Suggested solutions INF2440 Spring 2018 Written Exam v2

Eric Jul revised 2021-05-27

In Question 1, Inspera randomly changes the order for every access and every student!

So the answers below are not in line with the published versions of the exam – note that the English and the bokmål versions have the alternatives in different order!

Question 1.1 the correct alternative is the last one.

Question 1.2 alternative 4 and 6 are correct, the rest are wrong.

Question 1.3 the third alternative is the correct one.

Question 1.4 the third alternative is the correct one.

The CORRECT answers for the ENGLISH version:

Question 1.1 the correct alternative is no. 3

Question 1.2 alternatives 1 and 5 are correct, the rest are wrong.

Question 1.3 the first alternative is the correct one.

Question 1.4 the second alternative is the correct one.

The CORRECT answers for the bokmål version:

Question 1.1 the correct alternative is the first one.

Question 1.2 alternatives 5 and 6 are correct, the rest are wrong.

Question 1.3 the second alternative is the correct one.

Question 1.4 the third alternative is the correct one.

Sorry for the confusion!

Question 2.1

Yes, it will terminate. The program will generate two threads that will wait for each other at each call of sync because the cyclic barrier is initialized to two also. So first they will meet once at the barrier then when released, meet there again.

Question 2.2

No, the two threads created will synchronize at the cyclic barrier, but when released from the barrier, they compete and either one can proceed to the print statement first, so output can both be:

```
A0

A1

B1

B0

 Main TERMINATED
```

Or

```
A1

A0

B1

B0

 Main TERMINATED
```

Or two others similar where B1 and B0 are interchanged.

Note: Some have forgotten “ Main TERMINATED”

Question 2.3

No.

One example of where the program does not terminate:

Say thread 1, is VERY slow, then one execution sequence could be that thread 0 gets to the cyclic barrier first, then thread 2 whereafter they both print A0, A2. In the meantime thread 3 gets to the cyclic barrier the first time and blocks.

Then thread 0 gets to the cyclic barrier the second time and thereby releasing thread 0 and thread 3 from the barrier. Thread 3 prints A3 and then blocks at the barrier the second time.

Then thread 0 prints B0 and is done.

Then thread 2 reaches the barrier the second time – and since thread 3 is waiting there, they are both released.

Then thread 2 prints B2 and is done.

Then thread 3 prints B3 and is done.

Because thread 1 is very slow, only now does it reach the barrier the first time. As there is no other thread at the barrier, thread 1 blocks at the barrier. Because the three other threads are done, thread 1 will never proceed past the barrier.

Thus the program does not terminate.

Note: many conclude that the program terminates because, if the threads are executing at about the same speed, then, in most cases, the program WILL terminate. However, they overlook that we CANNOT assume that the threads execute at the same speed. Here is a version of the program where one of the threads is substantially slower than the other – by inserting a 2 second delay into one of the threads – and so it will, with a very high probability NOT terminate:

```
import java.util.concurrent.*;
class Problem {
  // felles data og metoder A
  static int num = 3;
  CyclicBarrier b = new CyclicBarrier(num);

  public static void main(String [] args) {
    Problem p = new Problem();
    p.utfoer(num+1); // num+1 == 4
    System.out.println(" Main TERMINATED");
  } // end main

  void utfoer (int antT) {
    Thread [] t = new Thread [antT];
    for (int i =0; i< antT; i++)
      ( t[i] = new Thread(new Arbeider(i))).start();
   try {
     for (int i =0; i< antT; i++) t[i].join();
   } catch(Exception e) {}
  } // end utfoer

  class Arbeider implements Runnable {
    // lokale data og metoder B
    int ind;

    void sync() {
      try{
        b.await();
      } catch (Exception e) { return;}
    }

    public Arbeider (int in) {ind = in;};

    public void run() {
      // kalles naar traaden er startet
```

```
        if (ind == 1) try {
          TimeUnit.SECONDS.sleep(2);
        } catch (Exception e) { return;};
        sync();
        System.out.println("A"+ind);
        sync();
        System.out.println("B"+ind);
      } // end run
   } // end indre klasse Arbeider
} // end class Problem
```

Question 2.4

The output is not always the same.

Here are two examples:

```
A3
A2
A0
B0
B3
B2
```

Where after the program blocks forever—thread one is waiting at the cyclic barrier while all the other threads have finished and so thread one will wait forever.

```
A0
A1
A2
A3
B0
B1
B2
B3
 Main TERMINATED
```

Where after the program terminates.

Note: many forget to include the printout from the main thread: " Main TERMINATED"

Question 3.1

Attached program contains a possible sequential solution.

Note: Some lost points because they did not pay attention to getting the first deserts correct and/or the final desert.

Question 3.2

Attached program contains a possible parallel solution. It works by splitting the primes into parts, one for each thread, with approximately the same number of primes in each part. Then each thread find the prime deserts presents in its part – with due care to find deserts that span across parts – this is done by adjusting the parts to start and end at a prime – and by including the last prime of one part as the first prime of the next part. Each thread finds a list of prime deserts of monotonically increasing length within its part. When all threads are done, the main thread then combines the lists of deserts starting with the first part and including only those deserts that are strictly larger than the previous desert.

Censor notes:

- Some do not start the threads.
- Some do not initialize their lists.
- A few have problems synchronizing the threads at the end.
- Some have problems dividing up the primes so that all primes are included.
- Some have problems finding two deserts where the end point of the first is the same as the start point of the next AND this point is on the boundary between two parts allocated to two different threads. For example [2, 3] and [3, 5]
- Some use ArrayList that is operated on by multiple threads; it is NOT thread safe (in contrast to arrays where multiple threads can operate on disjunct parts of the array).
- Some have problems combining the results from each array at the end.
- Some have confusing representations of deserts; this is not a problem – except for those than managed to confuse themselves and thus make programming mistakes because of the confusing representation.

Question 4.1

Bubblesort can be parallelized in more than one way.

Method using parallel passes.

One way is to use K threads where K is 1-2 x the number of cores.

Each thread starts a pass of the Bubblesort algorithm, i.e., thread T0 bubbles from 0 to N, thread T1 from 0 to N-1, and so on. Once a thread has reached to index J in the array, the following thread can bubblesort up to J-1. However, the next thread must NOT touch anything from J and up. Because the threads execute asynchronously, T1 could be faster than T0 and get to – and even pass – element J.

One simple solution is to have a lock for every element (except the last). Then a thread Tm that wants to compare and, possibly, exchange elements K and K+1 will lock element K to prevent T(m+1) from accessing element K or any element thereafter. After the compare and, possibly, exchange Tm unlocks element K, continues on to K+1, which is then locked and so forth.

However, this solution is cost-prohibitive in that the number of synchronizations is the same as the number of comparisons. This can be reduced by locking R elements at a time and having at most one thread working on the R elements in such a segment. One suitable choice for R could be to set R to N/K/10, so that the number of synchronizations is limited to K*10 per pass and at least 9 out of 10 segments are free, so that there will be many segments available and thus less chance of a core being free and no thread for it. With K threads there can then be K threads simultaneously bubbling thru the array each in a segment. If one thread is a little faster than the others, it will be delayed at the next lock. If K is, say, 2x the number of cores, it is likely that there will be a thread that is able to execute – the faster threads will be delayed and the slower given a chance to barge ahead.

Each thread enters a new pass by locking an entry region.

In the entry region, it grabs the next pass number, grabs the lock of the first segment, and thereafter releasing the lock for the entry region. This ensures that threads are started in the order of the pass numbers. We identify the pass numbers by the number of elements that must be treated in the pass, i.e., the first pass is N because it need to bubble thru all elements from 0 to N-1 for a total of N elements.

For locking the entry region and each segment, we use a semaphore initialized to one.

Each thread then bubbles thru the first segment. At the end of the segment, it grabs the lock of the next segment, performs the check, and possible exchange, on the two elements that span the two segments, then releasing the lock for the first segment. Thereafter it repeats this for the next segment until the last segment. Note that the number of segment decreases by one for every R passes. (The segment size could be reduced toward the end as to keep all cores active but then the synchronization overhead may become excessive.)

An alternative version is to split the array into K segments, one for each thread. Each thread, k, await that the thread, k-1, for the previous segment has done one pass, at which time k-1 performs the final check, and possible exchange with the last element of its segment, with the first element of the k'th segment. Thereafter it releases the k'th thread to perform one pass of the k'th segment. The k-1'th then goes back to the start of its segment and awaits that the k-2'th segment has completed another pass. Thread 0 can start a new pass at any time and does not wait for the previous (non-existent) pass. An extra lock is necessary between two segment, k-1, and k, as to ensure that the thread k-1 does not complete a pass before thread k has started the previous pass. . (The segment size could be reduced toward the end as to keep all cores active but then the synchronization overhead may become excessive.)

Both of these algorithms are faithful to the bubblesort algorithm – and both can achieve speedup for the parallel version.

Censor notes: many have merely divided the array into segment and then bubblesorted each segment using a thread.

- Some have then bubblesorted the entire array sequentially. This is a poor solution because the major part of the work is in the final bubblesort and thus little speedup is achieved. Actually, a speeddown is achieved in most cases.
- Some have then mergesorted the segments, which gives reasonable speedup but that is because mergesorting is much faster than bubblesort. This solution is not faithful to bubblesort.
- Some have then concatenated the segments two by two and then bubblesorted the doublesized segments using half as many threads. And thereafter concatenated and bubblesorted again until there is only one segment that is bubblesorted sequentially. This solution is faithful to bubblesort and so is worth more points than a non-faithful solution. But it is still not achieving any speedup because the last bubblesorts still has to do up to N/2 passes often taking ¾ of the time of the sequential version. So a speeddown results easily.

Censor notes: Across all solutions, there are problems with getting the boundaries right and also the actions required at the boundaries between sections. And in the merging, some cannot merge properly.

Question 4.2

The attached program contains a parallel version of Bubblesort using the parallelization described in 4.1.

Question 5.1 the correct alternatives are: 2,3,5,7,8. The rest are wrong.

Attached programs:

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

///--------------------------------------------------------
//      Fil: PrimeDesertPara.java
//      Implements Sequential and Parallel Prime Desert finding
//      written by: Eric Jul, University of Oslo, 2018
//
//      Cloned from:
```

```
//     Fil: EratosthenesSil.java
//     implements bit-array (Boolean) for prime numbers
//     written by:  Arne Maus , Univ of Oslo, 2013, 2015
//
//
//----------------------------------------------------------

/**
* Implements the bitArray of length 'bitLen' [0..bitLen ]
*   1 - true (is prime number)
*   0 - false
*  can be used up to 2 G Bits (integer range)
*/
public class PrimeDesertPara
{
   byte [] bitArr ;
   int bitLen;
   final  int [] bitMask ={1,2,4,8,16,32,64,128};
   final  int [] bitMask2 ={255-1,255-2,255-4,255-8,255-16,255-32,255-64,255-
128};
   CyclicBarrier readyToGo, allDone;


   PrimeDesertPara (int max) {
      bitLen = max;
      bitArr = new byte [(bitLen/16)+1];
      setAllPrime();
      generatePrimesByEratosthenes();
   }

   public static void main(String args[]) {
      PrimeDesertPara sil;

      if ( args.length != 2)  {
         System.out.println("use: >java PrimeDesertPara <Max>
<threadCount>");
         System.exit(0);
      }
      int num = Integer.parseInt(args[0]);
      int threadCount = Integer.parseInt(args[1]);
      if (!((num >= 5) && (threadCount >= 2))) {
         System.out.println("Bad parameters: Max must be at least 5 and
threadCount at least 2");
         System.exit(0);
      };

      // Here we generate the sil sequentially as we assumme that you already
have a parallel
      // version of the sil generation from Oblig 3
      long silTime = System.nanoTime();
      sil = new PrimeDesertPara(num);
      silTime = System.nanoTime() - silTime;
      System.out.println("Sil sequential generation time: " +
(silTime/1000000.0) + " ms");

      // now do the Prime Desert calculations
      sil.doit(sil, num, threadCount);
   } // end main()

   void doit(PrimeDesertPara sil, int num, int threadCount) {
      ArrayList<int[]> deserts = new ArrayList<int[]>();
      readyToGo = new CyclicBarrier(threadCount+1); // includes main() thread
      allDone = new CyclicBarrier(threadCount+1);

      // Sequential version of PrimeDesert
      long seqTime = System.nanoTime();  // Start sequential timing
      int[] desert = new int[2];
      desert[0] = 2;
      desert[1] = 3;
      deserts.add(desert);
      int desertLength = (3-2);
      int nextSP, nextEP, lastP;
      lastP = sil.lastPrime();
      nextSP = 3;
      nextEP = sil.nextPrime(nextSP);
      while (nextEP <= lastP) {
         if ((nextEP-nextSP) > desertLength) {
            desert = new int[2];
            desert[0] = nextSP;
            desert[1] = nextEP;
            deserts.add(desert);
            desertLength = nextEP-nextSP;
         }
         nextSP = nextEP;
         nextEP = sil.nextPrime(nextSP);
      };
      seqTime = System.nanoTime()-seqTime;

      System.out.println("Prime Desert Sequential time " +
(seqTime/1000000.0) + " ms\n");

      sil.printDeserts(deserts);



      // Parallel version

      long paraTime = System.nanoTime();

      // Create a desert list for each thread
      ArrayList<ArrayList<int[]>> desertLists = new
ArrayList<ArrayList<int[]>>();

      for (int i = 0; i < threadCount; i++) {
         desertLists.add(new ArrayList<int[]>());
      }

      // start threads
      int largestPrime = sil.lastPrime();
      int chunkSize = largestPrime/threadCount;
      int lastThread = threadCount - 1;
      for (int i = 0; i < threadCount; i++) {
         int from = i*chunkSize;
         int fromPrime = sil.nextPrime(from);
         int upto;
         if (i < lastThread) {upto = sil.nextPrime(from + chunkSize);} else
upto = largestPrime;
         System.out.println("Starting thread "+i+" from "+fromPrime+" upto
"+upto);
         new Thread(new Para(i, threadCount, sil, fromPrime, upto,
desertLists.get(i))).start();
      }

      try {
         readyToGo.await(); // await all threads ready to execute
      } catch (Exception e) {return;}

      // Now the threads are doing their thing

      try {
         allDone.await(); // await all worker threads DONE
      } catch (Exception e) {return;}

      // Combine results
      int currentLength = 0;
      int len;
      ArrayList<int[]> currentList;
      ArrayList<int[]> combinedList = new ArrayList<int[]>();
      for (int i = 0; i < threadCount; i++) {
         //System.out.println("Desert List " + i);
         currentList = desertLists.get(i);
         //sil.printDeserts(currentList);
         for (int j = 0; j < currentList.size(); j++) {
            len = currentList.get(j)[1] - currentList.get(j)[0];
            if (len > currentLength) {
               currentLength = len;
               combinedList.add(currentList.get(j));
            }
         }
      }


      paraTime = System.nanoTime()-paraTime;

      System.out.println("Prime Desert Parallel time " + (paraTime/1000000.0)
+ " ms\nSpeedup "+ seqTime*1.0/paraTime);

      sil.printDeserts(combinedList);

      for (int i = 0; i < threadCount; i++) {
         System.out.println("Desert List " + i);
         sil.printDeserts(desertLists.get(i));
      }

   }

   void printDeserts(ArrayList<int[]> deserts) {
      for (int i = 0; i < deserts.size(); i++) {
         int len = deserts.get(i)[1] - deserts.get(i)[0];
         System.out.println(" "+i+": ["+deserts.get(i)[0]+",
"+deserts.get(i)[1]+"] length: "+len);
      }
      System.out.println("------------------");
   }


   void setAllPrime() {
      for (int i = 0; i < bitArr.length; i++) {
         bitArr[i] = -1 ;     // alt    ( byte)255;
      }
   }

   void setNotPrime(int i) {
      bitArr[i/16] &= bitMask2[(i%16)>>1];
   }

   boolean isPrime (int i) {
      if (i == 2 ) return true;
      if ((i&1) == 0) return false;
      else  return (bitArr[i>>4] & bitMask[(i&15)>>1]) != 0;
   }

   ArrayList<Long> factorize (long num) {
      ArrayList <Long> fakt = new ArrayList <Long>();
      int maks = (int) Math.sqrt(num*1.0) +1;
      int pCand =2;

      while (num > 1 & pCand < maks) {
         while ( num % pCand == 0){
            fakt.add((long) pCand);
            num /= pCand;
         }
         pCand = nextPrime(pCand);
         // maks = (int) Math.sqrt(num*1.0) +1;
      }

      if (pCand>=maks) fakt.add(num);
      return fakt;
   } // end factorize


   int nextPrime(int i) {
      // returns next prime number after number 'i'

      int k;
      if (i < 2) return 2;
      if (i == 2) return 3;
      if ((i&1)==0) k =i+1; // if i is even, start at i+1
        else k = i+2;       // next possible prime
      while (!isPrime(k)) k+=2;
      return k;
   } // end nextPrime

   int lastPrime() {
      int j = ((bitLen>>1)<<1) -1;
      while (! isPrime(j) ) j-=2;
      return j;
   } // end lastPrime

   long largestLongFactorizedSafe () {
      long l;
      int i,j = ((bitLen>>1)<<1) -1;
      while (! isPrime(j) ) j -= 2;
      i = j-2;
      while (! isPrime(i) ) i -= 2;
      return (long)i*(long)j;
   } // end largestLongFactorizedSafe


        void printAllPrimes(){
              for ( int i = 2; i <= bitLen; i++)
               if (isPrime(i)) System.out.println(" "+i);
        } // end printAllPrimes

        int numberOfPrimesLess(int n){
              int num = 2;  // we know 2 and 3 are primes
              int p ;

              for (p=3 ; p < n;p = nextPrime(p) ){
                    num++;
              }
              return num;
        } // end numberOfPrimesLess


         void generatePrimesByEratosthenes() {
                    int m = 3, m2=6,mm =9;     // next prime
                    setNotPrime(1);      // 1 is not a prime

                    while ( mm < bitLen) {
                               m2 = m+m;
                               for ( int k = mm; k < bitLen; k +=m2){
                                     setNotPrime(k);
                               }
                               m = nextPrime(m);
                               mm= m*m;
                          }
              } // end generatePrimesByEratosthenes

   class Para implements Runnable {
      int ind, from, upto, threadCount;
      PrimeDesertPara sil;
      ArrayList<int[]> myDeserts;

      Para(int in, int c, PrimeDesertPara sil, int from, int upto,
ArrayList<int[]> myDeserts) {
         ind = in;
         threadCount = c;
         this.sil = sil;
         this.from = from;
         this.upto = upto;
         this.myDeserts = myDeserts;
      } // konstruktor
      public void run() { // Her er det som kjores i parallell:

      try {
         readyToGo.await(); // await all threads ready to execute
      } catch (Exception e) {return;}

      int[] desert = new int[2];
      desert[0] = from;
      desert[1] = sil.nextPrime(from);
      int desertLength = desert[1] - desert[0];
      myDeserts.add(desert);

      int nextSP, nextEP, lastP;
      lastP = upto;
      nextSP = desert[1];
      nextEP = sil.nextPrime(nextSP);
      while (nextEP <= lastP) {
         if ((nextEP-nextSP) > desertLength) {
            desert = new int[2];
            desert[0] = nextSP;
            desert[1] = nextEP;
            myDeserts.add(desert);
            desertLength = nextEP-nextSP;
         }
         nextSP = nextEP;
         if (ind < threadCount) {
            nextEP = sil.nextPrime(nextSP);
         } else if (nextEP < lastP) {
            nextEP = sil.nextPrime(nextSP);
         } else nextEP = lastP + 1;
      };

      // Done

      try {
         allDone.await(); // await all threads done
      } catch (Exception e) {return;}

      } // end run
   } // end class Para

} // end class PrimeDesert
```

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

///----------------------------------------------------------
//     Fil: BubblesortPara.java
//     Implements Sequential and Parallel BubbleSort
//     written by: Eric Jul, University of Oslo, 2018
//
//----------------------------------------------------------


public class BubblesortPara
{
   CyclicBarrier readyToGo, allDone;
   Semaphore lockEntryRegion;
   Semaphore[] sems;
   AtomicInteger passId;
   int[] arr;
   int segmentLength;


   BubblesortPara (int max) {

   } // end constructor BubblesortPara

   public void fillArr(int[] arr) {
      int seed = 123;
      int n = arr.length;
      Random r = new Random(seed);
      for (int i = 0; i < n; i++){
         arr[i] = r.nextInt(n);
      }
   } // end fillArr

   public void bubblesort(int[] arr) {
      int n = arr.length;
      int temp;
      for (int i=0; i < n; i++){
         for (int j=1; j < (n-i); j++){
            if(arr[j-1] > arr[j]){
               //swap elements
               temp = arr[j-1];
               arr[j-1] = arr[j];
               arr[j] = temp;
            }
         }
      }
   }

   public static void main(String args[]) throws InterruptedException {
      BubblesortPara b;
      final int min =1000;
      if ( args.length != 2)  {
         System.out.println("use: >java BubblesortPara <Max> <threadCount>");
         System.exit(0);
      }
      int num = Integer.parseInt(args[0]);
      int threadCount = Integer.parseInt(args[1]);
      if (!((num >= min) && (threadCount >= 2))) {
         System.out.println("Bad parameters: Max must be at least "+min+" and
threadCount at least 2");
         System.exit(0);
      };

      b = new BubblesortPara(num);

      b.doit(b, num, threadCount);

   } // end main()

   void doit(BubblesortPara b, int num, int threadCount) throws
InterruptedException {
      arr = new int[num];

      // Sequential version of Bubblesort
      b.fillArr(arr);
      long seqTime = System.nanoTime();  // Start sequential timing
      b.bubblesort(arr);
      seqTime = System.nanoTime()-seqTime;

      System.out.println("Bubblesort Sequential time " + (seqTime/1000000.0)
+ " ms\n");

      // Parallel version
      b.fillArr(arr); // reset the array to original content.

      long paraTime = System.nanoTime();
      readyToGo = new CyclicBarrier(threadCount+1); // includes main() thread
      allDone = new CyclicBarrier(threadCount+1);
      lockEntryRegion = new Semaphore(1, true);

      // Divide the array into threadCount*10 segments
      // Create a Semaphore to protect each segment
      int segmentCount = threadCount * 10;
      segmentLength = num/segmentCount;
      //System.out.println("Segment length: "+segmentLength+"  segment count:
"+segmentCount);
      sems = new Semaphore[segmentCount];
      passId = new AtomicInteger(num);

      for (int k = 0; k < segmentCount; k++) sems[k] = new Semaphore(1,
true);

      // fill array.
      b.fillArr(arr);

      // start threads
      int lastThread = threadCount - 1;
      for (int i = 0; i < threadCount; i++) {
         //System.out.println("Starting thread "+i);

         new Thread(new Para(i, b, num)).start();
      }

      try {
         readyToGo.await(); // await all threads ready to execute
      } catch (Exception e) {return;}

      // Now the threads are doing their thing

      try {
         allDone.await(); // await all worker threads DONE
      } catch (Exception e) {return;}

      // Combine results


      paraTime = System.nanoTime()-paraTime;

      System.out.println("Bubblesort Parallel time " + (paraTime/1000000.0) +
" ms\nSpeedup "+ seqTime*1.0/paraTime);



   }

   class Para implements Runnable {
      int ind;
      BubblesortPara b;
      int myPassId;
      int num, currentSeg, myLastSeg, temp;

      Para(int in, BubblesortPara b, int num) {
         ind = in;
         this.b = b;
         this.num = num;
      } // konstruktor

      public void run() { // Her er det som kjores i parallell:

         try {
            readyToGo.await(); // await all threads ready to execute
         } catch (Exception e) {return;}

         // *********************** Thread code for parallel part
************************
         int currentSeg;
         //System.out.println("T"+ind);
         while (passId.get() > 1) {
            // enter entry region and grab next available pass
            try {
               lockEntryRegion.acquire();
            } catch (Exception e) {
               return;
            }
            myPassId = passId.getAndDecrement();
            //System.out.println("T"+ind+" got myPassId: "+myPassId);
            if (myPassId <= 1) {
                // others have done the work, so quit
                lockEntryRegion.release();
                break;
            }
            currentSeg = 0;
            try {
               sems[currentSeg].acquire();
            } catch (Exception e) {
               return;
            }

            //System.out.println("Seg length "+segmentLength);

            lockEntryRegion.release();
            // end of entry region - protected by lockEntryRegion

            // Now repeatedly bubble thru the segments
            myLastSeg = (myPassId-1)/segmentLength;

            for (int s = 0; s <= myLastSeg; s++) {
               int j;
               // for each segment do the bubbling
               int segEnd = (s+1)*segmentLength-1;
               if (segEnd >= myPassId) segEnd = myPassId-1;
               //System.out.println("T"+ind+" starting seg "+s+" segEnd:
"+segEnd+" myLastSeg "+myLastSeg);
               for (j = s*segmentLength+1; j <= segEnd; j++) {
                  if (arr[j-1] > arr[j]) {
                      // swap elements
                      temp = arr[j-1];
                      arr[j-1] = arr[j];
                      arr[j] = temp;
                  }
               }
               if (s < myLastSeg) {
                  // Must handle overlap with next segment at boundary, so
must lock both segments

                  try {
                     sems[s+1].acquire();
                  } catch (Exception e) {
                     return;
                  }

                  if (arr[j-1] > arr[j]) {
                      // swap elements
                      temp = arr[j-1];
                      arr[j-1] = arr[j];
                      arr[j] = temp;
                  }
               }
               // done with this segment so release lock
               //System.out.println("T"+ind+" releasing seg "+s);
               sems[s].release();
            } // for each segment
         }

         // *********************** Thread specific code done
****************************

         try {
            allDone.await(); // await all threads done
         } catch (Exception e) {return;}

      } // end run
   } // end class Para

} // end class BubblesortPara
```