QUEEN'S UNIVERSITY BELFAST

Concurrent Assignment 3

By Dean Logan

(Student ID: 40294254)

Contents

I declare that this report does not contain the use of generative AI tools.

1. Analysis (using PageRank)

Part 1

(Note: the below values were collected from my own PC not from domjudge)

The first step in this process was to measure the duration of edgemap, this worked out to be 41.2790999. This value was calculated by adding the time for edgemap for each of the 58 iterations.

The duration of the program was calculated to be 79.1362417.

To calculate Amdahl's law, the fraction of the program that can be parallelized. This value will be denoted as p.

p = 41.2790999 / 79.1362417 = 0.5216206761054714

Amdahl's Law is the following where N is the number of threads:

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

Therefore, for speedup for 4 threads is:

Speedup = 1 / (1-0.5216206761054714) + (0.5216206761054714 / 4)

Speedup = 1 / 0.4783793238945286 + 0.1304051690263679

Speedup = 1 / 0.6087844929208965

Speedup = 1.642617398485439

Therefore, for speedup for 1000 threads is:

Speedup = 1 / (1-0.5216206761054714) + (0.5216206761054714 / 1000)

Speedup = 1 / 0.4783793238945286 + 5.216206761054714e-4

Speedup = 1 / 0.4789009445706341

Speedup = 2.088114486590886

Part 2

In CSR and COO representations, there are potential race conditions when updating the values associated with source and destination vertices in the relax method.

In these representations the updates to y[dst] in the relax method might cause race conditions because multiple threads can be updating the same y[dst] simultaneously. This may cause one of the threads to overwrite data from the other threads value, in other words this is an example of the lost update problem.

Part 3

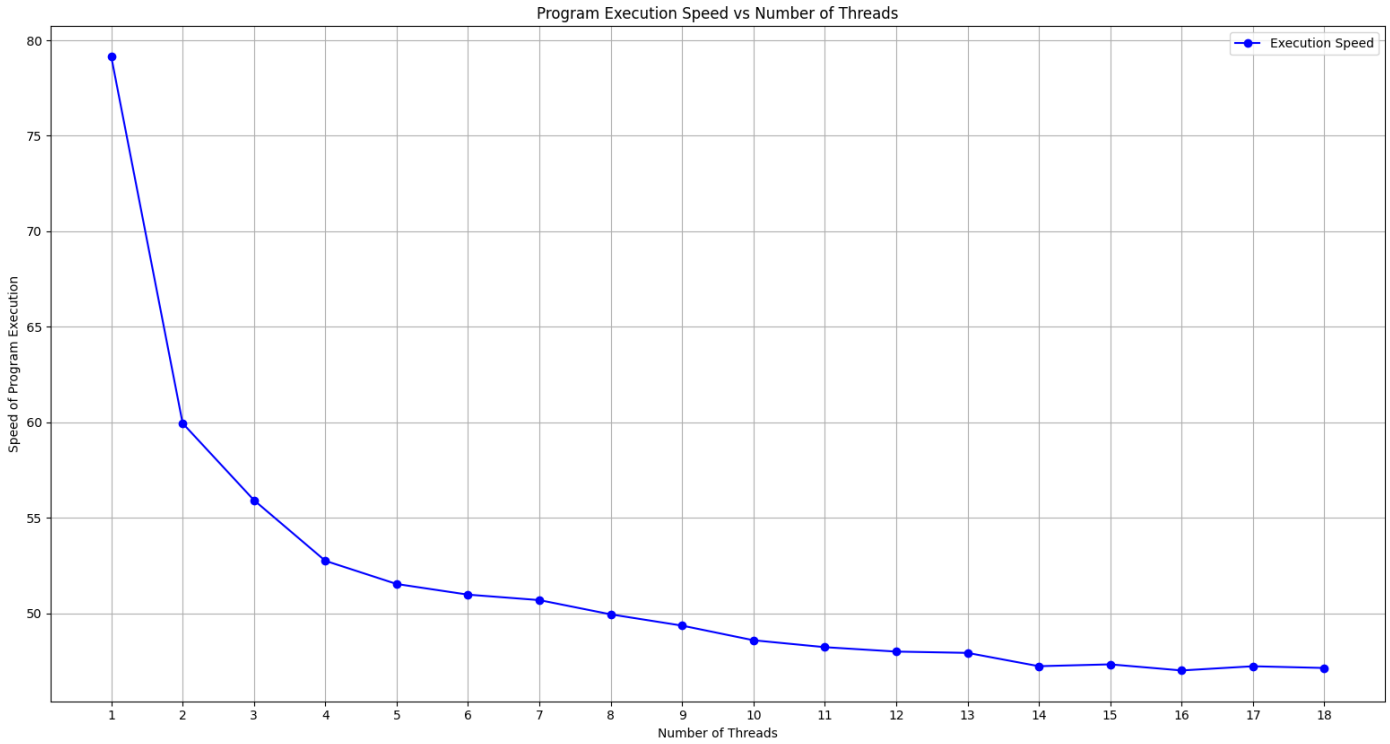
I would use the CSC representation for a concurrent program. The reason for this decision can come in part to the reasons given in question 1 part B

As mentioned, CSC is less prone to race conditions due to the fact that y in CSC is sequential making it more predictable for the hardware cache, compared to the random access pattern in CSR and COO making it less predictable and allowing for the potential for more race conditions.

Even though CSR was faster than CSC which was discovered within the first assignment, this means that CSC has a better opportunity to speed up when utilizing concurrent programming.

2. Parallel Edgemap (PageRank and Connected Components)

CPU: AMD Ryzen 5800X, 8 Core, 16 threads.



Below is a table of the threads and the execution times record that have been plotted on the chart above.

Threads	**Execution Time**
1	79.1362417
2	59.9532814
3	55.9259617
4	52.7513062
5	51.5374887
6	50.9806836
7	50.69486
8	49.9483402
9	49.359525
10	48.5931538
11	48.2308547
12	48.000364
13	47.932478
14	47.2351162
15	47.3311874
16	47.012477
17	47.2318632
18	47.142078836

Actual speedup with 4 threads = 79.1362417 / 52.7513062 = 1.500175965310978

The actual speedup of 1.5 for 4 threads is slightly less than the predicted speedup of 1.64 from Amdahl's Law. The result indicates that the parallelization is providing some speedup, but it may not be fully realizing the theoretical maximum predicted by Amdahl's Law.

Amdahl's Law assumes idealized parallelization and may not fully capture the intricacies of the hardware, such as cache effects, memory bandwidth limitations, or thread scheduling overhead. Gunther's Law.

This could be due to several reasons, such as overhead introduced by parallelization, load imbalance among threads, or simply due to the run-to-run variants in execution speeds.

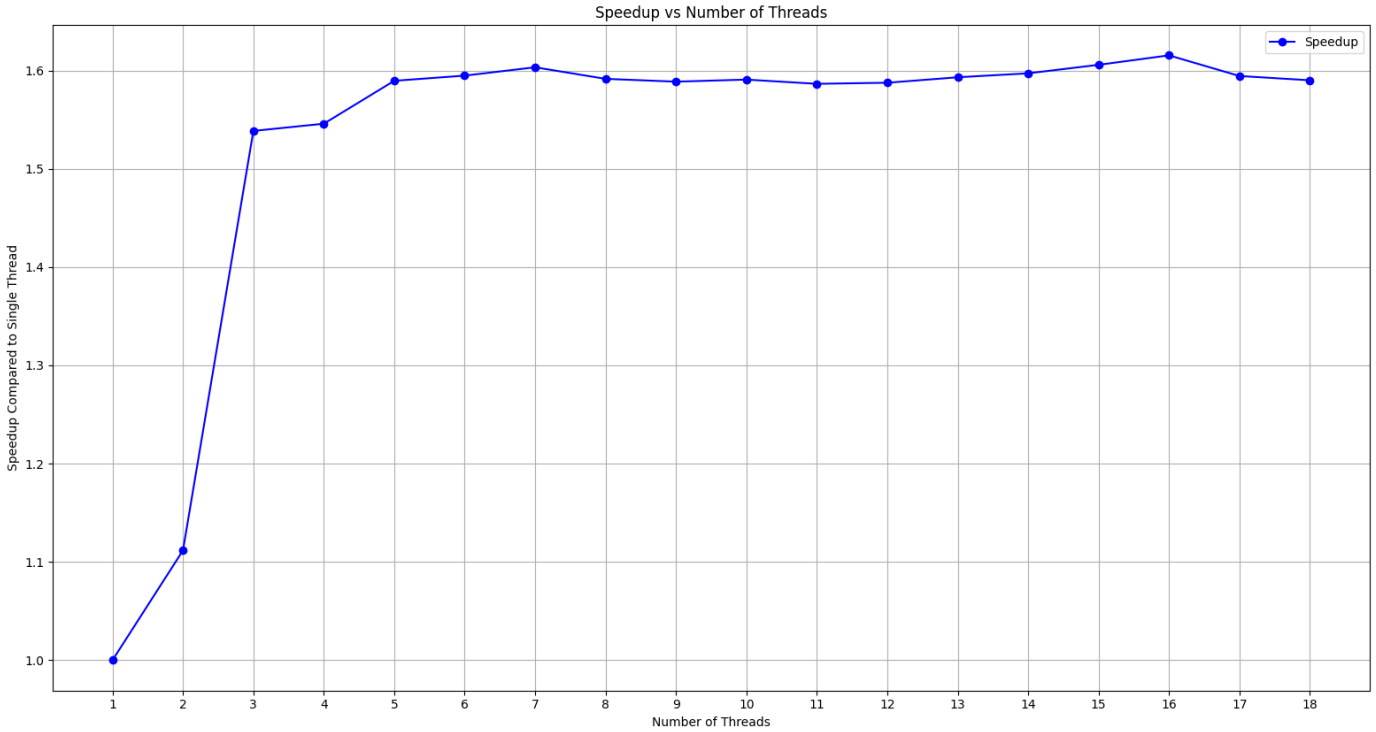
There is a clear diminishing return as the number of threads increases. This can be due to the factors listed when discussing how the speedup observed is not quite the same as Amdahl's Law predicted.

3. Disjoint Set Algorithm

Code for DSSCRelax:

```
public class DisjointSetCC {
    private static class DSCCRelax implements Relax {
        DSCCRelax(AtomicIntegerArray parent_) {
            parents = new AtomicIntegerArray(parent_.length());

            for (int i = 0; i < parent_.length(); i++) {
                parents.set(i, i);
            }
        }

        public void relax(int src, int dst) {
            union(src, dst);
        }

        public int find(int x)  {
            while (x != parents.get(x)) {
                int next = parents.get(x);
                parents.compareAndSet(x, x, parents.get(next));
                x = next;
            }
            return x;
        }



        private boolean sameSet(int x, int y) {
            return find(x) == find(y);
        }

        private boolean union(int x, int y) {
            while (true) {
                int u = find(x);
                int v = find(y);

                if (parents.get(u) < parents.get(v)) {
                    if (parents.compareAndSet(u, u, v)) {
                        return false;
                    }
                } else if (u == v) {
                    return true;
                } else if (parents.compareAndSet(v, v, u)) {
                    return false;
                }
```

```
            }
        }

        // Variable declarations
        private AtomicIntegerArray parents;
    }
```



Threads	Processing Time (seconds)	Speedup Compared To Single Thread Performance (seconds)
1	14.6703652	1.0
2	13.2025944	1.11164085341
3	9.54236275	1.53851306439
4	9.4785429	1.54583947847
5	9.2574753	1.58951170882
6	9.2234761	1.59477125746
7	9.160707	1.603265787
8	9.218437	1.59148241426
9	9.2337974	1.58863728372
10	9.2109057	1.59072597362
11	9.2453078	1.58643214589
12	9.236831	1.58752476379
13	9.1952057	1.59309084299
14	9.1708166	1.59710020998
15	9.1336068	1.60575904003
16	9.09682624	1.61538554266
17	9.1086117	1.59439348274
18	9.1332665	1.59004056774

The speed up in the above table was calculated by dividing the single thread performance with the processing time achieved using the given number of threads.

The graph and table show the speedup of the parallel code. It is important to note the considerable speed up that's achieved between running on 1 to 3 threads, after this the speedup levels off and the variants can be contributed to expected time variants between runs. This suggests that the code does not benefit from further parallelization from 3 threads onwards.

Below is a discussion on the compression used and some discussion of other things to note about the disjoint set code.

No path compression can lead to longer paths in the tree-like structure of the disjoint sets, longer paths can result in higher contention for shared resources, leading to performance problems.

Path splitting is a compromise between no path compression and halving compression. Path splitting can reduce contention compared to no path compression, but it may still have some level of contention.

Path halving involves fully compressing paths during the find operation, making each element on the path point directly to the root. In a concurrent setting, path halving can significantly reduce contention and improve performance. Shorter paths mean that threads are less likely to conflict when traversing the paths concurrently.

For the reasons listed above path halving was picked to implement into the find method as it is the best suited for concurrency.

Another design decisions of interest is the use of compareAndSet within the union method which performs a union operation between two sets, compareAndSet is used to ensure atomicity during the update of the parent pointers. This combined with the use of AtomicIntegerArray helps the code to be "thread-safe" which ensures that multiple threads can safely access and modify the data.

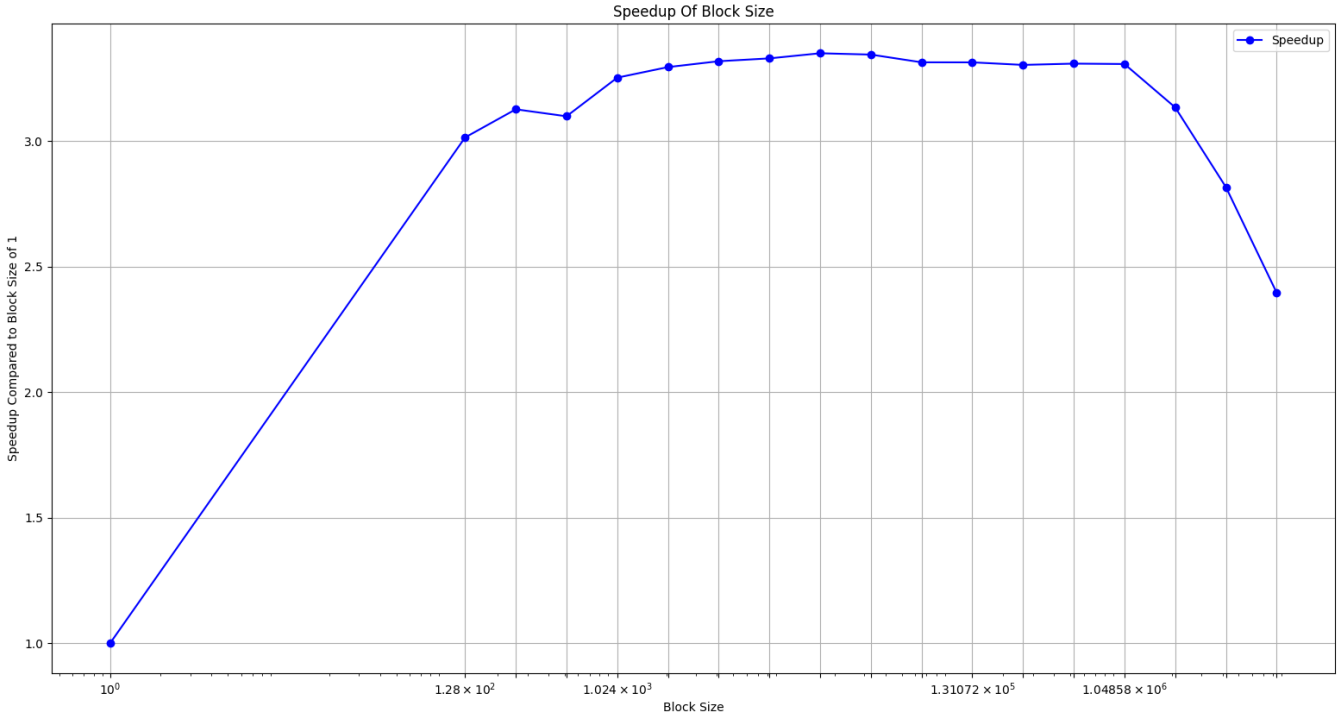
4. Pipeline Operation (Disjoint Set)

When first creating the algorithm the file format chosen was CSC however the algorithm does work with CSR as well.

Both the CSC and CSR file formats were chosen due to their advantages in sequential access patterns and efficient representation of sparse graphs. The COO format is not suitable for this implementation. COO represents each edge as a separate line in the file, meaning that the producer thread must access the file more frequently slowing it down. This inefficiency becomes particularly pronounced for large graphs, impacting the overall performance of the algorithm.

As each line only contains a single edge it means the producer thread will quickly become a bottleneck, not giving the consumer thread data to process fast enough. CSR and CSC contain multiple edges on each line meaning less I/O operations are needed to access the file reducing produces bottleneck.

Below is a chart showing the speedup when changing the block size (note: the buffer size remains the same during these runs at the max integer size of 2147483647, on the CSC orkut graph):



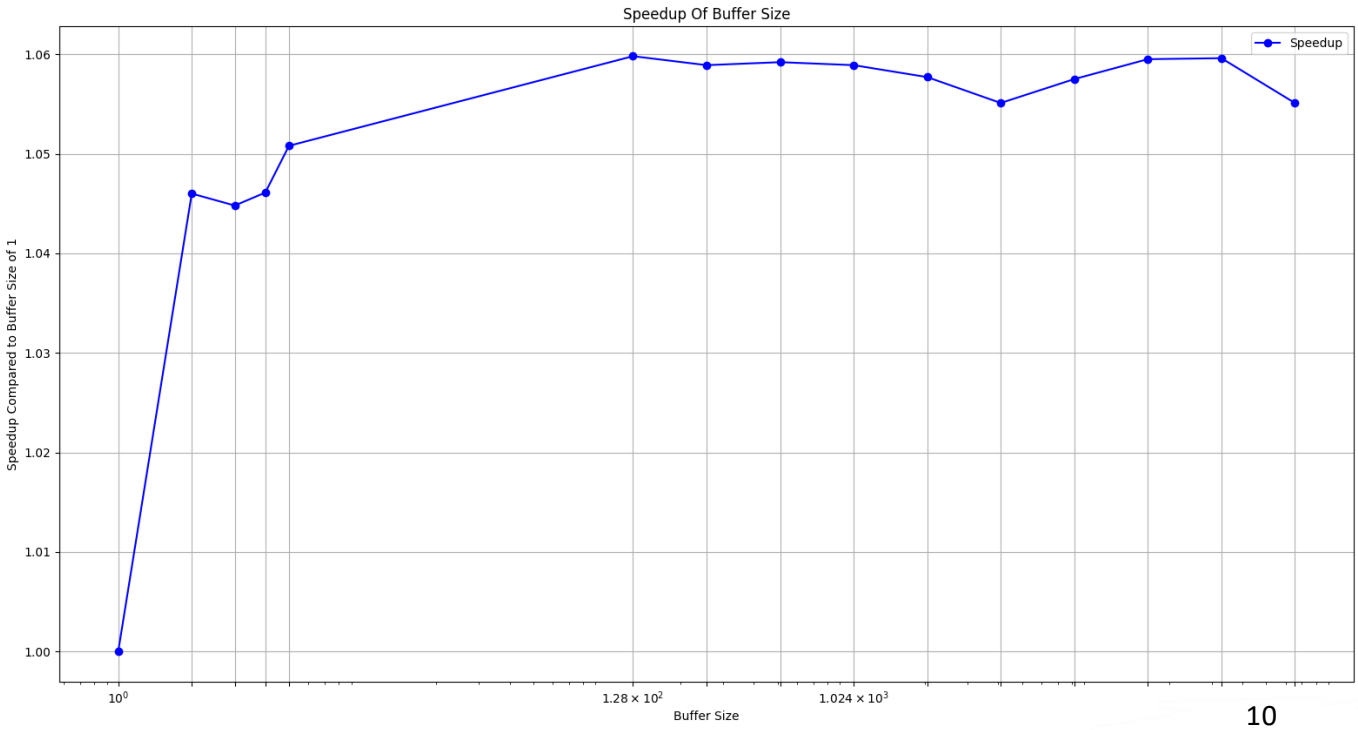
(Note: the above chart is using log scale)

Values shown on the above chart are shown below:

Block Size	Execution Time (seconds)	Speedup Compared To Block Size 1 (seconds)
1	44.0663816	1
128	14.6230084	3.0146416
256	14.0730901	3.1267432
512	14.1942411	3.0989488
1024	13.5565326	3.2528069
2048	13.3722779	3.2948543
4096	13.3155204	3.3182224
8192	13.2820387	3.3295749
16384	13.2353412	3.3499498
32768	13.25509170	3.3446806
65536	13.3373671	3.3139201
131072	13.3380625	3.3137614
262144	13.3762035	3.3035717
524288	13.35910790	3.3087008
1048576	13.3650236	3.3070993
2097152	14.0544914	3.1342808
4194304	15.6396729	2.8168133
8388608	18.4205845	2.3965318

The above table and graph shows that when the block size increases the speed of the algorithm improves. However, after a certain point there is diminishing returns. This can be attributed to the program becoming more sequential as the consumer thread is not completing the processing for all the edges within the block as fast as the producer thread is adding new blocks to the buffer. The "sweet spot" appears to be around a block size of 16384.

Below is a chart showing the speedup when changing the buffer size (note: the block size remains the same during these runs at 128, on the CSC orkut graph):



Values shown on the above chart are shown below:

Buffer Size	Execution Time (seconds)	Speedup Compared To Buffer Size 1 (seconds)
1	15.4273168	1
2	14.7454192	1.0460
3	14.7634755	1.0448
4	14.7443694	1.0461
5	14.6812155	1.0508
128	14.5762844	1.0598
256	14.5640506	1.0589
512	14.5853761	1.0592
1024	14.6065530	1.0589
2048	14.5505093	1.0577
4096	14.6611823	1.0551
8192	14.6140866	1.0575
16384	14.5614325	1.0595
32768	14.5604498	1.0596
65536	14.6607668	1.0551

Whenever you look at the values from within the table that are displayed on the graph you will notices that the there isn't much of a variant within the speedup values (around 0.01 seconds) this suggests that the buffer size doesn't have too much of an affect. It is important to note that even though the variants are small when the buffer size was smaller there was a relatively consistent performance benefit, this could be due to the time being saved in allocating the memory space for the additional elements within the buffer.

As the timing is consistent for anything over a buffer size of 1 it suggests that when running the algorithm on this PC (AMD Ryzen 5800X, 8 Core, 16 threads with 32GB RAM) the queue will only contain a single value on it before the consumer thread processes one element freeing up more space. This suggests that the bottle neck of the algorithm is within the producer thread as the consumer thread can process data much faster, my current theory for this is that the reading of the file is the bottleneck of this algorithm. If reading of the file was faster or if the block size was larger (making the consumer thread slow down as it must process more data before moving on) then a larger difference within the speedup when increasing the buffer size, then potentially a similar diagram to block size would reveal itself.

5. Linearizability and JMM

Part 1

It is assumed that at the start of the execution that none of these values are connected. As the union in A can happen before the sameSet in B meaning that this will be true. Then s.union(u,v) in B has no effect on s.sameSet(u,x) in A meaning it will still return false. Leaving the final operation in B to happen allowing the code to be linearizable and sequentially consistent.

A – s.union(x,y)

B – s.sameSet(x,y): true

B – s.union(u,v)

A – s.sameSet(u,x): false

B – s.union(v,y)

Part 2

A – s.union(x,y)

B – s.union(y,z)

A – s.union(x,z)

B – s.sameSet(x,z): true

As seen in the linearized execution above s.sameSet(x,z) would be true, this is because a union operation has just been run on thread B joining these values into the same set.

s.sameSet(x,z) is true because of the first two operations completed which joins x,y and later joins y,z as y is within the same set as x and z, this means that x,z are also within the same set.

Part 3

In a concurrent execution involving union(x, y) with index(x) < index(y) and find(x), the JMM ensures the visibility of parent(y) to the find method through implicit synchronization guarantees. The relevant load operations, such as r := find(x) and s := find(y), happen-before subsequent store operations within the same method, making it sequential consistency. This means that the code was written to attempt to maintain memory consistency, this helps secure visibility and systemic sequencing of operations in concurrent programs.

6. Competition – Connected Components

The algorithm I choose to further fine-tune is DisjointSetCC, this is mainly because it is the algorithm that I have been working on throughout this assignment therefore it is the one that I am most familiar with which will hopefully lead to better attempts in improving performance. This also allows me to implement ideas discovered throughout the work completed within the other questions.

The Attempts section just contains code snippets from each attempt for your convenience. If you prefer to not look at the code snippets within this file that section can be ignored.

All of the code is also available on the Gitlab repo here (https://gitlab2.eeecs.qub.ac.uk/40294254/csc3021-assignment-3) each attempt has its own branch.

After the investigation of question 3 and question 4 it has become clear that the bottleneck in the algorithm is mainly within the process of reading the graph information within the file.

Attempt 1 was to modify the producer / consumer model to a thread pool model. Instead of adding the edges to a buffer this attempt will run edgemap on a different thread for each line from the file, so there's no need for synchronization. In the producer-consumer model, the queue is a shared data structure that needs to be thread-safe, which can add overhead.

In the thread pool model, work is evenly distributed among the threads because each line is a separate task. In the producer-consumer model, if the producer is faster than the consumer, the queue can grow large, which can consume a lot of memory. Thread pooling also has the benefit of easily allowing a thread to begin more work once its execution has finished.

This attempt netted me a time of 7.4114151 with 12 threads on my own PC compared to the 9.09682624 achieved with the same number of threads on the algorithm used in q3.

As mentioned above most of the time is spent in reading the file. To speed this part of the algorithm up, I have decided to attempt to divide the file into different sections based on the number of threads provided.

This means for Attempt 2 I started by creating a function that will read the bytes of a file within a given range. This was done by using RandomAccessFile, first the byte positions for the start and end was calculated for each of the threads to determine the chunk of the file each thread would be reading.

Once each chunk of the file was allocated to its thread a check would be done to ensure that a threads chunk wasn't going to start halfway through a line within the file ensuring that each thread has completed lines to process. Once this is done the thread will process all of the lines within its chunk of the file.

The code that can be seen at Attempt 2 does give the correct answer consistently however there is one major problem, it is significantly slower. On my PC it took around 350 seconds while using 12

threads which is significantly longer to even a sequential execution, and I can't seem to figure out why this is the case.

I have been able to verify that the code works as expected (meaning that the correct answer is still being produced and each thread has been allocated the correct chunk) but the timing seems to be extremely off, I even checked to see if the overhead of calculating the chunks to each thread was causing the timing issue but it worked out to be less than a second, even the creation of each of the RandomAccessFile objects file didn't contribute to such a time increase. After spending a couple days trying to get to the bottom of this with no success, I decided to abandon this attempt and move onto the next one.

Attempt 3 involved a "rework" of the dividing the file into chunks idea. As I know when using a BufferedReader sequentially is faster than the previous RandomAccessFile attempt I decided to try and split up the file and read the chunks using BufferedReader.

As you cannot access specific byte locations using BufferedReader I decided to create a for loop that would just call readLine() to skip all the lines until the start of the threads chunk was reached. The idea is that within the for loop the lines are not being stored in memory and not having to be divided up into an array therefore the time that would normally be taken to do this would be saved.

However, this was not the case, the time turned out to be 7.4090819 on my PC and when compared to the first attempts time of 7.4114151 this is virtually the same. Now it is important to note that this is faster than the times achieved before starting Q6 but dividing the file into chunks this way does not improve time compared to just parallelising the processing of each line using thread pooling.

Attempt 4 was inspired by a bug that I came across during Attempt 2. While implementing Attempt 2, I originally passed the RandomAccessFile object in as a argument to the function that was being run in the threads, this lead each thread to perform the readLine() operation on the single object as Java works on pass by reference, instead of pass by value. This caused a bug whereas the same file object was being used for each "chunk" of the file, each thread would read lines that was not a part of their "chunk".

However, this gave me another idea what if, like the producer / consumer model there is a buffer (or queue) that allows threads to pick data from and process it. This means that the buffer is the file itself and the get operation is now readLine(), then the blocks within the buffer are the individual lines.

This approach netted over a full second of improvement compared to Attempt 1. Attempt 1 time was 7.4114151 Attempt 4 was 6.0441507.

Attempt 5 involved looking within DisjointSetCC to try and find a better file compression method. To do this I created four different find methods each with a different file compression method (no compression, splitting, halving and full compression) and ran the orkurt dataset on each then recorded the time.

My edgemap function remained unchanged from my previous attempt (Attempt 4) when recoding these timings. another thing to note up until this point I have been using path halving. The record times when running on my own PC where (rounded to the nearest whole number):

No path compression = 250 seconds

Path splitting = 7 seconds

Path halving = 6 seconds

Full path compression = 140 seconds

As you can see path halving was the fastest path finding algorithm out of all of them, therefore I have decided to stick with it instead of swapping to one of the other types.

Attempt 6 involved revisiting the strategy covered in Attempt 2 of dividing the file into chunks. After further investigation I discovered the timing issue within Attempt 2 was due to the readLine() operation, it turns out this operation is expensive for BufferedReader.

After some research I discovered FileChannel which allows for mapping a range of bytes of a file to memory acting as a buffer. Now that the data is in bytes instead of using some sort of readLine() method I can check the bytes for /n to denote an end of a line. Once this is done I can then proceed to check for spaces within the line to divide up the vertexes, once it is checked that there is more than one vertex listed on a line, the rest of the line is converted into a String[] split on the remaining spaces to be converted into ints so that relax can be run. This significantly improved performance compared to Attempt 4 (6.0441507), as this attempt gives a time of 3.435678.

Attempt 7 went even further with optimising reading from the file. Instead of converting the bytes into a String then into an String[], then converting each element of that array into an int, I just do a straight conversion of the bytes into an int, this once again improves the time immensely from a 3.435678 in Attempt 6 to 1.32423.

Attempt 8 was where I decided to combine a reworked version producer / consumer model with Attempt 7's approach of dividing up the file into chunks and converting the bytes directly into int's.

This means that there will be multiple producer threads that will identify where a line begins and ends within the bytes which will pass off the actual processing of these lines to other "consumer" threads. My implementation is not how a normal producer / consumer really works as my implementation doesn't use a shared queue/buffer, it uses a thread pool of 2 consumer threads that is paired with each producer thread for faster processing of these lines, freeing up the producer thread

To ensure that the user would still be able to dictate the number of threads to be used from the terminal I checked how many groups of 3 can be made up within this number (1 producer and 2 consumers in a thread pool make up a group) and any remaining threads are ran in the same way as in Attempt 7.

This resulted in an improvement when running on 12 threads for orkurt, Attempt 7 had a time of 1.32423 and Attempt 8 has a time of 0.586455301. However when it came to the rMat graph (the smallest one) this new overhead of creating and managing this thread pool actually slowed down the time, to avoid this from happening there is an if statement that will check the number of edges within the graph and if it does not meet a certain threshold the code will default to the approach taken in Attempt 7.

Attempt 8 has resulted with my fastest submission on domjudge for the COMPETITION question, the results of that submission are below (team23077 submitted to domjudge on 15/12/2023 at 14:57):

Run(s) on the provided sample data

Run 1

Description	
Runtime	2.277 sec
Result	CORRECT

Run 2

Description	
Runtime	2.146 sec
Result	CORRECT

Run 3

Description	
Runtime	2.294 sec
Result	CORRECT

Attempts

All the below code is also available on GitLab -> https://gitlab2.eeecs.qub.ac.uk/40294254/csc3021-assignment-3

There is a branch for each attempt.

The relevant code snippets from each attempt is included below, but as mentioned this is also on GitLab if you would prefer to read it there, this section is only here if you prefer to read the code snippets within this document.

Attempt 1

```
int getNext(BufferedReader rd) throws Exception {
    String line = rd.readLine();
    if (line == null)
        throw new Exception("premature end of file");
    return Integer.parseInt(line);
}

void readFile(BufferedReader rd, Relax relax) throws Exception {
    String line = rd.readLine();
    if (line == null)
        throw new Exception("premature end of file");
    if (!line.equalsIgnoreCase("CSC") && !line.equalsIgnoreCase("CSC-CSR"))
        throw new Exception("file format error -- header");

    num_vertices = getNext(rd);
    num_edges = getNext(rd);

    ExecutorService executor = Executors.newFixedThreadPool(12);
    List<Future<?>> futures = new ArrayList<>();

    for (int i = 0; i < num_vertices; ++i) {
        line = rd.readLine();
        if (line == null)
            throw new Exception("premature end of file");
        String[] elm = line.split(" ");
        assert Integer.parseInt(elm[0]) == i : "Error in CSC file";

        final int index = i;
        Callable<Void> task = () -> {
            for (int j = 1; j < elm.length; ++j) {
                relax.relax(Integer.parseInt(elm[j]), index);
            }
            return null;
        };
```

```
            futures.add(executor.submit(task));
        }

        for (Future<?> future : futures) {
            future.get();
        }

        executor.shutdown();
    }

    @Override
    public void edgemap(Relax relax) {
        try {
            InputStreamReader is = new InputStreamReader(new FileInputStream(file), "UTF-
8");
            BufferedReader rd = new BufferedReader(is);
            readFile(rd, relax);
        } catch (FileNotFoundException e) {
            System.err.println("File not found: " + e);
            return;
        } catch (UnsupportedEncodingException e) {
            System.err.println("Unsupported encoding exception: " + e);
            return;
        } catch (Exception e) {
            e.printStackTrace();
            return;
        }
    }
```

Attempt 2

```
    private static int getFileSize(RandomAccessFile rd) throws IOException {
        return (int) rd.length();
    }

    void readLinesInRange(int start, long end, Relax relax, int threadId) {
        try {
            RandomAccessFile rd = new RandomAccessFile(file, "r");
            rd.seek(start - 1);
            if(rd.readByte() != '\n'){
                rd.readLine();
            }
            long position = start;
            String line = rd.readLine();
            while (position < end && line != null) {
                String[] elm = line.split(" ");
                for (int j = 1; j < elm.length; ++j) {
                    relax.relax(Integer.parseInt(elm[j]), Integer.parseInt(elm[0]));
                }
```

```
                position = rd.getFilePointer();
                line = rd.readLine();
            }
            rd.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void edgemap(Relax relax){

        try {
            RandomAccessFile rd = new RandomAccessFile(file, "r");

            int fileSize = getFileSize(rd);
            int chunkSize = fileSize / numThreads;

            Thread[] threads = new Thread[numThreads];
            rd.readLine();
            rd.readLine();
            rd.readLine();
            int position = (int) rd.getFilePointer();
            rd.close();

            Thread thread = new Thread(() -> readLinesInRange(position, (0 == numThreads -
1) ? fileSize : (0 + 1) * chunkSize, relax,0));
            thread.start();
            threads[0] = thread;

            for (int i = 1; i < numThreads; i++) {
                int start = i * chunkSize + 1;
                long end = (i == numThreads - 1) ? fileSize : (i + 1) * chunkSize;
                thread = new Thread(() -> readLinesInRange(start, end,
relax,(int)Thread.currentThread().getId()));
                thread.start();
                threads[i] = thread;
            }
            // Wait for all threads to finish
            for (Thread threadL : threads) {
                try {
                    threadL.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
    }
```

Attempt 3

```
    void readRangeBuffered(int start, long end, Relax relax, int threadId) {
        try {
            InputStreamReader is = new InputStreamReader(new FileInputStream(file), "UTF-
8");
            BufferedReader rd = new BufferedReader(is);
            for(int i = 0; i < start; i++){
                rd.readLine();
            }

            for(int i = start-3; i < end; i++){
                try{
                    String[] elm = rd.readLine().split(" ");
                    for (int j = 1; j < elm.length; ++j) {
                        relax.relax(Integer.parseInt(elm[j]), i);
                    }
                }
                catch(Exception e){
                    break;
                }
            }
            rd.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    int getPageLength(){
        try {
            InputStreamReader is = new InputStreamReader(new FileInputStream(file), "UTF-
8");
            BufferedReader rd = new BufferedReader(is);
            int length = 0;
            while(rd.readLine() != null){
                length++;
            }
            rd.close();
            return length;
        } catch (IOException e) {
            e.printStackTrace();
            return -1;
        }
    }

    public void edgemap(Relax relax) {
        int totalLines = getPageLength();
```

```
        int linesPerThread = (totalLines + numThreads - 1) / numThreads; // Round up

        List<Thread> threads = new ArrayList<>();

        for (int i = 0; i < numThreads; i++) {
            final int startLine = i * linesPerThread + 3; // Start from line 3
            final int endLine = Math.min((i + 1) * linesPerThread + 2, totalLines); //
Adjust end line accordingly

            Thread thread = new Thread(() -> {
                readRangeBuffered(startLine, endLine, relax, numThreads);
            });

            thread.start();
            threads.add(thread);
        }

        // Wait for all threads to finish
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

Attempt 4

```
    void readFileLines(BufferedReader rd, Relax relax) {
        try{
            String line = rd.readLine();
            while (line != null) {
                String[] elm = line.split(" ");
                for (int j = 1; j < elm.length; ++j) {
                    relax.relax(Integer.parseInt(elm[j]), Integer.parseInt(elm[0]));
                }
                line = rd.readLine();
            }
        }
        catch(IOException e){
            e.printStackTrace();
        }
    }

    public void edgemap(Relax relax) {
        try {
            InputStreamReader is = new InputStreamReader(new FileInputStream(file), "UTF-
8");
```

```
            BufferedReader rd = new BufferedReader(is);
            rd.readLine();
            rd.readLine();
            rd.readLine();
            Thread[] threads = new Thread[numThreads];
            for (int i = 0; i < numThreads; i++) {
                Thread thread = new Thread(() -> {
                    readFileLines(rd, relax);
                });

                thread.start();
                threads[i] = thread;
            }

            // Wait for all threads to finish
            for (Thread thread : threads) {
                try {
                    thread.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        } catch (FileNotFoundException e) {
            System.err.println("File not found: " + e);
            return;
        } catch (UnsupportedEncodingException e) {
            System.err.println("Unsupported encoding exception: " + e);
            return;
        } catch (Exception e) {
            e.printStackTrace();
            return;
        }
    }
```

Attempt 5

```
        public int find(int x) { // full path compression
            int root = x;
            while (root != parents.get(root)) {
                root = parents.get(root);
            }

            while (x != root) {
                int next = parents.get(x);
                parents.compareAndSet(x, x, root);
                x = next;
            }

            return root;
```

```
        }

        public int find(int x)  { // halving
            while (x != parents.get(x)) {
                int next = parents.get(x);
                parents compareAndSet(x, x, parents.get(next));
                x = next;
            }
            return x;
        }

        public int find(int x) { // No compression
            while (x != parents.get(x)) {
                x = parents.get(x);
            }
            return x;
        }

        public int find(int x)  { // Splitting
            while (x != parents.get(x)) {
                int next = parents.get(x);
                parents.compareAndSet(x, x, parents.get(next));
                x = parents.get(x);
            }
            return x;
        }
```

Attempt 6

```
    void readLinesInRange(int start, int end, Relax relax, int threadId) {
        try (FileChannel fc = FileChannel.open(Paths.get(file), StandardOpenOption.READ)) {
            MappedByteBuffer buffer = fc.map(FileChannel.MapMode.READ_ONLY, start, end -
start);
            //System.out.println("thread: "+threadId+" is starting");
            int lineStart = 0;
            for (int i = 0; i < buffer.limit(); i++) {
                if (buffer.get(i) == '\n') {
                    processLine(buffer, lineStart, i, relax, threadId);
                    lineStart = i + 1;
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    void processLine(MappedByteBuffer buffer, int start, int end, Relax relax, int
threadId) {
```

```
        int spaceIndex = -1;
        for (int i = start; i < end; i++) {
            if (buffer.get(i) == ' ') {
                spaceIndex = i;
                break;
            }
        }

        if (spaceIndex != -1) {
            byte[] firstNumberBytes = new byte[spaceIndex - start];
            buffer.position(start);
            buffer.get(firstNumberBytes, 0, spaceIndex - start);
            int firstNumber;
            try{
                firstNumber = Integer.parseInt(new String(firstNumberBytes));
            } catch(Exception e){return;}
            //System.out.println("thread id "+threadId+"["+new
String(firstNumberBytes)+"]");
            buffer.position(spaceIndex);
            byte[] remainingBytes = new byte[end - spaceIndex ];
            buffer.get(remainingBytes, 0, end - spaceIndex );
            String remaining = new String(remainingBytes);
            //System.out.println("remaining: ="+remaining+"]");
            String[] remainingNumbers = remaining.split(" ");
            for (int i = 1; i < remainingNumbers.length; i++) {
                int num = Integer.parseInt(remainingNumbers[i]);
                relax.relax(firstNumber, num);
            }
        }
    }

    public void edgemap(Relax relax){
        try {
            RandomAccessFile rd = new RandomAccessFile(file, "r");

            int fileSize = getFileSize(rd);
            int chunkSize = fileSize / numThreads;

            rd.readLine();
            rd.readLine();
            rd.readLine();
            int position = (int) rd.getFilePointer();
            rd.close();

            List<Thread> threads = new ArrayList<>();

            Thread firstThread = new Thread(() -> readLinesInRange(position, (0 ==
numThreads - 1) ? fileSize : (0 + 1) * chunkSize, relax, 0));
```

```
            firstThread.start();
            threads.add(firstThread);

            for (int i = 1; i < numThreads; i++) {
                int start = i * chunkSize + 1;
                int end = (i == numThreads - 1) ? fileSize : (i + 1) * chunkSize;
                final int threadId = i;
                Thread thread = new Thread(() -> readLinesInRange(start, end, relax,
threadId));
                thread.start();
                threads.add(thread);
            }

            // Wait for all threads to finish
            for (Thread thread : threads) {
                try {
                    thread.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Attempt 7

```
    void readLinesInRange(int threadNum, Relax relax) {
        int start = threadNum * chunkSize + 1;
        int end = (threadNum == numThreads - 1) ? fileSize : (threadNum + 1) * chunkSize;
        MappedByteBuffer buffer;
        try{
            buffer = fc.map(FileChannel.MapMode.READ_ONLY, start, end - start);
        } catch(Exception e){ return; }

        int lineStart = 0;
        for (int i = 0; i < buffer.limit(); i++) {
            if (buffer.get(i) == '\n') {
                processLine(buffer, lineStart, i, relax);
                lineStart = i + 1;
            }
        }
    }

    void processLine(MappedByteBuffer buffer, int start, int end, Relax relax) {
        int spaceIndex = -1;
        for (int i = start; i < end; i++) {
```

```
            if (buffer.get(i) == ' ') {
                spaceIndex = i;
                break;
            }
        }

        if (spaceIndex != -1) {
            int firstNumber;
            try {
                firstNumber = parseIntegerFromBuffer(buffer, start, spaceIndex);
            } catch (Exception e) {
                return;
            }

            int numberStart = spaceIndex + 1;
            for (int i = spaceIndex + 1; i < end; i++) {
                if (buffer.get(i) == ' ') {
                    // System.out.println("firstNumber: "+firstNumber+" secondNumber:
"+parseIntegerFromBuffer(buffer, numberStart, i));
                    relax.relax(firstNumber, parseIntegerFromBuffer(buffer, numberStart,
i));
                    numberStart = i + 1;
                }
            }
            relax.relax(firstNumber, parseIntegerFromBuffer(buffer, numberStart, end)); //
Call relax for the last number in the line
        }
    }

    int parseIntegerFromBuffer(MappedByteBuffer buffer, int start, int end) {
        int result = 0;
        for (int i = start; i < end; i++) {
            result = result * 10 + (buffer.get(i) - '0');
        }
        return result;
    }


    public void edgemap(Relax relax){
        Thread[] threads = new Thread[numThreads];

        for (int i = 0; i < numThreads; i++) {
            final int threadNum = i;
            Thread thread = new Thread(() -> readLinesInRange(threadNum, relax));
            thread.start();
            threads[i] = thread;
        }

        // Wait for all threads to finish
```

```
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

Attempt 8

```
    void readLinesInRangePipelined(int threadNum, Relax relax, int threadPool) {
        ExecutorService executor = Executors.newFixedThreadPool(threadPool);
        int start = threadNum * chunkSize + 1;
        int end = (threadNum == numThreads - 1) ? fileSize : (threadNum + 1) * chunkSize;
        MappedByteBuffer buffer;
        try{
            buffer = fc.map(FileChannel.MapMode.READ_ONLY, start, end - start);
        } catch(Exception e){ return; }

        int lineStart = 0;
        for (int i = 0; i < buffer.limit(); i++) {
            if (buffer.get(i) == '\n') {
                final int lineEnd = i;
                final int lineStartFinal = lineStart;
                executor.submit(() -> processLine(buffer, lineStartFinal, lineEnd, relax));
                lineStart = i + 1;
            }
        }
        executor.shutdown();
        try {
            executor.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    void processLine(ByteBuffer buffer, int start, int end, Relax relax) {
        int spaceIndex = -1;
        for (int i = start; i < end; i++) {
            if (buffer.get(i) == ' ') {
                spaceIndex = i;
                break;
            }
        }

        if (spaceIndex != -1) {
```

```
            int firstNumber;
            try {
                firstNumber = parseIntegerFromBuffer(buffer, start, spaceIndex);
            } catch (Exception e) {
                return;
            }

            int numberStart = spaceIndex + 1;
            for (int i = spaceIndex + 1; i < end; i++) {
                if (buffer.get(i) == ' ') {
                    relax.relax(firstNumber, parseIntegerFromBuffer(buffer, numberStart,
i));
                    numberStart = i + 1;
                }
            }
            relax.relax(firstNumber, parseIntegerFromBuffer(buffer, numberStart, end)); //
Call relax for the last number in the line
        }
    }

    int parseIntegerFromBuffer(ByteBuffer buffer, int start, int end) {
        int result = 0;
        for (int i = start; i < end; i++) {
            result = result * 10 + (buffer.get(i) - '0');
        }
        return result;
    }

    public void edgemap(Relax relax){
        Thread[] threads;
        if(numThreads > 3 && num_edges > 1000000){
            int pairsOfThree = (int) Math.floor(numThreads / 3);
            int remainder = numThreads % 3;
            threads = new Thread[pairsOfThree + remainder];
            if(remainder != 0){
                Thread thread = new Thread(() -> readLinesInRange(0, relax));
                thread.start();
                threads[0] = thread;
            }
            for (int i = remainder; i < pairsOfThree+remainder; i++) {
                final int threadNum = i;
                Thread thread = new Thread(() -> readLinesInRangePipelined(threadNum,
relax, 2));
                thread.start();
                threads[i] = thread;
            }
        }
        else{
```

```
            threads = new Thread[numThreads];
            for (int i = 0; i < numThreads; i++) {
                final int threadNum = i;
                Thread thread = new Thread(() -> readLinesInRange(threadNum, relax));
                thread.start();
                threads[i] = thread;
            }
        }


        // Wait for all threads to finish
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    void readLinesInRange(int threadNum, Relax relax) {
        int start = threadNum * chunkSize + 1;
        int end = (threadNum == numThreads - 1) ? fileSize : (threadNum + 1) * chunkSize;
        MappedByteBuffer buffer;
        try{
            buffer = fc.map(FileChannel.MapMode.READ_ONLY, start, end - start);
        } catch(Exception e){ return; }

        int lineStart = 0;
        for (int i = 0; i < buffer.limit(); i++) {
            if (buffer.get(i) == '\n') {
                processLine(buffer, lineStart, i, relax);
                lineStart = i + 1;
            }
        }
    }
```