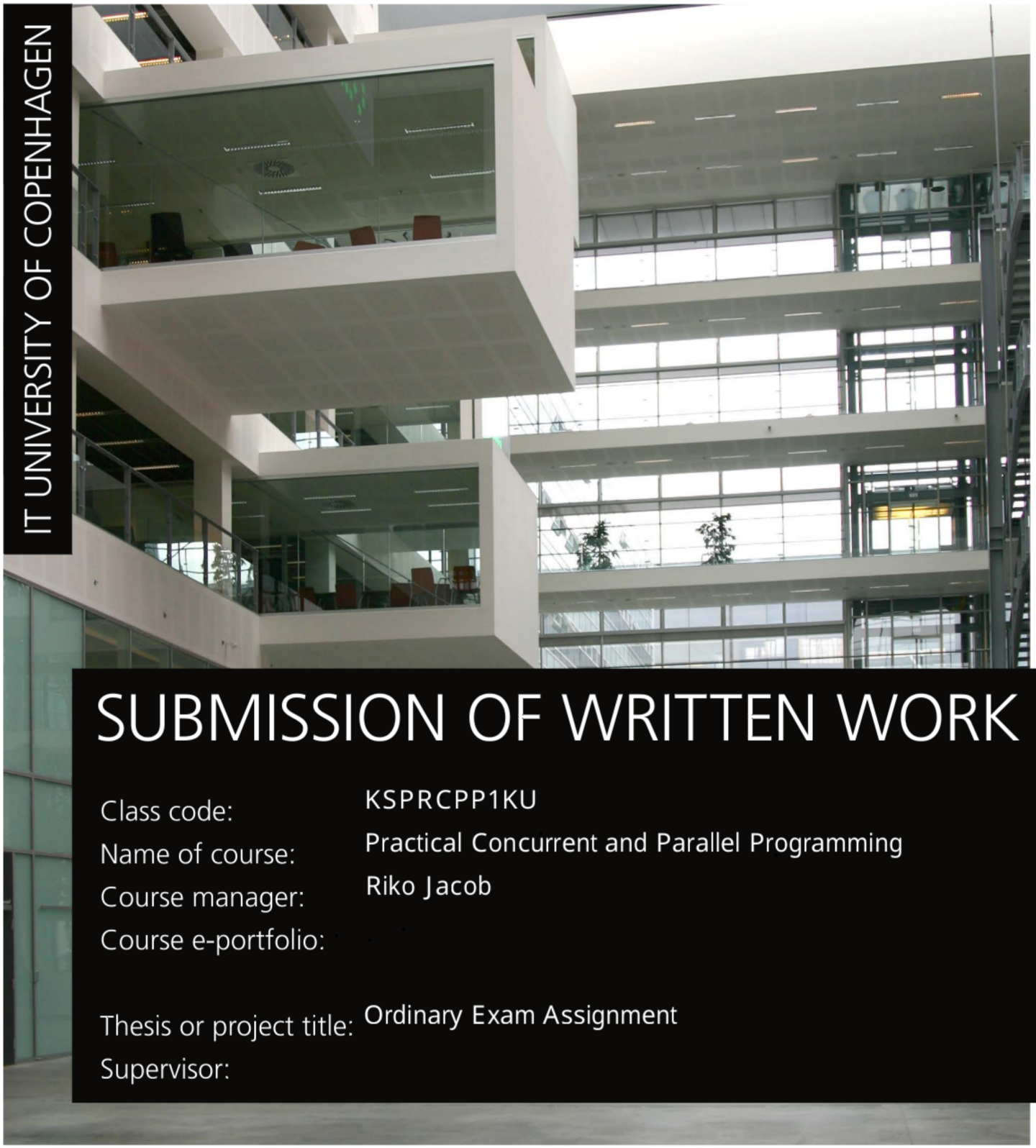

IT UNIVERSITY OF COPENHAGEN

SUBMISSION OF WRITTEN WORK

Class code:	KSPRCPP1KU
Name of course:	Practical Concurrent and Parallel Programming
Course manager:	Riko Jacob
Course e-portfolio:	
Thesis or project title:	Ordinary Exam Assignment
Supervisor:	

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Emil Christian Lynegaard	17/07-1994	ecly@itu.dk
2.			@itu.dk
3.			@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk

Contents

Practical Concurrent and Parallel Programming

Emil Lynegaard

December 12, 2017

I hereby declare that I have answered the exam questions myself without any outside help.

Throughout the report, there will be inline code snippets. Full versions of modified files can be found in the appendix. For all testing, the same machine will be used. Table 1 contains the output of running the method `SystemInfo` from `MyUnionFind.java`, found in section 8.3.

OS	Linux; 4.13.12-1-ARCH; amd64
JVM	Oracle Corporation; 1.8.0_144
CPU	null; 8 "cores"
Date	2017-12-11T09:20:13+0100

Table 1: System Info

---

1 Question 1

1.1

Seeing as we are interested in seeing how well each implementation performs on random input of different sizes, we use `Mark9` for the benchmarking, seeing as it calculates the per element mean time for us.

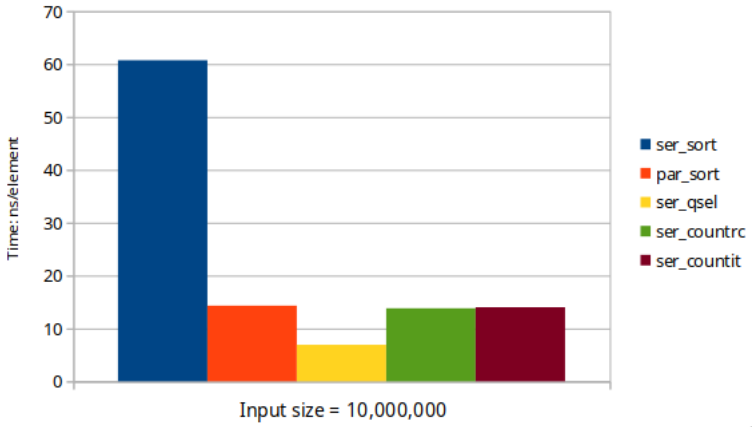

Figure 1: Plot of running time per element for given implementations. Tested with input size 10^7

From Graph 1 we see that, perhaps unsurprisingly, the serial `quickSelect` and `quickCount` implementations beat out serial sort entirely. We do however see parallel sort get rather close to the expected linear running time implementations, due to the test machine having a quad-core CPU, allowing an approximate 4 time speedup from its serial implementation. Between the expected linear running time algorithms, serial `quickSelect` beats out `quickCount` confidently. This may be due to `quickSelect` avoiding spending time allocating new memory, other than its initial copy.

1.2

Below is shown a parallel `quickCount` implementation using tasks, where `quickCountItTask` is the main function, and `filter` is the auxiliary method in which the parallel filtering is done. The filter method mainly serves to reduce code duplication as the original implementation is near identical for the filtering cases where we either have a too larger or a too small selected partition number.

To allow multiple tasks to safely increment `count`, we use a `final AtomicInteger` on line 27. In general for each iteration, all variables that we need to access from within the threads are made final, as we get compile errors otherwise. For task partitioning, we try to split up the input array in even intervals skipping the partition element. For indivisible numbers, we give the leftovers to the last created task. This can slow us down a little bit, since one task may end up doing more work than the others. In the `filter` method we take the same approach to partitioning, and once more use an `AtomicInteger` on line 7, to safely keep track of what index in the new

array `m` we want to insert the next value in. This ensures that we never try to write to the same index of `m` twice.

Other than these parallelization measures, the overall structure of `quickCountItTask` is similar to that of the given `quickCountTask`. For correctness we compare `quickCountItTask`'s output to that of our given reference implementations and observe that it yields expected results.

```
// Takes an arr, a partition, the size of the output array and a BiFunction,
// returning an array of the given size containing elements from arr for which
// f.apply(arr[i], partition) returns true.
public static int[] filter(int[] arr, int partition, int size,
     BiFunction<Integer,Integer,Boolean> f){
    int[] m = new int[size];
    ArrayList<Callable<Void>> filterers = new ArrayList<>();
    final AtomicInteger j = new AtomicInteger(0);
    final int step = arr.length/threadCount;
    for(int i=0;i<threadCount;i++) {
        final int from = i==0 ? 1 : i*step;
        final int to = i==threadCount-1 ? arr.length : i*step+step;
        filterers.add(() -> {
            for(int h= from; h<to; h++)
                if(f.apply(arr[h],partition)) m[j.getAndIncrement()]=arr[h];
            return null;
        });
    }
    try{ executor.invokeAll(filterers);
    } catch (InterruptedException e) { System.err.println("Threads interrupted");}
    return m;
}

static ExecutorService executor = Executors.newWorkStealingPool();
public static int quickCountItTask(int[] in) {
    int target = in.length/2;
    do {
        final AtomicInteger count = new AtomicInteger(0);
        final int[] inp = in;
        final int n = inp.length, p = inp[0];
        final int step = n/threadCount;

        //Counting
        ArrayList<Callable<Void>> counters = new ArrayList<>();
        for(int i=0;i<threadCount;i++) {
            final int from = i==0 ? 1 : i*step; //skip pivot
            //for indivisible numbers, just let the last thread take a larger chunk
            final int to = i==threadCount-1 ? inp.length : i*step+step;
            counters.add(() -> {
                for(int j = from; j<to; j++)
                    if(inp[j]<p) count.getAndIncrement();
                return null;
            });
        }
        try{ executor.invokeAll(counters);
        } catch (InterruptedException e) { System.err.println("Threads interrupted");}

```

```
47          if (count.get() == target) return p; //Terminated
48
49          //Filtering
50          boolean tooLargeP = count.get() > target;
51          int size = tooLargeP ? count.get() : n-count.get()-1;
52          if(tooLargeP) {
53              in = filter(inp, p, size, (x,y) -> x < y);
54          } else {
55              in = filter(inp, p, size, (x,y) -> x >= y);
56              target=target-count.get()-1;
57          }
58      } while( true );
59  }
```

1.3

Since the machine used for testing has 4 physical cores, this is the ideal amount of threads for the quickSelectIt. There is however only little difference between 4 and 8 since it supports hyperthreading[1]. Because of this, we set the `threadCount` to 4, and test with various input sizes. Figure 2 depicts the results.

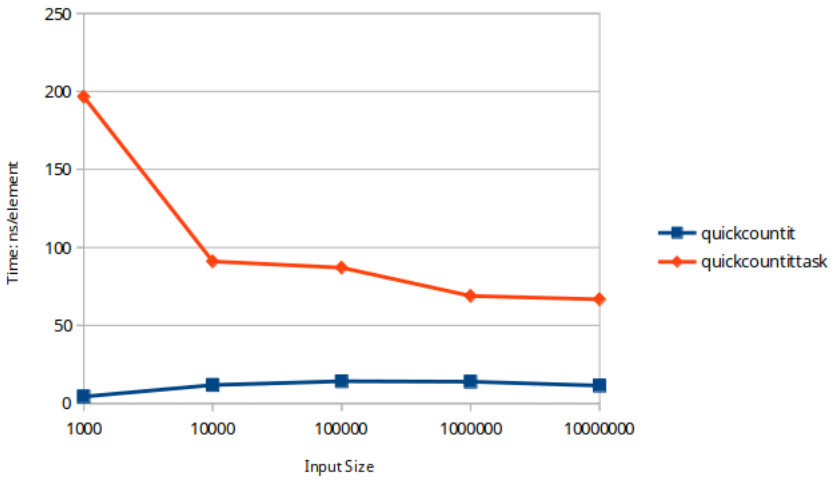

Figure 2: Plot of running times per element between `quickCountIt` and `quickCountItTask`

Perhaps in contrast to the expected outcome, `quickCountItTask` remains slower than `quickCountIt` even for large inputs. This can be due to how much it slows down on small inputs, with near $200ns/element$ for inputs of size 10^3. Based on the Figure 2 it seems that it will not be able to catch up to the serial implementation no matter the input size.

[1] https://ark.intel.com/products/80806/Intel-Core-i7-4790-Processor-8M-Cache-up-to-4_00-GHz

1.4

We assume that a single partitioning step indicates means an entire iteration of the `do-block` including counting and filtering. To test a single partitioning step we temporarily modify `quickCountItTask` to return after a single iteration. Table 2 depicts the running time per element for inputs of size 10^4 with a varying number of threads for a single step. Here we see that on smaller input sizes, the overhead of using multiple threads becomes increasingly noticeable.

Threads	1	2	4	8	16	32
Time: ns/element	15.0	28.4	32.7	35.7	35.9	36.4

Table 2: Running times per element for a single partitioning step in `quickCointItTask`, with input size 10^4, and a varying thread count

We now try to pin the thread count to 4 and vary the input size. This is depicted in Table 3. Here we see that with the pinned thread count, we seem to scale positively in the size of the input up until around 10^6. This makes sense in terms of the benefits of multithreading outweighing the overhead, when each thread gets a sufficiently large part of the input to work. Either way, our running time per element remains higher than we expect, so it may be that tweaks some tuning is needed for the implementation. This could for example be to try with thread local variables for the count step, to avoid superfluous synchronization among threads.

Input size	10^3	10^4	10^5	10^6	10^7	10^8
Time: ns/element	26.7	32.5	32.4	18.1	19.7	22.4

Table 3: Running times per element for a single partitioning step in `quickCointItTask`, with input size of thread count of 4 and varying input size

1.5

To make our implementation a hybrid, we declare a constant `CUTOFF` and add the following line to the top of the `do-block` in `quickCountItTask`.

```
1 if (in.length <= CUTOFF) return quickCountIt(in);
```

This change does make `quickCountItTask` faster, but even as a hybrid, it is unable to beat the sequential version, meaning that it still remains best to use the sequential version on its own. This finding may either be due to the overhead of thread synchronization, or other implementation specific quirks.

2 Question 2

2.1

```
1     (int) Arrays.stream(inp).skip(1).filter(i -> i < p).count();
```

Where `inp` is our input array, and `p` is our current partition candidate. Skip the first element as this is the index of the partition element with which we do not wish to compare.

2.2

```
        Arrays.stream(inp).skip(1).filter(i -> i < p).toArray();
        Arrays.stream(inp).skip(1).filter(i -> i >= p).toArray();
```

Similar to section 2.1 except now we get the output as an array of the filtered elements instead of the count.

2.3

```
public static int quickCountStream(int[] inp) {
    int partition=-1, count=0, n=inp.length;
    int target = n/2;
    do {
        partition=inp[0];
        final int p = partition;
        n=inp.length;
        count = (int) Arrays.stream(inp).skip(1).filter(i -> i < p).count();
        if (count == target) break;
        if (count > target)
            inp = Arrays.stream(inp).skip(1).filter(i -> i < p).toArray();
        else {
            inp = Arrays.stream(inp).skip(1).filter(i -> i >= p).toArray();
            target=target-count-1;
        }
    } while( true );
    return partition; // we are on target
}
```

Combining the two we get above implementation, which we see yields correct results by comparing it to the reference implementations for several instances.

2.4

```
    Arrays.stream(inp).parallel().skip(1).filter(i -> i < p).count();
    Arrays.stream(inp).parallel().skip(1).filter(i -> i < p).toArray();
    Arrays.stream(inp).parallel().skip(1).filter(i -> i >= p).toArray();
```

Here we simply throw `.parallel()` onto the start of our pipelines from before.

2.5

```
1  public static int quickCountStream(int[] inp) {
2      int partition=-1;
3      int target = inp.length/2;
4      // Since we have to be working with boxed Integers. We start off by converting.
5      List<Integer> list = Arrays.stream(inp).boxed().collect(Collectors.toList());
6      do {
7          partition = list.get(0);
8          final Integer p = partition;
9          Map<Boolean, List<Integer>> res = list.stream().skip(1).parallel()
10             .collect(Collectors.partitioningBy(i -> i < p));
11
12         List<Integer> smaller = res.get(true);
13         List<Integer> bigger = res.get(false);
14
15         if (smaller.size() == target) break;
16         if (smaller.size() > target) list = smaller;
17         else {
18             target=target-smaller.size()-1;
19             list = bigger;
20         }
21     } while( true );
22     return partition; // we are on target
23 }
```

To avoid having to constantly do boxing, since Collectors do not work with primitives, we start off by converting our `int[]` to `List<Integer>`.

The `partitioningBy` collector gives us a map of with two `Boolean` keys. The `true` entry on line 12, holding all elements larger than our partition element, and false entry on line 13 holding all the elements larger than or equal to our partition element. The size of `smaller` now represents our `count` from before, and the remainder of the code is similar to the given code.

2.6

In Figure 3 we have tested all implementations from Figure 1 again, and added the hybrid implementation as well as the stream based implementations. Here the `CUTOFF` for the parallel iterative `quickCount` is set to 10^4. Based on these tests, it appears that every parallel implementation is slower than the serial `quickSelect` and the serial recursive and iterative `quickCount`. While this is perhaps somewhat surprising, we can deduce that we either need smarter ways of parallelizing the implementations, or that `quickCount` does not lend itself positively to parallelization on my machine, despite it being very reasonable to implement.

3 Question 3

3.1 CPU pinning:

If we assume that the total number of threads is equal to the number of cores on the machine of execution, CPU pinning can be advantageous if the program gets all of the machine's CPU

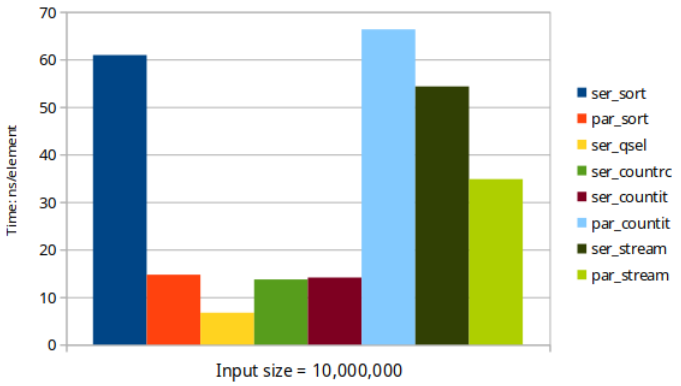

Figure 3: Plot of running times including parallel iterative and stream based. Tested with input size 10^7

time, and the list is evenly partitioned between threads. In these specific conditions, not having to schedule threads may be beneficial as these conditions should cause various threads to finish their iteration's work simultaneously. On the other hand, if the input is not evenly distributed and the system is using resources on other tasks as well, one partition may end up being poorly scheduled on a core with less free CPU time than one of the other available cores. Another benefit of thread pinning, which may be more significant, is that this will allow the threads to keep most of the relevant values in their designated cores' L1 cache, leading to less cache misses[2]. This benefit may even extend to L2 cache for some CPUs[3].

3.2 Load balancing:

By having each thread own certain data ranges, we may end up with several threads having nothing or very little to do after few iterations. For example, if the input is near sorted and we assign threads consecutive ranges, some threads may up filtering out all their elements in the first iteration depending on the selected pivot.

4 Question 4

4.1

Since `x` and `y` are parameters, we are not guaranteed to always grab locks in the same order, hence we are prone to dead-locks.

4.2

[2]https://en.wikipedia.org/wiki/CPU_cache#Cache_miss
[3]https://en.wikipedia.org/wiki/CPU_cache#Cache_hierarchy_in_a_modern_processor

```
union(0,1)
union(1,0)
```

Above example when executed in parallel may lead to the first call grabbing the lock on `nodes[0]`, the second call grabbing the lock on `nodes[1]` and both calls thereafter waiting to get the lock on the node that their counterpart already grabbed.

4.3

We modify the given `concurrent()` method in class `UnionFindTest` in file `MyUnionFind.java` to target the issue identified in 4.1.

```
1  public void deadlock(final int size, final UnionFind uf) throws Exception {
2      final int[] numbers = new int[size];
3      for (int i = 0; i < numbers.length; ++i) numbers[i] = i;
4      // Populate threads
5      final int threadCount = 32;
6      final CyclicBarrier startBarrier = new CyclicBarrier(threadCount+1),
7            stopBarrier = startBarrier;
8      Collections.shuffle(Arrays.asList(numbers));
9      for (int i = 0; i < threadCount; ++i) {
10         final boolean reverse = i%2==0;
11         Thread ti = new Thread(new Runnable() { public void run() {
12             try { startBarrier.await(); } catch (Exception exn) { }
13                 for (int j=0; j<100; j++)
14                     for (int i = 0; i < numbers.length - 1; ++i)
15                         if (reverse) uf.union(numbers[i + 1], numbers[i]);
16                         else uf.union(numbers[i], numbers[i + 1]);
17             try { stopBarrier.await(); } catch (Exception exn) { }
18         }});
19         ti.start();
20     }
21     startBarrier.await();
22     stopBarrier.await();
23     final int root = uf.find(0);
24     for (int i : numbers)
25         assertEquals(uf.find(i), root);
26     System.out.println("No deadlocks");
27 }
```

As seen on line 15 and 16, half the threads will now be attempting to union nodes in the reverse order of the other half. From my tests, calling the above defined `deadlock` method with parameters shown below, deadlocked every time.

```
UnionFindTest test = new UnionFindTest();
test.deadlock(itemCount, new BogusFineUnionFind(10_000));
```

By merely adding a check to the given `union()` method in class `BogusFineUnionFind` to ensure we always lock the lowest entry of the `nodes` array first, the deadlock test executes without

deadlocking. This is exactly what is done in the class `FineUnionFind` from the same file, which passes all tests.

5 Question 5

Specifications:

1. Pop returns an inserted item or the value null. It might block until another concurrent operation completes, but it will return without delay if no other operation is happening simultaneously. In particular, it will not block until another thread inserts some element.
2. For each element that is pushed, there is at most one pop operation that returns that element.
3. If there are no further concurrent operations, pop will succeed (i.e. return a non-null value) if so far there have been more successful push than pop operations.
4. If processor A pushed two elements x and y in this order, and processor B pops both elements, then this happens in reverse order. (There is no further constraint on ordering).

5.1

```java
import java.util.LinkedList;
public class MyStack<T> {
    private final Object lock;
    private final LinkedList<T> stack;

    public MyStack(){
        lock = new Object();
        stack = new LinkedList<T>();
    }

    public void push(T obj) {
        synchronized(lock){
            stack.push(obj);
        }
    }

    public T pop() {
        synchronized(lock){
            return stack.peek() != null ? stack.pop() : null;
        }
    }
}
```

Above generic implementation utilizes that Java's `LinkedList` ships with `push` and `pop`. Alternatively we could use the combination `addLast` and `removeLast` or `addFirst` and `removeFirst` of which the second pair by Java's documentation is equivalent to `push` and `pop`[4]. In `pop`, given

[4] https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html

that the specification states that we should return null if the list is empty, we use `peek` to check if there is a first element, if there is not, return null, otherwise `pop`. The locking could also be done implicitly on `this` by marking the methods as synchronized, but here we use an explicit global lock `Object` as it seems more in line with specification number 1.

5.2

Given the initially stated specifications, below are added bullet points (letters) matching specifications, describing why the implementation from 5.1 is sufficient.

1. (a) We take care of this explicitly in the implementation for `MyStack` in section 5.1, where we on line 19, `peek` prior to popping. If we blindly popped on an empty list, we would get a `NoSuchElementException`[5].
2. (a) Given that `pop` removes the single first element, and `push` only adds the element once, `pop` an element pushed once cannot be popped twice.

   (b) To set previous point in stone, since we are using a global single lock, everything happens sequentially, removing any chance of reading off an element that was removed in a different thread.
3. (a) Implied by 2.a.
4. (a) Since everything is handled sequentially due to the global lock, we have the guarantee that if one thread pushes two items, these will be pushed in the order of which the thread called push.

   (b) Given 3.a. and Java's Documentation stating that `push` and `pop` adds/removes from the head of the list, elements are bound to be popped in reverse compared to the order of which they were pushed.

5.3

5.3.1 Concurrency

To test for everything except reverse ordering (specification 4), we define the below shown test `concurrentTest`.

```
1  public static void concurrentTest(final int size, final int threads,
2          MyStack<Integer> stack) throws Exception {
3      final CyclicBarrier startBarrier = new CyclicBarrier(threads+1),
4           stopBarrier = startBarrier;
5
6      final int range = size/threads;
7      for (int i = 0; i < threads; ++i) {
8          final int nr = i;
9          Thread ti = new Thread(new Runnable() { public void run() {
10             try { startBarrier.await(); } catch (Exception exn) { }
11                 for(int j = range*nr; j<range*nr+range; j++)
12                     stack.push(j);
```

[5] https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html#pop()

```
13              try { stopBarrier.await(); } catch (Exception exn) { }
14          }});
15          ti.start();
16      }
17      startBarrier.await();
18      stopBarrier.await();
19      startBarrier.reset();
20      stopBarrier.reset();
21
22      final Set<Integer> pops = ConcurrentHashMap.newKeySet();
23      for (int i = 0; i < threads; ++i) {
24          final int nr = i;
25          Thread ti = new Thread(new Runnable() { public void run() {
26              try { startBarrier.await(); } catch (Exception exn) { }
27                  for(int j = range*nr; j<range*nr+range; j++)
28                      pops.add(stack.pop());
29              try { stopBarrier.await(); } catch (Exception exn) { }
30          }});
31          ti.start();
32      }
33
34      startBarrier.await();
35      stopBarrier.await();
36
37      if (pops.size() == size) System.out.println("Concurrency works :)");
38      else System.out.println("Concurrency doesn't work :(");
39  }
```

We create `CyclicBarrier`s on line 3, which we use to ensure that our threads are working simultaneously. We then assign different threads ranges between 0 and `size` in which they will push all numbers to the stack.

When all the threads are ready, we call `startBarrier.await()` on line 17 to start the pushing and `stopBarrier.await()` on the next line to wait for all of them to finish pushing.

We now divide the work similarly for popping threads. For each number they pop, we add it to the `ConcurrentHashMap` backed `Set` defined on line 22. When all threads are done popping, we can check whether the size of our `Set` is equal to the number of items we initially added to the `Stack`. Since there are no duplicates in a `Set`, and we only pushed unique numbers, if these are equal we can with reasonably confidence say that our implementation is working.

Scaling: In terms of scaling, since MyStack functions entirely sequentially, it scales poorly with multiple threads. In fact, the more threads we use, the more time will be spent locking and waiting for locks, making it slower and slower. In table 4 this behavior is illustrated.

Threads	1	2	4	8	16	32
Time (sec)	7.363	8.159	8.203	8.413	10.071	10.832

Table 4: Test times for concurrentTest with `size` 10^7

5.3.2 Ordering

To test that order works for multiple threads, we push numbers $[0..n]$ onto the stack from thread A, and pop `n` items off the stack from thread B, checking that these are the numbers $[n..0]$. This is shown below in method `testOrder`.

```
1  public static void testOrder(int n, MyStack<Integer> stack){
2      final AtomicBoolean working = new AtomicBoolean(true);
3      Thread A = new Thread(new Runnable() { public void run() {
4          for(int i=0; i<n; i++) stack.push(i);
5      }});
6      Thread B = new Thread(new Runnable() { public void run() {
7          for(int i=0; i<n; i++)
8              working.compareAndSet(true, n-1-i == stack.pop());
9      }});
10     try {
11         A.start(); A.join();
12         B.join(); B.join();
13     } catch (Exception e) {
14         System.out.println("Order dies >:(");
15     }
16     if(working.get()) System.out.println("Order works :)");
17     else System.out.println("Order doesn't work :(");
18 }
```

This behaviour was untested in `concurrentTest`, so a separate straight forward test to clear this up was needed, since specification number 4 was clear about this behavior.

5.4

Below is shown an implementation using striping, with a total of 32 stripes. To determine the stripe to use for a method call, we use `Thread.currentThread()hashCode()%STRIPES)` as shown on line 16 and 24. We use an `ArrayList` to store our `LinkedLists` since we cannot create arrays of parameterized types[6].

In `pop`, we use `i%STRIPES` to iterate through the stacks, starting from the stack at the computed `stripe`, with wraparound.

```
1  import java.lang.*;
2  import java.util.*;
3  public class MyStack<T> {
4      private Object lock;
5      private final List<LinkedList<T>> stacks;
6      private static final int STRIPES = 32;
7
8      public MyStack(){
9          lock = new Object();
10         stacks = new ArrayList<LinkedList<T>>();
11         for(int i = 0; i < STRIPES; i++)
12             stacks.add(new LinkedList<T>());
```

[6] https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#createArrays

```
13      }
14
15      public void push(T obj) {
16          int stripe = Thread.currentThread().hashCode()%STRIPES;
17          LinkedList<T> stack = stacks.get(stripe);
18          synchronized(stack){
19              stack.push(obj);
20          }
21      }
22
23      public T pop() {
24          int stripe = Thread.currentThread().hashCode()%STRIPES;
25          for (int i = stripe; i < stripe+STRIPES; i++){
26              LinkedList<T> stack = stacks.get(i%STRIPES);
27              synchronized(stack){
28                  if(stack.size() == 0) continue;
29                  return stack.pop();
30              }
31          }
32          return null;
33      }
34  }
```

5.5

With the new striping allowing actually run in parallel, we see that performance improved in our tests. The fastest new execution, as seen in table 5, was with 16 threads where it ran in 5.866 seconds. This is in contrast to the single threaded execution from table 4 that ran in 7.363 seconds with a single thread. Considering that we have 4 physical cores available, this is not that big of a performance improvement. This could be due to many threads hashing to the same stripe or due to the added iteration through all the stacks when we run into an empty one.

Threads	1	2	4	8	16	32
Time (sec)	8.016	7.473	7.465	6.313	5.866	7.337

Table 5: Test times for MyStack with striping for conccurentTest with size 10^7

5.6

Given our implementation described in 5.4, we modify line 29 to say:

```
1  return i == stripe ? stack.pop() : stack.removeLast();
```

If we are on our own stripe's stack, pop from the front, otherwise pop from the back.

5.7

Below is a description in pseudocode of what will go wrong given our changes in 5.6.

```
//Thread A // Stripe 0
myStack.push(0);
myStack.push(1);

//Thread B // Stripe 1
myStack.pop(); // this will return 0 - should be 1
myStack.pop(); // this will return 1 - should be 0
```

5.8

The code described in section 5.3.2 detects this issue and outputs "Order doesn't work :(" as expected.

6 Question 6

6.1

The given code has the flaw that it allows multiple threads to get into the else block before anyone changes the `state`. This means that in an example with thread `A` calling `consensus(x)` and thread `B` calling `consensus(y)`, both may retrieve return values indicating that parameter value `x` or `y` is the consensus, whereas only one of them will be equal to `state`. Hence we have a race condition.

6.2

Using synchronization on some lock, in this case on `this`, has a problem with termination. If one process crashes while holding the lock, or falls asleep for an extended amount of time, the whole system will halt as they wait to retrieve the lock. As such, this implementation is not fault tolerant.

6.3

Firstly, assuming that we are talking Java, this would fail to type-check, as we are returning an `AtomicInteger` from a method that supposedly returns an `int` and there is no implicit conversion from `AtomicInteger` to `int`. Even with this fixed, it would still fail, since two threads may enter the while loop before state changes, causing one of them to be stuck in the loop forever, since `state.compareAndSet(-1,x)` will then be returning false over and over. This will work if only one thread ever gets to enter the while loop and successfully `compareAndSet`s the `state`, since all other subsequent threads then merely get the value of the state, leading to consensus.

7 Question 7

7.1

Implementation found in appendix section 8.1.

7.2

Below is shown the output of three distinct runs of the Secure Communication System with Java+Akka. This assumes that distinct means differing key pairs and thereby varying encrypted texts.

```
$ java -cp scala.jar:akka-actor.jar:akka-config.jar:. SecComSys
public key: 18
private key: 8
cleartext: 'SECRET'
encypted: 'KWUJWL'
decrypted: 'SECRET'

$ java -cp scala.jar:akka-actor.jar:akka-config.jar:. SecComSys
public key: 2
private key: 24
cleartext: 'SECRET'
encypted: 'UGETGV'
decrypted: 'SECRET'

$ java -cp scala.jar:akka-actor.jar:akka-config.jar:. SecComSys
public key: 5
private key: 21
cleartext: 'SECRET'
encypted: 'XJHWJY'
decrypted: 'SECRET'
```

8 Appendix

8.1 SecComSys.java

```
1  // COMPILE:
2  // javac -cp scala.jar:akka-actor.jar SecComSys.java
3  // RUN:
4  // java -cp scala.jar:akka-actor.jar:akka-config.jar:. SecComSys
5
6  import java.util.*;
7  import java.io.*;
8  import akka.actor.*;
9
10 // -- HANDOUT ----------------------------------------------
```

```
11 class KeyPair implements Serializable {
12     public final int public_key, private_key;
13     public KeyPair(int public_key, int private_key) {
14         this.public_key = public_key;
15         this.private_key = private_key;
16     }
17 }
18
19 class Crypto {
20     static KeyPair keygen() {
21         int public_key = (new Random()).nextInt(25)+1;
22         int private_key = 26 - public_key;
23         System.out.println("public key: " + public_key);
24         System.out.println("private key: " + private_key);
25         return new KeyPair(public_key, private_key);
26     }
27
28     static String encrypt(String cleartext, int key) {
29         StringBuffer encrypted = new StringBuffer();
30         for (int i=0; i<cleartext.length(); i++) {
31             encrypted.append((char) ('A' + ((((int)
32                                 cleartext.charAt(i)) - 'A' + key) % 26)));
33         }
34         return "" + encrypted;
35     }
36 }
37
38 // -- MESSAGES ------------------------------------------------
39
40 class InitMessage implements Serializable {
41     public final ActorRef R;
42     public InitMessage(ActorRef R) {
43         this.R = R;
44     }
45 }
46
47 class RegisterMessage implements Serializable {
48     public final ActorRef pid;
49     public RegisterMessage(ActorRef pid) {
50         this.pid = pid;
51     }
52 }
53
54 class LookupMessage implements Serializable {
55     public final ActorRef pid;
56     public final ActorRef returnTo;
57     public LookupMessage(ActorRef pid, ActorRef returnTo) {
58         this.pid = pid;
59         this.returnTo = returnTo;
60     }
61 }
62
63 class KeyPairMessage implements Serializable {
```

```
64      public final KeyPair keyPair;
65      public KeyPairMessage(KeyPair keyPair) {
66          this.keyPair = keyPair;
67      }
68  }
69
70  class Message implements Serializable {
71      public final String Y;
72      public Message(String Y) {
73          this.Y = Y;
74      }
75  }
76
77  class CommMessage implements Serializable {
78      public final ActorRef pid;
79      public CommMessage(ActorRef pid) {
80          this.pid = pid;
81      }
82  }
83
84  class PubKeyMessage implements Serializable {
85      public final ActorRef recipient;
86      public final Integer publicKey;
87      public PubKeyMessage(ActorRef recipient, int publicKey) {
88          this.recipient = recipient;
89          this.publicKey = publicKey;
90      }
91  }
92
93  // -- ACTORS ------------------------------------------------
94
95  class RegistryActor extends UntypedActor {
96      public final Map<ActorRef, Integer> registry = new HashMap<>();
97
98      public void onReceive(Object o) throws Exception {
99          if (o instanceof RegisterMessage) {
100             RegisterMessage rm = (RegisterMessage) o;
101             KeyPair keyPair = Crypto.keygen();
102             registry.put(rm.pid, keyPair.public_key);
103             rm.pid.tell(new KeyPairMessage(keyPair), getSelf());
104         } else if (o instanceof LookupMessage) {
105             LookupMessage lm = (LookupMessage) o;
106             lm.returnTo.tell(new PubKeyMessage(lm.pid, registry.get(lm.pid)), getSelf());
107         }
108     }
109 }
110
111 class ReceiverActor extends UntypedActor {
112     public ActorRef registry;
113     public int publicKey;
114     public int privateKey;
115
116     public void onReceive(Object o) throws Exception {
```

```
117         if (o instanceof InitMessage) {
118             InitMessage im = (InitMessage) o;
119             im.R.tell(new RegisterMessage(getSelf()), getSelf());
120         } else if (o instanceof KeyPairMessage) {
121             KeyPairMessage kpm = (KeyPairMessage) o;
122             publicKey = kpm.keyPair.public_key;
123             privateKey = kpm.keyPair.private_key;
124         } else if (o instanceof Message) {
125             Message m = (Message) o;
126             String X = Crypto.encrypt(m.Y, privateKey);
127             System.out.print("decrypted: '" + X + "'\n");
128         }
129     }
130 }
131
132 class SenderActor extends UntypedActor {
133     public ActorRef registry;
134
135     public void onReceive(Object o) throws Exception {
136         if (o instanceof InitMessage) {
137             InitMessage im = (InitMessage) o;
138             registry = im.R;
139         } else if (o instanceof CommMessage) {
140             CommMessage cm = (CommMessage) o;
141             registry.tell(new LookupMessage(cm.pid, getSelf()), getSelf());
142         } else if (o instanceof PubKeyMessage) {
143             PubKeyMessage pkm = (PubKeyMessage) o;
144             String X = "SECRET";
145             System.out.print("cleartext: '" + X + "'\n");
146             String Y = Crypto.encrypt(X, pkm.publicKey);
147             System.out.print("encypted: '" + Y + "'\n");
148             pkm.recipient.tell(new Message(Y), getSelf());
149         }
150     }
151 }
152
153 // -- MAIN ------------------------------------------------
154
155 public class SecComSys {
156     public static void main(String[] args) {
157         final ActorSystem system = ActorSystem.create("SecComSys");
158         final ActorRef registry = system.actorOf(Props.create(RegistryActor.class),
                "reigstry");
159         final ActorRef receiver = system.actorOf(Props.create(ReceiverActor.class),
                "receiver");
160         receiver.tell(new InitMessage(registry), ActorRef.noSender());
161         final ActorRef sender = system.actorOf(Props.create(SenderActor.class),
                "sender");
162         sender.tell(new InitMessage(registry), ActorRef.noSender());
163         sender.tell(new CommMessage(receiver), ActorRef.noSender());
164         system.shutdown();
165     }
166 }
```

8.2 TestQuickSelect.java

```java
import java.util.*;
import java.util.stream.*;
import java.util.function.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;

class TestQuickSelect {
    public static int medianSort(int[] inp) {
        int w[] = Arrays.copyOf(inp, inp.length);
        Arrays.sort(w);
        return w[w.length/2];
    }
    public static int medianPSort(int[] inp) {
        int w[] = Arrays.copyOf(inp, inp.length);
        Arrays.parallelSort(w);
        return w[w.length/2];
    }

    public static int partition(int[] w, int min, int max) {
        int p = min; // use w[p] as pivot
        int left=min+1, right = max-1;
        while(left <= right) {
            while( w[left] <= w[p] && left < right ) left++;
            while( w[right] > w[p] && left <= right ) right--;
            if(left >= right) break;
            int t=w[left]; w[left]=w[right]; w[right]=t;
        }
        int t=w[p]; w[p]=w[right]; w[right]=t;
        return right;
    }

    public static int quickSelect(int[] inp) {
        int w[] = Arrays.copyOf(inp, inp.length);
        return quickSelect(w,0,w.length,w.length/2);
    }
    public static int quickSelect(int[] w, int min, int max, int target) {
        int p = partition(w,min,max);
        if( p < target ) return quickSelect(w,p+1,max,target);
        if( p > target ) return quickSelect(w,min,p,target);
        return w[target]; // p==target
    }

    public static int quickSelectIt(int[] inp) {
        int w[] = Arrays.copyOf(inp, inp.length);
        int target = w.length/2;
        int p = -1, min=0, max=w.length;
        do{
            p = partition(w,min,max);
            if( p < target ) min=p+1;
            if( p > target ) max=p;
        } while(p!=target);
```

```
52          return w[p];
53      }
54
55      public static int quickCountRec(int[] inp, int target) {
56          final int p=inp[0], n=inp.length;
57          int  count=0;
58          for(int i=1;i<n;i++) if(inp[i]<p) count++;
59          if(count > target) {
60              int m[] = new int[count];
61              int j=0;
62              for(int i=1;i<n;i++) if(inp[i]<p) m[j++]=inp[i];
63              return quickCountRec(m,target);
64          }
65          if(count < target) {
66              int m[] = new int[n-count-1];
67              int j=0;
68              for(int i=1;i<n;i++) if(inp[i]>=p) m[j++]=inp[i];
69              return quickCountRec(m,target-count-1);
70          }
71          return p; // we are on target
72      }
73
74      public static int quickCountIt(int[] inp) {
75          int p=-1, count=0, n=inp.length;
76          int target = n/2;
77          do {
78              p=inp[0];
79              count=0;
80              n=inp.length;
81              for(int i=1;i<n;i++) if(inp[i]<p) count++;
82              if(count > target) {
83                  int m[] = new int[count];
84                  int j=0;
85                  for(int i=1;i<n;i++) if(inp[i]<p) m[j++]=inp[i];
86                  inp = m;
87                  continue;
88              }
89              if(count < target) {
90                  int m[] = new int[n-count-1];
91                  int j=0;
92                  for(int i=1;i<n;i++) if(inp[i]>=p) m[j++]=inp[i];
93                  inp =m;
94                  target=target-count-1;
95                  continue;
96              }
97              break;
98          } while( true );
99          return p; // we are on target
100     }
101
102     // Takes an arr, a partition, the size of the outpuit array and a BiFunction,
103     // returning an array of the given size containing elements of arr for which
104     // f.apply(arr[i], partition) returns true.
```

```
105     public static int[] filter(int[] arr, int partition, int size,
            BiFunction<Integer,Integer,Boolean> f){
106         int[] m = new int[size];
107         ArrayList<Callable<Void>> filterers = new ArrayList<>();
108         final AtomicInteger j = new AtomicInteger(0);
109         final int step = arr.length/threadCount;
110         for(int i=0;i<threadCount;i++) {
111             final int from = i==0 ? 1 : i*step;
112             final int to = i==threadCount-1 ? arr.length : i*step+step;
113             filterers.add(() -> {
114                 for(int h= from; h<to; h++)
115                     if(f.apply(arr[h],partition)) m[j.getAndIncrement()]=arr[h];
116                 return null;
117             });
118         }
119         try{ executor.invokeAll(filterers);
120         } catch (InterruptedException e) { System.err.println("Threads interrupted");}
121         return m;
122     }
123
124     final static ExecutorService executor = Executors.newWorkStealingPool();
125     final static int CUTOFF = 10_000;
126     public static int quickCountItTask(int[] in) {
127         int target = in.length/2;
128         do {
129             if (in.length <= CUTOFF) return quickCountIt(in);
130             final AtomicInteger count = new AtomicInteger(0);
131             final int[] inp = in;
132             final int n = inp.length, p = inp[0];
133             final int step = n/threadCount;
134
135             //Counting
136             ArrayList<Callable<Void>> counters = new ArrayList<>();
137             for(int i=0;i<threadCount;i++) {
138                 final int from = i==0 ? 1 : i*step; //skip pivot
139                 // for indivisible numbers, just let the last thread take a larger chunk
140                 final int to = i==threadCount-1 ? inp.length : i*step+step;
141                 counters.add(() -> {
142                     for(int j= from; j<to; j++)
143                         if(inp[j]<p) count.getAndIncrement();
144                     return null;
145                 });
146             }
147             try{ executor.invokeAll(counters);
148             } catch (InterruptedException e) { System.err.println("Threads
                    interrupted");}
149
150             if (count.get() == target) return p; //Terminated
151
152             //Filtering
153             boolean tooLargeP = count.get() > target;
154             int size = tooLargeP ? count.get() : n-count.get()-1;
155             if(tooLargeP) {
```

```
156                 in = filter(inp, p, size, (x,y) -> x < y);
157             } else {
158                 in = filter(inp, p, size, (x,y) -> x >= y);
159                 target=target-count.get()-1;
160             }
161         } while( true );
162     }
163 
164     public static int quickCountStreamP(int[] inp) {
165         int partition=-1;
166         int target = inp.length/2;
167         // Since we have to be working with boxed Integers. We start off by converting.
168         List<Integer> list = Arrays.stream(inp).boxed().collect(Collectors.toList());
169         do {
170             partition = list.get(0);
171             final Integer p = partition;
172             Map<Boolean, List<Integer>> res = list.stream().skip(1).parallel()
173                 .collect(Collectors.partitioningBy(i -> i < p));
174 
175             List<Integer> smaller = res.get(true);
176             List<Integer> bigger = res.get(false);
177 
178             if (smaller.size() == target) break;
179             if (smaller.size() > target) list = smaller;
180             else {
181                 target=target-smaller.size()-1;
182                 list = bigger;
183             }
184         } while( true );
185         return partition; // we are on target
186     }
187 
188     public static int quickCountStream(int[] inp) {
189         int partition=-1;
190         int target = inp.length/2;
191         // Since we have to be working with boxed Integers. We start off by converting.
192         List<Integer> list = Arrays.stream(inp).boxed().collect(Collectors.toList());
193         do {
194             partition = list.get(0);
195             final Integer p = partition;
196             Map<Boolean, List<Integer>> res = list.stream().skip(1)
197                 .collect(Collectors.partitioningBy(i -> i < p));
198 
199             List<Integer> smaller = res.get(true);
200             List<Integer> bigger = res.get(false);
201 
202             if (smaller.size() == target) break;
203             if (smaller.size() > target) list = smaller;
204             else {
205                 target=target-smaller.size()-1;
206                 list = bigger;
207             }
208         } while( true );
```

```
209         return partition; // we are on target
210     }
211 
212     public static final int threadCount = 4;
213     public static void main( String [] args ) {
214         SystemInfo();
215         int a[] = new int[Integer.parseInt(args[0])];
216         Random rnd = new Random();
217         if( args.length == 1 ) {
218             int nrIt = 10;
219             for(int ll=0;ll<nrIt;ll++) {
220                 rnd.setSeed(23434+ll); // seed
221                 for(int i=0;i<a.length;i++) a[i] = rnd.nextInt(4*a.length);
222                 final int ra = quickCountRec(a,a.length/2);
223                 final int rb = medianPSort(a);
224                 if( ra !=rb ) {
225                     System.out.println(ll);
226                     System.out.println(ra);
227                     System.out.println(rb);
228                     System.exit(0);
229                 }
230             }
231             System.out.println();
232         } else {
233             rnd.setSeed(23434+Integer.parseInt(args[1])); // seed
234             for(int i=0;i<a.length;i++) a[i] = rnd.nextInt(4*a.length);
235             System.out.println(medianPSort(a));
236             System.out.println(quickCountRec(a,a.length/2));
237         }
238         double d=0.0;
239         d += Mark9("serial sort", a.length, x -> medianSort(a));
240         d += Mark9("parall sort", a.length, x -> medianPSort(a));
241         d += Mark9("serial qsel", a.length, x -> quickSelect(a));
242         d += Mark9("ser countRc", a.length,x -> quickCountRec(a,a.length/2));
243         d += Mark9("ser countIt", a.length,x -> quickCountIt(a));
244         d += Mark9("par countIt", a.length,x -> quickCountItTask(a));
245         d += Mark9("countStream", a.length,x -> quickCountStream(a));
246         d += Mark9("countStreamP", a.length,x -> quickCountStreamP(a));
247         System.out.println(d);
248     }
249 
250     public static double Mark9(String msg, int size, IntToDoubleFunction f) {
251         int n = 5, count = 1, totalCount = 0;
252         double dummy = 0.0, runningTime = 0.0, st = 0.0, sst = 0.0;
253         do {
254             count *= 2;
255             st = sst = 0.0;
256             for (int j=0; j<n; j++) {
257                 Timer t = new Timer();
258                 for (int i=0; i<count; i++)
259                     dummy += f.applyAsDouble(i);
260                 runningTime = t.check();
261                 double time = runningTime * 1e9 / count; // microseconds
```

```
                st += time;
                sst += time * time;
                totalCount += count;
            }
        } while (runningTime < 0.25 && count < Integer.MAX_VALUE/2);
        double mean = st/n/size, sdev = Math.sqrt((sst - mean*mean*n)/(n-1))/size;
        System.out.printf("%-25s %15.1f ns %10.2f %10d%n", msg, mean, sdev, count);
        return dummy / totalCount;
    }

    public static void SystemInfo() {
        System.out.printf("# OS: %s; %s; %s%n",
                System.getProperty("os.name"),
                System.getProperty("os.version"),
                System.getProperty("os.arch"));
        System.out.printf("# JVM: %s; %s%n",
                System.getProperty("java.vendor"),
                System.getProperty("java.version"));
        // The processor identifier works only on MS Windows:
        System.out.printf("# CPU: %s; %d \"cores\"%n",
                System.getenv("PROCESSOR_IDENTIFIER"),
                Runtime.getRuntime().availableProcessors());
        java.util.Date now = new java.util.Date();
        System.out.printf("# Date: %s%n",
                new java.text.SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ssZ").format(now));
    }
}
```

8.3 MyUnionFind.java

```
import java.util.concurrent.atomic.*;
import java.util.concurrent.*;
import java.util.*;

public class MyUnionFind {
    public static void main(String[] args) throws Exception {
        final int itemCount = 10_000;
        {   // Example of a working execution
            UnionFindTest test = new UnionFindTest();
            test.sequential(new FineUnionFind(5));
            test.concurrent(itemCount, new FineUnionFind(itemCount));
            test.deadlock(itemCount, new FineUnionFind(itemCount));
        }
        {   // Question 4.3
            UnionFindTest test = new UnionFindTest();
            test.sequential(new BogusFineUnionFind(5));
            test.concurrent(itemCount, new BogusFineUnionFind(itemCount));
            test.deadlock(itemCount, new BogusFineUnionFind(itemCount));
        }
    }
}
```

```
22
23  interface UnionFind {
24      int find(int x);
25      void union(int x, int y);
26      boolean sameSet(int x, int y);
27  }
28
29  // Test of union-find data structures, adapted from Florian Biermann's
30  // MSc thesis, ITU 2014
31  class UnionFindTest extends Tests {
32
33      public void sequential(UnionFind uf) throws Exception {
34          System.out.printf("Testing %s ... ", uf.getClass());
35          // Find
36          assertEquals(uf.find(0), 0);
37          assertEquals(uf.find(1), 1);
38          assertEquals(uf.find(2), 2);
39          // Union
40          uf.union(1, 2);
41          assertEquals(uf.find(1), uf.find(2));
42
43          uf.union(2, 3);
44          assertEquals(uf.find(1), uf.find(2));
45          assertEquals(uf.find(1), uf.find(3));
46          assertEquals(uf.find(2), uf.find(3));
47
48          uf.union(1, 4);
49          assertEquals(uf.find(1), uf.find(2));
50          assertEquals(uf.find(1), uf.find(3));
51          assertEquals(uf.find(2), uf.find(3));
52          assertEquals(uf.find(1), uf.find(4));
53          assertEquals(uf.find(2), uf.find(4));
54          assertEquals(uf.find(3), uf.find(4));
55      }
56
57      public void deadlock(final int size, final UnionFind uf) throws Exception {
58          final int[] numbers = new int[size];
59          for (int i = 0; i < numbers.length; ++i) numbers[i] = i;
60          // Populate threads
61          final int threadCount = 32;
62          final CyclicBarrier startBarrier = new CyclicBarrier(threadCount+1),
63                stopBarrier = startBarrier;
64          Collections.shuffle(Arrays.asList(numbers));
65          for (int i = 0; i < threadCount; ++i) {
66              final boolean reverse = i%2==0;
67              Thread ti = new Thread(new Runnable() { public void run() {
68                  try { startBarrier.await(); } catch (Exception exn) { }
69                      for (int j=0; j<100; j++)
70                          for (int i = 0; i < numbers.length - 1; ++i)
71                              if (reverse) uf.union(numbers[i + 1], numbers[i]);
72                              else uf.union(numbers[i], numbers[i + 1]);
73                  try { stopBarrier.await(); } catch (Exception exn) { }
74              }});
```

```
            ti.start();
        }
        startBarrier.await();
        stopBarrier.await();
        final int root = uf.find(0);
        for (int i : numbers)
            assertEquals(uf.find(i), root);
        System.out.println("No deadlocks");
    }

    public void concurrent(final int size, final UnionFind uf) throws Exception {
        final int[] numbers = new int[size];
        for (int i = 0; i < numbers.length; ++i)
            numbers[i] = i;
        // Populate threads
        final int threadCount = 32;
        final CyclicBarrier startBarrier = new CyclicBarrier(threadCount+1),
              stopBarrier = startBarrier;
        Collections.shuffle(Arrays.asList(numbers));
        for (int i = 0; i < threadCount; ++i) {
            Thread ti = new Thread(new Runnable() { public void run() {
                try { startBarrier.await(); } catch (Exception exn) { }
                for (int j=0; j<100; j++)
                    for (int i = 0; i < numbers.length - 1; ++i)
                        uf.union(numbers[i], numbers[i + 1]);
                try { stopBarrier.await(); } catch (Exception exn) { }
            }});
            ti.start();
        }
        startBarrier.await();
        stopBarrier.await();
        final int root = uf.find(0);
        for (int i : numbers)
            assertEquals(uf.find(i), root);
        System.out.println("passed");
    }
}

class Tests {
    public static void assertEquals(int x, int y) throws Exception {
        if (x != y)
            throw new Exception(String.format("ERROR: %d not equal to %d%n", x, y));
    }

    public static void assertTrue(boolean b) throws Exception {
        if (!b)
            throw new Exception(String.format("ERROR: assertTrue"));
    }
}
// Fine-locking union-find. Union and sameset lock on the intrinsic
// locks of the two root Nodes involved. Find is wait-free, takes no
// locks, and performs no compression.

```

```
128 // The nodes[] array entries are never updated after initialization
129 // inside the constructor, so no need to worry about their visibility.
130 // But the fields of Node objects are written (by union and compress
131 // while holding locks), and read by find without holding locks, so
132 // must be made volatile.
133 class FineUnionFind implements UnionFind {
134     private final Node[] nodes;
135 
136     public FineUnionFind(int count) {
137         this.nodes = new Node[count];
138         for (int x=0; x<count; x++)
139             nodes[x] = new Node(x);
140     }
141 
142     public int find(int x) {
143         while (nodes[x].next != x)
144             x = nodes[x].next;
145         return x;
146     }
147 
148     public void union(final int x, final int y) {
149         while (true) {
150             int rx = find(x), ry = find(y);
151             if (rx == ry)
152                 return;
153             else if (rx > ry) {
154                 int tmp = rx; rx = ry; ry = tmp;
155             }
156             // Now rx < ry; take locks in consistent order
157             synchronized (nodes[rx]) {
158                 synchronized (nodes[ry]) {
159                     // Check rx, ry are still roots, else restart
160                     if (nodes[rx].next != rx || nodes[ry].next != ry)
161                         continue;
162                     if (nodes[rx].rank > nodes[ry].rank) {
163                         int tmp = rx; rx = ry; ry = tmp;
164                     }
165                     // Now nodes[rx].rank <= nodes[ry].rank
166                     nodes[rx].next = ry;
167                     if (nodes[rx].rank == nodes[ry].rank)
168                         nodes[ry].rank++;
169                     compress(x, ry);
170                     compress(y, ry);
171                 } }
172         }
173     }
174 
175     // Assumes lock is held on nodes[root]
176     private void compress(int x, final int root) {
177         while (nodes[x].next != x) {
178             int next = nodes[x].next;
179             nodes[x].next = root;
180             x = next;
```

```
181         }
182     }
183 
184     public boolean sameSet(int x, int y) {
185         return find(x) == find(y);
186     }
187 
188     class Node {
189         private volatile int next, rank;
190 
191         public Node(int next) {
192             this.next = next;
193         }
194     }
195 }
196 
197 // Bogus Fine-locking union-find. Union and sameset lock on the intrinsic
198 // locks of the two root Nodes involved. Find is wait-free, takes no
199 // locks, and performs no compression.
200 
201 // The nodes[] array entries are never updated after initialization
202 // inside the constructor, so no need to worry about their visibility.
203 // But the fields of Node objects are written (by union and compress
204 // while holding locks), and read by find without holding locks, so
205 // must be made volatile.
206 
207 class BogusFineUnionFind implements UnionFind {
208     private final Node[] nodes;
209 
210     public BogusFineUnionFind(int count) {
211         this.nodes = new Node[count];
212         for (int x=0; x<count; x++)
213             nodes[x] = new Node(x);
214     }
215 
216     public int find(int x) {
217         while (nodes[x].next != x)
218             x = nodes[x].next;
219         return x;
220     }
221 
222     public void union(final int x, final int y) {
223         while (true) {
224             int rx = find(x), ry = find(y);
225             if (rx == ry)
226                 return;
227             synchronized (nodes[rx]) {
228                 synchronized (nodes[ry]) {
229                     // Check rx, ry are still roots, else restart
230                     if (nodes[rx].next != rx || nodes[ry].next != ry)
231                         continue;
232                     if (nodes[rx].rank > nodes[ry].rank) {
233                         int tmp = rx; rx = ry; ry = tmp;
```

```
                }
                // Now nodes[rx].rank <= nodes[ry].rank
                nodes[rx].next = ry;
                if (nodes[rx].rank == nodes[ry].rank)
                    nodes[ry].rank++;
                compress(x, ry);
                compress(y, ry);
            }
        }
    }

    // Assumes lock is held on nodes[root]
    private void compress(int x, final int root) {
        while (nodes[x].next != x) {
            int next = nodes[x].next;
            nodes[x].next = root;
            x = next;
        }
    }

    public boolean sameSet(int x, int y) {
        return find(x) == find(y);
    }

    class Node {
        private volatile int next, rank;
        public Node(int next) {
            this.next = next;
        }
    }
}
```

8.4 MyStack.java

```
import java.lang.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
public class MyStack<T> {
    private final Object lock;
    private final List<LinkedList<T>> stacks;
    private static final int STRIPES = 32;

    public MyStack(){
        lock = new Object();
        stacks = new ArrayList<LinkedList<T>>();
        for(int i = 0; i < STRIPES; i++)
            stacks.add(new LinkedList<T>());
    }

```

```
17      public void push(T obj) {
18          int stripe = Thread.currentThread().hashCode()%STRIPES;
19          LinkedList<T> stack = stacks.get(stripe);
20          synchronized(stack){
21              stack.push(obj);
22          }
23      }
24
25      public T pop() {
26          int stripe = Thread.currentThread().hashCode()%STRIPES;
27          for (int i = stripe; i < stripe+STRIPES; i++){
28              LinkedList<T> stack = stacks.get(i%STRIPES);
29              synchronized(stack){
30                  if(stack.size() == 0) continue;
31                  return i == stripe ? stack.pop() : stack.removeLast();
32              }
33          }
34          return null;
35      }
36
37      public static void concurrentTest(final int size, final int threads,
38              MyStack<Integer> stack) throws Exception {
39          final CyclicBarrier startBarrier = new CyclicBarrier(threads+1),
40                stopBarrier = startBarrier;
41
42          final int range = size/threads;
43          for (int i = 0; i < threads; ++i) {
44              final int nr = i;
45              Thread ti = new Thread(new Runnable() { public void run() {
46                  try { startBarrier.await(); } catch (Exception exn) { }
47                      for(int j = range*nr; j<range*nr+range; j++)
48                          stack.push(j);
49                  try { stopBarrier.await(); } catch (Exception exn) { }
50              }});
51              ti.start();
52          }
53          startBarrier.await();
54          stopBarrier.await();
55          startBarrier.reset();
56          stopBarrier.reset();
57
58          final Set<Integer> pops = ConcurrentHashMap.newKeySet();
59          for (int i = 0; i < threads; ++i) {
60              final int nr = i;
61              Thread ti = new Thread(new Runnable() { public void run() {
62                  try { startBarrier.await(); } catch (Exception exn) { }
63                      for(int j = range*nr; j<range*nr+range; j++)
64                          pops.add(stack.pop());
65                  try { stopBarrier.await(); } catch (Exception exn) { }
66              }});
67              ti.start();
68          }
69
```

```
70          startBarrier.await();
71          stopBarrier.await();
72
73          if (pops.size() == size) System.out.println("Concurrency works :)");
74          else System.out.println("Concurrency doesn't work :(");
75      }
76
77      public static void testOrder(int n, MyStack<Integer> stack){
78          final AtomicBoolean working = new AtomicBoolean(true);
79          Thread A = new Thread(new Runnable() { public void run() {
80              for(int i=0; i<n; i++) stack.push(i);
81          }});
82          Thread B = new Thread(new Runnable() { public void run() {
83              for(int i=0; i<n; i++)
84                  working.compareAndSet(true, n-1-i == stack.pop());
85          }});
86          try {
87              A.start(); A.join();
88              B.join(); B.join();
89          } catch (Exception e) {
90              System.out.println("Order dies >:(");
91          }
92          if(working.get()) System.out.println("Order works :)");
93          else System.out.println("Order doesn't work :(");
94      }
95
96      public static void main(String[] args){
97          int size = 10_000_000;
98          int threads = 32;
99          MyStack<Integer> stack = new MyStack<Integer>();
100         testOrder(size, stack);
101         try {
102             concurrentTest(size, threads, stack);
103         } catch (Exception e){
104             System.out.println("Concurrent test died >:(");
105         }
106         System.exit(0);
107     }
108 }
```