasons: Gunther's Law states that increasing the number of threads also increases the amount of overhead on the program and hence the speedup calculated here is not realistic. The other reason is the fact that most conventional laptops and PCs have quad-core processors. Processors that have more than 8 cores (and hence 16 threads) are fairly uncommon. Therefore the speedup at 1000 cores can be disregarded in this comparison.

When comparing the three sparse matrices it is clear that the COO format is considerably slower than CSR and CSC due to COO's storage format. COO does have a slightly higher parallel fraction as a result of the edgemap taking so long to complete and hence it will see a slightly greater speedup (at 4 threads) than CSR or CSC when run concurrently. This speedup however is negligible in comparison to the speedup it would need to compete with the speed of CSR and CSC. Finally, the fact that the COO edgemap method contains a race condition when calling the relax method makes it the definite worst option. Overall the COO sparse matrix is much worse in comparison to the CSR and CSC formats to be used concurrently and can therefore be disregarded.

That leaves the choice between CSC and CSR. From the table, it is clear that the average time taken to complete one power iteration step for CSR is approximately half the time it takes for CSC. This is an extremely big difference in performance. This is namely due to the calculateOutDegree() method where CSR only has to iterate through the vertices whereas CSC has to iterate through all the edges. The CSC sparse matrix should also be more suited to the orkut_undir graph data because there are many more rows than columns in the orkut_undir graph, so this can't be used as an excuse.

The CSC sparse matrix takes longer, so the parallel section is slightly greater than CSR, hence the speedup is slightly bigger than CSR (around 5% greater speedup). This is negligible compared to the 100% time difference between the time taken at each power iteration step.

So far CSR looks like a much better option than CSC as CSR is around twice as efficient. However, with the current implementation of the edgemap, the CSR sparse matrix implementation contains the possibility of a race condition whereas the CSC does not (this is explained in detail in Question 1B). This means that the CSC sparse matrix edgemap method could be converted to be used concurrently without any expected problems, whereas the race condition potential in the CSR matrix needs to be dealt with first before the edgemap method can be converted to concurrent execution. The magnitude of this problem must be analysed first before a final comparison between CSC and CSR.

The CSR sparse matrix edgemap method (specifically the relax method of PageRank) could be altered so that the y[dst] =… line could be converted to be done atomically, whether that's through creating a critical section around this line or by converting this line to be executed atomically (instead of the 3 atomic instructions it currently takes). The disadvantage of adding this mutual exclusion to the relax method is the overhead that it would entail as a consequence eg. all the threads waiting for the lock if a lock was used. Adding a lock in the relax method to create a critical section around the incrementation line (and hence mutual exclusion) would solve the race condition problem that is currently taxing the CSR method. A more efficient solution such as bare machine instructions would make for an even more efficient solution and reduce synchronization overheads – however these bare machine instructions still use locks at their core.

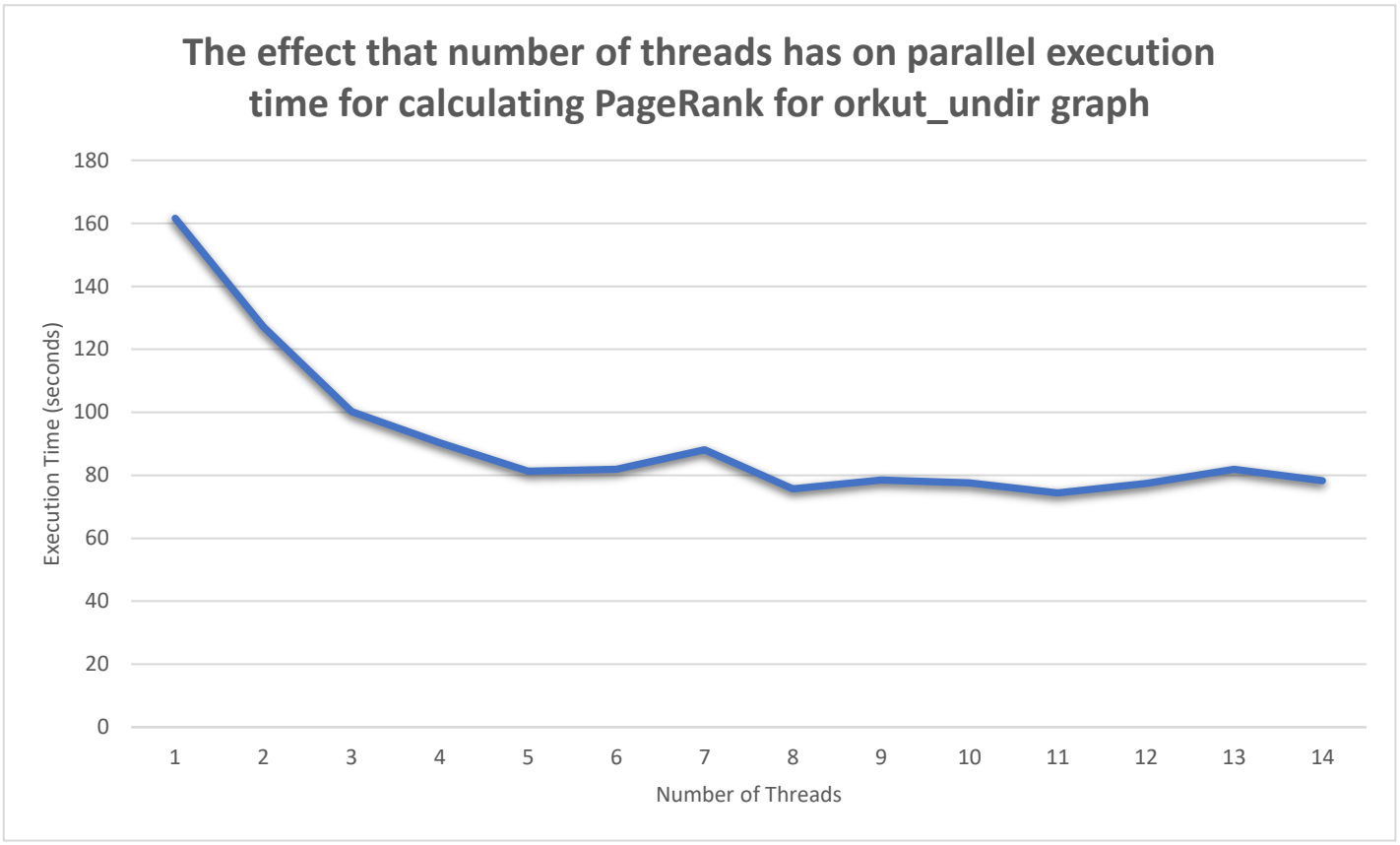
Locks would add too much of a performance overhead so the best option for making CSR thread-safe would be to use atomics/bare machine instructions such as the AtomicIntegerArray class and use a method such as getAndUpdate(). The methods within AtomicIntegerArray actually use locks and hence will add significant overheads to a very frequently called method such as relax.

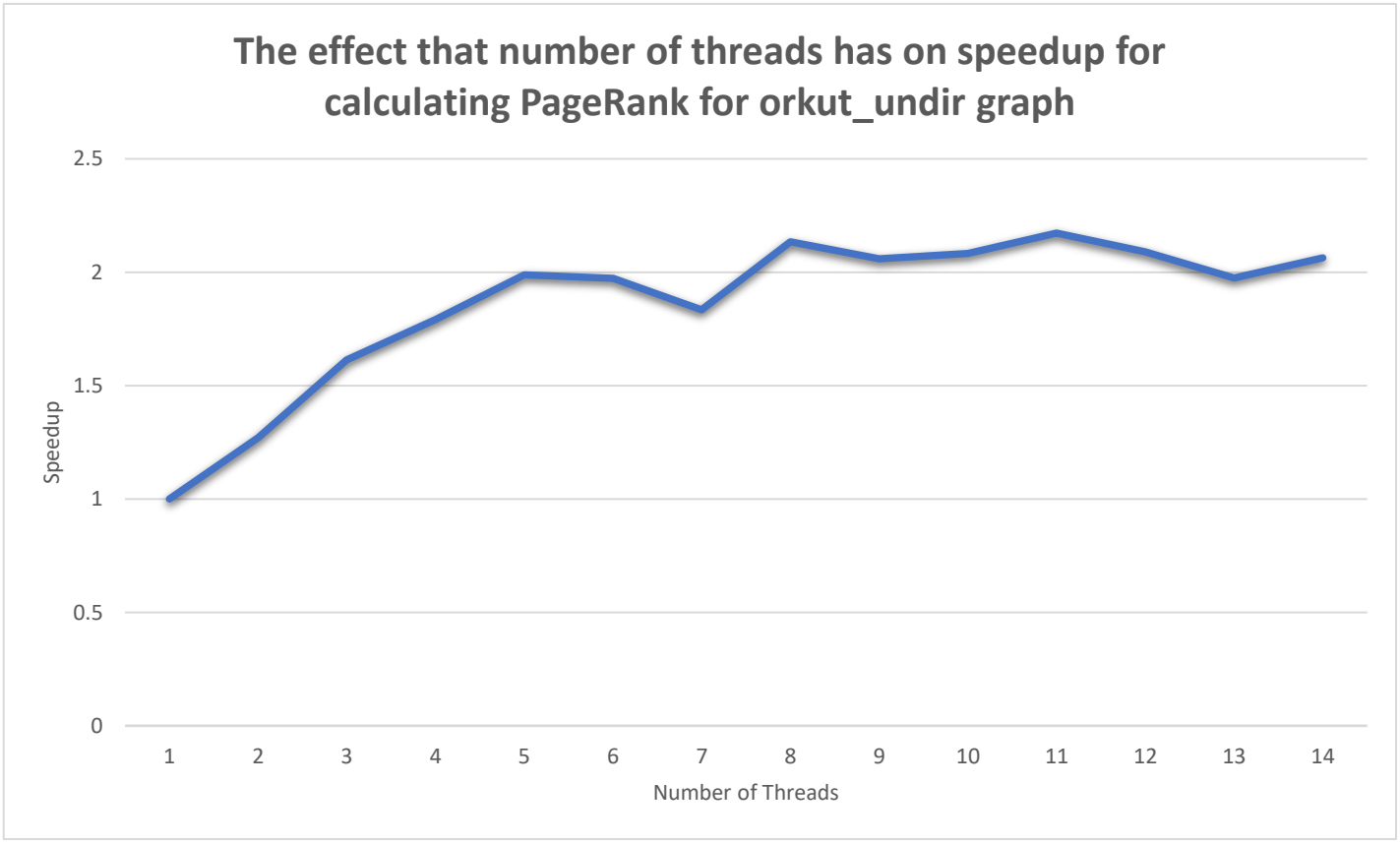
The decider between using the CSC sparse matrix (as it currently is implemented) or using the CSR format with added synchronization (critical section) between the y[dst] incrementation section comes down to the question of:

With the added synchronization using the AtomicIntegerArray class, would the CSR edgemap still be faster than the CSC edgemap method, taking into account the synchronization overheads of the CSR edgemap?

Initially, I predicted that a thread-safe CSR with the AtomicIntegerArray class would be more performant than CSC. However, after testing the CSR class with the AtomicIntegerArray class with some of its different methods later in this assignment, it turns out that getAndUpdate() (and the other methods) add a significant performance overhead due to locks used by the method and hence CSC will have the better performance. Therefore, I would use CSC as the graph format for this concurrent program due to the fact that it is more performant than a thread-safe CSR (once it has synchronization added).

Question 2

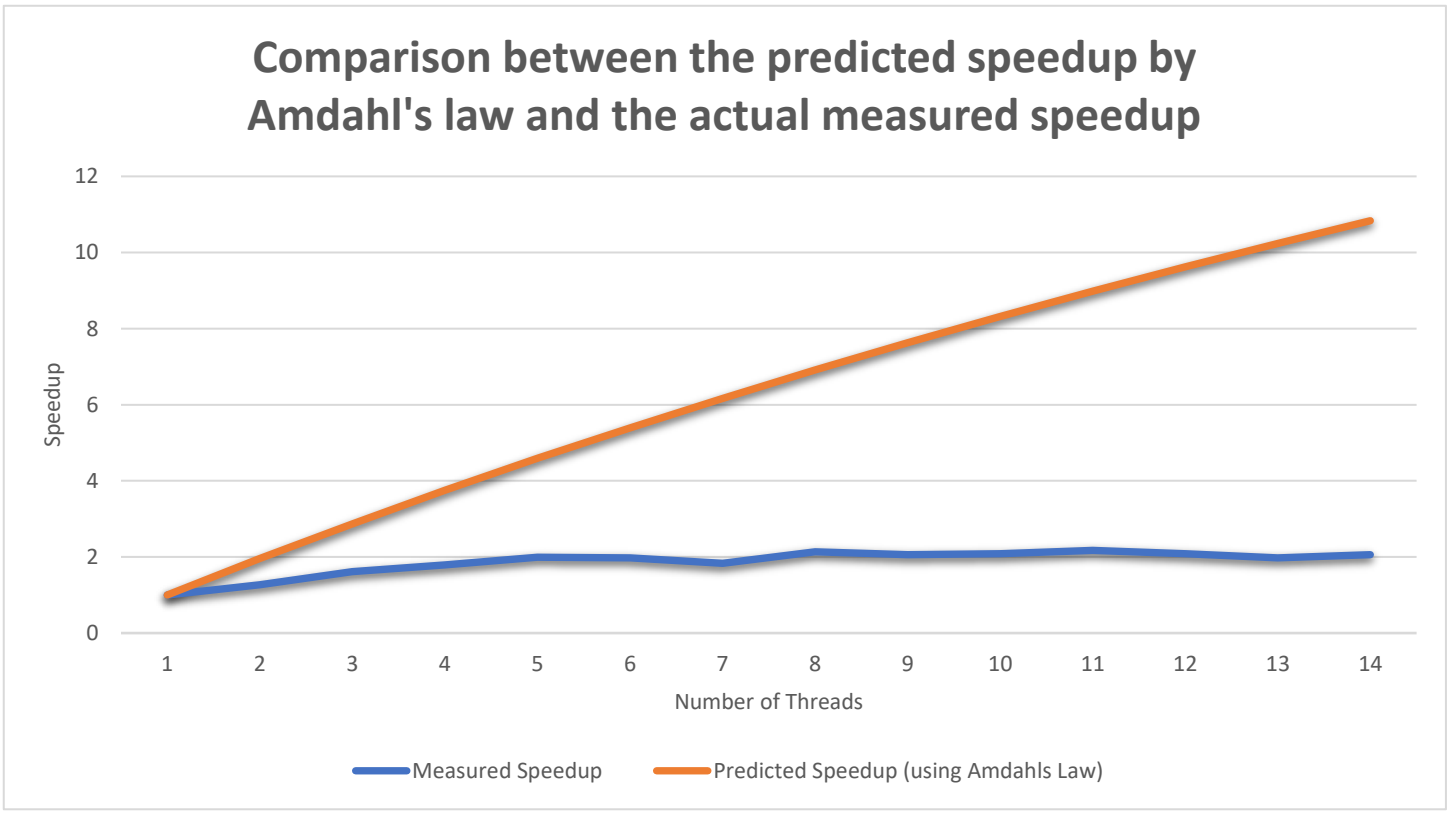

The effect that number of threads has on parallel execution time for calculating PageRank for orkut_undir graph
Execution Time (seconds)
180
160
140
120
100
80
60
40
20
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
Number of Threads


The effect that number of threads has on speedup for calculating PageRank for orkut_undir graph
Speedup
2.5
2
1.5
1
0.5
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
Number of Threads

The laptop that was used to record these execution times is a Dell XPS 15 with an Intel i7-9750H CPU with 6 cores and 12 threads.

As shown above, these two line graphs are used to show the effect of increasing the number of threads from 1 to T+2 when executing the PageRank calculations on the orkut_undir graph for the parallel CSC implementation (ICHOOSE). The first graph demonstrates the execution time against the number of threads and the second graph demonstrates the speedup against the number of threads.

As calculated in Question 1 part A, the parallel fraction for the CSC sparse matrix for the edgemap function compared to the sequential part of the code is **0.977525**. This parallel fraction was then used to calculate the predicted speedup of the CSC program from our range of threads 1 – 14 and was plotted on a graph to compare the predicted and measured speedup. This is shown in the graph below.



As shown in the graph above, it is clear that when increasing the number of threads used, the actual measured speedup is much less than the predicted speedup by Amdahl's Law. The speedup line from Amdahl's Law increases at a very high rate and will only start to plateau past 200/300 threads. Whereas, the measured speedup increases at a much lower rate up to a speedup of around 2.1 at thread 8 then plateaus from there.

When increasing the number of threads for parallel execution, the reason that the measured speedup is much less than the predicted speedup (by Amdahl's Law) is the fact that **Amdahl's Law doesn't take into account the realistic concurrent overheads that come with increasing the number of threads.**

Gunther's Law says that there are certain factors that limit the parallelism that we can have eg. lock synchronization, threads having to be idle because of synchronization constraints, and one thread having to complete something before the next one can complete a subsequent action. All of these

issues/overheads that come with adding in synchronization limit parallelism. The main synchronization added to the code for this question is where all the threads are called with the Thread.join() method. This join method makes the main thread wait until all the threads finish executing before the main thread can proceed and hence, this increase in the synchronization overhead decreases the efficiency of the program and hence limits parallelism.

Another key reason as to why the measured speedup is less than the predicted is related to the other overheads that concurrency brings – all the code that was added to the edgemap function of ParallelContextSimple to allow concurrent execution. The code added to this edgemap function is used to partition all the vertices into equal sizes, instantiate the threads with specific parameters, start the threads, and finally join the threads. This is a lot of extra functionality compared to the edgemap function in the ParallelContextSingleThread class – which just calls the matrix edgemap function. Increasing the number of threads also increases the code that has to be run as part of the ParallelContextSimple edgemap method due to more threads needing to be instantiated and more for-loop iterations, for example. Overall this further increase in concurrency overhead (and work) will further decrease the efficiency and limit the parallelism.

Finally, another reason is the fact that reading the actual graph text file takes a significant amount of time and hence the sequential section takes up a significant amount of time. Increasing the number of threads doesn't improve the performance of the sequential section – only the parallel section – so the calculated speedup will seem lower than what is theoretically predicted.

Question 3

When implementing the disjoint set algorithm, there are a couple of design decisions that needed to be made. The two main decisions relate to the **union** method and the **find** method.

Union method design choice

For the union method, in theory, implementations such as union by rank (when combined with path compression) and union by size have the best time complexity [O(m, α(m,n)) where α(m,n) is the functional inverse of the Ackermann's function and is nearly constant at large values of m and n], and hence, should have the best theoretical performance. However, in practice, it has been proven that randomized linking/naïve linking works just as well if not better than linking by rank, even though it is theoretically inferior. [Disjoint Set Union with Randomized Linking].

For concurrent disjoint sets, union by rank and union by size cannot be used – as to work concurrently with wait-freedom they would require two compare and set operations to take place atomically together (DCAS), which is not knowingly possible on current CPU architecture yet.

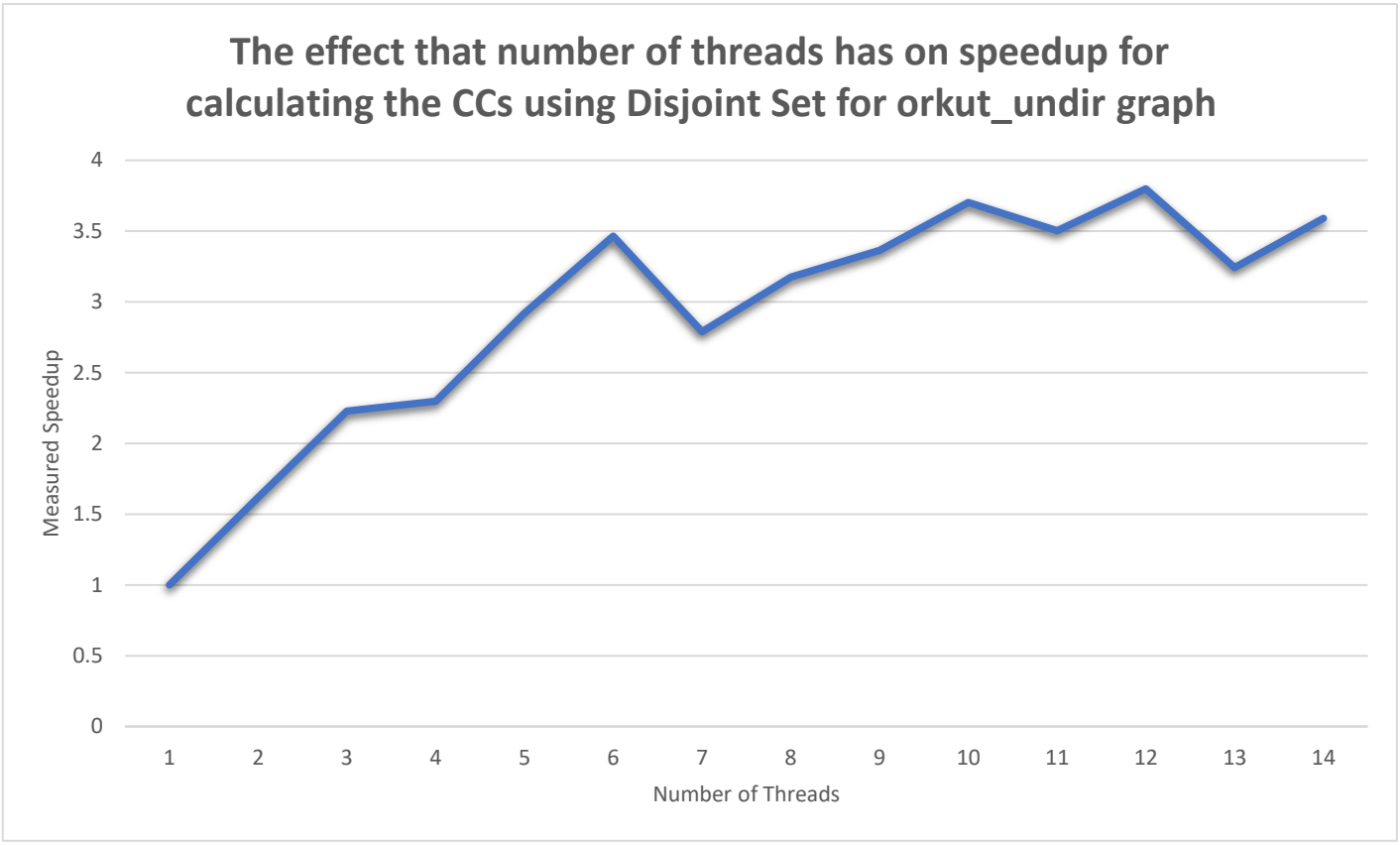
This leaves the randomized linking as the obvious choice for my concurrent implementation of disjoint set due to firstly its superior practical performance and secondly the fact that it can actually be ran concurrently with wait-freedom, unlike the other options. For my randomized linking implementation in my code, I iterated through every vertex and assigned an index to each one based on the iteration counter.

Find method design choice

The find method is used to find and return the root of any given vertex. The most basic implementation works as intended but the performance can be greatly improved through the use of some form of compaction along the way – where on traversing up the tree to find the root, the branch is shortened (by different amounts depending on the compaction method) through the use of a couple of different compaction options. By making the tree more shallow by the use of these compaction methods, it greatly improves the performance as the find method traverses the tree from bottom to top and hence a shallower tree will be more performant than a deeper tree.

There are 3 main forms of compaction: path compression, splitting and halving. Path compression is done using two traversals (recursively) which means it is not suitable for concurrent execution as it causes errors due to this double pass over the path.

This leaves splitting and halving as the two options for forms of compaction in this algorithm. Splitting and halving are similar in design but for concurrent execution it was proven in the 2016 paper by Siddhartha Jayanti and Robert Endre Tarjan that **splitting is superior in the concurrent setting** [A Randomized Concurrent Algorithm for Disjoint Set Union]. This was proven by the fact that two processes doing halving is the same as one process doing splitting, and hence splitting is the superior choice in this concurrent scenario – so I chose to go with splitting due to it being the form of compaction that had the best performance. I stuck with for one-try splitting for this section, however, it was found in that linked paper that two-try splitting is slightly superior due to having tighter bounds. I later experiment with two-try splitting in question 5.



As shown above, the line graph demonstrates that as the number of threads (that are used to calculate the connected components using the disjoint set) is increased, the speedup directly increases too. The speedup begins to plateau at around 10 threads.

This correlation is due to the concurrent aspect of the disjoint set that I've implemented. An increased number of threads means that the iterations of the ranged_edgemap method call can be split up among a greater number of threads executing simultaneously and hence increases the performance/speedup.

Sparse matrix data structure choice

For concurrent execution with the three sparse matrix structures, COO, CSC, and CSR, with the disjoint set class, there were no race conditions found (unlike with PageRank) – therefore, potentially any of the three sparse matrices could be used with the disjoint set class to calculate the connected components.

The COO sparse matrix was the slowest out of the three sparse matrices for its overall calculation – due to its > 50 seconds load time of the graph file, however, COO consistently had a much lower disjoint set total execution time than CSC or CSR (approx 2-3 times faster). This is most likely due to the simpler and more efficient ranged_edgemap function within the COO compared to the more complex nested for loop in the ranged_edgemap function within CSC and CSR classes(less efficient due to more numeric calculations for the nested part). Although COO saves around 0.5 seconds for the actual disjoint set calculations, reading the COO graph file takes over 50 seconds compared to around 20 seconds for CSC and CSR. Storing each individual edge pair when reading the COO graph file would be much less efficient in terms of memory compared to CSR and CSC. Since we are trying

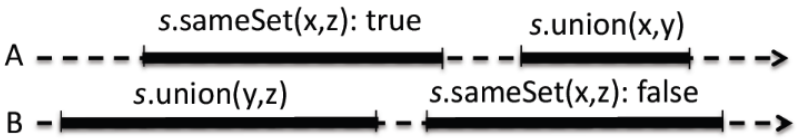
to find the best overall performance, including reading and writing the files, I decided not to use COO for this section as it is overall a lot less efficient than CSC and CSR for this particular task.

Between CSC and CSR, I found that the performance is very similar if not identical. This is different from what we've seen previously where CSR is normally a lot more efficient – which is mainly due to its calculate out degree function used as part of the calculations, which is much more efficient than the CSC and COO equivalents due to the nature of how the CSR sparse matrix data is stored. This is true for both PageRank and Connected Components.

However, the outdegree method of the matrices is not used as part of the disjoint set connected components calculations. This is a crucial fact because, for the disjoint set calculations, the performance obtained from using CSR and CSC is going to be very similar if not the same. Therefore, in my current implementation of the disjoint set class, there is no advantage to using either of these sparse matrices – I chose to keep using CSC in this question for consistency, as I've been using it in the previous questions. However, CSR could have been used instead with these calculations and the performance would be should be nearly identical to that of CSC – depending only on the difference in the shape of the graph (for the graph reading time).

Question 4

1.

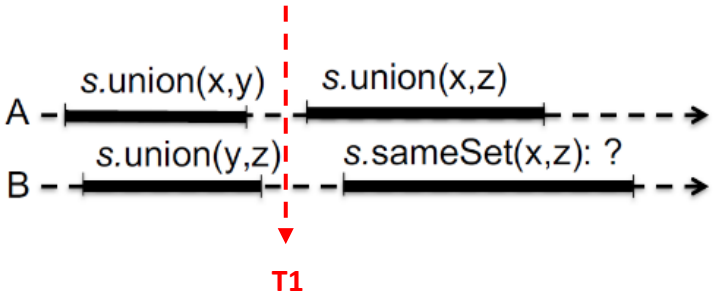


The following execution is **neither linearizable nor sequentially consistent.**

In linearizability, the real time order is respected, so the progress of the threads A and B are dependent on one another. Initially, y and z are linked together by thread B. Later, thread A links x to the y and z set. The reason that this whole execution is not linearizable is due to the s.sameSet(x,z): true statement by Thread A. For sameSet to return true for x and z, they must be in the same disjoint set but sets x and y are not linked until a sequentially later time by the same thread (s.union(x,y)by Thread A). Therefore, this execution is not linearizable.

With sequential consistency, real time order doesn't have to be respected so the progress of the threads A and B are independent of one another. However, we cannot reorder operations done on the same thread. Similar to linearizability, the sameSet(x,z):true call by Thread A causes sequential consistency of this execution to fail because for this call to return true, the s.union(x,y) call (on Thread A) **must have occurred sequentially before** the s.sameSet(x,y):true **call on the same thread** (Thread A). With sequential consistency **we cannot reorder sequential operations done on the same thread** – this execution fails to be sequentially consistent.

2.



For this linearizable execution, s.sameSet(x,z) **can only ever return true.**

In linearizability the real time order is respected, so the progress of the threads A and B are dependent on one another. This means that at point T1, as shown above, both the calls s.union(x,y) and s.union(y,z) must have occurred. These two union method calls link all three of x, y, and z into the same disjoint set when both methods are called. The order in which these two union methods are executed makes no difference to this outcome – the 3 sets will always be joined. Therefore, it is guaranteed that at point T1, sets x, y, and z are all within the same disjoint set.

Since at T1, x, y, and z will always be within the same set in all linearizable scenarios when Thread B calls s.sameSet(x,z) then true will always be returned. When Thread A calls s.union(x,z), it does nothing as x and z are already part of the same set. Therefore, whether s.union(x,z) or s.sameSet(x,z) is called first, it makes no difference as s.sameSet(x,z) will return true in either case.

Question 5

Starting out with the optimisations, there are a lot of possible options and paths to go down. The DisjointSet algorithm is very well optimised already – using one-try splitting to achieve a great performance time.

The main performance issue is the execution time for reading the input files for the graphs as it takes around 20 seconds for okut_undir.csc-csr. This is an extremely long time and compared to the 1-second execution time of DS, the read times need to be the key focus.

Current bottleneck

The Disjoint set calculations take around 0.6 seconds for the orkut_undir graph with 12 threads (either CSR or CSC). Reading the graph files take at least 17 seconds, therefore, the graph file reading should be the key area to focus on to improve performance for now, as it is currently the biggest bottleneck and hence the greatest time reduction will be achieved by optimizing it.

Looking into the different options and taking what I've learned from the advisory session, it seems that the parallel input for the graph reading is going to have the biggest impact on the performance as a theoretical speedup of 12 could be achieved for 12 threads potentially. My current thoughts on how this will be achieved:

- Split the graph text files up into 12 individual partitions in terms of bytes (for 12 threads)
- Iterate over the line until a new line character is found then start there
- Have a partial CSR implementation in each thread with a suitable data structure and size

Parallel Input File Reading

Reading the graph sequentially for CSR, the source and index arrays will be populated as the graph is read by the readFile method. To read the graph file in parallel, the starting positions of each thread could be predetermined then each thread could run for $1/12^{th}$ of the graph file.

The key issue with the current code is the fact that when each thread is reading the source and destination values from the graph file, there needs to be a way to store the source and destination pairs to be used later in the relax method.

The current implementation using source and index arrays is not going to be suitable for multiple threads due to race conditions in writing to the arrays. How are 12 threads meant to atomically update the large arrays without using any forms of locks/synchronization (as they'll add too much of an overhead)?

Partial arrays

The answer is clear that the large src and index arrays are not going to be suitable. Each thread should have its own partial array. Then these 12 arrays should be called individually later on in parallel later in the edgemap method.

Having 12 partial arrays is good in theory, but how should the size of these arrays be predetermined? It is impossible to predict how many edge pairs will be read by each thread (as the number will be different for each thread) without first iterating over the graph file – therefore it is impossible to predetermine the lengths of the partial arrays.

Either an alternative data structure should be used or a flexible form of arrays should be used. Using arraylists would be a good option. Array lists are resized to roughly 3/2 times the current length but these resize operations are performance costly and hence must be kept to a medium. The best partial array option would be to create 12 array lists and initialize the size to slightly more than 1/12 of the total edges to minimize arraylist resizing.

An important thinking point as well is the fact that 12 threads are being created to read the graph file then are being released, then later for ranged_edgemap the 12 threads are being started up again to read these partial arrays. What if the partial arrays could be

A better alternative to partial arrays (for disjoint set)

The reason for having these partial arrays is so that we can use multiple threads to read in the graph file, store the edges in the partial arrays, then later iterate over all these partial arrays in parallel later in edgemap. **What if we could read in the graph file in parallel, then as each edge is read from the graph file, the relax method could be called on that edge directly?** This would mean that no memory-costly and performance-costly partial arrays would be needed, as the relax method calls are all completed as the graph is being read from the file. Performance-wise and complexity-wise, this merging of the functionality in edgemap and readfile to be performed by the 12 threads would be vastly superior to that of partial arrays.

This improvement can only be made for the disjoint set algorithm as it only requires a single pass over the graph. Connected components needs to pass the graph multiple times so the edges need to be stored, hence connected components cannot be used for this implementation. This works out well as the DS algorithm is generally regarded as superior to the CC algorithm.

I implemented the changes so that as each individual edge is read through the graph file, then the relax method is called on that edge, then the next edge is read from the graph file, and so on.

Sequentially, this new code change is a big upgrade in terms of memory and paves the way for the parallel improvements going forward. However, this new change actually decreases the performance slightly of the DS algorithm. After a detailed analysis, this performance decrease is due to the fact that the buffered reading of the graph file is less performant because of the interruptions. Ideally the buffered reader is most efficient when it has a constant read through, but these calls to the relax method on each edge decreases the performance slightly. This is okay, as the program does actually become a lot more efficient in terms of memory/efficiency – the execution time just doesn't do it justice yet due to the nature of the buffered reader.

Improvements on the performance of the sequential program (before making introducing parallel execution)

The program was altered in the previous section to facilitate the graph to be read in parallel – with additional changes. To decide on the best way (performance-wise) to implement changes to make the program parallel – I knew it would make the most sense to optimise the program as much as possible in the sequential setting first, then I would be able to make the best decision as to which way the program should be made parallel. Below is a snippet of the new readfile/edgemap method:

```
for (int i = 0; i < num_vertices; i++) {
    line = rd.readLine();
    String elm[] = line.split(" ");

    for (int j = 1; j < elm.length; j++) {
        relax.relax(i, Integer.parseInt(elm[j]));
    }
}
```

The combined readfile/edgemap method currently takes around 24 seconds so I recorded the times taken for the 3 sections of this method:

rd.readLine()	line.split(“ ”);	Inner for-loop
4.55s	**7.38s**	**12.66s**

As shown above in the table above, there are some significant improvements that could be made for each of the sections of the method. This is for current implementation which reads line by line.

rd.readLine() section – This section utilises BufferedReader’s readLine method to read in each line of the text file fairly efficiently, then returns the line as a string. Under the hood, the readLine method has to read each character in the graph file then returns the line when it reaches a new line character. By itself, the readLine method is fairly efficient but the problem is that it returns the line which needs a lot of further processing on it to be able to extract the sources and destinations. Overall, readLine has a time complexity of O(N), where N is the number of characters in the graph text file. It has a worst case space complexity of O(N), when the graph file could be one line long.

line.split(“ “) section – This section uses the String method split, which splits up a string by a specified character and returns the individual components in the form of an array. Split is a very useful method, but under the hood every character in the string has to be iterated over. Split will add the individual substrings to the return array as it iterates over the string. This is a very costly operation as there is a fair bit of functionality behind split, so it becomes a severe bottleneck to the performance of our program. Time complexity of O(N) and space complexity of O(N) as the worst case scenario would be for a one line graph text file.

Inner for loop section – This section has been altered to call the relax method instead of storing the source index and destinations, which is a big improvement – especially in terms of space complexity and hence improves the runtime. This for loop is used to iterate over the array of components that were returned by the split method. This method does still take up a significantly large amount of

runtime. Firstly, each element is iterated over and then are parsed into Integers (which is another costly operation) and finally the relax method is called (an unavoidable operation so this can be ignored). The call to the relax method makes up a good proportion of the execution time of this section but this section is extremely inefficient as it has to do another iteration over all the components and hence has a time complexity of O(N) and space complexity of O(N) in the worst-case scenario.

Three sections altogether – When put together, these three inefficient sections of O(N) time complexity each form a final time complexity of O(3N). This is still technically O(N), but it doesn't hid the fact that for every character in the graph text file, they could be read over 3 times in the worst case scenario! This clearly proves the inefficiency of this current readfile/edgemap section and hence shows clearly where we can improve on it – read every character in the graph text file once or only slightly above once.

As a summary - the downside of reading files line by line is that every character in the line must be read to form the line, then the line must be split (and hence read again to do so), and then the relax method can finally be called for the edge pair. This current implementation is inefficient – it could be vastly improved on by reading character by character and calling the edges as soon as they are read (removes need for having to split the line after). In the CSR sparse matrix, the source is the first number in the line then all its destinations are the subsequent numbers on the same line.

Text file reader analysis

BufferedReader

I did very detailed research and analysis into figuring out the fastest way to read the orkut graph. I found that the current BufferedReader readLine method is **the fastest prebuilt method for reading text files line by line at 4.6 seconds**, however, using BufferedReader's read method for reading every character is very slow at **9.9 seconds.** This was a very unexpected result, as I assumed that the BufferedReader's read method would be faster than readLine. This is explained by the fact that:

Every read() method call makes an expensive method call.

Every readLine() method call still makes an expensive method call, however, for more bytes at once, so there are fewer calls.

Therefore, I realised I needed to find a fast method for reading character by character.

BufferedInputStream

After experimenting with other Readers, I found the BufferedInputStream to be the most efficient for reading every single character from the graph file at **1.3 seconds.** I had to set the buffer size to a huge number (ie. 1MB) and this is what facilitated the BIS to achieve such a fast read time. This was a superb execution time when trying to read the characters, but I found that when trying to do any sort of operations between reading the characters ended up slowing down the BIS massively due to the fact that its huge buffer potential was not being utilized. This was the same for splitting up BIS onto multiple threads – it is extremely poor for stop-start operations. Hence BIS would be suitable for the sequential scenario of reading every character then storing them straight away, but in this future parallel scenario where there are 12 threads that could be utililsed and the beefy relax

method needs to be called after every few character reads (hence interrupting BIS's optimised big buffer reading) – the BIS implementation is very poorly suited.

Memory mapping – FileChannel and MappedByteBuffer

Memory mapping is a mechanism that maps a portion of a file, or the whole file, on the disk to a range of addresses within an application's address space. This application can then access files on the disk the same way it access dynamic memory. Ie. Instead of the costly process of reading a character from the graph file then returning that character back to the application, we can instead directly store the graph text file in the memory – as if it was an array for example – then we can access it instantly. This is much more efficient than reading from the text file directly – which has a huge bottleneck there in terms of the throughput of characters that can be passed through – and they can only be sped up well with the use of a buffer. Additionally, (as shown with BIS) using a buffered read is inefficient for our stop-start use case. Therefore, in theory, memory mapping is the perfect solution as it in theory solves all the problems that come with the other readers.

I experimented with FileChannel and MappedByteBuffer and eventually got every character in the graph text file reading sequentially in **2.8 seconds** which is very good considering that this is done on one thread and will be split up later.

Now that I could successfully read every character from the graph file sequentially, I got to work on writing an algorithm for calling the relax method as sources and destinations are read from the graph file. I came up with an MVP (minimum viable product) solution which I uploaded to the competition server under **submitSequential2** and obtained an execution time of **around 7 seconds** (with 12 threads). Below are the SparseMatrixCSR methods: readFile and edgemap from that submission.

```java
public void edgemap(Relax relax) {
    try (RandomAccessFile file = new RandomAccessFile(source, "r")) {
        FileChannel fileChannel = file.getChannel();
        MappedByteBuffer buffer = fileChannel.map(
                FileChannel.MapMode.READ_ONLY,
                0,
                fileChannel.size()
        );

        readFile(buffer, relax);
    } catch (Exception e) {
    }
}
```

```java
void readFile(MappedByteBuffer buffer, Relax relax) {
    // skip to start of vertices
    int newLines = 3;
    while (newLines > 0) {
        if (buffer.get() == 10) {
            newLines--;
        }
    }

    // variable declarations
    byte prev = 10;
    int src = 0;
    int dst;
    int i = 0;

    while (buffer.remaining() > 0) {
        if (prev == 10) {
            prev = buffer.get();
            src = 0;
            while (prev != 10 && prev != 32) {
                src = (src * 10) + prev - 48;
                prev = buffer.get();
            }

        } else {
            while (prev != 10) {
                prev = buffer.get();
                dst = 0;
                while (prev != 10 && prev != 32) {
                    dst = (dst * 10) + prev - 48;
                    prev = buffer.get();
                }
                relax.relax(src, dst);
            }
        }
    }
}
```

edgemap

As shown above in edgemap, the file channel is opened for the graph file then I created a buffer equal to the same size of the channel, and set the start index of the buffer to 0. This meant that the 3 lines containing the graph format, vertices number, and edges number would still be in the buffer.

readFile

The readFile method skips over these 3 lines in the first while loop – it doesn't bother recording the vertices or edges number as they are already stored early when the sparse matrix is instantiated.

The main algorithm code basically runs through the buffer until it's empty. A key point that is utilized in this version of the algorithm is the fact that all the graph files finish on a new line. This fact helps simplifies things and makes it easier to break the while loop when the buffer is truly empty, and not before or after.

The main code itself is not the most clean or pretty and has a lot improvements that will be showcased further in the report, but for a first submission, it works correctly and is a good effort. The nested while loops within the outer while loop are not technically needed – but they significantly improve the performance of the program due to the fact that they reduce the number of if statement checks and therefore overall operations (hence improving execution time).

src = (src * 10) + prev – 48 for src and **dst = (dst * 10) + prev – 48** for dst

The source and destination numbers are made up of multiple characters/digits that are read from the graph file. When reading the graph file character by character, it took some research and time to figure out the most optimised way of building up the source and destination integers.

Option 1 – The first option I tried was use strings. I would initialize a source and destination string to "" then concatenate subsequent characters to the ends of those strings by using the + symbol. This way of concatenating the strings was extremely slow. This is because in Java, strings are immutable and hence when two strings are concatenated, a completely new string object must be created and the contents of the previous two strings must be read and copied into the new string. Hence, this costly operation dramatically affected the execution time.

Option 2 – The second option I tried was to use the StringBuilder class. This is a good class commonly used in Java DS&A Leetcode questions so I thought it would do the trick. To use the StringBuilder class, first you must instantiate two objects, then call the append method with each character. Finally to convert the StringBuilder Object to a String, you must call the toString() method. Overall, StringBuilder was much more efficient than concatenating the two strings, however, it was not as quick as I would like it to be for a couple of reasons. The object instantiation is costly and calling the toString() method for every source and destination value is very costly as well as the char array stored in the StringBuilder has to be converted to a String object and returned. Finally this string must be directly converted to an Integer by using Integer.parseInt() – this adds yet more complexity and hence unwanted execution time. I then started looking into better alternatives.

Option 3 – After looking into using the bytes directly for operations, I came up with this little integer operation as shown in my code submission above: **src = (src * 10) + prev – 48** for the source numbers and **dst = (dst * 10) + prev – 48** for the destination numbers. It basically builds the source

and destination number up as the MappedByteBuffer (MBB) traverses over the graph text file. So when a destination number has another digit at the end of it, the previous number is multiplied by then and summed to the new digit. I recognize that this operation is technically not the most efficient operation due to the reasonably large integer calculations going on in it, but it was efficient enough that I was happy with its performance. It could probably be further improved by working with the bytes directly and doing some signing to convert to an integer number – but the performance difference would only be marginal compared to other possible improvements.

Parallel File Channel

After optimizing the sequential file channel code to a respectable time of around **7 seconds** when run with orkut_undir (with 12 threads), I started experimenting with parallel execution with the file channel.

To read this graph text file in parallel, the optimum scenario is to split the file up into 12 almost-equally sized sections, then each thread should run those sections in parallel. In practice, since we only want to read the graph file once (and the numbers in the graph file are made up of a different number of digits) there's no way of determining where the 11 splits should be made in the file for the optimum equal 12 splits. This is one of the key thinking points of this parallel optimization.

When splitting up the graph file, the position in the graph file of the splits could be in the middle of a line, therefore it's essential that there is some way to ensure that the thread boundaries are determined and specified correctly beforehand, or else the graph calculations will be incorrect.

In the common scenario that a split is in the middle of a line, the split line should be extended to the end of the current line. Then the thread should start reading the graph file at the start of the next line.

The **most simple way to accomplish this in theory** is to create 12 buffers that start at each $1/12^{th}$ split in the graph file, then the buffers should have a size set to extend to the very end of the graph file. Ie. this would look like the following:

Thread/Buffer 0: startingPosition(0), size(fileChannelSize)

Thread/Buffer 1: startingPosition(fileChannelSize / 12), size(11 * fileChannelSize / 12)

Thread/Buffer 2: startingPosition(2 * fileChannelSize / 12), size(10 * fileChannelSize / 12)

Thread/Buffer 3: startingPosition(3 * fileChannelSize / 12), size(9 * fileChannelSize / 12)

etc…

This solution would work locally **but would have some practical implications**. We're setting each buffer size to extend to the end of the file channel to cover the scenario where the graph file could only be one line long, or only has one long line at the end. This is an essential case to cover for correctness. The practical implications are the fact that we're going to have to assign 6.5 times the minimum memory needed for the buffers. This would run locally but on the server, it would fail to run due to differences in memory allocation and JVM characteristics.

Problem

This example shows the problem that needs to be solved here. **The buffers must overlap enough so that they can run to the end of the line where the split is specified, but the buffer size must not be initialized to be larger than what it needs to be – or else the already-limited memory will be wasted and hence the server will fail to run.**

Overlap Solution Attempt #1

My first consistently working parallel solution was found in the **submitParallel4** submission and had an average execution time of around 1.5 seconds. I measured the longest line in the orkut graph (261000) and set size of each buffer to be (fileChannelSize / 12) + 261000. This meant that in the scenario that the split was on the longest line in the orkut graph, the buffer would be big enough to allow the split to reach the end of the line.

This solution worked **but was incorrect** due to the fact that this overlap was measured and hardcoded beforehand for the orkut graph. It also meant that the program wouldn't work for smaller graphs such the matj graph, as it is less than 261000 bytes.

Overlap Solution Attempt #2

This solution is found under **submitParallelRevamped** and is the one demonstrated in the "Parallel File Channel" section where you create 12 buffers that start at each $1/12^{th}$ split in the graph file, then the buffers should have a size set to extend to the very end of the graph file.

As explained in the previous section, this solution had practical implications due to the buffer sizes being set to take up 6.5 times the needed amount and hence failures on the server. At this point I was unaware of the buffer memory implications for the server so continued to fail to take this consideration into account.

Overlap Solution Attempt #3

This solution is found under **submitNew** and yet again fails due to my lack of understanding the memory implications of buffers and the server. I set each buffer to be of the maximum size, so I was allocating 12 times the minimum amount.

This solution was more complex than previous ones. At each buffer split, the last source or destination would be recorded then later a buffer would run through the graph file and retrieve all the missed sources and destinations for the possible 12 graph split lines. This extra complexity not only increased the allocated memory but gave a poorer execution time than previous ones.

Overlap Solution Attempt #4 – working correct solution

I finally realised the physical/memory implications of setting larger buffer sizes than needed. I took a step back and realised that the best way to determine the buffer start positions and sizes is to **go through the graph file initially at the splits then find the exact byte where the end of the line should be for each split.** This solution is found under **bufferRangesFindPrior3.0,** as shown below:

In the CSR class, I overloaded the constructor with this version as the original constructor was still be used by the validator. In this constructor, I create a buffer of fileChannelSize and jump to lastSplit + 1/12th fileChannelSize then find the end of the current line, then store it in a positions array. This array is then used later when the buffers are being initialized, for both buffer position and size.

If the graph text file doesn't have very many lines, the if statement checks if the start is greater than the fileChannelSize to ensure that this scenario is covered. For a one line graph, the first thread will cover the whole line then the rest of the threads be idle for the duration. This isn't an ideal scenario but it is covered and will work – it just wont be performant.

I created a separate file channel here then closed it before creating the other file channel for the buffers for the individual threads, to ensure that the buffer memory actually gets released. The memory never seemed to get released for this big buffer when done in the same file channel and hence caused out of memory exceptions when running on the server - as the memory is limited and is near its limit with the individual thread buffers already.

The edgemap method opens up a new file channel for the graph file then it basically creates all the buffers then passes them to the threads, then the threads are started and joined. Since the starting positions of the buffers are predetermined, there is no overlap and hence the buffer sizes remain at a minimum size – preventing any server errors.

```
class ThreadSimple extends Thread {
    private MappedByteBuffer buffer;
    private Relax relax;

    ThreadSimple(MappedByteBuffer buffer, Relax relax) {
        this.buffer = buffer;
        this.relax = relax;
    }

    public void run() {
        byte prev;
        int src;
        int dst;

        while (buffer.remaining() > 0) {
            prev = buffer.get();
            src = 0;

            while (prev > 32) {
                src = (src * 10) + prev - 48;
                prev = buffer.get();
            }
            while (prev != 10) {
                prev = buffer.get();
                dst = 0;
                while (prev > 32) {
                    dst = (dst * 10) + prev - 48;
                    prev = buffer.get();
                }
                relax.relax(src, dst);
            }

        }
    }

}
```

Shown above is the finalized optimised thread code. The thought considerations put into this final optimised version include keeping operations to a minimum ie. no unnecessary if statements or checks that decrease performance.

The key with this algorithm that maximises efficiency is the fact that we only need to check at every new line if the buffer has ended. This means we can keep the buffer.remaining() calls to a minimum and hence improve performance. This call will only be made on the outer while loop whenever a new line character is reached we know that every buffer will finish with a new line character. This is true thanks to predetermining the start and end positions of each buffer in the constructor earlier. This is true for the final buffer too, as the graph files always finish on a newline character.

As I've found with DS&A in the past, having a nested while loop approach rather than if statements maximises performance due to the fact that the nested while loops will continue to iterate continuously without breaking and going to the outer while loop – which would cause additional checks and operations, decreasing performance.

At the start of the outer while loop, the current character will always be a new line so the next character is found and this will be the first digit in our source. The first nested while builds up the source number digit by digit, until it reaches either a new line or a space. The **prev > 32** check is done specifically in this manner rather than **prev != 10 && prev != 32** because prev > 32 is more efficient as it only requires one operation rather than two.

When iterating through the source while loop, once a character is reached that is not a number, if it's a new line then this source has no destinations then the destinations inner while loop will be skipped and the outer while loop will run. If the character reached is a space however, then the destinations inner while loop will run.

The destinations while loop is similar to the sources while loop but it will run continuously for every destination on the line – calling the relax method on each pair – until a new line character is reached (specifying that the source has no more outgoing edges). There is another nested while loop within the destinations while loop to actually put together the destination numbers.

The src and dst lines were discussed earlier as to why they are coded in the way they are.

Overall, this thread run algorithm has been optimised to a very good level and maintains a respectable execution time.

Time complexity and space complexity of this run method (ignoring the time complexity of the relax.relax() method and the space complexity of the buffers).

Time complexity: O(N) – Where N is the number of bytes in the buffer.

Space complexity: O(1) – As no data structures are being initialised and no memory being allocated in this algorithm so the space complexity remains at linear.

Highest performance Union and Find methods

It was found that the "Union-Async and Union-Rem-CAS are the fastest implementations" [Exploring the Design Space of Static and Incremental Graph Connectivity Algorithm] for a range of different datasets, including orkut. It was also found that two-try splitting is slightly faster than one-try splitting and holds better bounds [A Randomized Concurrent Algorithm for Disjoint Set Union]. I implemented both union-async and two-try splitting **submitParallel6** and found them to be slightly faster than previous solutions, at around 1.35 seconds. I didn't use either in my final submission, however, due to the server execution time issues to do with these improvements, ie. it seemed to cause more memory allocation submission failures with these improvements rather than without.

Below is the code shown for these improved methods which I implemented in Java based on their original C++ implementation in the research paper [ConnectIt: A Framework for Static and IncrementalParallel Graph Connectivity Algorithms]:

```java
// two-try find atomic split
public int find(int x) {
    int u = x;
    while (true) {
        // 1
        int v = parent.get(u);
        int w = parent.get(v);
        if (v == w) {   // found root node so return it
            return v;
        } else {
            parent.compareAndSet(u, v, w);
        }

        // 2
        v = parent.get(u);
        w = parent.get(v);
        if (v == w) {   // found root node so return it
            return v;
        } else {
            parent.compareAndSet(u, v, w);
        }

        u = v;
    }

}
```

```java
// unite-async
private void union(int x, int y) {
    int u = x;
    int v = y;
    while (true) {
        u = find(u);
        v = find(v);
        if (u == v) {
            return;
        } else if (u > v && parent.get(u) == u && parent.compareAndSet(u, u, v)) {
            return;
        } else if (v > u && parent.get(v) == v && parent.compareAndSet(v, v, u)) {
            return;
        }
    }
}
```

Why I chose the Disjoint Set Algorithm over Connected Components Algorithm

In general, Disjoint set algorithms are considered to be more efficient than connected components algorithms. First of all, DS are tree-based and CC use a matrix or multiple huge arrays, making them less efficient to work with. Secondly, in DS the average time complexity is constant, even for large number of edges. Whereas CC requires time proportional to the number of edges in the graph, making them very inefficient for large number of edges.

Although DS algorithm on a whole is seen as more efficient than the standard CC algorithm, the thrifty label propagation algorithm is the fastest graph connected components algorithm. I was doubtful if it would be possible to implement the thrifty label propagation algorithm in Java correctly without any external help as it was very complex in terms of memory efficiency. I knew that if I went for the disjoint set algorithm, I would be able to do the parallel graph input/read without storing the edges as well – which would dramatically improve the graph read time which was the main bottleneck, so I went for DS in the end.

The main focus of this question was to improve the graph reading time as it was the main bottleneck – 20 seconds compared to 1 seconds for relax method. Therefore, using DS and the parallel split input method allowed the graph to be read and processed very efficiently. This would not have been possible with CC, as all the edges would have to be stored first to be used on multiple iterations – costly.

Bonus Initial failure – Attempt at a thread-safe CSR with PR

Before choosing between CC and DS, I'll try to optimise CC by making it thread-safe with CSR rather than the already thread-safe CSC, to see if I can get a performance improvement.

A first issue to focus on before deciding on whether to go with CC or DS, would be to attempt at making CSR thread-safe with CC. As discussed in Question 2, for parallel execution, CSR and COO had a race condition scenario for the relax method of both PageRank and Connected Components, therefore CSC was used.

CSR is nearly twice as fast as CSC for sequential executions of PR and CC so to improve the performance of the PR and CC calculations, using CSR instead of CSC would be an excellent first improvement. PageRank would be a good first starting point to make this improvement. Here is the following line that needs to me made atomic to avoid a race condition:

*y[dst] += w * x[src];*

The problem currently lies in that for CSR, multiple threads could have the same dst value when calling this relax function. Since this incrementation line is not atomic, this line essentially reads the value from y then updates it in a separate atomic operation, so another thread could modify the y[dst] value in between the incrementation operations which would lead to erroneous results and hence, we have our race-condition.

Three options to prevent the CSR race condition scenario:

1. Add a critical section around y[dst] += ….. line
2. Make the y[dst] line atomic
3. Use a HashSet data structure to keep track of what dst values the other threads are currently updating then only stall a thread if its dst value is currently being used by another thread
4.

Option 1 -

This option would work as intended but the locks would add too much of an overhead compared to the other 2 options in theory.

Option 2 -

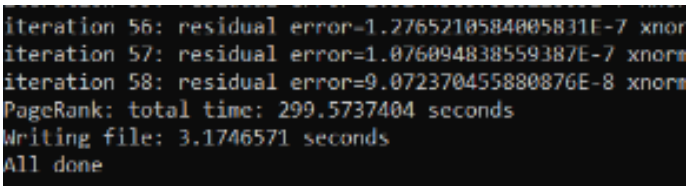
To make this line atomic we would need to convert the y array to an atomic array. The current issue is that there is no built-in AtomicDoubleArray like there is with the AtomicIntegerArray class. I stripped out Google guava's AtomicDoubleArray and will mess around with it to see how if it works as intended. Need to find if there is an atomic operation that will fulfill the desired functionality of the y[dst] line.

In the PageRank class, since it's an incrementation on an already existing position in the array, it looks like the getAndAdd() or addAndGet() methods would work perfectly. For the connected components class it looks like getAndUpdate() or updateAndGet() would be the only possible options but I'm not certain if they would work atomically as intended yet.

To use this new AtomicDoubleArray instead of the standard double array in PageRank, all the use places must be updated.

There were a lot of places in PageRank where the y double array had to be converted to AtomicDoubleArray object operations instead. After doing the conversions, CSR now works in parallel but it is much slower now than before, even slower than it was sequentially (300 seconds). All the atomic operation calls on the AtomicDoubleArray objects could be the cause so I'll try to reduce the number of atomic operation calls by converting most of the PageRank class back to double arrays. The only AtomicDoubleArray objects being used will be inside of the relax method of the SDFKRelax object then the values from the atomic array will be extracted to a normal double array. Hopefully, this improves the performance.

With the AtomicDoubleArray being used for just the relax method then being converted back to the normal array straight after, the performance is still very poor (> 5 seconds per iteration) and hence this atomic operation option is not a suitable option as it greatly decreases the performance of the PageRank calculations. I wasn't expecting such a performance overhead with those atomic operations. Looking deeper, each getAndAdd() operation calls the getAndUpdate() method which, although it is atomic, it performs multiple operations and in a way forms a critical section/locks around that itself. This is similar to how option 1 works – now know how the atomics work better and must rule this option out.



Option 3 -

Option 1 and 2 both created locks under the hood around the CS of the y[dst] section. These locks create a huge overhead on the program.

When executing the PageRank calculations with the multi-thread race condition for CSR, around half of the executions are correct and half are incorrect due to race conditions. This shows that it's quite uncommon for the CSR race condition to occur, so I need to find a way to add synchronization for that specific race condition scenario but not for all the other scenarios (as it would add too much overhead as seen previously). **I want to find a way to lock/suspend the current thread if another thread is currently performing the y[dst] line with the same value for dst as the current thread.**

3a

My first idea was to store the thread dst values in a hashset then when they're doing the y[dst] line – remove from the hashset after completing the incrementation. Then I'd have a while loop on the other thread. I then later realized that hashsets are not thread safe naturally. I realised I needed a better way of determining whether one thread was currently using that dst value. This led to using the index of an array to store the value instead.

3b

An array could be used to flag whether a certain dst value is in use by accessing the array at that index and then have waiting threads in a while loop. This implementation is flawed in that the synchronization of multiple waiting threads needs to be thought out carefully.

3c

By using an array of semaphores, the synchronization issue just discussed could be solved. Having a semaphore at every position in the array (set to 1 initially) for every vertex in the graph could be a janky but possible working solution?

```
*/
public void relax(int src, int dst) {
    double w = d / (double) outdeg[src];
    semaphores[dst].acquireUninterruptibly();
    y[dst] += w * x[src];
    semaphores[dst].release();
}
```

The code is shown in the screenshot above. This code fixed the race condition for the csr calculations but significantly worsened the performance – taking over 10 seconds per iteration, as shown below.

```
iteration 1: residual error=0.5792804349544183 xnorm=1.0 time=10.4164211
iteration 2: residual error=0.14965385477965856 xnorm=1.0 time=9.8919744
iteration 3: residual error=0.048276915070260724 xnorm=1.0 time=10.4635642
iteration 4: residual error=0.024286532788768994 xnorm=1.0 time=10.144250600000001
iteration 5: residual error=0.013925896960314774 xnorm=1.0 time=11.2307549
iteration 6: residual error=0.008717308842954134 xnorm=1.0 time=11.7220996
iteration 7: residual error=0.00572317665701188 xnorm=1.0 time=10.471035200000001
iteration 8: residual error=0.0038862042694218853 xnorm=1.0 time=10.192914400000001
iteration 9: residual error=0.0027082704976842554 xnorm=1.0 time=9.663317600000001
```

The semaphore solution definitely must be ruled out due to these massive overheads brought in as a consequence. I have not been successful so far in trying to fix the race condition of CSR with PR and CC calculations while maintaining a greater performance compared to CSC, so will move on to other improvements for the time being.