INF2440 Uke 6, våren2014 – Mer om oppdeling av et problem for parallellisering, mye om primtall + thread-safe

Arne Maus
OMS,
Inst. for informatikk

Oppsummering – Uke1

- Vi har gjennomgått hvorfor vi får flere-kjerne CPUer
- Tråder er måten som et Javaprogram bruker for å skape flere uavhengige parallelle programflyter i tillegg til main-tråden
- Tråder deler felles adresserom (data og kode)
- Vi kan gjøre mange typer feil, men det er alltid en løsning.
- En stygg feil vi kan gjøre: Samtidig oppdatering (skriving) på delte data, på samme variabel (eks: i++)
- Samtidig skriving på en variabel må synkroniseres:
  - Alle objekter kan nyttes som en synkroniseringsvariabel, og da kan vi bruke enten en synchronized metode for å gjøre det,
  - eller objekter av spesielle klasser som:
    - CyclickBarrier
    - Semaphore (undervises Uke2)
  - De inneholder metoder som await(), som gjør at tråder venter.
- Helst **unngå** samtidig skriving på felles variabel.

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

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 (etter Jit-kompilering)
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

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 (ikke pensum)
4. Hvorfor synkroniserer vi, og hva skjer da,
   1. Hvilke problemer blir løst (f.eks utsatte operasjoner)?
5. Ny, ‘bedre’ forklaring på Radix

Hva så vi på i uke 5

1. Mer om matrisemultiplikasjon –
   1. Radvis (ikke kolonnevis) beregning av C =AxB (speedup$\leq$ 64)
   2. Transponering kan også gjøres parallell (last-balansering)
2. Modell2-kode for sammenligning av kjøretider på (enkle) parallelle og sekvensielle algoritmer.
   1. Spesielt: Start med størst n først for å få kompilert og optimalisert koden mest mulig før vi kjører 'små' n.
3. Hvordan lage en parallell løsning – tre mislykkete og en vellykket måte parallellisere : i++
   1. Vellykket: lokal kopi av 'i' hver tråd, så addering til slutt.
4. Vranglås - et problem vi lett kan få (og unngå)
   1. Sortere synkroniserings-primitivene
5. Ulike strategier for å dele opp et problem for parallellisering – intro.

Hva skal vi se på i Uke6

1. Mer om ulike strategier for å dele opp et problem for parallellisering
2. Oppdeling av en algoritme i flere faser.
   1. Med synkronisering mellom hver fase
3. Om 'store' primtall og faktorisering (intro til Oblig2)
   1. Hvordan **lage og lagre** mange primtall
   2. Litt om hvordan faktorisere store tall N
      (= finne de primtall som multiplisert sammen gir N)
4. Om trådsikre-programmer og biblioteks-klasser (Api)

1) Om å parallelliser et problem

- Utgangspunkt: Vi har en sekvensiell effektiv og riktig sekvensiell algoritme som løser problemet.
- Vi kan dele opp både koden og data (hver for seg?)
- Vanligst å dele opp data
  - Som oftest deler vi opp data med en del til hver tråd, og lar 'hele' koden virke på hver av disse data-delene.
  - Eks: Matriser
    - radvis eller kolonnevis oppdeling av C til hver tråd
    - Omforme data slik at de passer bedre i cachene (transponere B)
  - Rekursiv oppdeling av data ('lett')
    - Eks: Quicksort
  - Primtalls-faktorisering av store tall N for kodebrekking:
    - $N= p_1 * p_2$
- Også mulig å dele opp koden:
  - Alternativ Oblig3 i INF1000: Beregning av Pi (3,1415..) med 17 000 sifre med tre ArcTan-rekker

Å parallellisere algoritmen

- Koden består en eller flere steg; som oftest i form av en eller flere samlinger av løkker (som er enkle, doble, triple..)
- Vi vil parallellisere med k tråder, og hver slikt steg vil få hver sin parallellisering med en CyclicBarrier-synkronisering mellom hver av disse delene
  + en synkronisert avslutning av hele algoritmen(join(), ..).
- Eks:
  - finnMax – hadde ett slikt steg: for (int i = 0 …n-1) -løkke
  - MatriseMult hadde ett slikt steg med trippel-løkke
  - Radix hadde 4 slike steg:
    - en enkelt løkke i radix2
    - tre steg i radixSort : a),b) og c) - alle enkeltløkker (gjenntatt 2 ganger)
    - Hver av disse må få sin parallellisering.

Å dele opp data – del 2

- For å planlegge parallellisering av ett slikt steg må vi finne:
  - Hvilke data i problemet er lokale i hver tråd?
  - Hvilke data i problemet er felles/delt mellom trådene?
- Viktig for effektiv parallell kode.
  - Hvordan deler vi opp felles data (om mulig)
  - Kan hver tråd beregne hver sin egen, disjunkte del av data
  - Færrest mulig synkroniseringer (de tar alt for 'mye' tid)

Viktig: Kopiere deler av felles data til lokale data

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

Om å parallellisere et problem; dele opp data – del 3

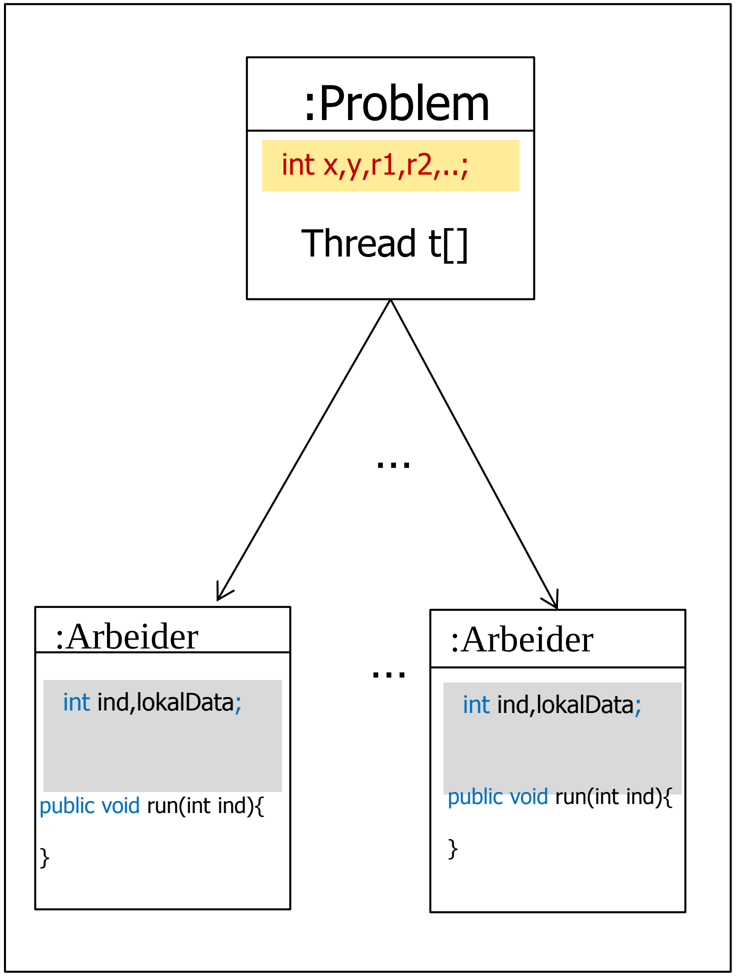
- Hvilke data i problemet er felles og som skal endres?
- Matrisemultiplikasjon: **Ingen delte data skal skrives** (pinlig parallelliserbart)
- Radix-sortering: Arrayene: a[] og b[] og count[] er felles.
  - a[] og b[] er lette å dele opp – count [] vanskeligere
  - Kan hver tråd få sin lokale kopi av count[] ?
  - Blir det samme oppdeling i hvert steg av algoritmen?
    - Sannsynligvis ikke samme oppdeling
  - En av stegene (finnMaks) har vi allerede parallellisert med en lokal kopi av max til hver tråd.
- Noen andre problemer kan deles opp rekursivt – eks QuickSort
  - Meget lett å parallelliser (mer om det senere)

Husk at vi skal lage effektive algoritmer !

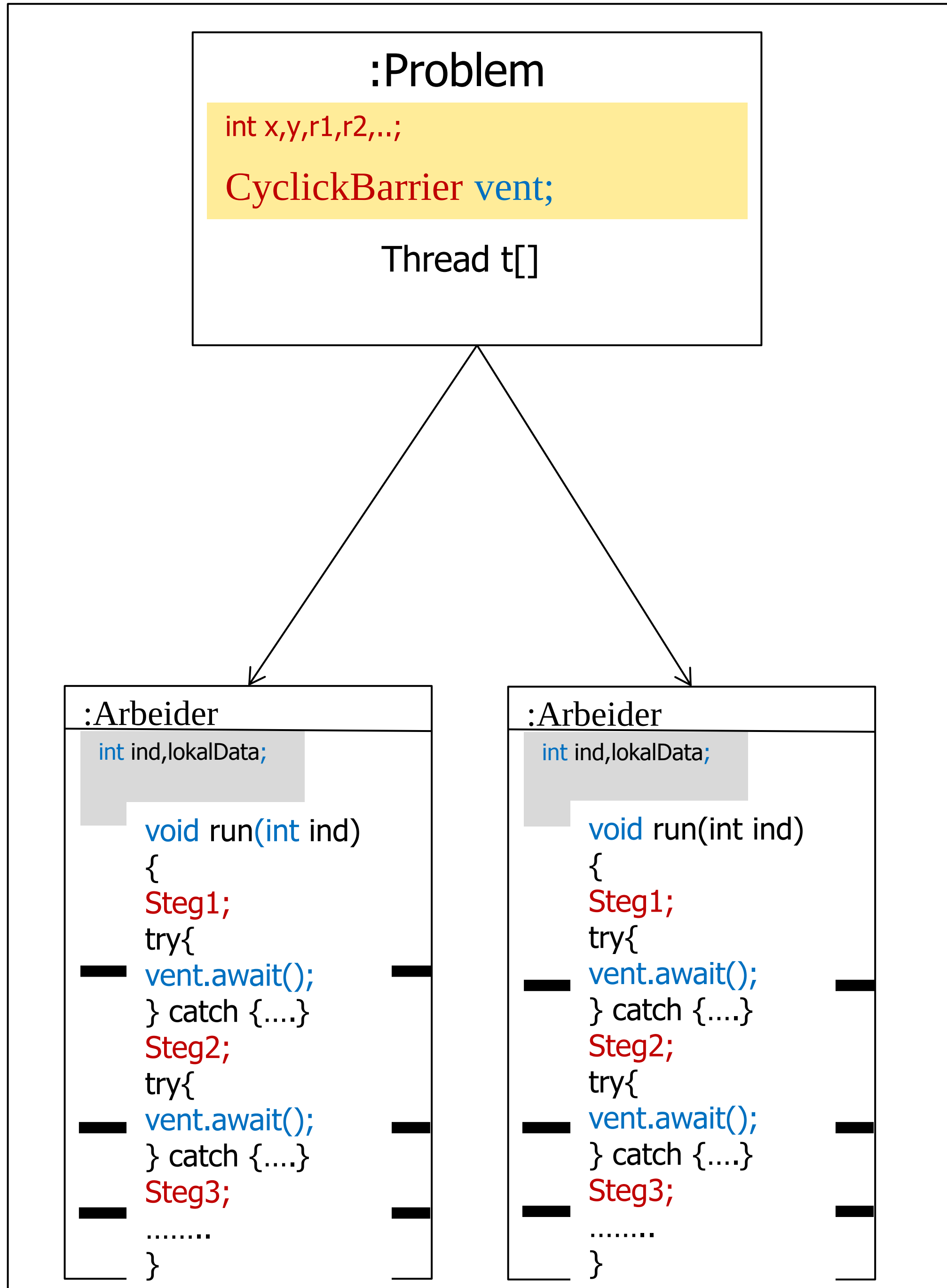
- Vi må ikke synkronisere for mange ganger !
  - Fordi hver synkronisering tar 'lang' tid (skriving av cachene til lageret, utføre utsatte operasjoner,..)
- Vi kan **ikke** synkronisere hver gang vi i den sekvensielle algoritmen bruker (leser/skriver) felles data.
- Regel for synkronisering:
  - Antall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

Vår 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
```



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

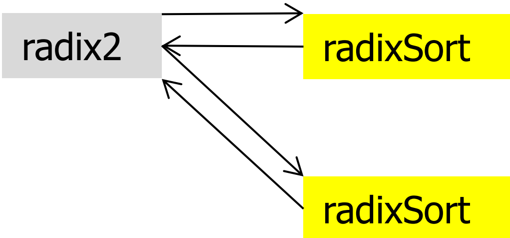


Effektivitet: Husk, ikke lag alt for mange tråder !

- Det er grovt sett ingen vits i å lage flere tråder enn vi har kjerner (av og til lønner det seg med færre tråder).
- Har vi f.eks en rekursiv metode for quicksort, må vi holde opp å si new Thread(…) hver gang vi deler opp området vi sorterer i to.
  - Fordi et rekursivt kall er mye raskere enn å si new Thread(..)
- Det vil si at i noen typer av rekursive løsninger så:
  - Så bruker vi den sekvensielle algoritmen hvis n<LIMIT
  - Vi parallelliserer så lenge vi har tråder igjen
  - Vi går så tilbake til den sekvensielle rekursive algoritmen i **hver tråd.**
  - Vi har da k stk sekvensielle algoritmer i parallell

Steg i radix2:

Vi har parallellisert Radix2 hvis vi greier å parallelliser steg 2) 3) og 4)




```
static void radix2(int [] a) {
  ...... sekvensielle enkelt-setninger ....

     // 1) FINN MAX i a[] –
     //     bruk parallell variant
```

Barrier

```
   ...... sekvensielle enkelt-setninger ....
          radixSort( a,b, bit1, 0);
          radixSort( a,b, bit2, bit1);
```

Barrier

```
 }
```

```
static void radixSort ( ..................){
      ...... sekvensielle enkelt-setninger ....

     // 2) tell opp hvor mange av hver sifferverdi
        for (int i = 0; i < n; i++)
         count[(a[i]>> shift) & mask]++;
```

Barrier

```
     // 3) Legg sammen count[] til hvor hver verdi
     //      i a[] skal i b[]
      for (int i = 0; i <= mask; i++) {
           j = count[i];
            count[i] = acumVal;
            acumVal += j;
       }
```

Barrier

```
     // 4 ) Flytt fra a[] til b[] i sortert på dette sifferet
      for (int i = 0; i < n; i++)
         b[count[(a[i]>>shift) & mask]++] = a[i];
```

Barrier

```
}// end radixSort
```

3) Om primtall

- Et primtall er :
  Et heltall som bare lar seg dividere med 1 og seg selv.
  - 1 er ikke et heltall (det mente mange på 1700-tallet)
- Ethvert tall N >0 lar seg faktorisere som et produkt av primtall:
  - $N = p_1 * p_2 * p_3 * \ldots\ldots\ldots * p_k$
  - Denne faktoringen er entydig; dvs. den eneste faktoriseringen av N
  - Hvis det bare er ett tall i denne faktoriseringen, er N et primtall.

Hvordan lage og lagre primtall

```
Z:\INF2440Para\Primtall>java PrimtallESil 2000000000
 max primtall m:2000000000
Genererte primtall <= 2000000000  paa     18949.04 millisek
med Eratosthenes sil  og det største primtallet er:1999999973
```

```
Z:\INF2440Para\Primtall>java PrimtallDiv 2000000000
Genererte alle primtall <=2000000000  paa 1577302.55  millisek med
 divisjon , og det største primtallet er:1999999973
```

- Å lage primtallene p og finne dem ved divisjon (del på alle primtall < SQRT(p)) er ca. 100 ganger langsommere enn Erotasthenes avkryssings-tabell (kalt Erotasthenes sil).

Å lage og lagre primtall (Erotasthenes sil)

- Som en bit-tabell (1- betyr primtall, 0-betyr ikke-primtall)
  - Påfunnet i bronsealderen av Eratosthenes (ca. 200 f.kr)
  - Man skal finne alle primtall < M
  - Man finner da de første primtallene og krysser av alle multipla av disse (N.B. dette forbedres/endres senere):
    - Eks: 3 er et primtall, da krysses 6, 9,12,15,.. Av fordi de alle er ett-eller-annet-tall (1,2,3,4,5,..) ganger 3 og følgelig selv ikke er et primtall. 6=2*3, 9 = 3*3, 12 =2*2*3, 15 = 5*3, ..osv
    - De tallene som ikke blir krysset av , når vi har krysset av for alle primtallene vi har, er primtallene
- Vi finner 5 som et primtall fordi, etter at vi har krysset av for 3, finner første ikke-avkryssete tall: 5, som da er et primtall (og som vi så krysser av for, …finner så 7 osv)

To helt avgjørende observasjoner

1) Hvis vi vet alle primtall < M, så kan vi faktorisere all tall N < M*M, fordi:
   - Hvis N ikke er et primtall selv så består faktoriseringen av minst to primtall N=p1*p2. Ett av p1 eller p2 er minst, si p1, og da ser vi at p1 ≤ SQRT(N).
   - Dvs. har delt N på alle primtall < M, så finner vi enten en faktor i faktoriseringa av N, eller fastslår at N er et primtall (fordi ingen av divisjonene hadde en rest ==0)

2) Når vi krysser av for et primtall p, så det første tallet vi trenger å krysse av for er p*p, fordi alle mindre multipla et krysset av for av mindre primtall.
   - Eks: Avkryssing for 5. Vi starter på 5*5 = 25 fordi 10,15,20 er allerede krysset av 2,3,4=2*2. Men etter 25 må vi krysse av 35,45,55,.. osv.

+ en til

3) Vi representerer bare oddetallene i bit-arrayen vår:

   1,3,5,7,9,11,…

   fordi vi vet at bare 2 av partallene er et primtall. Det er denne tallrekken vi krysser av i.

   (dette halverer lagringsplassen og arbeidet med avkryssing)

Litt regneregler for logaritmer med ulike grunntall

DEF: Vi har at $\log_a N$ (logaritmen til N med 'a' som grunntall) er det tallet x , som er slik at $a^x = N$

a sier vi er grunntallet til logaritmen (a =2,e,10,..)

$\log_2 x$ = antall bit i x, $\log_{10} x$ = antall desimale sifre i x

Regneregler om logaritmer med ulikt grunntall (som informatikere bruker vi ofte grunntall 2) :

$$\log_b x = \frac{\log_a x}{\log_a b} = \frac{1}{\log_a b} \log_a x$$

$$b = 2, a = e(2.7182..)$$

$$\log_2 x = 1.44 \ln x$$

$$\ln x = 0.693 \log_2 x$$

Hvor mange primtall er < N

$$\text{Antall primtall} < \text{N} = \text{teoretisk } \frac{\text{N}}{\ln \text{N}},$$

$$\text{men i praksis} = \frac{\text{N}}{\ln \text{N} - 1{,}06} = \frac{\text{N}}{0.693 \log_2 \text{N} - 1.06}$$

$$\text{N} = 1000, \text{antall primtall} = 171 = 17\%$$

$$\text{N} = 1\text{mill}, \text{antall primtall} = 78\,397 = 7{,}8\%$$

$$\text{N} = 2000\text{mill}, \text{antall primtall} = 98\,249\,139 = 4{,}9\%$$

og byte-arrayen med 7 oddetall i hver byte trenger N/14 byter

Hvordan faktorisere et stort tall (long)

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

M

For å faktorisere så store tall som M

$N = \sqrt{M} \Leftarrow$ for å lage all primtall < N

Q $= \sqrt{N}$ $\Leftarrow$ Vi må krysse av for disse primtallene

Vise at vi trenger bare primtallene <10 for å finne alle primtall < 100, avkryssing for 3 (3*3, 9+2*3,9+4*3, ….)

1	**3**	**5**	**7**	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75	77	79
81	83	85	87	89
91	93	95	97	99

1	**3**	**5**	**7**	**9**
11	13	**15**	17	19
21	23	25	**27**	29
31	**33**	35	37	**39**
41	43	**45**	47	49
51	53	55	**57**	59
61	**63**	65	67	**69**
71	73	**75**	77	79
81	83	85	**87**	89
91	**93**	95	97	**99**

Avkryssing for 5 (starter med 25, så 25+2*5, 25+4,5,..):

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75	77	79
81	83	85	87	89
91	93	95	97	99

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

Avkryssing for 7 (starter med 49, så 49+2*7,49+4*7,.):

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63 63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

Er nå ferdig fordi neste primtall vi finner: 11, så er 11*11=121 utenfor tabellen

Fasit fra nettet mot våre tall (vi vet at 2 er et primtall)

1	**3**	**5**	**7**	9
11	13	**15**	17	19
21	23	**25**	**27**	29
31	**33**	**35**	37	**39**
41	43	**45 45**	47	**49**
51	53	**55**	**57**	59
61	**63 63**	**65**	67	**69**
71	73	**75 75**	**77**	79
81	83	**85**	**87**	89
91	**93**	**95**	97	**99**

1	**2**	**3**	4	**5**	6	**7**	8	9	10
11	12	**13**	14	15	16	**17**	18	**19**	20
21	22	**23**	24	25	26	27	28	**29**	30
31	32	33	34	35	36	**37**	38	39	40
41	42	**43**	44	45	46	**47**	48	49	50
51	52	**53**	54	55	56	57	58	**59**	60
61	62	63	64	65	66	**67**	68	69	70
71	72	**73**	74	75	76	77	78	**79**	80
81	82	**83**	84	85	86	87	88	**89**	90
91	92	93	94	95	96	**97**	98	99	100

Et lite talleksempel

- Hvis vi krysser av med alle primtall < 100
- Så finner vi alle primtall < $100^2 = 10\,000$
- Da kan vi faktorisere alle tall
  $M < 10\,000^2 = 100\,000\,000 = 100$ mill,
  dvs. alle 8 sifrete tall

Den store fordelen med at for å krysse av for primtall p først med p*p, er at det er så få primtall vi skal krysse av med.

Avkryssing går fra å være en O(n) algoritme til en O($\sqrt{n}$) algoritme.

Det størst talleksemplet

- Hvis vi krysser av med alle primtall < 44 721
- Så finner vi alle primtall < $44\,722^2 = 2\,000\,057\,284$
- Da kan vi faktorisere alle tall
  M < 4 000 229 139 281 456 656, dvs. 18-19 sifrete tall
- Største tall med 58 bit er et mindre tall (18-sifret):
  288 230 376 151 711 743 = 3*59*233*1103*2089*3033169

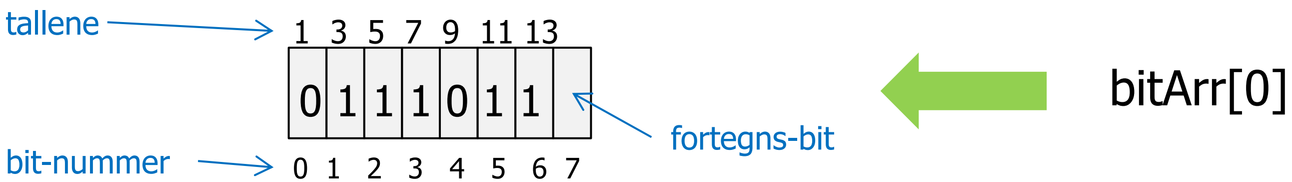
Vi har nådd vårt mål om å kunne knekke 58-bit koder (GSM) hvis vi vi har en bitarray for oddetallene som er ca.1 milliard bit lang, dvs.

1 milliard/14 = 73 millioner byter lang

Nyttige tips om implementering av dette

Tillegg:

- Da jeg laget en implementasjon av en slik bit-array nyttet en array av byter: **`byte [] bitArr = new byte [(len/14)+1];`**
- Husk at **alle** typer heltall (byte, short, int og long) har et fortegnsbit som sier om tallet er positivt eller negativt – helst ikke rør det !
- Det betyr at vi bruker 7 bit i hver byte til å representere om et tall er primtall (1) eller ikke primtall(0) – f.eks er 1,3,5,7,9,11,13 representert i byte [0], mens 15,17,.. er i byte[1]



- Husk et i/14 gir hvilken byte i bitArr[] tallet 'i' er representert i
- Husk at (i%14)>>1 gir hvilket bit-nummer i den byten tallet 'i' er.
- Du må av og til veksle mellom long og int – for eksempel slik (ttall er int:
  - long p = (long) tall + 2L;

Resultater fra kjøring av min sekvensielle løsning

```
3999999999998764380 = 2*2*3*5*17*151*212141*122421659
3999999999998764381 = 23*37*14683*309559*1034123
3999999999998764382 = 2*1429*58243*24030014353
3999999999998764383 = 3*1333333333332921461
3999999999998764384 = 2*2*2*2*2*7*179*347*287494451357
3999999999998764385 = 5*13*127*1303*63587*5848307
3999999999998764386 = 2*3*3*222222222222153577
3999999999998764387 = 3999999999998764387
3999999999998764388 = 2*2*11*19*4784688995213833
 100 faktoriseringer beregnet paa:   62991.7493ms -
 dvs:    629.9175ms. per faktorisering

 Stoerste 58-bit tall:288230376151711743 =
3*59*233*1103*2089*3033169
```

3) Om thread-safe og fail-fast

- En klasse er thread-safe hvis den tåler at to eller flere tråder samtidig aksesser dets metoder.
  - Kan for eksempel oppnås ved at alle metodene er synchronized (men det er treigt!)
- Fail-fast brukes i Java om iterereratorer, og prøver å garanterer at det kastes en ConcurrentModificationException når flere tråder samtidig har aksessert en mengde – eks:

```
ArrayList fak = new ArrayList();
 for (Long tall:fak)
    System.out.println(«Neste tall i fak:»+tall);
```

- De fleste av klassene i JavaAPI er **ikke** thread-safe !
  - Det må du selv fikse – lage f.eks egne metoder beskyttet av en ConcurrentLock som så kaller klassens tilsvarende metode mens låsen holdes.
  - Husk at ConcurrentLock er 5x fortere enn synchronized og gjør det samme.

Hva skal vi se på i Uke6

1. Mer om ulike strategier for å dele opp et problem for parallellisering
2. Oppdeling av en algoritme i flere faser.
   1. Med synkronisering mellom hver fase
3. Om trådsikre-programmer og biblioteks-klasser (Api)
4. Om 'store' primtall og faktorisering (intro til Oblig2)
   1. Hvordan lage og lagre mange primtall
   2. Noe om hvordan faktorisere store tall N –
      (= finne de primtall som ganget sammen gir N)