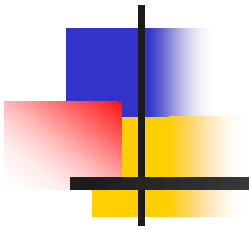


INF2440 Uke 12, v2014

Arne Maus
OMS,
Inst. for informatikk

Fra hjemmesida til INF2440:

To trykkfeil rettet i Oblig3 Rediger

1) Stegene i algoritmene ble i koden referert som a,b,c,c - skal selvsagt være: a,b,c,d.

2) En <= i koden er rettet til < , (ved finning av max i a[])

Ny Oblig3 her

2. apr. 2014 11:08

Underveisevaluering av INF2440 Rediger

Vennligst fyll ut underveiseevalueringa av INF2440 (få spørsmål)

31. mar. 2014 09:04

+ Ingen gruppe eller forelesninger 14.-20. april (påske)

Hva så vi på i Uke11

I) En del sluttkommentarer og optimalisering av Oblig2

II) Debugging av parallelle programmer

III) Et større problem – uløst problem siden 1742

- Vil du tjene 1 millioner $?
- Løs Goldbach's påstand (alternativt: motbevis den)
- Formulering og skisse av løsning av tre av Goldbachs problemer (Prog1, Prog2, Prog3)
- Parallellisering av disse (ukeoppgave neste uke)

IV) Oblig 3

Hva skal vi se på i Uke 12

I) Om «sifre» i Om Oblig 3

II) Om optimalisering av Oblig2 – nye tall med Java8

III) Java8

- Forbedringer
  - Feilen med tidtaking
  - Raskere Jit-kompilering
- Nytt
  - Metoder som parametre til metoder (closure, lamda)

IV) Litt mer om Goldbach – uløst problem siden 1742

- Løsning av to av Goldbachs problemer
- Parallellisering av Prog1 og Prog2

V) Alternative datastrukturer i Oblig3

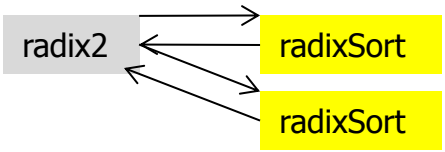
Om sifre i Oblig 3 – de er vilkårlig store

Et siffer i Radix-sortering er et bestemt antall bit, alt fra 1 til 31, **og vi velger selv hvor stort siffer vi sorterer med**.

Er et siffer numBit langt (= antall bit i sifferet), er de mulige sifferverdiene**:** **$0..2^{numBit}$ -1** (eks. er sifferet 8 bit langt, er sifferverdiene: 0,1,..,255).

Når vi sorterer med flere sifre, behøver ikke alle sifrene være like lange. Effektivitetsmessig lønner det seg at det er fra 8-13 bit i et siffer.

Et siffer i Oblig3 betyr **ikke** et desimalt siffer!




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

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

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

     // bestem antall bit i siffer1 og siffer2
     int bit1 = numBit/2,
         bit2 = numBit-bit1;

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

Ikke-optimal, men riktig kode

```
// Sekvensiell faktorisering av primtall
ArrayList<Long> factorize (long num) {
        ArrayList <Long> fakt = new ArrayList <Long>();
        int pCand =2;
        long pCand2=4L, rest= num;

        while (rest > 1 &&  pCand2 <= num) {
                while ( num % pCand == 0){
                        fakt.add((long) pCand);
                        rest /= pCand;
                }
                pCand = nextPrime(pCand);
                pCand2 =(long) pCand*pCand;
        }
        if (rest >1) fakt.add(rest);
        return fakt;
} // end factorize
```

Tips: Hva er grensene når vi har funnet første faktor

Om optimalisering av Oblig2

- Trikset i Oblig 2 var å dele ned øvre grense når vi fant en ny faktor som leses i den ytre løkka (hver gang vi leser nytt primtall) – rest og num er samme variabel !
  - I den sekvensielle løsningen (ca. 10x fortere)
  - I den parallelle løsningen
    - Da må vi ha en felles AtomicLong som leses hver gang i løkka, og som oppdateres inne i put (..)-metoden
- Vi får ulik speedup alt etter:
  - Java7 eller Java 8
  - Type maskin:
    - En 8(4) kjerner (Intel core i7 870 3.07GHz)
    - En 64(32) kjerner (Intel Xeon L7555 @ 1.87GHzXenon)

I) Speedup av Oblig2; java 7; 'optimal' kode; 8(4) kjerner :

	n	sekv(ms)	para(ms)	Speedup	PerElemSek	PerElemPar
EratosthesSil	2000000000	14131.98	7060.41	2.00		
Faktorisering	2000000000	24586.18	15845.43	1.55	245.8618	158.4543
Total tid	2000000000	38613.92	22938.85	1.68		
EratosthesSil	200000000	1078.20	268.08	4.02		
Faktorisering	200000000	5970.54	2817.03	2.12	59.7054	28.1703
Total tid	200000000	7050.84	3085.10	2.29		
EratosthesSil	20000000	68.80	17.71	3.88		
Faktorisering	20000000	683.02	407.84	1.67	6.8302	4.0784
Total tid	20000000	747.09	425.56	1.76		
EratosthesSil	2000000	6.02	1.92	3.14		
Faktorisering	2000000	146.56	214.39	0.68	1.4656	2.1439
Total tid	2000000	152.12	216.29	0.70		
EratosthesSil	200000	1.02	0.61	1.65		
Faktorisering	200000	67.47	163.95	0.41	0.6747	1.6395
Total tid	200000	68.53	164.62	0.42		
EratosthesSil	20000	0.06	0.10	0.55		
Faktorisering	20000	14.06	23.98	0.59	0.1406	0.2398
Total tid	20000	14.12	24.20	0.58		
EratosthesSil	2000	0.01	0.27	0.05		
Faktorisering	2000	23.20	46.89	0.49	0.2320	0.4689
Total tid	2000	23.24	47.25	0.49		

Ib) Om speedup av Oblig2; nå java8; 'optimal' kode ; 8(4) kjerner :

n		sekv(ms)	para(ms)	Speedup	PerElemSek	PerElemPar
EratosthesSil	2000000000	12089.92	6920.98	1.75		
Faktorisering	2000000000	17541.09	11076.74	1.58	175.4109	110.7674
Total tid	2000000000	29631.02	17965.84	1.65		
EratosthesSil	200000000	892.37	222.10	4.02		
Faktorisering	200000000	4462.84	1939.20	2.30	44.6284	19.3920
Total tid	200000000	5369.50	2175.76	2.47		
EratosthesSil	20000000	53.34	15.66	3.41		
Faktorisering	20000000	518.65	368.67	1.41	5.1865	3.6867
Total tid	20000000	580.31	384.33	1.51		
EratosthesSil	2000000	5.58	1.58	3.54		
Faktorisering	2000000	123.29	190.25	0.65	1.2329	1.9025
Total tid	2000000	128.62	197.85	0.65		
EratosthesSil	200000	1.15	0.57	2.02		
Faktorisering	200000	53.84	155.80	0.35	0.5384	1.5580
Total tid	200000	54.70	156.45	0.35		
EratosthesSil	20000	0.51	0.63	0.81		
Faktorisering	20000	42.73	121.64	0.35	0.4273	1.2164
Total tid	20000	43.44	122.27	0.36		
EratosthesSil	2000	0.40	0.53	0.74		
Faktorisering	2000	35.58	80.59	0.44	0.3558	0.8059
Total tid	2000	35.98	81.12	0.44		

Ib)Kjøring på en server med 64(32) kjerner 1.8 GHz; nå java7

n	sekv(ms)	para(ms)	Speedup	PerElemSek	PerElemPar	
EratosthesSil	2000000000	22668.56	843.83	26.86		
Faktorisering	2000000000	25555.76	10052.66	2.54	255.5576	100.5266
Total tid	2000000000	48272.10	10936.69	4.41		
EratosthesSil	200000000	1218.54	55.52	21.95		
Faktorisering	200000000	6302.25	2601.45	2.42	63.0225	26.0145
Total tid	200000000	7520.79	2654.02	2.83		
EratosthesSil	20000000	151.81	8.13	18.67		
Faktorisering	20000000	1078.74	1228.20	0.88	10.7874	12.2820
Total tid	20000000	1230.56	1237.31	0.99		
EratosthesSil	2000000	12.96	3.99	3.25		
Faktorisering	2000000	244.58	1192.46	0.21	2.4458	11.9246
Total tid	2000000	257.54	1197.00	0.22		
EratosthesSil	200000	1.26	3.94	0.32		
Faktorisering	200000	30.46	1178.90	0.03	0.3046	11.7890
Total tid	200000	31.72	1183.05	0.03		
EratosthesSil	20000	0.21	3.74	0.06		
Faktorisering	20000	10.30	1176.02	0.01	0.1030	11.7602
Total tid	20000	10.53	1179.76	0.01		
EratosthesSil	2000	0.11	3.10	0.04		
Faktorisering	2000	5.82	1117.49	0.01	0.0582	11.1749
Total tid	2000	5.94	1120.50	0.01		

Speedup er høyst forskjellig (java 7):

- Fra en 64(32) kjerners maskin:

```
EratosthesSil 2000000000  22668.56    843.83     26.86
Faktorisering 2000000000  25555.76  10052.66      2.54    255.5576    100.5266
Total tid     2000000000  48272.10  10936.69      4.41
```

- Fra en 8(4) kjerners maskin

```
EratosthesSil 2000000000  14131.98   7060.41      2.00
Faktorisering 2000000000  24586.18  15845.43      1.55    245.8618    158.4543
Total tid     2000000000  38613.92  22938.85      1.68
```

- Speedup på Eratosthenes Sil er nesten lineær med antall kjerner, mens Faktoriseringen har lite hjelp av 64 kjerner.
  - Erothosthenes Sil er 'perfekt' parallellisert; lik belastning på alle tråder
  - Erothosthenes Sil lage mange 'små' delproblmer som passer bedre i cachen
  - I Faktoriseringa er det i hovedsak tråd-0 som gjør alt arbeidet, mens de andre trådene terminerer raskt (unntatt når vi faktoriserer et primtall)
- Kan oppdelingen av Faktoriseringa gjøres bedre?
  - Kan vi bedre både gjennomsnittet og den største tida en faktorisering tar

Java 8 – endringer som er viktige for oss

- Forbedringer
  - Feilen med tidtaking er borte
  - Raskere Jit-kompilering
- Nytt
  - Metoder som parametre til metoder (closure, lamda)

```
import java.util.*;
import easyIO.*;
class FinnSum{
  public static void main(String[] args){
        int len = new In().inInt();
        FinnSum fs =new FinnSum();

        for(int k = 0; k < 20; k++){
              int[] arr = new int[len];
              Random r = new Random();
              for(int i = 0; i < arr.length; i++){
                    arr[i] = r.nextInt(len-1);
              }
             long start  = System.nanoTime();
             long sum = fs.summer(arr);
             long timeTaken = System.nanoTime() - start ;

            System.out.println(
               Format.align(k+1,2)+ ") s="+
             //  Format.align(sum,9)+"  paa:" +
                Format.align(timeTaken,10) + " nanosek");
    }} // end main

   long summer(int [] arr){
                long sum = 0;
                for(int i = 0; i < arr.length; i++)
                              sum += arr[i];
               return sum;
   } } // end summer
```

Denne koden feilet tidligere og ga tider =0

I Java8 er det rettet!

```
M:\INF2440Para\JavaTidFeil>java FinnSumFeil
8000
 1), sum      8000 tall:  400006.0 nanosek
 2), sum      8000 tall:  390557.0 nanosek
 3), sum      8000 tall:  397906.0 nanosek
 4), sum      8000 tall:  390907.0 nanosek
 5), sum      8000 tall:  390907.0 nanosek
 6), sum      8000 tall:  269470.0 nanosek
 7), sum      8000 tall:  270170.0 nanosek
 8), sum      8000 tall:  282418.0 nanosek
 9), sum      8000 tall:  307966.0 nanosek
10), sum      8000 tall:  308665.0 nanosek
11), sum      8000 tall:   47595.0 nanosek
12), sum      8000 tall:   47595.0 nanosek
13), sum      8000 tall:   47594.0 nanosek
14), sum      8000 tall:   68242.0 nanosek
15), sum      8000 tall:   50044.0 nanosek
16), sum      8000 tall:   47594.0 nanosek
17), sum      8000 tall:   47595.0 nanosek
18), sum      8000 tall:   47595.0 nanosek
19), sum      8000 tall:   47595.0 nanosek
20), sum      8000 tall:  130186.0 nanosek
21), sum      8000 tall:   48645.0 nanosek
22), sum      8000 tall:   48644.0 nanosek
23), sum      8000 tall:    6299.0 nanosek
24), sum      8000 tall:    5949.0 nanosek
25), sum      8000 tall:    5949.0 nanosek
26), sum      8000 tall:    5949.0 nanosek
27), sum      8000 tall:    3500.0 nanosek
28), sum      8000 tall:   12948.0 nanosek
29), sum      8000 tall:    2449.0 nanosek
30), sum      8000 tall:    3150.0 nanosek
31), sum      8000 tall:    2800.0 nanosek
32), sum      8000 tall:    2800.0 nanosek
33), sum      8000 tall:    2450.0 nanosek
34), sum      8000 tall:    2450.0 nanosek
35), sum      8000 tall:    2449.0 nanosek
```

Bytekode

maskinkode

Opt. nr.2

Opt. nr.3

Opt. nr.4

Kommentarer:

- Her ser vi 4 optimaliserings – steg, mer enn 160x fortere.
- Dette er et pent eksemplaret av tidtaking
- Mange andre merkeligere

```
M:\INF2440Para\JavaTidFeil>java -Xmx4000m FinnSumFeil
8000

SUM av: 10000 tall
 1), s= 49991934 paa   496945.0 nanosek
 2), s= 50007764 paa   488545.0 nanosek
 3), s= 50205805 paa   488896.0 nanosek
 4), s= 49974158 paa   496244.0 nanosek
 5), s= 50212042 paa   488196.0 nanosek
 6), s= 49888832 paa   488895.0 nanosek
 7), s= 49747261 paa   491346.0 nanosek
 8), s= 50006484 paa   559588.0 nanosek
 9), s= 50289938 paa   557838.0 nanosek
10), s= 50026668 paa    45495.0 nanosek
11), s= 50213253 paa    55994.0 nanosek
12), s= 49591428 paa    45495.0 nanosek
13), s= 49922815 paa    45495.0 nanosek
14), s= 49804343 paa    45495.0 nanosek
15), s= 50052316 paa    45495.0 nanosek
16), s= 50276803 paa   115138.0 nanosek
17), s= 50110195 paa    38846.0 nanosek
18), s= 49741273 paa     5950.0 nanosek
19), s= 49893873 paa     5600.0 nanosek
20), s= 50361802 paa     5949.0 nanosek
21), s= 49689512 paa     5599.0 nanosek
22), s= 50138174 paa     5600.0 nanosek
23), s= 50009243 paa     5600.0 nanosek
24), s= 49525075 paa     5949.0 nanosek
25), s= 49721224 paa     5600.0 nanosek
26), s= 49585537 paa     5600.0 nanosek
27), s= 49957238 paa     3850.0 nanosek
28), s= 49774628 paa     3499.0 nanosek
29), s= 50114074 paa     3849.0 nanosek
30), s= 49715015 paa    11899.0 nanosek
31), s= 50628212 paa     3149.0 nanosek
32), s= 50148180 paa     3150.0 nanosek
33), s= 50216877 paa     3149.0 nanosek
34), s= 50168993 paa     3499.0 nanosek
35), s= 50398927 paa     3150.0 nanosek
Speedup factor:   177.00
```

Bytekode

maskinkode

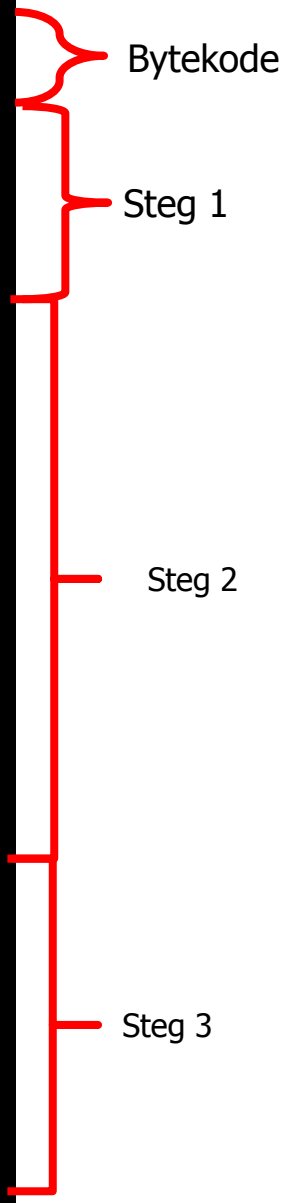
Opt. nr.2

Opt. nr.3

Kommentarer:

- Her ser vi 4 optimaliserings – steg, mer enn 170x fortere.
- Dette er det peneste eksemplaret av tidtaking
- Mange andre er merkeligere

```
M:\INF2440Para\JavaTidFeil>java -Xint
FinnSumFeil
10000

SUM av: 10000 tall
 1), s= 50423981 paa   302716.0 nanosek
 2), s= 49758776 paa   297117.0 nanosek
 3), s= 49894018 paa   297117.0 nanosek
 4), s= 50112273 paa   227125.0 nanosek
 5), s= 50247911 paa   227124.0 nanosek
 6), s= 50152687 paa   227125.0 nanosek
 7), s= 49993296 paa   227125.0 nanosek
 8), s= 49897281 paa   227474.0 nanosek
 9), s= 50216265 paa   227125.0 nanosek
10), s= 49931382 paa   227475.0 nanosek
11), s= 49678162 paa   175680.0 nanosek
12), s= 50258356 paa   175681.0 nanosek
13), s= 49854211 paa   175681.0 nanosek
14), s= 49766901 paa   175330.0 nanosek
15), s= 50558878 paa   175680.0 nanosek
16), s= 49956008 paa   175681.0 nanosek
17), s= 49996223 paa   175680.0 nanosek
18), s= 49886799 paa   176030.0 nanosek
19), s= 49920358 paa   175331.0 nanosek
20), s= 50062358 paa   175681.0 nanosek
21), s= 49789176 paa   175680.0 nanosek
22), s= 50144515 paa   175680.0 nanosek
23), s= 50272743 paa   175681.0 nanosek
24), s= 49880362 paa   175681.0 nanosek
25), s= 49988799 paa   160632.0 nanosek
26), s= 49866680 paa   143135.0 nanosek
27), s= 49807130 paa   143134.0 nanosek
28), s= 49676267 paa   143484.0 nanosek
29), s= 49800279 paa   159933.0 nanosek
30), s= 50115547 paa   143134.0 nanosek
31), s= 50260231 paa   143134.0 nanosek
32), s= 50521301 paa   143134.0 nanosek
33), s= 49877673 paa   143134.0 nanosek
34), s= 49752118 paa   142784.0 nanosek
35), s= 50059215 paa   150484.0 nanosek
Speedup factor:     2.00
```



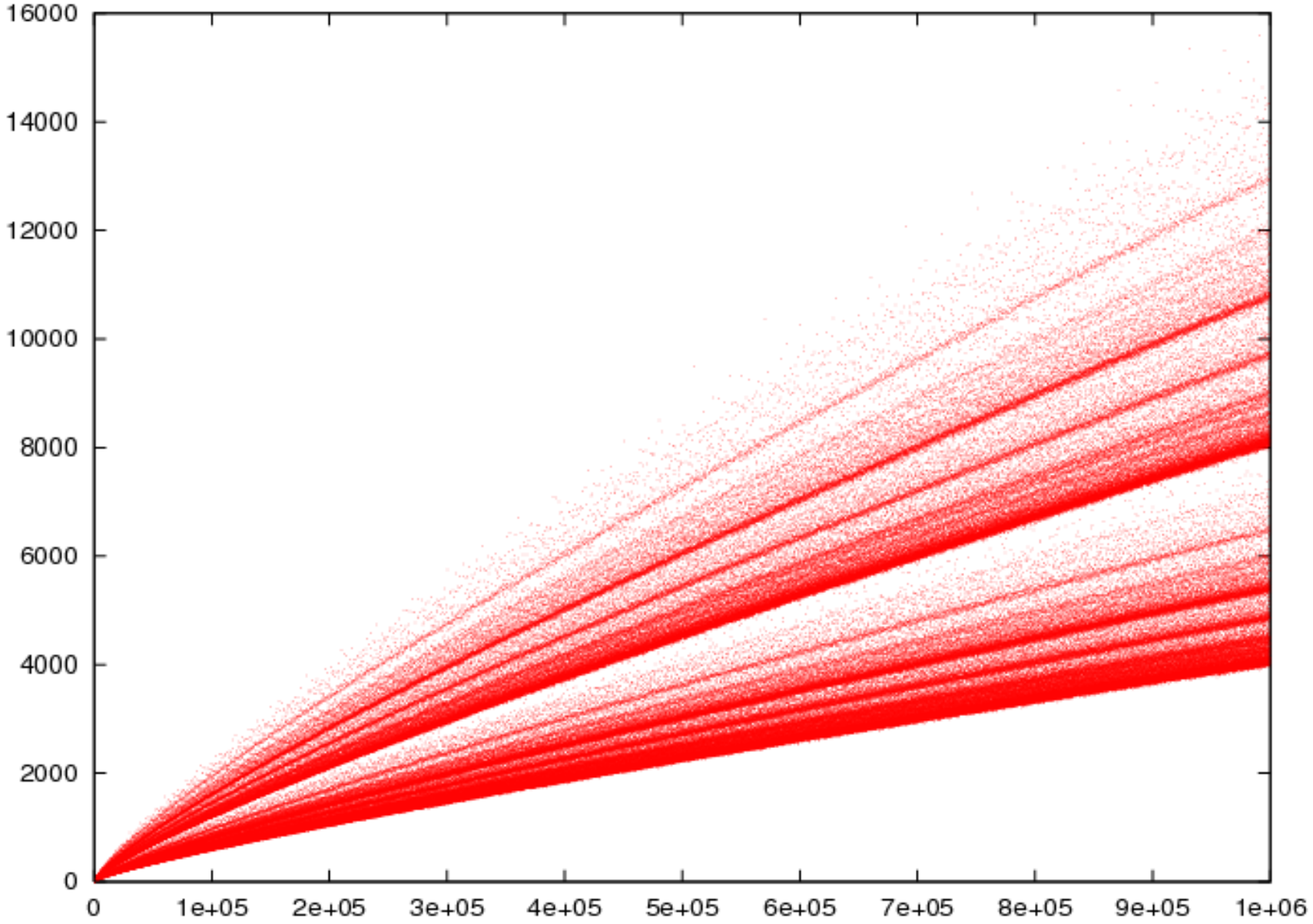
Kommentarer:

- Her har vi kjørt uten JIT-kompilering
- Likevel får vi en speedup faktor på 2
- Forbedringene kommer i 3 steg, hva er de ?
- Er det selve JVM (dvs. java) som JIT-kompileres?
- Eller er det at bytekoden og JVM kommer bedre opp i cache-systemet ?

III) Christian Goldbachs påstand i 1742:

- Alle partall m =n+n > 4 kan skrives som en sum av to primtall som er oddetall.
  - m = p1+p2 (p1 $\leq$ p2)
  - Eks: 6 = 3+3, 14 = 7+7 og 14 = 11+3,….
- Antall slike ulike summer av to primtall for en gitt m kaller vi G(m).
- Bevis at G(m) > 0 $\forall$ m.

G(n) for alle tall < 1 mill – varierer mye, men har en skarp nedre grense som vi ikke greier å vise er > 0 !



Hvordan vise Goldbach med et program

- En riking i USA har utlovet $1mill for løsning på 10 matematiske problemer som lenge var uløst.
  - Goldbach er en av disse problemene, og premien er ikke hentet.
  - To måter å vise Goldbach:
    - Komme med et matematisk bevis for at den er sann.
    - Komme med et mot-eksempel: Partallet M som ikke lar seg skrive som summen av to primtall.

Hvordan løser vi dette ? Første sekvensielt

- Hvilket program skal vi lage?
  - **a)** **Prog1**: Som for alle tall m=6,8,….,n finner minst én slik sum m= p1+p2
  - **b)** **Prog2**: Som regner ut G(m) for m=6,8,….,n
  - **c)** **Prog3**: Som tester noen tall > 4×10^{18} om Goldbach gjelder også for disse tallene
    (sette ny rekord – ikke testet tidligere, kanskje finne ett tall som ikke lar seg skrive som summen av to primtall.)

Forskjellen på Prog1/Prog2 og Prog3 er at i Prog1 og Prog2 antar vi at vi vet alle primtall < m, det trenger vi ikke i Prog3 (som da blir en god del langsommere).

Prog1 og Prog2 bruker 'bare' Eratosthenes Sil fra Oblig2.

Prog3 bruker både Eratosthenes Sil og faktoriseringa fra Oblig2.

Skisse av Prog1 : Finn minst én slik sum m=p1+p2, ∀ m <n, m partall, p1 ≤ p2; sekvensielt program

```
<Les inn: n>
<Lag e= Eratosthens Sil(n)>

for (int m = 6 ; m < n; m+=2) {
    // for neste tall m, prøv alle primtall p1 ≤ m/2
     for (int p1 = 3, p1 <= m/2; e.nextPrime(p1){
         if ( e.isPrime(m-p1)) {
             //  System.out.println( m+" = "+ p1 + " + "+ (m-p1) );
             break;
         }
     } // end p1
      if (p1 > m/2) System.out.println(
                    " REGNEFEIL: (Goldbach bevist for n < 4*10**18:"
                     + m + " kan ikke skrives som summen av to primtall ");
} // end m
```

Skisse av Prog2 : Finn G(m) antall slike summmer: m = p1+p2, $\forall$ m <n, m partall, p1 $\leq$ p2; sekvensielt program

```
<Les inn: n>
<Lag e= Eratosthens Sil(n)>
int Gm;

for (int m = 6 ; m < n; m+=2) {
     Gm = 0;
    // for neste tall m, prøv alle primtall p1 ≤ m/2
     for (int p1 = 3, p1 <= m/2; e.nextPrime(p1){
         if ( e.isPrime(m-p1)) { Gm++; }
     } // end p1
     if (Gm == 0)
            println(" REGNEFEIL: (Goldbach bevist for n < 4*10**18):"
                        + m + " kan ikke skrives som summen av to primtall ");
      else println(" Antall Goldbachsummer i "+m+" er:"+Gm);
} // end m
```

Prog1: Hvordan parallellisere å finne én sum ?

- Vi har en dobbelt løkke (m og p1)
- Parallelliser den innerste løkka:
  - Deler opp primtallene og lar ulike tråder summere med ulike p1
  - Sannsynligvis uklokt?
- Parallelliserer den ytterste løkka:
  - Hvis vi lar hver tråd få 1/k-del av tallene å finne p1 for.
  - Med n= 2mrd. får hver tråd 250 mill. tall å sjekke.

```
<Les inn: n>
<Lag e= Eratosthens Sil(n)>

for (int m = 6 ; m < n; m+=2) {
   // for neste tall m, prøv alle primtall p1 ≤ m/2
    for (int p1 = 3, p1 <= m/2; e.nextPrime(p1){
        if ( e.isPrime(m-p1)) {
              break;  // funnet sum
        }
    } // end p1
     if (p1 > m/2) System.out.println(" REGNEFEIL for "+
m);
} // end m
```

  - Litt ulik belastning på trådene, da større tall må lete lenger etter den første p1 hvor m-p1 er primtall.
- Parallelliserer ved å lage en rekursjon på ytterste løkke.
  - Kan bruke PRP
  - Mye kortere kode

Konklusjon: Forsøker først å parallellisere den ytterste løkka ?

Viktig poeng: Hvilke rutiner bruker vi?

- Vi har sekvensielle og parallelle versjoner av Eratosthenes Sil og Faktorisering. Hvilken versjon bruker vi i Prog1,2,3?

```
<Les inn: antall>  /*  Prog3 –skisse */
<Lag e= Eratosthens Sil(2147480000)> // nær øvre grense for int

for (long m = 4*10^18 ; m < 4*10^18+antall; m+=2) {
    // for neste tall m, prøv alle primtall p1 ≤ m/2
     for (int p1 = 3, p1 <= m/2; e.nextPrime(p1){
         if ( e.faktorisering (m-p1).size() == 1 )) {
              //  Funnet Goldbach sum
              break;
         }
     } // end p1
      if (p1 > m/2) System.out.println( " BINGO: Funnet $1. mill:"
                    + m + " kan ikke skrives som summen av to primtall ");
} // end m
```

- Kan vi parallellisere inne i en parallellisering ?
- Bør vi evt. gjøre det – og hvorfor / hvorfor ikke ?

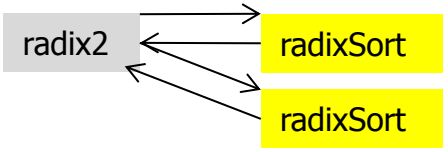
Prog 1,

finn én sum:	n	sekv(ms)	para(ms)	Speedup	SekElem(us)	ParaElem(us)
EratosthesSil	2000000000	14872.89	7386.36	2.01		
Goldbach Sum	2000000000	131447.23	31222.86	4.21	0.06572	0.01561
Total tid	2000000000	146350.58	38813.12	3.77		
EratosthesSil	200000000	1156.82	336.60	3.44		
Goldbach Sum	200000000	11224.35	2646.68	4.24	0.05612	0.01323
Total tid	200000000	12395.19	2990.45	4.14		
EratosthesSil	20000000	69.13	29.94	2.31		
Goldbach Sum	20000000	946.12	252.25	3.75	0.04731	0.01261
Total tid	20000000	1015.25	273.34	3.71		
EratosthesSil	2000000	6.26	1.92	3.27		
Goldbach Sum	2000000	81.11	18.89	4.29	0.04056	0.00945
Total tid	2000000	87.38	20.61	4.24		
EratosthesSil	200000	0.56	0.40	1.38		
Goldbach Sum	200000	6.84	2.90	2.36	0.03421	0.01452
Total tid	200000	7.42	3.30	2.25		
EratosthesSil	20000	0.06	0.43	0.13		
Goldbach Sum	20000	0.84	1.73	0.48	0.04187	0.08665
Total tid	20000	0.89	2.04	0.44		
EratosthesSil	2000	0.01	0.36	0.03		
Goldbach Sum	2000	0.35	1.81	0.19	0.17411	0.90483
Total tid	2000	0.36	2.17	0.17		

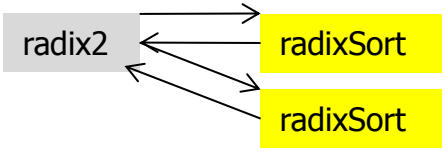
Oblig 3

- Parallelliser Radix-sortering med to sifre
- Skriv rapport om speedup for n= 1000, 10 000, 100 000, 1 mill., 10 mill og 100 mill.
- Radix består av to metoder, begge skal parallelliseres.
- Den første har et parallelt steg
  a) finn max(a[]) – løst tidligere
- Den andre har tre steg – løses i parallell effektivt:
  b) tell hvor mange det er av hvert sifferverdi i a[] i count[]
  c) legg sammen verdiene i count[] til 'pekere til b[]
  d) flytt tallene fra a[] til b[]
- Steg b) er løst i ukeoppgave (hvor bla. hver tråd har sin kopi av count[])

Den første av to algoritmer som sekvensiell 2-siffer Radix består av.




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

     //a)  finn max verdi i a[]
      for (int i = 1 ; i < n ; i++)          // KORREKSJON : i <n
           if (a[i] > max) max = a[i];

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

      // bestem antall bit i siffer1 og siffer2
      int bit1 = numBit/2,
          bit2 = numBit-bit1;

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

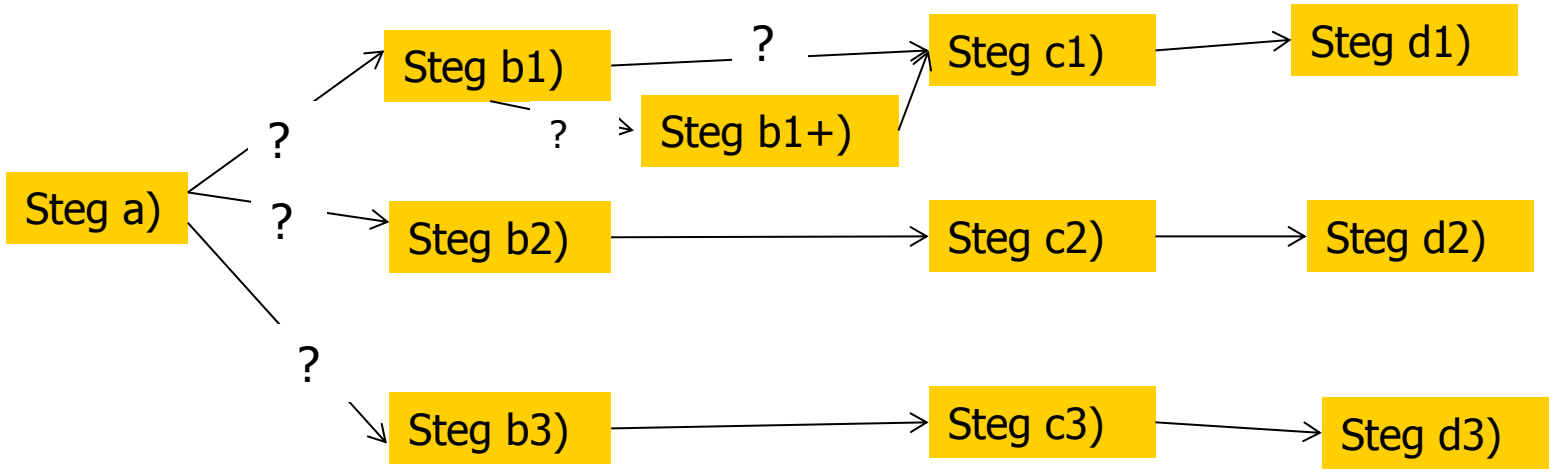



```
/** Sort a[] on one digit ; number of bits = maskLen, shiftet up 'shift' bits */
static void radixSort ( int [] a, int [] b, int maskLen, int shift){
      int  acumVal = 0, j, n = a.length;
      int mask = (1<<maskLen) -1;
      int [] count = new int [mask+1];

     // b) count=the frequency of each radix value in a
      for (int i = 0; i < n; i++)
         count[(a[i]>> shift) & mask]++;

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

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

}// end radixSort
```

Datastrukturer og grep i Oblig3 (hvordan parallellisere)

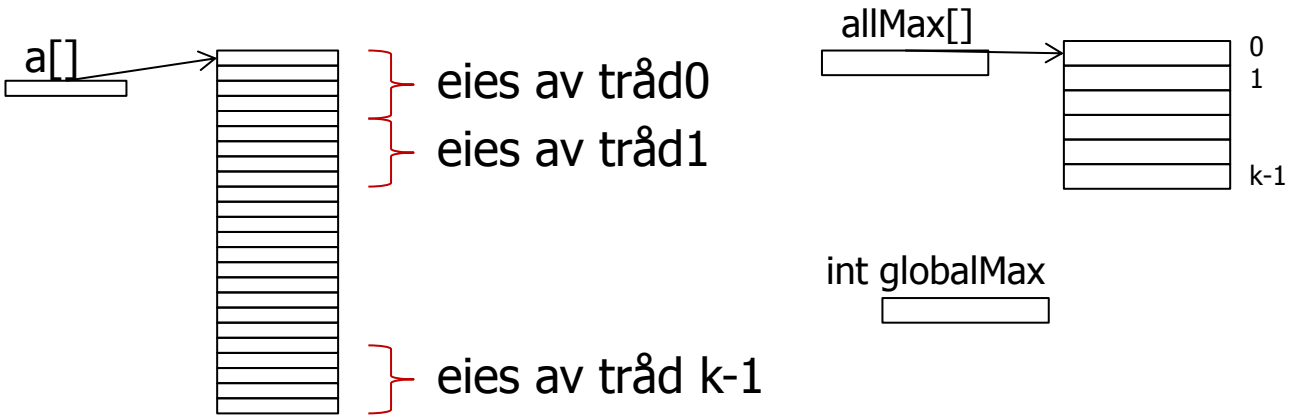
- Radix består av 4 steg (løkker) – flere valg for datastruktur og oppdeling ved parallellisering
  - a) finn max verdi i a[]
  - b) count= oppttelling av ulike sifferverdier i a[]
  - c) Summér opp i count[] akkumulerte verdier (pekere)
  - d) Flytt elementer fra a[] til b[] etter innholdet i count[]
- Generelt skal vi se følgende teknikker med k tråder:
  - Dele opp data i a [] i k like deler
  - Dele opp verdiene i a[] i k like deler => dele opp count[] i k like deler
  - Kopiere delte data til hver tråd – her lokal count[]-kopi en eller to ganger
  - Innføre ekstra datastrukturer som ikke er i den sekvensielle løsningen

Roadmap – oversikt over drøftelsen av Oblig3 – flere alternativer på steg b og senere endringer i c og d



FinnMax Tell sifferverdier lag pekere flytt a[] til b[]

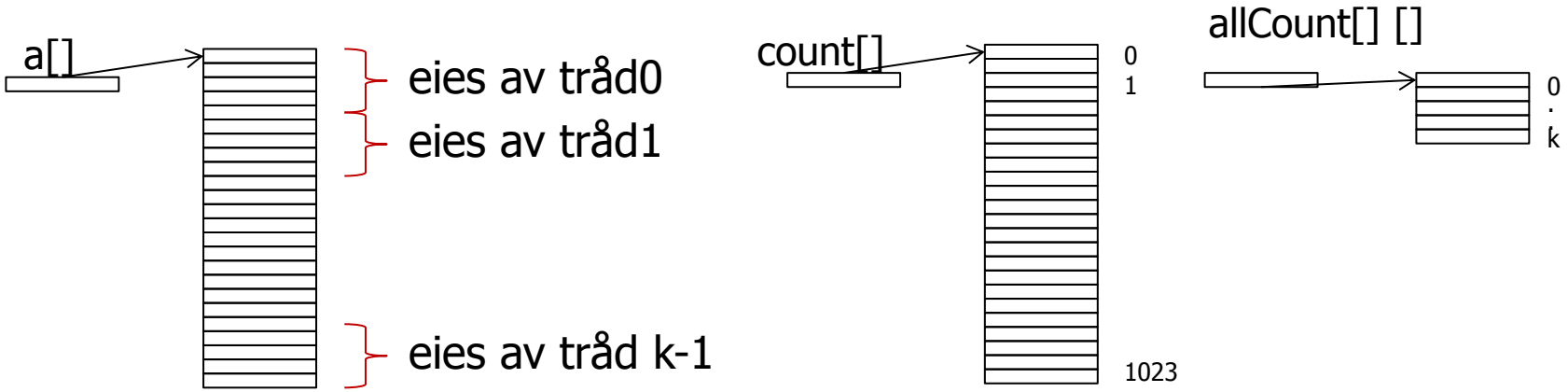
a) finn max verdi i a[]



- Tråd-i finner max i sin del av a[] og legger svaret i allMax[i]
- <sync på en CyclicBarrier cb>
- Nå har alle trådene sin max i allMax[] – valg nå:
  - Skal en av trådene (f.eks. tråd-0) finne svaret og legge det i en felles globalMax (mens de andre trådene venter i så fall nok en <sync på en CyclicBarrier cb>) ?
  - Skal alle trådene hver regne ut en lokal globalMax (de får vel samme svar?) og fortsette direkte til steg b)

b) count= oppttelling av ulike sifferverdier i a[]

Anta at det er 10 bit i et siffer – dvs. 1024 mulige sifferverdier



- Skal:
  1. Hver tråd ha en kopi av count[]
  2. Eller skal count være en AtomicIntegerArray
  3. Eller skal de ulike trådene gå gjennom hele a[] og tråd-0 bare ta de små verdiene, tråd-1 de nest minste verdien,..(dvs: dele verdiene mellom trådene)

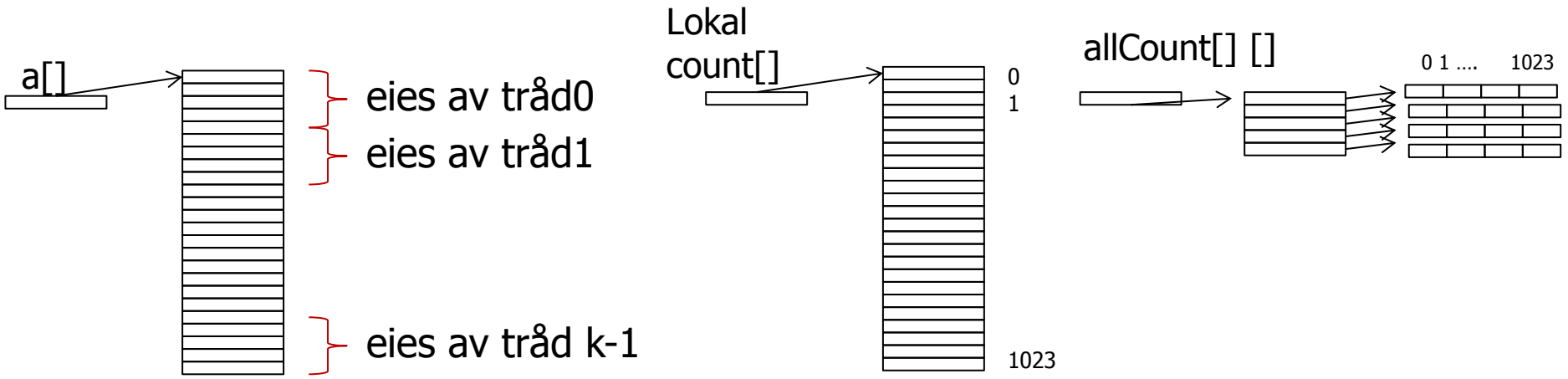
Om delvis deklarasjoner av arrayer

int allCount[] []; - da lages ?

int allCount[] [] = new int [k][]; - da lages ?

int [] count = new int[10] ;
allCount[k] [0] = x;

Løsning b1) – lokale count[] i hver tråd som etter opptelling settes inn i allCount[] []

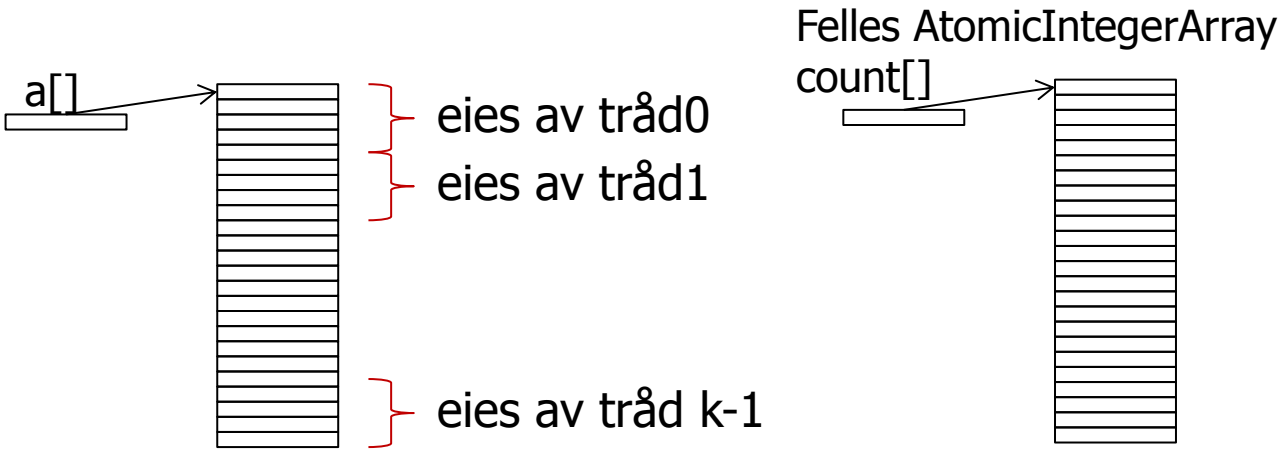


- Hver tråd-i teller opp de ulike sifferverdien i sin del av a[] opp i sin lokale count[] som så hektes inn i allCount[] []
- <sync på en CyclicBarrier cb>
- Nå er hele opptellingen av a[] i de k count[]-ene som henger i allCount[] []
- Denne løsningen trenger (kanskje) et tillegg b1-b:
  - Summering av verdiene i allCount[][] til en felles globalCount[]

b1+) : Summering av verdiene i allCount[][] til en felles: globalCount[]

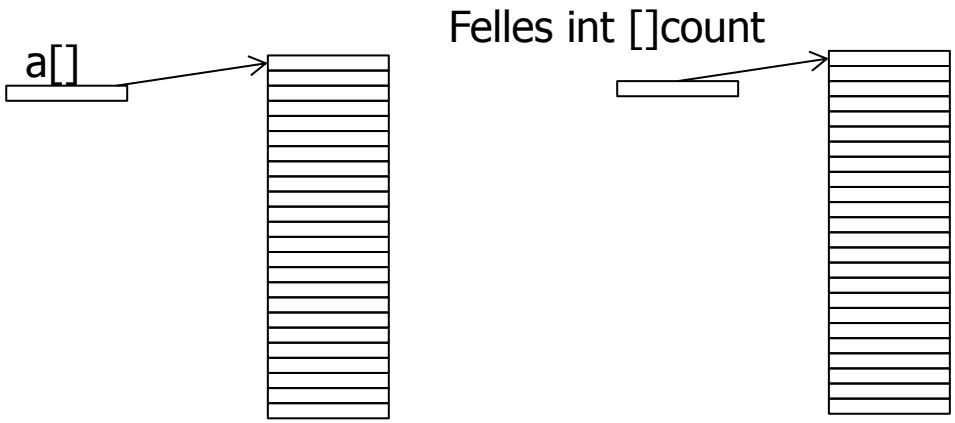
- Deler opp de mulige sifferverdiene (1024) på de k trådene slik at tråd-0 for de 1024/k minste, tråd-1 de 1024/k nest minste,…
- Tråd-i summerer sine verdier (sine kolonner) 'på tvers' i de k count[] – arrayene som henger i allCount[] [], inn i globalCount[]
- Bør allCount[][] transponeres før summering (mer cache-vennlig) ??
- <sync på en CyclicBarrier cb>
- Da er globalCount[] fullt oppdatert
- Spørsmål:
  - Trenger vi globalCount[] , eller holder det med allCount[][]?
  - Svar : Avhenger av neste steg c)

Løsning b2) – Eller skal count[] være en AtomicIntegerArray?



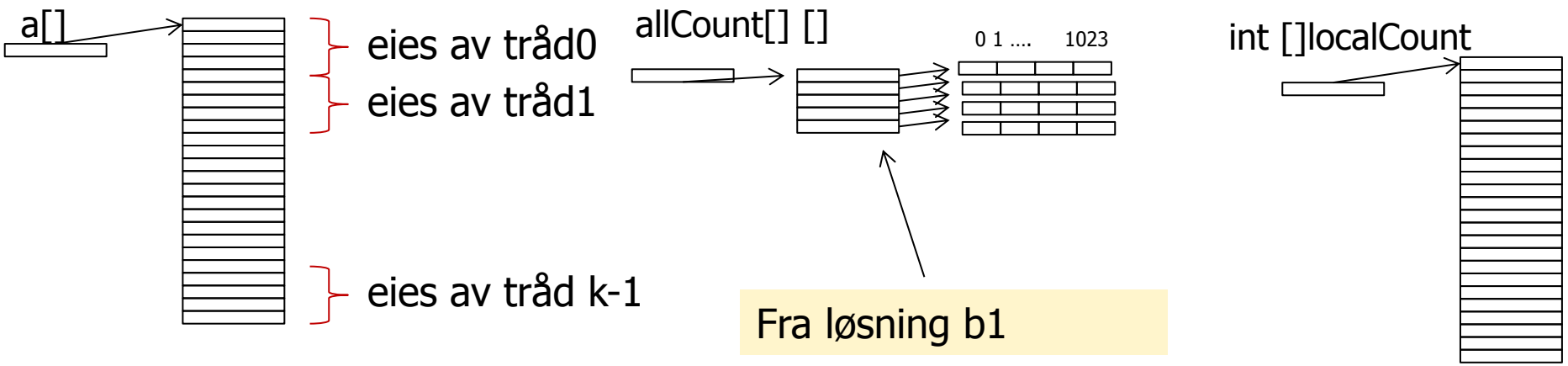
- Alle tråder oppdaterer AtomicIntegerArray count[] samtidig uten fare for synkroniserings-problemer med sifferverdiene fra sin del av a[] pga synkroniseringa av AtomicIntegerArray-elementene
- <sync på en CyclicBarrier cb>
- Spørsmål:
  - Hvor mange synkroniseringer trenger vi med denne løsningen

Løsning b3) – Eller skal de ulike trådene gå gjennom hele a[] og tråd-0 bare ta de n/k minste verdiene, tråd-1 de n/k nest minste verdiene,.. (dvs. dele verdiene mellom trådene)



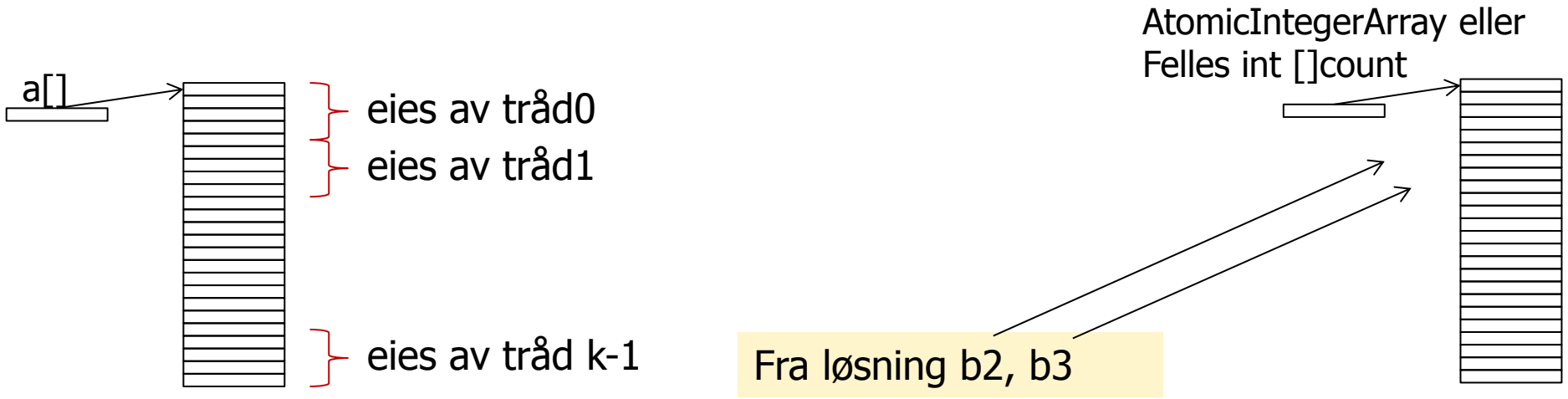
- Tråd-i oppdaterer count[] bare med **de verdiene** som tråd-i eier uten fare for synkroniserings-problemer med sifferverdiene fra hele a[] – ingen andre skriver slike sifferverdier.
- <sync på en CyclicBarrier cb>
- Spørsmål:
  - Hvor mange synkroniseringer trenger vi med denne løsningen
  - Hvor mye av a[] leser hver tråd

c1) **Gitt b1**: Summér opp i count[] akkumulerte verdier (pekere)



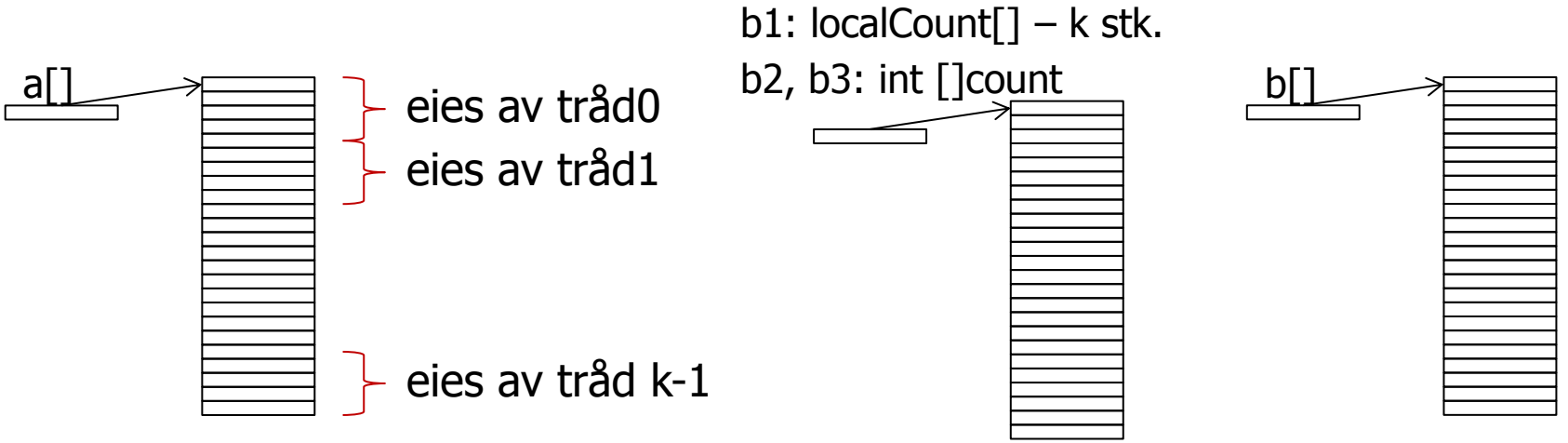
- Vi trenger et prinsipp for at hver tråd finner akkurat hvor den skal plassere sin del av a[].
  - Hver tråd får **nok en kopi** av count[]: localCount[] – initielt tom (=0)
  - Tråd-i sin localCount[t] er lik summen av alle tråder sin opptelling i allCount[r][s] for alle r og s<t + sum av allCount[r][t] r < i når s == t
  - Vitsen er at før tråd-i plasserer sine sifferverdier s, må det først gjøres plass til alle lavere sifferverdier + de sifferverdiene i tråder med indeks mindre enn i.
- <sync på en CyclicBarrier cb>

c2 og c3) Summér opp i count[] akkumulerte verdier (pekere)



- I løsning b2 og b3 har vi samme situasjon som sekvensiell løsning. Kan parallelliseres som følger :Deler verdiene i count[] mellom de k trådene som før. Alle summerer sine verdier og legger de i en separat array delSum[k].
- <sync på en CyclicBarrier cb>
- Deretter justerer du dine plasser i count[] med summen av delsummene i delSum[k] fra tråder med mindre indeks enn deg.
- Spørsmål:
  - Hvor lang tid tar dette kontra at tråd-0 gjør hele jobben og de andre venter.

d1,2 og 3) Flytt elementer fra a[] til b[] etter innholdet i count[]



- Løsning b1: Nå kan alle trådene i full parallell kopiere fordi hver localCount[] peker inn i ulike plasser i b[].
- Løsning b2: Alle trådene kan i full parallell kopiere over fra a[] til b[] fordi hver gang blir synkronisering foretatt av AtomicIntegerArray.
- Løsning b3: Nå kan alle trådene i full parallell flytte elementer fra a[] til b[] fordi hver tråd bare flytter sine sifferverdier. Hver tråd går gjennom hele a[] .
- Alle b1,b2,b3: <sync på en CyclicBarrier cb>
  - Spørsmål: Hvilken av løsningene tar lengst tid?

Hva skal så vi på i Uke 12

I) Om «sifre» i Om Oblig 3

II) Om optimalisering av Oblig2 – nye tall med Java8

III) Java8

- Forbedringer
  - Feilen med tidtaking
  - Raskere Jit-kompilering
- Nytt:
  - Metoder som parametre til metoder (closure, lamda)
  - Akkumulatorer i `java.util.concurrent`

IV) Goldbach's påstand – uløst problem siden 1742

- Løsning av en av Goldbachs problemer
- Parallellisering av denne

V) Datastrukturen i Oblig3 – ulike alternativer