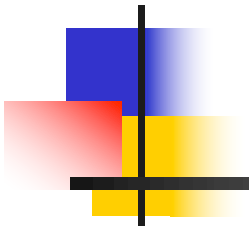
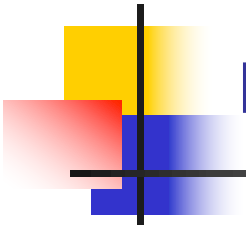


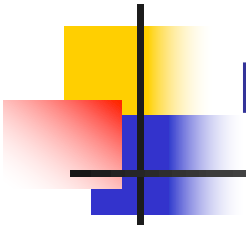
IN3030 Uke 6, våren 2020

Eric Jul
Programming Technology Group
Programming Section
Inst. for informatikk



Resume of Week 05 lecture

1. Sketch about synchronization
2. Different synchronizations for finding Max in an array:
    1. No synch: WRONG
    2. Use Java's `synchronized` keyword: SLOW
    3. Using a ReentrantLock: BETTER BUT STILL SLOW
    4. Using a Reentrant Lock only when necessary: BETTER
    5. Postponing synchronization by using local max and then synching ONLY at the end: EXCELLENT
    6. Using CyclicBarrier & letting Thread 0 find max of all the local max *without* synchronization: EXCELLENT
3. Java Microbenchmark Harness – how to get more detailed info out of the system



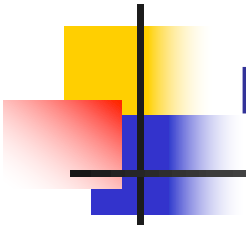
Plan for uke 06

Første time:

1. Oblig 1: comments
2. Modellkode2 -forslag for testing av parallell kode
3. Ulike løsninger på i++
4. Vranglås - et problem vi lett kan få (og unngå)

Annen time:

1. Ulike strategier for å dele opp et problem for parallellisering:
2. Om primtall – Eratosthenes Sil (ES)
3. Hvordan representere (ES) effektivt i maskinen
4. Oblig 3: Primtall

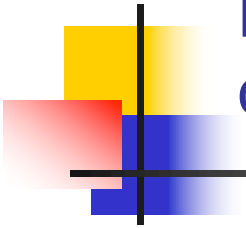


Reasons to fail oblig 1

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

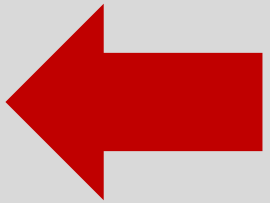
Modell-kode for tidssammenligning av (enkle) parallelle og sekvensiell algoritmer



- 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.

```
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 ;
```

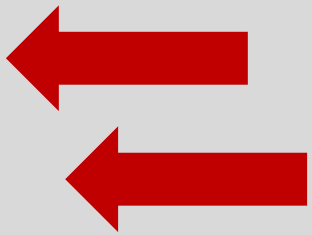
```
/** for også utskrift på fil */
synchronized void println(String s) {
        ut.outln(s);
        System.out.println(s);
 }

 /** 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++;
 }
```



```
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
```

```
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»!**

Vranglås *(deadlock):* Rekkefølgen av flere synkroniseringer fra flere ulike tråder: går det alltid bra?

- Anta at du har to ulike parallelle klasser A og B og at som begge bruker to felles synkroniseringsvariable: Semphorene 'vent' og 'ferdig' (begge initiert til 1).
- A og B synkroniserer seg *ikke* i samme rekkefølge:

```
A sier:
try{
   vent. acquire();
   ferdig. acquire();
} catch(Exception e)
{return;}

…gjør noe….

ferdig.release();
vent.release();
```

```
B sier:
try{
     ferdig. acquire();
     vent. acquire();
} catch(Exception e)
{return;}

…gjør noe….

vent.release();
ferdig.release();
```

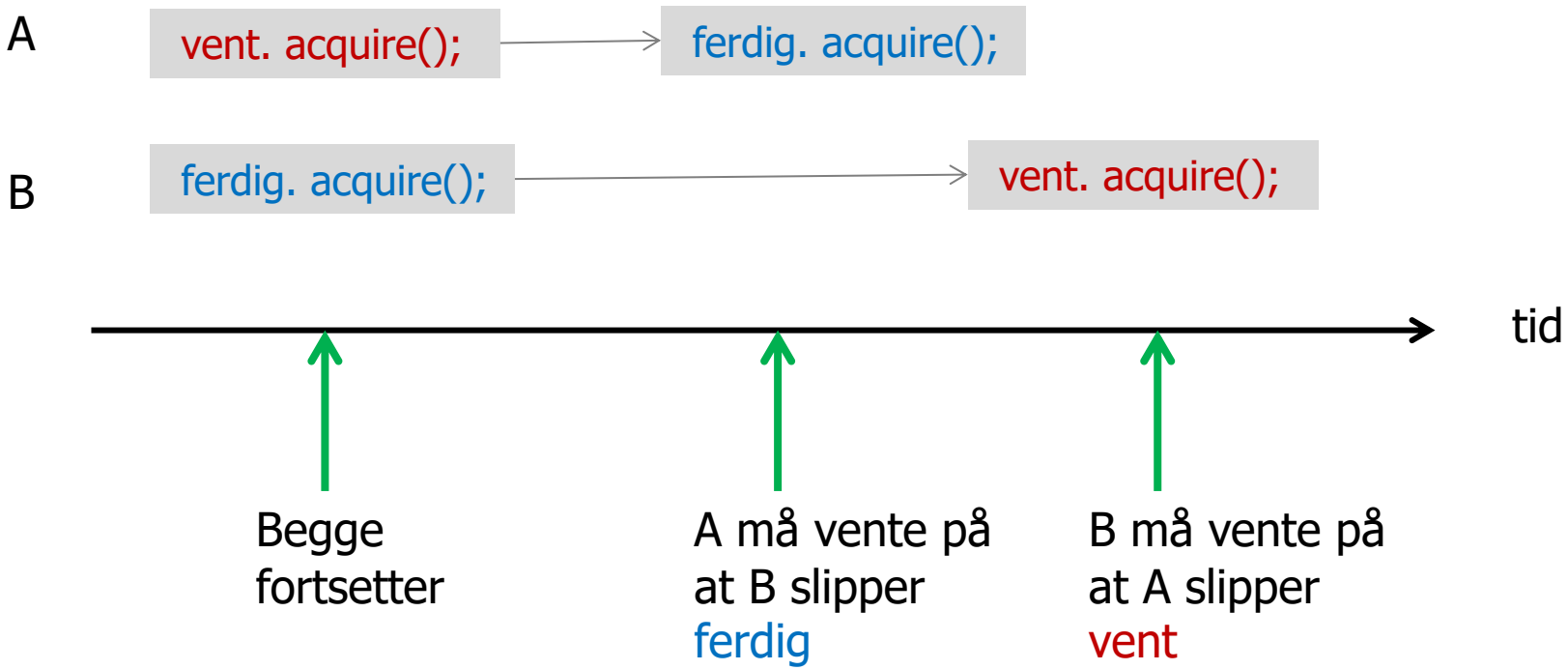
Ytre klasse VrangLaas med to indre klasser SkrivA og SkrivB

```
public class VrangLaas{
    int  a=0,b=0, antGanger;              // Felles variable a,b
    Semaphore ferdig, vent ;
    SkrivA aObj;
    SkrivB bObj;

     public static void main (String [] args) {
          if (args.length != 1) {
                System.out.println(" bruk: java <ant ganger oeke> );
          } else {
                int antKjerner = Runtime.getRuntime().
                                 availableProcessors();
                System.out.println("Maskinen har "+ antKjerner +
                   " prosessorkjerner.\n");
                 VrangLaas p =  new VrangLaas();
                 p.antGanger  = Integer.parseInt(args[0]);
                 p.utfor();
           }
        } // end main

        void  utfor () {
              vent   = new Semaphore(1);
              ferdig = new Semaphore(1);
               (aObj =  new SkrivA()).start();
               (bObj =  new SkrivB()).start();
        } // utfor
```

```
class SkrivA extends Thread{
        public void run() {
            for (int j = 0; j<antGanger; j++) {
                    try {  // wait
                          vent.acquire();
                          ferdig.acquire();
                    } catch (Exception e) {return;}

                    a++;
                    System.out.println(" a: " +a);
                     vent.release();
                     ferdig.release();
            } // end j
        } // end run A
} // end class SkrivA

class SkrivB extends Thread{
        public void run() {
            for (int j = 0; j<antGanger; j++) {
                    try {  // wait
                        ferdig.acquire();
                        vent.acquire();
                    } catch (Exception e) {return;}

                    b++;
                    System.out.println(" b: " +b);
                     vent.release();
                     ferdig.release();
            } // end j
        } // end run B
  } // end class SkrivB
} // end class VrangLaas
```

Vranglås – del 2

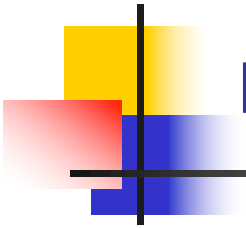
- Dette kan gi såkalt vranglås (deadlock) ved at begge trådene venter på at den andre skal bli gå videre.
- Hvis operasjonene blandes slik går det galt (og det skjer også i praksis!)



Vranglås - løsning

- A og B venter på hverandre til evig tid – programmet ditt henger!
- Løsning: Følg disse enkle regler i hele systemet (fjerner **all** vranglås):

1. Hvis du skal ha flere synkroniserings-objekter i programmet, så må de sorteres i en eller annen rekkefølge.
2. Alle tråder som bruker to eller flere av disse, må be om å få vente på dem (s.acquire(),..) i samme rekkefølge som de er sortert !
3. I hvilken rekkefølge disse synkroniserings-objektene slippes opp (s. release(),..) har mer med hvem av de som venter man vil slippe løs først, og er ikke så nøye; gir ikke vranglås.

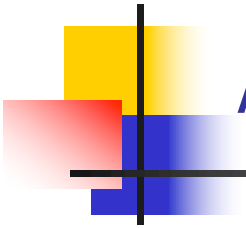


End of first lecture IN3030 week 06

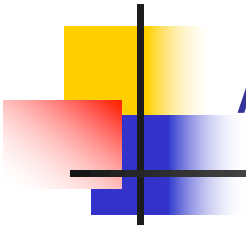
Om å parallelliser et problem

- Utgangspunkt: Vi har en sekvensiell effektiv og riktig sekvensiell algoritme som løser problemet.
- Vi kan dele opp både koden og data (hver for seg?)
- Vanligst å dele opp data
  - Som oftest deler vi opp data, og lar 'hele' koden virke på hver av disse data-delene (en del til hver tråd).
  - Eks: Matriser
    - radvis eller kolonnevis oppdeling av C til hver tråd
    - Omforme data slik at de passer bedre i cachene (transponere B)
  - Rekursiv oppdeling av data ('lett')
    - Eks: Quicksort
- Også mulig å dele opp koden:
  - Alternativ Oblig3 i INF1000: Beregning av Pi (3,1415..) med 17 000 sifre med tre ArcTan-rekker
  - Primtalls-faktorisering av store tall N for kodebrekking:
    - $N = p_1 * p_2$

Å dele opp algoritmen



- Koden består en eller flere steg; som oftest i form av en eller flere samlinger av løkker (som er enkle, doble, triple..)
- Vi vil parallellisere med k tråder, og hver slikt steg vil få hver sin parallellisering med en CyclickBarrier-synkronisering mellom hver av disse delene + en synkronisert avslutning (join(), ..).
- Eks:
  - finnMax – hadde ett slikt steg: for (int i = 0 …n-1) -løkke
  - MatriseMult hadde ett slikt steg med trippel-løkke
  - Flere steg mulig: Eksempler senere i kurs (Radix)



Å dele opp data – del 2

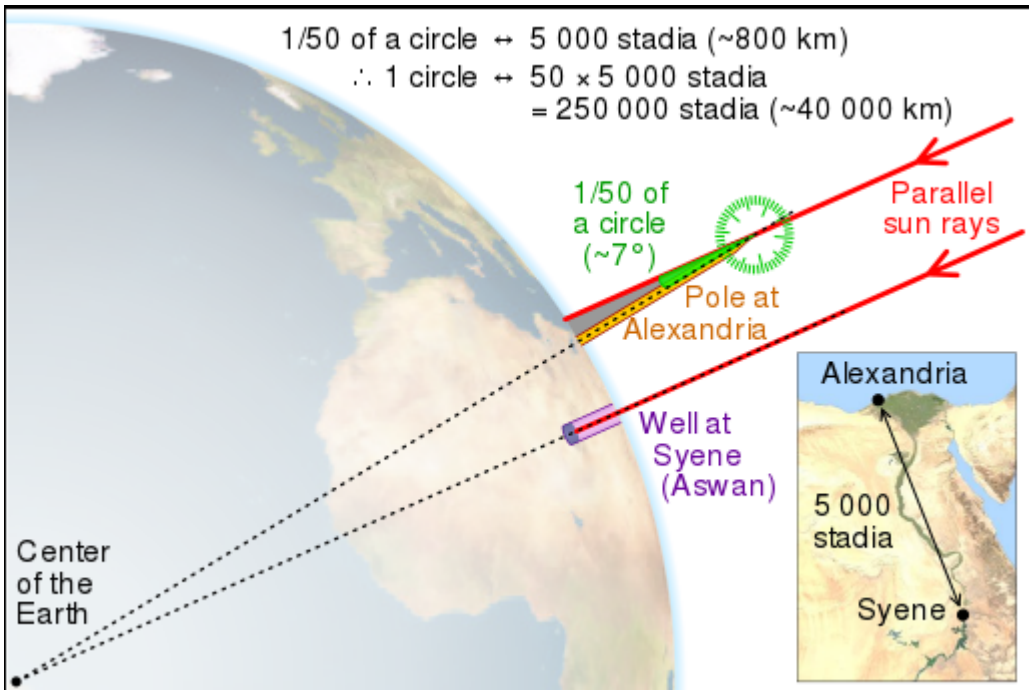
- For å planlegge parallellisering av ett slikt steg må vi finne:
  - Hvilke data i problemet er lokale i hver tråd?
  - Hvilke data i problemet er felles/delt mellom trådene?
- Viktig for effektiv parallell kode.
  - Hvordan deler vi opp felles data (om mulig)
  - Kan hver tråd beregne hver sin egen, disjunkte del av data
  - Færrest mulig synkroniseringer (de tar 'mye' tid)

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).



2 måter å lage primtall

Ønsker at finne alle primtal $p_i < N$

- Lage en tabell over alle de primtallene vi trenger
  - Eratosthene sil

- Dividere alle tall < N med alle oddetall < $\sqrt{N}$?
  - Divisjonsmetoden
  - (Hvorfor ikke oddetallmopp til N?)

Hvad er raskest?

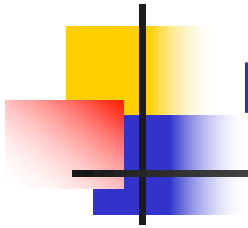
- A) Med Eratosthenes sil:

```
Z:\INF2440Para\Primtall>java PrimtallESil 2000000000
 max primtall m:2000000000
Genererte alle primtall <= 2000000000  paa     18 949 millisek
med Eratosthenes sil  og det største primtallet er:1999999973
```

- Med gjentatte divisjoner

```
Z:\INF2440Para\Primtall>java PrimtallDiv 2000000000
Genererte alle primtall <=2000000000  paa   1 577 302  millisek med
 divisjon , og det største primtallet er:1999999973
```

- Å lage primtallene p og finne dem ved divisjon (del på alle oddetall < SQRT(p), p = 3,5,7,..) er ca. 100 ganger langsommere enn Eratosthenes avkryssings-tabell (kalt Eratosthenes sil).



Finne primtall -- Eratosthenes sil

- Hvordan?

Å lage og lagre primtall (Erotosthenes sil)

- Som en bit-tabell (1- betyr primtall, 0-betyr ikke-primtall)
  - Påfunnet i jernalderen av Eratosthenes (ca. 200 f.kr)
  - Man skal finne alle primtall < M
  - Man finner da de første primtallene og krysser av alle multipla av disse (N.B. dette forbedres/endres senere):
    - Eks: 3 er et primtall, da krysses 6, 9,12,15,.. Av fordi de alle er ett-eller-annet-tall (1,2,3,4,5,..) ganger 3 og følgelig selv ikke er et primtall. 6=2*3, 9 = 3*3,
      12 =2*2*3, 15 = 5*3, ..osv
    - De tallene som *ikke blir* krysset av, når vi har krysset av for alle primtallene vi har, er primtallene
- Vi finner 5 som et primtall fordi, etter at vi har krysset av for 3, finner første ikke-avkryssete tall: 5, som da er et primtall (og som vi så krysser av for, …finner så 7 osv)

Litt mer om Eratothenes sil

- Vi representerer ikke partallene på den tallinja som det krysses av på fordi vi vet at 2 er et primtall (det første) og at alle andre partall er ikke-primtall.
- Har vi funnet et nytt primtall p, for eksempel. 5, starter vi avkryssingen for dette primtallet først for tallet p*p (i eksempelet: 25), men etter det krysses det av for p*p+2p, p*p+4p,.. (i eksempelet 35,45,55,…osv.). Grunnen til at vi kan starte på p*p er at alle andre tall t < p*p slik det krysses av i for eksempel Wikipedia-artikkelen har allerede blitt krysset av andre primtall < p.
- Det betyr at for å krysse av og finne alle primtall < N , behøver vi bare å krysse av på denne måten for alle primtall p ≤ sqrt(N). Dette sparer svært mye tid.

Vise at vi trenger bare primtallene <10 for å finne alle primtall < 100, avkryssing for 3 (3*3, 9+2*3,9+4*3, ….)

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75	77	79
81	83	85	87	89
91	93	95	97	99

1	3	5	7	9
11	13	**15**	17	19
21	23	25	**27**	29
31	**33**	35	37	**39**
41	43	**45**	47	49
51	53	55	**57**	59
61	**63**	65	67	**69**
71	73	**75**	77	79
81	83	85	**87**	89
91	**93**	95	97	**99**

Avkryssing for 5 (starter med 25, så 25+2*5, 25+4,5,..):

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75	77	79
81	83	85	87	89
91	93	95	97	99

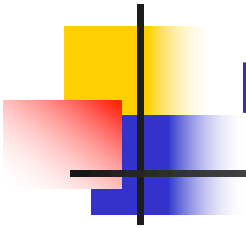
1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

Avkryssing for 7 (starter med 49, så 49+2*7,49+4*7,.):

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63 63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

Er nå ferdig fordi neste primtall vi finner: 11, så er 11*11=121 utenfor tabellen

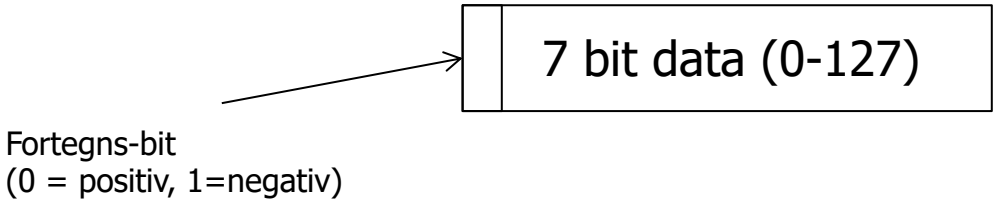


Hvordan representeres tallene?

- Kun oddetall – 2 kjenner vi!
- Array of Boolean?
  - Problem: 32 bit per primtall
- Kompakter bitarray
    - Kun 1 bit per oddetall

Hvordan bruke 8 eller 7 bit i en **byte-array** for å representere primtallene

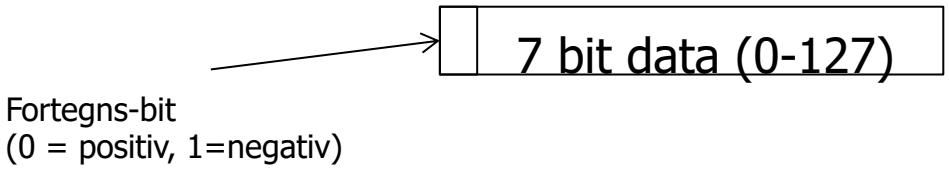
En byte = 8 bit heltall:



- Vi representer alle oddetallene (1,3,5,,,) som ett bit (0= ikke-primtall, 1 = primtall)
- Bruke alle 8 bit :
  - Fordel: mer kompakt lagring og litt raskere(?) adressering
  - Ulempe: Kan da ikke bruke verdien i byten direkte (f.eks som en indeks til en array), heller ikke +,-,* eller /-operasjonene på verdien
- Bruke 7 bit:
  - Fordel: ingen av ulempene med 8 bit
  - Ulempe: Tar litt større plass og litt langsommere(?) adressering

Hvordan representere 8 (eller 7) bit i en byte-array

byte = et 8 bit heltall



- Bruker alle 8 bitene til oddetallene:
  - Anta at vi vil sjekke om tallet ***k*** er et primtall, sjekk først om ***k*** er 2, da ja, hvis det er et partall (men ikke 2) da nei – ellers sjekk så tallets bit i byte-arrayen
    - Byte nummeret til ***k*** i arrayen er da:
      - Enten: ***k***/16, eller: ***k*** >>>4 (shift 4 høyreover uten kopi av fortegns-bitet er det samme som å dele med 16)
    - Bit-nummeret er i denne byten er da enten (***k*** %16)/2 eller (***k*** &15)>>1
  - Hvorfor dele på 16 når det er 8 bit
    - fordi vi fjernet alle partallene – egentlig 16 tall representert i første byten, for byte 0: tallene 0-15
  - Om så å finne bitverdien – se neste lysark.

Bruke 7 bit i hver byte i arrayen

- Anta at vi vil sjekke om tallet ***k*** er et primtall sjekk først om ***k*** er 2, da ja, ellers hvis det er et partall (men ikke 2) da nei – ellers:
- Sjekk da tallets bit i byte-arrayen
  - Byte nummeret til ***k*** i arrayen er da: ***k*** /14
  - Bit-nummeret er i denne byten er da: (***k*** %14)/2
- Nå har vi byte nummeret og bit-nummeret i den byten. Vi kan da ta AND (&) med det riktige elementet i en av de to arryene som er oppgitt i skjelett-koden og teste om svaret er 0 eller ikke.
- Hvordan sette alle 7 eller 8 bit == 1 i alle byter)
  - 7 bit: hver byte settes = 127 (men bitet for 1 settes =0)
  - 8 bit: hver byte settes = -1 (men bit for 1 settes = 0)

- Konklusjon: bruk 8 eller 7 bit i hver byte (valgfritt) i Oblig3

Faktorisering av et tall M i sine primtallsfaktorer

- Vi har laget og lagret ved hjelp av Erotosthanes sil alle (unntatt 2) primtall < N i en bit-array over alle odde-tallene.
  - 1 = primtall, 0=ikke-primtall
  - Vi har krysset ut de som ikke er primtall
- Hvordan skal vi så bruke dette til å faktorisere et tall M < N*N ?
- **Svar:** Divider M med alle primtall $p_i < \sqrt{M}$ (p_i =2,3,5,..), og hver gang en slik divisjon M%p_i ==0, så er p_i en av faktorene til M. Vi forsetter så med å faktorisere ett mindre tall M'=M/p_i.
- Faktoriseringen av M = p_i *..* p_k er da produktet av alle de primtall som dividerer M uten rest.
- HUSK at en p_i kan forekommer flere ganger i svaret. eks: 20= 2*2*5, 81 = 3*3*3*3, osv
- Finner vi ingen faktorisering av M, dvs. ingen $p_i \leq \sqrt{M}$ som dividerer M med rest == 0, så er M selv et primtall.

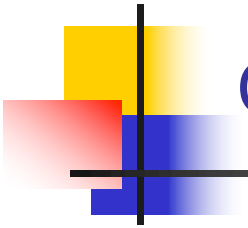
Hvordan parallellisere faktorisering ?

1. Gjennomgås neste uke - denne uka viktig å få på plass en effektiv sekvensiell løsning med om lag disse kjøretidene for N = 2 mill:

```
M:\INF2440Para\Primtall>java PrimtallESil 2000000
 max primtall m:2 000 000
Genererte primtall <= 2000000  paa        15.56 millisek
med Eratosthenes sil (  0.00004182 millisek/primtall)
.....
3999998764380 = 2*2*3*5*103*647248991
3999998764381 = 37*108108074713
3999998764382 = 2*271*457*1931*8363
3999998764383 = 3*19*47*1493093977
3999998764384 = 2*2*2*2*2*7*313*1033*55229
3999998764385 = 5*13*59951*1026479
3999998764386 = 2*3*3*31*71*100964177
3999998764387 = 1163*1879*1830431
3999998764388 = 2*2*11*11*17*23*293*72139
 100 faktoriseringer beregnet paa:     422.0307ms -
 dvs:      4.2203ms. per faktorisering
```

Faktorisering av store tall med 18-19 desimale sifre

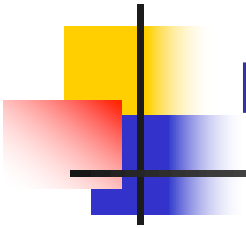
```
Uke5>java PrimtallESil 2140000000
max primtall m:2 140 000 000
bitArr.length:133 750 001
Genererte primtall <= 2 140 000 000  paa    11030.36 millisek
med Eratosthenes sil ( 0.00010530 millisec/primtall)
 antall primtall < 2 140 000 000 er: 104 748 779, dvs:  4.89% ,
 og det største primtallet er: 2 139 999 977

4 579 599 999 999 999 900 = 2*2*3*5*5*967*3673*19421*221303
4 579 599 999 999 999 901 = 45795999999999901
4 579 599 999 999 999 902 = 2*22897999999999951
4 579 599 999 999 999 903 = 3*31*13188589*3733758839
4 579 599 999 999 999 904 = 2*2*2*2*2*19*71*106087842846553
4 579 599 999 999 999 905 = 5*7*1308457142857142 83
.............
4 579 599 999 999 999 997 = 11*416327272727272727
4 579 599 999 999 999 998 = 2*121081*18911307306679
4 579 599 999 999 999 999 = 3*17*19*6625387*713333333
100 faktoriseringer beregnet paa:  333481.4427ms
 dvs:   3334.8144ms. per faktorisering

 largestLongFactorizedSafe: 4 579 599 841 640 001 173= 2139999949*2139999977
```



Oblig 3: Primtall

Oblig Deadlines

IN3030/IN4330 Oblig Dates			
2020	Published	Deadline	Corrected by (first attempt)
O1	22.01.2020	04.02.2020	19.02.2020
O2	05.02.2020	18.02.2020	04.03.2020
O3	19.02.2020	10.03.2020	25.03.2020
O4	11.03.2020	31.03.2020	22.04.2020
O5	15.04.2020	05.05.2020	20.05.2020



Hva har vi sett på i uke 06

1. Modell2-kode for sammenligning av kjøretider på (enkle) parallelle og sekvensielle algoritmer.
2. Hvordan lage en parallell løsning – ulike måter å synkronisere skriving på felles variable
3. Vranglås - et problem vi lett kan få (og unngå)
4. Ulike strategier for å dele opp et problem for parallellisering:
5. Hvorfor lage en avkryssingstabell over alle oddetall for å finne alle primtall (Eratosthenes sil) – steg 1 i Oblig 2
6. Oblig 3: Primtall og faktorisering