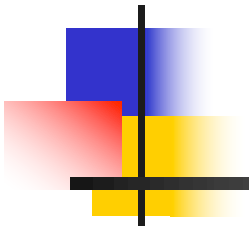
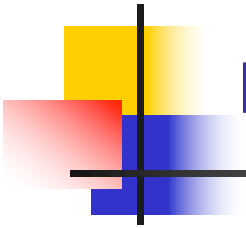


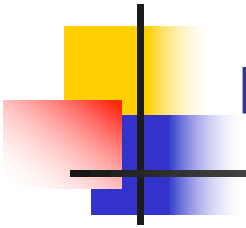
IN3030 Uke 6, våren 2021

Eric Jul
Programming Technology Group
Programming Section
Inst. for informatikk
Universitetet i Oslo



Plan for uke 06

1. Train Collisions in a mountain pass between Bolivia and Peru – real life synchronization problems
2. Oblig comments
3. Modellkode2-forslag for testing av parallell kode
4. Ulike løsninger på i++
5. Vranglås - et problem vi lett kan få (og unngå)
6. Ulike strategier for å dele opp et problem for parallellisering:
7. Om primtall – Eratosthenes Sil (ES)
8. Hvordan representere (ES) effektivt i maskinen

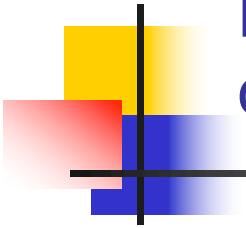


Reasons to fail oblig 1

Reasons to fail oblig 1:

- NO Report (!)
- Lacking tables in the report
- Lacking explanation in the report
- Lacking diagrams in the report
- Not thread safe code
- Too much synchronization

Modell-kode for tidssammenligning av (enkle) parallelle og sekvensiell algoritmer

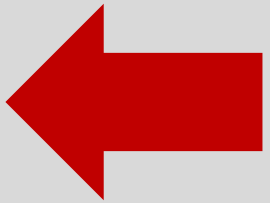


- En god del av dere har laget programmer som virker for:
  - Kjøre både den sekvensielle og parallelle algoritmen
  - Greier å kjøre begge algoritmene 'mange' ganger for å ta mediantiden for sekvensiell og parallell versjon
  - Helst skriver resultatene ut på en fil for senere rapport-skriving
  - Dere kan slappe av nå, og se på Arnes løsning
- For dere andre skal jeg gjennomgå Arnes kode slik at dere har et skjelett å skrive kode innenfor
  - Det mest interessante i dette kurset er tross alt hvordan vi:
    - Deler opp problemet for parallellisering
    - Hvordan vi synkroniserer i en korrekt parallell løsning.
- Eksempel: utfør `i++` i alt N gange
- Poeng: want *safe* AND *fast*

```
class Para implements Runnable{
            int ind, minI=0, fra,til,num;
            Para(int i) { ind =i; } // konstruktor

    /*** HER er dine egen parallelle metoder som IKKE er synchronized */
    void  parallellMetode(int ind) {
            for (int j=0; j<n; j++){
                        minI++;
            }
           allI [ind] = minI;
     }

     void paraInitier(int n) {
            num = n/antTraader;
            fra = ind*num;
            til = (ind+1)*num;
            if (ind == antTraader-1) til =n;
             minI =0;
     }// end paraInitier
```

```
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import easyIO.*;
// file: Modell2.java
// Lagt ut  feb 2017 - Arne Maus, Ifi, UiO
// Som BARE et eksempel, er problemet med å øke fellesvariabelen i  n*antKjerner  ganger løst

class Modell2{// ****** Problemets FELLES DATA HER
              int i;
              final String navn = "TEST AV i++ med synchronized oppdatering";
               // Felles system-variable - samme for 'alle' programmer
               CyclicBarrier vent, ferdig, heltferdig ; // for at trådene og main venter på hverandre
               int antTraader;
               int antKjerner;
               int numIter ;          // antall ganger for å lage median (1,3,5,,)
               int nLow,nStep,nHigh;  // laveste, multiplikator, hoyeste n-verdi
               int n;                 // problemets størrelse
               String filnavn;
               volatile boolean stop = false;
               int med;
               Out ut;
                int [] allI;

               double [] seqTime ;
               double [] parTime ;
```

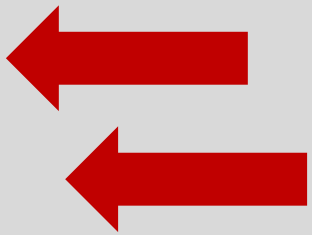
```
/** for også utskrift på fil */
synchronized void println(String s) {
        ut.outln(s);
        System.out.println(s);
 }

 /** for også utskrift på fil */
 synchronized void print(String s) {
         ut.out(s);
         System.out.print(s);
  }
  /** initieringen i main-tråden */
  void intitier(String args) {
        nLow = Integer.parseInt(args[0]);
        nStep = Integer.parseInt(args[1]);
        nHigh = Integer.parseInt(args[2]);
        numIter = Integer.parseInt(args[3]);
        seqTime = new double [numIter];
        parTime = new double [numIter];
        ut = new Out(args[4], true);

         antKjerner = Runtime.getRuntime().availableProcessors();
         antTraader = antKjerner;
         vent   = new CyclicBarrier(antTraader+1); //+1, også main
         ferdig = new CyclicBarrier(antTraader+1); //+1, også main
          heltferdig = new CyclicBarrier (2);       // main venter på tråd 0
          allI = new int [antTraader];
```



```
                // start trådene
                 for (int i = 0; i< antTraader; i++)
                        new Thread(new Para(i)).start();

     } // end initier

public static void main (String [] args) {
   if ( args.length != 5) {
     System.out.println("use: >java Modell2 <nLow> <nStep> <nHigh> <num iter> <fil>");
   } else {
                new Modell2().utforTest(args);
   }
} // end main
```

```
void  utforTest () {
        intitier();
        println("Test av  "+ navn+ "\n med "+
        antKjerner + " kjerner , og " + antTraader+" traader,  Median av:" + numIter+" iterasjon\n");
        println("\n     n      sekv.tid(ms)   para.tid(ms)    Speedup ");

        for (n = nHigh; n >= nLow; n=n/nStep) {
              for (med = 0; med < numIter; med++) {
                        long t = System.nanoTime();  // start tidtagning parallell
                        // Start alle trådene parallell beregning nå
                          try {
                             vent.await();   // start en parallell beregning
                             ferdig.await(); // vent på at trådene er ferdige
                          } catch (Exception e) {return;}
                          try {   heltferdig.await(); // vent på at tråd 0 har summert svaret
                          } catch (Exception e) {return;}

                           // her kan vi lese svaret

                          t = (System.nanoTime()-t);
                          parTime[med] =t/1000000.0;
                           println(« svaret er:» + i + «for n =» +n);

                           t = System.nanoTime(); // start tidtagning sekvensiell
                           //**** KALL PÅ DIN SEKVENSIELLE METODE  H E R ********
                           sekvensiellMetode (n,numIter);
                           t = (System.nanoTime()-t);
                           seqTime[med] =t/1000000.0;
              } // end for med
```

```
            println(Format.align(n,10)+
                    Format.align(median(seqTime,numIter),12,3)+
                    Format.align(median(parTime,numIter),15,3)+
                    Format.align(median(seqTime,numIter)/median(parTime,numIter),13,4));
        } // end n-loop
       exit();
 } // utforTest

/** terminate parallel threads*/
void exit() {
        stop = true;
        try { // start the other threads and they terminate
                    vent.await();
        } catch (Exception e) {return;}
        ut.close();
 }   // end exit

 /*** HER er din egen sekvensielle metode som  selvsagt IKKE ER synchronized, */
 void sekvensiellMetode (int n,int numIter){
         for (int j=0; j<n; j++){
                        i++;
          }
} // end sekvensiellMetode

/*** Her er evt. de parallelle metodene som ER synchronized  - treig*/
 synchronized void addI() {
         i++;
 }
```



```
public void run() { // Her er det som kjores i parallell:
        while (! stop) {
            try {  // wait on all other threads + main
                    vent.await();
             } catch (Exception e) {return;}
            if (! stop) {
                    paraInitier(n);
                   //**** KALL PÅ DINE PARALLELLE METODER  H E R ********
                     parallellMetode(ind); // parameter: traanummeret: ind

                     try{ // make all threads terminate
                             ferdig.await();
                     } catch (Exception e) {}
            } // end ! stop thread

              // tråd nr 0 adderer de 'numThreads' minI - variablene til en felles verdi
           if (ind == 0) { i =0;
              for (int j = 0; j < antTraader; j++) { i += allI[j]; }

              try {   heltferdig.await();  // si fra til main at tråd 0 har summert svaret
              } catch (Exception e) {return;}

        } // end tråd 0
     } // end while !stop
  } // end run

} // end class Para
```

Hvor lang tid tar et synchronized kall? Demoeks. hadde n synchroniced metode for all skriving til felles 'i'.

- Kjørte modell-koden for n=10 000 000 (3 ganger)

```
M:\INF2440Para\ModelKode>java Modell2 100 10 10000000 3 test-14feb.txt
Test av  TEST AV i++ med synchronized oppdatering
 med 8 kjerner , og 8 traader,  Median av:3 iterasjoner


    n       sekv.tid(ms)   para tid(ms)    Speedup
 10000000        6.704      11024.957       0.0006
  1000000        0.658       1084.411       0.0006
   100000        0.071         98.566       0.0007
    10000        0.007         10.927       0.0006
     1000        0.001          1.057       0.0010
      100        0.000          0.192       0.0018
```

- Svar: Et synchronized kall tar ca. 1000/(8*1000 000)ms = 0.15 μs = 150ns. = ca. 500 instruksjoner.

Finnes det alternativer & riktig kode?

- a) Bruk av ReentrantLock (import java.util.concurrent.locks.*;)

```
// i felledata-omraadet i omsluttende klasse
ReentrantLock laas = new ReentrantLock();
          .............
 /*** HER skriver du eventuelle  parallelle metoder som ER synchronized */
  void addI() {
        laas.lock();
              i++;
        try{ laas.unlock();} catch(Exception e) {return;}
 } // end addI
```

- Kjøring:

```
M:\INF2440Para\ModelKode>java ModellAlt 1000000 5 test.txt
Test av  TEST AV i++ med ReentrantLock oppdatering
 med 8 kjerner , og 8 traader

Median of 5:  Sekv. tid:        0.70 ms, Para tid:      212.44 ms,
 Speedup:  0.003, n = 1000000
```

- 5x fortere enn synchronized !

b) Alternativ b til synchronized: Bruk av AtomicInteger

- Bruk av AtomicInteger (import java.util.concurrent.atomic.*;)

```
 // i felledata-omraadet i omsluttende klasse
AtomicInteger i = new AtomicInteger();
        .............
/*** HER skriver du eventuelle  parallelle metoder som ER synchronized */
 void addI() {
      i.incrementAndGet();
 } // end addI
```

- Kjøring:

```
M:\INF2440Para\ModelKode>java ModellAlt 1000000 5 test.txt
Test av  TEST AV i++ med AtomicInteger oppdatering
 med 8 kjerner , og 8 traader

Median of 5:  Sekv. tid:        0.66 ms, Para tid:       235.91 ms,
 Speedup:  0.003, n = 1000000
```

- **Konklusjon:** Både ReentrantLock og AtomicInteger er 5x fortere enn synchronized metoder + at all parallell kode kan da ligge i den parallelle klassen.

c) : Lokal kopi av i hver tråd og en synchronized oppdatering fra hver tråd til sist.

```
/*** HER skriver du eventuelle  parallelle metoder som ER synchronized */
        synchronized void addI(int tillegg) {
                i = i+ tillegg;
        } // end addI
        .......
         class Para implements Runnable{
                int ind;
                int minI=0;
              .....
                 /*** HER skriver du parallelle metode som IKKE er synchronized */
                void  parallellMetode(int ind) {
                        for (int j=0; j<n; j++)
                                minI++;
                } // end parallellMeode

                public void run() {
                                ......
                                if (! stop) {
                                  //**** KALL PÅ DIN PARALLELLE METODE  H E R ********
                                        parallellMetode(ind);
                                        addI(minI);
                                        try{ .....
```

Kjøring av alternativ C (lokal kopi først):

- Kjøring:

```
M:\INF2440Para\ModelKode>java ModellAlt 1000000 5 test.txt
Test av  TEST AV i++ forst i lokal i i hver traad, saa synchronized
oppdatering av i,  med 8 kjerner , og 8 traader

Median of 5:  Sekv. tid:        0.71 ms, Para tid:        0.47 ms,
 Speedup:  1.504, n = 1000000
```

- Betydelig raskere, ca. 500x enn alle de andre korrekte løsningene og noe raskere enn den sekvensielle løsningen
- Eneste riktige løsning som har speedup > 1.
- **Husk:** Ingen vits å lage en parallell algoritme hvis den sekvensielle er raskere.

Oppsummering av kjøretider

Løsning	kjøretid	Speedup
Sekvensiell	0,70 ms	1
Bare synchronized	1015,72 ms	0,001
ReentrantLock	212.44 ms	0,003
AtomicInteger	235,91 ms	0,003
Lokal kopi, så synchronized oppdatering 1 gang per tråd	0,47 ms	1,504

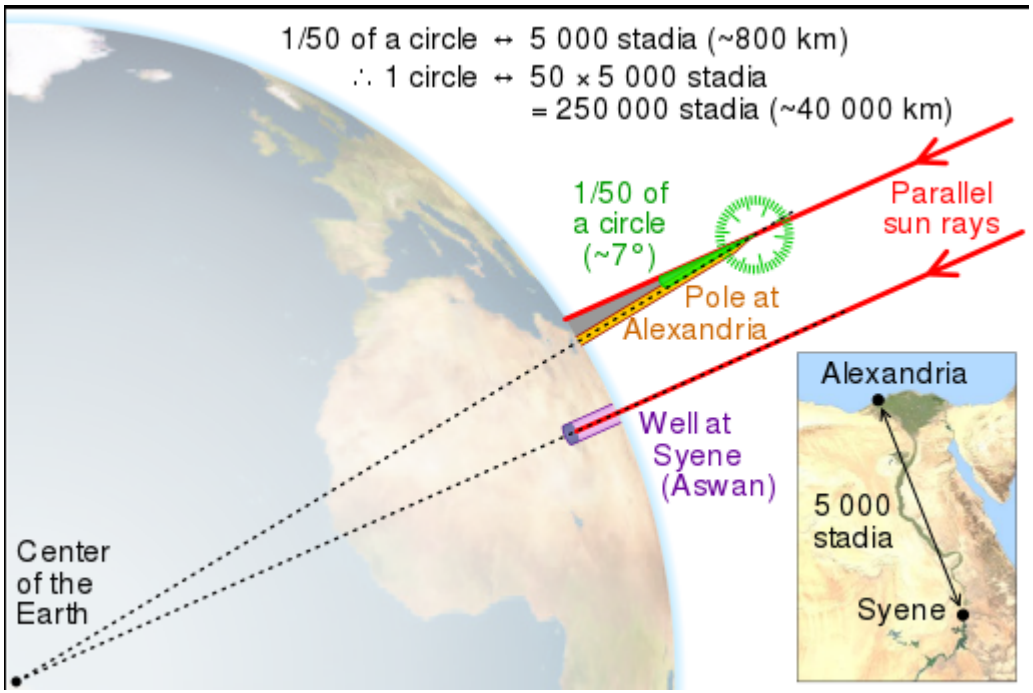
- Oppsummering:
  - Synkronisering av skriving på felles variable tar lang tid, og må minimeres (og synchronized er spesielt treg)
  - Selv den raskeste er 500x langsommere enn å ha lokal kopi av fellesvariabel int i , og så addere svarene til sist.
  - **Synkronisering kan «drepe»!**

Om primtall – og om Eratosthenes sil (oblig 3)

- Oblig 3: Primtall og faktorisering av ikke-primtall.
- Et primtall er :
  Et heltall som bare lar seg dividere med 1 og seg selv.
  - 1 er ikke et heltall (det mente mange på 1700-tallet, og noen mener det fortsatt)
- Ethvert tall N > 1 lar seg faktorisere som et produkt av primtall:
  - $N = p_1 * p_2 * p_3 * \ldots\ldots\ldots * p_k$
  - Denne faktoringen er entydig (pånær rækkefølge); dvs. den eneste faktoriseringen av N – gjøres entydig hvis tall i faktoriseringen sorteres
  - Hvis det bare er ett tall i denne faktoriseringen, er N selv et primtall

Litt mer om Eratosthenes

Eratosthenes, matematikker, laget også et estimat på jordas radius som var < 1,5% feil, grunnla geografi som fag, fant opp skuddårsdagen + at han var sjef for Biblioteket i Alexandria (den tids største forskningsinstitusjon).



End of lecture uke 06 v2021

$\sqrt{N}$