Re-Examination, Practical Concurrent and Parallel Programming
25-28 March 2019

These exam questions comprise 8 pages; check immediately that you have them all.
The exam questions are handed out in digital form from LearnIT
Your solution must be handed in no later than **Monday 25 March 2019 at 14:00** according to these rules:

- Your solution must be handed in through LearnIT.
- Your solution must be handed in as a single PDF file, including cited 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` (more precisely the exam standard front page) You need to fill out one line about you and the name of the course as PCPP.
- Additionally, your complete source code is to be handed in as one `zip` file.

There are 15 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, by private email to `rikj@itu.dk` with a CC: to `sap@itu.dk`. Expect the answer in the course LearnIT news forum. You should occasionally check the forum for news about clarifications to the exam set.

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.

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. 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. Additionally you must hand in your operational source code in a directory structure that shows which question the code is relevant for as one zip file on learnit.

1 Accounting System

In this question we consider an accounting system. In this system all the accounts are represented as an array of integers corresponding to their current balance. The system supports transferring between two accounts, depositing into an account, summing all the balances and transferring all the balances from one Accounts object to another. In this question, the balance of an account is allowed to be negative.

The one really important feature of the system is that no money gets lost, i.e., a transfer between two accounts may under no circumstances change the sum of the balances, and deposits may not be lost.

Concretely, you should implement the following interface which is specified in `accounting/Accounts.java`:

```
interface Accounts {
    // (Re)initializes n accounts with balance 0 each.
    public void init(int n);

    // Returns the balance of account "account"
    public int get(int account);

    // Returns the sum of all balances.
    public int sumBalances();

    // Change the balance of account "to", by amount;
    // negative amount is allowed (a withdrawel)
    public void deposit(int to, int amount);

    // Transfer amount from account "from" to account "to"
    public void transfer(int from, int to, int amount);

    // Transfer all the balances from other to this accounts object.
    public void transferAccount(Accounts other);
}
```

The balances of the individual accounts are initially 0.

There is a naive, sequential, not thread safe reference implementation in the file `accounting/UnsafeAccounts.java`. Furthermore the file `accounts/Runner.java` contains a sequential test for testing implementations of the `Accounts.java` interface.

All of your code needs to have a focus on performance. In particular, you should avoid unnecessary indirections.

Question 1.1

Write a concurrent test in `Runner.java` that shows that the `accounting/UnsafeAccounts.java` implementation is prone to race conditions.

Show output of your test and argue or explain why it is an error, i.e., what would be expected from a correct implementation.

Question 1.2

Write a concurrent test in `Runner.java` that shows that the `accounting/UnsafeAccounts.java` implementation has visibility issues.

Show the output of your test and argue or explain why it is an error, i.e., what would be expected from a correct implementation.

Question 1.3

Create a new class called `LockAccounts.java` which implements the `Accounts.java` interface. This implementation should be thread safe by using locks. You may use as many locks as you find useful.

- Make sure that if two operations concern disjoint sets of accounts, the transfer and deposit operations should not block each other.
- Implement the sum of balances as one loop over all accounts, risking inconsistent results.
- Show that this version passes the tests from Question 1.1 and 1.2.
- Argue that your solution can not deadlock
- Explain a situation where the reported sum could be inconsistent.

Question 1.4

Often a sum over a number of variables will either have one variable storing the sum of balances or would iterate over all the accounts to compute it on demand. The former is a concurrency bottleneck, the latter is computationally expensive.

Here, we consider a variant that avoids both disadvantages. It is based on an array of integers `sums` representing partial sums, and a lock for each of these integers. The length of the array is roughly the number of threads that are expected to use the data structure, each thread writes and reads from one of the partial sums, based on its hash value (modulo the length of the array). To report the overall sum, sum over the array of partial sums, while holding all the corresponding locks, such that no other operations can interfere.

Implement this in a new class called `LockAccountsFast.java`.

Question 1.5

Create a new class called `STMAccounts.java` which implements the `Accounts.java` interface. Implement a thread safe version that uses Software Transactional Memory, with the multiverse library.

- Show that this version passes the tests from Question 1.1 and 1.2.
- `sumBalances()` can be implemented by either risking inconsistent results or by risking being competing with very many other method calls. Implement one of these choices and explain the other.

Question 1.6

Create a new class called `CASAccounts.java` which implements the `Accounts.java` interface. Implement a thread safe version that is based on compare and set (CAS).

- In this implementation `sumBalances()` must be consistent, which can be achieved by adding single variable `sum` which represents the global sum, which is updated in each deposit and `transferAccount()` method.
- Each operation does not need to be entirely atomic, but no writes can be lost (i.e. the final result of applying multiple operations should be the same as the serial implementation).
- Show that this version passes the tests from Question 1.1 and 1.2.
- Argue for the progress guarantees that this implementation achieves, i.e. does it livelock?
- Does your implementation achieve constant time operations?

Question 1.7

In this question, we focus on a scenario where we wish to apply a stream of transfers and deposits (denoted Transactions) to an account in parallel. You can find an implementation of a Transaction in `accounts/Transaction.java`. Notice that if a Transaction's `from` field is -1 then it is considered a deposit, whereas if it is $[0, 1, ..., n-1]$ it is a transfer. Note that the order in which transactions are processed is irrelevant because we allow negative balances. Observe that in this context, the created transactions are pseudorandom and deterministic in the following sense: We produce transactions by creating indices with `range`, that are turned into a transaction by the constructor that takes the index as a seed for the pseudo random generator. This process is deterministic and will produce the same sequence every time it is called, but there is no intended structure or correlation between the different transactions in the same stream.

At the bottom of `accounts/Runner.java` you can see two methods which generate transactions, and have space for your implementations of the following two questions.

Question 1.7.1 Direct Stream Based

In this question you must implement the described functionality by using any of the above thread safe implementations and Java's stream library. You should use a stream to distribute the list of transactions to multiple threads and apply each transaction on the shared concurrent data-structure in parallel.

Show your input, i.e. number of accounts and number of transactions. Print the balances of all accounts (at least for small n), and the result of calling `sumBalance()` after applying all the transactions.

Question 1.7.2 Stream Collect Based

In this question we implement an alternative parallel solution using the stream `collect` operator which can compute the result of applying all transactions on a common data-structure. Hint: you may need to use the `transferAccounts()` method to solve this.

- Implement and ensure that you get the same results as in 1.7.1.
- Does your implementation give the correct results when using `UnsafeAccounts.java`? Why?
- Show your input, i.e., number of accounts and number of transactions.
- Print the balances of all accounts (at least for small n), and the result of calling `sumBalance()` after applying all the transactions.

Question 1.7.3 Compare the performance of the different implementation

Benchmark the implementation from 1.7.1 and 1.7.2. (and the serial `UnsafeAccounts`). Explain your choice of type of inputs and parameters. You can for example show the effect of varying the number of threads, the number of transactions and/or the number of bank accounts.

Report the results and reflect on what you see. Which method is the fastest, and why? Which is slowest, and why? Discuss if there might be a concurrent benchmark where another implementation would be the fastest?

2 Buffered Merging Priority Queue

The following questions are centered around sorting the k-smallest elements of an input of length n, here usually for $k = \frac{n}{4}$. We use a construction that can be seen as something like merge sort, heap sort or using a priority queue, depending on how you look at it.

A simple priority queue supports the operations specified by the interface given in `bufferedMergePQ/PQ.java`, shown here:

```
public interface PQ {
    // check if the PQ is empty
    boolean isEmpty();

    // get the value of the minimum element without removing it
    int peek();

    // get and remove the minimum element.
    int getMin();

    // print the queue.
    void print(int depth);

    // needed for stopping threads in BoundedBufferThreadMerge
    default void shutDown() {
    };

    // needed for starting threads in BoundedBufferThreadMerge
    default void start() {
    };
}
```

We consider a special version of the priority queue, namely a buffered recursive implementation that is based on merging. In the file `bufferedMergePQ/BufferedPQ.java` (together with `bufferedMergePQ/Parameters.java` and `bufferedMergePQ/SerialPQPair.java`) a sequential implementation of this data structure can be seen.

It might be useful to look at this implementation while reading the following explanations.

The data structure consists of a *buffer* of the currently smallest elements and two recursive data structures, that initially hold half of the elements each.

As long as the buffer is not empty, `getMin()` and `peek()` can be immediately answered. If the buffer becomes empty, we fill it by merging the minimum elements the two recursive data structures. Such a merge is done by repeatedly taking the smallest between the minimum element of the left and the right data structures, until the buffer has the desired number of elements.

One interesting feature of this construction is that several of the steps can be done in parallel, as detailed in the following questions.

Question 2.1: Warmup: Benchmarking the Naive Implementation

This question is meant to introduce you to the structure of the code used in the following questions.

Using the benchmarking framework in `bufferedMergePQ/Runner.java`, benchmark the naive solution in `OneBufferPQ.java` with or without parallel sorting, as expressed by the `sortParallel` flag of the `Parameters`. Compare performance to the serial implementation `BufferedPQ.java` using the serial creation of the recursive data structures expressed in `SerialPQPair.java`.

Experiment with `cutOff` (size of the smallest data structure at the base), `bufLen` and with different size n of the data structure.

Note that the only changes to code (if any) are in `Runner.java`. Report your results and reflect on reasons for the performance differences that you see.

Question 2.2: Parallel Initialisation

Implement a parallel version of `SerialPQPair` that performs the creation of the two recursive priority queues in parallel. Implement this in `ParallelPQPair.java` implementing the interface `PQPair`.

Question 2.3: Parallel Filling of the Buffer

In this question we add more task parallism to the buffered priority queue. The core idea is to introduce a new buffer `nextBuffer` which will be filled up concurrently while other threads can take elements from the current `buffer`. When the `buffer` needs to be filled, we can simply use the concurrently computed `nextBuffer`.

Copy the file `BufferedPQ.java` and make your own `BufferedPQP.java` where you implement your solution to the question.

- Add a field `nextBuffer` to the priority queue.
- Augment `getMin()` to fill up `nextBuffer` concurrently, while elements from buffer can be removed using `getMin()`. This can for instance be done with an `ExecutorService`.
- Explain why in the setting where only one thread is calling `pq.getMin()` (at the root of the recursive data structures, like in `Runner.java`), but multiple threads are running internally, we still do not need any thread safety measures on the priority queue.

Question 2.4: Connected Bounded Buffers

In this question we try yet another way of solving the problem. The core idea is to have a thread for each buffer constituting the priority queue. Each thread repeatedly pulls from the buffers of the two recursive priority queues and places the items into its own buffer. The thread will wait if its buffers are currently empty or full.

There is a skeleton of such an implementation in `bufferedMergePQ/BoundedBufferThreadMerge.java`, which already has implemented parallelism mechanism, but is missing thread safety.

- Add the required code to achieve thread safety and avoid busy waiting, using locks.
- Add a concurrent test in `Runner.java` for which the given skeleton fails the test and your implementation passes.
- Make multiple mutations to your concurrency control mechanisms which breaks the concurrency tests. For each set of mutations write why your concurrency catches this problem.

Question 2.5: Overall Benchmarking

The last 4 questions introduced different parallel implementations of a simple priority queue. Benchmark these implementations for different parameters (similar to Question 2.1). Discuss your finding in comparison to expectations we can have based on observations about similar programs we looked at during the course. Note that the parallel versions are not guaranteed to run faster than the sequential on your machine.

Question 3: Message Passing

This question is independent of the rest of the questions of this exam.

Consider the following system specified in with Erlang in `messagePassingMergeSort/MergeSort.erl`, which sorts a list of numbers using a distributed version of merge-sorting. Besides start(/main), the system consists of **three actors**: **tester**, **sorter**, and **merger**.

START: The **start** actor spawns a **tester** actor, a **sorter** actor, and sends an *initialization* message to the **tester** containing the address of the **sorter**.

TESTER: Upon receiving an *initialization* message, the **tester** sends off a message to the **sorter**, instructing it to sort a list of numbers and send the result to the **tester** (on Figure 2, this is designated as "–[6,5,3,1,8,7,2,4]$_X$–>"). Upon receiving a *sorted list* response, the **tester** prints the result to standard output.

SORTER: When the **sorter** receives a list along with a recipient for the result (e.g., "–[6,5]$_B$–>"), it does the following (depending on the length of the list). If the list is *longer than one*: the **sorter** splits the list into two lists each half as long; it spawns a **merger** to eventually merge the two sorted half lists; it spawns to two **sorters** which will each receive a half list to sort along with instructions to send the result back to the newly spawned **merger**. If the list is *one element*: the **sorter** will immediately forward it onto the designated recipient.

MERGER: The merger first waits for a message designating a recipient (where to eventually send the result). Then, it awaits two sorted half lists. Once it has received these three messages, it merges the two sorted half lists into a long sorted list and sends off the result to the designated recipient.

Figure 1 shows three distinct runs of the `MergeSortSystem` and Figure 2 provides a visualization of how **Merge Sort** works algoritmically. For more information on Merge Sort, please consult Wikipedia.

```
Merged: 7 8
Merged: 5 6
Merged: 2 4
Merged: 1 3
Merged: 2 4 7 8
Merged: 1 3 5 6
Merged: 1 2 3 4 5 6 7 8
RESULT: 1 2 3 4 5 6 7 8
```

```
Merged: 2 4
Merged: 1 3
Merged: 5 6
Merged: 1 3 5 6
Merged: 7 8
Merged: 2 4 7 8
Merged: 1 2 3 4 5 6 7 8
RESULT: 1 2 3 4 5 6 7 8
```

```
Merged: 7 8
Merged: 2 4
Merged: 2 4 7 8
Merged: 1 3
Merged: 5 6
Merged: 1 3 5 6
Merged: 1 2 3 4 5 6 7 8
RESULT: 1 2 3 4 5 6 7 8
```

Figure 1: Three distinct runs of the MergeSortSystem

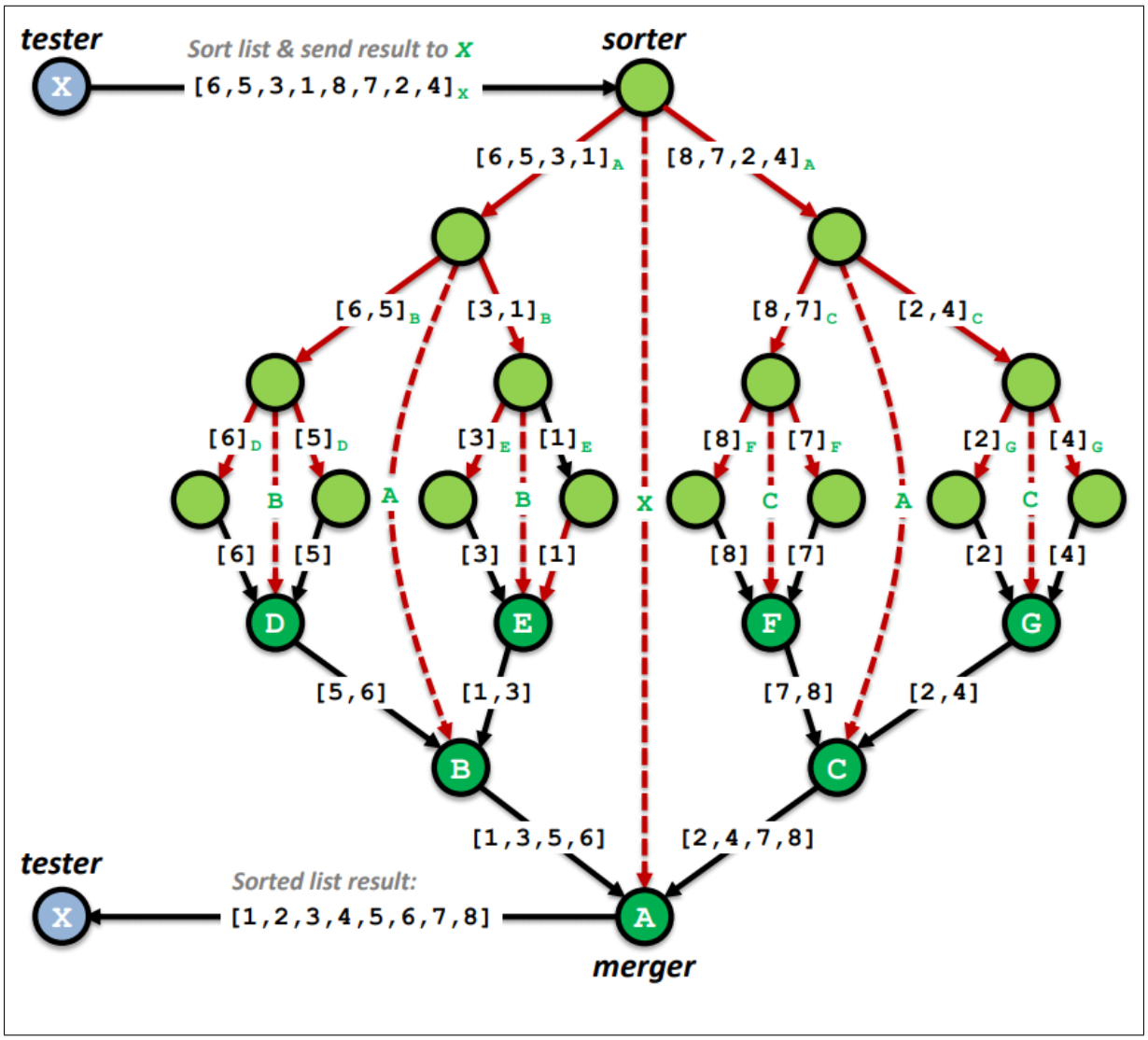

Figure 2: Communication diagram for the merge sort

Remember that you can find the Erlang specification of the system in `MergeSort.erl`.

- Implement a `JAVA+AKKA` version of this system as close to the ERLANG *specification* as realistically possible, while respecting the natural ways of programming in JAVA/AKKA (e.g., recursion is often naturally replaced by iteration in a language like JAVA).
- Test your system and document it by providing three distinct results of running your system.