Examination, Practical Concurrent and Parallel Programming
6-7 January 2020

These exam questions comprise 9 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 **Tuesday, January 7th 2020 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 a number of questions and sub-questions. For full marks, all these 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 `thdy@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 keeps 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.

What does it mean to "Show" and "Argue"? Many of the questions below ask you to "show" something. It may say "Show that your method works as expected" or "Show that this program has a race condition". In every case, you are expected to write tests that either convince the reader about the correctness or erroneousness of the code. Sometimes this may be a sequential test, other times it may require a concurrent test. If you are showing that something doesn't work, include outputs from your tests and explain why it is an error, i.e., what would be expected from a correct implementation.

The question may also say "Show that method A is faster than method B". In such a case you need to argue about performance, which is best done by making measurements and plots. You will have to decide which measurements, inputs and plots best and most honest communicate to the reader.

Finally, some question may ask you to "Argue" or "Explain" a certain point. In these cases, tests are typically not needed, though they may help, and you will instead have to convince the reader logically based on what you know about the correctness of concurrent programs.

1 Search Trees

This question is about parallel processing of a Binary Search Tree, which is a binary tree with the property that values are kept in sorted order.

More precisely:

Definition 1 (Binary Search Tree) *A Binary Search Tree is either a 'leaf' or a branch (v, t_l, t_r), where v is a value and t_l and t_r are binary search trees themselves. Furthermore, we have the search tree property ($\triangle$): If (v, t_l, t_r) is a branch in a tree, then all values in t_l are smaller than v and all values in t_r are larger than v.*

We can write such a tree in Java with the following code:

```
interface Node {
    Node insert(long value);
}
class Branch implements Node {
    long value; Node left, right;
    Branch(long value, Node left, Node right) {
        this.value = value; this.left = left; this.right = right;
    }
    public Node insert(long newValue) {
        if (newValue < value)
            return new Branch(value, left.insert(newValue), right);
        if (newValue > value)
            return new Branch(value, left, right.insert(newValue));
        return this;
    }
}
class Leaf implements Node {
    public Node insert(long newValue) {
        return new Branch(newValue, this, this);
    }
}
```

Note that a new value is inserted into the left sub-tree if it is smaller than the node value, and into the right sub-tree if it is larger. If the value is already in the tree it is ignored. (The tree is a collection without any duplicates, also known as a set.)

```
Node root = new Leaf();
root = root.insert(5);
root = root.insert(2);
root = root.insert(7);
root = root.insert(6);
```

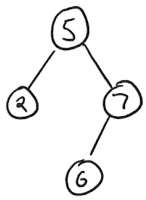

Figure 1: Example tree with 4 values.

Question 1.1 [10%]

1. Add a method `boolean contains(long value)` to `Node` (and `Branch` and `Leaf`) that determines whether the tree contains the given value. The code should take advantage of the search tree property ($\triangle$) to run fast.
2. Show that your method works sequentially. (See definition of "Show" above).
3. Show that the tree (given code + your code) is prone to race conditions.

   Hint: To update the functional tree from multiple threads you need to store root in some mutable holder object, like an `AtomicReference`.
4. Explain whether the code is correct if only one thread is calling insert?

Question 1.2 [15%]

1. Add a method `<V> V reduce(LongFunction<BiFunction<V, V, V>> f, V unit)` to `Node`, `Leaf` and `Branch`. [1]

   Calling `t.reduce(f, u)` on the tree in figure 1 should compute `root.reduce(f, u) = f(5, f(2, u, u), f(7, f(6, u, u), u))`. This is similar to the reduce on streams we covered in the class, but can be used to summarize trees.

   Show that it works by computing the sum of the values in the tree like this: `root.reduce(v -> (v1, v2) -> v + v1 + v2, 0L)`.
2. Show how to use `reduce` to compute

   (a) The height of the tree.

[1] Note that since Java has no `TriFunction` interface, we use `LongFunction<BiFunction<V, V, V>>` as a function that takes a **long** and two arguments of type `V` and returns a value of type `V`. This technique is known as "currying". Note that to call this function you will need to write `f.apply(a).apply(b,c)` rather than `f(a,b,c)`.

(b) The maximum value in the tree.

3. In order to get a parallel speed-up, we change the method signature to `<V> V reduce2(LongFunction<BiFunction<Callable<V>, Callable<V>, V>> f, V unit)`. Implement this method in the tree, and show that it works by computing the sum in parallel like this:

```
static ExecutorService executor = ...
static long sum(Node tree) {
    return tree.reduce2(v -> (s1, s2) -> {
        Future<Long> f1 = executor.submit(s1);
        Future<Long> f2 = executor.submit(s2);
        try {
            return v + f1.get() + f2.get();
        } catch (InterruptedException | ExecutionException e) {
            throw new RuntimeException(e);
        }
    }, 0L);
}
```

4. Use `reduce2` to implement a function `static long sumPositives(Node tree)`, which returns the sum of *all values greater than 0*. That is, if the tree contains $\{-3, 1, 4\}$, the function should return 5.

   Hint: To get the best performance, you should take advantage of parallelism, as well as property ($\triangle$) of binary search trees.

Question 1.3 [20%]

A different way of getting parallel reads is by using Java streams. An initial idea would be to implement a method in `Branch` like

```
public LongStream stream() {
    return LongStream.concat(left.stream(),
        LongStream.concat(LongStream.of(value), right.stream()));
}
```

(and respectively in `Node` and `Leaf`.)

1. Show and Explain why this approach fails to give any parallelism when we later call `root.stream().parallel().sum()`.

2. Instead recall spliterators from lecture 5. Add a method `Spliterator.OfLong spliterator()`, returning a custom implementation of `tryAdvance` and `trySplit` designed to take advantage of the tree recursive nature of the tree.

   You can get a parallel stream with `StreamSupport.longStream(root.spliterator(), true)`. Show that your code works, e.g. by using it to sum the values in a tree.

3. Make measurements comparing this approach with `reduce` and `reduce2` from question 1.2. Measure how the two solutions scale with the size of the tree, as well as the number of threads.

   Also try summing only the numbers in the tree that are prime numbers.

   Hint: Because of the problem of "balance", which we will discuss in the next section, to get proper performance from `reduce2`, it'll work best to insert only random numbers into the tree, rather than say the sequential numbers from 1 *to* 10^6.

2 Self-balancing binary search trees

The tree classes from section 1 had a major problem: The running time of `insert` could be $\Omega(n)$ if, say, the user would insert the numbers $1 - n$ in order. The standard computer science way to get this down to $O(\log n)$ time is by balancing the tree. See the Wikipedia page if you have never heard of this concept before.

In this question we will implement a self-balancing binary search tree known as a Treap. The idea is to add an extra field to all nodes called `long priority`. In addition to the search property, ($\triangle$), we now also enforce the "heap property" ($\bigcirc$)*: if node A is the parent of node B, then A must have priority smaller than or equal to B.* [2] Our main goal of this section will be finding a way to do insertions in parallel, while keeping the two properties invariant.

We add the following methods to `Node` and a new class called `Holder`, which just holds a reference to a node:

```
interface Node {
    Node insert(long newValue, long newPriority);
    Node[] split(long value);
}
class Holder {
    Node node;
    Holder(Node node) { this.node = node; }
    public void insert(long newValue, long newPriority) {
        node = node.insert(newValue, newPriority);
    }
}
class Leaf implements Node {
    public Node insert(long newValue, long newPriority) {
        return new Branch(newValue, this, this, newPriority);
    }
    public Node[] split(long value) {
        return new Node[]{this, this};
    }
}
```

The treap can be used like this

[2] This is similar to the idea of a Heap data structure, hence the name Treap.

```
Holder root = new Holder(new Leaf());
root.insert(5, ThreadLocalRandom.current().nextLong());
...
```

The idea of insertion is to generate a new random priority and then walk down the tree until we find a node with higher priority than the one we have generated. We then use the `split` method to cut a tree into two trees, the left one consisting of only nodes with value < `newValue` and the right one with the nodes of value > `newValue`. (Note: We discard nodes with value = `newValue`.) Finally, we create a new branch with the new value and priority, and with the new pair of split trees as our children.

The new `Branch` class shows how this is implemented, except the `split` method itself:

```
class Branch implements Node {
    final long value, priority;
    final Holder left, right;
    Branch(long value, Node left, Node right, long priority) {
        this.left = new Holder(left);
        this.right = new Holder(right);
        this.value = value;
        this.priority = priority;
    }
    public Node insert(long newValue, long newPriority) {
        if (newValue == value)
            return this;
        if (newPriority < priority) {
            Node[] pair = split(newValue);
            return new Branch(newValue, pair[0], pair[1], newPriority);
        } else if (newValue < value) {
            left.insert(newValue, newPriority);
        } else if (newValue > value) {
            right.insert(newValue, newPriority);
        }
        return this;
    }
    ...
}
```

Question 2.1 [15%]

In this question you will complete the code to insert values into thee tree. This will also give you the intuition you need to make the tree concurrent in the next question.

1. Implement `Node[] split(long value)` for `Branch`. The method should return an array with 2 trees: One in which all values are smaller than `value` and one in which they are bigger. The method should not change anything

in the original tree, and should take time proportional to the height of the tree.

2. Show that your code is correct (using sequential tests.) You may want to add a `boolean contains(long value)` method or a `reduce` method to be able to properly test it together with the `insert` method.

3. Show that inserting the values from 1 to 10^6 in order is much faster in the new tree than in the one from question 1.

4. Show that things don't work if multiple threads try to update the tree at the same time.

5. Show that adding **synchronized** to the methods in `Holder` suffices to remove the problems, though it may degrade performance.

Question 2.2 [10%]

1. Make a concurrent version of the treap using Software Transactional Memory, rather than using **synchronized** everywhere.

2. Show that this performs better than using **synchronized** everywhere, when most threads are reading (calling `contains`) and few are writing (calling `insert`.)

Question 2.3 [20%]

The problem with **synchronized** is that a single thread would take exclusive ownership of the entire treap, even though it's entirely possible that two threads would be working on completely separate parts of the tree. We can fix this by using a more fine-grained approach to locking.

There are two situations we need to handle:

- We are calling `split` on a tree that another thread is currently modifying.
- Another thread starts modifying a tree that we are currently calling `split` on.

We handle this by having threads "mark" the nodes as they are descending the tree, so that once we try to split a node, we know it is currently safe to do so. Once we try to split a node, we need to lock it, so no other thread can try to walk into it while we are modifying it.

1. It turns out, Java has a lock called `ReentrantReadWriteLock` which is a perfect fit for implementing the above locking scheme. Explain how the operations on this lock may be related to the marking and locking operations described above.

2. Now implement this approach to locking. You should be able to contain all code related to concurrency in the `Holder` class.

   Hint: One thing to be aware of when using ReadWriteLock is what happens when you are currently holding the `.readlock()` and want to upgrade to a `.writelock()`.

3. Show that your code passes your tests from question 2.1. Make sure you have enough tests, as bugs can be subtle.

4. Argue whether the code always completes, if it can ever deadlock, and where its linearisation points are.

Question 2.4 Benchmarking [10%]

Include a plot similar to that on page 11 of A Practical Concurrent Binary Search Tree, comparing the treaps from question 2.1, 2.2, question 2.3 and Java's `ConcurrentSkipListSet`.

Your benchmarks should include different read/write loads, different numbers of threads and different input sizes.