Assignments for week 5

Anders Degn Lapiki, Jacob Kjærulff Furberg, Jonas Ishøj Nielsen

01/10/2020

Contents

1 5.1 ExecutorAccountExperiments

1.1 5.1.1 - green

Called Mark 7 with transfer and got the following results, which after being converted to milliseconds shows it is proportional to 50 milliseconds, which is the simulated transaction time.
Mark7("transfer", i -> t.transfer2());

```
Transfer 3962 from 9 to 6
transfer                      50380143.8 ns  111394.14          8
Transfer 5086 from 9 to 0
transfer                      50402605.0 ns   98626.40          8
Transfer 1130 from 6 to 0
transfer                      50411271.3 ns  342727.75          8
Transfer 190 from 0 to 6
transfer                      50428788.8 ns  311347.21          8
Transfer 516 from 4 to 2
transfer                      50362100.0 ns   90714.71          8
```

1.2 5.1.2 - green

Our solution sorts which lock to use first depending on the account's Id. We are uncertain if the Account id is unique, and therefore chose to keep the tie lock.

1.3 5.1.3 - green

Running it on different computers gave different results. On one it seems like there is not thread limit, almost like JIT removes threads that don't further affect any threads' state.

On the second computer it threw an OutOfMemory exception when creating thread #4170.

We used some code we found online for the purpose[1]. The code was:

```
public class DieLikeADog {
    private static Object s = new Object();
    private static int count = 0;
    public static void main(String[] argv){
        for(;;){
            new Thread(new Runnable(){
                    public void run(){
                        synchronized(s){
                            count += 1;
                            System.err.println("New
                                thread #"+count);
```

[1] https://stackoverflow.com/questions/763579/how-many-threads-can-a-java-vm-support

```
                        }
                        for(;;){
                            try {
                                Thread.sleep(1000);
                            } catch (Exception e){
                                System.err.println(e);
                            }
                        }
                    }
                }).start();
        }
    }
}
```

1.4 5.1.4 - green

Seen in `ExecutorAccountExperiments.java`.

1.5 5.1.5 - green

Yes, after adding the ExecutorService.shutdown() to gracefully wait for all threads to finish and a loop waiting for the service to be terminated before continuing the code.

```
exec.shutdown();
while (!exec.isTerminated()) {}
System.out.println("done");
```

1.6 5.1.6 - yellow

The code from 5.5 does most of the work. We just add a timer to start at the creation of the executo, and stop the timer when the executor is terminated. Running with 10 accounts and 5 transactions took: 0.1302652 ns.

1.7 5.1.7 - yellow

The optimal thread count for 50,40,30,20,10,1 ms sleep can be calculated with the the formular:

$$N_{threads} = N_{cpu} * U_{cpu} * (1 + W/C)$$

Where:
N_{cpu} is: 8.
U_{cpu} is: 1, since we decided to have full utilization.
C is: 0.4.

W is: the wait time we are testing.

The results are seen below.

```
NTHREADS 50 = 1008
NTHREADS 40 = 808
NTHREADS 30 = 608
NTHREADS 20 = 408
NTHREADS 10 = 208
NTHREADS 1 = 24
```

1.8 5.1.8 - red

Yes it makes sense with `Thread.Sleep()`, it is the value of W in the formula.

Since we used the formula for the previous item, at the end when the sleep time is only 1ms, the amount of threads is 24. This makes sense since the execution of the code takes 0.4ms. So a thread will on average spend 2/3 of its time asleep. Thus the amount of threads should be 3 times larger than the amount of CPU cores.
To optimize the number of threads used it should be min of the calculated result and number of transactions, as it doesn't make sense creating a pool size with a large part that will never be used.

2 5.2

2.1 5.2.1 - Yellow

Result can be seen in `out5_2_1.txt`.

2.2 5.2.2 - Yellow

Result can be seen in `out5_2_2.txt`.

2.3 5.2.3 - Yellow

The resulting graphs can be seen in figure1 and figure2.

2.4 5.2.4 - Yellow

As neither makes unneeded threads, it makes sense that the curve bottoms out. The number of cores would also allow 8 times w/c, which should be above the 32 taskcount used.

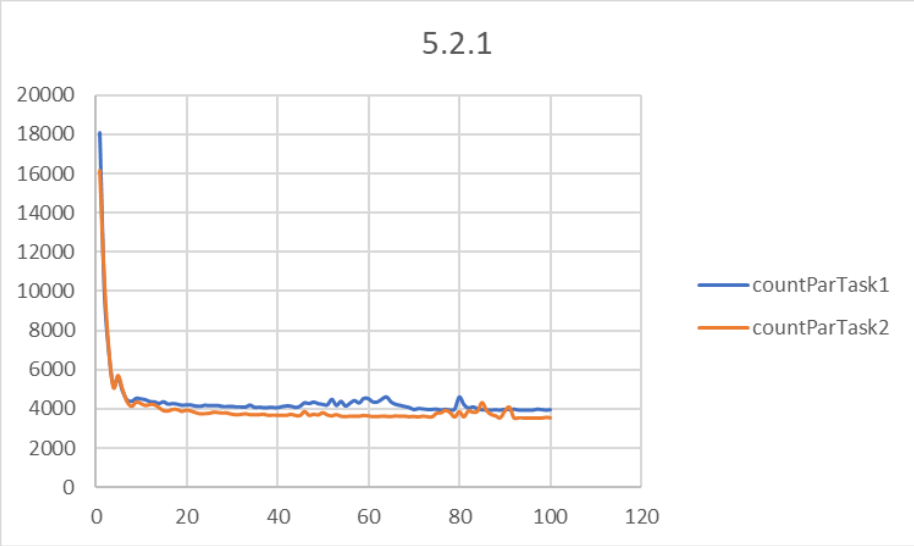

Figure 1: countParTask* for 5.2.1

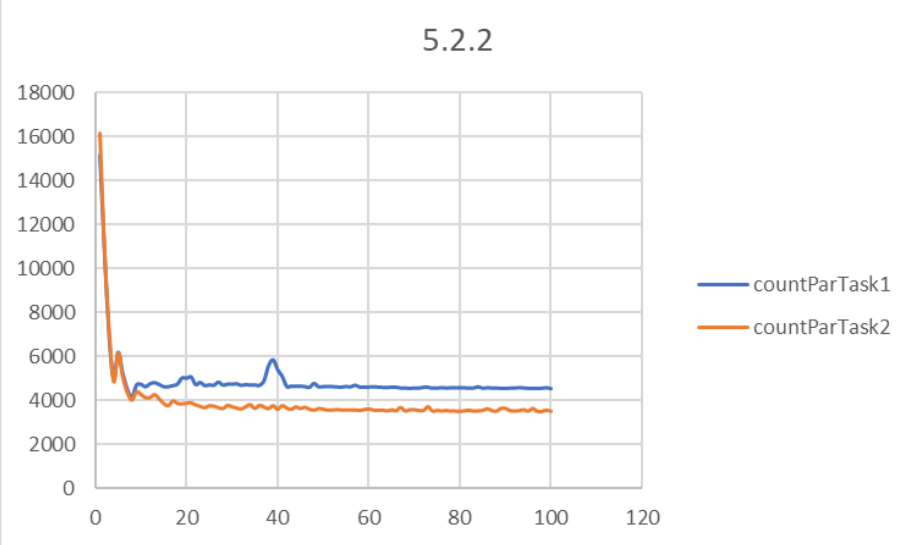

Figure 2: countParTask* for 5.2.2

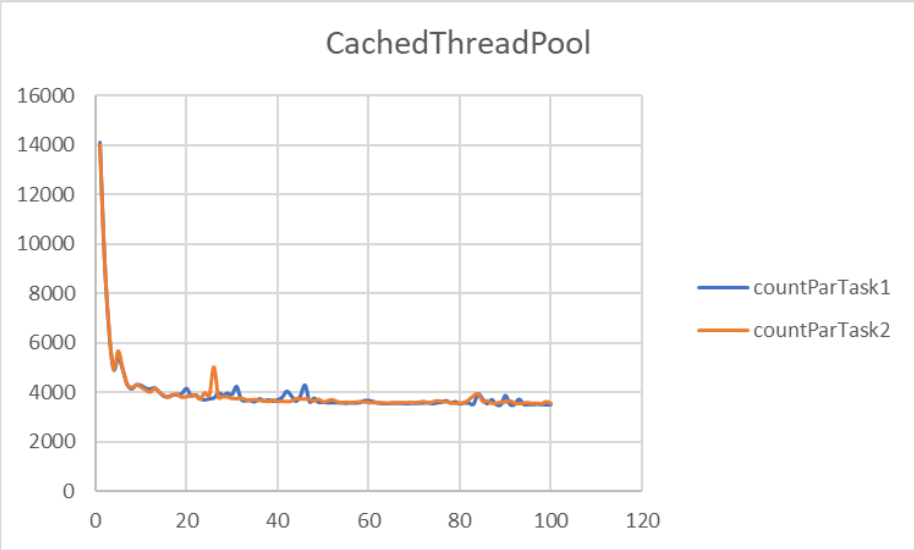

Figure 3: countParTask* for 5.2.5 cache

2.5 5.2.5 - Red

The resulting graphs can be seen in figure3 and figure4. The graph figure4 has a large lag spike since at that time avast decided to run a virus check. It can be assumed that it would have flattened out so we decided not to run again.

3 5.3

3.1 5.3.1 - yellow

The connection and the code works.

3.2 5.3.2 - yellow

Result can be seen in `5.3.2.txt`.

3.3 5.3.3 - yellow

Running the code five times gave the following results in seconds:

```
5.734445464
3.727174802
3.88584372
3.386530756
3.635910129
```

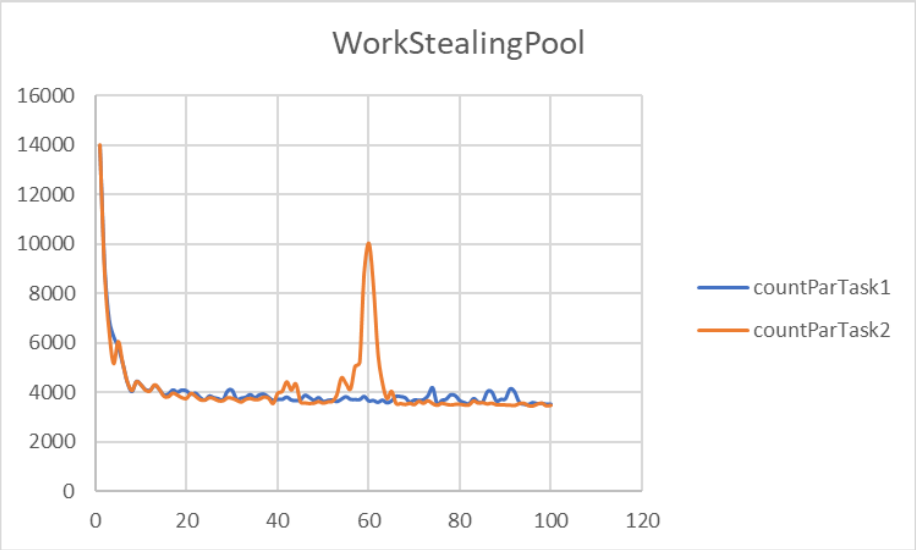

Figure 4: countParTask* for 5.2.5 work stealing

3.4 5.3.4 - yellow

The code can be found in: `TestDownload.java`.

3.5 5.3.5 - red

The time is bounded by the slowest task. The following results are from an WorkStealingPool.

```
2.210262394
2.11396841
2.080261369
2.142347714
1.990759771
```

The following results are from an CachedThreadPool.

```
2.056195673
2.076905397
2.099084433
3.067924193
2.069620845
```

The following results are from an FixedSizePool with optimal thread pool size, assuming w/c to be 50, which is probably too low.

```
2.149035367
2.25106756
2.092000593
2.064738294
2.122252411
```