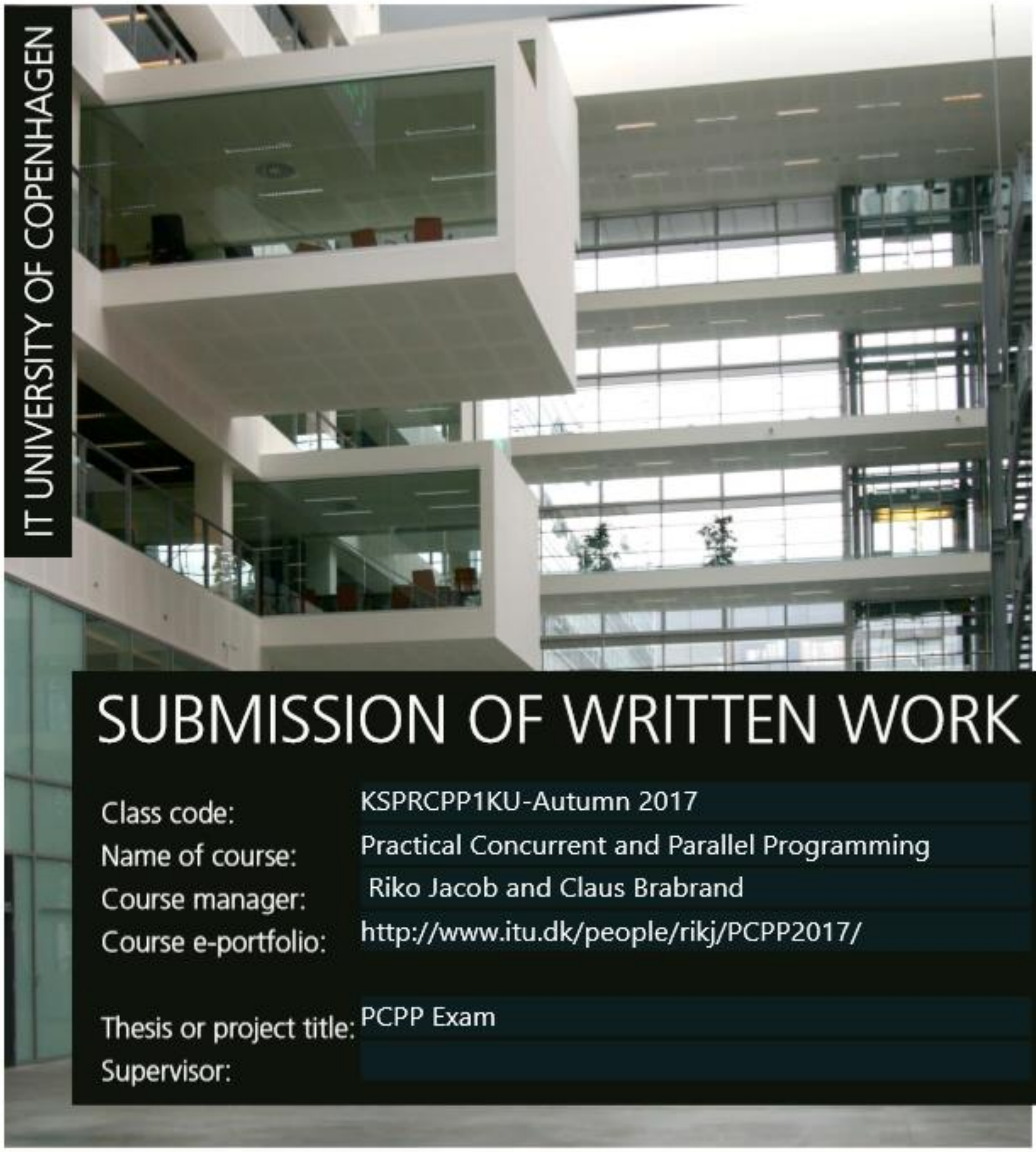


SUBMISSION OF WRITTEN WORK

Class code:	KSPRCPP1KU-Autumn 2017
Name of course:	Practical Concurrent and Parallel Programming
Course manager:	Riko Jacob and Claus Brabrand
Course e-portfolio:	http://www.itu.dk/people/rikj/PCPP2017/
Thesis or project title:	PCPP Exam
Supervisor:	

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Dennis Thinh Tan Nguyen	01/04-1993	dttn@itu.dk
2.			@itu.dk
3.			@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk

Dennis Thinh Tan Nguyen
PCPP Exam 2017
dttn@itu.dk

12/12-2017

Contents

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

Dennis Thinh Tan Nguyen / 11-12-2017

[illegible]

Question 1

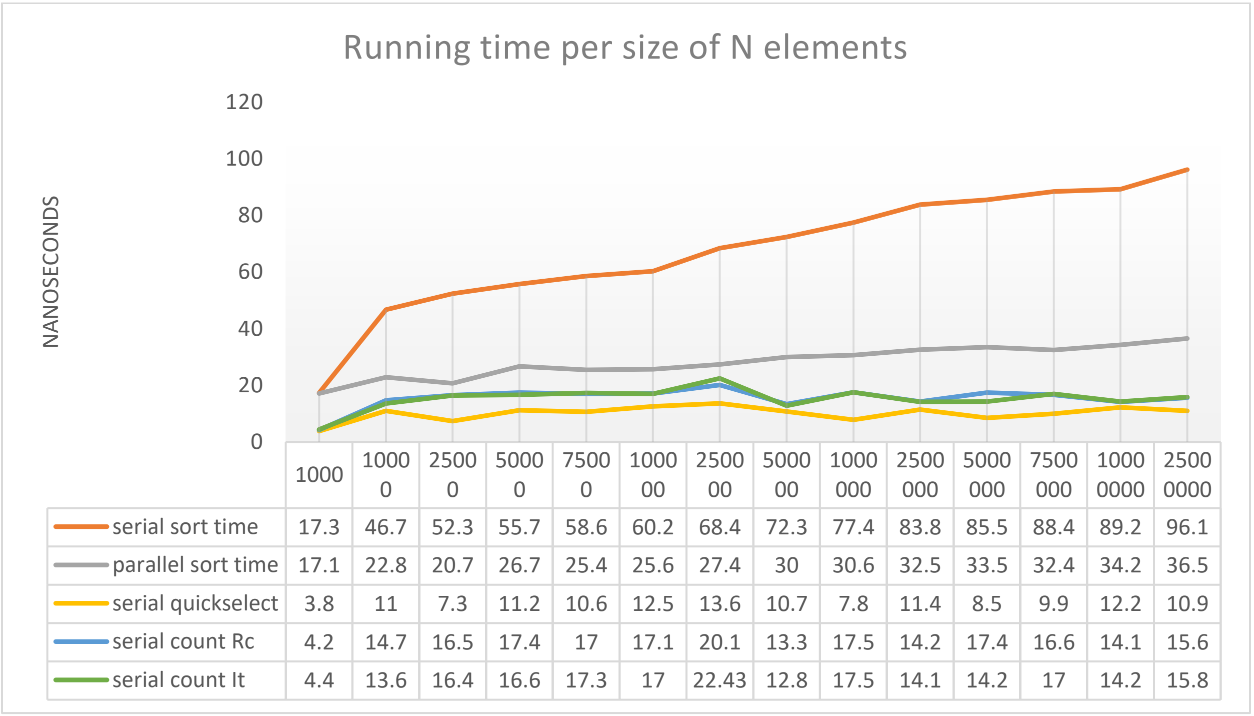
1:

I have produced two graphs where Figure 1 illustrates the total running time about an array N size. Figure 2 Illustrates the running time per element of an array of N size. Looking at Figure 1, it can be seen that serial and parallel sort time is almost equal when sorting an array of 1000 in size. However, as the size of the array is increasing the running time for serial sort is increasing at a linear rate that is significantly higher than parallel sort. This is expected since the sorting of the parallel sort is done in parallel and may thus have a higher throughput compared to sorting the array serially.

Comparing Quickselect, count recursive and count iterative it can be seen that quickselect has the fastest running time whereas count recursive and count iterative is a bit slower. Also, count recursive and count iterative has almost equal running time. This is expected since the implementation is almost must the same but the main difference is one is being recursive, and the other is iterative.

Overall using quicksort either in parallel or serial is slower than using quick select and its alternate implementations due to the required sorting of the whole array before selecting element N/2.

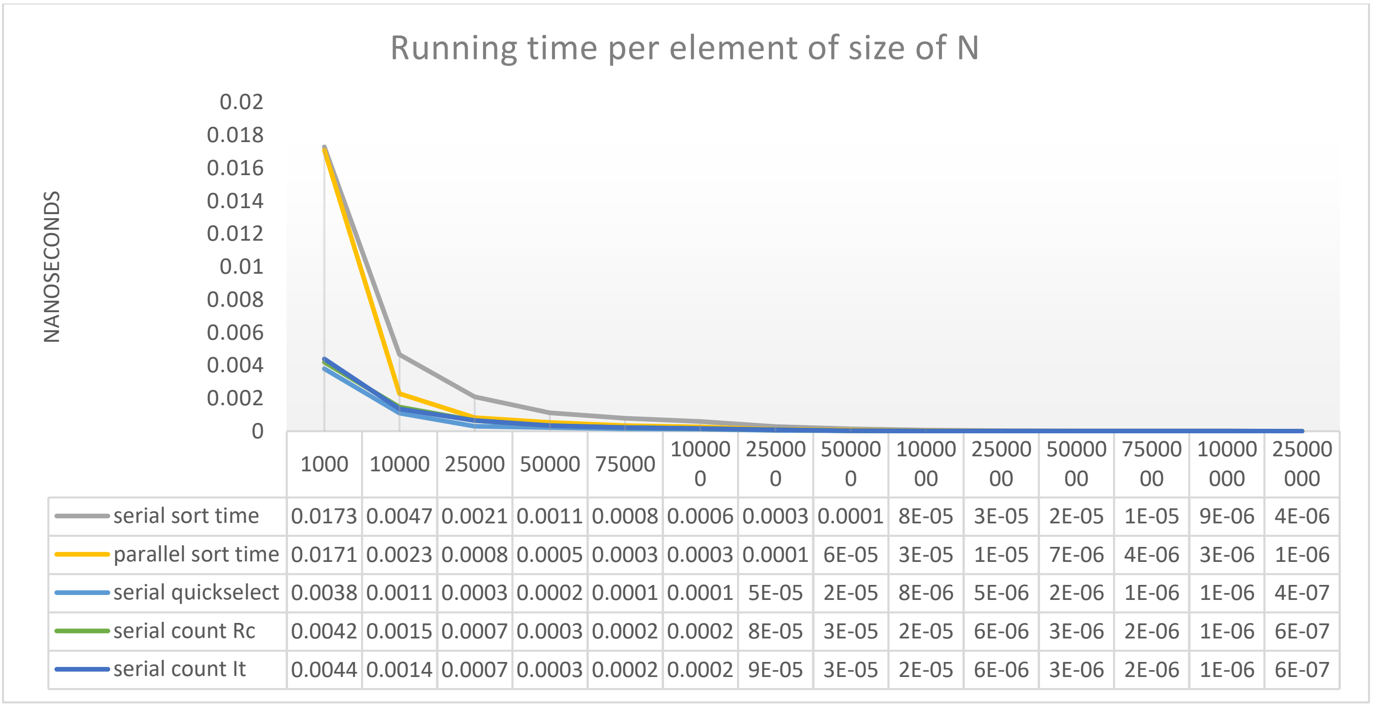
With quick select, no sorting is required as it selects the median in an unsorted set of N elements.



	1000	10000	25000	50000	75000	100000	250000	500000	1000000	2500000	5000000	7500000	10000000	25000000
serial sort time	17.3	46.7	52.3	55.7	58.6	60.2	68.4	72.3	77.4	83.8	85.5	88.4	89.2	96.1
parallel sort time	17.1	22.8	20.7	26.7	25.4	25.6	27.4	30	30.6	32.5	33.5	32.4	34.2	36.5
serial quickselect	3.8	11	7.3	11.2	10.6	12.5	13.6	10.7	7.8	11.4	8.5	9.9	12.2	10.9
serial count Rc	4.2	14.7	16.5	17.4	17	17.1	20.1	13.3	17.5	14.2	17.4	16.6	14.1	15.6
serial count It	4.4	13.6	16.4	16.6	17.3	17	22.43	12.8	17.5	14.1	14.2	17	14.2	15.8

Figure 1 - Running per size N

Looking at Figure 2, it can be seen that the running time per elements improves as the number of N elements increases. However, this improvement reaches a point where it is constant. Based on Figure 2 with elements N where 1 <= N <= ~100000 it can be seen that all selection methods start out slow but after 100000 elements the running time is constant.



	1000	10000	25000	50000	75000	100000	250000	500000	1000000	2500000	5000000	7500000	10000000	25000000
serial sort time	0.0173	0.0047	0.0021	0.0011	0.0008	0.0006	0.0003	0.0001	8E-05	3E-05	2E-05	1E-05	9E-06	4E-06
parallel sort time	0.0171	0.0023	0.0008	0.0005	0.0003	0.0003	0.0001	6E-05	3E-05	1E-05	7E-06	4E-06	3E-06	1E-06
serial quickselect	0.0038	0.0011	0.0003	0.0002	0.0001	0.0001	5E-05	2E-05	8E-06	5E-06	2E-06	1E-06	1E-06	4E-07
serial count Rc	0.0042	0.0015	0.0007	0.0003	0.0002	0.0002	8E-05	3E-05	2E-05	6E-06	3E-06	2E-06	1E-06	6E-07
serial count It	0.0044	0.0014	0.0007	0.0003	0.0002	0.0002	9E-05	3E-05	2E-05	6E-06	3E-06	2E-06	1E-06	6E-07

Figure 2 - Running per element of Size N

See ***Appendix Q1 – System Information*** for my system specifications.

2:

Counting elements – *See appendix Q1.2 - and Q1.1.1 - Count parallel* for code

I have created a new method called quickCountItParallel that takes an int array as input. The method is very similar the provided quickCountIt method. The quickCountItParallel uses a helper method called countParallel that will do the countings in parallel. I use an executer with a cached thread pool to invoke N number of callable<integer> tasks.

The number of task is defined based on my input parameter taskCount. For each task, I calculate the "from" index and the "to" index which defines the section of the array the tasks works in. Each task has their local count which will then be returned when they finish counting.

Each task is wrapped in a future that allows me to await and synchronize their results which I sum up in a local variable that I later return.

Regarding visibility, since each thread does not mutate a shared state but only a local counter and it only reads from their own section of the array, I can assume that there are no visibility issues here.

Filtering elements – *See appendix Q1.2 - and Q1.1.2 - filter parallel* for code

They way filtering is implement is similar to the count parallel method. I use N number callable tasks that returns Void in their wrapped future. The reason for this is so I can use the invokeAll method from the executer and await them all. The task are created based on if count < target or count > target, since both conditions have a different filter operations.

```
if (count > target) { tasks.add(() -> {…}}
else if (count < target) { tasks.add(() -> {…}}
```

Also the filtered results are stored in a final int[] called m, and is declared based on count < target or vice versa.

```
final int[] m = count > target ? new int[count] : new int[n - count - 1];
```

During filtering, the task do access the same shared dataset but the dataset is declared final to prevent it from mutating and also be usable in the lambda notation. Yet this does not ensure immutability of the stored values, but this is fine since the threads only read from the dataset and does not write to it.

When the filtering is done, the filtered results *m* I s returned back to the caller which will set that as the new input and do another iteration. When all is done, that is when count == target, then the while loop is broken and the median is returned

```
//filter elements
 try {
   if(count == target){break;}
   int[] filtered = filterParallel(inp,count,target,p,n,taskCount);
   if(count < target){target = target-count-1;}
   inp = filtered;
 }
```

Regarding visibility, each thread does mutate a shared state J which I declare as an atomic integer to ensure that whenever a thread increment J, the operation is atomic and visible to the other threads.

```
final AtomicInteger j = new AtomicInteger();
…

if (inp[i] < p) {
    m[j.getAndIncrement()] = inp[i]; //No lock for inp since only read
}
```

Given input of 1000 elements and 4 tasks my results were 1894 which is also what the serialized quickCountIt returns and thus the implementation is assumed to be correct.

3:

See ***Appendix Q1 – System Information*** for my system specifications.

Thread/Size (serial does not use threads**)**	**Serial time ns**	**Parallel time ns**
1/100	4.7	26285.0
1/500	3.1	2381.8
1/1000	5.1	3164.5
1/100000	18.0	84.9
1/10000000	14.1	26.7
Thread/Size (serial does not use threads**)**	**Serial time ns**	**Parallel time ns**
2/100	4.7	32623.5
2/500	3.1	6214.2
2/1000	5.0	5790.7
2/100000	18.8	175.1
2/10000000	15.0	65.6
Thread/Size (serial does not use threads**)**	**Serial time ns**	**Parallel time ns**
4/100	4.7	59360.4
4/500	3.1	9963.7
4/1000	5.1	8248.1
4/100000	18.8	197.8
4/10000000	14.3	59.6

Based on the benchmark results, the serial implementation is significantly faster than the parallel implementation. Especially when the size of elements is less low. One may assume that there might be too much overhead when creating the tasks and synchronizing them again, whereas in the serial implementation this overhead is not a problem.

However, if you increase the size of elements, the running time of the parallel implementation is amortized over a larger size of elements.

4:

See ***Appendix Q1 – System Information*** for my system specifications.

I removed the do-while loop to ensure only one iteration was executed. Again, the same relation between task, size and overhead is the same as the benchmark above

Thread/Size (serial does not use threads**)**	**Serial time ns**	**Parallel time ns**
1/100000	3.1	12.3
1/1000000	7.9	10.3
1/10000000	3.9	8.7
Thread/Size (serial does not use threads**)**	**Serial time ns**	**Parallel time ns**
2/100000	3.0	27.9
2/1000000	3.1	25.5
2/10000000	3.9	23.7
Thread/Size (serial does not use threads**)**	**Serial time ns**	**Parallel time ns**
4/100000	3.0	25.8
4/1000000	3.6	25.5
4/10000000	3.9	18.6

5:

Based on the benchmark results the threshold is set to 100.000. Thus, if the size of elements are smaller than the threshold, the serial version is executed.

```
public static int quickCountItParallelHybrid(int[] inp) {
  final int threshold = 100000;
  if(inp.length < threshold){
    return quickCountIt(inp);
  }
  final int taskCount = 4;
  int p, count;
  int target = inp.length/2;
  do {
    p=inp[0];
    int n=inp.length;
    //Counting elements smaller than pivot
    try {
      count = countParallel(inp,p,n,taskCount);
    } catch (Exception e) {
      return -1;
    }
    //filter elements
    try {
      if(count == target){break;}
      int[] filtered = filterParallel(inp,count,target,p,n,taskCount);
      if(count < target){target = target-count-1;}
      inp = filtered;
    }
    catch (Exception ignored) {}
  } while( true );
  return p;
}
```

Question 2

1:

```java
public static int quickCountItStream(int[] inp){
  int p=-1, count=0, n=inp.length;
  int target = n/2;
  do {
    p =inp[0];
    n=inp.length;

    //Counting elements smaller than pivot
    final int pivot = p;
    IntStream countStream = Arrays.stream(inp);
    IntStream countResults = countStream.skip(1).filter(e -> e < pivot);
    count = (int) countResults.count();

    //Filter part hidden…
       break;
  } while( true );
  return p; // we are on target
}
```

2:

```java
public static int quickCountItStream(int[] inp){
  int p=-1, count=0, n=inp.length;
  int target = n/2;
  do {
    p =inp[0];
    n=inp.length;

    //Counting part hidden…
 IntStream filterStream = Arrays.stream(inp);
 if(count > target) {
  int[] small = filterStream.skip(1).filter(e -> e < pivot).toArray();
  inp = small;
  continue;
 }
  if(count < target) {
   int[] larger = filterStream.skip(1).filter(e -> e >= pivot).toArray();
   inp = larger;
   target=target-count-1;
   continue;
  }
       break;
 } while( true );
return p; // we are on target
}
```

3:

Given input of 1000 elements my results were 1894 which is also what the serialized quickCountIt returns and thus the implementation is assumed to be correct.

4:

```
//Count part
IntStream cStream = Arrays.stream(inp);
IntStream countResults = cStream.skip(1).parallel().filter(e -> e < pivot);
count = (int) countResults.count();

//Stream part
IntStream fStream = Arrays.stream(inp);
int[] small = fStream.skip(1).parallel().filter(e -> e < pivot).toArray();
inp = small;
```

5:

```
public static int quickCountItStreamPartitioned(int[] inp){
  int p=-1, count=0, n=inp.length;
  int target = n/2;
  do {
    p = inp[0];
    n=inp.length;
    //Filter and count elements
    final int pivot = p;

    Map<Boolean, List<Integer>> elements =
Arrays.stream(inp).skip(1).boxed().collect(Collectors.partitioningBy(e ->
e < pivot));
    count =  (elements.get(true)).size();

    if(count > target) {
      inp = unbox(elements.get(true));
      continue;
    }
    if(count < target) {
      inp = unbox(elements.get(false));
      target=target-count-1;
      continue;
    }
    break;
  } while( true );
  return p; // we are on target

}
private static int[] unbox(List<Integer> arr){
  int[] r = new int[arr.size()];
  for (int i = 0; i<arr.size() ;i++){
    r[i] = arr.get(i);
  }
  return r;
}
```

6:

Type	Time in ns
Stream parallel	3823.8
Stream serial	53.1

Based on my findings the parallel implementation is a lot slower the serial implementation. The reason might be the overhead that is involved when splitting the work among several threads and joining them again, whereas in the serial implementation this overhead does not exists.

Question 3

Based on the following description it may be possible to implement such thread based version. Since each thread owns are a specific part of the data it is safe for them to work on the data safely, assumed that the threads do not access each other's data. The mutable shared states are a global pivot and a global counter.

It is assumed that these shared states are accessed atomically through locks or optimistic concurrency such as CAS. For the threads to synchronize probably, barrier.Await () must be called before selecting a pivot point, before counting and before filtering. The reason for this is to avoid some thread to start ahead on counting or filtering when some other threads are still not done with their current operations. If this was not in place, one might risk that some threads may read the wrong values of the global variables

Regarding performance issues, since each thread must wait for the other threads at each barrier point one can assume that the implementation is only as fast as the slowest thread in the thread pool that is still computing. Thus, one may regard such barrier point as bottleneck.

Finally, the issue of load balancing some threads may own parts of the dataset such that the global pivot point specifies a number that requires a longer count or filter operation for such data subset. Therefore, some threads may have a greater load of work than others and thus the work has not been distributed in a balanced manner.

Question 4

1:

The implementation is prone to a dead-lock at the sequenced synchronized keywords due to the risk of threads awaiting each others:

```
synchronized (nodes[rx]) {
  synchronized (nodes[ry]) {
         //…
}
```

2:

Given thread A has acquired a lock on nodes[rx] and tries to get a lock on nodes[ry], meanwhile another thread B has acquired a lock on nodes[ry] right before Thread A has acquired it. Also, given thread B tries to get a lock on nodes[rx] after Thread A has acquired it. It can be stated that Thread A is waiting for thread B to release a lock and Thread B waits for thread A to release a lock, thus a dead-lock is in place.

3:

The below implementation dead-locks after a few iterations. I use a cyclic barrier to ensure both threads start at the same time to increase the chance of them interleaving each other. Both threads try to union the reverse value of each other and at some point in time, both threads would deadlock as expected.

```
public void deadlock() throws InterruptedException, BrokenBarrierException {

    while(true) {
        final UnionFind uf = new BogusFineUnionFind(100);
        final CyclicBarrier startbarrier = new CyclicBarrier(3);
        final CyclicBarrier stopbarrier = new CyclicBarrier(3);

        Thread a = new Thread(() -> {
            try { startbarrier.await(); } catch (Exception exn) { }
            uf.union(20, 55);
            try { stopbarrier.await(); } catch (Exception exn) { }

        });

        Thread b = new Thread(() -> {
            try { startbarrier.await(); } catch (Exception exn) { }
            uf.union(55, 20);
            try { stopbarrier.await(); } catch (Exception exn) { }

        });

        a.start(); b.start();
        startbarrier.await();
        stopbarrier.await();
        System.out.println("NO DEADLOCK");
     }
}
```

Question 5

1:

```java
class MyStack implements Stack<Integer> {

    private final LinkedList<Integer> stack = new LinkedList<>();
    private final Object lock = new Object();
    @Override
    public Integer pop() {
        Integer item ;
        synchronized (lock) {
            try
            {
                item = stack.pop();
            }
            catch (NoSuchElementException ignored) {
                item = null;
            }
        }
        return item;
    }
    @Override
    public void push(Integer item) {
        synchronized (lock){
            stack.push(item);
        }
    }
}

interface Stack<T>{
    T pop();
    void push(T item);
}
```

2:

I had created a sequential that tests the implemented stack which can be seen in the *Appendix Q5.2 – Sequential test of stack.*

I have tested single push and pop operations to see if the correct number of items are pushed and popped and multiple push and pop operation to see if the order of the items is in reverse order of push operations.

Based on my tests, all operation passed and I assume that the implementation is correct.

3:

I had created a concurrent test which uses N number of pusher and N number of poppers which will continuously push K number of random integers and pop k number of integers. Both the pushers and poppers are created as N callable tasks which will return the sum of integers they push or pop. The tests pass if the sum of pushed integers equals to the sum of pupped integers.

Please **see *Appendix Q5.3 – Concurrent test of stack* for implementation.**

Regarding Correctness, the tests passes with : pushers = 16, poppers = 16, push = 10000;

Regarding Performance testing, I use Mark7 to benchmark my stack based on different numbers of pushers and poppers. My following results are listed in the table below

Push = 1000

Pushers threads	**Poppers threads**	**Time in seconds**
1	1	0,0105~
2	2	0.0400~
3	3	0.101~
4	4	0.540~

The running time increases as I increase the number of pushers on poppers. The reason might be due to the high contention when each thread tries to access the same queue and its operations. Therefore the current implementation may not be scalable when using many multiple threads.

4:

For the implementation of my stack that utilizes lock striping, I use a LinkedList array which contains all the stacks, an array of locks as well as the number of locks. The number of locks is the same number of stacks. When popping, the hashcode is retrieved from the thread, and the stripe is calculated by taking lock count modulus hashcode. To ensure each operation is atomic and visible, I retrieve the lock based on the stripe and synchronize my pop operations with the retrieved lock. If the threads current stack is empty, it will try to pop the other stacks with an auxiliary method called **popOthers(int myNum).** The operations and synchronization are almost the same as pop(), but the only difference is that it tries to pop all until one pops a result or all others pops null.

The push operation also retrieves the hashcode from the thread and the stripe is calculated by taking lock count modulus hashcode. The stripe is used to lock the operation when accessing and pushing an element to the assigned stack.The push operation also retrieves the hashcode from the thread and the stripe is calculated by taking lock count modulus hashcode. The stripe is used to lock the operation when accessing and pushing an element to the assigned stack.

See *Appendix Q5.4 – Implementation of stripedStack* for implementation of the lock striping stack implementation.

5:

Below is my benchmark results:

Push = 1000

Pushers threads	Poppers threads	Time in seconds
1	1	0.0201~
2	2	0.0203~
3	3	0.0206~
4	4	0.0236~

It can be see that the running time is almost constant when I increase the number of threads compared to the non-lock-striped implementation that has an increasing running time when thread count increases. Thus, this implementation does improve the overall running time

6:

Below is a snippet of my implementation where I use removeLast() to steal from another stack:

```
private Integer popOthers(int myNum){
  Integer item = null;
  for(int i = 0 ; i <locks.length;i++{
     if(i == myNum){continue;}
          try{
              synchronized (locks[i]){
                  item = stacks[i].removeLast(); <-- Steals from other
              }
              break;
          }catch (NoSuchElementException ignored) {}
 }
 return item;
```

See *Appendix Q5.6 – Implementation of stripedStack with stealing* for full implementation of a stealing stack

7:

Based on the requirement *"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)".* If thread B steals from thread A's stack, then Thread B does not pop the two elements in reverse order, such that Thread A push *x* and *y, while B* pops *y* and *x*.

8:

I create two threads where one pushes 1,2,3,.., N such that the stack is in ascending order and the other thread will the pop the value. Thus it compares its previous popped value X with its current value Y such that X == Y-1. Also to ensure that the stacks manage to be populated I call barrier.await() after it has pushed its elements. If this is not in place, the popping thread may pop the biggest (first) value since the pushing thread has not pushed the remaining values and thus compare two wrong values.

I had tested this with the wrong stealing stack which fails the test and also with the correct implementation of the stack which passes it.

See *Appendix Q5.8 – Wrong order test* for my test implementation.

Question 6

1:

Assume there are two threads A and B such that they try to access consensus() concurrently. Thread A inputs the integer 2 and since 2 is greater than > -1 which is the initial state, the thread runs through the method until the else statement. Then assume Thread B also inputs the integer 1 before Thread A manages to run through the else statement, then the state would still be -1, and thus Thread B would also reach the else statement. Consequently, Thread A would write to the state but Thread B would then overwrite it again. This form of race condition would then violate the consistent condition of consensus since Thread B did not decide on the same value as Thread A.

2:

According Herlihy and Shavitr definition:

> *"We are interested in wait-free solutions to the consensus problem, that is, wait-free concurrent implementation of consensus objects. (…) any class that implements consensus in a wait-free manner a consensus protocol. "*[1]

Taken this definition into consideration, by utilizing the synchronized keyword, the class is no longer wait-free since any call to the method consensus would make all other calling threads wait. Thus by Herlihy and Shavitr definition of consensus protocol, this implementation violates such definition.

An example of this execution would be; Given two Threads A and B such that they call the method consensus concurrently. Either Thread A or B gain access to the method. Assume that Thread B gain access first, then Thread A must wait for thread B to finish its operations. Thus the implementation is not wait-free and not a consensus protocol.

3:

The problem with variant is the risk of it running in an infinite loop. Assume that two Threads A and B access the CAS consensus variant concurrently, and both threads read the state as < 0. Both threads would enter the while loop. Assume Thread B performs a successful CAS operation on the state then thread A would fail its CAS operation since the expected value is no longer -1. Thus the CAS would fail, the if statement would be false and finally Thread A would retry forever since nothing is returned and it cannot break the loop. [2]

A proposed implementation might be as following.

```java
AtomicInteger state = new AtomicInteger(-1);
int consensus(int x) { //invariant: x > 0
    assert(x>0);
    final int s = state.get();
    if(s > 0) return s;
    state.compareAndSet(-1,x); //Try set state, if fail -> state has already been set
    return state.get();
}
```

[1] Herlihy and Shavit: The Art of Multiprocessor Programming. Revised first edition, Morgan Kaufmann 2012. P.100 chapter 5.1

[2] Assume that line 4 of the proposed code should return state.get() and state since the consensus method returns an int and not state object. (Otherwise the code would not compile and run)

Question 7

For this assignment, I implemented the system in JAVA+AKKA as close as the specified ERLANG implementation in regards to functionality. For the test sessions, one of each actor is used. That is a register, a sender, and a receiver. Each actor is initialized and executed in the same order as specified in the ERLANG implementation

The output results after few runs are as following:

- public key: 21 private key: 5 clearText:SECRET encrypted: NZXMZO decrypted message:SECRET
- public key: 13 private key: 13 clearText:SECRET encrypted: FRPERG decrypted message:SECRET
- public key: 10 private key: 16 clearText:SECRET encrypted: COMBOD decrypted message:SECRET

Based on the output it can be stated that the implemented solution does generate the correct keys, encrypt and decrypt the correct message.

Please see ***Appendix Q7.0 – Message Passing implementation*** for my implementation of the secret communication system.

Appendix

Q1 – System Information

# OS: Windows 10; 10.0; amd64

# JVM: Oracle Corporation; 9.0.1

# CPU: Intel64 Family 6 Model 94 Stepping 3, GenuineIntel; 4 "cores"

# Date: 2017-12-12T12:48:37+0100

Q1.2 - QuickCountItParallel

```
public static int quickCountItParallel(int[] inp) {
  final int taskCount = 1;
  int p, count;
  int target = inp.length/2;
  do {
    p=inp[0];
    int n=inp.length;
    //Counting elements smaller than pivot
    try {
      count = countParallel(inp,p,n,taskCount);
    } catch (Exception e) {
      return -1;
    }
    //filter elements
     try {
       if(count == target){break;}
       int[] filtered = filterParallel(inp,count,target,p,n,taskCount);
       if(count < target){target = target-count-1;}
       inp = filtered;
     }
     catch (Exception ignored) {}
  } while( true );
  return p;
}
```

Q1.1.1 - Count parallel

```
private static int countParallel(int[] inp, int p, int n, int taskCount) throws
InterruptedException, ExecutionException {
  final ExecutorService executor = Executors.newCachedThreadPool();
  final int perTask = n / taskCount;
  List<Callable<Integer>> tasks = new ArrayList<>();
  for (int task=0; task<taskCount; task++) {
    final int from = perTask * task+1;
    final int to = (task+1 <= taskCount-1) ? perTask * (task+1) : n-1;

    //Creating tasks
    tasks.add(() -> {
      int count = 0;
      for(int i=from;i<=to;i++) {
        if (inp[i] < p) {count++;}
      }
      return count;
    });
  }
  int totalCount = 0;

    List<Future<Integer>> futures = executor.invokeAll(tasks);
    for (Future<Integer> task : futures)
      totalCount += task.get();

  executor.shutdown();
  return totalCount;
}
```

Q1.1.2 – Filter parallel

```
private static int[] filterParallel(final int[] inp, final int count, final
int target, int p, int n, int taskCount) throws InterruptedException,
ExecutionException {
  final ExecutorService executor = Executors.newCachedThreadPool();
  final int perTask = n / taskCount;
  List<Callable<Void>> tasks = new ArrayList<>();

  final AtomicInteger j = new AtomicInteger();
  final int[] m = count > target ? new int[count] : new int[n - count - 1];

  for (int task=0; task<taskCount; task++) {
    final int from = perTask * task + 1;
    final int to = (task + 1 <= taskCount - 1) ? perTask*(task + 1) : n - 1;

    //Creating tasks
    if (count > target) {
      tasks.add(() -> {
        for (int i = from; i <= to; i++) {
          if (inp[i] < p) {
              m[j.getAndIncrement()] = inp[i];
          }
        }
        return null; //Done
      });
    }
    else if (count < target) {
      tasks.add(() -> {

        for (int i = from; i <= to; i++) {
            if (inp[i] >= p) {
              m[j.getAndIncrement()] = inp[i];
            }
        }
        return null; //Done
      });
    }
  }
    executor.invokeAll(tasks);
    executor.shutdown();
    return m;
}
```

Q1.1.5 – Hybrid CountIt

```
public static int quickCountItParallelHybrid(int[] inp) {
  final int threshold = 100000;
  if(inp.length < threshold){
    return quickCountIt(inp);
  }
  final int taskCount = 4;
  int p, count;
  int target = inp.length/2;
  do {
    p=inp[0];
    int n=inp.length;
    //Counting elements smaller than pivot
    try {
      count = countParallel(inp,p,n,taskCount);
    } catch (Exception e) {
      return -1;
    }
    //filter elements
    try {
      if(count == target){break;}
      int[] filtered = filterParallel(inp,count,target,p,n,taskCount);
      if(count < target){target = target-count-1;}
      inp = filtered;
    }
    catch (Exception ignored) {}
  } while( true );
  return p;
}
```

Q5.2 – Sequential test of stack

```
class Main{

    public static void main(String[] args){
        Stack<Integer> stack = new MyStack();
        sequentialTest(stack);
    }

    private static void sequentialTest(Stack<Integer> s){
        System.out.println(s.pop()==null);
        s.push(1);
        System.out.println(s.pop()==1);
        s.push(2); s.push(3); s.push(4);
        System.out.println(s.pop()==4);
        System.out.println(s.pop()==3);
        System.out.println(s.pop()==2);
        System.out.println(s.pop()==null);
    }

}
```

Q5.3 – Concurrent test of stack

```
private static void concurrentTest(Stack<Integer> stack) throws Exception{
    ExecutorService executor = Executors.newCachedThreadPool();
    final int pushers = 16, poppers = 16, push = 10000;
    CyclicBarrier barrier = new CyclicBarrier(pushers+poppers+1);
    List<Future<Integer>> popResults = new ArrayList<>();
    List<Future<Integer>> pushResults = new ArrayList<>();

 //Add PopPusher
 for (int i = 0 ; i < pushers ; i++) {
    pushResults.add(executor.submit(()->{
        int numberPushed = 0;
        final Random random = new Random();

        barrier.await();
        for (int j = 0; j < push; j++) {
            int randomInt = random.nextInt();
            stack.push(randomInt);
            numberPushed += randomInt;
        }
        return numberPushed;
    }));
 }
 //Add poppers
 for (int i = 0 ; i < poppers ; i++)     {
    popResults.add(executor.submit(() -> {
        Integer item;
        int sumPopped = 0;
        barrier.await();
        for(int j = 0 ; j < push ; j++){
            item = stack.pop();
            if(item != null) {
                sumPopped += item;
            }
            else{j--;}
        }
        return sumPopped;
    }));
 }
    //Get and sum results
    int sumPush = 0, sumPop = 0;

    barrier.await();

    for (Future<Integer> sum : pushResults) { sumPush = sumPush +sum.get(); }
    for (Future<Integer> sum : popResults) { sumPop  = sumPop + sum.get(); }

    executor.shutdown();
    if(sumPush==sumPop)System.out.println("Concurrency test passed.");
    else System.out.println("Concurrency test failed.");}
```

Q5.4 – Implementation of stripedStack

```
class MyStackLockStriped implements Stack<Integer> {
    private final LinkedList<Integer>[] stacks = new LinkedList[32];
    private final int lockCount;
    private final Object[] locks;

    public MyStackLockStriped() {
        this.lockCount = 32;
        this.locks = new Object[lockCount];
        for (int stripe=0; stripe<lockCount; stripe++) {
            this.locks[stripe] = new Object();
        }
        for (int i=0; i<stacks.length; i++) {
            this.stacks[i] = new LinkedList<>();
        }
    }

    @Override
    public Integer pop() {
     final int stripe = (Thread.currentThread().hashCode())%lockCount,
               hash =stripe;
     Integer item ;
     try{
         synchronized (locks[stripe]) {
             item = stacks[hash].pop();
         }
     }
     catch (NoSuchElementException ignored) {
         item = popOthers(hash);
     }
     return item;
   }
   private Integer popOthers(int myNum){
     Integer item = null;
     for(int i = 0 ; i <locks.length;i++{
        if(i == myNum){continue;}
            try{
                synchronized (locks[i]){
                    item = stacks[i].pop();
                }
                break;
            }catch (NoSuchElementException ignored) {}
    }
    return item;
}
    @Override
    public void push(Integer item) {
        final int stripe = (Thread.currentThread().hashCode())%lockCount,
                  hash = stripe;
        synchronized (locks[stripe]){
            stacks[hash].push(item);
        }
    }
}
```

Q5.6 – Implementation of stripedStack with stealing

```
class MyStackLockStriped implements Stack<Integer> {
    private final LinkedList<Integer>[] stacks = new LinkedList[32];
    private final int lockCount;
    private final Object[] locks;

    public MyStackLockStriped() {
        this.lockCount = 32;
        this.locks = new Object[lockCount];
        for (int stripe=0; stripe<lockCount; stripe++) {
            this.locks[stripe] = new Object();
        }
        for (int i=0; i<stacks.length; i++) {
            this.stacks[i] = new LinkedList<>();
        }
    }

    @Override
    public Integer pop() {
     final int stripe = (Thread.currentThread().hashCode())%lockCount,
               hash =stripe;
     Integer item ;
     try{
         synchronized (locks[stripe]) {
             item = stacks[hash].pop();
         }
     }
     catch (NoSuchElementException ignored) {
         item = popOthers(hash);
     }
     return item;
   }
   private Integer popOthers(int myNum){
     Integer item = null;
     for(int i = 0 ; i <locks.length;i++{
        if(i == myNum){continue;}
            try{
                synchronized (locks[i]){
                    item = stacks[i].removeLast(); <-- Steals from other
                }
                break;
            }catch (NoSuchElementException ignored) {}
    }
    return item;
}
    @Override
    public void push(Integer item) {
        final int stripe = (Thread.currentThread().hashCode())%lockCount,
                  hash = stripe;
        synchronized (locks[stripe]){
            stacks[hash].push(item);
        }
    }
}
```

Q5.8 – Wrong order test

```
private static int concurrentStealTest(Stack<Integer> stack) throws
Exception{
    ExecutorService executor = Executors.newCachedThreadPool();
    final int pushers = 1, poppers =1, push = 10;
    CyclicBarrier barrier = new CyclicBarrier(pushers+poppers+1);
    List<Future<Integer>> popResults = new ArrayList<>();
    List<Future<Integer>> pushResults = new ArrayList<>();
    //Add PopPusher
    for (int i = 0 ; i < pushers ; i++) {
        pushResults.add(executor.submit(()->{
            int numberPushed = 0;
            int element = push;

            for (int j = 0; j < push; j++) {
                stack.push(element);
                numberPushed += element;
                element--;
            }
            barrier.await();
            return numberPushed;
        }));
    }
    //Add poppers
    for (int i = 0 ; i < poppers ; i++)    {
        popResults.add(executor.submit(() -> {
            Integer item;
            int sumPopped = 0, prev = Integer.MIN_VALUE;
            barrier.await();
            for(int j = 0 ; j < push ; j++){
                item = stack.pop();
                if(item != null) {
                    sumPopped += item;
                    if(item-1 == prev || prev == Integer.MIN_VALUE) {
                        prev = item;
                    }
                    else{
                    throw new Exception("WRONG ORDER);
                    }
                }
                else{ j--;}
            }
            return sumPopped;
        }));
    }
    //Get and sum results
    int sumPush = 0, sumPop = 0;
    barrier.await();
    for (Future<Integer> sum : pushResults) {sumPush = sumPush + sum.get(); }
    for (Future<Integer> sum : popResults) {sumPop  = sumPop + sum.get(); }
    executor.shutdown();
    assertEquals(sumPush,+sumPop);
    System.out.println("Concurrency Wrong order test passed.");
    return 1;
}
```

Q7.0 – Message Passing implementation

```
public class SecComSys {
  public static void main(String[] args){

   final ActorSystem system = ActorSystem.create("SecComSys");
      final ActorRef registryActor =
                          system.actorOf(Props.create(Registry.class),"registry");

   final ActorRef receiverActor =
                          system.actorOf(Props.create(Receiver.class),"receiver");

   receiverActor.tell(new InitMesg(registryActor), ActorRef.noSender());

   final ActorRef senderActor = system.actorOf(Props.create(Sender.class),"sender");

   senderActor.tell(new InitMesg(registryActor), ActorRef.noSender());
   senderActor.tell(new CommMesg(receiverActor), ActorRef.noSender());
 }
}
class Registry extends UntypedActor{
    private final HashMap<ActorRef,KeyPair> register = new HashMap<>();
    @Override
    public void onReceive(Object message) throws Exception {
        if(message instanceof RegisterMesg){
            RegisterMesg m = (RegisterMesg) message;
            KeyPair kp = Crypto.keygen();
            register.put(m.pid,kp);
            m.pid.tell(kp,m.pid);
        }
        else if(message instanceof LookUpMesg){
            LookUpMesg m = (LookUpMesg) message;
            KeyPair kp = register.get(m.lookUpPid);
            PubKey pubKey = new PubKey(kp.public_key,m.lookUpPid);
            m.returnPid.tell(pubKey,m.returnPid);
        }
    }
}
class Sender extends UntypedActor{
    ActorRef regstryPid;
    @Override
    public void onReceive(Object message) throws Exception {
        if(message instanceof InitMesg){
            InitMesg init = (InitMesg) message;
            regstryPid = init.registryPid;
        }
        else if(message instanceof CommMesg){
            CommMesg m = (CommMesg) message;
            LookUpMesg lookUp = new LookUpMesg(m.ReceiverPid,this.getSelf());
            regstryPid.tell(lookUp,ActorRef.noSender());
        }
        else if(message instanceof PubKey){
            PubKey m = (PubKey) message;
            String X = "SECRET";
            System.out.println("clearText:"+X);
            String Y = Crypto.encrypt(X,m.publicKey);
            System.out.println("encrypted: "+Y);

            EncryptedMesg em = new EncryptedMesg(Y);
            m.receiverPid.tell(em,ActorRef.noSender());
        }
    }
}
```

```
class Receiver extends UntypedActor{
    private ActorRef registerPid;
    private int pubkey;
    private int privkey;

    @Override
    public void onReceive(Object message) throws Exception {
        if(message instanceof InitMesg){
            InitMesg m = (InitMesg) message;
            registerPid = m.registryPid;
            RegisterMesg registrate = new RegisterMesg(this.getSelf());
            registerPid.tell(registrate,ActorRef.noSender());
        }
        else if(message instanceof KeyPair){
            KeyPair kp = (KeyPair) message;
            pubkey = kp.public_key;
            privkey = kp.private_key;
        }
        else if(message instanceof EncryptedMesg){
            EncryptedMesg m = (EncryptedMesg) message;
            String decrypted = Crypto.encrypt(m.Message,privkey);
            System.out.println("decrypted message:"+ decrypted);
        }
    }
}
class KeyPair implements Serializable {
    public final int public_key, private_key;

    public KeyPair(int public_key, int private_key) {
        this.public_key = public_key;
        this.private_key = private_key;
    }
}
//Messages
class RegisterMesg implements Serializable{
    public final ActorRef pid;

    RegisterMesg(ActorRef pid) {
        this.pid = pid;
    }
}
class CommMesg implements Serializable{
    public final ActorRef ReceiverPid;
    CommMesg(ActorRef receiverPid) {
        ReceiverPid = receiverPid;
    }
}
class LookUpMesg implements Serializable{
    public final ActorRef lookUpPid;
    public final ActorRef returnPid;
    LookUpMesg(ActorRef lookUpPid, ActorRef returnPid) {
        this.lookUpPid = lookUpPid;
        this.returnPid = returnPid;
    }
}
class InitMesg implements Serializable{
    public final ActorRef registryPid;

    InitMesg(ActorRef registryPid) {
        this.registryPid = registryPid;
    }
}

class EncryptedMesg implements Serializable{
    public final String Message;
    EncryptedMesg(String message) {
        Message = message;
    }
}
class PubKey implements Serializable{
    public final int publicKey;
    public final ActorRef receiverPid;
    PubKey(int publicKey, ActorRef receiverPid) {
        this.publicKey = publicKey;
        this.receiverPid = receiverPid;
    }
}
class Crypto {
    static KeyPair keygen(){
        int public_key = (new Random()).nextInt(25)+1;
        int private_key = 26 - public_key; System.out.println("public key: " +
public_key);

        System.out.println("private key: " + private_key);
        return new KeyPair(public_key, private_key);
    }
    static String encrypt(String cleartext, int key) {
        StringBuffer encrypted = new StringBuffer();

        for (int i=0; i<cleartext.length(); i++) {
            encrypted.append((char) ('A' + ((((int) cleartext.charAt(i)) - 'A' +
key) % 26)));
        }
        return "" + encrypted; }
}
```