UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Eksamen i:	INF2440— Effektiv parallellprogrammering
Eksamensdag:	2. juni 2015
Tidspunkter:	09.00 – 13.00
Oppgavesettet er på:	3 sider + 2 sider vedlegg
Vedlegg:	I) Skisse av Modell2-koden og II) Den sekvensielle radix-sorteringa fra Oblig4
Tillatte hjelpemidler:	Alle trykte og skrevne notater, utskrifter, bøker ol.

- Kontroller at oppgavesettet er komplett, og les nøye gjennom oppgavene før du løser dem. Poengangivelsen øverst i hver oppgave angir maksimalt antall poeng.
- Du kan legge dine egne forutsetninger til grunn og gjøre rimelige antagelser, så lenge de ikke bryter med oppgavens "ånd". Gjør i så fall rede for disse forutsetningene og antagelsene.
- Til eksamen skal svarene skrives på gjennomslagspapir. Da må du huske å skrive hardt nok til at besvarelsen blir mulig å lese på alle gjennomslagsarkene, og ikke legge andre deler av eksamensoppgaven under når du skriver.
- Til eksamen skal du selv beholde underste arket etter levering av de to øverste til eksamensinspektøren. Nummerer sidene, og husk å skrive kandidatnummeret ditt på besvarelsen.

I vedlegg I) finner du en litt forenklet versjon av Modell2-koden som ble nyttet i kurset (en ytre klasse med main-tråden og en indre klasse hvor objekter av denne blir egne tråder) I den skissen er det markert med store bokstaver ulike områder av denne koden som det blir referert til i oppgavene. I vedlegg II) er den sekvensielle koden for radix-sortering fra oblig4 (som du kanskje trenger i oppg. 7), og i vedlegg III er KvikkSort fra forelesningen Uke9 som du kanskje trenger i oppgave 4.

Oppgave 1 (20 poeng)

a) Hvilke feil kan vi få ved at det ikke synkroniseres mellom to tråder som har felles variable i Java som det skrives på. (svar: Maksimalt 20 linjer)

a) - To tråder kan se ulike verdier av samme felles variabel
- Utsatte operasjoner er ikke utført (umulig å bruke kildekoden til forståelse av hva som skulle ha skjedd)

Kommentar: 5 poeng for hver av de to momentene

b) Definer vranglås i et parallelt, trådbasert program. Gi et eksempel på en mulig vranglås med bruk av tre Semaphorer mellom tre tråder. (svar: Maksimalt ¾ side).

Kode som de tre trådene gjør i run() :

```
try {
        a[ind].acquire();
        a[(ind+1)%3].acquire();
} catch (Exception e) {return;}

a[ind].release();
a[(ind+1)%3].release();
```

```
// Lager tre semaforer kalt 0,1,2, alle initiert til  (eller
   også 2 eller 3) kan få vranglås i alle tilfellene,
    se Oppgave1.java
// Mulig vranglås ved acquire() (ingen greier å si release():
// tid ->
// t0:   0          1               (venter på t1 for å få 1)
// t1:         1          2         (venter på t2 for å få 2)
// t2:                2        0    (venter på t0 for å få 0)
```

Oppgave 2 (15 poeng)

Anta at du i felt A i vedlegg I deklararer følgende variabeler, og at du **har startet 3 tråder**:

```
CyclicBarrier ab = new CyclicBarrier(2),
              bc = new CyclicBarrier(2),
              ca = new CyclicBarrier(2);
```

Tråd **A** eksekverer før eller siden i sin run()-metode:

```
try{ ab.await();
     bc.await();
catch (Exception e) {return;}
```

Tråd **B** eksekverer før eller siden i sin run()-metode:

```
try{ bc.await();
     ca.await();
catch (Exception e) {return;}
```

Tråd **C** eksekverer før eller siden i sin kode run()-metode:

```
try{ ab.await();
     ca.await();
catch (Exception e) {return;}
```

Spørsmal: Går dette bra; bekriv tre ulike situasjoner av rekkefølger mellom disse synkroniseringene og de ulike ventesituasjoner som da oppstår.

Dette går bra fordi de tre trådene bruker barrierene de bruker i samme sorterte rekkefølge: (ab, bc, ca).Tre ventesituasjoner: hvor ab. = ab.await(); .. osv. v.på: = venter på; | = vent

1)

```
TrA: ab      |      bc   (v.på TrC)
```

```
TrB:    bc     |      ca     (v.på TrA)
TrC:       ab   ca |         (v.på TrB)
```

2)

```
TrB: bc     |     ca      (v.på TrC)
TrC:    ab |         ca (v.på TrA)
TrA:       ab   bc      (v.ikke)
```

3)

```
TrC: ab   |     ca |        (v.på TrA, så B)
TrA:    ab        bc      (v.ikke)
TrB:        bc   |   ca     (v.på TrA, )
Kommentar: Så mange forvirrende diagrammer under spørsmålsrunden. Kanskje
litt forvirrende at den er langt fler enn 3 mulige sekvensierieringer : vel
6 som ikke fletter deler av setninger og langt fler når setningene blandes.
```

Oppgave 3 (15 poeng)

Anta at du har to tråder som aksesserer en felles variabel, deklarert i felt A slik: `int i =0,` og at hver av trådene har deklartert i sitt lokale område B i hver tråd:

```
ReentrantLock lock = new ReentrantLock();
```

Tråd0 utfører følgende kode:

```
lock.lock();
try { i = i+2;
} finally { lock.unlock();}
```

før den terminerer, mens

Tråd1 utfører følgende kode:

```
lock.lock();
try { i = i-3;
      i = i+1;
} finally { lock.unlock();}
```

før den terminerer,

Hvilke mulige verdier kan **i** ha etter at begge trådene har terminert? Regn med at Tråd1 leser i fra hovedlageret for hver av sine to setninger.
Tegn diagrammer med tidsakse som viser ***hvordan*** minst tre av de verdiene du hevder kan bli verdien av **i** til sist, kan fremkomme (når leser og skriver Tråd0 og Tråd1 hvilke verdier?).
Dette blir ikke synkronisert i det hele tatt fordi de to låste reginionene bruker hver sin Lock. – lurespørsmål.

Mulige verdier: **-2, -1,0,2, 3:**

```
-2) T0: les:0                        skriv 2
    T1:    les:0 skriv -3  les -3         skriv -2
-1) T0:                 les:-3              skriv-1
    T1: les:0  skriv:-3   les:-3 skriv:-2
 0) T0: les:0  skriv 2
    T1:              les 2 skriv -1  les -1  skriv 0
 2) T0: les:0                                skriv:2
    T1:      les:0 skriv-3    les -3  skriv -2
 3) T0: les:0                skriv 2
    T1:      les:0 skriv-3          les 2  skriv 3
```

Oppgave 4 (40 poeng)

Skriv først en sekvensiell og så en parallell metode for å sortere radene i en positiv (dvs. alle elementene er ≥ 0) todimensjonal array **`int a[][] = new int[n][n];`** slik at den raden med størst sum er nederst (radnummer: n-1), den med nest største sum er nest nederste rad (radnummer: n-2),……., osv.

Når du sorterer radene skal du flytte om på pekerne til radene framfor å flytte selve elementene. Du kan anta at n ≤ 20 000, slik at innstikksortering ikke blir aktuelt, men f.eks kvikksort som du her skriver koden til. Bruk som utgangspunkt den Kvikksort-koden du finner i Vedlegg III og skriv ned hvordan du vil modifisere den for denne oppgaven.

N.B. Du skal **ikke** skrive hele programmet, men bare den sekvensielle metoden og de datastrukturer og den/de metodene (inkludert sorteringa) du trenger; og tilsvarende det parallelle tilfellet skrive de datastrukturer og den/de metodene (inkludert sorteringa) du trenger og som kalles fra run() – metoden (også sorteringsmetoden). For data-deklarasjonene, skriv med kommentar i koden i hvilke områder (A eller B) i vedlegg I) du tenker dem plassert.

Kommentar: Se min løsning i Oppgave4.java. Mange studenter brukte tid på dette. De to viktigste grepene er her: i) at radene summeres i en egen array før det sorteres. ii) at det innføres en ny parameter til kvikksort (arrayen med sum-verdiene) og: iii) at særlig 'swap ' skrives om hvor de angitte elementene i a[] og sumVerdier[] byttes om på samme måte. Se mitt forslag i Oppg5.java. BÅDE summeringen (trivielt: hver tråd får et visst antall rader å summere) og 'omstokkingen av pekere i a[] skal parallelliseres.

Kommentar: Et av problemene her er faktisk også å få tak i radpekerne i a[][]. Det gjøres slik at det er a[][] som er parameter over alt hvor vi ikke har pekt ut en spesiell rad (da kan vi bruke a[i] som parameter) , men når vi bruker den (pekerne) sier vi a[i],..

```
A:      int [][] a;
        long [] radSum;
        int n, antT;

         long sumRad (int [] a ) {
                 long sum =0;
                 for (int j = 0 ; j < a.length; j++)
                        sum += a[j];
                 return sum;
         } // end sumRad


 // Parallell Kvikksort uten insertSort
```

```
void quicksortPara(int[][] a, long [] rs, int left, int right, int level)
          int piv = partition (a,rs, left, right);   // del i to
          int piv2 = piv-1;
          long pivotVal = rs[piv];
          while (piv2 > left  && rs[piv2] == pivotVal) {
               piv2--;       // skip like elementer i midten
       }
          if (level-- > 0){
                 Thread t1= new Thread (new Arbeider2(a,rs,left, piv2, level)),
                        t2 =new Thread (new Arbeider2(a,rs,piv+1, right, level));
                        t1.start();
                        t2.start();
                 try{
                        t1.join();
                        t2.join();
                 }catch (Exception e) {return;}
          } else {
               if ( piv2-left > 0) quicksortSek(a,rs, left,  piv2);
               if ( right-piv > 0) quicksortSek(a, rs, piv + 1, right);
       }
} // end quicksortSek


// del opp a[] i to: smaa og storre
int partition (int [][]a,long[] rs, int left, int right) {
 long pivVal = rs[(left + right) / 2];
 int index = left;
       // plasser pivot-element helt til høyre
       swap(a,rs, (left + right) / 2, right);
        for (int i = left; i < right; i++) {
                if (rs[i] <= pivVal) {
                     swap(a, rs, i, index);
                     index++;
                }
  }
 swap(a, rs,index, right); // sett pivot tilbake
  return index;
} // end partition

 void swap(int [][] a, long [] rs, int l, int r) {
          int []temp1 = a[l];
          long temp2 = rs[l];
          a[l] = a[r];
          rs[l] = rs[r];
          a[r] = temp1;
          rs[r] = temp2;
 } // end swap
```

```
Fra main-tråden:
 Thread [ ] tr = new Thread[antT];
       for (int i =0; i< antT;i++){
           (tr[i] = new Thread(new Arbeider(i))).start();;
       } // end initarray
       try{
       for (int i =0; i< antT;i++)
             tr[i].join();
        }catch(Exception e) {return;}

  // sorter i parallell
  // finn level = parallellisering i toppen av treet
      int level = 0;
      while ( (antT >> level) > 0 ) level++;

      quicksortPara( a, radSum, 0,n-1, level) ;


………………….
      class Arbeider2 implements Runnable {
 B:                  int[][] a;
                     long [] rs;
                     int left, right, level;
            public void run( ) {
                   quicksortPara(a, rs, left, right, level);
            } // end run
      } // end indre klasse Arbeider2

      class Arbeider implements Runnable {
B:                   int ind; // lokale data og metoder B
                     int part, left,right;
            public void run( ) {
                  // kalles når tråden er startet
                  for (int i = left; i < right; i++)
                     radSum[i] = sumRad(a[i]);
            } // end run
       } // end indre klasse Arbeider
```

Oppgave 5 (30 poeng)

Anta at du har deklarert en array: `double tallene[] = new double [n];` og du kan anta at elementene i `tallene[]` er sortert stigende.

Du skal nå skrive først en sekvensiell og så en parallell metode (anta at du har k kjerner) som snur innholdet av `tallene[]` slik at tallene blir sortert synkende. Skriv heller ikke her hele programmet, men bare de metodene og evt. data som du trenger i A og B området i vedlegget + hva innholdet av run()-metoden i trådene er.

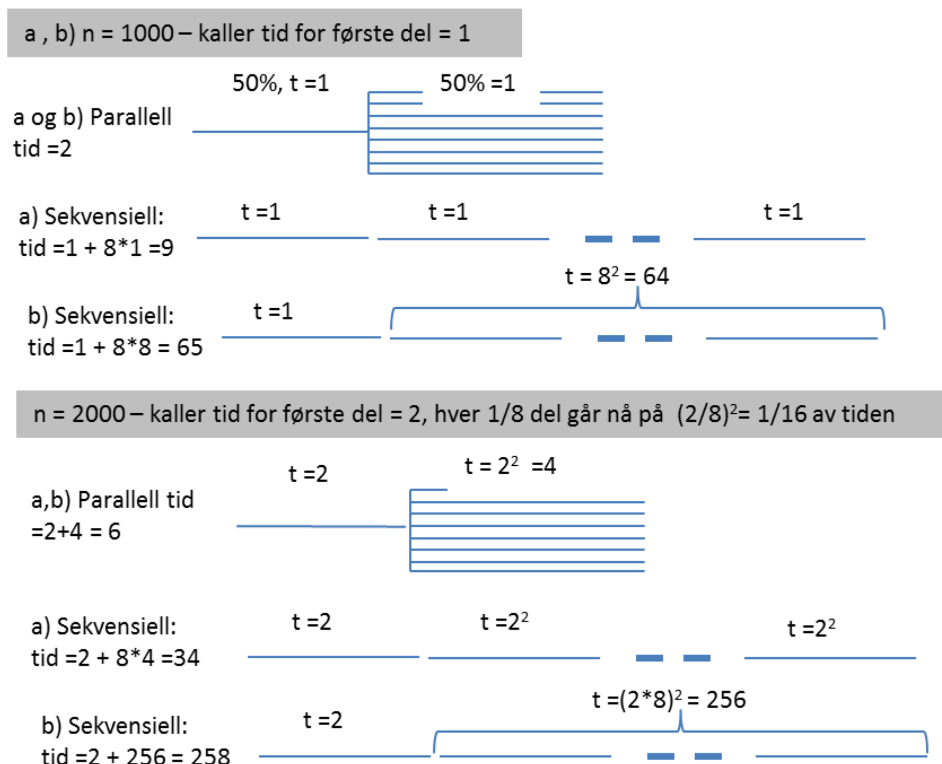
Kommentar: Ganske lett: vi skal bare bytte om første og siste, nest første og nest siste. Merk at parameteren for ombytting bare må gå til n/2. går den helt til n bytter vi om innholdet 2 ganger og kommer tilbake til utgangspunktet. Parallelisering går lett – se oppgave5.java. Hver tråd får ansvar for sin del av den første halvdel av tallene[] etter kjent mønster (bruker gjerne metode swap():

```
Arbeider ( int in) {
      ind = in;
      part = n/(antT*2);
      left = ind*part;
      if (ind == antT-1) right = n/2;
      else right = left + part;
}
```

Oppgave 6 (30 poeng)

Du skal her se på konsekvenser av Gustafsons lov. Anta at du har et program som med n =1000 først bruker 50% av kjøretiden i en sekvensiell kode som har kompleksitet O(n), og så 50 % av kjøretiden i en perfekt parallellisert kode med k =8 kjerner hvor koden har kompleksitet $O(n^2)$. Regn ut den speedup du får med n= 1000 og n =2000 hvor du i begge tilfeller bare har parallellisert den siste delen av koden. Utled svaret, ikke bare kom med et tallsvar!

Kommentar: Vanskelig. Her er det to mulige tolkninger som begge skal gis full poengsum hvis riktig:
Tolkning a) : Når vi deler opp data i 8 like store deler, går det 8 ganger fortere (alle algoritmer de har sett så lengt er slik) og når n=2000 går den første delen på 2x tid og den siste delen på 4x.
Tolkning b) Siden dette er data delt i 8 deler og $O(n^2)$, går hver 8-de del 64 x fortere – og da hele 64 ganger langsommere.



Vi ser da at vi får flg. Speedup S:
a) n = 1000 : S = 9/2 = **4,5** , n= 2000: S = 34/6= **5, 67**
b) n = 1000: S = 65/2 = **32,5,** n = 2000: S = 258/6 = **43**

(Nå kan man jo lure på om det finnes algoritmer som følger b), og det finnes – bla. Shell-sort som parallellisering av innstikksortering)

Oppgave 7 (60 poeng)

Det er to typer av 'den midterste verdien' av n elementer: $a_0, a_1, .. a_{n-1}$:

- Det geometriske gjennomsnittet G = $(\sum_{k=0}^{n-1} a_k)/n$.
- Medianverdien M, som er slik at halvparten av a_i-ene er $\leq$ M og halvparten er $\geq$ M.

(for enkelhets skyld antar vi her at n er et oddetall og et stort tall.)

Oppgave:

a) (5 poeng) Lag en skisse av hvordan du lett kan parallellisere det å finne det geometriske gjennomsnittet G.

b) (20 poeng) Forklar hvorfor det er klart mulig, men ikke like raskt å forsøke å finne medianverdien M med parallell kode. (Hint: Hva må vi først gjøre før vi kan finne M?)

c) (Vanskelig: 35 poeng) Her skal du kan ta utgangspunkt i løsningen din på Oblig4. Da kan du bla. droppe nesten all flyttingen av elementene for siffer 2 fra b til a, men likevel finne medianverdien M både i den sekvensielle og parallelle koden.

Kommentar: Kanskje ikke så vanskelig som jeg trodde (unntatt c)). Burde ikke kalt G bare for det geometriske gjennomsnittet (det heter det artimetriske gjennomsnitt – min feil), men siden det var definert med formel ble det vel ikke noe problem.

Poenget med denne oppgaven er å se hvor lite/mye man må gjøre av en full radix-sortering for å finne medianen.

a) Å parallellisere det å lage et gjennomsnitt er bare å la hver tråd summere sine tall fra left til right (som oppgave 5) av sin delmengde av tallene.

b) For å finne medianen, må man enten delvis eller helt sortere tallene først og så finne medianen på plass n/2.

c) For å se hva vi kan gjøre, må vi se på siffer 2 (det siste i radix-sorteringa i vedlegget). Metoden radix bare regner ut parametre til de to kallene til radixSort og vi ser at det er i sorteringen på det siste sifferet (siffer 2 som det står i oppgaven) at vi skal gripe inn og forkorte koden (siffer 2 kan gjenkjennes ved at parameteren shift > 0). I metoden er det tre løkker merket b) c) og d), og vår jobb er da å finne element n/2 med minimalt arbeid.

Vi må for siffer 2 også telle alle de ulike sifferverdiene - dvs. løkke b) er uendret.

I løkke c) er det nok at vi stopper når acumVal > n/2; da kan vi avslutte c) (ny siste setning i løkke c): if (acumVal > n/2) { break;} eller slik det er i Oppgave7.java.

Løkke d) kan også forkortes slik:

```
for (int i = 0; i < n; i++) {
        int index =  count[(a[i]>>shift) & mask]++;
        if (index == n/2  )  { median = a[i]; break;} // median er en global variabel
}
```

Eller kortere:

```
  for ( i = 0; count[(a[i]>>shift) & mask]++ < nHalve; i++) ;

            median = a[i];
```

Vi skriver ikke til den arrayen som egentlig er a[] , den blir ikke endret og vi har funnet medianen i det som koden her sier er a[] (= egentlig den arrayen b[] vi har laget).

Rundt de originale setningene c) og d) ligger f.eks :

```
 if (shift == 0 ) { // siffer 1
….Original c) og d) her
} else {
   // c) Add up in 'count' - accumulated values
   for ( i = 0; acumVal < nHalve ; i++) {
         j = count[i];
         count[i] = acumVal;
         acumVal += j;
    }
   // d) move numbers in sorted order a to b
        for ( i = 0; count[(a[i]>>shift) & mask]++ < nHalve; i++) ;

            median = a[i];
  } // end else
```

2) Parallellisering er omlag som Oblig4. Fra koden I Oblig7.java:

```
     for (int val = 0; sumC <= nHalve ; val++) {
          for (int i = 0; i < ind; i++) {
                    sumC += allCount[i][val];
           }
           localCount [val] = sumC;
           for (int i = ind; i < numThreads; i++) {
                    sumC += allCount[i][val];
           }
   }
// no need to sync between c) and d) - only local variables
// d) move numbers in sorted order a to b
 for (ii = left; ii < right; ii++) {
          if (localCount[(a[ii]>>shift) & mask]++ == nHalve) median = a[ii];
}
```

Appendix I – Modell2 kodeskisse:

```
import java.util.concurrent.*;
class Problem {
    //  felles data og metoder A
     public static void main(String [] args) {
          Problem p = new Problem();
          p.utfoer(8);
     }
     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) {}
     }

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

            Arbeider (int in) {ind = in;}
           public void run( ) {
                  // kalles når tråden er startet
           } // end run
     } // end indre klasse Arbeider

} // end class Problem
```

Appendix II – Sekvensiell Radix-sortering fra Oblig 4:

```
  class SekvensiellRadix{

          static void radix (int [] a) {
                    // 2 digit radixSort: a[]
                    int max = a[0], numBit = 2, n =a.length;
                   // a) finn max verdi i a[]
                    for (int i = 1 ; i < n ; i++)
                              if (a[i] > max) max = a[i];

                    while (max >= (1L<<numBit) )numBit++; // antall siffer i max

                    // bestem antall bit i siffer1 og siffer2
                    int bit1 = numBit/2,
                                   bit2 = numBit-bit1;
                    int[] b = new int [n];
                    radixSort( a,b, bit1, 0);    // første siffer fra a[] til b[]
```

```
            radixSort( b,a, bit2, bit1);// andre siffer, tilbake fra b[] til a[]
        } // end radix


    /** Sort a[] on one digit ; number of bits = maskLen, shiftet up 'shift' bits */
    static void radixSort ( int [] a, int [] b, int maskLen, int shift){
            int  acumVal = 0, j, n = a.length;
            int mask = (1<<maskLen) -1;
            int [] count = new int [mask+1];

           // b) count=the frequency of each radix value in a
            for (int i = 0; i < n; i++) {
                  count[(a[i]>> shift) & mask]++;
            }

           // c) Add up in 'count' - accumulated values
            for (int i = 0; i <= mask; i++) {
                   j = count[i];
                        count[i] = acumVal;
                        acumVal += j;
             }
           // d) move numbers in sorted order a to b
            for (int i = 0; i < n; i++) {
                  b[count[(a[i]>>shift) & mask]++] = a[i];
            }
    }// end radixSort

}// end class SekvensiellRadix
```

Appendix III – Kvikksortering (uten innstikkSort) fra forelesningen Uke 9:

```
// sekvensiell Kvikksort uten insertSort
void quicksortSek(int[] a, int left, int right) {
        int piv = partition (a, left, right);   // del i to
        int piv2 = piv-1, pivotVal = a[piv];
        while (piv2 > left  && a[piv2] == pivotVal) {
            piv2--;      // skip like elementer i midten
        }
        if ( piv2-left > 0) quicksortSek(a, left,  piv2);
        if ( right-piv > 0) quicksortSek(a, piv + 1, right);
} // end quicksortSek

// del opp a[] i to: smaa og storre
int partition (int [] a, int left, int right) {
    int pivVal = a[(left + right) / 2];
    int index = left;
   // plasser pivot-element helt til høyre
   swap(a, (left + right) / 2, right);
```

```
    for (int i = left; i < right; i++) {
        if (a[i] <= pivVal) {
         swap(a, i, index);
         index++;
        }
    }
    swap(a, index, right); // sett pivot tilbake
    return index;
 } // end partition

void swap(int [] a, int left, int right) {
     int temp = a[left];
     a[left] = a[right];
     a[right] = temp;
} // end swap
```