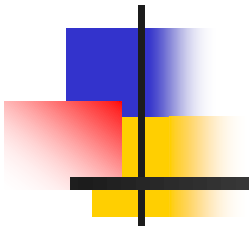
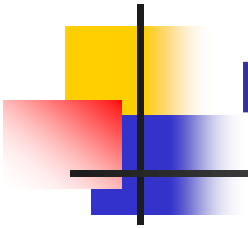


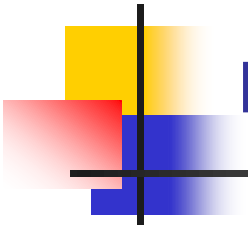
INF2440 Uke 4, våren2014 – Avsluttende om matrisemultiplikasjon og The Java Memory Model + 'bedre' forklaring Radix

Arne Maus
OMS,
Inst. for informatikk



Hva så vi på i uke 3

1. Presisering av hva som er pensum
2. Samtidig skriving av flere tråder i en array?
    1. Går det langsommere når aksessen er til naboelementer?
3. Synlighetsproblemet (hvilke verdier ser ulike tråder)
4. Java har 'as-if sequential' semantikk for et sekvensielt program.
5. Viktigste regel om lesing og skriving på felles data.
6. To enkle regler for synkronisering og felles variable
7. Effekten på eksekveringstid av cache
    1. Del 1 – Radix-sortering sekvensiell
8. Kommentarer til Oblig 1



Plan for uke 4

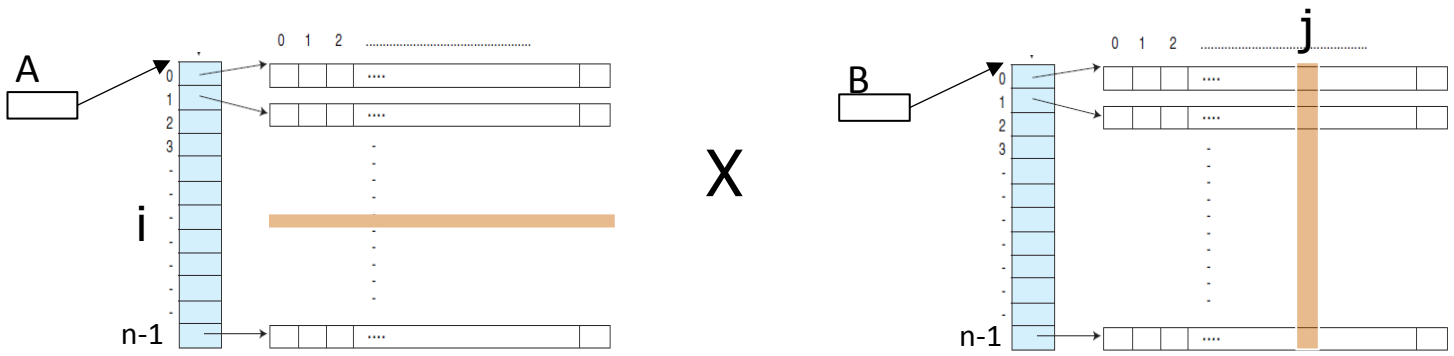
1. Kommentarer til svar på ukeoppgaven om matrise-multiplikasjon
    1. Hvorfor disse gode resultatene (speedup > 40)
    2. Hvordan bør vi fremstille slike ytelsestall – 8 alternativer
2. Vi bruker ikke PRAM modellen for parallelle beregninger
    1. Hva er PRAM og hvorfor er den ubrukelig for oss
3. Hva skjer egentlig i lageret (main memory) når vi kjører parallelle tråder - the Java Memory Model
4. Hvorfor synkroniserer vi, og hva skjer da ?
5. Ny, 'bedre' forklaring på Radix
6. + (hvis tid): Vranglås - et problem vi lett kan få (og unngå)

1) Ukeoppgaven denne uka, matrisemultiplisering

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

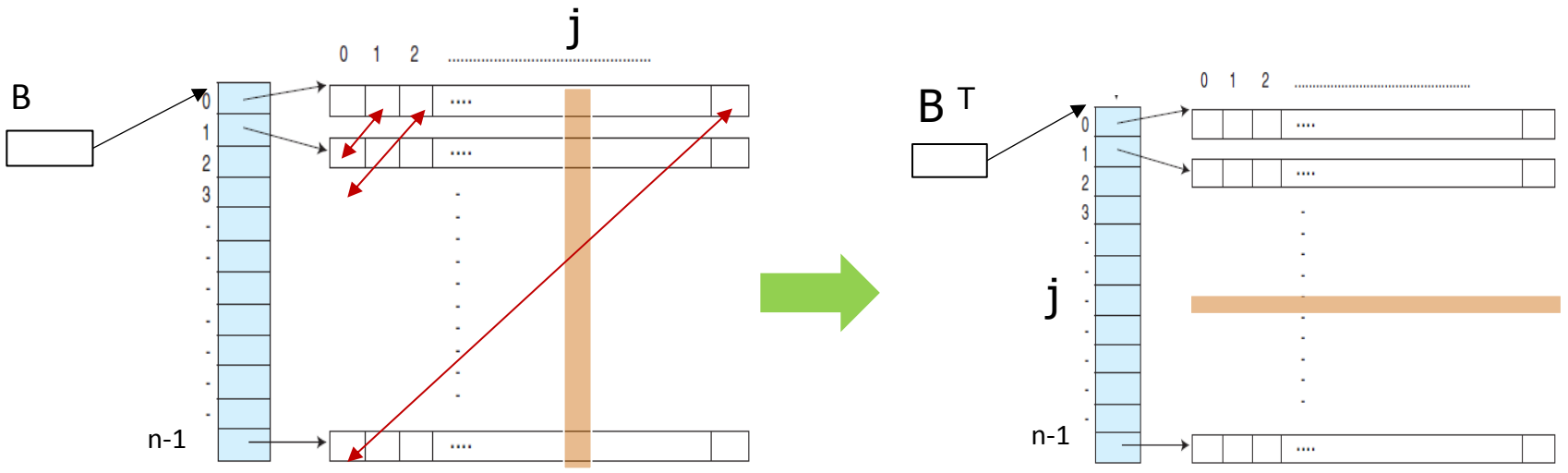
$$c[i][j] = \sum_{k=0}^{n-1} a[i][k] * b[k][j]$$

$C_{i,j} =$



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.

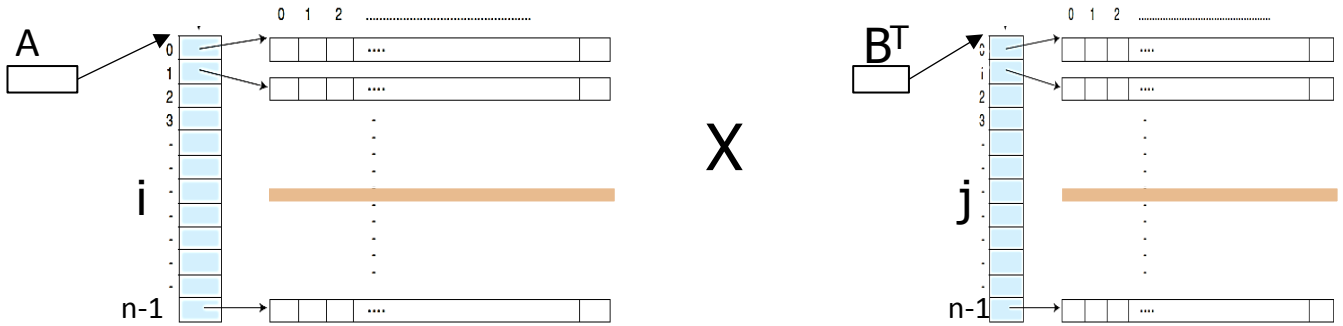


Begrunnelse: Det blir for mange cache-linjer (hver 64 byte) fra B i cachene

Vi har da at litt ny formel for C

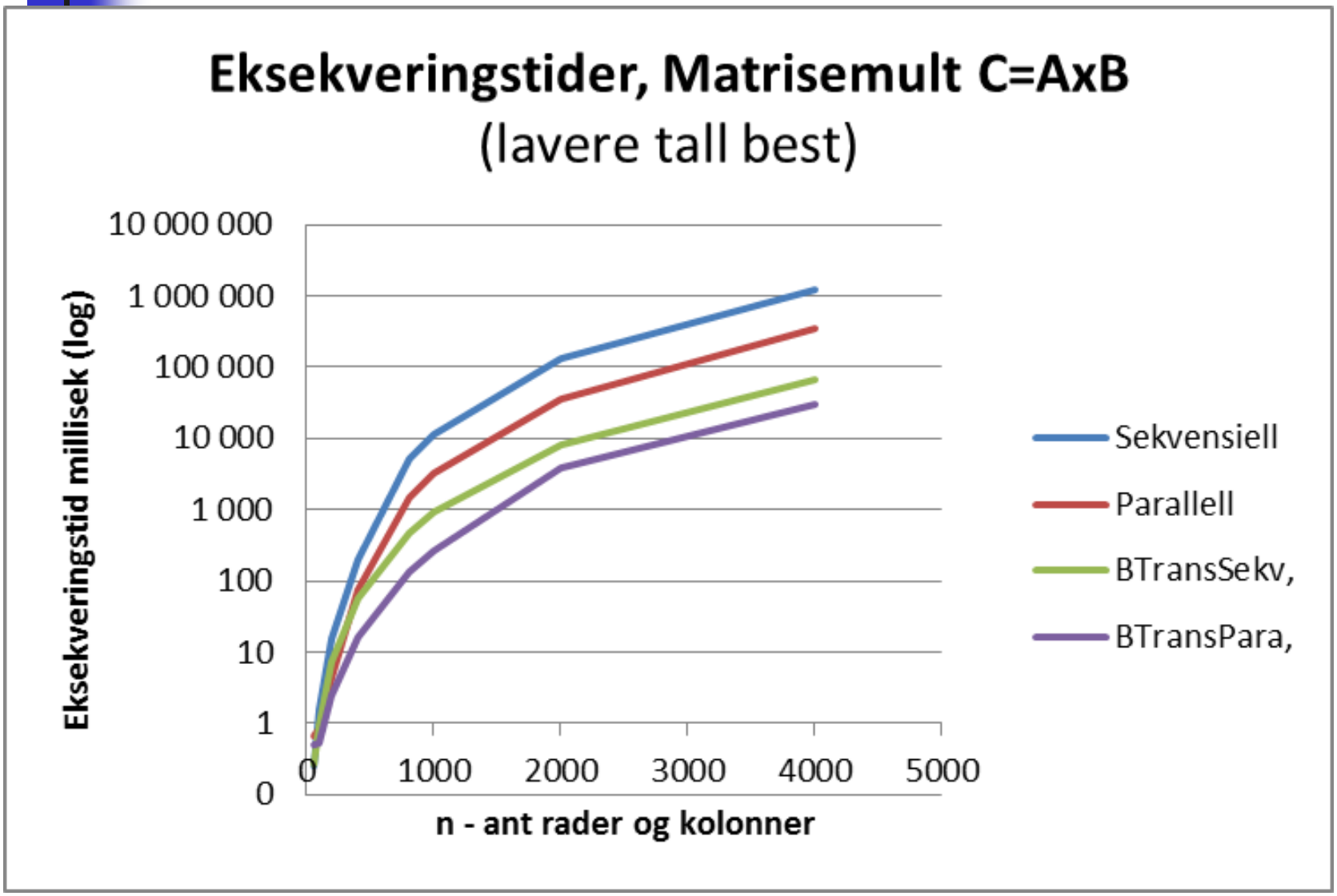
$$c[i][j] = \sum_{k=0}^{n-1} a[i][k] * b^T[j][k]$$

$C_{i,j} =$

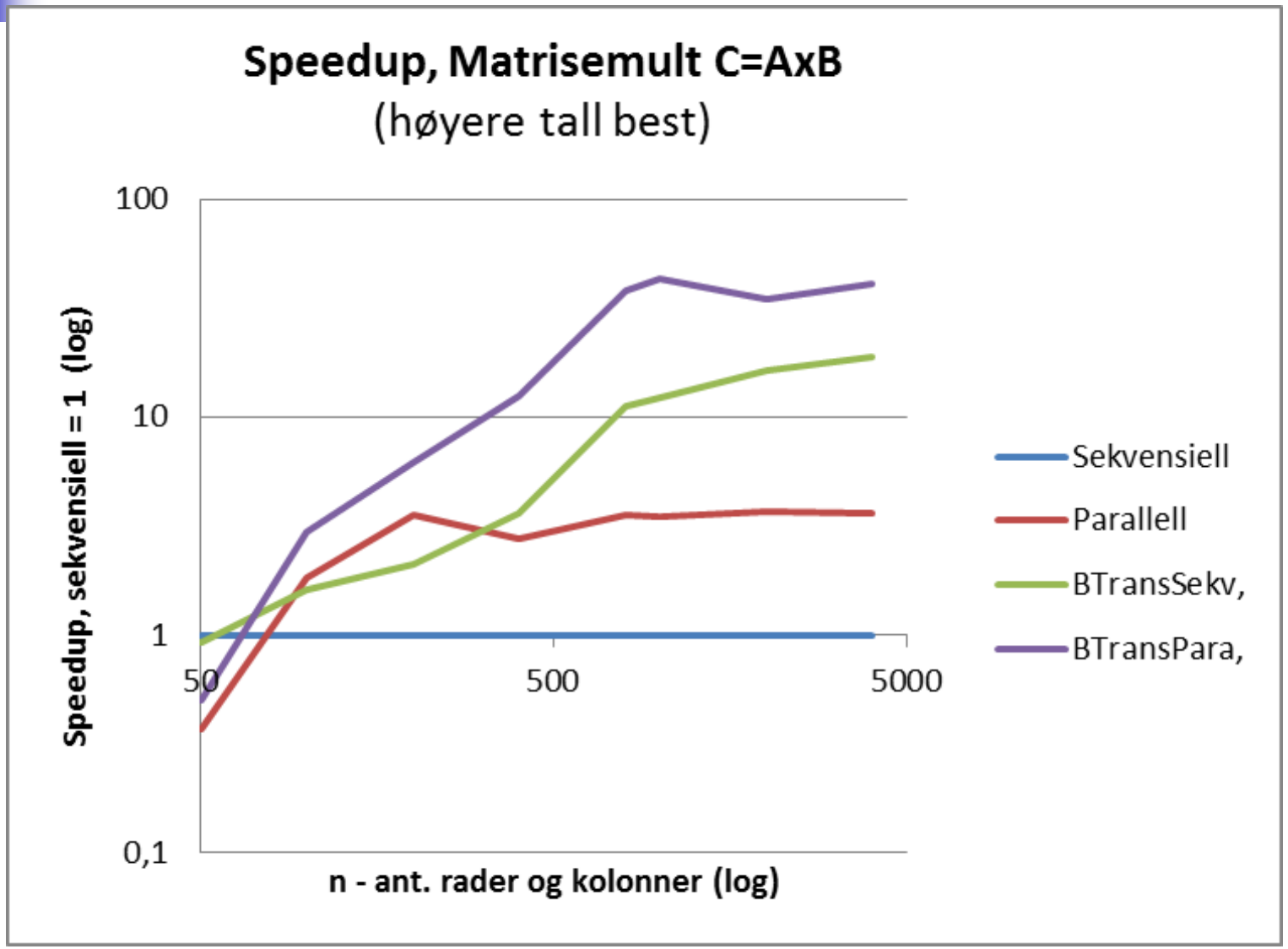


Vi får multiplisert to rader med hverandre – går det fortere ?

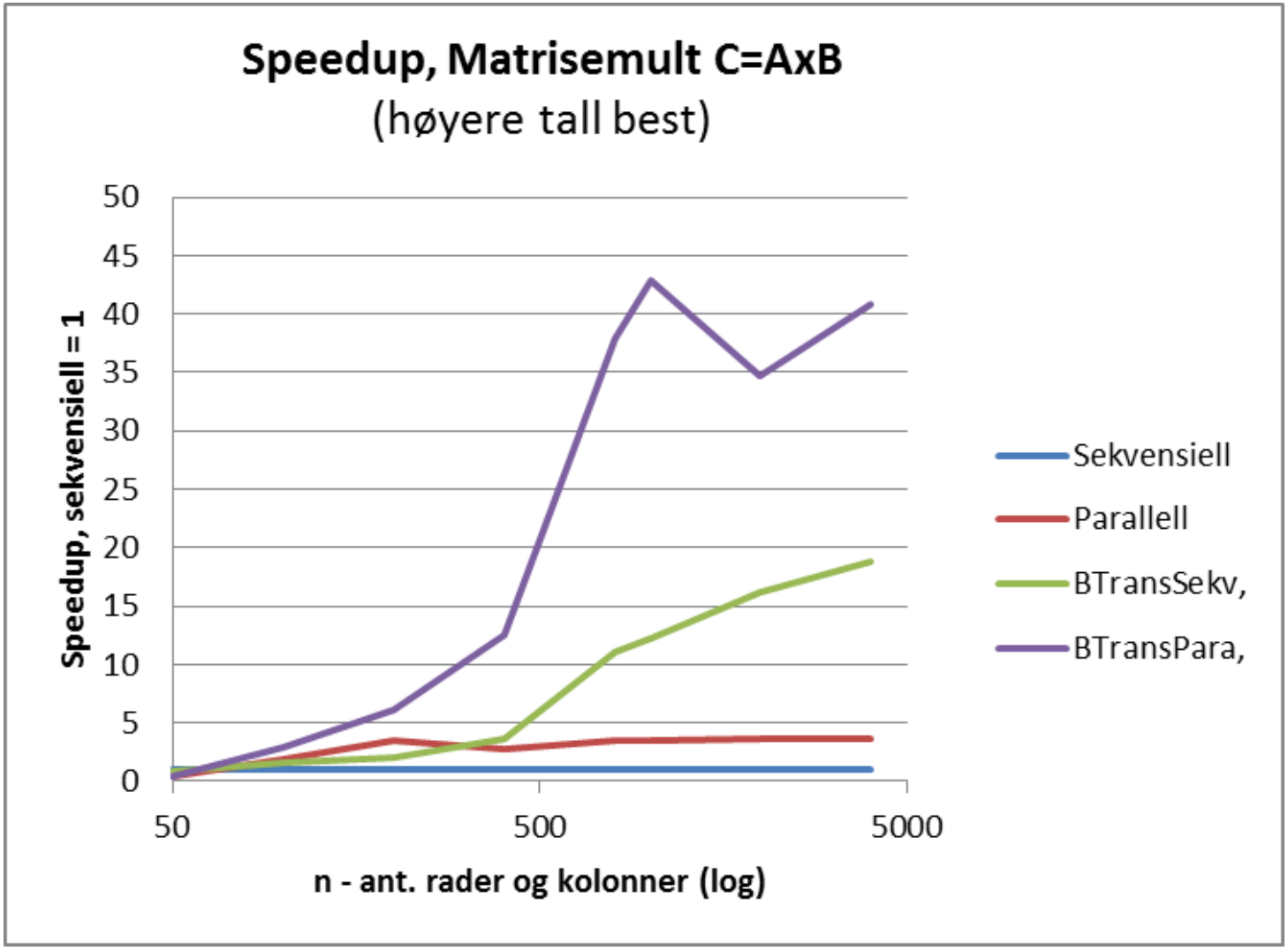
Kjøretider – i millisek. (y-aksen logaritmisk)



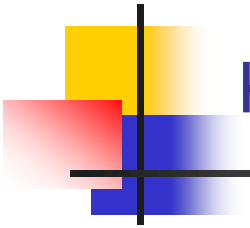
Kjøretidsresultater – Speedup , y-aksen logaritmisk



Speedup – med lineær y-akse



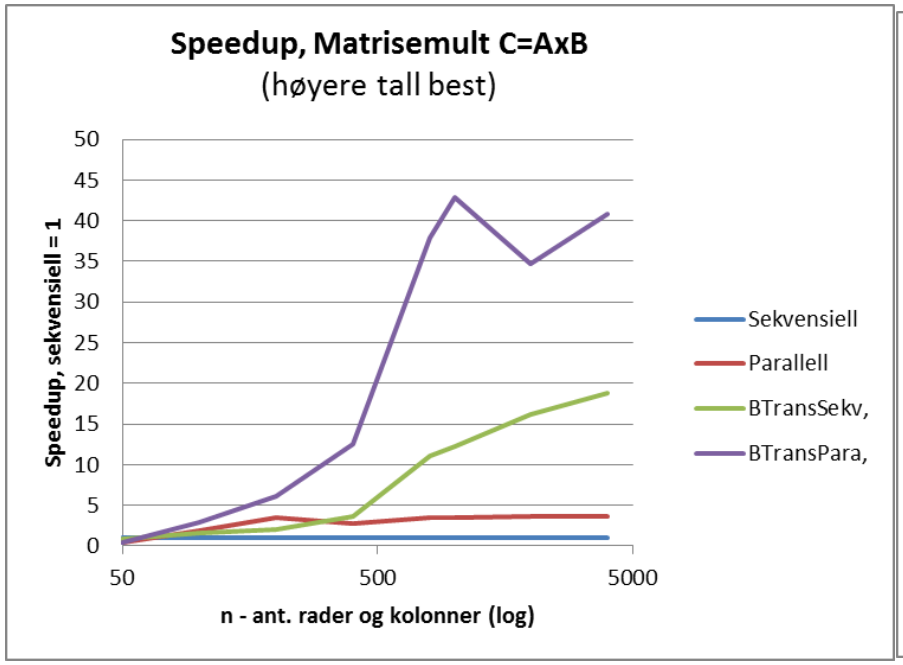
Konklusjon – Matrisemultiplikasjon

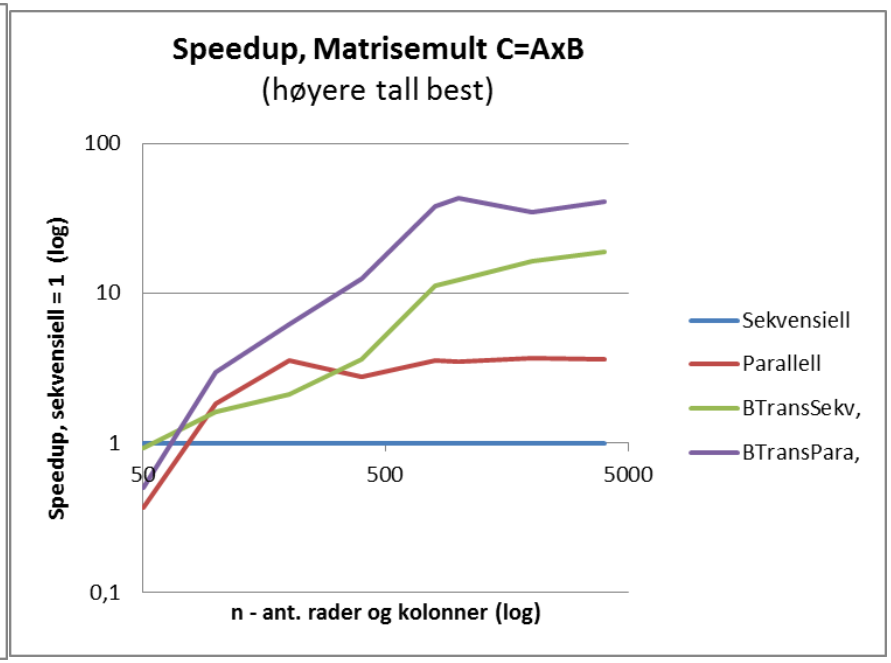


- En rett frem implementasjon gikk ikke særlig fort (når n= 1000), tok det :
  - 11,24 sek med vanlig sekvensiell løsning
  - 3,24 sek. med rett fram parallellisering
  - 0,91 sek med sekvensiell + transponering av B
  - 0,26 sek med først transponering av B, så parallellisert
- Det viktigste når vi skal parallelliser er:
  - Ha den 'beste' sekvensielle algoritmen
  - Så kan du parallellisere den
- Hvis du er venner med cachen, vil parallellisering av en slik algoritme i tillegg nesten alltid lønne seg
- Dette er eksempel på at det å aksessere data fortløpende, ikke på tvers av data , kan gi en speedup på 10-100 ganger.
- (bare multiplikasjonen – ikke transponeringen, ble parallellisert)

Speedup – Hvordan fremstille det?

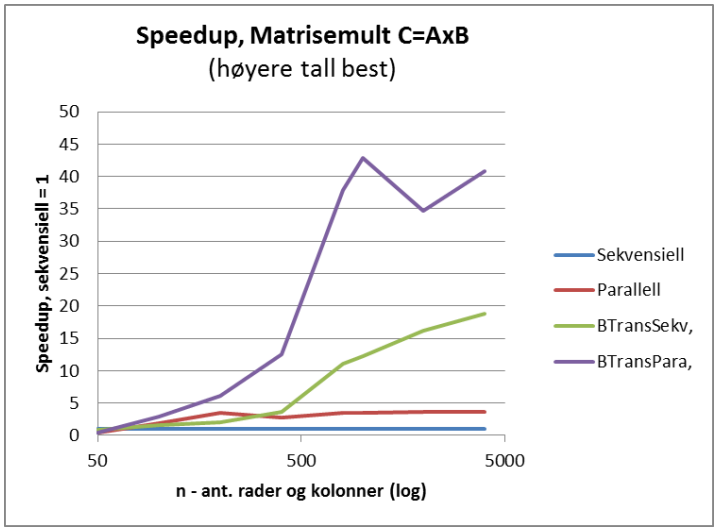
- Disse to kurvene er like ! Hvordan ?

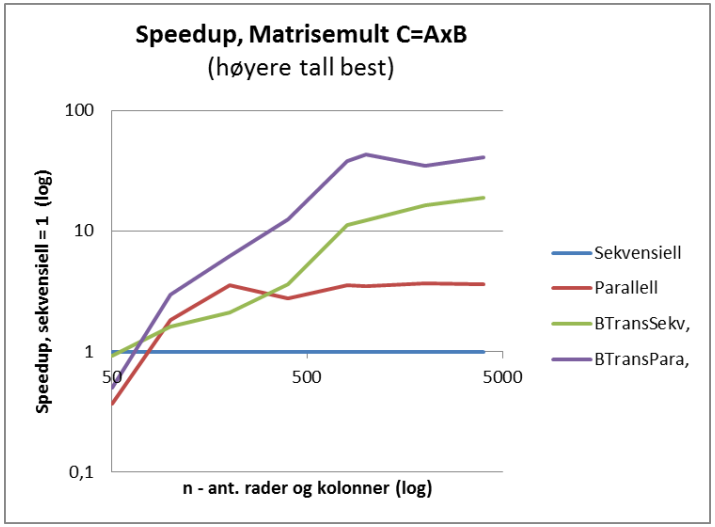


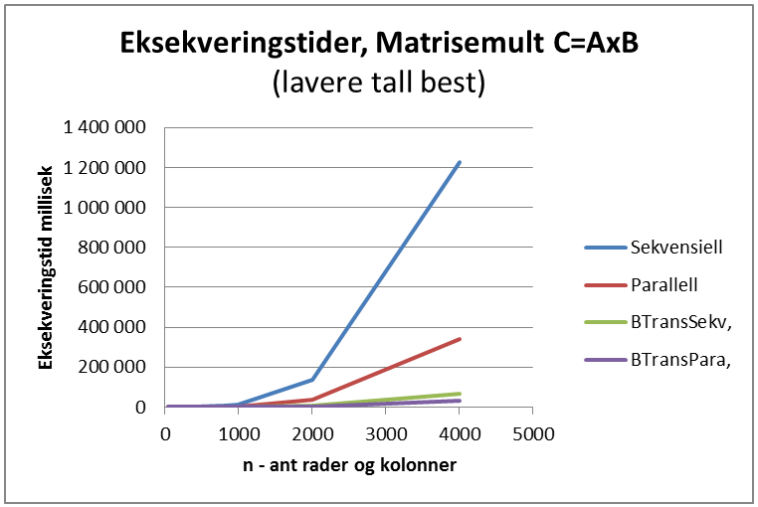


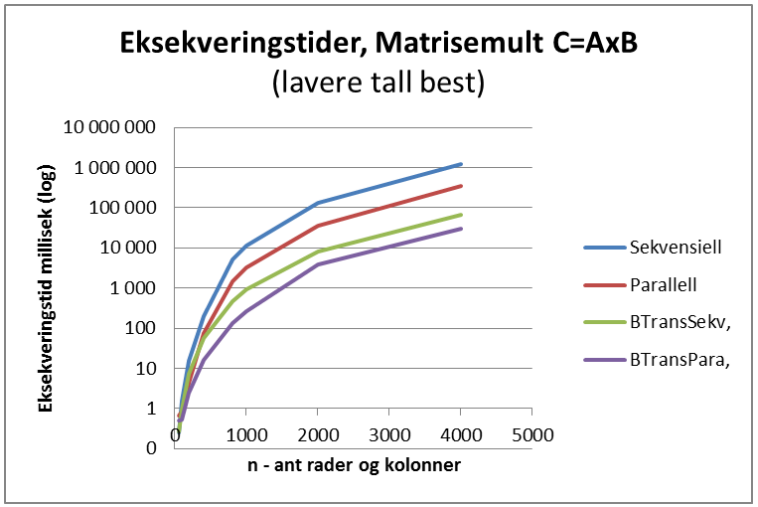
Hvordan framstille ytelse I

- Disse fire kurvene fremviser samme tall! Hvordan ?

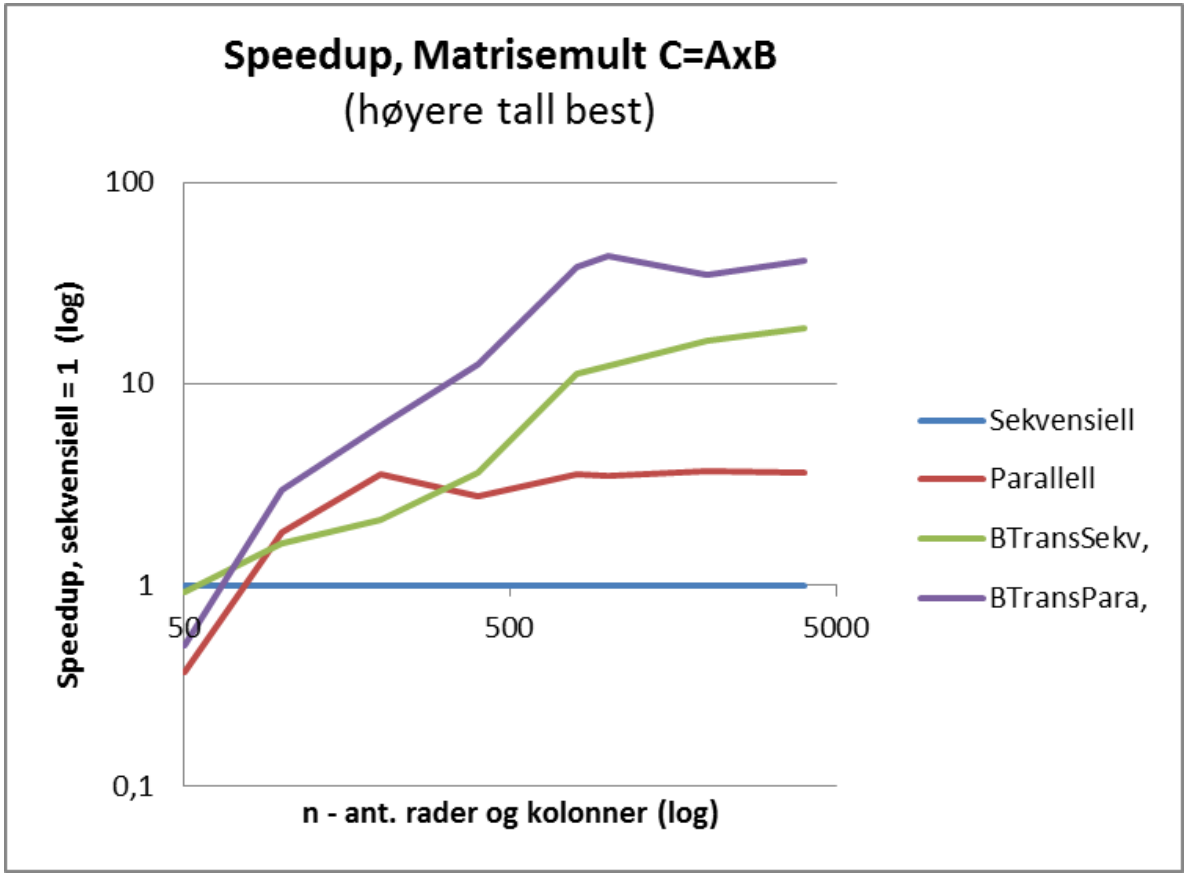






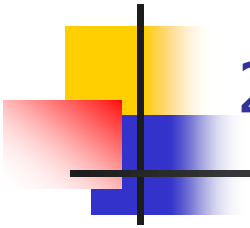


Både logaritmisk x- og y-akse



Fordel med log-akse er at den viser fram nøyaktigere små verdier, men vanskelig å lese nøyaktig mellom to merker på aksene.

2) PRAM modellen for parallelle beregninger



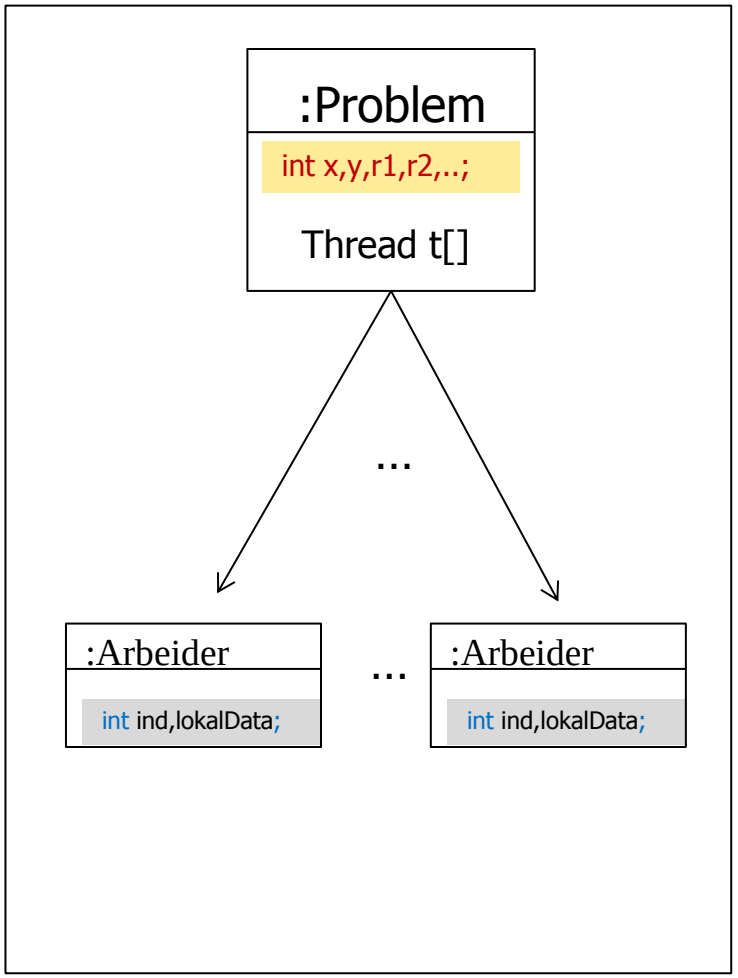
- PRAM (Parallel Random Access Memory) antar to ting:
  - Du har uendelig mange kjerner til beregningene
  - Enhver aksess i lageret tar like lag tid,
    - ignorerer f.eks problemet med cache-hukommelsen
- Da blir mange algoritmer lette å beregne og programmere
- Problemet er at begge forutsetningene er helt gale.
- Svært mange kurs og lærebøker er basert på PRAM

- PRAM vil si at de to sekvensielle algoritmene (med og uten transponering) gikk like fort
- Dette kurset bruker **ikke** PRAM-modellen!

Noen kommentarer til 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
```



3) Utsatte operasjoner i JIT-kompilering

- JIT-kompilatoren kan godt tenke seg å utsette å gjøre noen setninger – eks:

```
y = 17;
x = 4;
z = x + 6;
```

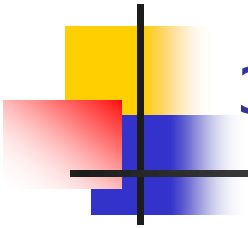
- Vi ser at **`x`** trenges i neste setning, så den blir utført, men 'ingen' trenger **`y`** så den kan vente (til vi får tid, eller droppes hvis ingen andre setninger 'bruker' **`y`**)
- Vi kan ikke vite at **`y == 17`** selv om **`x == 4;`**
- Vi trenger da en måte å ordne opp i bl.a:
  - Utsatte operasjoner blir gjort
  - Alle trådene ser de samme verdiene på felles data.

Repitisjon:
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



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.

3) The Java memory model (JMM) – hva skjer i lageret når vi utfører ulike setninger i et Java-program med tråder?

- JMM definerer hva som er lovlige tilstander på variable når vi har flere tråder , og særlig når flere tråder leser hva en tråd har skrevet; data-kappløp (data-race) om en variabel – lest før eller etter den ble skrevet til – gammel eller 'ny' verdi?.
- JMM prøver å tillate så mye som mulig av kompilator-optimaliseringer (særlig ombytting av setninger) uten at vi får ulogiske/gale resultater.
- Et sentralt begrep er 'hendt før' (happens before) som betyr at av og til må vi vite at noen setninger har blitt utført, før den setningen vi nå ser på kan utføres.
- Det gode budskapet er: Hvis alle trådene oppfyller regel c :
  - Hvis bare én tråd skriver på en variabel må også bare denne tråden lese denne variabelen før synkronisering – ingen andre tråder må lese den før synkronisering.

så sier JMM at programmet er fritt for data-kappløp og vi kan resonere om programmet vårt som om det utføres sekvensielt!

Et kort innblikk i JMM 1 – tillatte og ikke tillatte effekter (ikke pensum)

Initially, x == y == 0

Thread 1	Thread 2
1: r2 = x;	3: r1 = y
2: y = 1;	4: x = 2

r2 == 2, r1 == 1 violates sequential consistency.

- Kommentar : r2 kan lett bli 2, men da er T2 utført først og r1 blir da 0;
- alternativt T1 først og da er r1==1, men r2==0

Initially, x == 0, ready == false. ready is a volatile variable.

Thread 1	Thread 2
x = 1;	if (ready)
ready = true;	r1 = x;

If r1 = x; executes, it will read 1.

Figure 3: Use of Happens-Before Memory Model with Volatiles

Et kort innblikk i JMM 2– tillatte og ikke tillatte effekter (ikke pensum)

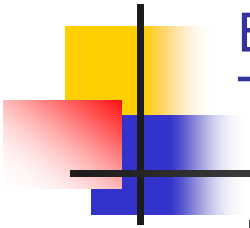
Initially, x == y == 0

Thread 1	Thread 2
1: r1 = x	6: r3 = y
2: if (r1 == 0)	7: x = r3
3: x = 1	;
4: r2 = x	;
5: y = r2	;

Compiler transformations can result in r1 == r2 == r3 == 1

Figure 12: Behavior that must be allowed

- Det rare er her hvordan r1 kan bli 1;
  bare hvis 1: blir utført etter 2: og 3:
- **Konklusjon**: Å sette seg inn i og programmere etter JMM er meget vanskelig (nær umulig å gjøre det riktig) hvis vi tillater data-kappløp om variable.
- Slik programmering overlater vi til de som programmerer våre synkroniserings-mekanismer (CyclickBarrier, Semaphore ,..).



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

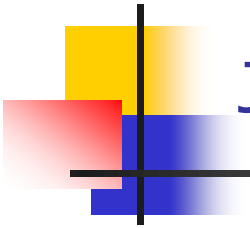
Hva gjør vi hvis vi vil (i en tråd) lese hva en annen tråd har skrevet ?

- 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 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!

JMM og volatile variable – del 1



- Enhver variabel i Java kan deklareres som volatile – eks:
  - volatile int i =0;
  - volatile boolean stop = false;
- En volatile variabel skal ikke caches, men skrives så fort som mulig ned i lageret. Er tenkt å dekke behovet for delte felles variable som kan oppdateres og straks leses av andre tråder.
- Å oppdatere (skrive til) en volatile garanterer også at alle utsatte operasjoner i koden utføres før denne skrivingen.
- Imidlertid er det lett å gjøre feil med volatile variable:
  - Hvis vi deklarerer volatile int i; i vårt program som testet i++, vil det likevel gå galt ved at vi mister oppdateringer.
  - Grunnen til det er at i++ fortsatt er tre operasjoner (les i, i = i+1; skriv i), og den samme problemer ved at to tråder gjør dette samtidig.

JMM og volatile variable – del 2

- volatile variable har likevel en nyttig funksjon, ved at den kan nyttes til at en tråd (også main) kan signalisere til de andre trådene at nå er oppgaven løst – ferdig.
- Antar da at alle trådene har en 'evig' løkke i sin run-metode:

```
public void run() {
 while (! stop);
            try {  // wait on all other threads + main
                    vent.await();
            } catch (Exception e) {return;}

             if (! stop) {

                    sort (a,threadIndex);  // parallell-algoritmen

              } else {
                 try {  // wait on all other threads + main
                       ferdig.await();
                 } catch (Exception e) {return;}

             } // end else
      } // end while
 } // end run
```

```
I ytre felles klasse:
CyclicBarrier vent, ferdig;
volatile boolean stop = false;
….
   /** Terminate infinite loop */
    synchronized void exit(){
        stop= true;
        try{ // start all treads to
              //  terminate themselves
                    vent.await();
        } catch (Exception e) {return ;}
    } // end exit
```

6) Synkronisering på samme synkroniseringsobjekt løser 'alle' problemer - løser synlighet og utsatte operasjoner.

- Hvis alle arbeidertrådene gjør en synkronisering på samme synkroniseringvariabel (en Semaphore, en CyclicBarrier,..)
- Før de slipper løs, er :
  - Alle felles variable som er skrevet på, skrevet med i hoved-lageret
  - Alle utsatte operasjoner er utført
- Alle trådene kan da etter å ha gjennomgått den samme synkroniseringa lese samme verdier på alle felles variable!
- Ingen problemer med en vanskelig JMM hvis vi følger regel C om lesing/skriving på felles variable:
  - «en tråd skriver på en variabel, ingen andre tråder leser eller skriver på den»
- Vi må altså først synkronisere på en felles synkroniserings-variabel for at andres skrivinger skal bli fornuftig lesbart for alle trådene og evt. kunne skrives på av en annen tråd.

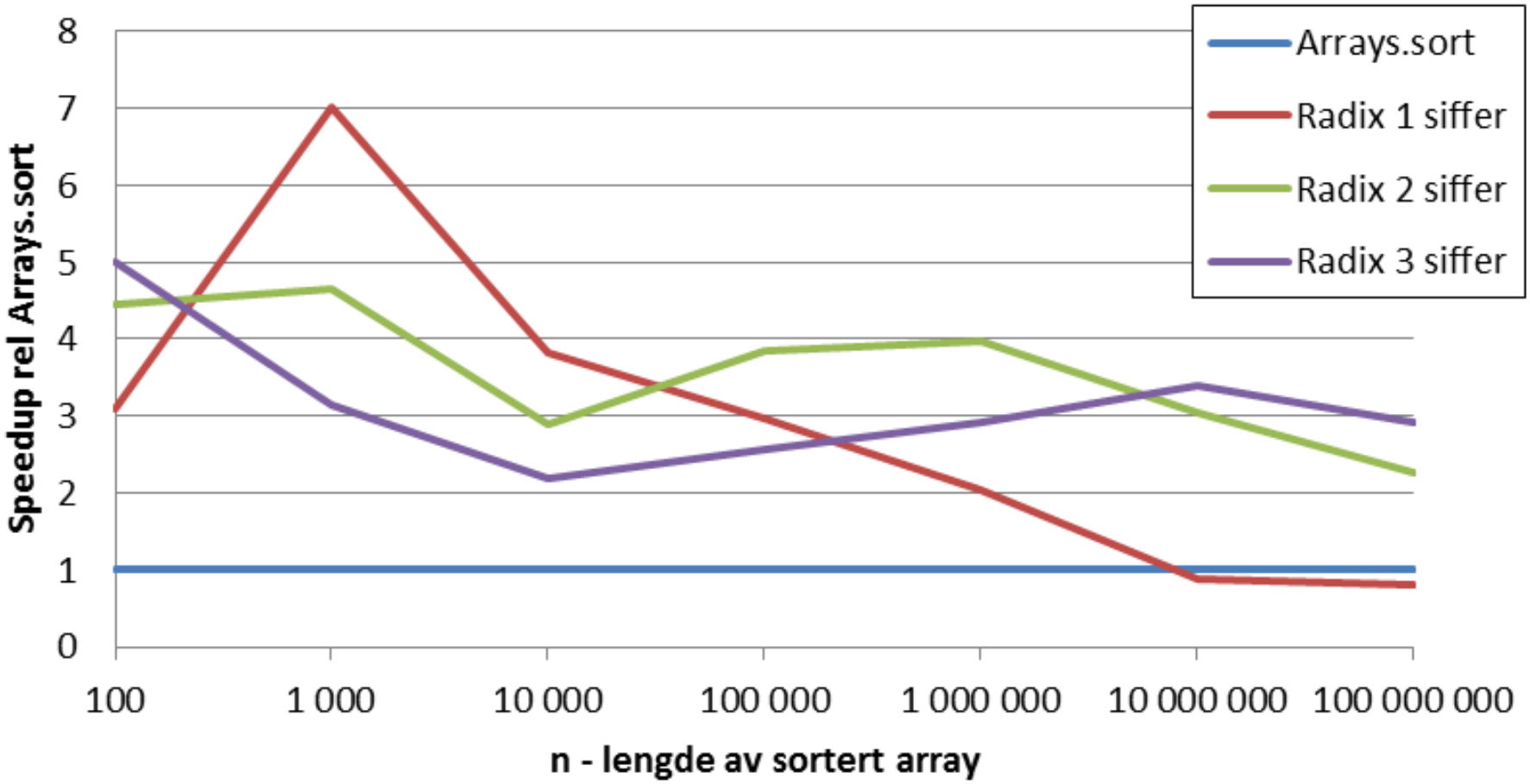
5) Radix-sortering– den sekvensielle algoritmen

- Vi aksepterer at vi forrige gang greide å finne verdien av et siffer i a[]:
  - (a[i]>> shift) & mask – regner ut sifferverdien av et siffer i a[i] som :
    - Har ett eller flere sifre til høyre for seg (mindre signifikante) som til sammen i sum har 'shift' bit
    - Mask inneholder så mange 1-ere nederst som det er bit i det sifferet vi vil finne nå – og er ellers 0.

- Anta at vi skal sortere denne a[] på to sifre,

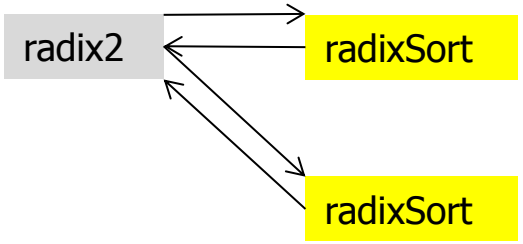
	a
0	6 7
1	4 1
2	7 0
3	1 1
4	0 3
5	1 0
6	3 2

Jevnt over alle n = lengden av a[], er Radix2 bra/best. Velger den !



Høyre, 'minst signifikant siffer først' sortering på to sifre: Radix2 som vi nå bruker

- Radix-sortering, nå 2 siffer:
  - Radix2: Radix-sortering på to sifre
- Radix 2 bestås av to metoder:
  - radix2 som først regner ut max-verdien i a[]. Så regnes ut noen konstanter, som antall bit i de to sifrene a[] skal sorteres med.
  - Deretter kalles metoden radixSort for hvert av de to sifrene (dvs. to ganger)



```
/** Regn ut størrelsen på de to sifrene i a[] og kall RadixSort()*/
static void radix2(int [] a) {
    // 2 siffer radixSort: a[left..right]
    int max = a[0], numBit = 2, n =a.length;

    // finner max = største element i a[]
    for (int i = 1 ; i <= n ; i++)
        if (a[i] > max) max = a[i];

    // numBit = antall bit i max
    while (max >= (1<<numBit) )numBit++;

    int bit1 = numBit/2,    // lengden i bit av første siffer
       bit2 = numBit-bit1; // lengden i bit av andre siffer

    int[] b = new int [n];
    radixSort( a,b, bit1, 0);   // sortert fra a[] til b[] på første siffer
    radixSort( b,a, bit2, bit1); // sortert fra b[] til a[] på andre siffer
}
```

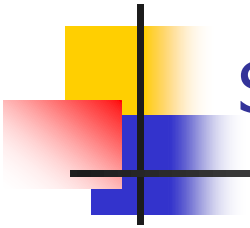
```
/** Sorter  a[] på ett siffer som har maskLen antall bit,
      og hvor det sifferet er shiftet opp 'shift' antall bit i a[] */
static void radixSort ( int [] a, int [] b, int maskLen, int shift){
      int  acumVal = 0, j, n = a.length;
      int mask = (1<<maskLen) -1;
      int [] count = new int [mask+1];

     // a) count= hvor mange ganger finnes de ulike siffervediene i a[]
      for (int i = 0; i < n; i++)
         count[(a[i]>> shift) & mask]++;

     // b) Addér sammen count slik at count[i] sier hvor i b[] vi skal
     //      plassere første element i a[] vi finner med sifferverdien 'i'
      for (int i = 0; i <= mask; i++) {
           j = count[i];
            count[i] = acumVal;
            acumVal += j;
       }

     // c) Finn siffervediene i a[] og flytt de til b[] der count[sifferverdien] sier.
     //      Øk count[sifferverdien] til neste plass i b[]
      for (int i = 0; i < n; i++)
         b[count[(a[i]>>shift) & mask]++] = a[i];

     // steg d) kan fjernes for Radix2 (og Radix4,6,..) ved å bytte om
      // på a og b i kallet
        /* d) kopier b tilbake til a
              for (int i = 0; i < n; i++)  a[i] = b[i] ;
          */
 }// end radixSort
```



Stegene i en radixSort:

a) Tell opp i en array count slik at count[k] = hvor mange ganger k er en sifferverdi a[].
   - Eks. hvor mange tall i a[] = 0 i dette sifferet ?

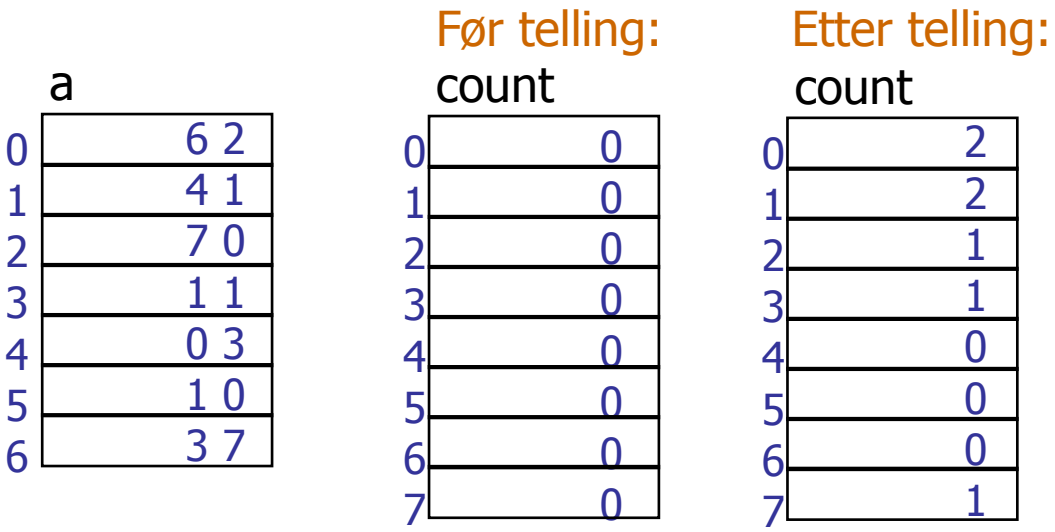
b) Legg sammen antallene i count slik at count[k] sier hvor i b[] vi skal plassere første element i a[] vi finner med sifferverdien 'k'

c) Finn sifferverdien i a[k] og flytt a[k] til b[] der count[sifferverdien] sier a[k] skal være.
Øk count[sifferverdien] med 1 til neste plass i b[]

Radix-sortering – steg a) første, bakerste siffer

Vi skal sorterere på siste siffer med 3 bit sifferlengde (tallene 0-7)

a) Tell opp sifferverdier i count[]:



Radix-sortering – steg b) finne ut hvor sifferverdien skal plasseres

De a[i] ene som inneholder 'j' – hvor skal de flyttes sortert inn i b[] ?
- Hvor skal 0-erne stare å flyttes, 1-erne, ….osv

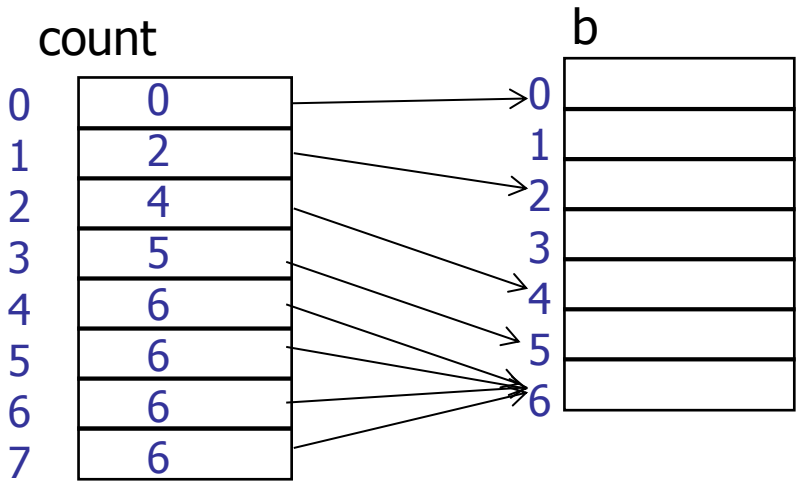
b) Adder opp sifferverdier i count[]:

Før addering:

count

0	2
1	2
2	1
3	1
4	0
5	0
6	0
7	1

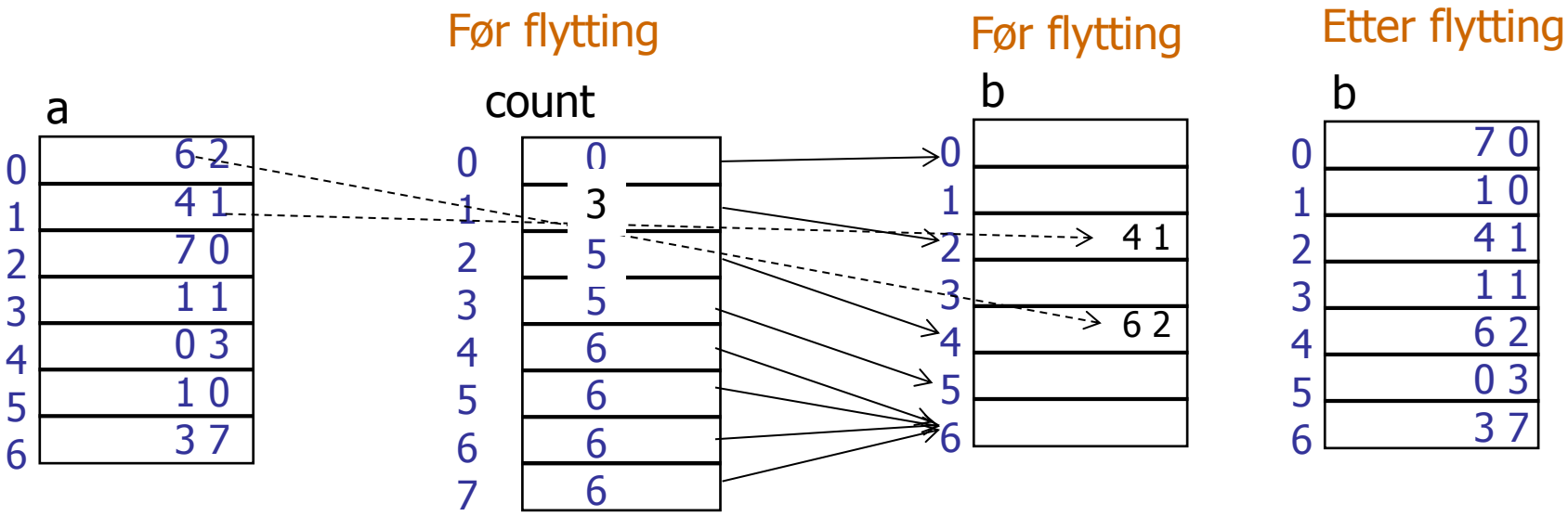
Etter addering :



Kan også sies sånn: Første 0-er vi finner plasserer vi b[0], første 1-er i b[2] fordi det er 2 stk 0-ere og de må først. 2-erne starter vi å plassere i b[4] fordi 2 stk 0-ere og 2 stk 1-ere må før 2-erne,….osv.

Radix-sortering – steg c) flytt a[k] til b[] der count[s] 'peker', hvor s= sifferverdien i a[k]

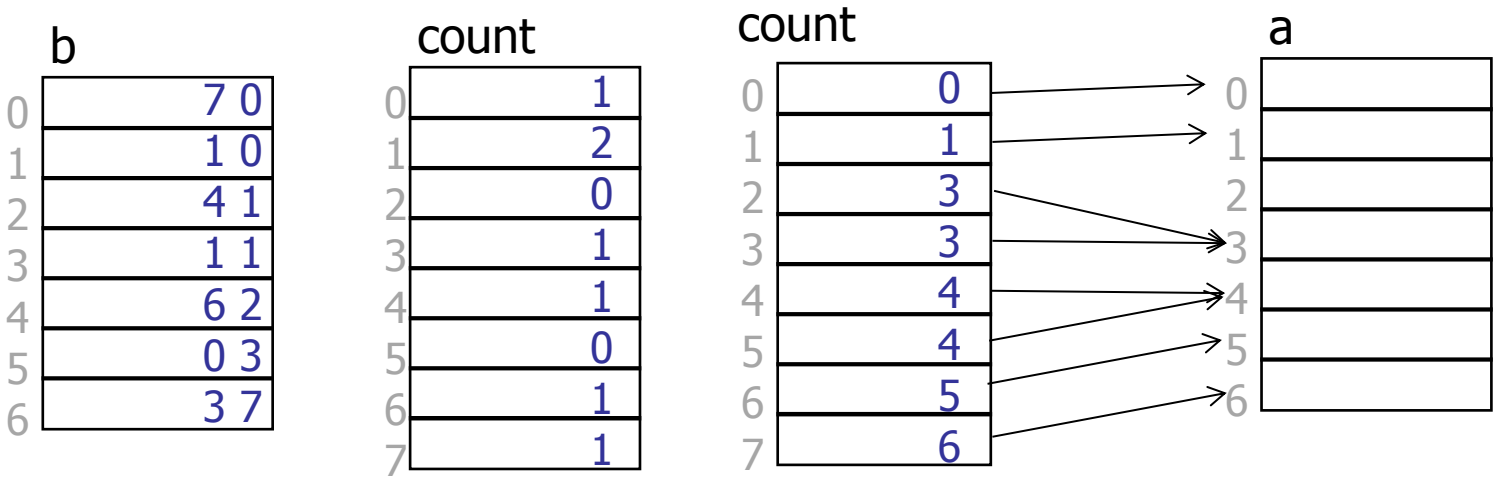
c) flytt a[k] til b[] der count[s] 'peker', hvor s= sifferverdien i a[k], øk count[s] med 1.



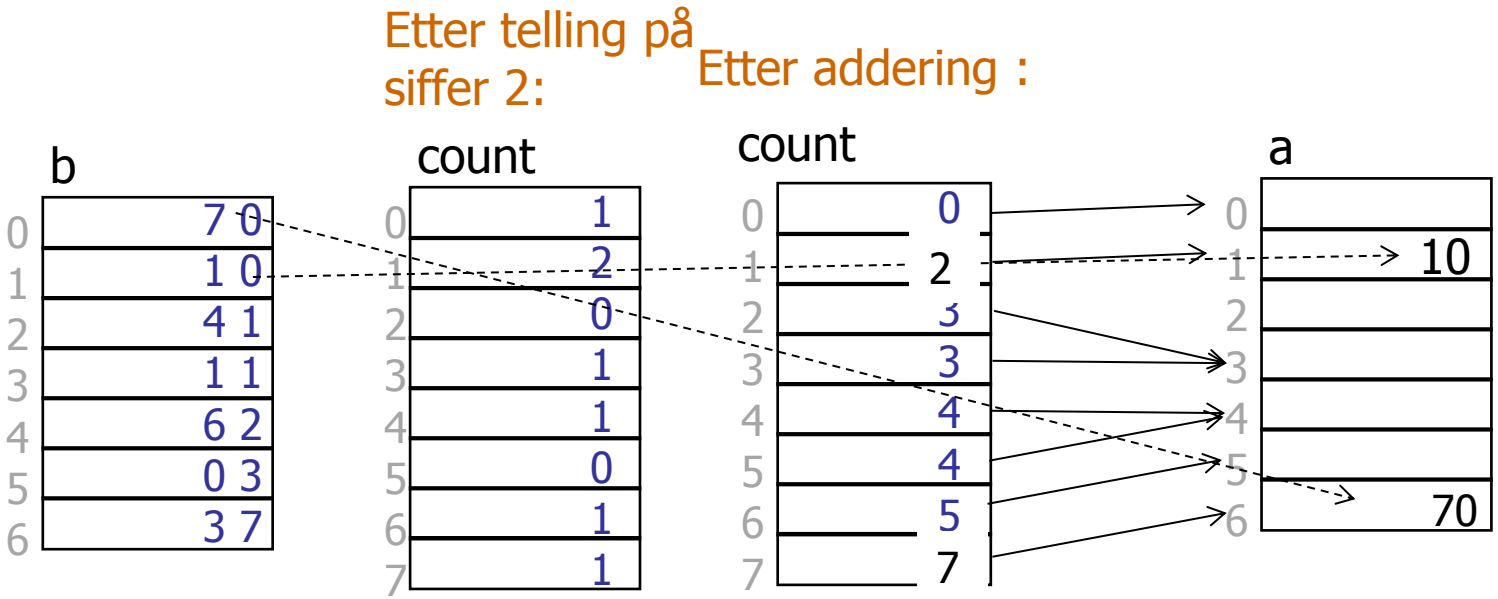
Så sortering på siffer 2 – fra b[] til a[] trinn a) og b)

Etter telling på siffer 2:

Etter addering :

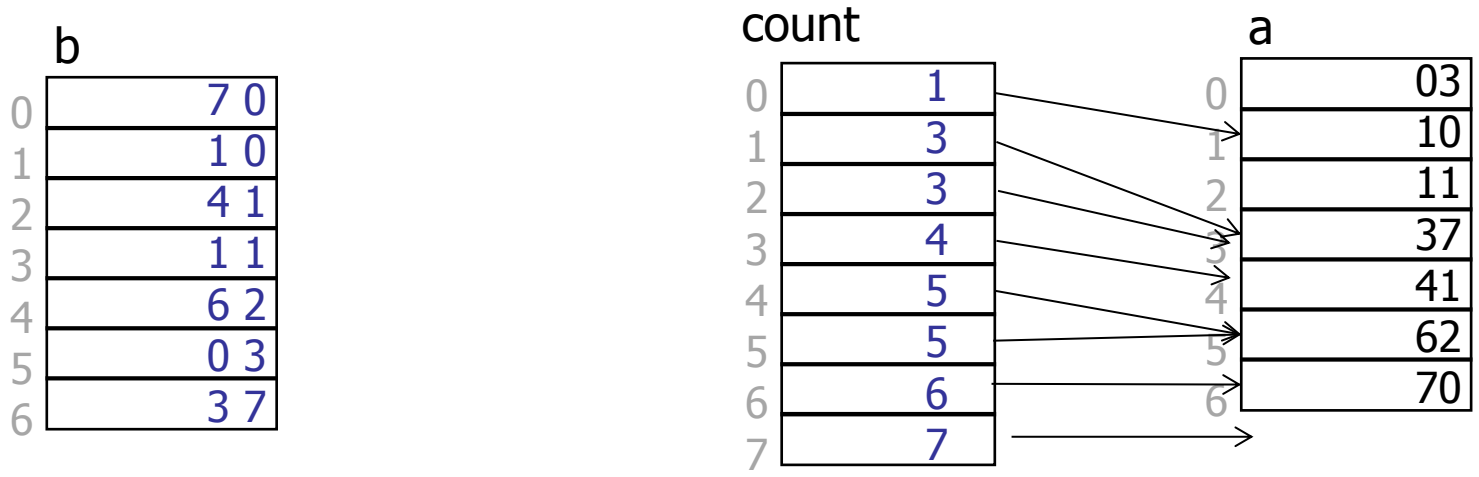


Så sortering på siffer 2 – fra b[] til a[] trinn c)



Situasjonen etter sortering fra b[] til a[] på siffer 2

Etter flytteing



a[] er sortert !

Hva her vi sett 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
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 da ?
5. Ny, 'bedre' forklaring på Radix