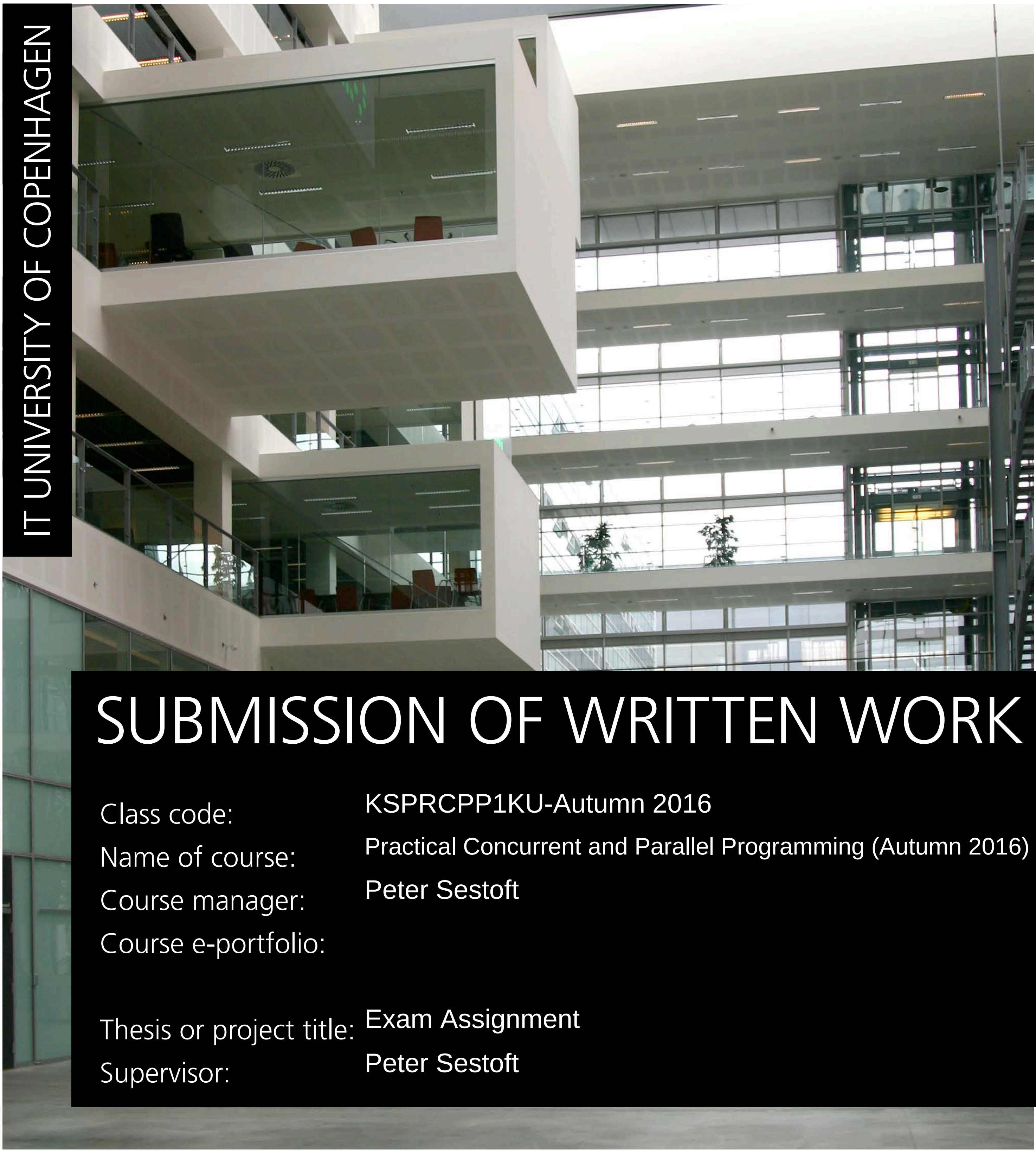

SUBMISSION OF WRITTEN WORK

Class code:	KSPRCPP1KU-Autumn 2016
Name of course:	Practical Concurrent and Parallel Programming (Autumn 2016)
Course manager:	Peter Sestoft
Course e-portfolio:	
Thesis or project title:	Exam Assignment
Supervisor:	Peter Sestoft

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Irina Alina Gabriela Luca	24/05-1993	irlu@itu.dk
2.			@itu.dk
3.			@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk

Examination
Practical Concurrent and Parallel Programming

- 10/11 January 2017 -

Irina Alina Gabriela Luca
irlu@itu.dk

'I hereby declare that I have answered these exam questions myself without any outside help.'

Irina Alina Gabriela Luca, 11.01.2017

Question 1

Subquestion 1.1

The results after compiling and running KMeans1 (along with the SystemInfo()) are listed below:

```
# OS:   Windows 10; 10.0; amd64
# JVM:  Oracle Corporation; 1.8.0_111
# CPU:  Intel64 Family 6 Model 94 Stepping 3, GenuineIntel; 8 "cores"
# Date: 2017-01-10T09:26:21+0100
Used 108 iterations
class com.company.KMeans1 Real time:     7.101

Used 108 iterations
class com.company.KMeans1 Real time:     7.155

Used 108 iterations
class com.company.KMeans1 Real time:     7.853
```

Subquestion 1.2

The resulting code after parallelizing the Assignment step is the following:

```
// Exercise 1.2: Part 1
private final ExecutorService executor
       = Executors.newWorkStealingPool();
private int P;
private int perTask;

public KMeans1P(Point[] points, int k, int P) { // P = # tasks
   [...]
   this.P = P;
   this.perTask = points.length / P;
}

// Exercise 1.2: Part 2 (Assignment step: put each point in exactly one cluster)
List<Callable<Void>> assignmentTasks = new ArrayList<Callable<Void>>();
final Cluster[] clustersF = clusters;
for (int t = 0; t < this.P; t++) {
   final int from = this.perTask * t,
             to = (t + 1 == this.P) ? this.points.length : this.perTask * (t + 1);
   assignmentTasks.add(() -> {
       for (int i = from; i < to; i++) {
           Cluster best = null;
           for (Cluster c : clustersF)
               if (best == null || points[i].sqrDist(c.mean) <
points[i].sqrDist(best.mean)) {
                   best = c;
               }
           best.add(points[i]);
       }
       return null;
   });
}
try {
   executor.invokeAll(assignmentTasks);
```

```java
} catch (InterruptedException exn) {
   System.out.println("Interrupted: " + exn);
}
```

As **Bloch item 68** states it, **‘the general mechanism for executing tasks is the executor service’**. As such, the code listed above declares a single `ExecutorService` executor, which the tasks will be submitted to later on. A `newWorkStealingPool()` is used in this context, because it provides better scalability. Notably, the executor was created once, since redeclaration would yield costful overhead. Additional fields were needed for the class, where this.P represents the number of desired tasks and this.perTask represents how much work each task is expected to handle. In the second part of the code above, a `List<Callable< Void>>` has been declared, in order to add the tasks to it, as well as a `final Cluster[] clustersF` , needed for capture in lambda. Each task establishes a range to work on, through the two variables: `final int from` and `final int to.` As such, the work for each range is done inside the lambda, where each point in the range is assigned to a suitable cluster, after looping through all the clusters and finding it. In the end, by calling `executor.invokeAll(tasks);` , tasks are submitted and one can then wait for them all to complete.

Subquestion 1.3

The resulting code after parallelizing the Update step is the following:

```java
// Exercise 1.3: (Update step: recompute mean of each cluster)
ArrayList<Cluster> newClusters = new ArrayList<>();
converged = true;
List<Callable<Cluster>> tasksUpdate = new ArrayList<Callable<Cluster>>();
int clustersTaskCount = clusters.length;
final Cluster[] clustersF = clusters;
final AtomicBoolean convergedF = new AtomicBoolean(true);
for (int t = 0; t < clustersTaskCount; t++) {
   final int tF = t;
   tasksUpdate.add(() -> {
       Cluster cluster = null; // Task local cluster
       Point mean = clustersF[tF].computeMean();
       if (!clustersF[tF].mean.almostEquals(mean))
           convergedF.set(false);
       if (mean != null) {
           cluster = new Cluster(mean);
       } else
           System.out.printf("===> Empty cluster at %s%n", clustersF[tF].mean);
       return cluster;
   });
}

try {
   List<Future<Cluster>> futures = executor.invokeAll(tasksUpdate);
   for (Future<Cluster> fut : futures) {
       Cluster resultedCluster = fut.get();
       if (resultedCluster != null) {
           newClusters.add(resultedCluster);
       }
   }
} catch (InterruptedException exn) {
   System.out.println("Interrupted: " + exn);
} catch (ExecutionException exn) {
   throw new RuntimeException(exn.getCause());
} finally {
   converged = convergedF.get(); // get the accumulated converged and assign it
   clusters = newClusters.toArray(new Cluster[newClusters.size()]);
}
```

As the code above shows it, a `List<Callable<Cluster>>` (having as a return type `Cluster` instead of `Void`) was defined, so that in the `try/catch/finally`, one would loop through the `Cluster` results, where each result is represented by `fut.get()`, and add them in the final `Cluster` collection, for those that are not `null`. Compared to the Assignment step, we do not split the work per ranges, but instead, per individual cluster. A local `Clusters[]` array was needed, in order to be able to reference it inside the lambda, as well as an additional `AtomicBoolean(true)` 'accumulator', which assigns the final converged value to the `boolean converged` variable in the end, after obtaining the results from the `futures`. Notably, the iteration index `final int tF = t;` pointing at each cluster task has to be declared final, as well, in order to operate inside the lambda. Since `fut.get()` can throw `InterruptedException`, a suitable `try/catch` was used.

Subquestion 1.4

The code fragment below shows how to make the nested `Cluster` class thread-safe, with locking:

```
public void add(Point p) {
   synchronized (this) { // intrinsic lock
       points.add(p);
   }
}
```

The Update step creates a parallel task for each and every cluster, such that points belonging to different clusters do not collide with each other. The Assignment step is creating tasks that all work concomitantly on the clusters. Consequently, in order to make the `Cluster` class thread-safe, one can use an intrinsic lock only on the `add()` method, which should suffice, given the fact that the state represented by the points must be guarded whenever a `Point` is to be added and also given the fact that the addition only happens in the Assignment step. By locking on `this`, one would ensure visibility and mutual exclusion in the actual context. Moreover, `computeMean()` is only called in the Update step, after the points have been assigned to the clusters, which means that `computeMean()` does not need to use any synchronization mechanism. If one needed to add points and compute the mean at the same time, both methods would require synchronization and usage of the same lock for synchronization, since according to **'Java Concurrency in Practice' (Brian Goetz)**, **'for each mutable state variable that may be accessed by more than one thread, all accesses to that variable must be performed with the same lock held'** (page 28).

Subquestion 1.5

The execution time for KMeans1P, along with the `SystemInfo()` and the number of iterations used are listed below:

```
# OS:   Windows 10; 10.0; amd64
# JVM:  Oracle Corporation; 1.8.0_111
# CPU:  Intel64 Family 6 Model 94 Stepping 3, GenuineIntel; 8 "cores"
# Date: 2017-01-10T13:26:46+0100
Used 108 iterations
class com.company.KMeans1P Real time:     2.690

Used 108 iterations
class com.company.KMeans1P Real time:     2.274

Used 108 iterations
class com.company.KMeans1P Real time:     2.370.
```

As noticed, parallelization decreases the execution time significantly, from ~7s to ~2s. Moreover, using tasks instead of threads in order to parallelize the work is convenient, since creating many threads takes time and memory and would not have suited the actual context.

Subquestion 1.6

Means representing the first 5 clusters are listed below:

```
mean = (87.96523339545512, 68.02660923585371)
mean = (10.68258565222153, 47.90993881688516)
mean = (17.94392765121714, 57.92167357590444)
mean = (48.03811144575212, 28.07238725661512)
mean = (78.02580276649083, 68.03938084899140).
```

As noticed, the values are very close to the ones obtained in **Subquestion 1.1**.

Subquestion 1.7

After removing the thread-safety mechanisms and rerunning, the following exception is thrown:

```
Exception in thread "main" java.lang.RuntimeException: java.lang.NullPointerException
       at com.company.KMeans1P.findClusters(TestKMeans.java:257)
       at com.company.TestKMeans.timeKMeans(TestKMeans.java:46)
       at com.company.TestKMeans.main(TestKMeans.java:30)
       at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) [...].
```

Most likely, the exception stems from the Assignment step, where race conditions are allowed when two threads are trying to access the state of the same cluster, in order to add points to it. Therefore, the synchronization mechanisms are required for points' addition to clusters.

Question 2

Subquestion 2.1

After running KMeans2, just as it is given, I get the following results:

```
# OS:   Windows 10; 10.0; amd64
# JVM:  Oracle Corporation; 1.8.0_111
# CPU:  Intel64 Family 6 Model 94 Stepping 3, GenuineIntel; 8 "cores"
# Date: 2017-01-10T13:53:47+0100
Used 108 iterations
class com.company.KMeans2 Real time:     7.276

Used 108 iterations
class com.company.KMeans2 Real time:     6.928

Used 108 iterations
class com.company.KMeans2 Real time:     5.787.
```

Subquestion 2.2

The following code shows the task parallelization for the Assignment step, using p = 8 tasks, each handling 1/8 of the 200,000 points:

```java
// Exercise 2.2: Part 1
private final ExecutorService executor = Executors.newWorkStealingPool();
private int P;
private int perTask;

public KMeans2P(Point[] points, int k, int P) {
   [...]
   this.P = P;
   this.perTask = points.length / P;
}
// Exercise 2.2 (the Assignment step)
List<Callable<Void>> assignmentTasks = new ArrayList<Callable<Void>>();
for (int t = 0; t < this.P; t++) {
   final int from = this.perTask * t,
             to = (t + 1 == this.P) ? this.points.length : this.perTask * (t + 1);
   assignmentTasks.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 {
   executor.invokeAll(assignmentTasks);
} catch (InterruptedException exn) {
   System.out.println("Interrupted: " + exn);
}
```

Just as in **Question 1**, the executor is defined once, after the existing fields, along with `P`, representing the amount of tasks and `perTask`, representing the amount of work each task has to handle. Identically, a `List<Callable<Void>>` is used, such that each task will be gradually added to it. When each task proceeds, it iterates through the clusters and finds the most suitable one (`best`) and assigns it to the `myCluster[pi]`, meaning that the point identified by index `pi` belongs to `myCluster[pi]`. All in all, the approach used here is very similar to the one described in the previous exercise with regards to how task splitting is implemented in the Assignment step, even though the clusters representation differs here.

Subquestion 2.3

The code fragment below shows the task parallelization for the Update step, using p = 8 tasks:

```
// Update step: recompute mean of each cluster
for (Cluster c : clusters) {
   c.resetMean();
}
// START: task parallelization for Update step
List<Callable<Void>> updateTasks = new ArrayList<Callable<Void>>();
for (int t = 0; t < this.P; t++) {
   final int from = this.perTask * t,
            to = (t + 1 == this.P) ? this.points.length : this.perTask * (t + 1);
   updateTasks.add(() -> {
       for (int pi = from; pi < to; pi++) {
           myCluster[pi].addToMean(points[pi]);
       }
       return null;
   });
}
try {
   executor.invokeAll(updateTasks);
} catch (InterruptedException exn) {
   System.out.println("Interrupted: " + exn);
}
// END: task parallelization for Update step
converged = true;
for (Cluster c : clusters)
   converged &= c.computeNewMean();
```

This time, the Update step can also use the approach of distributing the work across tasks similarly to the Assignment step. However, the only fragment of the Update task code that requires parallelization is represented by the `for` loop accessing the `points[]` array. The same task parallelization mechanisms are used, just as in the previous questions. Interestingly enough, after parallelizing the Update task and having both the Assignment task and the Update task parallelized, compared to when only having the Assignment task parallelized, the execution time is actually slower, most likely because of some task overhead created when parallelizing just a single operation such as `myCluster[pi].addToMean(points[pi]);` . However, the difference is not significant.

Subquestion 2.4

By adding the keyword `synchronized` on the method addToMean() only, the nested class Cluster becomes thread-safe in this context:

```
public synchronized void addToMean(Point p) {
   sumx += p.x;
   sumy += p.y;
   count++;
}
```

Consequently, the primitive values defined by `sumx`, `sumy` and `count`, used in order to compute the mean of the cluster, would be safely used in a multithreaded context, where the synchronization ensures visibility (**'locking is not just about mutual exclusion; it is also about memory visibility [...]'- Goetz, page 37**) and mutual exclusion, such that only one thread at a time would be allowed to access the values and mutate them. Furthermore, in the actual context, `computeNewMean()` does not need synchronization, because the `boolean` values which it returns are sequentially aggregated after the parallelization phase.

Subquestion 2.5

The execution time using KMeans2P, along with the number of iterations used and the `SystemInfo()` are listed below:

```
# OS:   Windows 10; 10.0; amd64
# JVM:  Oracle Corporation; 1.8.0_111
# CPU:  Intel64 Family 6 Model 94 Stepping 3, GenuineIntel; 8 "cores"
# Date: 2017-01-10T15:01:36+0100
Used 108 iterations
class com.company.KMeans2P Real time:     2.249

Used 108 iterations
class com.company.KMeans2P Real time:     2.090

Used 108 iterations
class com.company.KMeans2P Real time:     1.928.
```

Subquestion 2.6

The means of the first 5 clusters are listed below:

```
mean = (87.96523339545513, 68.02660923585371)
mean = (10.68258565222153, 47.90993881688517)
mean = (17.94392765121714, 57.92167357590442)
mean = (48.03811144575212, 28.07238725661510)
mean = (78.02580276649084, 68.03938084899140).
```

Subquestion 2.7

After removing the thread-safety features and running KMeans2P again, the program does not finish the execution. The SystemInfo() print is listed in the console, along with the print for infinite loops diagnosis:

```
# OS:   Windows 10; 10.0; amd64
# JVM:  Oracle Corporation; 1.8.0_111
# CPU:  Intel64 Family 6 Model 94 Stepping 3, GenuineIntel; 8 "cores"
# Date: 2017-01-11T11:04:31+0100
[1][2][3][4][5][6][7][8][9][10][11][12][13][14][15][16][17][18][19][20][21][22][23][24][25]
[26][27][28][29][30][31][32][33][34][35][36][37][38][39][40][41][42][43][44][45][46][47][48
][49][50][51][52][53][54][55][56][57][58][59][60][61][62][63][64][65][66][67][68][69][70][7
1][72][73][74][75][76][77][78][79][80][81][82][83][84][85][86][87][88][89][90][91][92][93][
94][95][96][.............................].
```

The infinite loop most likely stems from the fact that `addToMean()` allows, now, multiple threads to access the primitives `sumx`, `sumy` and `count`, used to compute the mean, but without synchronization, visibility and mutual exclusion are lacking, therefore allowing the existence of race conditions. Notably, the three fields would become corrupt (lost updates, stale values etc.) and even more, they would generate a corrupt nature for `computeNewMean()`, as well. Consequently, the boolean values returned by `computeNewMean()` could influence the execution with the obtained infinite loop, because the convergence needed to complete the the execution would never occur.

Subquestion 2.8

The described approach represents a reordering of operations, such that the partial parallelization in the Update step would not be required there anymore. However, just by moving the `addToMean()` operation from one parallelized code fragment to another one and saving up a loop execution does not make a difference if the `Cluster` class is not thread-safe. Since the task parallelization code uses the `Cluster` class, it must be thread-safe . If keeping the `synchronized` keyword on `addToMean()` and ensuring thread-safety, the execution completes and it yields the following results, proving a slightly slower performance than before trying out the described approach:

```
# OS:   Windows 10; 10.0; amd64
# JVM:  Oracle Corporation; 1.8.0_111
# CPU:  Intel64 Family 6 Model 94 Stepping 3, GenuineIntel; 8 "cores"
# Date: 2017-01-10T15:18:57+0100
Used 108 iterations.
class com.company.KMeans2Q Real time:     3.736

Used 108 iterations
class com.company.KMeans2Q Real time:     3.528

Used 108 iterations
class com.company.KMeans2Q Real time:     4.961.
```

Subquestion 2.9

The code for KMeans2Q's `findClusters()` is listed below:

```
public void findClusters(int[] initialPoints) {
   final Cluster[] clusters = GenerateData.initialClusters(points, initialPoints,
Cluster::new, Cluster[]::new);
   boolean converged = false;
   while (!converged) {
       iterations++;
       {
           // Exercise 2.8-9: Reset the Mean before assigning
           for (Cluster c : clusters) {
               c.resetMean();
           }
           // START: Assignment step
           List<Callable<Void>> assignmentTasks = new ArrayList<Callable<Void>>();
           for (int t = 0; t < this.P; t++) {
               final int from = this.perTask * t,
                         to = (t + 1 == this.P) ? this.points.length : this.perTask * (t +
1);
               assignmentTasks.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 {
               executor.invokeAll(assignmentTasks);
           } catch (InterruptedException exn) {
               System.out.println("Interrupted: " + exn);
           }
           // END: Assignment step
        }
        {
           // START: Update step -- recompute mean of each cluster
           converged = true;
           for (Cluster c : clusters)
               converged &= c.computeNewMean();
           // END: Update step -- recompute mean of each cluster
       }
//             System.out.printf("[%d]", iterations); // To diagnose infinite loops
   }
   this.clusters = clusters;
}
```

Subquestion 2.10

For the purpose of this exercise, I have created an extra class (named KMeans2PAssignmentPOnly), where task parallelization occurs only in the Assignment step. Consequently, the three classes I will compare are the following:

- KMeans2P, which task-parallelizes both steps and performs `addToMean()` in the Update step;
- KMeans2Q, which which task-parallelizes only the Assignment step and resets the mean before assigning;
- KMeans2PAssignmentPOnly, which task-parallelizes only the Assignment step and performs `addToMean()` in the Update step;

The execution times for the above mentioned classes are listed in the table below:

	KMeans2P	KMeans2Q	KMeans2PAssignmentPOnly
1st run	2.034s	3.581s	1.542s
2nd run	1.715s	2.884s	1.488s
3rd run	1.657s	3.959s	1.496s

The SystemInfo() and the number of iterations are listed below:

```
# OS:   Windows 10; 10.0; amd64
# JVM:  Oracle Corporation; 1.8.0_111
# CPU:  Intel64 Family 6 Model 94 Stepping 3, GenuineIntel; 8 "cores"
# Date: 2017-01-10T15:43:07+0100
Used 108 iterations.
```

While on my machine, KMeans2P is faster than KMeans2Q, KMeans2PAssignmentPOnly can yield even better performance. Therefore, parallelization can often result in better execution time, but KMeans2PAssignmentPOnly shows that the manner of constructing programs can also be powerful in terms of performance.

Question 3

Subquestion 3.1

The thread-safe implementation of the `Cluster` subclass using transactional memory is listed below:

```
static class Cluster extends ClusterBase {
   private Point mean;
   private TxnDouble sumx, sumy;
   private TxnInteger count;

   public Cluster(Point mean) {
       this.mean = mean;
       sumx = newTxnDouble(0.0);
       sumy = newTxnDouble(0.0);
       count = newTxnInteger(0);
```

```
    }

    public void addToMean(Point p) {
        atomic(() -> {
            sumx.incrementAndGet(p.x);
            sumy.incrementAndGet(p.y);
            count.increment();
        });
    }

    // Recompute mean, return true if it stays almost the same, else false
    public boolean computeNewMean() {
        Point oldMean = this.mean;
        this.mean = new Point(sumx.atomicGet() / count.atomicGet(), sumy.atomicGet() /
count.atomicGet());
        return oldMean.almostEquals(this.mean);
    }

    public void resetMean() {
        sumx = newTxnDouble(0.0);
        sumy = newTxnDouble(0.0);
        count = newTxnInteger(0);
    }

    @Override
    public Point getMean() {
        return mean;
    }
}
```

The above class can be safely used in a multithreaded environment, since it uses the concurrent-safe transactional operations provided by the `Multiverse library`. The elements contributing to its thread-safety consist of the following changes:

- The int and double fields are declared as transactional variables, while their associated transactional methods are being used in order to ensure atomicity and thread-safe memory writes and reads. All fields are private, which is good practice, and the field Point mean does not require to be a Txn reference, since the mean will always be assigned a `new Point` containing a value dependant on the previously mentioned Txn fields;
- The `addToMean()` content must be wrapped in an atomic block, specific to the Multiverse library, such that it will define a transaction, thus being atomic, consistent and isolated.
- Atomic reads are required when computing the `new Point`'s value to be assigned to our mean and are ensured through `atomicGet()` in `computeNewMean()`.

Subquestion 3.2

The execution time using KMeans2Stm and the number of iterations used, along with SystemInfo() are listed below:

```
# OS:   Windows 10; 10.0; amd64
# JVM:  Oracle Corporation; 1.8.0_111
# CPU:  Intel64 Family 6 Model 94 Stepping 3, GenuineIntel; 8 "cores"
# Date: 2017-01-10T16:37:21+0100
```

```
Jan 10, 2017 4:37:21 PM org.multiverse.api.GlobalStmInstance <clinit>
INFO: Initializing GlobalStmInstance using factoryMethod
'org.multiverse.stms.gamma.GammaStm.createFast'.
Jan 10, 2017 4:37:21 PM org.multiverse.api.GlobalStmInstance <clinit>
INFO: Successfully initialized GlobalStmInstance using factoryMethod
'org.multiverse.stms.gamma.GammaStm.createFast'
Used 108 iterations
class com.company.KMeans2Stm Real time:     4.272
Used 108 iterations
class com.company.KMeans2Stm Real time:     4.112
Used 108 iterations
class com.company.KMeans2Stm Real time:     3.782.
```

Subquestion 3.3

The means of the first 5 clusters are listed below:

```
mean = (87.96523339545503, 68.02660923585385)
mean = (10.68258565222153, 47.90993881688516)
mean = (17.94392765121714, 57.92167357590441)
mean = (48.03811144575211, 28.07238725661514)
mean = (78.02580276649080, 68.03938084899140).
```

Subquestion 3.4

After removing all the thread-safety features, I get a Multiverse library exception:

```
# OS:   Windows 10; 10.0; amd64
# JVM:  Oracle Corporation; 1.8.0_111
# CPU:  Intel64 Family 6 Model 94 Stepping 3, GenuineIntel; 8 "cores"
# Date: 2017-01-10T16:46:50+0100
Jan 10, 2017 4:46:50 PM org.multiverse.api.GlobalStmInstance <clinit>
INFO: Initializing GlobalStmInstance using factoryMethod
'org.multiverse.stms.gamma.GammaStm.createFast'.
Jan 10, 2017 4:46:50 PM org.multiverse.api.GlobalStmInstance <clinit>
INFO: Successfully initialized GlobalStmInstance using factoryMethod
'org.multiverse.stms.gamma.GammaStm.createFast'.
Exception in thread "main" org.multiverse.api.exceptions.TxnMandatoryException
```

However, if only removing the `atomic()` block in `addToMean()`, but still use `sumx/y.atomicGet()`, then the execution never completes, such that an infinite loop occurs. Therefore, the mean gets corrupt, because it relies on the corrupt updates inside `addToMean()`. That will further on determine that `computeNewMean()` will generate a wrong `converged` boolean value, such that the `while` loop in KMeans2Stm will never stop executing.

Question 4

Subquestion 4.1

The stream-based Assignment step code for KMeans3 is listed below:

```
// Assignment step: put each point in exactly one cluster
final Cluster[] clustersLocal = clusters;
Stream<Point> sPoints = Arrays.stream(points);
Map<Cluster, List<Point>> groups = sPoints.collect(Collectors.groupingBy(point -> {
       Cluster best = null;
       for (Cluster c : clustersLocal)
           if (best == null || point.sqrDist(c.mean) < point.sqrDist(best.mean))
               best = c;
       return best;
}));
clusters = groups.entrySet().stream().map(kv -> {
       Cluster oldCluster = kv.getKey();
       List<Point> points = kv.getValue();
       return new Cluster(oldCluster.getMean(), points);
}).toArray(size -> new Cluster[size]); // from StackOverflow:
http://stackoverflow.com/questions/23079003/how-to-convert-a-java-8-stream-to-an-array
```

Subquestion 4.2

The stream-based Update step code for KMeans3 is listed below:

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

Subquestion 4.3

The execution time for using the KMeans3 class and the number of iterations used, along with the SystemInfo() are listed below:

```
# OS:   Windows 10; 10.0; amd64
# JVM:  Oracle Corporation; 1.8.0_111
# CPU:  Intel64 Family 6 Model 94 Stepping 3, GenuineIntel; 8 "cores"
# Date: 2017-01-10T18:36:38+0100
       Used 108 iterations
class com.company.KMeans3 Real time:     6.608
Used 108 iterations
class com.company.KMeans3 Real time:     6.444
Used 108 iterations
class com.company.KMeans3 Real time:     6.388.
```

Subquestion 4.4

The parallelized code fragment for the Assignment step is shown below:

```
Stream<Point> sPoints = Arrays.stream(points);
Map<Cluster, List<Point>> groups = sPoints.parallel().collect(Collectors.groupingBy(point
-> {
   Cluster best = null;
   for (Cluster c : clustersLocal)
       if (best == null || point.sqrDist(c.mean) < point.sqrDist(best.mean))
           best = c;
   return best;
}));
clusters = groups.entrySet().stream().parallel().map(kv -> {
   Cluster oldCluster = kv.getKey();
   List<Point> points = kv.getValue();
   return new Cluster(oldCluster.getMean(), points);
}).toArray(size -> new Cluster[size]);
```

During the Assignment step, the parallelization consists of applying `.parallel()` to our stream of points, thus making good use of the parallelization mechanisms which Java 8 enforces.

Subquestion 4.5

The parallelized code fragment for the Update step is shown below:

```
Cluster[] newClusters = Stream.of(clusters).parallel().map(cluster ->
cluster.computeMean()).toArray(size -> new Cluster[size]);
```

As shown in the code fragments above, including the ones from **Subquestion 4.4**, the parallelization is used either in the `collect` phase or in the `map` phase. In the firstly mentioned phase, one can simply process each point parallely and assign/group it to the right cluster, while in the secondly mentioned phase, the mapper can also process the entrySet of clusters in a stream-based parallelized manner, since they will not collide with each other.

Subquestion 4.6

The execution time using the KMeans3P class and the number of iterations used, along with the SystemInfo() are listed below:

```
# OS:   Windows 10; 10.0; amd64
# JVM:  Oracle Corporation; 1.8.0_111
# CPU:  Intel64 Family 6 Model 94 Stepping 3, GenuineIntel; 8 "cores"
# Date: 2017-01-10T19:42:20+0100
Used 108 iterations.
class com.company.KMeans3P Real time:     2.006

Used 108 iterations
class com.company.KMeans3P Real time:     1.725

Used 108 iterations
class com.company.KMeans3P Real time:     1.566.
```

Subquestion 4.7

No change needs to be made in the Cluster class in order to make it thread-safe for use inside KMeans3P, since it uses no shared state. As such, the points are wrapped in an unmodifiable list (`this.points = Collections.unmodifiableList(points);`), which contains points that are immutable, as well, and consequently thread-safe **('Immutable objects are always thread-safe.' - Goetz, page 46-47**). Moreover, the `computeMean()` always returns a new cluster reference. When operating on the points and clusters in a stream-based parallelized fashion, different threads will operate on different instances of the intermediate results. Therefore, partitions are handled in independently, while the combining the results will also be handled safely.

Subquestion 4.8

The table below lists the execution times of the KMeans1, KMeans1P, KMeans2, KMeans2P, KMeans2Q, KMeans2Stm, KMeans3 and KMeans3P algorithms:

	KMeans1	KMeans 1P	KMeans 2	KMeans 2Q	KMeans 2P	KMeans 2Stm	KMeans 3	KMeans 3P
1st run	7.206s	2.378s	7.813s	4.391s	1.783s	4.269s	7.062s	1.734s
2nd run	8.173s	2.206s	7.976s	3.025s	1.712s	3.958s	7.067s	1.656s
3rd run	8.076s	2.201s	6.659s	5.030s	1.692s	5.180s	7.242s	1.603s

As shown, the stream-based parallelized version yields the best performance with regards to the execution time. However, an extremely close performance is also illustrated by KMeans2P, where the representation of the clusters involves an array of points and an array of clusters, both of the same size (200,000). The worst performance is given by KMeans1, the very initial implementation, sequential, most likely because it does not utilize any kind of optimization.
Generally, one can notice how the parallelized/ optimized versions provide a much better performance, except for the KMeans2Stm. Possibly, the implementation based on the Multiverse library becomes slower because of eventual roll-backs when transactions try to perform and fail, since changes belong to each and every transaction and will be committed and pushed globally only if they succeeded or otherwise, they will be rolled back. Naturally, `addToMean()` plays a quite central role in the algorithm, and since it performs write updates on the Txn variables in the `Cluster` subclass and represents the longest transaction, as well, that might explain why KMeans2Stm's execution time is not as well placed in the above hierarchy, since the optimistic approach provides automatically very scalable read-parallelism, but becomes less effective when writes are the dominant factor.
Finally, the best performance is gained by KMeans3P, implemented in a stream-based fashion. One reason to it might be the pipeline execution which KMeans3P uses everywhere, including the `Cluster` subclass. However, the implemented parallelization strategy might add its own contribution to it.

Question 5

Subquestion 5.1

The sequential test passes the assertions when both running with the option -ea and using the Tests class functionality, as shown below:

```
private static void sequentialTest(UnboundedQueue<Integer> queue) throws Exception {
       System.out.println("%nsequentialTest " + queue.getClass());
       Tests.assertTrue(queue.dequeue() == null); // tests if queue is empty
       queue.enqueue(3);
       queue.enqueue(5);
       queue.enqueue(4);
       Tests.assertTrue(queue.dequeue() == 3);
       Tests.assertTrue(queue.dequeue() == 5);
       queue.enqueue(8);
       Tests.assertTrue(queue.dequeue() == 4);
       Tests.assertTrue(queue.dequeue() == 8);
       queue.enqueue(12);
       queue.enqueue(8);
       queue.enqueue(12);
       Tests.assertTrue(queue.dequeue() == 12); // tests right order of the dequeued item
       Tests.assertTrue(queue.dequeue() == 8);
       Tests.assertTrue(queue.dequeue() == 12);
       Tests.assertTrue(queue.dequeue() == null); // tests if queue is empty
       Tests.assertTrue(queue.dequeue() == null); // tests if queue is empty
       for(int i = 1; i <= 10; i++) {
           queue.enqueue(i);
       }
       for(int i = 1; i <= 10; i++) {
           Tests.assertTrue(queue.dequeue() == i);
       }
       System.out.println("%nsequentialTest ... Passed!");
   }
}
```

As such, the passed tests prove how the FIFO order of items is respected, as well as how a dequeued item is always null when the queue is empty. Since the queue in unbounded, one can enqueue() items unboundedly, as much as the memory allows it. Notably, the tests have been run on a queue containing integers only.

Subquestion 5.2

The concurrent test is illustrated by the following code:

```
public class QueueTests extends Tests {
   protected CyclicBarrier startBarrier, stopBarrier;
   protected final MSQueueNeater<Integer> queue;
   protected final int nTrials, nPairs;
   protected final AtomicInteger enqueueSum = new AtomicInteger(0);
   protected final AtomicInteger dequeueSum = new AtomicInteger(0);

   public QueueTests(MSQueueNeater<Integer> queue, int npairs, int ntrials) {
       this.queue = queue;
       this.nTrials = ntrials;
```

```
        this.nPairs = npairs;
        this.startBarrier = new CyclicBarrier(npairs * 2 + 1);
        this.stopBarrier = new CyclicBarrier(npairs * 2 + 1);
    }

    void test(ExecutorService pool) {
        try {
            for (int i = 0; i < nPairs; i++) {
                pool.execute(new Producer());
                pool.execute(new Consumer());
            }
            startBarrier.await(); // wait for all threads to be ready
            stopBarrier.await();  // wait for all threads to finish
            assertEquals(enqueueSum.get(), dequeueSum.get());
//             System.out.println("enqueueSum.get(): " + enqueueSum.get());
//             System.out.println("dequeueSum.get(): " + dequeueSum.get());
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    class Producer implements Runnable {
        public void run() {
            try {
                Random random = new Random();
                int sum = 0;
                startBarrier.await();
                for (int i = nTrials; i > 0; --i) {
                    int item = random.nextInt();
                    // Can enqueue all the time,
                    // because it is unbounded
                    queue.enqueue(item);
                    sum += item;
                }
                enqueueSum.getAndAdd(sum);
                stopBarrier.await();
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        }
    }

    class Consumer implements Runnable {
        public void run() {
            try {
                startBarrier.await();
                int sum = 0;
                for (int i = nTrials; i > 0; --i) {
                    Integer dequeuedItem = null;
                    // Try to deque until an actual item succeeds and is not be null,
                    // so the local sum can be updated
                    while(dequeuedItem == null) {
                        dequeuedItem = queue.dequeue();
                    }
                    sum += dequeuedItem;
                }
                dequeueSum.getAndAdd(sum);
                stopBarrier.await();
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

As shown in the code, I created a `QueueTests` class, which includes both a `Consumer` and a `Producer` class. The queue instance I ran the test on works on Integer(s). As an `ExecutorService` pool, I used a `newCachedThreadPool`, which dynamically adapts the amount of threads. The `QueueTests` class defines two `CyclicBar rier` instances, one that will make sure all the testing threads are ready to start at the same time and another one that will make sure all the testing threads terminate before checking the results. Additionally, two `AtomicInteger` variables, an `enqueueSum` which will accumulate the local sum variables created by the `Producer` and a `dequeueSum` which will accumulate all the Integer values that a `Consumer` will deque. Furthermore, I defined two variables in order to instantiate the number of trials used in both the `Producer` and the `Consumer`, as long as the number of `Runnable`(s) to execute to the pool `Execut orService`. Importantly, when instantiating the `CyclicBarrier`(s), one must define `n` test threads, plus one main thread at the same time, thus adding up to `N = # producers + # consumers + 1`. One could use one `CyclicBarrier` for both starting and stopping, precisely because it is cyclic, but the code becomes clearer by distinguishing them. Furthermore, the `Producer` can enqueue unboundedly to the queue instance, at the same time adding the local sum to the `AtomicInteger` global `enqueueSum`, while the `Consumer` will try to dequeue an Integer until it gets it, thus updating the dequeue global sum with the local sum. In the end, the `test()` method will assert the equality between the two global sums. Notably, the sums approach implemented does not give any information about the order in which the queue handles the items, which the sequential test does. Naturally, after the test is completed, the pool needs to be shutdown, as performed in the method below:

```
private static void parallelTest(MSQueueNeater<Integer> queue, int pairs, int trials) {
   final ExecutorService pool = Executors.newCachedThreadPool();
   new QueueTests(queue, pairs, trials).test(pool);
   pool.shutdown();
   System.out.println("%nparallelTest ... Passed!");
}.
```

Subquestion 5.3

In the Michael-Scott implementation, the operation defined by `tail.compareAndSet(last, next);` is catching the queue in an intermediate state, where the tail is lagging behind, thus moving the tail further and helping out the next enqueuer. After removing the above mentioned operation, the parallel test still runs fine, most likely because it cannot observe how the nodes are linked and whether the tail is lagged behind. A similar mutation, which is also not observed by the parallel test is setting the last as a reference to head, instead of tail. Same as before, test still does not catch the inconsistency.

However, when trying to change the condition in `enqueue()` before setting the tail from the form
```
if (next != null)
   tail.compareAndSet(last, next);
```

into the form
```
if (next == null)
   tail.compareAndSet(last, next);,
```
I get a `NullPointerException`:

```
Exception in thread "pool-1-thread-1" Exception in thread "pool-1-thread-5"
java.lang.RuntimeException: java.lang.NullPointerException
       at com.company.QueueTests$Producer.run(QueueTests.java:59)[...].
```

The reason might be that tail will then be set to null and the enqueuer will never actually get successful in its attempt.
Moreover, if trying to mutate the enqueuer again by turning the following snippet
```
if (next != null)
   tail.compareAndSet(last, next);
```

into the following
```
if (next != null)
   tail.compareAndSet(next, last);,
```
then the program never terminates. The reason to it might be that the tail will lag behind, which generates an always failed attempt to move it to its expected position.

Question 6

Subquestion 6.1

The implementation reflecting the system in Java+Akka is the following:
```
package com.company;

import java.io.*;
import akka.actor.*;

// -- MESSAGES --------------------------------------------------
class InitMessage implements Serializable {
   private ActorRef odd, even, collector;
   public InitMessage(ActorRef odd, ActorRef even, ActorRef collector) {
       this.odd = odd;
       this.even = even;
       this.collector = collector;
   }
   public InitMessage(ActorRef collector) {
       this.collector = collector;
   }

   public ActorRef getOdd() {
       return odd;
   }

   public ActorRef getEven() {
       return even;
   }

   public ActorRef getCollector() {
       return collector;
   }
}
class NumMessage implements Serializable {
   private final int number;
   public NumMessage(int number) {
       this.number = number;
   }

   public int getNumber() {
       return number;
   }
}
// -- ACTORS --------------------------------------------------
class DispatcherActor extends UntypedActor {
   private ActorRef odd, even;
   @Override
   public void onReceive(Object message) throws Throwable {
       if (message instanceof InitMessage) {
           InitMessage initMessage = (InitMessage) message;
           ActorRef collector = initMessage.getCollector();
           this.even = initMessage.getEven();
           this.odd = initMessage.getOdd();

           this.odd.tell(new InitMessage(collector), ActorRef.noSender());
           this.even.tell(new InitMessage(collector), ActorRef.noSender());

       } else if(message instanceof NumMessage) {
           NumMessage numMessage = (NumMessage) message;
           int number = numMessage.getNumber();
           if(number % 2 == 0) {
               this.even.tell(new NumMessage(number), ActorRef.noSender());
           } else {
               this.odd.tell(new NumMessage(number), ActorRef.noSender());
           }
       }
   }
}
class WorkerActor extends UntypedActor {
   private ActorRef collector;
   @Override
   public void onReceive(Object message) throws Throwable {
       if (message instanceof InitMessage) {
           InitMessage initMessage = (InitMessage) message;
           this.collector = initMessage.getCollector();
       } else if(message instanceof NumMessage) {
           NumMessage numMessage = (NumMessage) message;
           int number = numMessage.getNumber();
           int res = number * number;
           this.collector.tell(new NumMessage(res), ActorRef.noSender());
       }
   }
}
class CollectorActor extends UntypedActor {
   @Override
   public void onReceive(Object message) throws Throwable {
       if (message instanceof NumMessage) {
           NumMessage numMessage = (NumMessage) message;
           System.out.println(numMessage.getNumber());
       }

   }
}
// -- MAIN --------------------------------------------------
public class StartAkka {
   public static void main(String[] args) throws InterruptedException {
       final ActorSystem system = ActorSystem.create("StartAkka");

       // -- SPAWN PHASE ----------
       final ActorRef Dispatcher = system.actorOf(Props.create(DispatcherActor.class),
"Dispatcher");
       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");

       // --- INIT PHASE ----------
       Dispatcher.tell(new InitMessage(Odd, Even, Collector),ActorRef.noSender());

       // -- COMPUTE PHASE ----------
       for(int i = 1; i <= 10; i++) {
           Dispatcher.tell(new NumMessage(i),ActorRef.noSender());
       }

       try {
           System.out.println("Press return to inspect...");
           System.in.read();
           System.out.println("Press return to terminate...");
           System.in.read();
       } catch(IOException e) {
           e.printStackTrace();
       } finally {
           system.shutdown();
       }
   }
}
```

Subquestion 6.2

After running the program, I get the following results:

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

Resources

[1] - learnIT slides
[2] - Brian Goetz and Tim Peierls, ‘Java concurrency in practice’, 2006;
[3] - Joshua Bloch, ‘Effective Java’, Second Edition.