Parallel and Distributed Computing
DD2443 - Pardis24
Exercises for Lecture 9

Name: Casper Kristiansson

September 26, 2024

Exercise 1

Question

Write a class, `ArraySum`, that provides a method:

```
static public int sum(int[] a)
```

that uses divide-and-conquer to sum the elements of the array argument in parallel.

Answer

```
public class ArraySum {
    public static int sum(int[] a) {
        ForkJoinPool pool = new ForkJoinPool();
        return pool.invoke(new SumTask(a, 0, a.length));
    }

    private static class SumTask extends RecursiveTask<Integer> {
        private final int[] array;
        private final int start, end;

        public SumTask(int[] array, int start, int end) {
            this.array = array;
            this.start = start;
            this.end = end;
        }

```

```java
        @Override
        protected Integer compute() {
            if (end - start < 1000) {
                int sum = 0;
                for (int i = start; i < end; i++) sum += array[i];
                return sum;
            } else {
                int mid = (start + end) / 2;
                SumTask leftTask = new SumTask(array, start, mid);
                SumTask rightTask = new SumTask(array, mid, end);
                leftTask.fork();
                return rightTask.compute() + leftTask.join();
            }
        }
    }
}
```

Exercise 2

Question

```java
Queue qMin = (q0.size() < q1.size()) ? q0 : q1;
Queue qMax = (q0.size() < q1.size()) ? q1 : q0;

synchronized (qMin) {
    synchronized (qMax) {
        int diff = qMax.size() - qMin.size();
        if (diff > THRESHOLD) {
            while (qMax.size() > qMin.size()) {
                qMin.enq(qMax.deq());
            }
        }
    }
}
```

The code above shows an alternate way of rebalancing two work queues: first, lock the smaller queue, then lock the larger queue, and rebalance if their difference exceeds a threshold. What is wrong with this code?

Answer

The problem with this segment of code is that it could lead to a possible deadlock because of inconsistent locking order. This can happen because of qMin and qMax where they are acquired based on the sizes of the queue. For example, a situation where a deadlock could happen is where one thread tries to lock qMin and then tries to lock qMax. At the same time, another thread tries to lock qMax and then tries to lock qMin. In this situation, both threads could end up waiting for each other to release the locks and therefore a deadlock would happen.

A possible solution to this would be to improve a consistent locking order. This can be completed by comparing an object's physical address (reference) address. So based on the address the different threads can acquire based on that.

Exercise 3

Question

1. In the `popBottom()` method of HSLS Fig. 16.11, the `bottom` field is `volatile` to assure that in `popBottom()` the decrement at Line 15 is immediately visible. Describe a scenario that explains what could go wrong if `bottom` were not declared as `volatile`.

2. Why should we attempt to reset the `bottom` field to zero as early as possible in the `popBottom()` method? Which line is the earliest in which this reset can be done safely? Can our `BoundedDEQueue` overflow anyway? Describe how.

Answer

1. If the bottom was not declared as volatile the change happening on line 15 for the popBottom might not always be visible directly to the other threads. This means that it could lead to inconsistency where one thread might decrease the value while another sees another update. This means that we could end up in a situation where a queue is empty but one thread sees it as being nonempty and might perform a dequeue.

2. When resetting the bottom to zero at line 27 it is important to notify that the dequeue is empty. This can be done by adding a simple check with newBottom is equal to oldTop to ensure that there are no more tasks that remain.