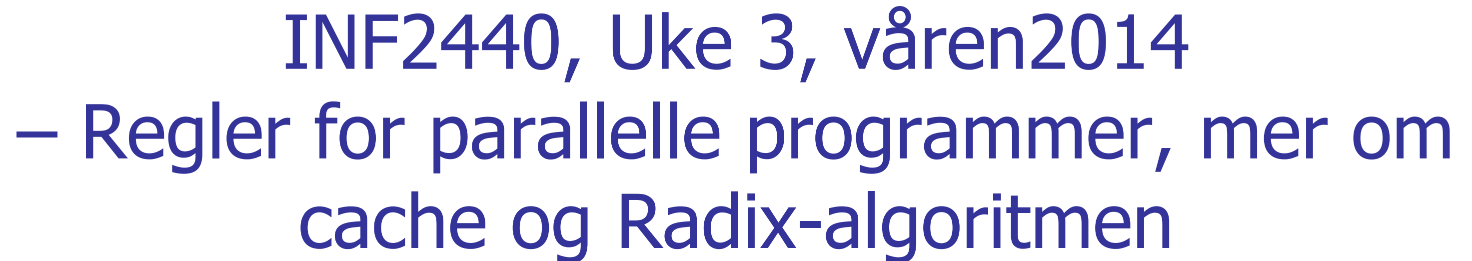


INF2440, Uke 3, våren2014 – Regler for parallelle programmer, mer om cache og Radix-algoritmen

Arne Maus
OMS,
Inst. for informatikk

Hva har vi sett på i Uke2

- I) Tre måter å avslutte tråder vi har startet.
  - join(), Semaphor og CyclicBarrier.
- II) Ulike synkroniseringsprimitiver
  - Vi skal bare lærte oss noen få - ett tilstrekkelig sett
- III) Hvor mye tid bruker parallelle programmer
  - JIT-kompilering, Overhead ved start, Synkronisering, Operativsystem og søppeltømming
- IV) ‘Lover’ om Kjøretid
  - Amdahl lov
  - Gustafsons lov
- Noen algoritmer følger Amdahl, andre (de fleste) følger Gustafson

Plan for uke 3

1. Presisering av hva som er pensum
2. Modell for hvordan vi programmerer
3. Viktige regler om lesing og skriving på felles data.
4. Samtidig skriving med flere tråder i en array?
    1. Går det langsommere når aksessen er til naboelementer?
    2. Hvor fort kan JIT-kompilert kode gå?
5. Synlighetsproblemet
    1. Hvilke verdier ser ulike tråder som leser variable som en annen tråd skriver på?
6. Java har 'as-if sequential' semantikk
7. Effekten på eksekveringstid av cache
    1. Lese og skrive a[b[i]]=i;
    2. Radix-sortering sekvensiell, del 1 –
8. Intro til Oblig 1

1) Hva er pensum ?

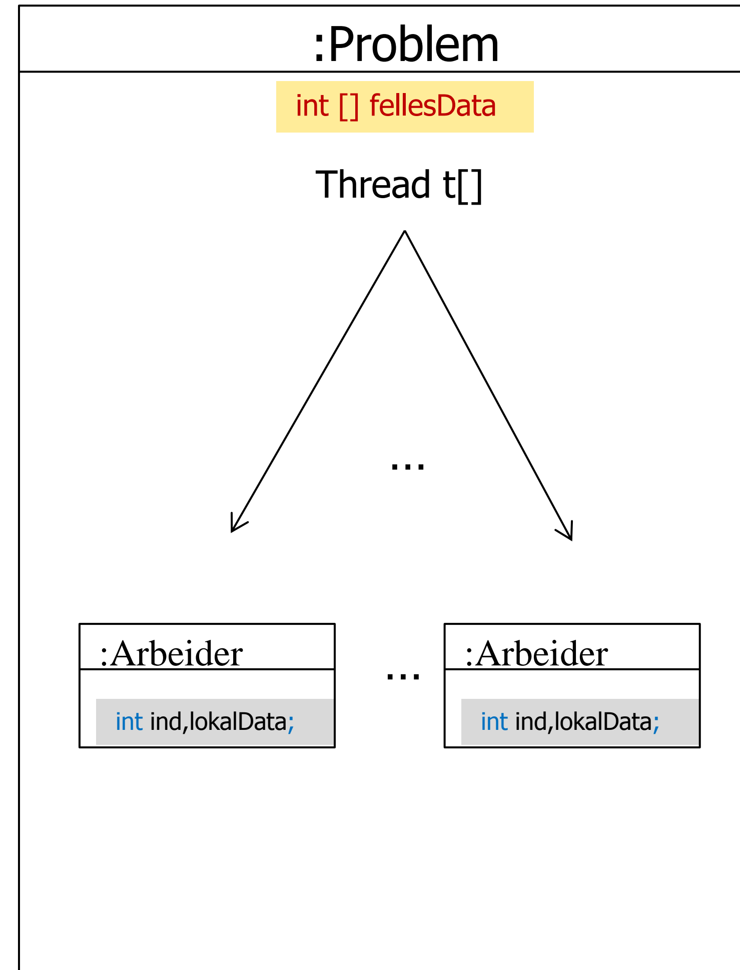
- Alt det som er forelest er pensum
  - Alt som står på foilene kan der spørres om til eksamen.
- Ukeoppgavene er pensum
  - Man skal ha les og forstått oppgavene + forstått løsningene
  - Aller helst bør du selv ha løst oppgavene
- Obligene er pensum
  - Selvsagt også den løsningsmetode som nyttes, skal kunne spørres om til eksamen'

Støttelitteratur – forventes lest:

- Kap 18 og 19 i Rett på Java
- Utvalgte deler av:
  - Brian Goetz, T.Perlis, J. Bloch, J. Bobeer, D. Holms og Doug Lea::"Java Concurrency in practice"
- Utvlgte deler av kap 1,2,og3 i :
  - Michael McCool, Arch D. Robison, James Reinders: «Structured Parallel Programming »

2) Modell for parallelle programmer

```
import java.util.concurrent.*;
class Problem { int [] fellesData ; // 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
```



Slik skal virkelige programmer se ut (**ikke** i kurset)

```
class ProblemX{
     <felles data>

      <type> løsX(...) {
      if (n < LIMIT ){ løsXSekvensielt(param)
      } else {
         <start tråder. De løser hver sin del av
            problemet og sammen hele problemet>;
         <vent på at trådene er ferdige>;
         <hent svaret i felles data og returner>
     } // end løsX

     class ArbeiderTråd{
       <Lokale data for en tråd>;
        ArbeiderTråd (param) {
            <lokale data = param>;
        public static void run() {
            <her løses denne trådens del av problemet
               i ett eller flere steg med synkronisering
                mellom hvert steg>;
         } // end run
     } // end ArbeiderTråd
} // end class ProblemX
```

Dette gjør at programmet blir mer effektivt

- I et virkelig brukerprogram vil vi ha testen:

```
if (n < LIMIT ){ løsXSekvensielt(param)
 } else {
```

- I INF2440 skal vi **ikke** ha denne testen fordi vi er mer interessert i å se når en parallell løsning er langsommere og når den er raskere.
- Vi kan si vi bestemmer LIMIT for ulike problemer:
  - For FinnMax er LIMIT ca = 1 mill.
  - For andre problemer er LIMIT langt lavere f.eks 40 000 for sortering
- I sekvensielle programmer, som sortering gjøres også en slik test og man bruker 'innstikkSortering' hvis n < 32.
  - Arrays.sort() – som er Quicksort, bruker LIMIT = 47

Regler som gjør at programmet blir riktig - I

1. Alle arbeider-tråder har en lokal variabel: **int indeks** (=0,1,2,..,antTråder-1)
2. Vi antar at brukere som kaller løsX-metoden kjører på main-tråden.
   - Dårlig idé og la en tråd i et 'annet' parallelt problem kalle på en parallell løsning som løsX. Blir fort for mange tråder og tregt.
3. Vi lar trådene løse **hele** problemet.
4. Main-tråden bare initierer felles data og starter hver tråd før den legger seg og venter på at trådene blir ferdige.
   Da er hele problemet løst og ligger i felles data.
5. Problemet som arbeider-trådene skal løse kan bestå av ett eller flere steg. Vi synkroniserer da alle arbeider-trådene med en CyclicBarrier mellom hvert av stegene.

(fortsettes neste foil)

Regler som gjør at programmet blir riktig - II

6. Må ett av stegene (f.eks det siste) være sekvensielt, lar vi bare tråd med indeks == 0 gjøre det:

   ```
   if (indeks == 0) {
      < Gjør det sekvensielle steget før neste synkronisering>;
   }
   ```

   De andre arbeider-trådene går her bare rett til neste barrier-synkronisering (eller avslutning).
7. Arbeider-trådene initierer bare lokale variable i sin konstruktør.
   - Husk at objektet ikke er ferdig når konstruktoren kjører. Mye galt kan skje (se boka JCiP kap 3.2) hvis andre tråder får en peker til objektet før det er ferdig.
   - Ingen kall til andre metoder i konstruktoren.
8. All handling i arbeider-trådene skjer i run() og i metoder kalt fra run().

3) Meget viktige regler om lesing og skriving på felles data.

- Før (og etter) synkronisering på felles synkroniserings-objekt 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++ går galt)
  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.

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

Har vist pkt. A og B, skal nå vise pkt. C

Synlighetsproblemet (hvilke verdier ser ulike tråder som leser variable som en annen tråd skriver på)

- Lage et testprogram som har:
  - To felles variable. int a,b;
  - To klasser, arbeider-tråder SkrivA og SkrivB,
  - en som øker a (100 00 ganger) og som skriver
    ned verdien av a og b i hver sin lokale arrayer: mA[] og mB[].

```
for (int j = 0; j<antGanger; j++) {
        a++;
        mA[j] =a;
        mB[j] =b;
}
```

  - og en annen tråd som tilsvarende øker b

```
for (int j = 0; j<antGanger; j++) {
        b++;
        mA[j] =a;
        mB[j] =b;
}
```

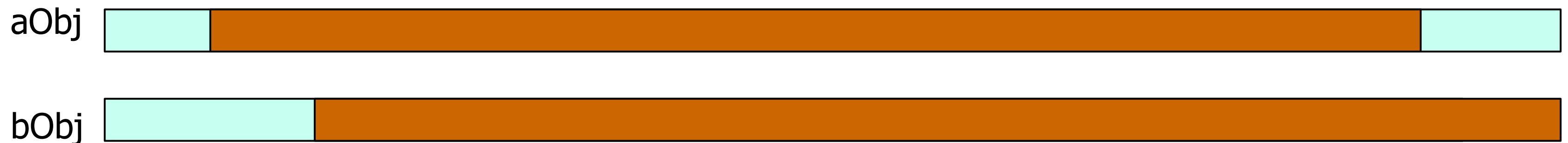
Ytre klasse SamLes med to indre klasser SkrivA og SkrivB

```
public class SamLes{
      int  a=0;   // Felles variabel a
      CyclicBarrier sync, vent ; // begge starter 'samtidig'
      int  antGanger ;
      SkrivA aObj;
      SkrivB bObj;
      int b; // Felles variabel

void utskrift() {  ...   };

void  utfor () {

             vent = new CyclicBarrier((int)antTraader+1);
              sync = new CyclicBarrier((int)antTraader);

             (aObj =  new SkrivA()).start();
             (bObj =  new SkrivB()).start();

              try{
                // main venter på aObj og bObj ferdige
                           vent.await();
             } catch (Exception e) {return;}
        utskrift();
 } // utfor
```

```
class SkrivA extends Thread{
       int [] mB = new int[antGanger],
              mA = new int[antGanger];
       public void run() {
                  try {  // wait on the other thread
                               sync.await();
                  } catch (Exception e) {return;}

                  for (int j = 0; j<antGanger; j++) {
                        a++;
                        mA[j] =a;
                        mB[j] =b;
                   }
                   try {  // wait on the other thread + main
                               vent.await();
                   } catch (Exception e) {return;}
     } // end run A
} // end class Para

class SkrivB extends Thread{
       int [] mB = new int[antGanger],
              mA = new int[antGanger];
       public void run() {
                  try {  // wait on the other thread
                               sync.await();
                  } catch (Exception e) {return;}

                  for (int j = 0; j<antGanger; j++) {
                        b++;
                        mA[j] =a;
                        mB[j] =b;
                   }
                   try {  // wait on the other thread + main
                               vent.await();
                   } catch (Exception e) {return;}
     } // end run B
} // end class SamLes
```

Hva tester vi her ?

- Ser de to trådene (aObj og bObj) alltid oppdaterte verdier av den andre variabelen (ser f.eks objA b helt oppdatert) ?
- Utskrift vanskelig: Selv om starter nesten likt, må de synkroniseres på utskrift (og ikke skrive ut alt!):



Starter utskrift ut når a-verdiene i de to objektene er like og >0, og skriver da ut de 10 neste verdiene av a og b i aObj og bObj

Er det feil her (gamle verdier, e.l)

```
                SkrivA                                SkrivB
a.mA[722]= 723  a.mB[722]= 1458  |  b.mA[1457]= 723  b.mB[1457]= 1458
a.mA[723]= 724  a.mB[723]= 1458  |  b.mA[1458]= 724  b.mB[1458]= 1459
a.mA[724]= 725  a.mB[724]= 1460  |  b.mA[1459]= 725  b.mB[1459]= 1460
a.mA[725]= 726  a.mB[725]= 1460  |  b.mA[1460]= 726  b.mB[1460]= 1461
a.mA[726]= 727  a.mB[726]= 1461  |  b.mA[1461]= 727  b.mB[1461]= 1462
a.mA[727]= 728  a.mB[727]= 1463  |  b.mA[1462]= 728  b.mB[1462]= 1463
```

- NB. SkrivA (=aObj) har a-ene riktige (oppdatert) og SkrivB har b-ene oppdatert
- For eksempel. første og andre linje tvilsomme sammen:
  - A har akkurat økt a fra 722 til 723, og ser b som 1458, MEN
  - B har akkurat økt b fra 1457 til 1458, og ser a som 723
  - I neste linje ser A fortsatt b som 1458, men a i aObj er lik 724
- Dette kan bare forklares ved at A og B operasjonene blandes
- Vi vet ikke når b for aObj har en verdi (eks 1458 eller1460) hvilken a-verdi som hører til disse.
- Og noen verdier for b (1459, 1452) ser aldri aObj, men bObj ser dem.
- **Konklusjon:** Ulike tråder kan se ulike verdier for felles variable og man vet ikke når en tråd har oppdatert (skrevet) på 'sine' variable og det er synlig i annen tråd.

4) Kan det gå galt når to tråder samtidig skriver i ulike plasser i en array?

- Et problemet kunne være at når en av tråden lester opp et element i a[i] (int = 4 byte), så er cachelinja 64 byte, så den får med seg mange elementer før og etter a[i].
- Disse 'andre' elementene er det andre tråder som skriver på.
- Vi skriver et testprogram (ParaArray) hvor 10 tråder med indeks : 0,1,2,..,9 som øker hvert sitt element i en array tall[index] 100 000 ganger.
- Ville det gå fortere hvis hver av trådene skrev i array-elementer som ikke ligger på samme cachlinje eks i a[trådindex*20]

Skriving på nærliggende elementer i en array.

```
class ParaArray{
                int  []tall;
                CyclicBarrier b ;
                int antTraader, antGanger ;
....
class Para implements Runnable{
   int indeks;
  Para(int i) { indeks =i;}
     public void run() {
                 for (int j = 0; j< antGanger; j++) {
                         oekTall(indeks);
                  }
                 try {  // wait on all other threads + main
                      b.await();
                 } catch (Exception e) {return;}
     } // end run
      void oekTall(int i) { tall[i]++; }
  }
} // end ParaArray
```

- Cache-linja er nå 64 byte (og en int er 4 byte)
- Går det greit med at flere tråder (indeks=0,1,…,k-1) skriver på a[tråd.indeks] mange ganger i parallell?
- Tester: Vi lageret program som gjør det :

```
>java ParaArray 10 100000000
Maskinen har 8 prosessorkjerner.
Tid 100000000 kall * 10 Traader =
0.032600 sek,
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
```

Konklusjon:

- Skriving i **ulike** elementer samtidig i en array går bra.
- Dette skal vi bruke mye i kommende algoritmer.

men skriving samtidig i **samme** element, eks:i++, går galt!

Går det langsommere når aksessen er til naboelementer?

- Laget to varianter hvor :
  - Skrivingen gikk tett: a[0] , a[1], a[2],..   (= delte cache-linjer)
  - Skrivingen gikk spredt: a[0], a[20], a[40],..(= ikke-delte cache-linjer)
- Testet det for n = 10, 100,…,1000 mill
- N.B. Dette er sammenligning av to parallelle algoritmer
  (medianer av 9 målinger)

```java
void  utfor () {
     long t;
     vent = new CyclicBarrier((int)antTraader+1); // vent mellom algoritmer

     for (int m = 0; m < 9; m++) {
               tall = new int[antTraader*50];

               // 1) TETT AKSESS, 1,2,3..
               t = System.nanoTime();
               for (int i = 0; i< antTraader; i++)
                    new Thread(new Para(i,1 )).start();
               try{
                  // main thread wenter
                  vent.await();
               } catch (Exception e) {return;}
               tid[0][m] = (System.nanoTime()-t)/1000000000.0;

               // 2) SPREDT AKSESS, 0, 20, 40..
               t = System.nanoTime();
                for (int i = 0; i< antTraader; i++)
                       new Thread(new Para(i,20 )).start();
                try{
                     // main thread wenter
                     vent.await();
                } catch (Exception e) {return;}
               tid[1][m] = (System.nanoTime()-t)/1000000000.0;
     } // end for m

     Arrays.sort(tid[0]);
     Arrays.sort(tid[1]);
     utskrift(tid, 0);
     utskrift(tid,1);
} // utfor
```

```java
class Para implements Runnable{
   int i, step;
    Para (int i, int step) {
               this. i =i;  this. step = step*i;
     }

     public void run() {
                    for (int j = 0; j<antGanger; j++) {
                           okTall(step);
                    }
                    try {  // wait on all other threads + main
                           vent.await();
                    } catch (Exception e) {return;}
     } // end run

     void okTall(int i) { tall[i]++;}
} // end class Para class
```

```
>java  ParaArray10 8 100
Maskinen har 8 prosessorkjerner.

Tid 100 kall * 8 Traader = 0.001356 sek, tett aksess: 1,2,3,...
Tid 100 kall * 8 Traader = 0.001208 sek, spredd aksess: 0,20,40,..

Tid 1000 kall * 8 Traader = 0.001062 sek, tett aksess: 1,2,3,...
Tid 1000 kall * 8 Traader = 0.001343 sek, spredd aksess: 0,20,40,..

Tid 10 000 kall * 8 Traader = 0.001781 sek, tett aksess: 1,2,3,...
Tid 10 000 kall * 8 Traader = 0.001826 sek, spredd aksess: 0,20,40,..

Tid 100 000 kall * 8 Traader = 0.001608 sek, tett aksess: 1,2,3,...
Tid 100 000 kall * 8 Traader = 0.001337 sek, spredd aksess: 0,20,40,..

Tid 1000 000 kall * 8 Traader = 0.001517 sek, tett aksess: 1,2,3,...
Tid 1000 000 kall * 8 Traader = 0.001301 sek, spredd aksess: 0,20,40,..

Tid 10 000 000 kall * 8 Traader = 0.002711 sek, tett aksess: 1,2,3,...
Tid 10 000 000 kall * 8 Traader = 0.002709 sek, spredd aksess: 0,20,40,..

Tid 100 000 000 kall * 8 Traader = 0.018095 sek, tett aksess: 1,2,3,...
Tid 100 000 000 kall * 8 Traader = 0.016519 sek, spredd aksess: 0,20,40,..

Tid 1000 000 000 kall * 8 Traader = 0.174122 sek, tett aksess: 1,2,3,...
Tid 1000 000 000 kall * 8 Traader = 0.159448 sek, spredd aksess: 0,20,40,..
```

Konklusjon om hastighet til naboaksess:

- Forrige utskrift viser for alle tall < 1 mill. mer effekten av optimalisering ved JIT-kompilering (10 000x mer data gir ingen synbar økning i eksekveringstida)
- Det går ca. 10 % langsommere å aksessere nabo-elementer i en array (som er på samme cache-linje, hver 64 byte), enn aksess av elementer som ikke er på samme cache-linje.

5.2) Hvor mye raskere går JIT-et kode – ett eksempel?

- Hvor fort er egentlig optimalisert JIT-kompilering sammenlignet med bytekode interpretert ?
- Kjøring av koden i forrige test
  - Med JIT – kompilering:

```
>java  ParaArray10 8 10000000
Maskinen har 8 prosessorkjerner.
Tid 10000000 kall * 8 Traader = 0.002520 sek, tett aksess: 1,2,3,...
Tid 10000000 kall * 8 Traader = 0.002605 sek, spredd aksess: 0,20,40,..
```

  - Uten (java –Xint) :

```
>java  -Xint ParaArray10 8 10000000
Maskinen har 8 prosessorkjerner.
Tid 10000000 kall * 8 Traader = 1.946415 sek, tett aksess: 1,2,3,...
Tid 10000000 kall * 8 Traader = 1.205889 sek, spredd aksess:
0,20,40,..
```

Ikke generelt, men disse løkkene: 772 ganger fortere

6) Java har 'as-if sequential' semantikk

- Java-kompilatoren med etterfølgende JIT-kompilering til maskinkode + optimalisering:
  - Er egentlig laget for å få til et raskest mulig sekvensielt program (som kjører på en kjerne)
- For å optimalisere koden gjøres mye rart som:
  - Noen setninger uføres ikke i den rekkefølge de står i koden (noen utsettes)
  - Noen kall til metoder kan bli flyttet på (opp eller ned)
  - Noen variable trenges ikke og optimaliseres bort
  - Uttrykk forenkles
  - Sjekk på om arrayer aksesseres utenfor grensene behøver ikke å foretas hver gang (bare først og sist)

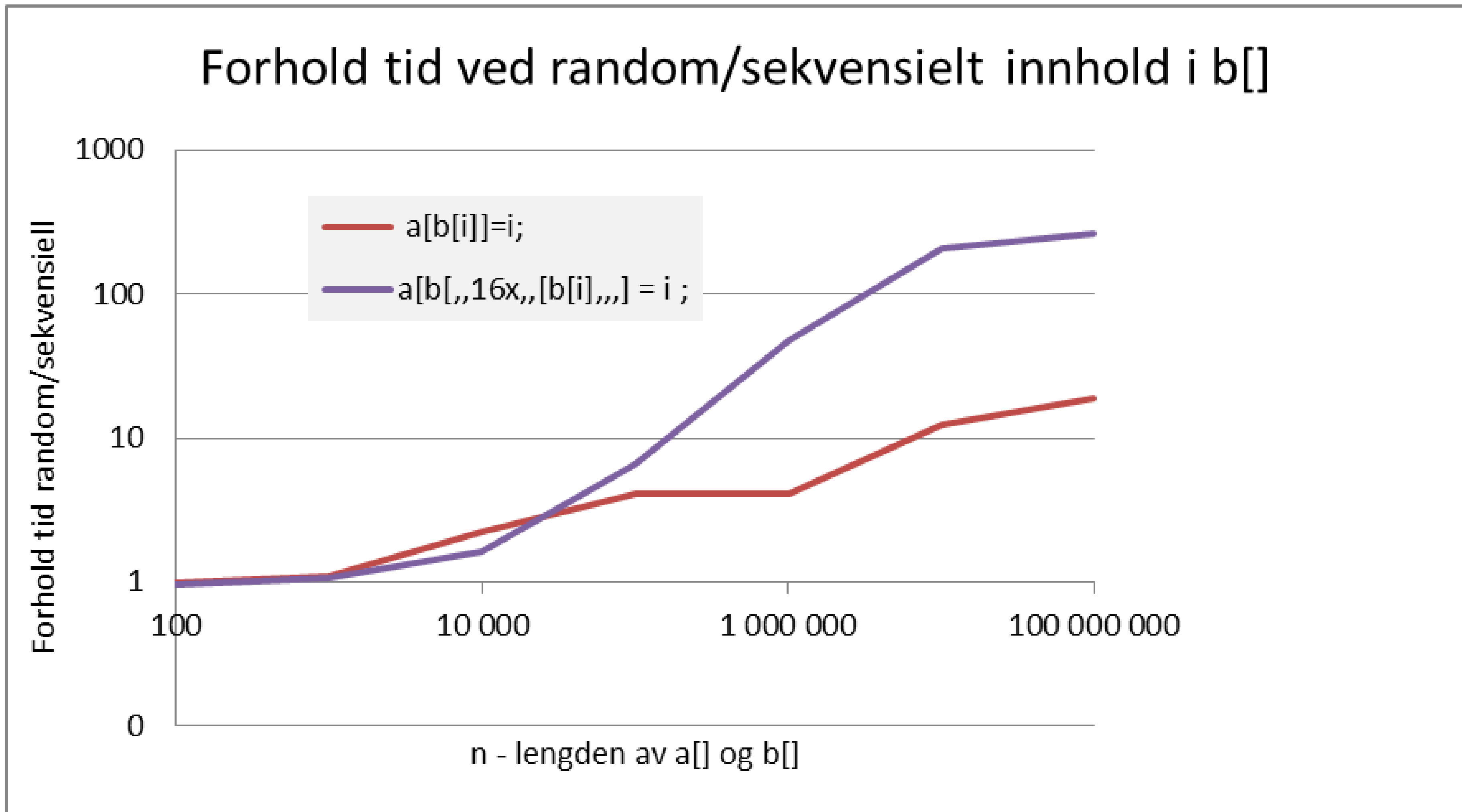
Java har ‘as-if sequential’ semantikk - forts

- Det vi utfører så i parallell i flere eksemplarer, er ikke akkurat den koden vi ser.
- MEN: Java lover deg at det som utføres gir akkurat samme resultat (på utskrift, fil, skjerm,..) som om programmet du har skrevet ble utført slavisk sekvensielt, en setning ad gangen slik det står i koden.
- Dette kalles at: Java har ‘as-if sequential’ semantikk. (semantikk = virkning, virkemåte)

7) Effekten på eksekveringstider av cache

1) Hvor lang tid tar det å utføre n ganger (n=100, 1000, 10 000,….,100 mill):
   1) a[b[i]]=i;
   2) a[b[b[b[b[b[b[b[b[b[b[b[b[b[b[b[b[i]]]]]]]]]]]]]]]]] = i;
2) Avhenger av hva b[] inneholder:
   1) Hvis b[i] = i (sekvensiell) , så er a[b[i]] = a[i] og vi har 'alt' i cachen
   2) Hvis innholdet i b[] er tilfeldig trukket mellom 0:n-1, så er hver les/skriv i lageret en hopping frem og tilbake i a[] – ingen nytte av cachen
3) Neste graf viser hvor mange ganger lenger tid det tar å utføre ganger de to måtene å fylle b[]
   – enten b[i] = i, eller b[i] = random(0..n-1)

Hvor mange ganger tregere går random innhold i b[] enn b[] = 0,1,2,3,.. ?



Konklusjon – nestet aksess a[b[i]]

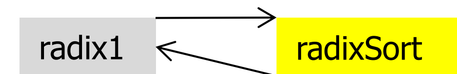
- For ‘små’ verdier av n <1000, gir cachen god aksess til både hele a[] (viktigst), og til b[].
- For store verdier av n > 100 0000 blir det meget langsommere, og vi kan få mellom 12 – 240 ganger langsommere kode (pga. cache-miss) når innholdet av b[] er ‘tilfeldig’.
- Slike uttrykk a[b[i]] og a[b[c[i]]] finner vi i Radix-sortering som vi skal nå granske.

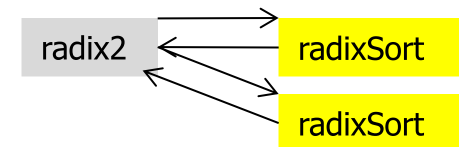
7) Radix-sortering sekvensielt – kode og effekten av cache

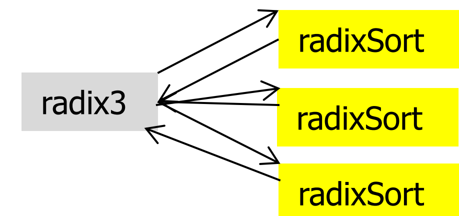
- Dels er denne gjennomgangen av vanlig Radix-sortering viktig for å forstå en senere parallell versjon.
- Dels viser den effekten vi akkurat så – tilfeldig oppslag i lageret med korte eller lange arrayer b[] i uttrykk som a[b[i]] kan gi uventede kjøretider.
- Ideen bak Radix er å sortere tall etter de ulike sifrene de består av og flytte de frem og tilbake mellom to arrayer a[] og b[] slik at de stadig blir sortert på ett siffer mer.

Om høyre, 'minst signifikant siffer først' Radix

- Radix-sortering, her vist 3 varianter:
  - R1: Radix-sortering med ett siffer
  - R2: Radix-sortering med to sifre
  - R3: Radix-sortering med tre sifre
- Alle tre består av to metoder:
  - radix1,radix2 eller radix3 som
    - Først regner ut max-verdien i a[]. Så regnes ut noen konstanter som antall bit i det/de sifrene a[] skal sorteres med.
    - Deretter kalles metoden radixSort for hvert siffer det skal sorteres etter







```java
static void radix1 (int [] a) {
    // 1 digit radixSort: a[]
    int max = a[0], numBit = 1, n = a.length;

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

    while (max >= (1<<numBit) )numBit++;

    int [] b = new int [n];
    radixSort( a,b, numBit, 0);
}

static void radix2(int [] a) {
    // 2 digit radixSort: a[]
    int max = a[0], numBit = 2, n =a.length;

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

    while (max >= (1<<numBit) )numBit++;

    int bit1 = numBit/2,
        bit2 = numBit-bit1;

    int[] b = new int [n];
    radixSort( a,b, bit1, 0);
    radixSort( a,b, bit2, bit1);
}
```

```java
static void radix3(int [] a) {
    // 3 digit radixSort: a[]
    int max = a[0], numBit = 3, n = a.length;

    // finn max i a[]
    for (int i = 0 ; i <= n ; i++)
        if (a[i] > max) max = a[i];

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

    int bit1 = numBit/3,
        bit2 = bit1,
        bit3 = numBit-(bit1+bit2);

    int [] b = new int [n];
    radixSort( a,b, bit1, 0);
    radixSort( a,b, bit2, bit1);
    radixSort( a,b, bit3, bit1+bit2);
}
```

```java
/** Sort a[] on one digit ; number of bits = maskLen, shiftet up 'shift' bits */
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=the frequency of each radix value in a
      for (int i = 0; i < n; i++)
         count[(a[i]>> shift) & mask]++;

     // b) Add up in 'count' - accumulated values
      for (int i = 0; i <= mask; i++) {
           j = count[i];
            count[i] = acumVal;
            acumVal += j;
       }

     // c) move numbers in sorted order a to b
      for (int i = 0; i < n; i++)
         b[count[(a[i]>>shift) & mask]++] = a[i];

    // d) copy back b to a
      for (int i = 0; i < n; i++)
         a[i] = b[i] ;

}// end radixSort
```

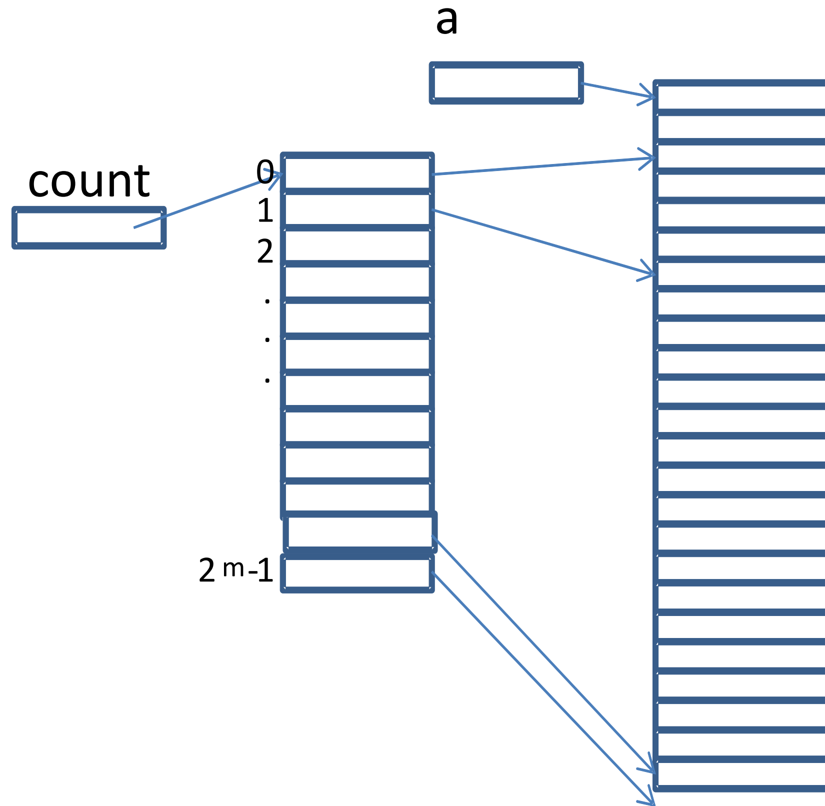
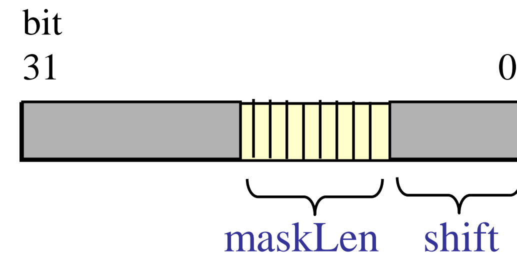


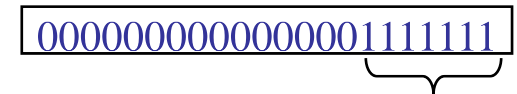
Figure 1. *The use of array count in any radix algorithm when sorting on a digit with* numbit *bits. The illustration is after sorting. We see that there are two elements in a[] with the value 0 on that digit, 4 elements with value 1,…,and 1 element with value* $2^{numbit} -1$.

Forklaring av: count[(a[i]>> shift) & mask]++; del1

- Tar det innenfra og ut; a[i]>> shift
  - Ethvert ord i lageret består a 0-ere og 1-ere (alt er binært)
  - Java har flere shift-operasjoner feks.:
    - a[i]>>b betyr: shift alle bit-ene i a[i] b antall plasser til høyre
      og fyll på med b stk 0-er på venstre del av a[i].
    - a[i]<<b betyr: shift alle bitene i a[i] b antall plasser til venstre
      og fyll på med b stk 0-er på høyre del av a[i].
    - De bit-ene som shiftes ut av a[i] går tapt i begge tilfeller.
    - a<<1 er det samme som a*2,
      a<<2 er det samme som a*4,
      a<<k er det samme som $a*2^k$

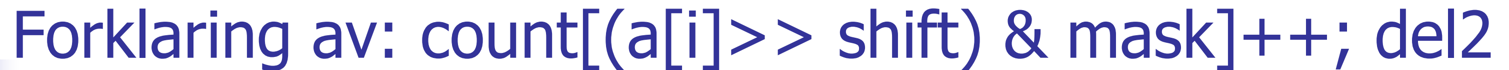
Ett element i a[]:





mask = maskLen 1’ere

Forklaring av: count[(a[i]>> shift) & mask]++; del2



- Java har flere bit-logiske operasjoner, for eksempel & (og):
  - a & b er et tall som har 1-ere der både a og b har en 1-ere, og resten er 0.
  - Eks: a&1 = et tall som er null over alt unntatt i bit_0 som har samme bit-verdi som bit_0 i a.
  - Vi kan betrakte b som en maske som plukker ut de bit-verdiene i a hvor b har 1-ere.
- Poenget er at: (a[i]>> shift) & mask er raskeste måte å finne hvilken verdi a[] har for et gitt siffer (sifferverdien) :
  - Først skifter vi bit-ene i a[i] ned slik at sifferet vi er interessert i ligger helt nederst til høyre.
  - Så &-er vi med en maske som bare har 1-ere for så mange bit vi har i det sifferet vi er interessert i nederst (og 0 ellers).
- count[(a[i]>> shift) & mask] er da det elementet i count[] som har samme indeks som sifferverdien i a[i].
- Det elementet i count[] øker vi så med 1 (++ operasjonen)

Eksempel (shift = 3 og mask =7) – vi vil ha 2dre siffer

- a[i] = 764 (i 8-tallsystemet)= 0000..000111110100
- a[i] >>3 =   0000000..000111110
- (a[i] >>3) & 0000000..000000111 = 00000000..00110 =6

Vi kan velge fritt hvor lange (antall bit) sifre og hvor mange sifre vi vil ha sortere på, men summen av antall bit i sifrene vi sorterer på må være større eller lik antall bit i max, det største tallet i a[].

Et godt valg er å ha en øvre grense på bit-lengden av et siffer – f.eks = 11, og da heller ta så mange sifre det trengs for å sortere a[] .

Stegene i en radix-sortering

radix1,2,eller3:

- Finn maks verdi i a[] og bestem antall sifre med mer.
  - FinnMax har vi parallellisert

radixSort (en gang for hvert siffer):

a) Tell opp hvor mange det er i a[] med de ulike mulige sifferverdiene på dette sifferet.
b) Adder sammen verdiene til en array som sier hvor vi skal flytte et element i a[] med en gitt sifferverdi.
c) Flytt elementene fra a[] til b[] slik at de minste verdier kommer øverst,..osv
d) Kopier b[] tilbake til a[]

Stegene a, b og c skal vi senere parallellisere (d kan fjernes)

Tidsforbruket i Radix – opptelling av operasjoner

- radix1,2,3:
  - Først gås hele a[] igjennom for å minne største verdi: **S n** operasjoner
    - Dette kan vi effektivt parallellisere (jfr. ukeoppgave 1)
- radixSort (S = sekvensiell, R = random)
  a. Les S i a[], les/skriv R i count[] **S n + R 2n**
  b. En les og skriv i count[] :
     a. for radix2 og3 – S lite antall
     b. for radix1: - **S 2n**
  c. S 2 les a[], R les/skriv count[],
     R skriv i b[] - **S 2n + R 3n**
  d. S 2n i a[] og b[] - **S 2n**

  SUM :

  radix1: Sekv: 8n + Random: 5n

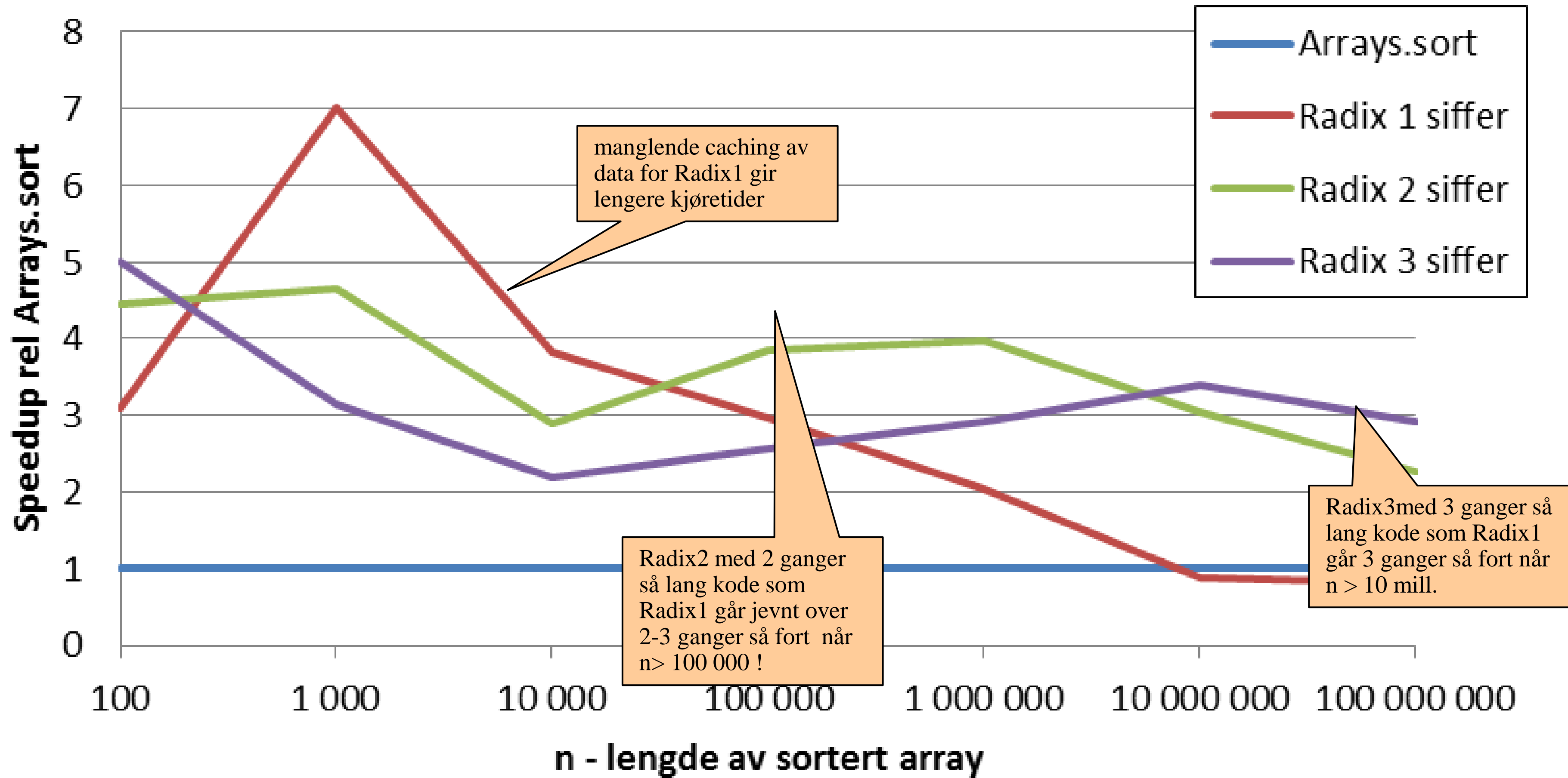
  radix2: Sekv: 11n + Random:10n,

  radix3: Sekv: 16n + Random:15n

Hva forventer vi av kjøretider

- Radix1 har færrest operasjoner og bør gå raskest
- Radix3 har klart 2-3 ganger så mange operasjoner og gå tilsvarende langsommere

Hva viser dette om cachen ?



Hva fikk vi av kjøretider?

- Alt for mye data til cachen (for stor count[]), ødela kjøretidene for Radix1.
- At 3x så lang kode i Radix3 kan gå 3x så fort – dvs, en speedup på 9 per kodelinje fordi Radix3s data (liten count[]) går fint inn i cache2 og cache 1.
- Radix2 og Radix3 var alltid betydelig raskere enn Quicksort (Arrays.sort())
- Konklusjon: Uten at vi tar hensyn til at datastrukturen i algoritmene våre passer godt inn i cache-hukommelsene, kan det gå svært dårlig.
- Konklusjon: Vi må senere prøve å parallellisere Radix2 – en utmerket og rask algoritme.

8) Kommentarer til Oblig 1

- Oblig 1 har ikke spesifisert hvordan dere skal finne de 50 første i parallell
- Men sagt at dere skal ta inspirasjon av hvordan dere løste Ukeoppgaven med FinnMax.
- I FinnMax hadde hver tråd en kopi av max som den lokalt fant maks på sin del av arrayen.
- Deretter, når alle trådene var ferdige med det, ble det etter en synkronisering funnet (sekvensielt) hvilken av de lokale maks-verdiene som da lå i en array, som var størst.
- Dette mønsteret for å løse en oppgave lar seg kopiere over i Oblig 1.
- N.B. Bruk ver2 av Oblig1.

Hva her vi sett på i Uke3

1. Presisering av pensum
2. Modell for hvordan vi programmerer
3. Ufravikelige regler om lesing og skriving på felles data.
4. Samtidig skriving med flere tråder i en array?
    1. Går det langsommere når aksessen er til naboelementer?
    2. Hvor fort kan JIT-kompilert kode gå?
5. Synlighetsproblemet
    1. Hvilke verdier ser ulike tråder som leser variable som en annen tråd skriver på?
6. Java har 'as-if sequential' semantikk
7. Effekten på eksekveringstid av cache
    1. Lese og skrive a[b[i]]=i;
    2. Radix-sortering sekvensiell, del 1 –
8. Kommentarer til Oblig 1