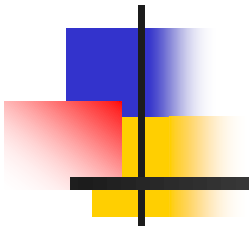


# INF2440 Uke 15, v2014
# oppsummering

Arne Maus
OMS,
Inst. for informatikk

# Hva så vi på i Uke14

I) Sjekke Goldbachs hypotese for n > $4*10^{18}$ .

- Jeg skisserte en meget dårlig (ineffektiv) algoritme for dere.
- Hvordan ser en mer effektiv algoritme ut ?
- Ga noen 'verdensrekorder' på Goldbach-summer > $4*10^{18.}$
- Parallellisering av denne skissert

II) Delaunay triangulering – de beste trekantene!

- Brukes ved kartlegging, oljeleting, bølgekraftverk,..
- Spill-grafikk: ved å gi tekstur, farge og glatte overflater på gjenstander, personer, våpen osv.
- Er egentlig flere algoritmer etter hverandre. Skissering av hvordan disse kan parallelliseres.

# Resten av IN F2440

- 2.mai – denne forelesningen
  - Oppsummering av pensum, presisering av krav til eksamen
- Det blir ingen vanlig gruppeøvelser neste uke, bare orakeltjeneste for fullføring av Oblig3.
  - Dukker det ikke opp noen innen 15 min. er timen avlyst
- 26. mai – prøveeksamen
  - Utdeling av prøveeksamenssett kl. 11.00 utenfor Simula
  - Gjennomgang kl. 15.15 - 17 i Store Aud, (K.Nygaards Hus)
  - Husk at de 'beste' oppgavene spares til eksamen
- 2.juni kl. 14.30
  - Tillatt å ha med all skriftlig materiale
  - Ha med utskrift av alle forelesningene (definerer pensum) , ukeoppgaver og Obligene med dine løsninger
  - Intet elektronisk – ingen maskineksamen

# Hva skal vi se på i Uke15

I) Presisering og utvidelse av en uttalelse om en eller flere tråder i en GUI.

II) Repetisjon og oversikt

i. Hvorfor lage parallelle programmer
ii. Hvordan får vi til parallellitet – lokale og delte data
    i. Synkroniseringsmekanismer
iii. Ekstra problemer med parallellitet (data-kappløp, synlighet og vraglås
iv. JIT-kompilering, cachene og parallellitet
v. Tidtaking og synkronisering
vi. Regler for delte data, og når kan vi byte en av dem
vii. Hvordan parallellisere en sekvensiell algoritme
    i. Oppdeling av data eller verdiene
    ii. Kopier av delte data – lokal oppdatering
    iii. Rekursjon, oppdeling i faser
viii. Ett problem som vel ikke lar seg meningsfylt parallellisere (ny)

# I) Vi kan trenge to tråder i en GUI-applikasjon

- En for å 'spille innholdet'
- En for at brukeren skal kunne betjene knappene – styre f.eks en avspilling.

# 1. Hvorfor lage parallelle programmer?

Det lages ikke raskere maskiner nå, bare flere på hver brikke.

Vi må lære oss å utnytte hele maskinen (aller kjernene i en multi-kjern maskin). Dette kurset: Praktisk kurs i hvordan lage parallelle programmer (algoritmer) som er:

- **Riktige**
  - Parallelle programmer er klart vanskeligere å lage enn sekvensielle løsninger på et problem.
- **Effektive**
  - dvs. raskere enn en sekvensiell løsning på samme problem – dvs speedup >1 (for stor nok n)
  - Speedup = Sekvensiell/parallell tid

***«Ingen grunn til å lage lengere, mer kompliserte parallelle programmer hvis de ikke går fortere»***

# Mange mulige synsvinkler

Mange nivåer i parallellprogrammering:

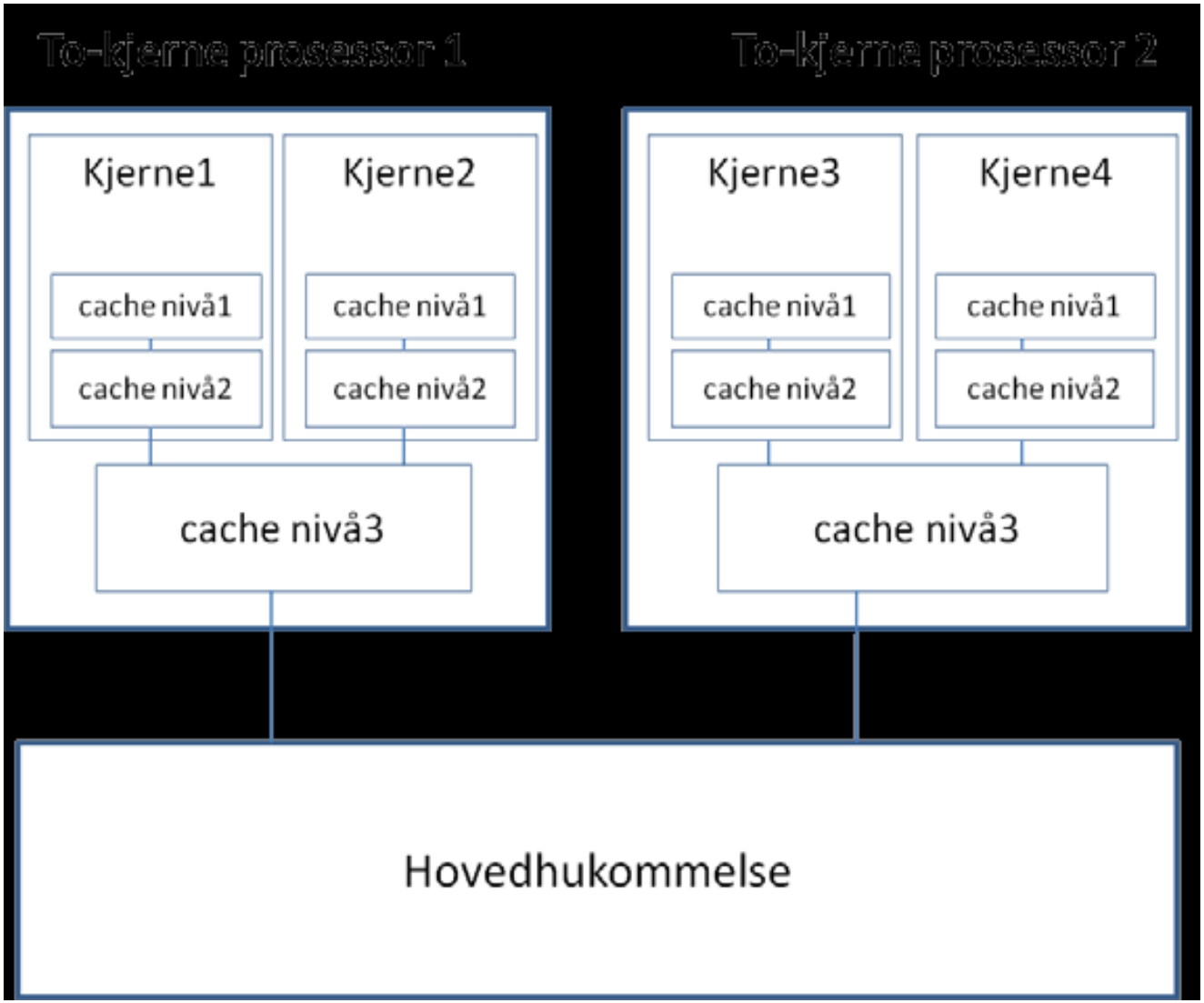
1. Maskinvare
2. Programmeringsspråk
3. Programmeringsabstraksjon.
4. Hvilke typer problem egner seg for parallelle løsninger?
5. Empiriske eller formelle metoder for parallelle beregninger

**INF2240**: Parallellprogrammering av ulike algoritmer med tråder på multikjerne CPU i Java – empirisk vurdert.

# Maskinvare og språk for parallelle beregninger og Ifi-kurs

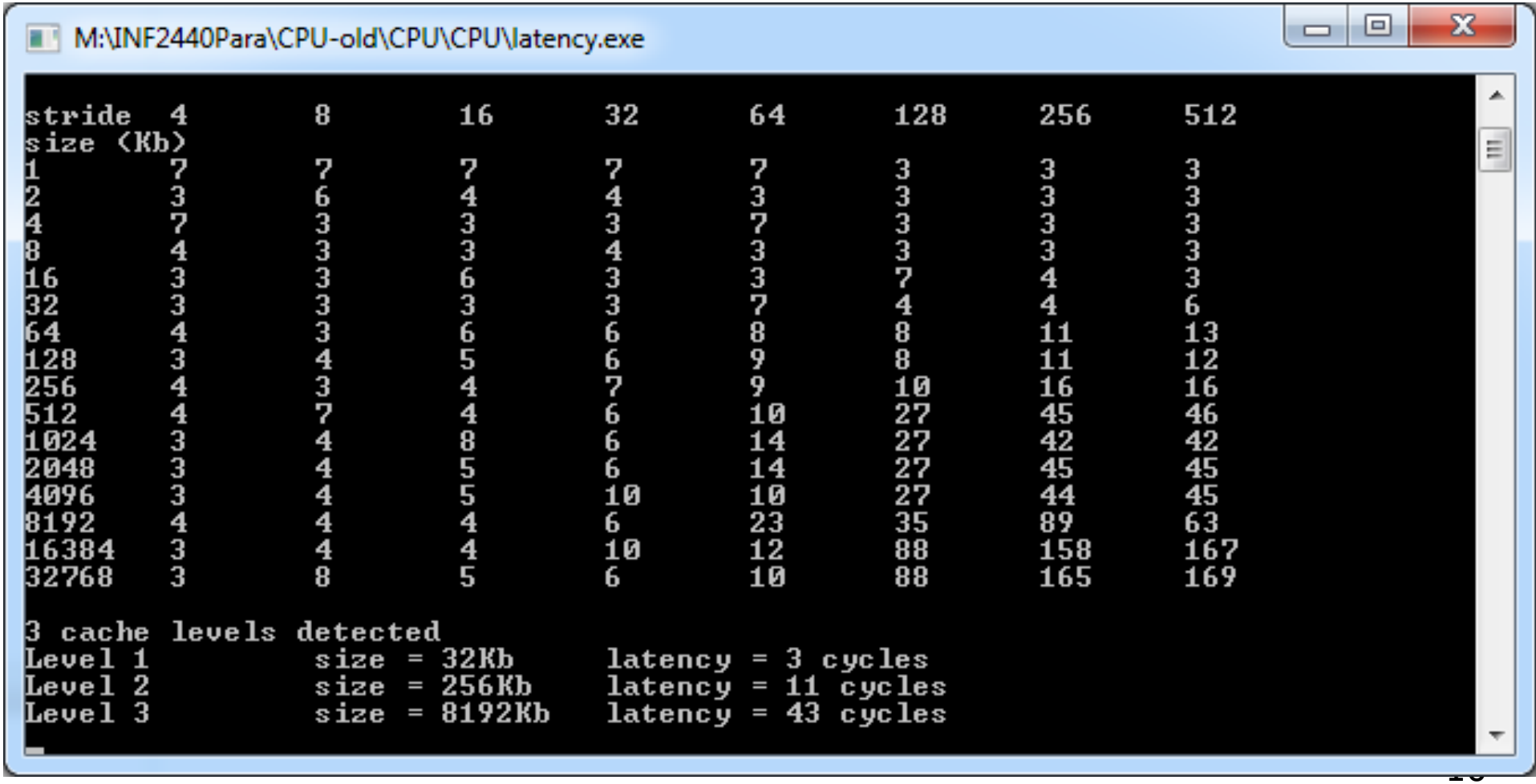
1. Klynger av datamaskiner - **INF3380**
   - jfr. Abelklyngen på USIT med ca 10 000 kjerner. 640 maskiner (noder), hver med 16 kjerner
   - C, C++, Fortran, MPI –de ulike programbitene i kjernene sender meldinger til hverandre
2. Grafikkkort GPU med 2000+ små-kjerner, **INF5063**
   - Eks Nvidia med flere tusen småkjerner (SIMD – maskin)
   - Cell prosessoren
3. Multikjerne CPU (2-100 kjerner per CPU) **INF2440**
   - AMD, Intel, ARM, Mobiltelefoner,..
   - De fleste programmeringsspråk: Java, C, C++,..
4. Mange maskiner løst koblet over internett. **INF5510**
   - Planet Lab
   - Emerald
5. Teoretiske modeller for beregningene **INF2140**, **INF4140**
   - PRAM modellen og formelle modeller (f.eks FSM)

# Maskin ca. 2010 med to dobbeltkjerne CPU-er



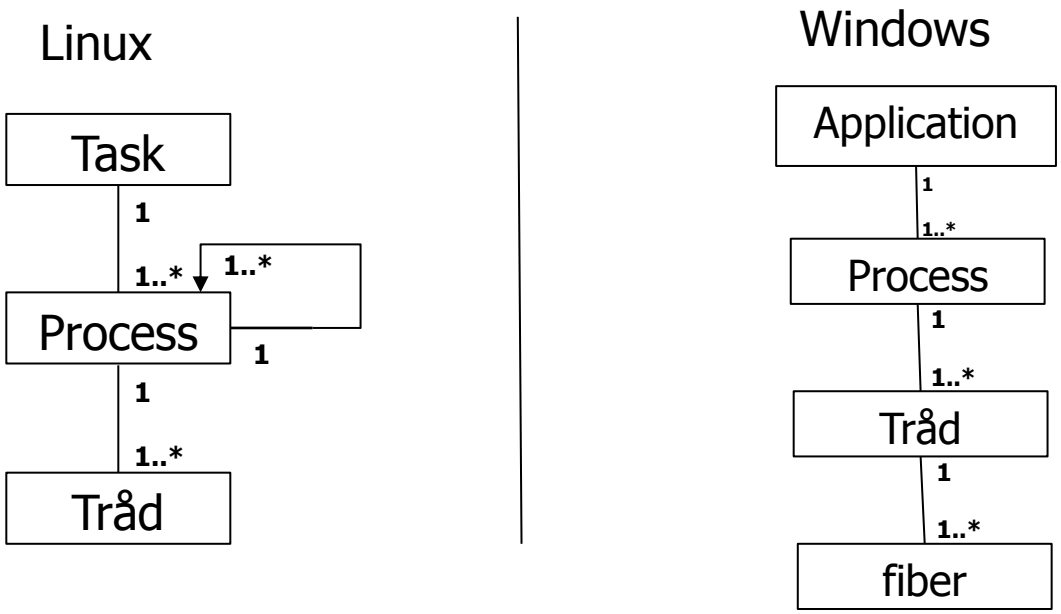
# Testing av forsinkelse i data-cachene

- Programmet : latency.exe (fra CPUZ)



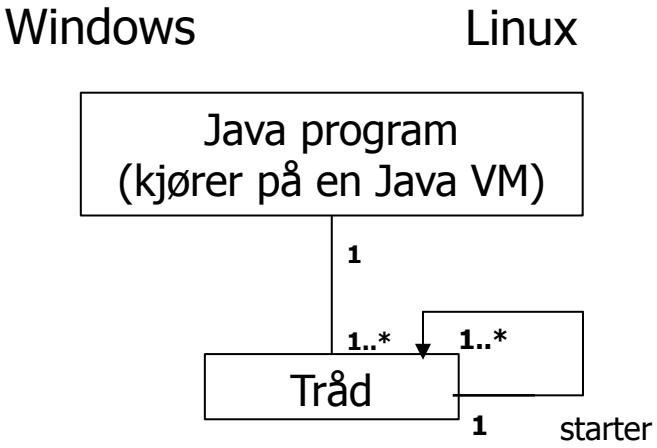
# Operativsystemet og tråder

- De ulike operativsystemene (Linux, Windows) har ulike begreper for det som kjøres; mange nivåer (egentlig flere enn det som vises her)




Heldigvis forenkler Java dette

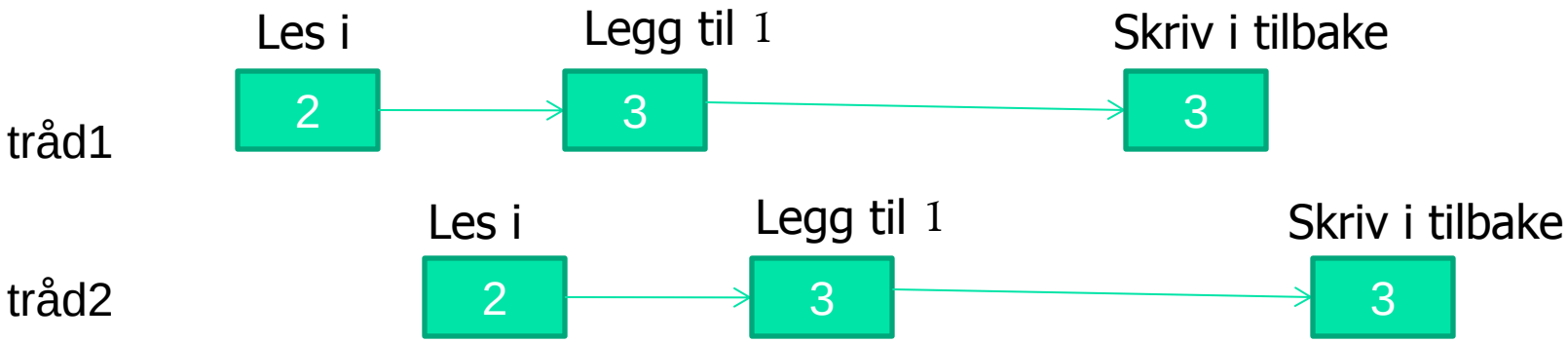
# Java forenkler dette ved å velge to nivåer




- **Alle trådene i et Java-program deler samme adresserom** (= samme plasser i hovedhukommelsen), og alle trådene kan lese og skrive i de variable (objektene) programmet har og ha adgang til samme kode (metodene i klassene).

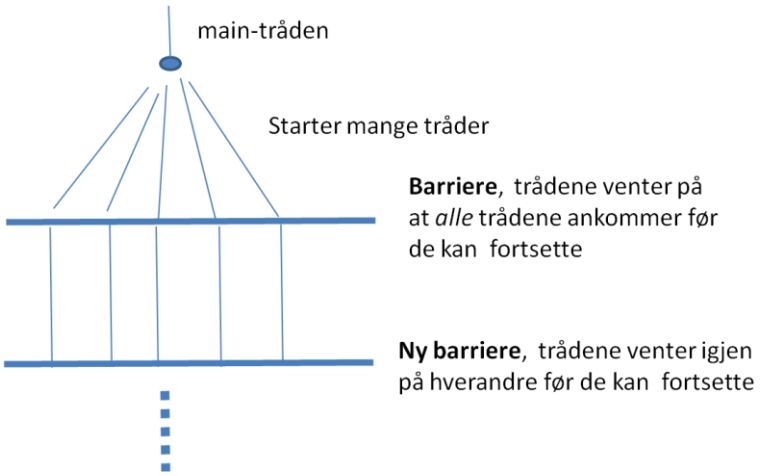
# Ett problem : operasjoner blandes ved samtidige oppdateringer (data-kappløp)

- Samtidig oppdatering -
  flere tråder sier gjentatte ganger: **i++ ;**
  - **i++** er 3 operasjoner: a) les i, b) legg til 1, c) skriv i tilbake
  - Anta i =2, og to tråder gjør i++
  - Vi kan få svaret 3 eller 4 (skulle fått 4!)
  - Dette skjer i praksis !




| Antall tråder n | 1 | 2 | 20 | 200 | 2000 |
|---|---|---|---|---|---|
| Svar 1.gang<br>2. gang | 100 000<br>100 000 | 200000<br>159234 | 1290279<br>1706068 | 16940111<br>16459210 | 170127199<br>164954894 |
| Tap 1.gang<br>2. gang | 0 %<br>0% | 0%<br>20,4% | 35,5%<br>14,6% | 15,3%<br>17,7% | 14,9%<br>17,5% |

# CyclicBarrier. Hva gjør den?




- Man lager først et objekt **`b`** av klassen CyclicBarrier med et tall **`ant`** til konstruktoren = det antall tråder den skal køe opp før alle trådene slippes fri 'samtidig'.
- Tråder (også main-tråden) som vil køe opp på en CyclicBarrier sier await() på den.
- De **`ant-1`** første trådene som sier await(), blir lagt i en kø.
- Når tråd nummer **`ant`** sier await() på **`b,`** blir alle trådene sluppet ut av køen 'samtidig' og fortsetter i sin kode.
- Det sykliske barriere objektet **`b`** er da med en gang klar til å være kø for nye **`ant`** tråder.

# Tråder i Java (lett revidert og kompilerbar )

- Er én programflyt, dvs. en serie med instruksjoner som oppfører seg som ett sekvensielt program – og kjører på en kjerne
- Det kan godt være (langt) flere tråder enn det er kjerner.
- En tråd er ofte implementert i form av en indre klasse i den klassen som løser en del av problemet vårt (da får alle trådene greit **felles data**):

```
import java.util.concurrent.*;
class Problem { int [] fellesData ; // dette er felles, delte data for alle trådene
                public static void main(String [] args) {
                  Problem p = new Problem();
                        p.utfoer();
                  }
                  void utfoer () { Thread t = new Thread(new Arbeider());
                      t.start();
                   }

              class Arbeider implements Runnable {
                  int i, lokalData; // dette er lokale data for hver tråd
                     public void run() {
                           // denne kalles når tråden er startet
                     }
               } // end indre klasse Arbeider
} // end class Problem
```

# Avslutning med join() - enklest

- Logikken er her at i den rutinen hvor alle trådene lages, legges de også inn i en array. Main-tråden legger seg til å vente på den tråden som den har peker ti skal terminere selv.
  Venter på alle trådene etter tur at de terminerer:

```
 // main –tråden i konstruktøren
Thread [] t = new Thread[n];
for (int i = 0; i < n; i++) {
   t[i] = new Thread (new Arbeider(..));
   t[i].start();
 }
  ……….
  // main vil vente her til trådene er ferdige
for(int i = 0; i < n; i++) {
       try{ t[i].join();
        }catch (Exception e){return;};
}     ……..
```

## Sammenligning av Amdahl og Gustafson + egne betraktninger

- Amdahl antar at oppgaven er fast av en gitt lengde(n)
- Gustafson antar at du med parallelle maskiner løser større problemer (større n) og da blir den sekvensielle delen mindre.
- Min betraktning:
  1. En algoritme består av noen sekvensielle deler og noen parallelliserbare deler.
  2. Hvis de sekvensielle delene har lavere orden – f.eks O(log n), men de parallelle har en større orden – eks O(n) så vil de parallelle delene bli en stadig større del av kjøretida hvis n øker (Gustafson)
  3. Hvis de parallelle og sekvensielle delene har samme orden, vil et større problem ha samme sekvensielle andel som et mindre problem (Amdahl).
  4. I tillegg kommer alltid et fast overhead på å starte k tråder (1-4 millisek.)

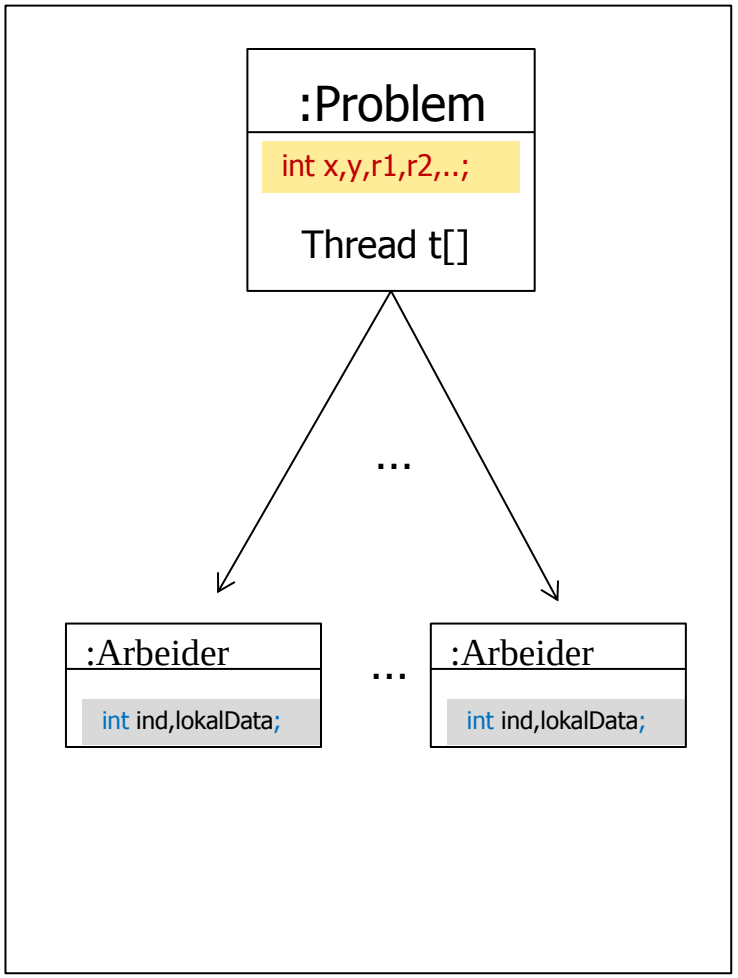
  Algoritmer vi skal jobbe med er mer av type 2 (Gustafson) enn type 3(Amdahl) men vi har alltid overhead, så små problemer løses lettest sekvensielt.

Konklusjon: For store problemer bør vi ha håpet om å skalere nær lineært med antall kjerner hvis ikke vi f.eks får kø og forsinkelser når alle kjernene skal lese/skrive i lageret.

# En Java-modell for parallelle programmer

```
import java.util.concurrent.*;
class Problem { int x,y,r1,r2,..; //  felles data
      public static void main(String [] args) {
            Problem p = new Problem();
            p.utfoer(12);
      }
      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, lokaleData; // lokale data
         Arbeider (int in) {ind = in;)
            public void run(int ind) {
                  // kalles når tråden er startet
            } // end run
       } // end indre klasse Arbeider
} // end class Problem
```



# Viktigste regel om lesing og skriving på felles data.

- Før (og etter) synkronisering på felles synkroniseringsvariabel gjelder :
  A. Hvis ingen tråder skriver på en felles variabel, kan alle tråder lese denne.
  B. To tråder må aldri skrive samtidig på en felles variabel (eks. i++)
  C. Hvis derimot en tråd skriver på en variabel må bare den tråden lese denne variabelen før synkronisering – ingen andre tråder må lese den før synkronisering.

Muligens ikke helt tidsoptimalt, men enkel å følge – gjør det mulig/mye enklere å skrive parallelle programmer.

Regel C er i hovedsak 'problemet' i Java Memory Model

Vi har sett 2 tilfeller hvor vi bevisst bryter regel C – Når?

# To typer av felles variable

- De som er en sum eller annen beregning og hvor det bare er **ett riktig, summert tall** som vi skal ha på denne plassen til sist.
  - En slik variabel kalle en akkumulator.
  - Da må vi følge regel C – bare én skriver og ingen andre leser.
- Av og til er det mange tall som er riktige som denne felles variabelen. Vi har sett to slike eksempler:
  - Avskjærings-verdien i faktoriseringa (der hvor vi delte ned øvre grense med nye faktorer hver tråd fant i et tall).
  - I Goldbach-summen m = p1+p2 kan flere tråder samtidig skrive ned hver sin p1 i samme plass i p1save[i]. Og siden p2 og m utledes fra p1 og i, er det likegyldig hvilken p1 vi velger.

# 3) Hva er en memory-modell og hvorfor trenger vi en!

- Vi har hittil sett at med flere tråder så:
  - Instruksjoner kan bli utsatt og byttet om
    - av CPU, kompilator, cache-systemet
  - Ulike tråder kan samtidig se ulike verdier på felles variabel
  - Programmet du skrev er optimalisert og har liten direkte likhet med det som eksekverer
- Vi må likevel ha en viss orden på (visse steder i eksekveringen):
  - Rekkefølgen av instruksjonene
  - Synlighet - når ser én tråd det en annen har skrevet ?
    - Maskinvare og hukommelse hvor alle-ser-det samme-alltid tar for lang tid
  - Atomære operasjoner
    - Når en operasjon først utføres, skal hele utføres uten at andre operasjoner blander seg inn og økelegger (som i++)
- Ulike maskinvare-fabrikanter kan ulike memory-modeller, men Intel/AMD modellen er rask og understøttes av Java.

## Effekten av synkronisering på samme synkroniseringsvariabel og The Java memory model (JMM)

Hva gjør vi hvis vi vil (i en tråd) lese hva en annen tråd har skrevet ?

- Har den andre skrevet enda og har det kommet ned i lageret?

Svar: Vi lar begge (alle) trådene **synkronisere** på samme synkroniseringsvariabel (en CyclicBarrier eller en Semaphore,…):

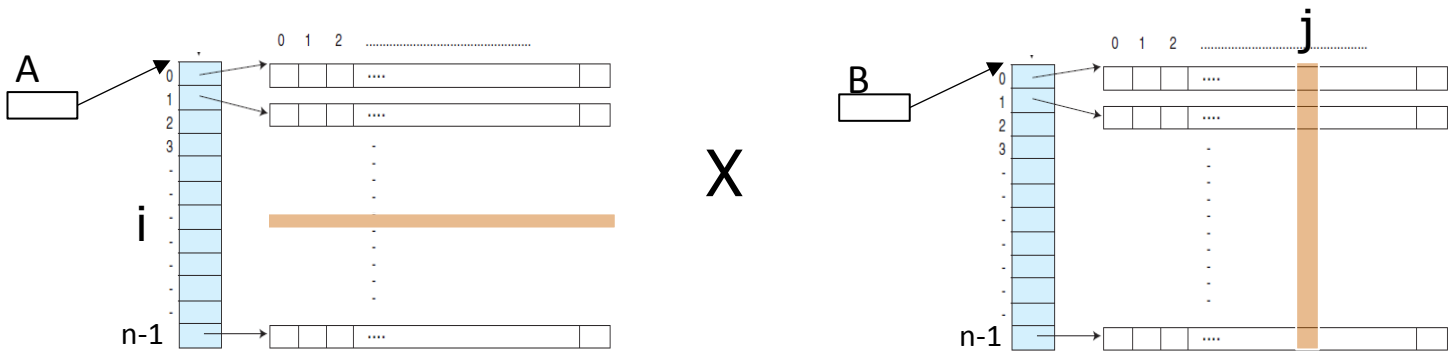
- Da skjer følgende før noen av trådene fortsetter:
  1. Alle felles variable blir skrevet ned fra cachene og ned i lageret.
  2. Alle operasjoner som er utsatt, blir utført før noen av trådene slipper gjennom synkroniseringen.
  3. Følgelig ser alle trådene de samme verdiene på alle felles variable når de fortsetter etter synkroniseringa!

# Effekten av cache ved matrisemultiplisering

- Matriser er todimensjonale arrayer
- Skal beregne C=AxB (A,B,C er matriser)

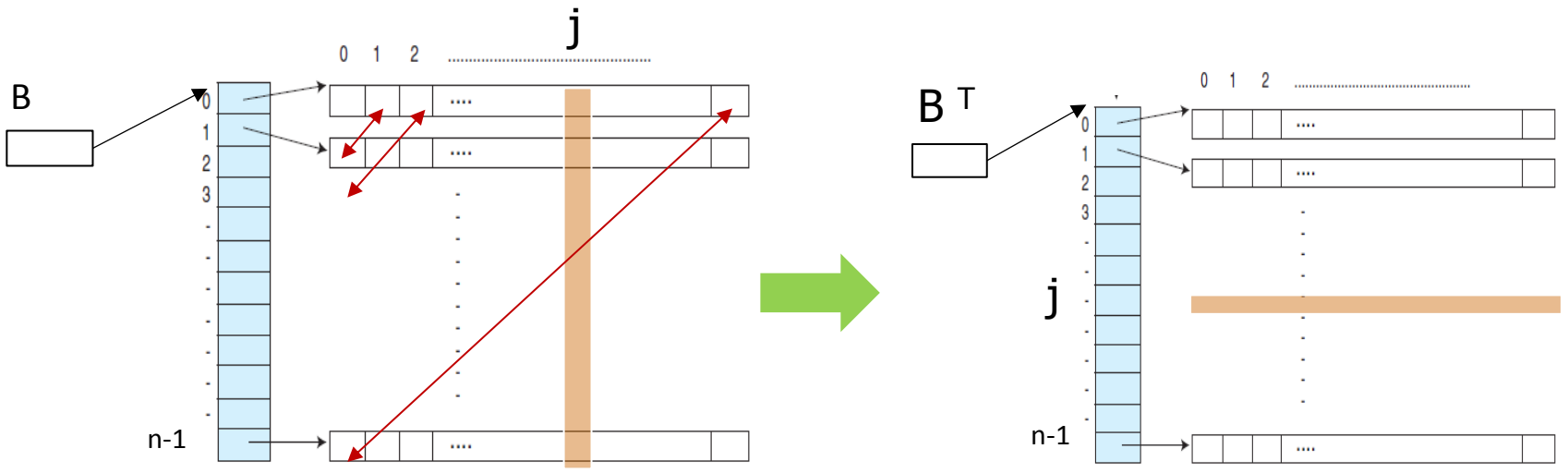
$$c[i][j] = \sum_{k=0}^{n-1} a[i][k] * b[k][j]$$

$C_{i,j} =$



# Idé – transponer B (=bytte om rader og kolonner)

- Bytt om elementene i B ($B_{i,j}$ byttes om med $B_{j,i}$ )
- Da blir kolonnene lagret radvis.

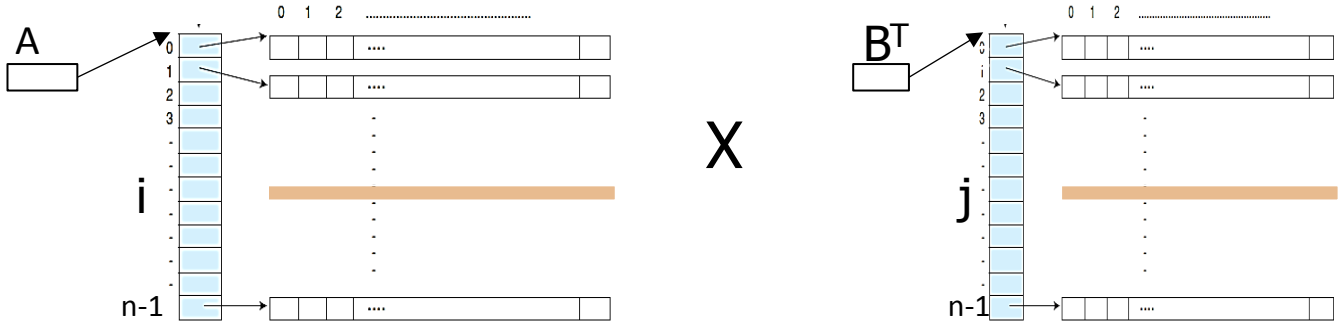



Begrunnelse: Det blir for mange cache-linjer (hver 64 byte) fra B i cachene når vi multipliserer etter lærebok-definisjonen

# Vi har da at litt ny formel for C

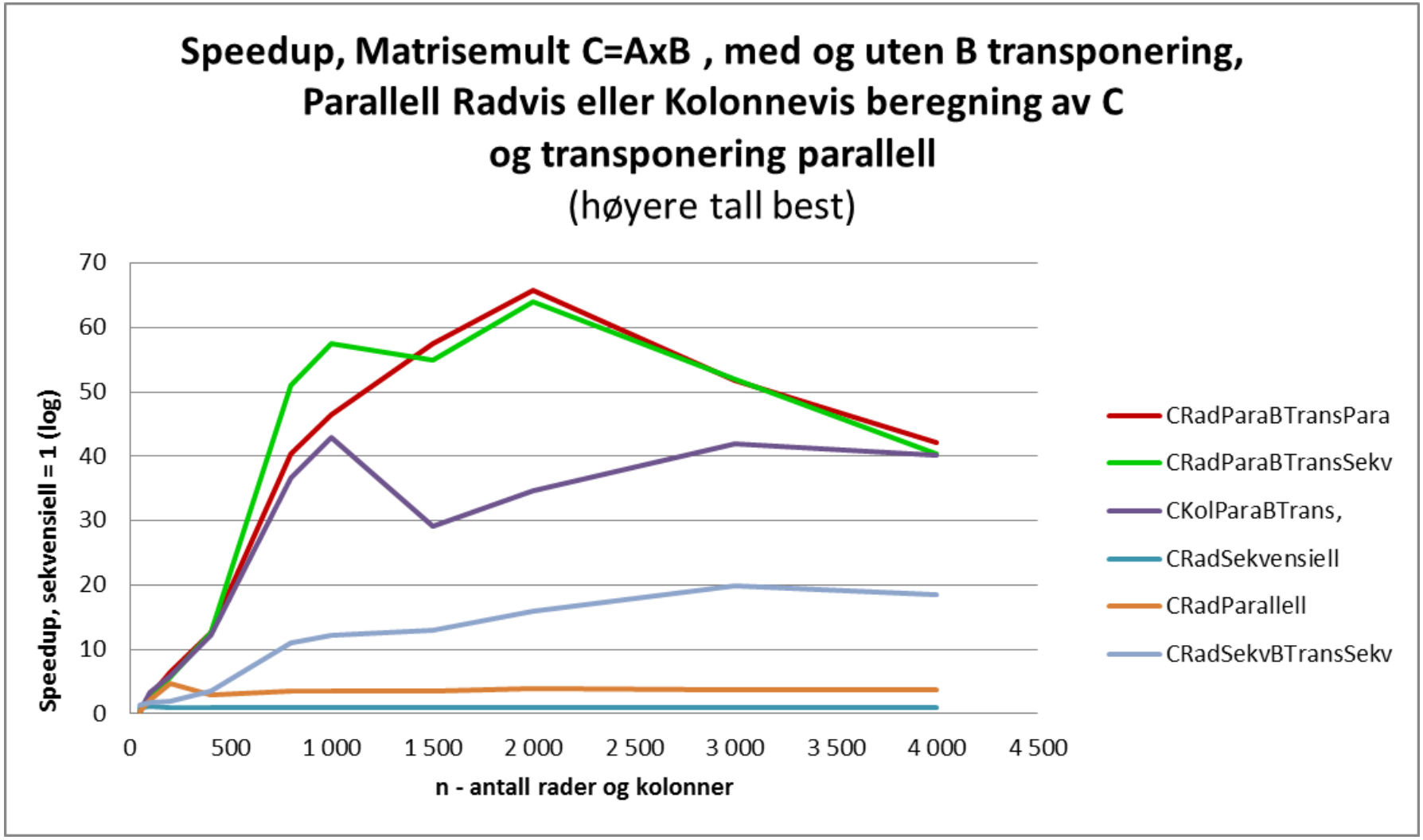
$$c[i][j] = \sum_{k=0}^{n-1} a[i][k] * b^T[j][k]$$

$C_{i,j} =$




Vi får multiplisert to rader med hverandre – går det fortere ?

# Kjøretider – speedup av ulik parallellisering: Radvis eller kolonnevis av transponering og multiplikasjon.



# Litt konklusjon om Matrisemultiplikasjon

- Parallelliserte C kolonnevis
  - Speedup: 3,5
- Så transponerte vi B
  - Speedup: ca 40
- Så beregnet vi C radvis i parallell
  - Speedup: opp til 64
- Så parallelliserte transponering av  B
  - Speedup: opp til 66

(Ganske liten effekt av å parallellisere transponeringen – hvorfor?)

# 3) Finnes det alternativer til synchronized metoder?

- 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, men speeddown likevel.

# Oppummering

| 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 | 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.

# 4) Vranglås: 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 'fortsett' (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();
```

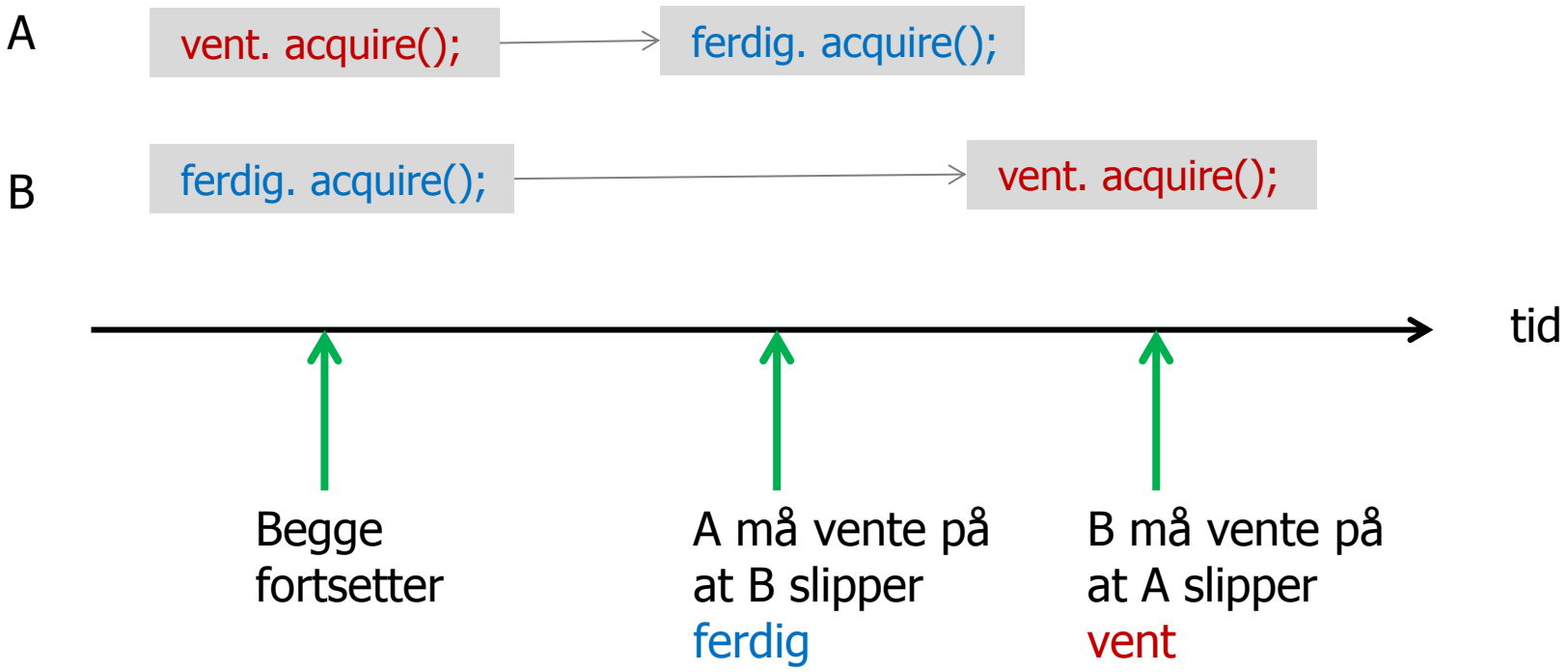
```
B sier:
try{
     ferdig. acquire();
     vent. acquire();
} catch(Exception e)
{return;}

…gjør noe….

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

# 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-primitiver 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-primitiver 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.

# 1) 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 med en del til hver tråd, og lar 'hele' koden virke på hver av disse data-delene.
  - 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
  - Primtalls-faktorisering av store tall N for kodebrekking:
    - N= $p_1 * p_2$ - vi lar hver tråd finne faktorer i sin del av tallinja.
- Også mulig å dele opp koden:
  - Alternativ Oblig3 i INF1000: Beregning av Pi (3,1415..) med 17 000 sifre med tre ArcTan-rekker

# Å parallellisere 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 CyclicBarrier-synkronisering mellom hver av disse delene
  + en synkronisert avslutning av hele algoritmen(join(), ..).
- Eks:
  - finnMax – hadde ett slikt steg: for (int i = 0 …n-1)  -løkke
  - MatriseMult hadde ett slikt steg med trippel-løkke
  - Radix hadde 4 slike steg:
    - en enkelt løkke i radix2
    - tre steg i radixSort : a),b) og c)  - alle enkeltløkker (gjenntatt 2 ganger)
    - Hver av disse må få hver sin parallellisering.

# 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 alt for 'mye' tid)

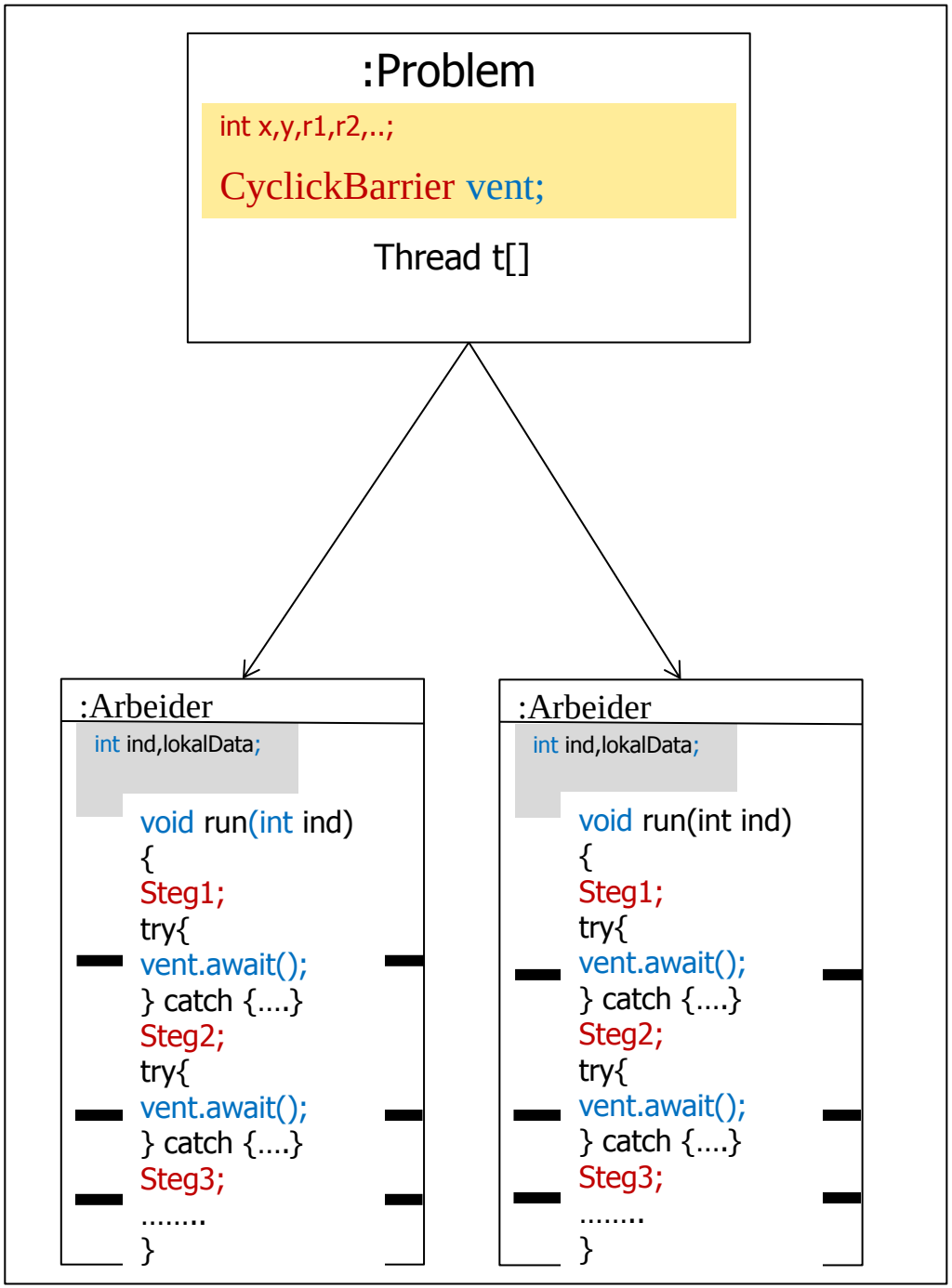
# Viktig: Kopiere deler av felles data til lokale data

- Kan vi kopiere aktuelle deler av felles data til hver tråd (ha en lokal en kopi av disse i lokale data i hver tråd)?
- Hver tråd oppdaterer så sin kopi – og etter en synkronisering kan disse lokale kopiene 'summeres/syes sammen' slik at vi får riktig felles resultat i de felles data?
  - Da vil **en** for-løkke bli til **to** steg:
    - Steg 1: Lag kopi de felles data og kjør løkka på 'sin' lokale del av data.
    - Synkroniser på en CyclicBarrier
    - Steg 2: De lokale data samles/adderes til slik data blir som i den sekvensielle algoritmen (hvis neste steg kan bruke disse lokale kopiene, beholdes de)
    - Disse sammen-satte data er nå igjen felles, delte data.
- Eks: FinnMaks haddde en **int max;** som felles data. Den kunne lett kopieres til hver tråd som **int mx**, og felles resultat ble beregnet som de største av disse **mx**–ene fra alle trådene.

# Husk at vi skal lage effektive algoritmer !

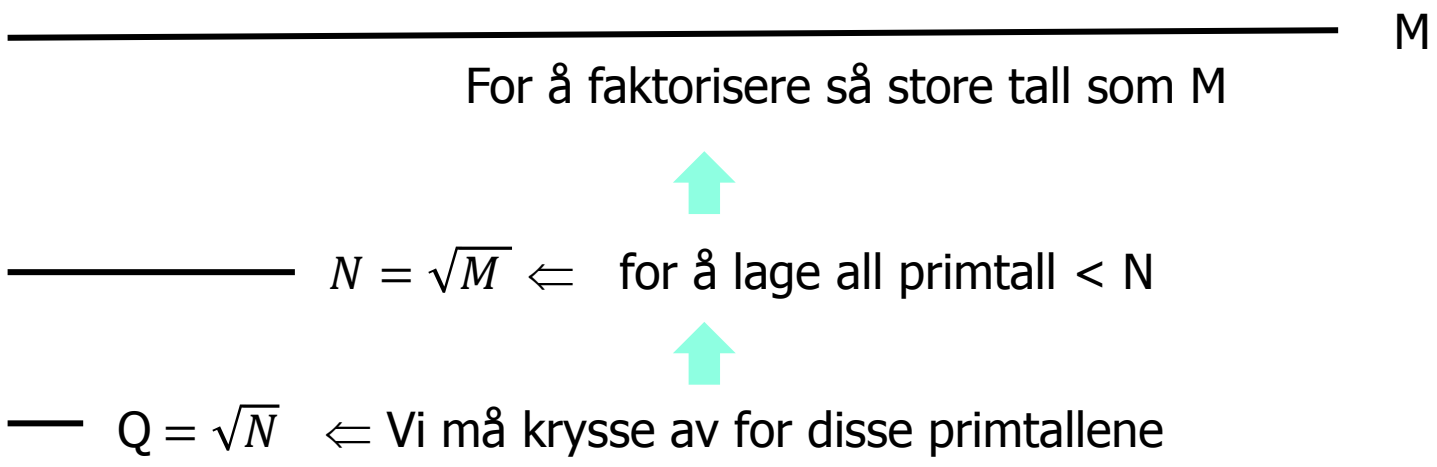
- Vi må ikke synkronisere for mange ganger !
  - Fordi hver synkronisering tar 'lang' tid (skriving av cachene til lageret, utføre utsatte operasjoner,..)
- Vi kan **ikke** synkronisere hver gang vi i den sekvensielle algoritmen bruker (leser/skriver) felles data.
- Regel for synkronisering:
  - Antall synkroniseringer på felles data må være av en lavere orden enn selve algoritmen.
    - Eks:
      - O($n \log n$), O(n) eller O($\log n$) synkroniseringer (under tvil: O($n^2$) ) hvis algoritmen er O($n^3$)
      - O($\log n$) hvis algoritmen er O(n) eller høyere.
    - Aller helst bare et fast antall synkroniseringer – uavhengig av n – f.eks antallTråder + antall faser

Synkronisering av algoritme i flere steg. Med venting på en felles CyclicBarrier mellom hvert steg:



# Hvordan faktorisere et stort tall (long)

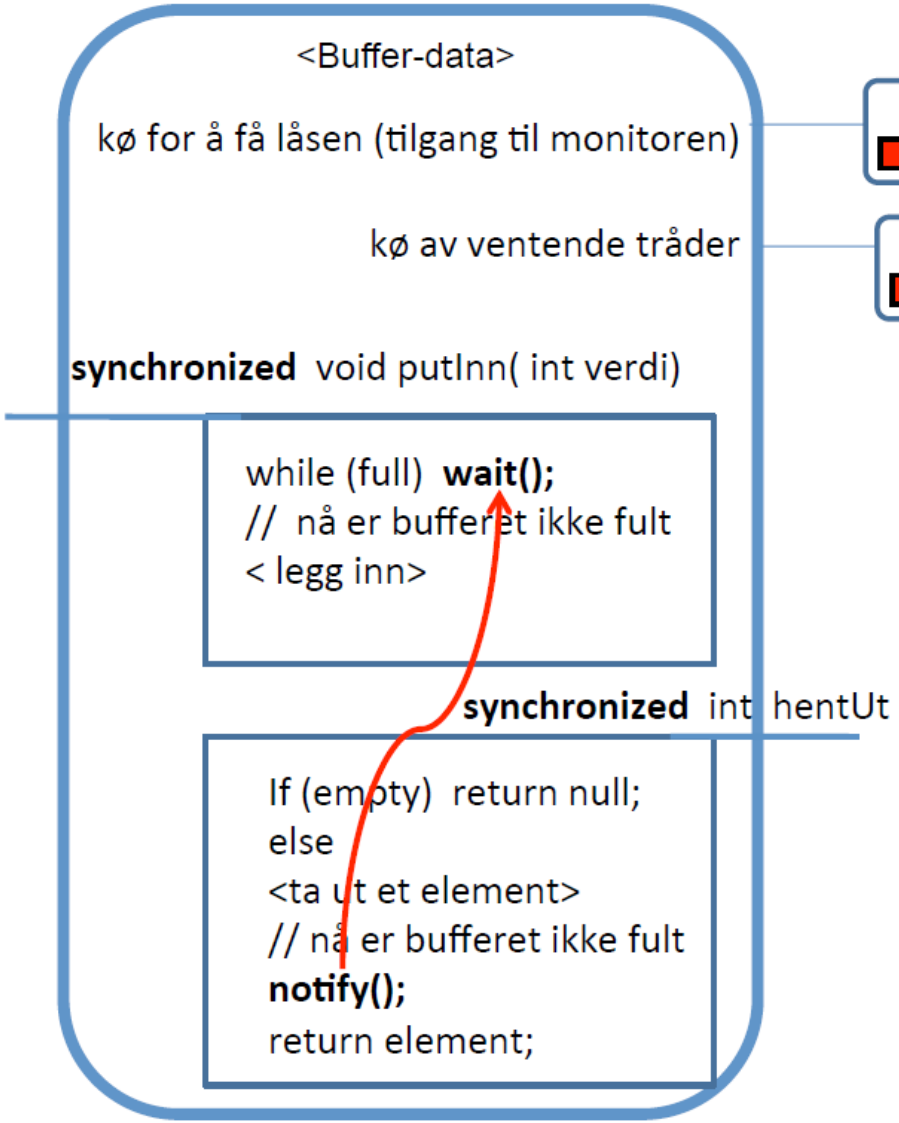
- Anta at vi har en long M:
- Vi kan faktorisere den hvis vi vet alle primtall < N=$\sqrt{M}$
- For å finne alle primtall < N, må vi krysse av for alle primtall Q <$\sqrt{N} = \sqrt{\sqrt{M}}$



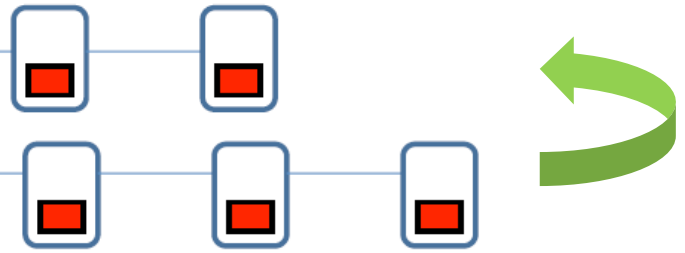
# Flere (synkroniserings-) metoder i klassen Thread.

- **getName()** Gir navnet på tråden (default: Thread-0, Thread-1,..)
- **join():** Du venter på at en tråd terminerer hvis du kaller på dens join() metode.
- **sleep(t):** Den nå kjørende tråden sover i minst 't' millisek.
- **yield():** Den nå kjørende tråden pauser midlertidig og overlater til en annen tråde å kjøre på den kjernen som den første tråden ble kjørt på..
- **notify():** (arvet fra klassen Object, som alle er subklasse av). Den vekker opp **en** av trådene som venter på låsen i inneværende objekt. Denne prøver da en gang til å få det den ventet på.
- **notifyAll():** (arvet fra klassen Object). Den vekker opp **alle de** trådene som venter på låsen i inneværende objekt. De prøver da alle en gang til å få det de ventet på.
- **wait():** (arvet fra klassen Object). Får nåværende tråd til å vente til den enten blir vekket med notify() eller notifyAll() for dette objektet.

## To køer i en basal Java monitor:




En kø av ventende tråder på hele monitoren



notify()

En kø av ventende tråder på ”wait”-instruksjoner (wait-set).

Startes av notify () og/eller notifyAll()

Legges da i den andre køen

(først ? (Nei, ingen garanti))

Derfor er det nødvendig med ”while …”

# En effektiv Threadpool ?

- Vi har med modell2-koden selv startet en samling av k tråder som- venter til vi vil kjøre problemet (evt. flere ganger):
  - enten med samme n for å få bedre tider (median)
  - eller for en ny n .
- Ideen om å lage en samling av tråder som 'er klar for' å løse neste oppgave har vi også i Java-biblioteket.
- Vi skal her se på: Executors.newFixedThreadPool i java.util.concurrent.

# Grunnidéene i Executors.newFixedThreadPool

- Du starter opp et fast antall tråder
- Hvis du under kjøring trenger flere tråder enn de startet må du vente til en av dem er ferdig og da er ledig
- For hver tråd som gjør noe er det tilknyttet et objekt av klassen **Future**:
  - Den sier deg om tråden din er ferdig
  - Du kan legge deg og vente på et eller alle Future-objekter
    - som join() med 'vanlige' tråder
  - Future-objektet bringer også med seg svaret fra tråd når den er ferdig, hvis den har noen returverdi
- En tråd som har terminert kan straks brukes på nytt fordi main-tråden venter ikke på tråden, men på tilhørende Future

# For å administrere en slik mengde (pool) av tråder

- Må man først lage en  pool (med tilhørende Vektor av Futures):

```
class MinTraadpool{
MinTraadpool pt = new MinTraadpool ();
int antTraader = Runtime.getRuntime().availableProcessors();
ExecutorService pool =  Executors.newFixedThreadPool(pt.antTraader);
List <Future> futures = new Vector <Future>();
```

- Hvordan lage trådene og slippe dem ned i poolen (svømmebasenget):

```
for (int j =0; j < antTraader; j++) {
        Thread QParExec = new Thread(new FindExec(j));
        futures.add(pool.submit(QParExec)); // submit starts the Thread
}
```

# 1 &2)Test på effektivitet – tre implementasjoner av parallell FindMax i int [] a ;  speedup:

- a) Med ExecutorService og FixedThreadPool
  - a1)Med like mange tråder som kjerner:
  - a2) Med 2x tråder som kjerner
- b) Med modell2–kode med 2 CyclicBarrier for start&vent

| n | a1)Pool 1x | a2) Pool 2x | b) Barrier |
|---|---|---|---|
| 1000 | 0.016 | 0.037 | 0,049 |
| 10000 | 0.249 | 0,235 | 0,011 |
| 100000 | 0.256 | 0,302 | 0,105 |
| 1000000 | 0.469 | 0,399 | 0,453 |
| 10000000 | 1.353 | 1,344 | 1,339 |
| 100000000 | 1.615 | 1,6662 | 1,972 |
| | | | |

- Konklusjon: Om lag like raske !?

# Oppsummering om kjøretider:

- Metodekall tar svært liten tid: **2-5** μs og kan også optimaliseres bort (og gis speedup >1)
- Å lage et objekt av en klasse (og kalle en metode i det) tar liten tid: ca.**785** μs
- Å lage en tråd og starte den opp tar en del tid: ca.**1500** μs, men lite ca. **180** μs for de neste trådene (med start() )
- Å lage en en tråd-samling og legge tråder (og Futures) opp i den tar ca. **7000** μs for første tråd og ca. **210** μs for neste tråd.
- Å bruke trådsikre heltall (AtomicInteger og AtomicIntegerArray) mot vanlige int og int[] tar vanligvis **10-20 x** så lang tid , men for mye bruk (> $10^6$ ) tar det bare ca. 2x tid (det siste er sannynligvis en cache-effekt)

**Sequential Quicksort**

| Number of Elements | Sorting Time |
|---|---|
| 1 000 | 104 μs |
| 2 500 | 288 μs |
| 5 000 | 607 μs |
| 7 500 | 973 μs |
| 10 000 | 1318 μs |
| 25 000 | 3632 μs |
| 50 000 | 7980 μs |
| 75 000 | 12214 μs |
| 100 000 | 16980 μs |

Figure 6.2: Quicksort Sorting Times

En parallell løsning må alltid sammenlignes med hvor mye vi kunne ha sortert sekvensielt på denne tida!

# Konklusjon om å parallellisere rekursjon

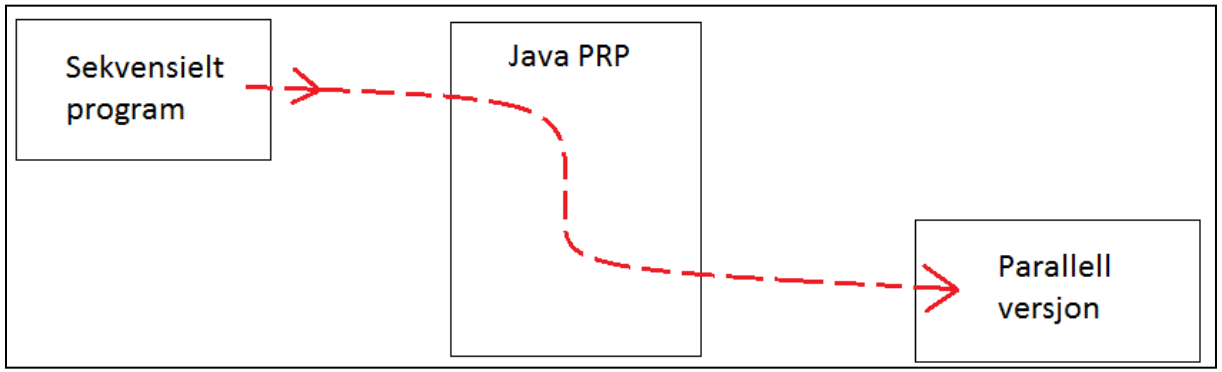
- Antall tråder må begrenses !
- I toppen av treet brukes tråder (til vi ikke har flere og kanskje litt mer)
- I resten av treet bruker vi sekvensiell løsning i hver tråd!
- Viktig også å kutte av nedre del av treet (her med insertSort) som å redusere treets størrelse drastisk (i antall noder)
- Vi har for n = 100 00 gått fra:

| n | sekv.tid(ms) | para.tid(ms) | Speedup | |
|---|---|---|---|---|
| 100000 | 34.813 | 41310.276 | 0.0008 | Ren trådbasert |
| 100000 | 8.118 | 823.432 | 0.0099 | Med insertSort |
| 100000 | 7.682 | 5.198 | 1.4777 | + Avkutting i toppen |

- Speedup > 1 og ca. 10 000x fortere enn ren oversettelse at et rekursivt kall = new Thread(….) er bare å ha noen tråder i toppen av treet og så igjen vanlig rekursjon.

# I Uke 9 så vi på å overføre Rekursjon til tråder - fra Peter Eidsviks (kommende) masteroppgave

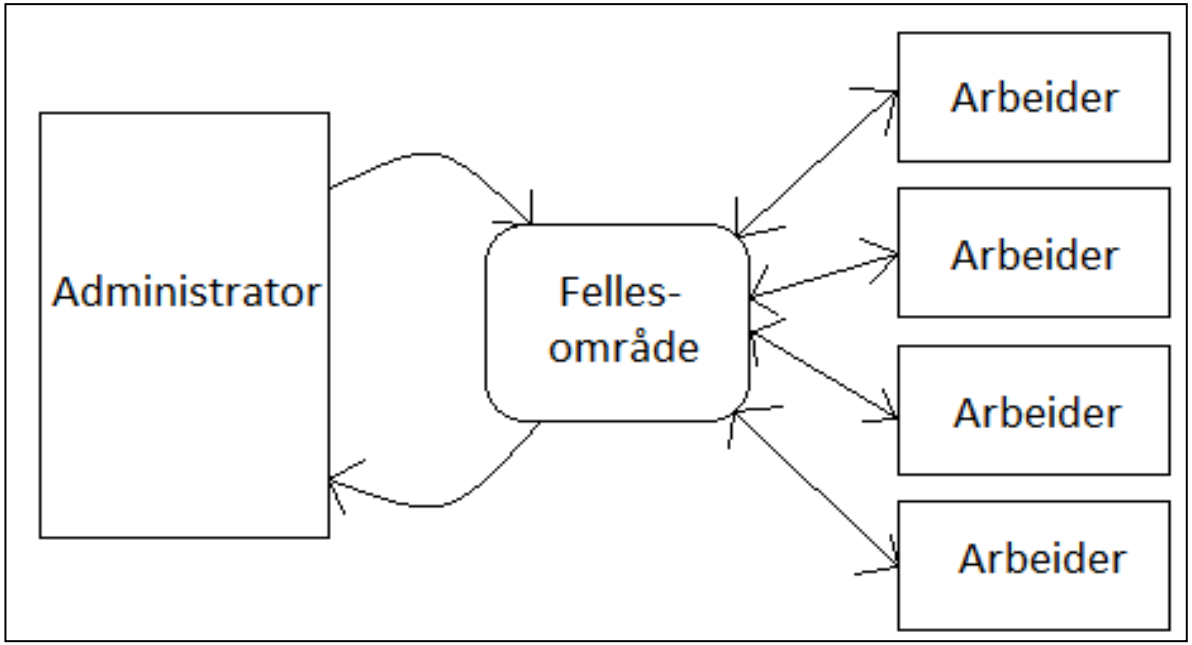
- Vi skal nå automatisere det
- Vi lager en preprosessor: javaPRP
  - dvs. et Java-program som leser et annet Java-program og omformer det til et annet, gyldig Java-program (som er det parallelliserte programmet med tråder)




- For at JavaPRP skal kunne gjøre dette, må vi legge inn visse kommentarer i koden:
  - Hvor er den rekursive metoden
  - Hvor er de rekursive kallene
- Bare rekursive metoder med to eller flere kall, kan parallelliseres .

# Hvordan gjøres dette? Administrator – arbeider modell

- Oppgaver legges ut i et fellesområde
- Arbeiderne tar oppgaver og legger svar tilbake i fellesområdet



# II) Debugging – feilfjerning parallelt

- Antar at vi har et program som lar seg starte opp
- Felles problem i sekvensiell og parallell kode:
- Er det en feil her?
  - Terminerer programmet
  - Gale resultater
  - Samme feil hver gang ?
    - Et sekvensielt program kan gjøres deterministisk
      - (eks. Random r = new Random(123) vil produsere samme tall-rekke ved neste kjøring)
    - Parallelle programmer kan ikke gjøres deterministiske
      - 'Never same result twice'
  - Er feil avhengig av størrelsen på problemet som løses av programmet , av n ?

**Råd1**: Finn først det 'minste' eksempelet som feiler

- **Råd2: Ikke** debug **ved å gå linje for linje gjennom koden** !
- **Råd3**: Bruk binær feilsøking:
  - Plasser en System.out.println(..) '**midt i koden**' og sjekk de data som der skrives ut – virker de riktige?
    - Hvis ja: feilen nedenfor, nei: feilen er ovenfor
  - Her: Siden problemet feilet med n=200, så:
    - Skrev ut alle primtallene < n (200)
  - Vi hadde to algoritmer (Eratosthenes eller faktoriseringen) – var det den første eller den andre ?
- **Råd4:** Litt analyse (papir og blyant):
- **Råd5**: Forstå feilen, se på koden
- **Råd6:** Legg System.out.println(s) inn i en:
  synchronized void println(String s) i den ytre klassen
  - Da vil utskrifter ikke blandes.
- **Råd7**: Hvis ingen ting skjer når vi starter programmet:
  - Lag en 5-6 setninger av typen : print(«A»); print(«B»);..og plasser de jevnt over rundt i programmet (i toppen av løkker)
  - Da ser vi hvor langt programmet kom før det hang seg.

# Hva så vi på i Uke13

I) Om vindus- (GUI) programmering i Java

- Tråder ! Uten at vi vet om det !

II) Hvordan pakke inn en parallell algoritme slik at andre kan bruke den.

- Kan du gi din Oblig4 til en venn eller en ny jobb som ikke har peiling på parallellitet og si at her er en metode som sorterer 3-10x fortere enn Arrays.sort() ?
- Nødvendige endringer til algoritmene, effektivitet !
- Fordelinger av tall vi skal sortere.
- Fornuftig avslutning av programmet ditt (hva med trådene)
- Brukervennlig innpakking !
- Dokumentasjon

III) Litt om løsning på prog1 og prog2 – finn én sum for alle n < 2 mrd. og alle (antall) Goldbach-summer for alle partall < 1 . mill. sekvensielt og parallelt.

.

```
D:\INF2440Para\Static-ABC>java ClassC 8000
1) s=32087203  paa:651339 nanosek
2) s=31544151  paa:156291 nanosek
3) s=32044431  paa:158986 nanosek
4) s=32121418  paa:271006 nanosek
5) s=32022808  paa:  46194 nanosek
6) s=32046616  paa:   6159 nanosek
7) s=32040178  paa:   6544 nanosek
8) s=32010535  paa:   7699 nanosek
9) s=32175599  paa:   7699 nanosek
10) s=32414924  paa:   6544 nanosek
11) s=32087937  paa:   3465 nanosek
12) s=32184513  paa:   3464 nanosek
13) s=32247691  paa:   5005 nanosek
14) s=32041925  paa:   4620 nanosek
15) s=32015535  paa:   5004 nanosek
16) s=32083860  paa:   3849 nanosek
17) s=31979593  paa:   5390 nanosek
18) s=31661029  paa:   4620 nanosek
19) s=31688502  paa:   5004 nanosek
20) s=31983432  paa:   5005 nanosek
21) s=32031745  paa:   3464 nanosek
22) s=31954441  paa:   5004 nanosek
23) s=32008609  paa:   5004 nanosek
24) s=32380944  paa:   5004 nanosek
25) s=31992932  paa:   4235 nanosek
26) s=31929029  paa:   3464 nanosek
27) s=32065607  paa:   3079 nanosek
```

8000 tall summert 27 ganger – stadig kortere kjøretid pga. JIT kompilering til maskinkode og hyper-optimalisering i flere steg.
Speedup : 211,5

# Ett problem som vel ikke lar seg parallellisere.

- Innstikksortering – kan den parallelliseres ?? Neppe:
  - Praktisk fordi den brukes bare til å sortere arrayer < 100 lang
  - men også teoretisk fordi det er vanskelig å se logikken i å delsortere a[] i k deler og så,…?

```
/** sort double-array a[left..right] */
void insertSort( double a[], int left, int right) {
        int i,k;
        double t;
        for (k = left+1 ; k <= right; k++) {
                t = a[k] ;
                i = k;
                 while ( a[i-1]  > t ) {
                        a[i] = a[i-1];
                        if (--i == left) break;
                  }
                  a[i] = t;
          } // end k
} // end insertSort
```

# Oppsummering:
# Parallellisering av en algoritme med k tråder:

- Finn hvilke steg/faser den sekvensielle algoritmen består av (løkkene og/eller rekursjon)
- For hvert steg:
  - Finn de delte data – de som flere tråder skal skrive på samtidig .
  - Kopiér heller de delte data til hver tråd. Det er mye raskere enn å synkronisere hver aksess til delte data.
  - Synkroniser (CyclickBarrier) slik at alle blir 'samtidig' ferdig med sin oppdatering av sin kopi av delte data.
  - Samkjør så hver tråds kopi av til et felles svar (slik det blir i det sekvensielle tilfellet) – dette er muligens en sekvensiell operasjon
- Overhold reglene for samtidig lesing/skriving
- Husk å synkronisering til sist slik at main-tråden vet at alle trådene har blitt ferdige – eller bruk join().

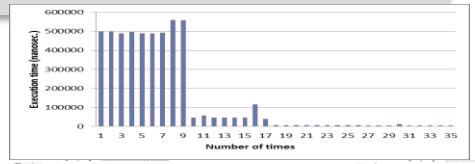
# Teaching B.S. students to parallelize algorithms with speedup > 1

on a shared memory multicore machine using Java threads.

"*No reason to write complicated parallel algorithms if they are not any faster in measured execution time*"

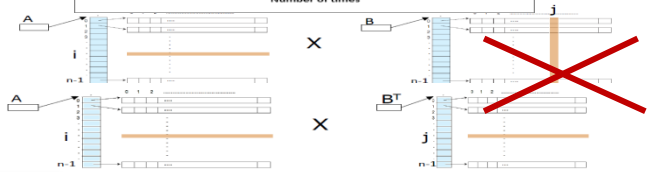
**1. JIT-compilation** – speedup 170.

ex: Calling a method that does sum of 8000 integers, 35 times – measured execution time from interpreted byte code to hyper optimized machine code (from 496 945 to 3 150 nanosec.).



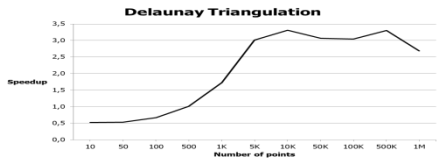

**2. Three level cache** – speedup 20.

ex : Matrix-multiplication – Since Java stores Matrixes by row, text-book definition C=AxB (=∑row*column) is very slow versus: first transpose B and then do:
C = Ax'B$^{T}$. (=∑row*row)



**3. Parallelization** – speedup 3-4 (on 4(8) cores).

ex: Any algorithm on top on JIT and cache speedup. Speedup example 1&2 sum : Speedup up to 3.3. Matrix mult: speedup 66 (=20*3.3) compared with text-book def. Delaunay Triangulation: see graph.



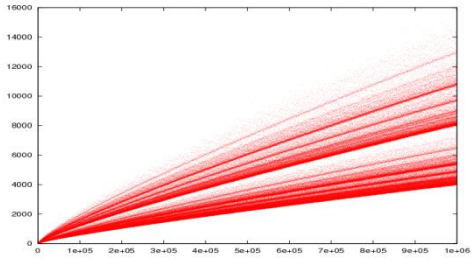

**Compulsary assignments:** (sequential and parallel version + speedup):

a) **Find the 50 largest elements** among 100, 1000,..,100 mill. integers.

b) **Factorize** any integer $m < 4.2*10^{18}$ in less than 1 sec.

c) **Radix sorting** parallelized (sequential version given).

A) Number of Goldbach sums in $m$: $4 < m < 1$ mill.

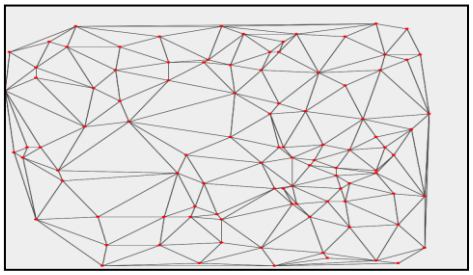


Other major algorithms presented to the students through exercisis:

**A) Goldbachs conjecture**
(any even number $m$ is the sum of two primes)

**B) Delaunay triangulation**
(convex hull + triangulation – two algorithms)

B) Delaunay triangulation of 100 random points



Arne Maus (arnem@ifi.uio.no) and Stein Gjessing (steing@ifi.uio.no)