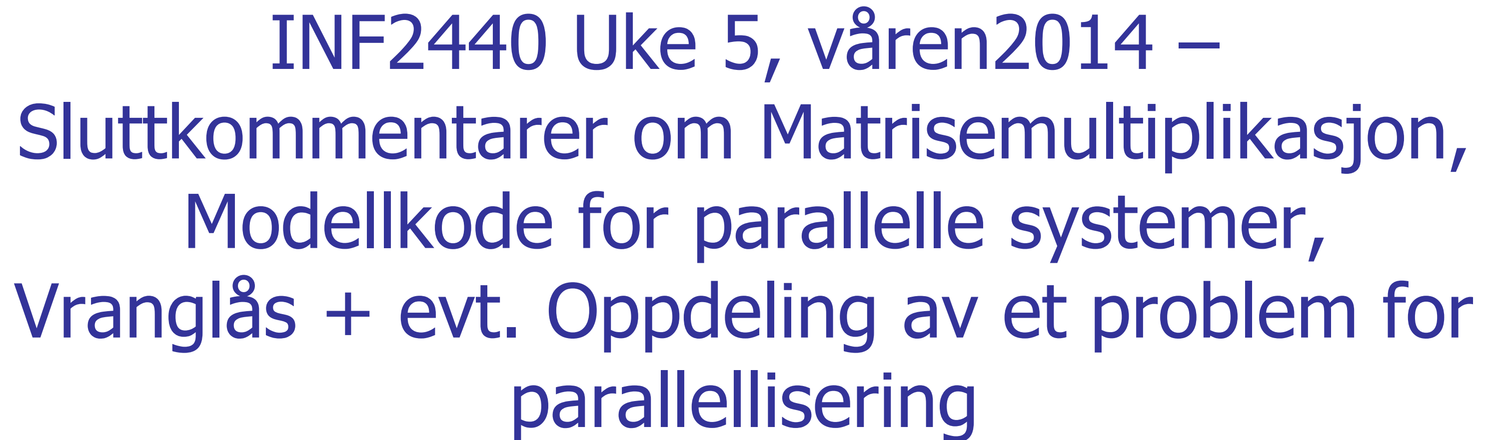


INF2440 Uke 5, våren2014 – Sluttkommentarer om Matrisemultiplikasjon, Modellkode for parallelle systemer, Vranglås + evt. Oppdeling av et problem for parallellisering

Arne Maus

OMS,

Inst. for informatikk

Hva så vi på i Uke4

1. Kommentarer til svar på ukeoppgaven om matrise-multiplikasjon
   1. Hvorfor disse gode resultatene (speedup > 40)
   2. Hvordan bør vi fremstille slike ytelsestall – 8 alternativer
2. Hvorfor vi ikke bruker PRAM modellen for parallelle beregninger som skal gå fort.
3. Hva skjer egentlig i lageret (main memory) når vi kjører parallelle tråder - the Java Memory Model
4. Hvorfor synkroniserer vi, og hva skjer da,
   1. Hvilke problemer blir løst ?
5. Ny, ‘bedre’ forklaring på Radix

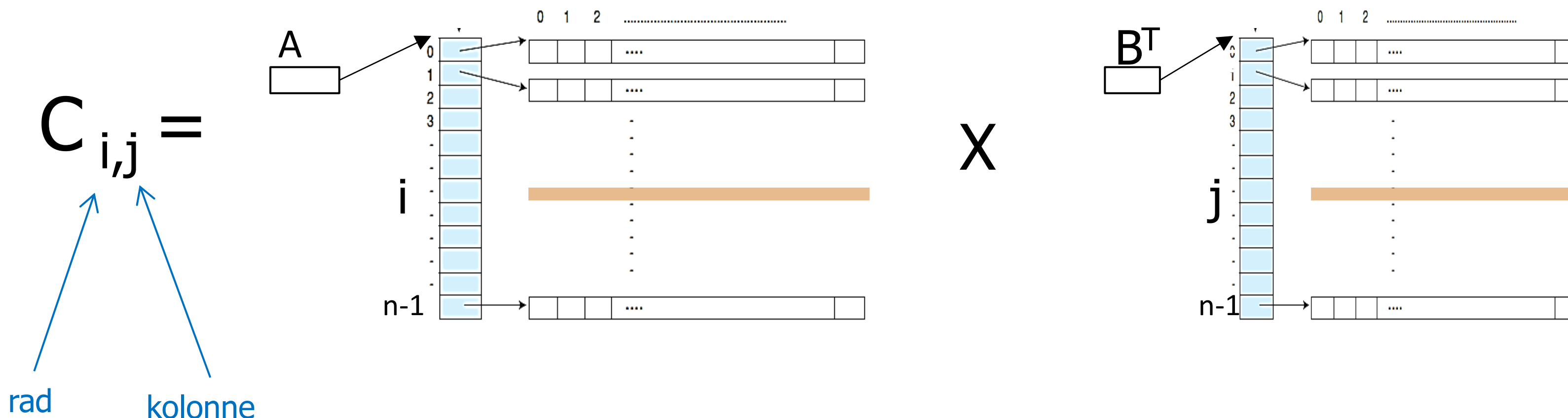
Plan for uke 5

1. Avslutning om matrisemultiplikasjon –
   1. Radvis (ikke kolonnevis) beregning av C
   2. Transponering kan også gjøres parallell (last-ballansering)
2. Modellkode-forslag for testing av parallell kode
3. Ulike løsninger på i++
4. Vranglås - et problem vi lett kan få (og unngå)
5. Ulike strategier for å dele opp et problem for parallellisering:

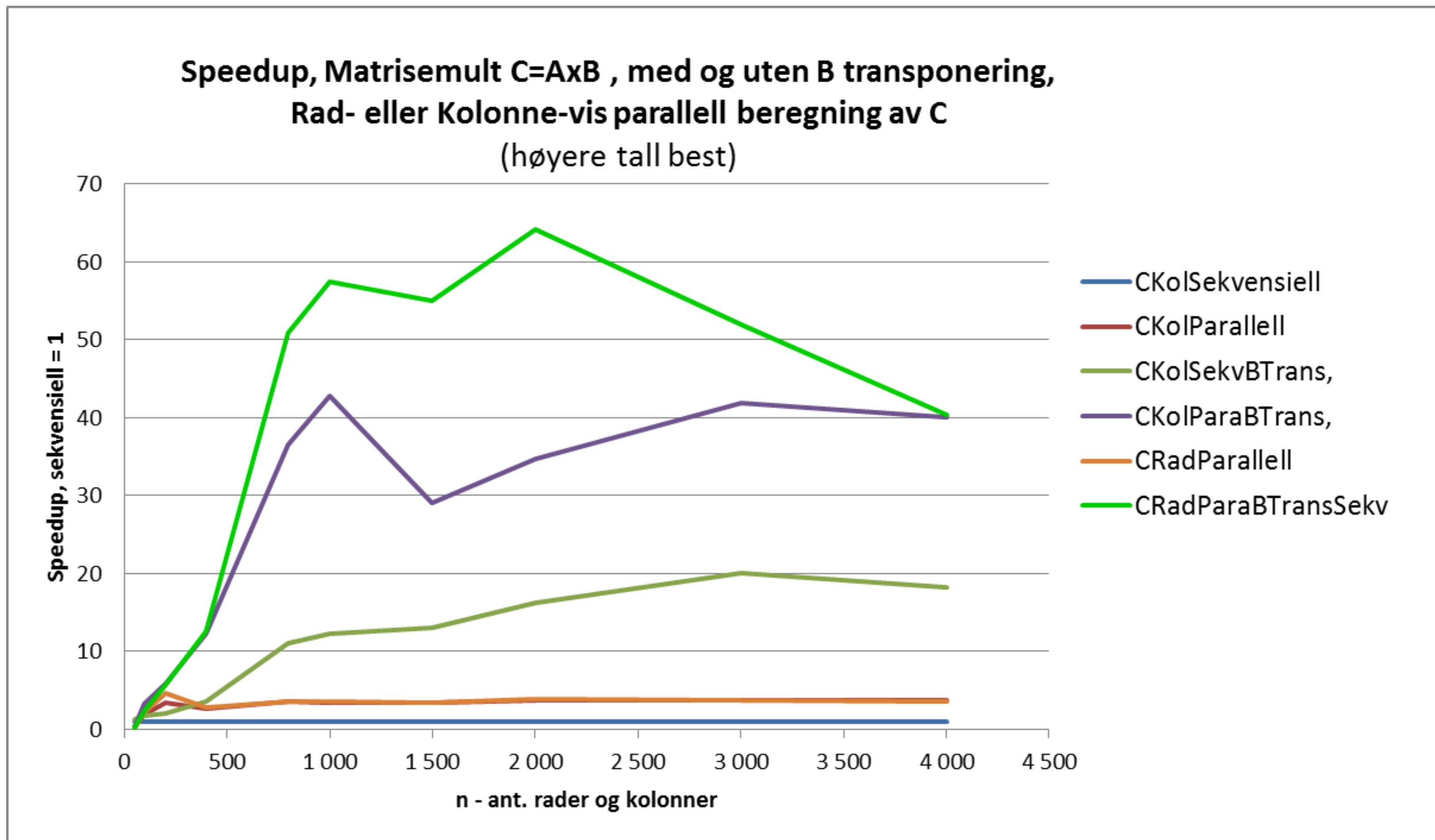
1) Ukeoppgaven denne og forrige uke, matrisemultiplisering, nå radvis oppdeling av C

- Matriser er todimensjonale arrayer
- Skal beregne $C = A x B^T$ (A,B,C er matriser)

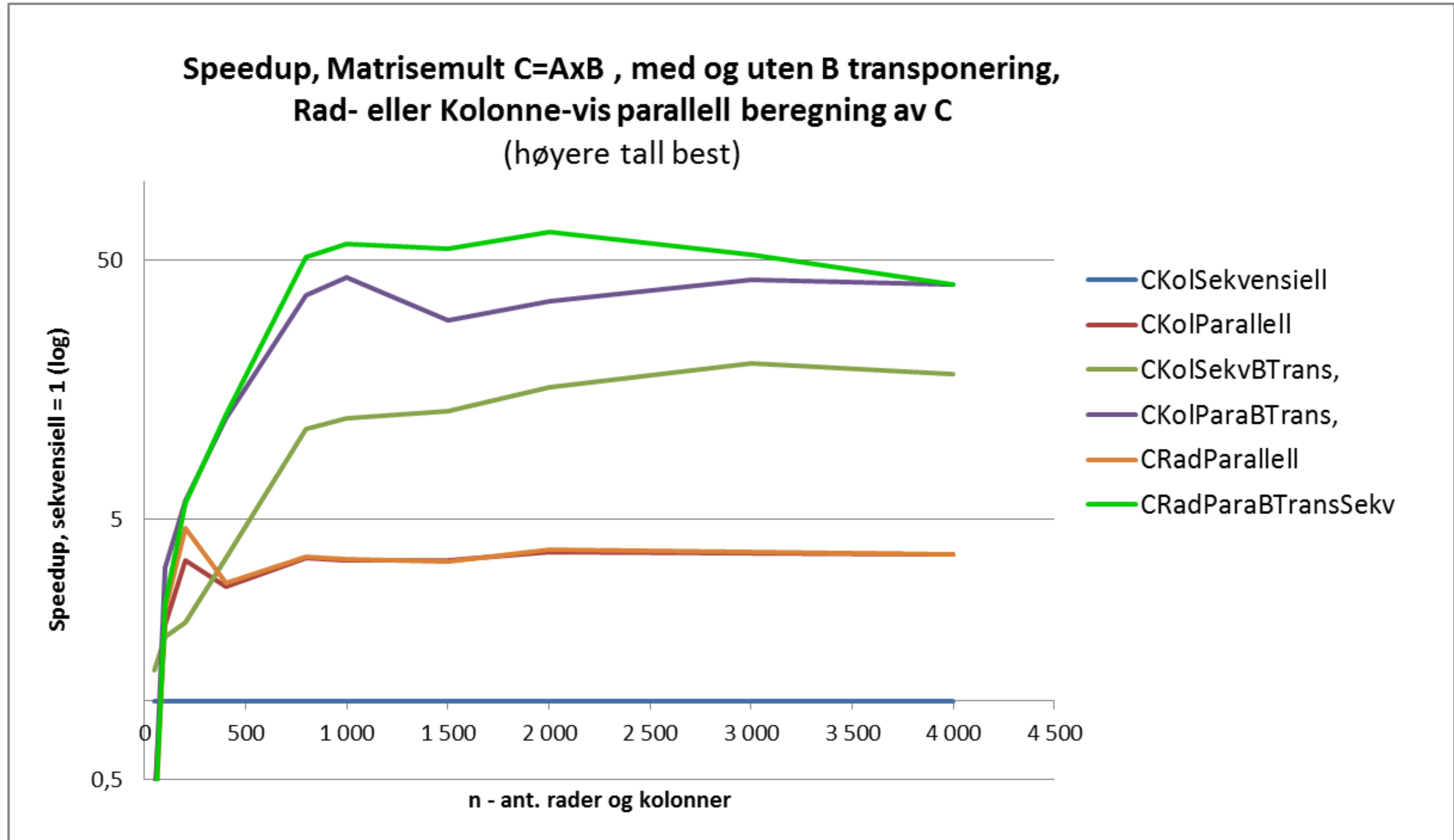
$$c[i][j] = \sum_{k=0}^{n-1} a[i][k] * b^T[j][k]$$



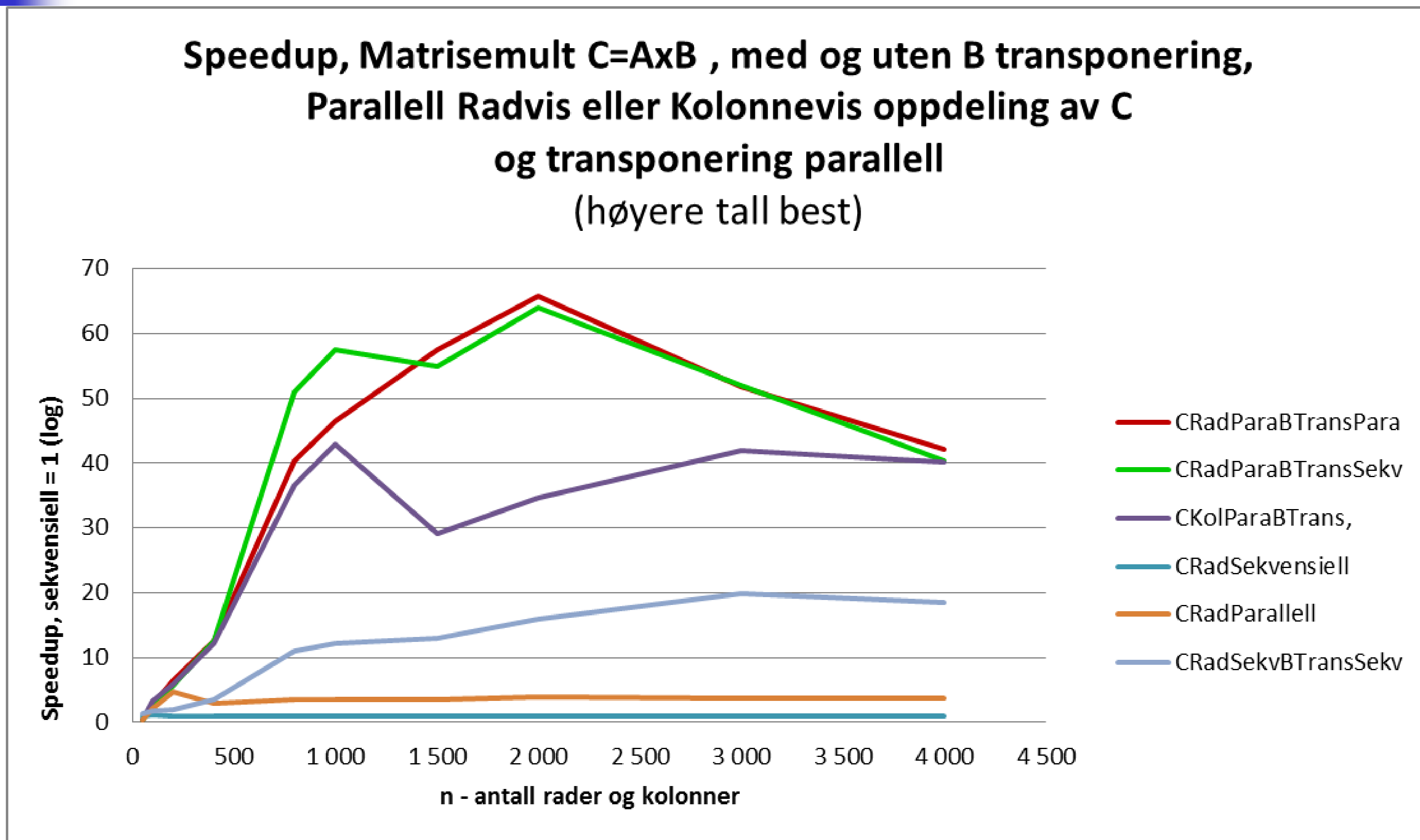
Idéen: La tråd 0 beregne de n/k første radene i C , ikke kolonnene. Den er god fordi en tråd da trenger en mindre del av A i cachene!



Parallellisere C etter rader er enda bedre enn etter kolonner (log y-akse)



Tilleggsforsøk: Som forrige, men + parallellisering av transponeringen (i tillegg til multipliseringen) rød linje



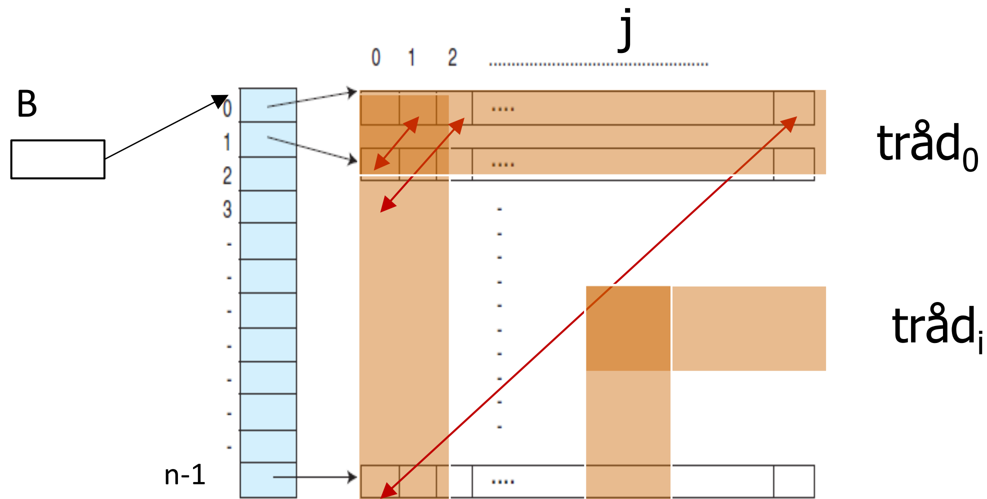
Litt konklusjon om Matrisemultiplikasjon

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

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

To måter for parallell transponering med k tråder

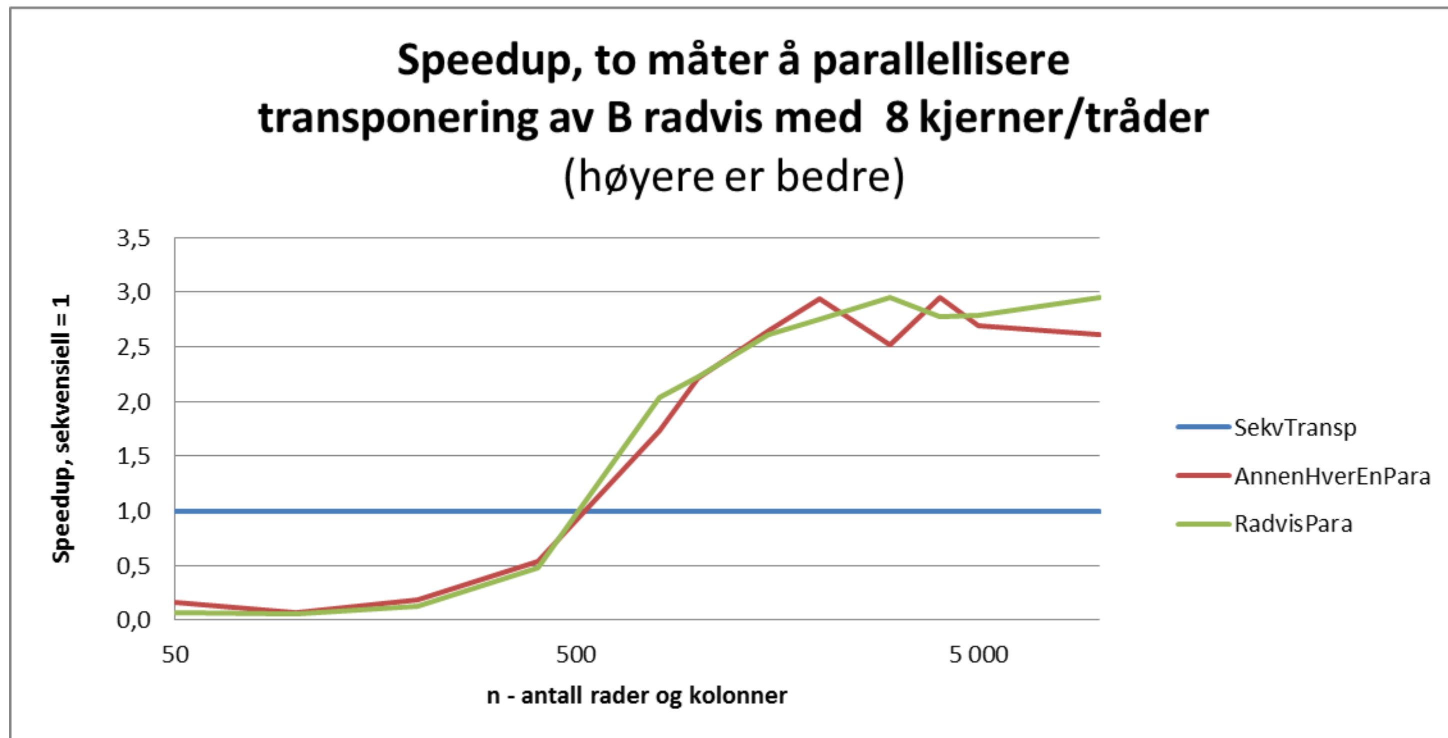
- Radvis – $tråd_0$ får de første n/k radene i B,…osv
  - Men da får $tråd_0$ mye mer å gjøre enn de siste trådene og forsinker at trådene blir ferdige



- Alternativ: $Tråd_0$ tar radene: 0, k, 2k,..; $tråd_1$ tar: 1,k+1,…
- Er forsøk på lastballansering
  (alle trådene gjør om lag like mye)

Egen graf, to måter å parallellisere transponering av B

Sekvensiell eksekveringstid: 0,09 ms (n=50) til 1318 ms. (n= 10 0000)



Om parallell transponering og lastballansering

- N.B: Hele transponeringen må tas **før** multipliseringen
  - Selv om vi bare skal 'bare' regne ut noen få rader i C, trenger vi hele B^T (eller hele B) for å regne ut disse C-radene.
- Parallellisering ikke viktig for matrisemultiplisering fordi når n=1000
  - Selve multiplikasjonen tar 195 ms. parallellisert
  - Transponering sekvensiell tar 5.5 ms sekvensielt, 2.5 ms parallelt
- Kjøretiden for mult er $O(n^3)$, mens transponering er $O(n^2)$ selv etter parallellisering
- Last-balansering er ofte et viktig prinsipp:
  - La alle trådene få om lag like mye å gjøre,
- men virket ikke særlig bra her

2) 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 min løsning
- For dere andre skal jeg gjennomgå min 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.

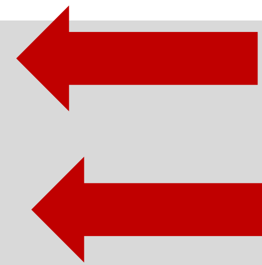
```
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import easyIO.*;
// file: Modell2.java
// Lagt ut 14. feb 2014 - Arne Maus, Ifi, UiO
// Som BARE et eksempel, er problemet med å øke fellesvariabelen i  n*antKjerner  ganger løst

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 ; // for at trådene og main venter på hverandre
             int antTraader;
             int antKjerner;
             static int numIter ;          // antall ganger for å lage median (1,3,5,,)
             static int nLow,nStep,nHigh;  // laveste, multiplikator, hoyeste n-verdi
             static int n;                 // problemets størrelse
             static String filnavn;
             volatile boolean stop = false;
             int med;
             Out ut;

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

```java
/** 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() {
            seqTime = new double [numIter];
            parTime = new double [numIter];
            ut = new Out(filnavn, true);

            antKjerner = Runtime.getRuntime().availableProcessors();
            antTraader = antKjerner;
            vent   = new CyclicBarrier(antTraader+1); //+1, også main
            ferdig = new CyclicBarrier(antTraader+1); //+1, også main

            // 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 {
            nLow = Integer.parseInt(args[0]);
            nStep = Integer.parseInt(args[1]);
            nHigh = Integer.parseInt(args[2]);
            numIter = Integer.parseInt(args[3]);
            filnavn = args[4];
            new Modell2().utforTest();
     }
} // 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 trådene
                               ferdig.await(); // vent på at trådene er ferdige
                            } catch (Exception e) {return;}
                            t = (System.nanoTime()-t);
                            parTime[med] =t/1000000.0;

                             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
```

```
        stop = true;

        try {  // start the other threads and they terminate
                        vent.await();
        } catch (Exception e) {return;}

        ut.close();
    } // utforTest (utføres av main-tråden)


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

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



```
class Para implements Runnable{
            int ind;
            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++){
                        addI();
          }
     }

    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) {
                        //**** 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
           } // while
      } // 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.

3) 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 synchcronized 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.

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();
```

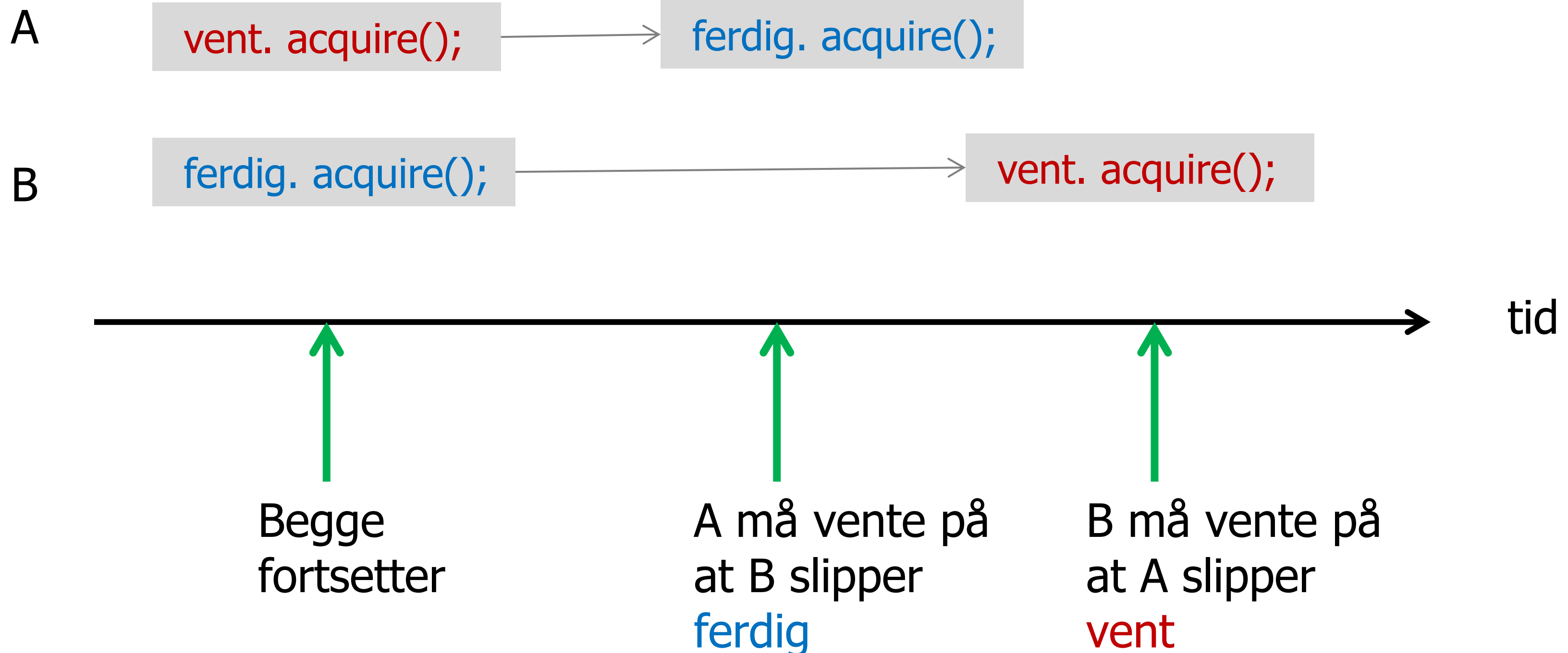
Ytre klasse VrangLaas med to indre klasser SkrivA og SkrivB

```
public class VrangLaas{
    int  a=0,b=0;              // 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-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.

5) 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
  - 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 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 'mye' tid)

Hva har vi sett på i uke 5

1. Mer om matrisemultiplikasjon –
   1. Radvis (ikke kolonnevis) beregning av C
   2. Transponering kan også gjøres parallell (last-balansering)
2. Modell2-kode for sammenligning av kjøretider på (enkle) parallelle og sekvensielle algoritmer.
3. Hvordan lage en parallell løsning – ulike måter å synkronisere skriving på felles variable
4. Vranglås - et problem vi lett kan få (og unngå)
5. Ulike strategier for å dele opp et problem for parallellisering: