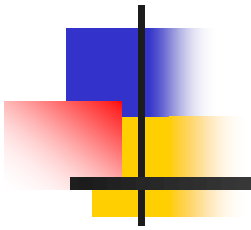
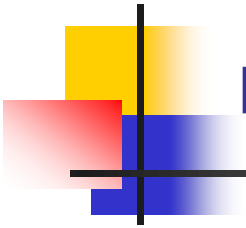


INF2440 – Prøveeksamen, løsningsforslag, 26 mai 2014

Arne Maus
OMS,
Inst. for informatikk



Prøveeksamen

Er en modell av hva du får til eksamen:

- like mange (+-1) oppgaver som eksamen
- Samme type oppgave på samme sted.
- Prøveeksamen er grovt sett de oppgavene av to oppgavesett som ikke var gode nok til å komme med til eksamen.

- Flere trykkfeil i prøveeksamen enn i eksamens-settet:
  - for eksempel: gjennomgang kl 15.15
  - **Oppgave 6 må rettes:** f.eks slik setning s1 er: int num=1;
  - **Oppgave 4** må vi anta at trådene ikke utfører koden sin samtidig (ikke data-kappløp om `int teller2`)
  - **Oppgave 5**: burde (?) spesifisert: stigende sortert

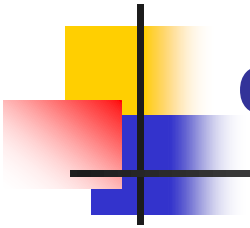
Appendix: Model2 –kode skisse

```
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();

           for (int i =0; i< antT; i++) t[i].join();
     }

     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
```

Oppgave 1 (10 poeng)

- Når vi synkroniserer i Java, hva annet skjer i tillegg til at vi evt. får ordnet trådene tidsmessig (svar kortfattet, maksimalt 10 linjer)
  - Vi får skrevet ned alle felles variable i hovedhukommelsen og vi får utført alle utsatte operasjoner. Det vi si at alle trådene som har synkronisert på **samme** synkroniseringsvariabel, vil etter en synkronisering kunne lese samme verdi på alle felles variable.

- Hvilken effekt ville det ha på programutførelsen av et parallelt program i Java hvis alle metodene var synchronized og deklarert i A-området (se vedlegget)
  - Vi vil grovt sett fått et sekvensielt program fordi første tråden som starter å eksekvere i sin run()-metode, vil kalle en synchronized metode som kaller en annen synchronized metode, som .. inntil tråden er ferdig med sin kode og neste tråd får slippe til som oppfører seg på samme måte. Dvs. vi får utført kodene til hver tråd etter hverandre og **ikke** samtidig – et sekvensielt program.



Oppgave 2 (10 poeng)

- Du skal beskrive forskjellene av Gustafsons lov kontra Amdahls lov. Anta at du har et program som for en gitt n har 10% sekvensiell kode. Hva sier de to lovene om hva som vil skje med hvor stor maksimal speedup vi kan få på dette programmet hvis vi kjører programmet med en langt større n på problemet. Hvilken av de to lovene er mest optimistiske med hensyn til å få størst mulig speedup.
  - Amdahls lov sier at med 10% sekvensiell kode vil vi maksimalt få speedup på 10 fordi den sekvensielle delen er en konstant del, uavhengig av størrelsen på problemet. Hos Gustafson derimot er den sekvensielle delen (mer eller mindre) konstant, slik for et dobbelt så stort problem vil det bare være 5%, osv, slik at Gustafson er langt mer optimistisk om speedup.

Oppgave 3(20 poeng)

Goldbach hadde egentlig to påstander i 1743. Den første om partall har vi jobbet mye med. Hans andre påstand var at ethvert oddetall kan skrives sum summen av tre primtall. Dette er visstnok bevist i 2013 i et 133 siders bevis (sist revidert 14. april 2014!).

- a) Skisser hvordan du vil lage et sekvensielt program som sjekker dette for alle oddetall n < M. Du kan godt bruke din løsning på Goldbachs påstand om partall som en del av løsningen.

- Svar: La M være et oddetall. Finn et primtall p3 < M.
  Da er (M-p3) et partall, og
  M = Goldbach3(M) = p3 + Goldbach2(M-p3) = p3 + p1 + p2

  = M-p3

Oppgave 3(20 poeng)

b) Skisser så hvordan du vil parallellisere det å vise Goldbachs påstand om alle oddetall n < M.

- Svar: Vi deler M opp i k like store deler slik:
  Anta k tråder; og at M er delelig med k.
  $Tråd_i$ vil da prøve Goldbach3(m) for $(M*i)/k \leq m < (M*(i+1))/k$.

c) **Kan puffes:** Hvis Goldbachs påstand om partall var bevist (noe det ikke er), men anta likevel det. Bevis da at Goldbachs påstand om oddetall er sant (meget lett, tips: 3 er et primtall).

Svar : Gitt vilkårlig oddetall Od, Od≥7. Da er Od-3 et partall

(Od-3)=p1+p2, og da er M=p1+p2+3

Oppgave 4 (15 poeng)

I denne oppgaven tenker vi oss ikke data-kappløp om teller2.
Anta at at du **har startet 5 tråder og at** du i felt A i vedlegget deklarerer følgende to variable:

```
int teller2 =0
Semaphore abc = new Semaphore(2);
```

Trådene eksekverer kode i sin run()-metode som før eller siden alle prøver å utføre:

```
try{abc.aquire();
catch (Exception e) {return;}
 teller2 ++;
```

a) Hva er verdien av **`teller2`** etter at første tråden prøver å utføre denne koden.

Svar: teller2 = 1; og den første tråden fortsetter

b) Hva er verdien av **`teller2`** etter at andre tråden prøver å utføre denne koden.

Svar: teller2 = 2; og den første og andre tråden fortsetter

c) Hva er verdien av `teller2` etter at tredje tråden prøver å utføre denne koden.

Svar: teller2 = 2; og den første og andre tråden fortsetter og den tredje tråden venter på abc-køen

Oppgave 4 (15 poeng) fors

d) Hva er verdien av **`teller2`** etter at fjerde tråden prøver å utføre denne koden.

Svar: teller2 = 2; og den første og andre tråden fortsetter og den tredje og fjerde tråden venter på abc-køen

Oppgave 5 (40 poeng)

Skriv først en sekvensiell og så en parallell metode for å løse følgende problem: I en matrise: `int[][]a = new int[n][n]` er det slik at to av radene er stigende sortert, mens de resterende n-2 radene ikke er sortert (dvs. i hver av disse n-2 radene er det minst en indeks `i` slik at i rad k er `a[k][i] > a[k][i+1])`. Oppgaven din nå er å finne disse to radene. La til slutt main-tråden skrive ut svaret (indeksen på de to radene som er sortert)

N.B. Du skal **ikke** skrive hele programmet, men bare den sekvensielle metoden, de datastrukturer du trenger og den/de metodene du trenger i det parallelle tilfellet som kalles fra run() - metoden. For begge deler, skriv med kommentar i koden hvilke områder (A eller B) i vedlegget du tenker disse plassert.

Oppgave 5 -sekvensiell

```
void finnToRaderSekvensiell (int [][]a){
            int [] aa;
             int j;

              for (int i = 0; i< a.length; i++) {
                        // for hver rad i
                        aa = a[i];
                        for ( j = 1; j < aa.length; j++){
                              if (aa[j-1] > aa[j]) break;
                        }
                         if(j == aa.length)  System.out.println(«Rad: « + i + «er sortert»)
                } // end rad i
} // end finnToRaderSekvensiell
```

Oppgave 5 - parallell

```
// i A - området
ArrayList svar = new  ArrayList();
synchronized void reistrerPara(int i){ svar.add(i); }

// i B – området:
void finnToRaderPara (int [][]a){
            int [] aa;
             int j;
             int left = (ind)*ant;
             int right= (ind+1)*ant;
            if (ind == antTraader-1) right = n; ;

            for (int i = left; i< right; i++) {
                        aa = a[i];
                        for ( j = 1; j < aa.length; j++){
                              if (aa[j-1] > aa[j]) break;
                        }
                      if (j == aa.length)  reistrerPara(i);
                } // end rad i
} // end finnToRaderSekvensiell
```

Oppgave 6 (20 poeng)

Du har et trådbasert system og vil øke en **int** variabel **num** hver gang med 1 slik at den til slutt blir så stor at den av maskinen blir betraktet som et negativt tall ved at verdien kommer opp i fortegnsbitet til variabelen. For det formålet starter du opp 10 tråder. Du skal da benytte følgende to programsetninger s1 og s2:

```
// setninger s1
int num =1; // feil rettet her

// setninger s2
synchronized int leggTil(int num) {
    num++ ;
    return num;
}
```

Vi tenker oss at det i hver run-metode i trådene utføres følgende kode:

```
   while (num >  0 ){
     num = leggTil(num);
   }
```

Selve problemløsningen din tar utgangspunkt følgende sekvensielle kode:

```
    while (num >= 0) num++;
  System.out.println (" num < 0: ” + num+ ",  i mål”);
```

Oppgave 6 – forts.

Spørsmål: Hva er effekten av å plassere de to setningene (s1 og s2) ulike steder i Model2koden (vedlegget) – N.B det er koden i run()-metoden som oppdaterer **`num`** – IKKE **`leggTi`**l-metoden i s2. Den framskaffer 'bare' et tall som er 1 større enn den gamle verdien av **`num`** (slik run()-metoden ser **`num`**).

a) **s1 og s2 i A ?**

Svar: Virker, men svært langsomt – lager egentlig sekvensiell kode av problemet.

b) **s1 i A og s2 i B?**

Svar: Virker IKKE hvis vi vil ha med alle oppdateringene fordi den synkroniserte metoden i hver tråd synkroniserer bare på hvert sitt tråd-objekt, og vi vil få mange mistete oppdateringer, Derimot er dette mye raskere enn a) fordi vi får full parallellitet på problemet . Tvilsom, men raskere.

c) s1 i B og s2 i A?

Svar: Her er num en lokal variabel i hver tråd , men derimot så synkroniseres alle trådenes oppdateringer av sine lokale 'num'-variable med de andre trådene – uten at det er nødvendig. Dvs. dette går langsomt uten at det var nødvendig i det hele tatt med synkroniseringen. Vi får her k svar – ikke ett.

d) s1 og s2 i B?

Svar: Full fart fordi all synkroniseringene er lokal i hver tråd (men denne synkroniseringen er da heller ikke nødvendig). Vi får også her k svar – ikke ett.

Gi en kort begrunnelse på hvert av punktene a)-d).

Oppgave 6 - testing

- Det kom opp en rekke intressante spørsmål etter gjennomgang av oppgave 6. Følgende vet vi etter å ha testet a) til d) + en sekvensiell versjon, at kjøretidene varierer voldsomt

Sekvensiell kode : 1,3 sek

a) s1 og s2 i A : 1675,036 sek

b) s1 i A og s2 i B : 206,045 sek

c) s1 i B og s2 i A : 202,553 sek

d) s1 og s2 i B : 59,232 sek

Grunnen til at a) tar spesielt lang til er at dette lager svært mange synkroniseringer og sekvenserer all aksess til num.

Grunnen til sekvensiell kode er så mye raskere enn selv d) er at synchronized metoder er om mulig det langsomste synkroniseringsmåten vi har i Java, og her gjør vi over 2 milliarder kall.

Oppgave 7 (80 poeng)

Flettesortering, sekvensiell og parallell.

(For dere som har glemt hva flette-sortering er: Du har i en array a[] to sorterte sekvenser s1 og s2 (av lengde l1 og l2) som ligger etter hverandre i a[] – dvs. at hvis s1's siste element er i a[k], så starter s2 i a[k+1]. Kopier disse to over i en array b[] slik at du stadig tar det minste elementet av s1 eller s2 i a[] som enda ikke er kopiert som neste element du kopierer over til b[]. Du plukker altså elementene fra toppen hver av de to sekvensene til alle elementene er kopiert over til b[]. I b[] er det etter kopieringa da en sortert sekvens av lengde = l1+l2.)

a) Du skal nå først skrive en rekursiv versjon av flettesortering, som består av to metoder (omtrent som quicksort) - her er den første oppgitt:

```
void fletteSort(int [] a) {
          if (a.length <= 50) insertSort(a,0,a.length-1);
          else {
                int [] b = new int [a.length];
                flette(a,b,0,a.length-1);
          }
   } // end fletteSort
```

Du skal skrive den rekursive metoden flette:

```
void flette (int [] a, int [] b, int v, int h){
      < ..din kode her.. >
   }
```

b) Skriv en effektiv parallell versjon av din sekvensielle, rekursive flettesorteringsalgoritme.

UTEN tilbakekopiering b[] til a[]

```
void flette (int [] a, int [] b, int v, int h, int dybde){
        if (h-v < INSERT_LIM && (dybde%2 == 0)) {
            insertSort (a,v,h);
        } else {
                int mid = (v+h)/2;
                flette (b,a,v,mid, dybde+1);       // sort left part  b[v..mid]
                flette (b,a,mid+1,h, dybde+1);   // sort right part b[mid+1..h]

                int vInd = v, hInd = mid+1, aInd = v;;

                // loop until all elenents moved to 'a'
                while (aInd <= h) {
                    // move left elements while all right elem moved or left smaller
                     while ( vInd <= mid && ( hInd == h+1 || b[vInd] <= b[hInd]  )) {
                                a[aInd++] = b[vInd++];
                     }

                    // move right elements while all left elem moved or right smaller
                     while ( hInd <= h   && ( vInd == mid+1|| b[hInd] <= b[vInd] )) {
                                a[aInd++] = b[hInd++];
                     }
                }
        } // end else
} // end flette
```

Med tilbakekopiering b[] til a[]

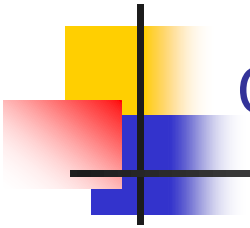
```
void flette (int [] a, int [] b, int v, int h){
        if (h-v < INSERT_LIM ) {
            insertSort (a,v,h);
        } else {
                int mid = (v+h)/2;
                flette (a,b,v,mid, dybde+1);       // sort left part  b[v..mid]
                flette (a,b,mid+1,h, dybde+1);   // sort right part b[mid+1..h]

                int vInd = v, hInd = mid+1, bInd = v;;

                // loop until all elenents moved to 'a'
                while (bInd <= h) {
                    // move left elements while all right elem moved or left smaller
                     while ( vInd <= mid && ( hInd == h+1 || a[vInd] <= a[hInd]  )) {
                                b[bInd++] = a[vInd++];
                     }

                    // move right elements while all left elem moved or right smaller
                     while ( hInd <= h   && ( vInd == mid+1|| a[hInd] <= a[vInd] )) {
                                b[bInd++] = a[hInd++];
                     }
                }
                for (int i = v; i <= h; i++) a[i] = b[i];
        } // end else
} // end flette
```

```
// i del A
  int [] a;
   static int LEVEL_MAX =4;
   int numThreads =1;

   public  void sort (int [] to) {
         int [] from = new int [to.length];
         System.arraycopy(to,0,from,0,a.length);
         CountDownLatch toChild = new CountDownLatch (1);
         pFlette(from,to,0, from.length-1,0,toChild);
         try{
              toChild.await();
         } catch (Exception e) {};
       }
   } // end sort

// i del  B
class Para implements Runnable{
         int [] a,c;
         int low,high,level;
         CountDownLatch fromParent;

         Para(int []a,int[]c,int i,int j,int level, CountDownLatch fromParent) {
            this.a=a; this.c = c; low =i; high=j;this.level=level;
            this.fromParent = fromParent;
          }

          public void run() {
              numThreads++;
              pFlette(a,c,low,high,level,fromParent);
          } // end run
} // end Para
```

Parallell flette – starter med kall på sort(a) med tidtaking rundt

```
void pFlette (int [] from, int [] to, int  aStart, int bEnd, int level, CountDownLatch  fromParent) {
      int bStart = (bEnd + aStart+1)/2;

      if ( level < LEVEL_MAX  ){
         // since < another parallel layer

          CountDownLatch  toChild = new CountDownLatch(LEVEL_MAX-level) ;

          new Thread(new Para(to,from,bStart,bEnd,level+1,toChild)).start();//right part
          pFlette (to,from, aStart,bStart-1, level+1,fromParent); // left part

          try{                        //her kunne vi brukt join();
             toChild.await();
          } catch (Exception e) {};
      } else  if (bEnd-aStart <= 50 ) {
                insertSort (a,aStart,bEnd);
       } else {
          // resten av treet raskere med rekursjon
          if(bStart-aStart > 1)  pFlette (to, from,aStart, bStart-1,level+1,null); //
          if(bEnd-bStart>0)     pFlette (to,from, bStart, bEnd,level+1,null); //
      }
```

```
      // on backtrack mergesort into one partition
     // from[aStart..bStart-1] and from[bStart..bEnd] to to[aStart..bEnd]
       int aPek = aStart,
       bPek = bStart,
       aEnd = bStart-1,
       next = aStart;

       while (aPek <= aEnd && bPek <=bEnd) {
           if (from[aPek] < from[bPek]) {
              to[next++] = from[aPek++];
           } else {
              to[next++] = from[bPek++];;
           }
     }
     // at most  one of these loops will now perfom
     while (aPek <= aEnd) { to[next++] = from[aPek++];  }
     while (next <= bEnd && bPek <= bEnd) { to[next++] = from[bPek++];}

     if ( level <= LEVEL_MAX )
     fromParent.countDown();

  }// end pFlette
```



Om eksamen

- Kunnskapsspørsmål: oppg.1 og 2 (synkronisering og Amdahl, Gustafson)
- Hvordan virker synkronisering oppg.4 (Semaphore) og 6 (ulik plassering av setninger og deklarasjoner
- Skissert et parallelt program – oppg. 3 (Goldbach)
- Lag et sekvensielt og parallelt program – oppgavene 5 ('lett') og 7 (vanskeligere)

- Ikke til eksamen : bevis en matematisk sats.
- 74 kandidater til eksamen.

LYKKE TIL !