Examination, Practical Concurrent and Parallel Programming
10–11 January 2017

These exam questions comprise 9 pages; check immediately that you have them all.

The exam questions are handed out in digital form from LearnIT and from the public course homepage on Tuesday 10 January 2017 at 09:00 local time.

Your solution must be handed in no later than **Wednesday 11 January 2017 at 15:00** according to these rules:

- Your solution must be handed in through LearnIT.
- Your solution must be handed in as a single PDF file, including source code, written explanations in English, tables, charts and so on, as further specified below.
- Your solution must have a standard ITU front page, available at http://studyguide.itu.dk/SDT/Your-Programme/Forms

There are 6 main questions. For full marks, all these questions must be satisfactorily answered.

If you find unclarities, inconsistencies or misprints in the exam questions, then you must describe these in your answers and describe what interpretation you applied when answering the questions.

Your solutions and answers must be made by you and you only. This applies to program code, examples, tables, charts, and explanatory text (in English) that answer the exam questions. You are not allowed to create the exam solutions as group work, nor to consult with fellow students, pose questions on internet fora, or the like. You are allowed to ask for clarification of possible mistakes, misprints, and so on, in the course LearnIT discussion forum. You should occasionally check the forum for news about mistakes and unclarities.

Your solution must contain the following declaration:

> **I hereby declare that I have answered these exam questions myself without any outside help.**
> (name) (date)

When creating your solution you are welcome to use all books, lecture notes, lecture slides, exercises from the course, your own solutions to these exercises, internet resources, pocket calculators, text editors, office software, compilers, and so on.

You are **of course not allowed to plagiarize** from other sources in your solutions. You must not attempt to take credit for work that is not your own. Your solutions must not contain text, program code, figures, charts, tables or the like that are created by others, unless you give a complete reference, describing the source in a complete and satisfactory manner. This holds also if the included text is not an identical copy, but adapted from text or program code in an external source.

You need not give a reference when using code from these exam questions or from the mandatory course literature, but even in that case your solution may be easier to understand and evaluate if you do so.

If an exam question requires you to define a particular method, you are welcome to define any auxiliary methods that will make your solution clearer, provided the requested method has exactly the result type and parameter types required by the question. Similarly, when defining a particular class, you are welcome to define auxiliary classes and methods.

The exam will be followed by a **cheat check** ("snydtjek"): The study administration randomly selects 20 percent of students; the list will be published on the course LearnIT page at 15:00 on Wednesday. The selected students must present themselves at Peter's office 4D15 on Thursday 12 January at 11:00. Each will be questioned for 5 minutes about his/her own exam handin. (This is only to discover possible cheating, and otherwise does not influence the grade). Students who sit the exam from some remote location must be available for interviews via Skype, and must seek permission for this beforehand (and send contact request to Skype handle "sestoft" with an explanation).

What to hand in Your solution should be a short report in PDF consisting of text (in English) that answers the exam questions, with relevant program fragments shown inline or supplied in appendixes, and a clear indication which code fragments belong to which answers. You may need to use tables and charts and possibly other figures. Take care that the program code retains a sensible layout and indentation in the report so that it is readable.

Computing data point clusters

Questions 1 through 4 of this exam concern the grouping of data points into clusters. Here we consider points (x, y) in the 2-dimensional plane. We are given a set $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ of such points and a desired number k of clusters, or point sets. The goal is to group the points into k clusters $S_1, \ldots, S_k$ such that each point (x_i, y_i) belongs to exactly one cluster S_j: the one whose mean (also called centroid) is closest to the point.

The *k-means algorithm* is a classical way to compute such a clustering, though not necessarily an optimal one. The algorithm maintains k tentative clusters $S_1, \ldots, S_k$, where each cluster S_j has a *mean* denoted m_j. The algorithm has an initialization step followed by a number of iterations:

- Randomly select k of the given points as the initial means m_j of the k clusters.
- Iterate these substeps until the cluster assignment does not change:
  - **Assignment step:** For each point (x_i, y_i) find the cluster S_j whose mean m_j is closest to the point, and assign the point to that cluster.
  - **Update step:** For each cluster S_j, recompute its mean $m_j = (mx_j, my_j)$ as the coordinate-wise average of all the points in the cluster. Thus $mx_j = \frac{1}{n_j} \sum_{(x_i,y_i) \in S_j} x_i$ where n_j is the number of points in cluster S_j. Similarly, my_j is the average of the y-coordinates.

The substeps assignment, update, assignment, update, ... are performed until the clustering has converged. The clustering has converged when each cluster's mean remains the same from iteration to iteration. However, computer floating-point arithmetics results depend on summation order, so it is safer to require only that the mean changed by less than ε since last iteration, where ε is a small positive number such as 10^{-10}.

Here is a Java implementation of the k-means algorithm (from class KMeans1 in file TestKMeans.java):

```
Cluster[] clusters = GenerateData.initialClusters(...);
boolean converged = false;
while (!converged) {
  iterations++;
  { // Assignment step: put each point in exactly one cluster
    for (Point p : points) {
      Cluster best = null;
      for (Cluster c : clusters)
        if (best == null || p.sqrDist(c.mean) < p.sqrDist(best.mean))
          best = c;
      best.add(p);
    }
  }
  { // Update step: recompute mean of each cluster
    ArrayList<Cluster> newClusters = new ArrayList<>();
    converged = true;
    for (Cluster c : clusters) {
      Point mean = c.computeMean();
      if (!c.mean.almostEquals(mean))
        converged = false;
      if (mean != null)
        newClusters.add(new Cluster(mean));
      else
        System.out.printf("===> Empty cluster at %s%n", c.mean);
    }
    clusters = newClusters.toArray(new Cluster[newClusters.size()]);
  }
}
```

The outer while-loop iterates until the clustering converges.

Source code for the questions File TestKMeans.java contains:

- Class TestKMeans for running the various algorithms and measuring their wall-clock time usage.
- Class KMeans1, with a working sequential implementation of the k-means algorithm (shown above). It has a nested class Cluster for representing clusters.
- Class KMeans2, with another working sequential implementation of the k-means algorithm (shown further below). It has its own nested class Cluster.
- Class KMeans3, with an incomplete stream-based sequential implementation of the k-means algorithm (shown further below). It has its own nested class Cluster.
- Class Point for representing, printing and approximate comparison of immutable 2D points.
- Abstract class ClusterBase with methods for printing and approximate equality of clusters. All Cluster classes must have this superclass.
- Class GenerateData for generating test data: point sets and an initial clustering. For this exam, we generate $n = 200000$ points and try to group them into $k = 81$ clusters.
- Class Timer for simple wall-clock time measurements.

You should **not** change class Point, ClusterBase, GenerateData or Timer in any way.

The k-means algorithm description above and the example implementations should suffice for the exam questions. If you want more background you may consult Wikipedia, a textbook chapter or Victor Lavrenko's Youtube video at `https://www.youtube.com/watch?v=_aWzGGNrcic` with an example for $k = 2$ at time 4:25.

Parallelization Since each iteration of the k-means algorithm strongly depends on the result of the previous one, the algorithm as a whole cannot be parallelized. However, each step can perform many operations in parallel:

- In the assignment step, we can create p parallel tasks, for instance $p = 8$, and let each task assign 25,000 of the 200,000 points to their closest cluster; the closest cluster of one point does not depend on that of any other point. Task 0 would take care of points $(x_0, y_0), \ldots, (x_{24999}, y_{24999})$, task 1 would take care of points $(x_{25000}, y_{25000}), \ldots, (x_{49999}, y_{49999})$, and so on, very much as in the lecture 1 TestCountPrimes example.
- In the update step, computing the mean of cluster S_j can be done in parallel with computing the means of all other clusters; the mean of one cluster does not depend on that of any other cluster. So we might create a parallel task t_j for each cluster S_j to compute its mean.

Question 1 (20 %):

The k-means algorithm implementation in method `findClusters` from class KMeans1 holds an array `points` of the 200,000 points to be clustered, and an array `clusters` of the 81 clusters.

Each cluster is an instance of nested class Cluster, which has an array list of the points belonging to the cluster, and an immutable mean. A newly created cluster has a given mean and initially no points. A point `p` may be added to cluster `c` by calling `c.add(p)` as illustrated in the assignment step code above. When all points have been added, the actual mean of the points may be computed by calling `c.computeMean()` as illustrated in the update step code above. If the new mean is different from the immutable mean stored in the existing cluster, then a new Cluster is created with the new mean and (so far) no assigned points.

In this question you must create a new class KMeans1P that is a parallelized version of KMeans1.

Use tasks and executors, creating a single ExecutorService before the cluster-finding while loop and then create tasks separately in the assignment step and the update step. You may use an array list of Runnable or Callable<Void> in the assignment step, and an array of Callable<Cluster> in the update step (because each task in the update step produces a new cluster).

1. Run k-means clustering using the KMeans1 class by compiling and running the given file. Show the execution time and the number of iterations used, along with the information produced by benchmark method `SystemInfo` for your system. You do not need to use any of the "`Mark`" benchmarking methods.

   Hint for later: The list of cluster means printed by KMeans1 can be taken as the "correct" ones for the remaining questions, except that the computed coordinates may vary in the last few digits. Also, the clusters may be printed in different orders.

2. Now start making class KMeans1P, as a copy of KMeans1. Show how to parallelize the assignment step as outlined above, using $p = 8$ tasks, each handling $1/8$ of the 200,000 points. Show the resulting code, explain what you are doing and why.

3. Show how to parallelize the update step as outlined above, using one task for each cluster. Show the resulting code, explain what you are doing and why.

4. Show how to make the nested Cluster class threadsafe using locking. Show the resulting code, explain the changes you make and why they are sufficient and necessary to make the class threadsafe for use in KMeans1P.

   Do not change the types or modifiers of the `points` and `mean` fields in Cluster.

5. Show the execution time for k-means clustering using KMeans1P and the number of iterations used, along with the information produced by benchmark method `SystemInfo` for your system. You do not need to use any of the "`Mark`" benchmarking methods.

6. Show the means of the first 5 clusters as printed by `km.print()`.

7. Now remove the threadsafety features you added to class Cluster in question 4, run KMeans1P, and report what happens. Explain the result.

Question 2 (20 %):

Consider now class KMeans2 which uses a different representation of the clusters. As before there is an array `points` of the 200,000 points to be clustered, but now there is also an array `myCluster` of the 200,000 clusters to which the points belong. The idea is that point `points[pi]` belongs to cluster `myCluster[pi]`.

Each cluster is an instance of a new nested class Cluster, which has a mutable `mean` field, mutable fields `sumx` and `sumy` in which to accumulate the sum of the x_i and y_i coordinates of points (x_i, y_i) assigned to the cluster, and a mutable field `count` to hold the number of such points.

The assignment step updates the `myCluster` array for each point index `pi`. The update step first sets the fields `sumx`, `sumy` and `count` of all clusters to zero, then adds each point `points[pi]` to its cluster `myCluster[pi]`'s sum fields, and finally computes the new mean for each cluster. The latter operation will update the existing cluster's `mean` field and return true if the cluster has converged, that is, if the mean remains unchanged.

The KMeans2 version of the core algorithm for finding clusters is shown below:

```
final Cluster[] clusters = GenerateData.initialClusters(...);
final Cluster[] myCluster = new Cluster[points.length];
boolean converged = false;
while (!converged) {
  iterations++;
  {
    // Assignment step: put each point in exactly one cluster
    for (int pi=0; pi<points.length; 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;
    }
  }
```

```
    {
      // Update step: recompute mean of each cluster
      for (Cluster c : clusters)
        c.resetMean();
      for (int pi=0; pi<points.length; pi++)
        myCluster[pi].addToMean(points[pi]);
      converged = true;
      for (Cluster c : clusters)
        converged &= c.computeNewMean();
    }
    // System.out.printf("[%d]", iterations); // To diagnose infinite loops
  }
```

In this question you must create a new class KMeans2P that is a parallelized version of KMeans2.

Use tasks and executors, creating a single ExecutorService before the cluster-finding while loop and then create tasks separately in the assignment step and the update step. You may use an array list of Runnable or Callable<Void> for the tasks in each of the two steps.

The assignment step may be parallelized roughly as in KMeans1P. The update step, however, cannot be parallelized as in KMeans1P because there is no efficient way to find the points belonging to a given cluster. Instead, one can use the same approach as for the assignment step: create a small number (say, 8) of tasks and distribute the work of processing the `points` array over those tasks.

1. Show the execution time for k-means clustering using the KMeans2 class (as given to you) and the number of iterations used, along with the information produced by benchmark method `SystemInfo` for your system. You do not need to use any of the "`Mark`" benchmarking methods.

2. Now start making class KMeans2P, as a copy of KMeans2. Show how to parallelize the assignment step, using $p = 8$ tasks, each handling $1/8$ of the 200,000 points. Show the resulting code, explain what you are doing and why.

3. Show how to parallelize the update step as outlined above, using $p = 8$ tasks. Show the resulting code, explain what you are doing and why.

4. Show how to make the nested Cluster class in KMeans2P threadsafe using locking. Show the resulting code, explain the changes you made and why they are sufficient and necessary to make the class threadsafe for use in KMeans2P.

   Do not change the types or modifiers of the `mean`, `sumx`, `sumy` and `count` fields in Cluster.

5. Show the execution time for k-means clustering using KMeans2P and the number of iterations used, along with the information produced by benchmark method `SystemInfo` for your system. You do not need to use any of the "`Mark`" benchmarking methods.

6. Show the means of the first 5 clusters as printed by `km.print()`.

7. Now remove the threadsafety features you added to class Cluster in question 4, run KMeans2P, and report what happens. Explain the result.

8. In KMeans2 and KMeans2P, the assignment step stores the `best` cluster reference in `myCluster[pi]`, and later the update step performs `myCluster[pi].addToMean(p)`.

   It seems simpler to just do `best.addToMean(p)` in the assignment step, so that the update step's only work is to compute the new means for the 81 clusters. Then one could get rid of the `myCluster` array.

   Would this approach be threadsafe if it were implemented as a change to KMeans2P? Explain why or why not.

9. Now make a new class KMeans2Q that implements the simpler approach discussed in the preceding subquestion. Show the resulting code (all of method `findClusters`).

10. Show the execution time for k-means clustering using KMeans2Q and the number of iterations used, along with the information produced by benchmark method `SystemInfo` for your system.

    Which is faster on your machine, KMeans2P or KMeans2Q?

Question 3 (10 %):

In this question you must create a new class KMeans2Stm that is a parallelized version of KMeans2.

Use tasks and executors as in KMeans2P, but make nested class Cluster threadsafe by using software transactional memory (as implemented by the Multiverse library) instead of locking.

The parallelization of the assignment step and the update step should remain exactly as before. Only the nested Cluster class should change.

1. Show how to make the nested Cluster class threadsafe using software transactional memory as implemented by the Multiverse library. Show the resulting code, explain the changes you made and why they are sufficient and necessary to make the class threadsafe for use in KMeans2Stm.

2. Show the execution time for k-means clustering using KMeans2Stm and the number of iterations used, along with the information produced by benchmark method `SystemInfo` for your system. You do not need to use any of the "`Mark`" benchmarking methods.

3. Show the means of the first 5 clusters as printed by `km.print()`.

4. Now remove the threadsafety features you added to class Cluster in question 1, run the program, and report what happens. Explain the result.

Question 4 (15 %):

The k-means algorithm can also be implemented using Java 8 streams in both the assignment step and the update step. Class KMeans3 contains a sketch of such an implementation; you must finish it.

Naturally, using streams opens the possibility of easily parallelizing each step; this you will be asked to do also, in a new class KMeans3P.

The k-means clustering algorithm can be implemented using streams (sequentially) as follows:

- A Cluster (nested class in the KMeans3 code sketch) has an unmodifiable arraylist of the points belonging to the cluster, and an immutable mean. A newly created cluster holds both a list of points and a mean, which may not be the mean of the points in the list (unless the clustering has converged); see below.

- The assignment step can be implemented as follows:

  Create a stream of all the given (200,000) points, and group the points by the cluster whose mean is closest to the point, using a stream `groupingBy` collector. The result is a map from Cluster to a list of the Points closest to that cluster. For a map entry `kv`, the key `kv.getKey()` is an old cluster and the value `kv.getValue()` is the list of points for which that cluster's mean was the closest one.

  Then create a stream of the set of entries of that map, and transform (using `map`) each entry `kv` into a new Cluster with the old cluster's mean and the new list of points. The result is a stream of temporary clusters, which should then be turned into an array of temporary clusters (whose means are not yet updated).

- The update step can be implemented as follows: Create a stream of clusters from the array, transform (using `map`) each temporary cluster into a new one whose mean is that of its list of points; then turn the resulting stream of clusters into an array. If the temporary clusters and the new clusters are equal then the algorithm has converged.

A sketch of the KMeans3 stream-based version of the core algorithm for finding clusters is shown below:

```
Cluster[] clusters = GenerateData.initialClusters(...);
boolean converged = false;
while (!converged) {
  iterations++;
  { // Assignment step: put each point in exactly one cluster
    final Cluster[] clustersLocal = clusters;  // For capture in lambda
    Map<Cluster, List<Point>> groups = ... TODO ...
    clusters = groups.entrySet().stream().map(...) ... TODO ...
  }
  { // Update step: recompute mean of each cluster
```

```
      Cluster[] newClusters = ... TODO ...
      converged = Arrays.equals(clusters, newClusters);
      clusters = newClusters;
    }
  }
```

1. Complete the stream-based assignment step code for KMeans3 shown above.
2. Complete the stream-based update step code for KMeans3 shown above.
3. Show the execution time for k-means clustering using the KMeans3 class and the number of iterations used, along with the information produced by benchmark method `SystemInfo` for your system. You do not need to use any of the "`Mark`" benchmarking methods.
4. Create a new class KMeans3P in which you parallelize the stream-based assignment step you created above. Show the resulting code, explain what you are doing and why.
5. Likewise parallelize the update step as outlined above. Show the resulting code, explain what you are doing and why.
6. Show the execution time for parallel k-means clustering using the KMeans3P class and the number of iterations used, along with the information produced by benchmark method `SystemInfo` for your system. You do not need to use any of the "`Mark`" benchmarking methods.
7. Do you need to make any changes to the nested Cluster class to make it threadsafe for use inside KMeans3P? Explain why or why not.
8. Summarize the execution times of the KMeans1, KMeans1P, KMeans2, KMeans2P, KMeans2Q, KMeans2Stm, KMeans3 and KMeans3P algorithms in a table. Discuss the results.

Question 5 (20 %):

(This question is unrelated to the preceding ones). File TestMSQueueNeater.java declares an interface UnboundedQueue<T> that describes an unbounded and nonblocking queue of items of type T:

```
interface UnboundedQueue<T> {
  void enqueue(T item);
  T dequeue();
}
```

The same file also declares a class MSQueueNeater<T> that implements the interface using the lock-free Michael-Scott approach. However, the `enqueue` and `dequeue` methods have been simplified and rewritten in a possibly clearer style, different from that of the paper studied in the course.

Note that the Michael-Scott queue is nonblocking: the `dequeue` method returns null when the queue is empty, instead of blocking until the queue becomes non-empty (as a blocking queue would do).

You must write sequential and parallel tests to investigate whether this version of the Michael-Scott queue works.

1. Write a precise sequential test that makes it plausible that the MSQueueNeater data structure works when manipulated by a single thread.
2. Write an approximate concurrent test that makes it plausible that MSQueueNeater works when used by 20 concurrent threads: 10 producer threads that enqueue elements, and 10 consumer threads that dequeue elements.
3. Make mutations to the `enqueue` and `dequeue` methods that examine the strength of your concurrent test.

   Make several mutations that cause the concurrent test to fail in three different ways (eg. by not terminating, by throwing a null reference exception, or by throwing a test assertion exception from a method in class Tests).

   For each attempted mutation you must describe the code change, what effect you think the mutation might have, what observed effect it actually does have on the concurrent test (if any), and reflections on why the mutation had that observed effect (or no effect).

Question 6 (15 %):

(This question is unrelated to the preceding one). File start.erl contains an Erlang specification of a system that has three distinct phases: spawn, init, and compute, as shown in the communication diagram in Figure 1.

In the *spawn phase* (the top red part of the communication diagram), the start actor spawns the four actors involved in the system: a dispatcher actor, two worker actors called odd and even, and a collector actor. In a real system, there is likely more than $N = 2$ worker actors.

In the *init phase* (the middle green part of the communication diagram), the start actor sends an initialization message (containing the references of the actors odd, even, and collector) to the dispatcher actor. Upon reception of this initialization message, the dispatcher actor saves the references of the odd and even actors and forwards the initialization message (containing the reference of the collector actor) to the odd and even actors. (Hint: In Java, you are welcome to reuse the exact same initialization message superfluously including the references to the odd and even actors.) When the worker actor (i.e., either the odd or even actor) receives an initialization message, it will simply save the reference of the collector.

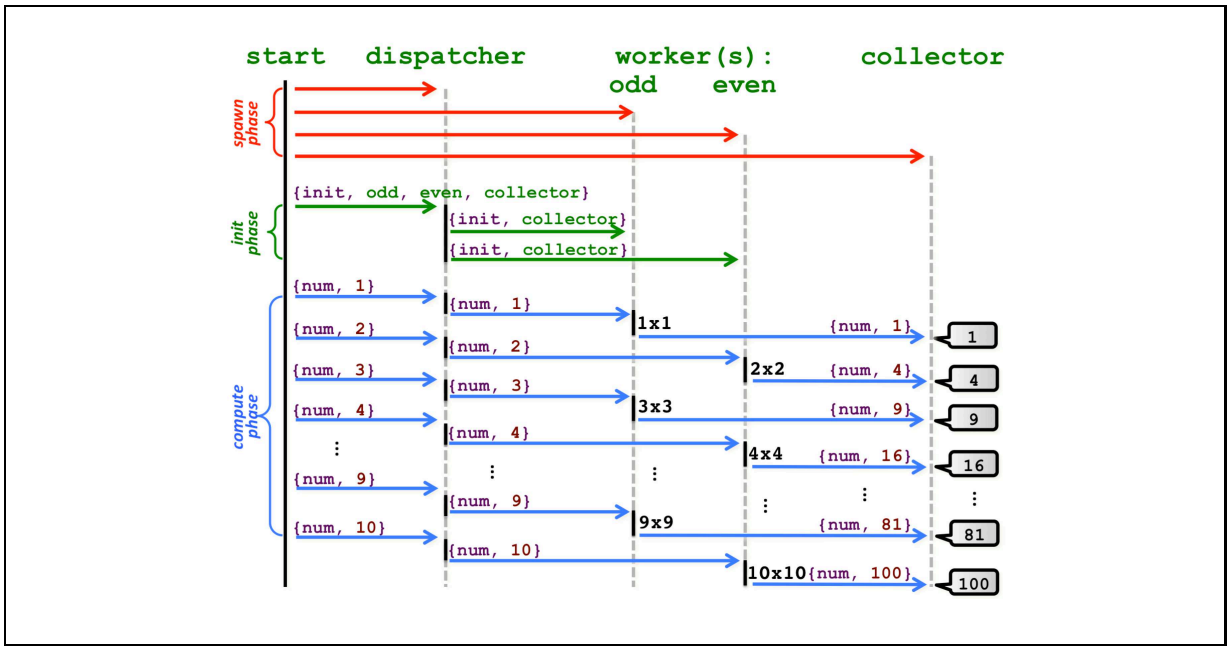

Figure 1: The three phases of the system shown by a communication diagram.

In the *compute phase* (the bottom blue part of the communication diagram), the start actor will repeatedly send number messages (containing successively the integers from 1 to 10) to the dispatcher actor. Upon reception of a number message, the dispatcher actor will determine whether the number is odd or even and then dispatch (forward) it to the appropriate actor (i.e., either to the odd or even actor). When a worker actor (i.e., either the odd or even actor) receives a number message, it will compute the square of the number and forward it to the collector. Finally, when the collector actor receives a number message, it will simply print the number to standard output. The system's communication during the compute phase is shown in Figure 2. In a real system, the computation performed by a worker is usually more involved than computing the square of a number, and the collector will, in fact, collect or aggregate the results instead of just printing them.

1. Implement the system in Java+Akka (as close to the Erlang version as reasonably possible).
2. Run your system to demonstrate that it produces $\{1, 4, 9, 16, \ldots, 100\}$ in some order. The order may vary according to the scheduling between the odd and even actors; in other words, the resulting numbers may not necessarily be printed in increasing order.

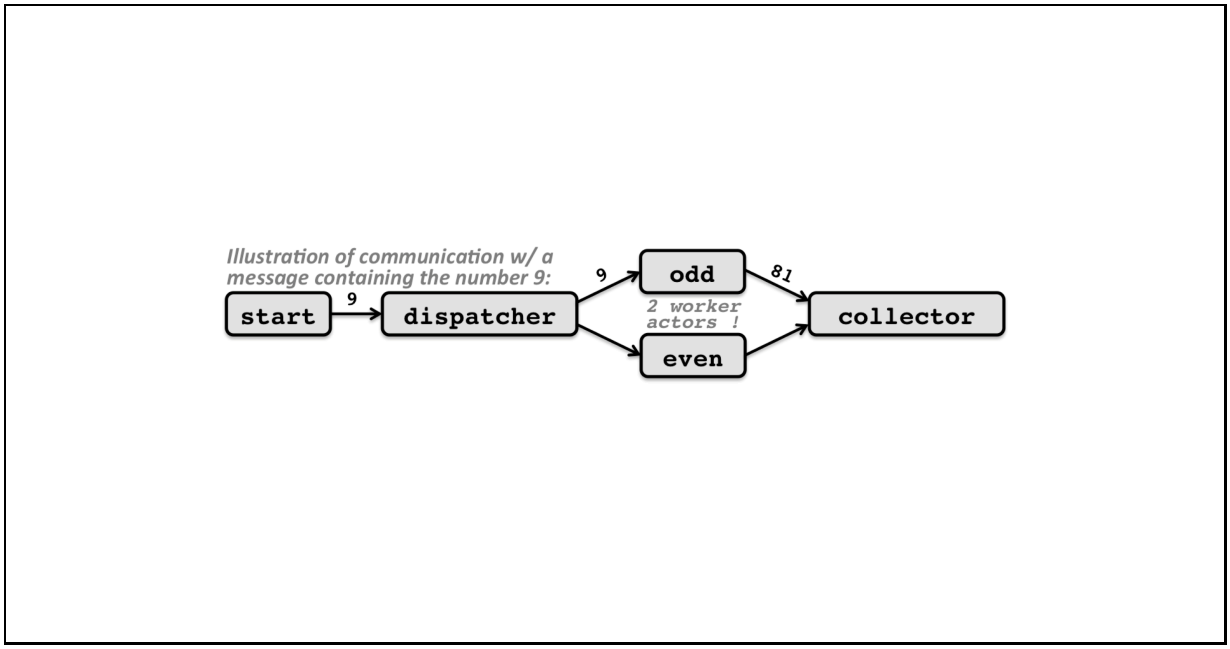

Figure 2: Communication during the compute phase.