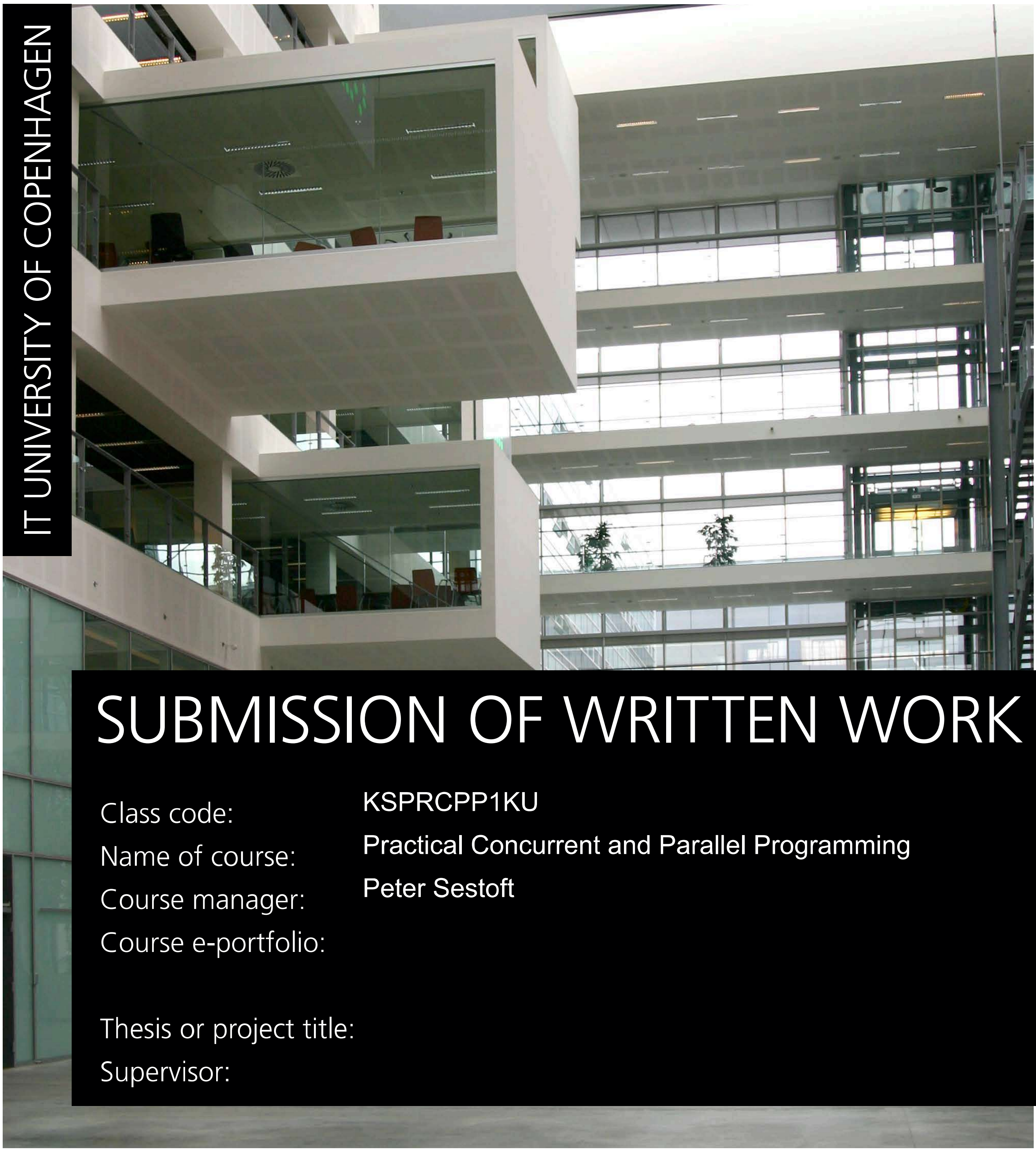

SUBMISSION OF WRITTEN WORK

Class code:	KSPRCPP1KU
Name of course:	Practical Concurrent and Parallel Programming
Course manager:	Peter Sestoft
Course e-portfolio:	
Thesis or project title:	
Supervisor:	

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Anders Wind Steffensen	10/02-1993	awis@itu.dk
2.			@itu.dk
3.			@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk

Practical Concurrent and Parallel Programming - Autumn 2016

Anders Wind Steffensen

January 10-11th 2017

I hereby declare that I have answered these exam questions myself without any outside help.
Anders Wind Steffensen 11/01-2017

Contents

1 Question 1

1.1)

Here the running time and iterations of `KMeans1` can be seen

```
# OS:    Windows 10; 10.0; amd64
# JVM:   Oracle Corporation; 1.8.0_101
# CPU:   Intel64 Family 6 Model 94 Stepping 3, GenuineIntel; 8 "cores"
# Date: 2017-01-10T09:30:50+0100
...
Used 108 iterations
class kmean.KMeans1  Real time:     5.313
...
Used 108 iterations
class kmean.KMeans1  Real time:     5.453
...
Used 108 iterations
class kmean.KMeans1  Real time:     5.011
```

1.2)

In figure 1 the assignment step of `KMeans1P` can be seen. To split the points into eight parts; start and ending indexes are calculated for each task, such that they do not work on the same elements of `points`. To access clusters from inside the lambda, I needed to make a local final reference to `clusters` – which is fine since the reference is not changed in the assignment step. Inside the `Callable<Void>`(which is in the form of a lambda) the code is almost identical to the original code, but of course only iterating over the points of the task.

This approach allows the multiple threads to acquire tasks and do the work in parallel, since each Point is independent from one another. Most of the work done on Clusters are read only and therefore pose no threat-safe problems, but there is one method which modifies shared state which we will cover in question 1.4.

1.3)

In figure 2 the implementation of the update step of `KMeans1P` can be seen. Just like the assignment step, I make an effectively final reference to `clusters` such that it is accessible in the `Callable<Cluster>` lambda. The logic is the same as in the original code, but each task is responsible for a cluster. After creating a lambda it is submitted to the executor service, which in the end iterates over all its `Futures`, and adding the results to the `newClusters` collection.

It should be noted that I changed the `converged` variable to be of `AtomicBoolean` type since I first of all needed it to be final to access it inside the task. Furthermore since it had to

```
final AtomicBoolean converged = new AtomicBoolean();
ExecutorService executorService = newWorkStealingPool();
while (!converged.get()) {
{ // Assignment step: put each point in exactly one cluster
    List<Callable<Void>> tasks = new ArrayList<>();
     // notice that taskCount corrosponds to "P" in the exam assignment.
    final int taskCount = 8, perTask = points.length / taskCount;
    for (int t = 0; t< taskCount; t++) {
        final int from = perTask * t,
                  to = (t+1 == taskCount) ? points.length : perTask * (t+1);
        final Cluster[] finalClusters = clusters; // effective final while we are
            ↪ working in parallel
        tasks.add(() -> {
            for(int i = from; i<to; i++) {
                final Point p = points[i];
                Cluster best = null;
                for (Cluster c : finalClusters)
                    if (best == null || p.sqrDist(c.mean) < p.sqrDist(best.mean))
                        best = c;
                best.add(p);
            }
            return null;
        });
    }
    try {
        List<Future<Void>> futures = executorService.invokeAll(tasks);
        for (Future<?> future : futures)
            future.get();
    } catch (InterruptedException exn) {
        System.out.println("Interrupted: " + exn);
    } catch (ExecutionException e) {
        throw new RuntimeException(e);
    }
}
```

Figure 1: Assignment step of KMeans1P

```
{ // Update step: recompute mean of each cluster
    converged.set(true);
    final ArrayList<Cluster> newClusters = new ArrayList<>();
    final Cluster[] finalClusters1 = clusters; // effective final while working in
        ↪ parallel
    Future[] futures = new Future[clusters.length];
    for (int t = 0; t < clusters.length; t++) {
        final int thisCluster = t;
        futures[t] = (executorService.submit((() -> {
            Cluster c = finalClusters1[thisCluster];
            Cluster result = null;
            Point mean = c.computeMean();
            if (!c.mean.almostEquals(mean))
                converged.set(false);
            if (mean != null)
                result = new Cluster(mean);
            else
                System.out.printf("===> Empty cluster at %s%n", c.mean);
            return result;
        })));
    }
    try {
        for (Future<Cluster> future : futures)
            newClusters.add(future.get());
    } catch (InterruptedException exn) {
        System.out.println("Interrupted: " + exn);
    } catch (ExecutionException e) {
        throw new RuntimeException(e);
    }
    clusters = newClusters.toArray(new Cluster[newClusters.size()]);
}
```

Figure 2: Update step of KMeans1P

be modified from potentially multiple threads – a thread-safe class was needed. This of course creates a synchronization point and slows the implementation down a bit.

1.4)

Only one change was needed to make the `Cluster` class thread-safe for our purpose. That was to add a lock on the `add` method as seen in figure 3. The fields of the class are private and immutable. It is only the add method which modifies the collection `points`. Therefore we have eliminated the unsafe shared mutable data by using atomic access to the data. If we had not added this we could risk that we lost writes(the added points).

I have also added locking on the same lock in the `computeMean` method, since it uses the points collection which could be modified. In our case however it does not matter since these two methods are never called concurrently.

```
private final Object lock = new Object();
public void add(Point p) {
    synchronized (lock) {
        points.add(p);
    }
}
public Point computeMean() {
    double sumx = 0.0, sumy = 0.0;
    int count = 0;
    synchronized (lock) {
        for (Point p : points) {
            sumx += p.x;
            sumy += p.y;
        }
        count = points.size();
    }
    return count == 0 ? null : new Point(sumx/count, sumy/count);
}
```

Figure 3: Thread safety of Cluster in KMeans1P

1.5)

Here the running time and iterations of `KMeans1P` can be seen

```
# OS:   Windows 10; 10.0; amd64
# JVM:  Oracle Corporation; 1.8.0_101
# CPU:  Intel64 Family 6 Model 94 Stepping 3, GenuineIntel; 8 "cores"
# Date: 2017-01-10T19:42:09+0100
...
Used 108 iterations
class kmean.KMeans1P Real time:     1.566
...
Used 108 iterations
class kmean.KMeans1P Real time:     1.470
...
Used 108 iterations
class kmean.KMeans1P Real time:     1.444
```

1.6)

These are the first five means of `KMeans1P`

```
mean = (87,96523339545504, 68,02660923585384)
mean = (10,68258565222153, 47,90993881688516)
mean = (17,94392765121714, 57,92167357590439)
mean = (48,03811144575210, 28,07238725661513)
mean = (78,02580276649086, 68,03938084899141)
```

1.7)

If the locking is removed I get a `NullPointerException` in `computeSum`. This is most likely caused when the collection was modified by `add` since it was not atomic. Therefore the length property might be unrepresenting, due to a race condition. This then creates an error in the later iteration of `points`.

2 Question 2

2.1)

Here the running time and iterations of `KMeans2` can be seen:

```
# OS:    Windows 10; 10.0; amd64
# JVM:   Oracle Corporation; 1.8.0_101
# CPU:   Intel64 Family 6 Model 94 Stepping 3, GenuineIntel; 8 "cores"
# Date: 2017-01-10T19:42:09+0100
...
Used 108 iterations
class kmean.KMeans2  Real time:     5.482
...
Used 108 iterations
class kmean.KMeans2  Real time:     5.452
...
Used 108 iterations
class kmean.KMeans2  Real time:     4.558
```

2.2)

In figure 4 the implementation of the assignment step of `KMeans2P` can be seen. The approach to use tasks is similar to that of `KMeans1P`. I calculate ranges of the `points` array for each task, and each task then iterates over its points. The method inside the lambda is similar to what it was in `KMeans2`. At the end the `invokeAll` method starts all the threads with my executorService(a `WorkStealingPool`) and get the results to ensure that all tasks are done.

2.3)

In figure 5 the implementation of the update step of `KMeans2P` can be seen. The only part which is calculated concurrently using tasks is the updating of means. The approach is just like the assignment step above. The `points` array is divided in equal parts based on the number of tasks(eight) and then each task iterates over its points. These tasks are added to a list of `Callable<Void>` which are then invoked by the `executorService`. The resetting of mean and the calculation of the `converged` variable (based on the mean) is still sequentially computed.

2.4)

In figure 6 the synchronization code can be seen. For the purposes of our exercise, actually only the lock in `addToMean` is required, since it is the only method on `Cluster` which is called from the tasks. A lock is needed since the method modifies shared fields; `mean`, `sumx`, `sumy`, and therefore we need to have atomic access to the data.

```
ExecutorService executorService = newWorkStealingPool();
[...]
{ // Assignment step: put each point in exactly one cluster
List<Callable<Void>> tasks = new ArrayList<>();
for(int t = 0; t<taskCount; t++) { // notice that taskCount corrosponds to "P" in
    ↪ the exam assignment.
    final int from = perTask * t,
              to = (t+1 == taskCount) ? points.length : perTask * (t+1);
    tasks.add(() -> {
        for (int pi=from; pi<to; pi++) {
            Point p = points[pi];
            Cluster best = null;
            for (Cluster c : clusters)
                if (best == null || p.sqrDist(c.mean) < p.sqrDist(best.mean))
                    best = c;
            myCluster[pi] = best;
        }
        return null;
    });
}
try {
    List<Future<Void>> futures = executorService.invokeAll(tasks);
    for (Future<?> future : futures)
        future.get();
} catch (InterruptedException exn) {
    System.out.println("Interrupted: " + exn);
} catch (ExecutionException e) {
    throw new RuntimeException(e);
}
```

Figure 4: Assignment step of KMeans2P

```
{ // Update step: recompute mean of each cluster
    for (Cluster c : clusters)
        c.resetMean();
    List<Callable<Void>> tasks = new ArrayList<>();
    for(int t = 0; t<taskCount; t++) {  // notice that taskCount corrosponds to "P
        ↪ " in the exam assignment.
        final int fromPoints = perTask * t,
                  toPoints = (t+1 == taskCount) ? points.length : perTask * (t+1);
        tasks.add(() -> {
            for(int pi = fromPoints; pi<toPoints; pi++) {
                myCluster[pi].addToMean(points[pi]);
            }
            return null;
        });
    }
    try {
        List<Future<Void>> futures = executorService.invokeAll(tasks);
        for (Future<?> future : futures)
            future.get();
    } catch (InterruptedException exn) {
        System.out.println("Interrupted: " + exn);
    } catch (ExecutionException e) {
        throw new RuntimeException(e);
    }
    converged = true;
    for (Cluster c : clusters)
        converged &= c.computeNewMean();
}
```

Figure 5: Update step of KMeans2P

```
private Object lock = new Object();
public void addToMean(Point p) {
    synchronized (lock) {
        sumx += p.x;
        sumy += p.y;
        count++;
    }
}
// Recompute mean, return true if it stays almost the same, else false
public boolean computeNewMean() {
    Point oldMean;
    synchronized (lock) {
        oldMean = this.mean;
        this.mean = new Point(sumx / count, sumy / count);
    }
    return oldMean.almostEquals(this.mean);
}
public void resetMean() {
    synchronized (lock) {
        sumx = sumy = 0.0;
        count = 0;
    }
}
```

Figure 6: Thread safety of Cluster in KMeans2P

The other methods shown are included to show where the lock also could have been added, since these methods also modifies the shared fields.

2.5)

Here the running time and iterations of `KMeans2P` can be seen:

```
# OS:   Windows 10; 10.0; amd64
# JVM:  Oracle Corporation; 1.8.0_101
# CPU:  Intel64 Family 6 Model 94 Stepping 3, GenuineIntel; 8 "cores"
# Date: 2017-01-10T19:42:09+0100
...
Used 108 iterations
class kmean.KMeans2P Real time:     1.309
...
Used 108 iterations
class kmean.KMeans2P Real time:     1.231
...
Used 108 iterations
class kmean.KMeans2P Real time:     1.269
```

2.6)

These are the first five means of `KMeans2P`

```
mean = (87,96523339545512, 68,02660923585373)
mean = (10,68258565222153, 47,90993881688518)
mean = (17,94392765121715, 57,92167357590444)
mean = (48,03811144575210, 28,07238725661512)
mean = (78,02580276649087, 68,03938084899140)
```

2.7)

If the lock `addToMean` was removed then `KMeans2P` did not succeed in finishing in less than a minute on the 200000 points. The logic of `KMeans2(P)` relies on the mean to be equal to that of the previous iteration, but if added points are "lost" due to race-conditions on `count`, `sumx` and/or `sumy`, then it would be very unlikely that two iterations would compute exactly the same mean - though it of course would be possible.

I tried to decrease the number of points and k to much lower values, 200 and 10 respectively and the program managed to terminate with results albeit probably not correct results, at very varying running times – this further supports my argument.

2.8)

The two approaches are equally tread-safe when `addToMean` is thread-safe. First of all because `addToMean` is commutative (that is calling it first with one point and then another wont change the outcome). Since each element in the arrays are independent from one another, other iterations do not override `myCluster[pi]`'s best. Therefore its safe to directly add the mean in the assignment step.

2.9)

On figure 7 my implementation of `KMeans2Q`. First the update step has been simplified to first calculate the converge and then resetting the mean - one could also put the resetting of the mean before the assignment step. Also note that `myClusters` is gone and that `Cluster best` directly gets the point p added to its mean.

```java
public void findClusters(int[] initialPoints) {
final Cluster[] clusters = GenerateData.initialClusters(points, initialPoints,
    ↪ Cluster::new, Cluster[]::new);
boolean converged = false;
ExecutorService executorService = newWorkStealingPool();
while (!converged) {
    final int taskCount = 8, perTask = points.length / taskCount;
    iterations++;
    { // Assignment step: put each point in exactly one cluster
        List<Callable<Void>> tasks = new ArrayList<>();
        for(int t = 0; t<taskCount; t++) {
            final int from = perTask * t,
                      to = (t+1 == taskCount) ? points.length : perTask * (t+1);
            tasks.add(() -> {
                for (int pi=from; pi<to; pi++) {
                    Point p = points[pi];
                    Cluster best = null;
                    for (Cluster c : clusters)
                        if (best == null || p.sqrDist(c.mean) < p.sqrDist(best.
                            ↪ mean))
                            best = c;
                    best.addToMean(p);
                }
                return null;
            });
        }
        try {
            List<Future<Void>> futures = executorService.invokeAll(tasks);
            for (Future<?> future : futures)
                future.get();
        } catch (Exception exn) {
            throw new RuntimeException(exn);
        }
    }
    { // update step
        converged = true;
        for (Cluster c : clusters)
            converged &= c.computeNewMean();
        for (Cluster c : clusters)
            c.resetMean();
    }
}
this.clusters = clusters;
executorService.shutdown();
}
```

Figure 7: Implementation of KMeans2Q

2.10)

Here the running time and iterations of `KMeans2Q` can be seen:

```
# OS:    Windows 10; 10.0; amd64
# JVM:   Oracle Corporation; 1.8.0_101
# CPU:   Intel64 Family 6 Model 94 Stepping 3, GenuineIntel; 8 "cores"
# Date: 2017-01-10T19:42:09+0100
...
Used 108 iterations
class kmean.KMeans2Q Real time:     2.656
...
Used 108 iterations
class kmean.KMeans2Q Real time:     2.880
...
Used 108 iterations
class kmean.KMeans2Q Real time:     2.187
```

Therefore `KMeans2P` is actually the quickest of the two implementations even though it generates more tasks and reiterates the points. This is probably because the calculation in the assignment step is expensive, and in `KMeans2P` the threads do not wait for synchronization on the cluster. Therefore `KMeans2P` probably works more in parallel on the expensive step and allow the threads to work fully with no waiting time.

```
static class Cluster extends ClusterBase {
    private Point mean;
    private final TxnDouble sumx, sumy;
    private final TxnInteger count;
    public Cluster(Point mean) {
        sumx = newTxnDouble();
        sumy = newTxnDouble();
        count = newTxnInteger();
        this.mean = mean;
    }
    public void addToMean(Point p) {
        atomic (() -> {
            sumx.set(sumx.get() + p.x);
            sumy.set(sumy.get() + p.y);
            count.increment();
        });
    }
    public boolean computeNewMean() {
        Point oldMean = this.mean;
        atomic( () -> {
            this.mean = new Point(sumx.get()/count.get(), sumy.get()/count.get());
        });
        return oldMean.almostEquals(this.mean);
    }
    public void resetMean() {
        atomic( () -> {
            sumx.set(0.0);
            sumy.set(0.0);
            count.set(0);
        });
    }
    @Override
    public Point getMean() { return mean; }
}
```

Figure 8: Thread safety using transaction in KMeans2Stm

3 Question 3

3.1)

In figure 8 my implementation of the transactional memory / multiverse version of `Cluster` can be seen. Most noticeable the `sumx`, `sumy` and `count` fields are now the transactional data types `TxnDouble` and `TxnInteger`, and instead of the synchronized keyword for locking, the atomic keyword is used with a lambda for those parts of the methods which should be handled as transactions. Again note that the most important part the exam assignment is that `addToMean` is done atomically to ensure that `KMeans2Stm`. Mean does not change in the assignment part and therefore it does not pose a threat that it is accessed concurrently. The rest of the methods are not called on the same `Cluster` concurrently.

3.2)

Here the running time and iterations of `KMeans2Stm` can be seen:

```
# OS:    Windows 10; 10.0; amd64
# JVM:   Oracle Corporation; 1.8.0_101
# CPU:   Intel64 Family 6 Model 94 Stepping 3, GenuineIntel; 8 "cores"
# Date: 2017-01-10T19:42:09+0100
...
Used 108 iterations
class kmean.KMeans2Stm Real time:     3.214
...
Used 108 iterations
class kmean.KMeans2Stm Real time:     2.924
...
Used 108 iterations
class kmean.KMeans2Stm Real time:     3.287
```

3.3)

These are the first 5 means of `KMeans2Stm`

```
mean = (87,96523339545510, 68,02660923585381)
mean = (10,68258565222153, 47,90993881688515)
mean = (17,94392765121715, 57,92167357590438)
mean = (48,03811144575212, 28,07238725661516)
mean = (78,02580276649083, 68,03938084899143)
```

3.4)

Since the `KMeans2Stm` logic is exactly the same as in `KMeans2P` the same concurrency problems would occur if I removed the transactions code. The program would "never" terminate because each modification to the `sumx`, `sumy` and `count` fields could be overwritten due to a race-condition which would be different each time.

```
{ // Assignment step: put each point in exactly one cluster
    final Cluster[] clustersLocal = clusters;  // For capture in lambda
    Stream<Point> pointStream = Arrays.stream(points);
    Map<Cluster, List<Point>> groups = pointStream.collect(Collectors.groupingBy((
        ↪ Point p) -> {
        Cluster best = null;
        for (Cluster c : clustersLocal)
            if (best == null || p.sqrDist(c.mean) < p.sqrDist(best.mean))
                best = c;
        return best;
        // REFERENCE: Stream based version
        // Arrays.stream(clustersLocal).min((Cluster c1, Cluster c2) -> (p.sqrDist
            ↪ (c1.mean) > p.sqrDist(c2.mean)) ? 1 : -1).get()));
    }));
    clusters = groups.entrySet().stream().map( (Map.Entry<Cluster, List<Point>> kv
        ↪ ) -> new Cluster(kv.getKey().mean, kv.getValue())).toArray(size -> new
        ↪ Cluster[size]);
}
```

Figure 9: The assignment step of KMeans3

```
{ // Update step: recompute mean of each cluster
     Cluster[] newClusters = (Cluster[]) Arrays.stream(clusters).map((Cluster c)
         ↪ -> c.computeMean()).toArray(size -> new Cluster[size]);
    converged = Arrays.equals(clusters, newClusters);
    clusters = newClusters;
}
```

Figure 10: The update step of KMeans3

4 Question 4

4.1)

In figure 9 the implementation of the assignment step of `KMeans3` can be seen. The `groupingBy` is based on the same logic in `KMeans1` and the code is very alike. One could implement this with streams as well, but it seemed to be very inefficient, therefore I went for the lambda expression instead. To see the stream version, see the reference in figure 9.

4.2)

In figure 10 the implementation of the update step of `KMeans3` can be seen.

```
{ // Assignment step: put each point in exactly one cluster
    final Cluster[] clustersLocal = clusters;  // For capture in lambda
    Stream<Point> pointStream = Arrays.stream(points).parallel();
    Map<Cluster, List<Point>> groups = pointStream.collect(Collectors.groupingBy((
        ↪ Point p) -> {
        Cluster best = null;
        for (Cluster c : clustersLocal)
            if (best == null || p.sqrDist(c.mean) < p.sqrDist(best.mean))
                best = c;
        return best;
    }));
    clusters = groups.entrySet().stream().parallel().map(
            kv -> new Cluster(kv.getKey().mean, kv.getValue())).toArray(size ->
                ↪ new Cluster[size]
    );
}
```

Figure 11: The assignment step of KMeans3P

4.3)

Here the running time and iterations of `KMeans3` can be seen:

```
# OS:   Windows 10; 10.0; amd64
# JVM:  Oracle Corporation; 1.8.0_101
# CPU:  Intel64 Family 6 Model 94 Stepping 3, GenuineIntel; 8 "cores"
# Date: 2017-01-10T19:42:09+0100
...
Used 108 iterations
class kmean.KMeans3  Real time:     4.972
...
Used 108 iterations
class kmean.KMeans3  Real time:     4.935
...
Used 108 iterations
class kmean.KMeans3  Real time:     4.973
```

4.4)

In figure 11 the implementation of the assignment step of `KMeans3P` can be seen. One of the nice features and concepts of the stream system is that if something does not have side effects then it is easy to parallelize. Simply by putting the `.parallel()` after the stream makes the pipeline happen in parallel. In this case the `groupBy collect` and later the mapping transformation to the clusters again happens in parallel. Even though the `groupby collect` reads from a shared collection `clustersLocal` it does not have any side effect and therefore it can still be parallelized.

4.5)

In figure 12 the implementation of the update step of `KMeans3P` can be seen. Again simply the `.parallel()` has been added to the stream.

```
{ // Update step: recompute mean of each cluster
    Cluster[] newClusters = Arrays.stream(clusters).parallel().map(
            c -> c.computeMean()).toArray(size -> new Cluster[size]
    );
    converged = Arrays.equals(clusters, newClusters);
    clusters = newClusters;
}
```

Figure 12: The update step of KMeans3P

4.6)

Here the running time and iterations of `KMeans3P` can be seen:

```
# OS:   Windows 10; 10.0; amd64
# JVM:  Oracle Corporation; 1.8.0_101
# CPU:  Intel64 Family 6 Model 94 Stepping 3, GenuineIntel; 8 "cores"
# Date: 2017-01-10T19:42:09+0100
...
Used 108 iterations
class kmean.KMeans3P Real time:     1.289
...
Used 108 iterations
class kmean.KMeans3P Real time:     1.201
...
Used 108 iterations
class kmean.KMeans3P Real time:     1.182
```

4.7)

No it was not necessary. Concurrency problems happen when we have shared mutable state, but since the data-structure is immutable no such problems can occur with this data-structure. We can see that `Cluster` is immutable both because the fields are final but also because the points list is unmodifiable - therefore any changes must be made by creating a new cluster.

4.8)

In table 1 the running times of the different implementations can be seen. The sample was of size 3 for each implementation. The number of nodes were 200000 and $k = 81$.

Implementation	1st Iteration	2nd Iteration	3rd Iteration	Avg Time in seconds
KMeans1	5.313	5.453	5.011	5.259
KMeans1P	1.566	1.470	1.444	1.493
KMeans2	5.482	5.452	4.558	5.164
KMeans2P	1.309	1.231	1.269	1.270
KMeans2Q	2.656	2.880	2.187	2.574
KMeans2Stm	3.214	2.924	3.287	3.141
KMeans3	4.972	4.935	4.973	4.960
KMeans3P	1.289	1.201	1.182	1.224

Table 1: The running times of the different implementations of KMeans

It seems that the `KMeans2P` and `KMeans3P` are the fastest by far, though `KMeans1P` is not far behind. Both of the implementations seem to allow the first loop in general to work at in full parallel without synchronization or locking on anything in the assignment phase. Overall these implementations receive a speed-up factor of about four on my machine which has eight logical threads. This seems fairly reasonable since some of the code is still sequential, and creating tasks and starting threads also have some overhead.

In the middle of the scale at about a speed-up factor of two we have `KMeans2Q`, and `KMeans2Stm`. I have already argued for why I believe `KMeans2Q` is slower than `KMeans2P` but `KMeans2Stm` should in some sense be just as fast as the `KMeans2P`. `KMeans2Stm` only differentiate itself by using transactions instead of locks, but maybe it is simply the overhead of the multiverse framework.

The slowest implementations are predictably the non parallel implementations `KMeans1`,`KMeans2`, `KMeans3`.

5 Question 5

5.1)

On figure 13 the sequential test of the `MSQueueNeater` data structure is shown. It tests that enqueues and dequeues work as they should both on their own and intertwined.

5.2)

On figure 14, 15 and 16 the framework of the concurrent test can be seen. It is based on a producer-consumer pattern. I call the test with the following arguments `size=1000000`, `producers=10`, `consumers=10`. The `ExecutorService` is a `CachedThreadPool`.

5.3)

To make the concurrent test not terminate I mutated the enqueue method by removing the while loop. By doing so the elements might never get added to the queue since each `compareAndSet` operation could fail, and therefore we are not ensured that an element certainly gets enqueued. The sequential test never detects this because it cannot fail a `compareAndSet`. The location of the modification is marked as `MODIFICATION 1` in figure 17.

To provoke a `NullPointerException` the `tail.compareAndSet(last, next);` line in enqueue is modified to have the last argument be `null`, the sequential tests wont fail but the concurrent one will. The reason why this does not fail in the sequential test is because the tail never has another element after it. Therefore when working concurrently the `compareAndSwap` is there to move the tail reference to the true last element. But if we make this sanitising step faulty we will see that the data structure fails. The same is the case for the same line in dequeue.

This bug first became apparent when i added the random boolean check(artificially delaying production) in the producer class since it then exercised how the queue handled modifications when it had none to a few elements. The modification is marked as `MODIFICATION 2` in figure 17.

I also managed to provoke a `AssertionError` originating from the assertion at REFERENCE 1 line in 16 - that is the elements produced by a single thread were out of order. By changing all `compareAndSet` to `set` operations and removing them from if statements. By doing this we basically allow race conditions again because the variables becomes mutable by multiple threads at the same time. Therefore what happens is probably that one thread adds its element before another but updates the tail/head reference after. If this happens enough times a thread could

```
assert queue.dequeue() == null;
queue.enqueue(1);
assert queue.dequeue() == 1;
assert queue.dequeue() == null;
assert queue.dequeue() == null;
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);
assert queue.dequeue() == 1;
assert queue.dequeue() == 2;
assert queue.dequeue() == 3;
assert queue.dequeue() == null;
assert queue.dequeue() == null;
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);
assert queue.dequeue() == 1;
queue.enqueue(4);
queue.enqueue(5);
assert queue.dequeue() == 2;
assert queue.dequeue() == 3;
assert queue.dequeue() == 4;
queue.enqueue(6);
assert queue.dequeue() == 5;
assert queue.dequeue() == 6;
assert queue.dequeue() == null;
assert queue.dequeue() == null;
// "stress" test
Random r = new Random();
int total = 100_000, amtAdded = 0, amtRemoved = 0;
while(amtAdded != total || amtRemoved != total) {
  if(r.nextBoolean() && amtAdded != total) {
    queue.enqueue(amtAdded);
    amtAdded++;
  } else {
    if(amtRemoved != total && amtRemoved < amtAdded) {
      assert queue.dequeue() == amtRemoved;
      amtRemoved++;
    }
  }
}
assert queue.dequeue() == null;
```

Figure 13: The sequential test of MSQueueNeater

```
public UnboundedQueueManyToManyConcurrencyTest(final int size, final int producers
    ↪ , final int consumers, UnboundedQueue<ThreadAndNum> queue) {
    this.size = size;
    this.producers = producers;
    this.consumers = consumers;
    this.queue = queue;
    this.startBarrier = new CyclicBarrier((producers+consumers) + 1);
    this.stopBarrier = new CyclicBarrier((producers+consumers) + 1);
  }
  @Override
  public void runTest(ExecutorService pool) {
    try {
      for(int i = 0; i < producers; i++) {
        pool.execute(new Producer(i));
      }
      for(int i = 0; i < consumers; i++) {
        pool.execute(new Consumer(i));
      }
      startBarrier.await();
      System.out.println("Starting many-to-many test");
      stopBarrier.await();
      assertTrue(queue.dequeue() == null);
      assertEquals(enquedSum.get(), dequedSum.get());
      System.out.println(" - many-to-many test success");
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
    pool.shutdown();
  }
```

Figure 14: The main class for the concurrency test of MSQueueNeater

```
class Producer implements Runnable
{
  final int threadNumber;
  Producer(int threadNumber) {
    this.threadNumber = threadNumber;
  }
  @Override
  public void run() {
    try {
      final Random r = new Random();
      int sum = 0;
      startBarrier.await();
      for(int i = 0; i<size;) {
          if(r.nextBoolean()) { // important for MODIFICATION 2
              queue.enqueue(new ThreadAndNum(threadNumber, i+1));
              sum += i+1;
              i++;
          }
      }
      enquedSum.getAndAdd(sum);
      stopBarrier.await();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Figure 15: The producer of the MSQueueNeater concurrency test

have its element appear before the other elements it has added. The modification is marked as `MODIFICATION 3` in figure 17.

Unfortunately my test does not detect errors when making small adjustments to the data structure such as changing single `compareAndSet()` to `set` operations. Therefore it would be a good idea to extend the test further such that we can argue that the data structure is implemented in a thread-safe way without any unnecessary code (like the much discussed if-statement in the original implementation by Michael and Scott).

```java
class Consumer implements Runnable
{
  final int threadNumber;
  final int[] producerLastNumber;
  Consumer(int threadNumber) {
    this.threadNumber = threadNumber;
    this.producerLastNumber = new int[producers];
  }
  @Override
  public void run() {
    try {
      int sum = 0;
      startBarrier.await();
      for(int i = 0; i<size; ) {
        ThreadAndNum item = queue.dequeue();
        if(item != null) {
          assert producerLastNumber[item.threadNumber] < item.number; // REFERENCE
            ↪ 1: This is the assertion that fails because of MODIFICATION 3
          producerLastNumber[item.threadNumber] = item.number;
          sum += item.number;
          i++;
        }
      }
      dequedSum.getAndAdd(sum);
      stopBarrier.await();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Figure 16: The consumer of the MSQueueNeater concurrency test

```
public void enqueue(T item) { // at tail
  Node<T> node = new Node<T>(item, null);
  while (true) {              // MODIFICATION 1: (not actually done here)
    final Node<T> last = tail.get(), next = last.next.get();
    if (next != null) {
      //tail.compareAndSet(last, null); // MODIFICATION 2: (outcommented)
      tail.set(next);         // MODIFICATION 3:
    } else {                  // MODIFICATION 3:
      last.next.set(node);    // MODIFICATION 3:
      tail.set(node);         // MODIFICATION 3:
      return;
    }
  }
}
public T dequeue() { // from head
  while (true) {
    final Node<T> first = head.get(), last = tail.get(), next = first.next.get();
    if (next == null) {
      return null;
    } else if (first == last) {
      //tail.compareAndSet(last, null); // MODIFICATION 2: (outcommented)
      tail.set(next);         // MODIFICATION 3:
    } else {                  // MODIFICATION 3:
      head.set( next);        // MODIFICATION 3:
      return next.item;
    }
  }
}
```

Figure 17: The mutation changes of the enqueue and dequeue methods

```
public static void main(String[] args)
{
    final ActorSystem system = ActorSystem.create("OddEven");
    final ActorRef dispatcher = system.actorOf(Props.create(DispatcherActor.class)
        ↪ , "dispather");
    final ActorRef odd = system.actorOf(Props.create(WorkerActor.class), "odd");
    final ActorRef even = system.actorOf(Props.create(WorkerActor.class), "even");
    final ActorRef collector = system.actorOf(Props.create(CollectorActor.class),
        ↪ "collector");
    dispatcher.tell(new InitMessage(odd, even, collector), ActorRef.noSender());
    for(int i = 1; i<=10; i++) {
        dispatcher.tell(new NumMessage(i), ActorRef.noSender());
    }
    try {
        System.out.println("Press␣to␣Terminate");
        System.in.read();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        system.shutdown();
    }
}
```

Figure 18: The start implementation of the system specified by start.erl

6 Question 6

6.1)

In the figures 18, 19, 20, 21 and 22 my Java AKKA implementation of the specification given in `start.erl` is shown.

6.2)

The system works and produces the following output:

```
4
1
9
25
16
36
49
64
81
100
```

This is of course not in order, but as stated in the exam assignment that is expected.

```java
public class DispatcherActor extends UntypedActor {
    private ActorRef odd, even;
    @Override
    public void onReceive(Object message) throws Throwable {
        if(message instanceof InitMessage) {
            odd = ((InitMessage) message).getOdd();
            even = ((InitMessage) message).getEven();
            odd.tell(message, ActorRef.noSender());
            even.tell(message, ActorRef.noSender());
        }
        else if(message instanceof NumMessage) {
            int value = ((NumMessage) message).getNumber();
            if(value % 2 == 0) {
                even.tell(message, ActorRef.noSender());
            } else {
                odd.tell(message, ActorRef.noSender());
            }
        }
    }
}
```

Figure 19: The dispatcher actor implementation specified by start.erl

```java
public class WorkerActor extends UntypedActor {
    private ActorRef collector;
    @Override
    public void onReceive(Object message) throws Throwable {
        if(message instanceof InitMessage) {
            collector = ((InitMessage) message).getCollector();
        } else if(message instanceof NumMessage) {
            int value = ((NumMessage) message).getNumber();
            collector.tell(new NumMessage(value*value), ActorRef.noSender());
        }
    }
}
```

Figure 20: The worker actor implementation specified by start.erl

```java
public class CollectorActor extends UntypedActor {
    @Override
    public void onReceive(Object message) throws Throwable {
        if(message instanceof NumMessage) {
            System.out.println(((NumMessage) message).getNumber());
        }
    }
}
```

Figure 21: TThe collector actor implementation specified by start.erl

```
public class InitMessage implements Serializable {
    private final ActorRef odd;
    private final ActorRef even;
    private final ActorRef collector;
    public InitMessage(ActorRef odd, ActorRef even, ActorRef collector) {
        this.odd = odd;
        this.even = even;
        this.collector = collector;
    }
    public ActorRef getOdd() { return odd; }
    public ActorRef getEven() { return even; }
    public ActorRef getCollector() { return collector; }
}

public class NumMessage implements Serializable {
    private final int number;
    public NumMessage(int number) { this.number = number; }
    public int getNumber() { return number; }
}
```

Figure 22: The init and num message implementation specified by start.erl