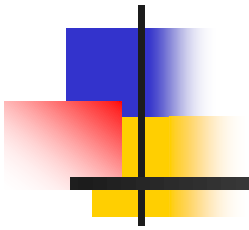


INF2440 Uke 8, v2014 : Ulike Threadpools, JIT-kompilering og mer om faktorisering av primtall

Arne Maus
OMS,
Inst. for informatikk

Hva har vi sett på i uke7

1. Om faktorisering av ethvert tall M < N*N (oblig 2)
   (finne de primtallene < N som ganget sammen gir M)
2. Flere metoder i klassen Thread for parallellisering
3. Tre måter å programmere monitorer i Java eksemplifisert med tre løsninger av problemet: Kokker og Kelnere
   1. med sleep()og aktiv polling.
   2. med synchronized methods , wait>() og notify(),..
   3. med Lock og flere køer (Condition-køer)

Hva skal vi se på i uke 8:

1. En effektiv Threadpool ?
   - Executors.newFixedThreadPool
2. Mer om effektivitet og JIT-kompilering !
3. Om problem(?) mellom long og int
4. Presisering av det å faktorisere ethvert tall M
5. Om parallellisering av
   - Laging og lagring av Eratosthenes Sil
   - Faktorisering av ethvert tall M < N*N
     (finne de primtallene < N som ganget sammen gir M).
6. Utsettelse av innleveringsfristen for Oblig2 en uke.

1) En effektiv Threadpool ?

- Vi har med modell2-koden selv startet en samling av k tråder som- venter til vi vil kjøre problemet (evt. flere ganger):
  - enten med samme n for å få bedre tider (median)
  - eller for en ny n .
- Ideen om å lage en samling av tråder som 'er klar for' å løse neste oppgave har vi også i Java-biblioteket.
- Vi skal her se på: Executors.newFixedThreadPool i java.util.concurrent.

Grunnidéene i Executors.newFixedThreadPool

- Du starter opp et fast antall tråder
- Hvis du under kjøring trenger flere tråder enn de startet må du vente til en av dem er ferdig og da er ledig
- For hver tråd som gjør noe er det tilknyttet et objekt av klassen **Future**:
  - Den sier deg om tråden din er ferdig
  - Du kan legge deg og vente på et eller alle Future-objekter
    - som join() med 'vanlige' tråder
  - Future-objektet bringer også med seg svaret fra tråd når den er ferdig, hvis den har noen returverdi
- En tråd som har terminert kan straks brukes på nytt fordi main-tråden venter ikke på tråden, men på tilhørende Future

For å administrere en slik mengde (pool) av tråder

- Må man først lage en pool (med tilhørende Vektor av Futures):

```
class MinTraadpool{
MinTraadpool pt = new MinTraadpool ();
int antTraader = Runtime.getRuntime().availableProcessors();
ExecutorService pool =  Executors.newFixedThreadPool(pt.antTraader);
List <Future> futures = new Vector <Future>();
```

- Hvordan lage trådene og slippe dem ned i poolen (svømmebasenget):

```
for (int j =0; j < antTraader; j++) {
        Thread QParExec = new Thread(new FindExec(j));
        futures.add(pool.submit(QParExec)); // submit starts the Thread
}
```

For å administrere en slik mengde (pool) av tråder

- Slik venter man på framtider (Futures) – get returnerer svaret (hvis noe – ikke her)

```
while (!futures.isEmpty()) {
     Future top = futures.remove(0);
     try {
          if (top != null)  top.get();
    } catch (Exception e) { System.out.println("Error Futures");}
}
```

- Trådene som startes er vanlige (indre) klasser med en parallell metode 'run()' :

```
class FindExec implements Runnable {
    int index;
    public FindExec(int ind) { index = ind;   }
    public void run() {
       <kall parallell metode>
 }}
```

Mange muligheter med slike pooler:

- Man kan lukke poolen: `pool.shutdown();`
- Man kan lage ca. 13 ulike typer av pooler: Fast størrelse, variabel størrelse, cachete (gjenbruk), med tidsforsinkelse, med og uten ThreadFactory (et objekt som lager tråder når det trengs),..
- Det hele koker ned til spørsmålene:
  - Vi trenger å ha en rekke tråder som parallelliserer problemet vårt
  - Er en slik trådpool effektiv?

1 &2)Test på effektivitet – tre implementasjoner av parallell FindMax i :int [] a – speedup:

- a) Med ExecutorService og FixedThreadPool
  - a1)Med like mange tråder som kjerner:
  - a2) Med 2x tråder som kjerner
- b) Med modell2–kode med 2 CyclicBarrier for start&vent

n	a1)Pool 1x	a2) Pool 2x	b) Barrier
1000	0.016	0.037	0,049
10000	0.249	0,235	0,011
100000	0.256	0,302	0,105
1000000	0.469	0,399	0,453
10000000	1.353	1,344	1,339
100000000	1.615	1,6662	1,972

- Konklusjon: Om lag like raske !?

2) Nok et problem/overraskelse med JIT-kompilering

- Som vi husker går JIT-kompilering i flere steg:
  - Oversettelse til maskinkode av hyppig brukt kode
  - Mer bruk ⇒ optimalisering av den maskinoversatte koden
    - Si 50 000 ganger
  - Enda mer bruk ⇒ super-optimalisering av koden en gang til
- Først noen resultater; sammenligning av Barrier-løsningen av FinnMax med samme løsning skrevet litt annerledes:

Speedup:

n	b) Barrier	Ny Barrier
1000	0,049	0,14
10000	0,011	0,66
100000	0,105	0,57
1000000	0,453	**3,40**
10000000	1,339	**7,69**
100000000	1,972	**9,38**

Konklusjon: Hurra, en mye bedre parallellisering ?

- **Speedup: 1,972 < 9,38 !!!**
- En klart bedre løsning ?
- At speedup er mye større, er det da sikkert at parallelliseringen er mye bedre/raskere ?
- Svar: JA , men bare hvis den sekvensielle koden som parallelliseringen sammenlignes med er den samme.
- Det eneste som var forskjellig mellom disse to programmene var at jeg hadde forsøkt å gjøre den **sekvensielle** koden raskere ved å 'inline'- selve koden på kallstedet. Den parallelle koden var den samme.
- Vi tester og ser etter hva som har skjedd her!

Hva var forskjellen mellom de to Barrier-kodene for n = 100 000 000

- Fra b) Barrier –programmet (>java FinnBarrier.java):

```
Median parallel time    :26.697743
Median sequential time:52.128282,
, Speedup:  1.98, n = 100000000
```

- Fra Ny Barrier-programmet (INF2440Para\FinnMax>java FinnM):

```
Max verdi parallell   i a:99989305,    paa:  23.691937 millisek.
Max verdi sekvensiell i a:99989305, paa: 223.66429 millisek.

Median sequential time:223.593249, median parallel time:23.84312,
 n=  100000000, Speedup:  9.38
```

- Konklusjon: Det er særlig den sekvensielle kjøretiden er langsommere, **ikke** at den parallelle er raskere.

Hva var forskjellen mellom de to kodene **UTEN JIT-kompilering,** n = 100 000 000 (java –Xint BarrierMax ..) ?

- Fra b) Barrier –programmet:

```
Median parallel time:     533.367279
Median sequential time:1772.424558,
 Speedup:  3.35, n = 100000000
```

- Fra Ny Barrier-programmet:

```
Max verdi parallell   i a:99989305,    paa: 541.823362 millisek.
Max verdi sekvensiell i a:99989305, paa: 2334.368515 millisek.

Median sequential time:2334.184786, median parallel time:546.844936,
 n=  100000000, Speedup:  4.19
```

- Konklusjon: Det er særlig den sekvensielle kjøretiden som er bedre optimalisert , ca.30x for b) mens bare ca. 10x for NyBarrier!
- Hvorfor ?

Her er forskjellen mellom de to programmene

Ny Barrier ,'langsom' (10x) sekvensiell optimalisering. Koden for sekvensiell metode er lagt rett inn i en større metode (utfor()) som ikke kalles mange ganger

```
// sekvensiell  finnMax
t = System.nanoTime();
 totalMax = 0;
 for (int i=0;i < a.length;i++)
    if (a[i] > totalMax) totalMax = a[i];
 t = (System.nanoTime()-t);
seqTime[j] =t/1000000.0;
```

b) Barrier , mye raskere sekvensiell 34x optimalisering.Laget en metode av sekvensiell kode som kalles fra utfor-metoden.

```
int doSequentialVersion(int [] a) {
 int max= -1;
 for (int i = 0; i< a.length; i++)
     if (a[i] > max) max = a[i];
        return max;
 } // end doSequentialVersion
```

Optimalisatoren i JIT ser ut til å være langt bedre til å optimalisere (små) metoder enn en løkke inne i en større metode .

Observasjon– JIT-kompilering/optimalisering

- Vi bør ta hensyn til hvordan optimalisatoren virker
- Generelt ser den ut til å greie å øke hastigheten på interpretert kode med 10x
- Små metoder som vi finner i klassen for en bit array for pimtall implementer i en byte-array (med metoder som isPrime(i) , setNotPrime() , nextPrime(m),…) og : int doSequentialVersion(int [] a) i FinnMax er meget velegnet for optimalisering
- Så små metoder kan optimaliseres i vårt tilfelle: 1772/52= 34 x
- Når vi skriver kode bør vi bruke denne kunnskapen fordi
  - Gir enklere kode – lettere å skrive
  - Gir raskere kode
- Grunnen til dette er nok at ved at programmereren har lagt kode inn i en metode sier hun at avhengigheten til resten av programmet er begrenset, grovt sett til parameterne. Lettere da å optimalisere!
- Oppskrift: Best optimalisering hvis en metode bare behandler parameterne og lokale variable i metoden.

3) Problemer med forholdet mellom long og int

- Java prøver hele tiden å være mest økonomisk når den regner ut uttrykk:
  - Regner ut billigst først og så konverterer ved tilordning.

```
public static void main (String [] args) {
        int m = 2000000000;

        long tall = m*m;
        System.out.println("M:"+m+", tall (m*m):"+ tall);
            tall = (long)m*m;
        System.out.println("M:"+m+", tall((long)m*m):"+ tall);
            tall = (long)m*(long)m;
        System.out.println("M:"+m+", tall((long)m*(long)m):"+ tall);
}
```

```
M:\INF2440Para\Powerpoint\Uke8>java LongEksempel
M:2000000000, tall (m*m):-1651507200
M:2000000000, tall((long)m*m):4000000000000000000
M:2000000000, tall((long)m*(long)m):4000000000000000000
```

4) Om primtall: 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.

Spørsmål: Er det riktig at vi trenger bare primtall p<$\sqrt{N}$ for å faktorisere N

Fra epost fra en av studentene:
==========
«Har sett på faktoriseringen av tall og sett på invarianten :
For å faktorisere (feks) 10000 så må man ha alle primtall < sqrt(10000) == 100.
men faktoriserer du 9999 får du 3*3*11*101.

101 > 100 ? »
…….

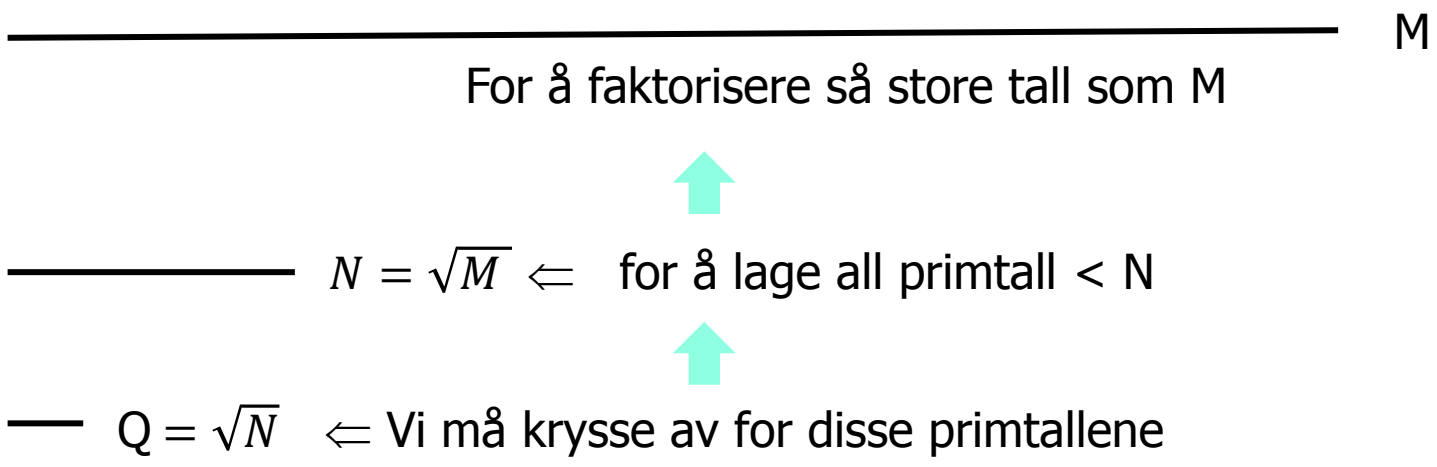
- Hva har vi sagt og som er riktig:
  - Vi kan faktorisere alle tall N ved å vite alle primtall p<$\sqrt{N}$
- Vi har **ikke sagt:** Alle primtallsfaktorene i n er <$\sqrt{N}$.
- Hva er forskjellen ???

Hvorfor kan vi faktorisere alle tall N ved å vite alle primtall p<$\sqrt{N}$

- Vi viser først:
  Ingen tall N kan ikke ha to eller flere faktorer > $\sqrt{N}$
  - Anta det motsatte:
    N = p1*p2*..pk, og at p1 og p2 begge er > $\sqrt{N}$.
    p1 =a + $\sqrt{N}$ og p2 = b + $\sqrt{N}$.
    Da er: p1*p2= (a+ $\sqrt{N}$)*(b+ $\sqrt{N}$) = ab+(a+b) $\sqrt{N}$ + $(\sqrt{N})^2$ og siden alle disse er > 0 og $(\sqrt{N})^2$ =N, så har vi vist at to (eller flere) sånne faktorer > $\sqrt{N}$ ikke finnes i faktoriseringa av N.
- Det er altså høyst en slik faktor > $\sqrt{N}$ i faktoriseringa av N.
- Når vi har faktorisert N med alle de små primtallene < $\sqrt{N}$ står vi igjen med en rest N' < N. Hvis N' >1 må den da selv være et primtall (og da den siste faktoren i N), fordi den ikke er delbar med noe primtall < $\sqrt{N'}$
- Tilsvarende bevis som over. Hvis N' kunne faktoriseres, så må minst en faktor være < $\sqrt{N'}$, og de har vi allerede testet !

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 <$\sqrt{N} = \sqrt{\sqrt{M}}$



+ 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)

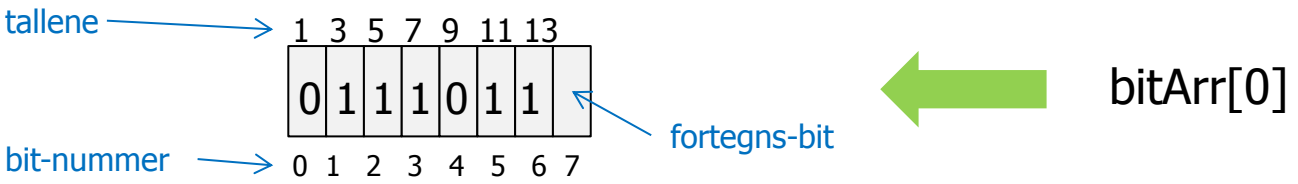
Å lage og lagre primtall (Erotasthenes sil)

- 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)

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 (tall er int):
  - long p = (long) tall + 2L;

Noen metoder fra klassen EratosthenesSil

```
class PrimeArray
 {
            byte [] bitArr ;
            int  bitLen;
            final  int [] bitMask ={1,2,4,8,16,32,64};
            final  int [] bitMask2 ={255-1,255-2,255-4,255-8,255-16,255-32,255-64};

 void crossOut(int i) {bitArr[i/14] &= bitMask2[(i%14)>>1]; }

 int nextPrime(int i) {
              // returns next prime number after number 'i'
                int k ;
                if ((i&1)==0) k =i+1; // if i is even, start at i+1
                else k = i+2;       // next possible prime
                while (!isPrime(k)) k+=2;
                return k;
  } // end nextPrime

boolean isPrime (int i) {
             // for all numbers i
             if (i == 2 ) return true;
             if ((i&1) == 0) return false;    // 0 i siste bit dvs. partall
             else  return (bitArr[i/14] & bitMask[(i%14)>>1]) != 0;
}
```

Parallellisere EratosthenesSil – del1

- Her er det to alternativer for å krysse ut ikke-primtall på N bit med k tråder i parallell:
  a) La tråd-0 krysse av for p = 3, tråd-1for p=5,.. osv til vi har krysset av for alle p < $\sqrt{N}$.
     a) Problem1 : Skjev fordeling av arbeidet mellom trådene. Antall kryss for p=3 er mange flere enn for 5 og særlig enn for 7 og…
     b) Problem2: Trådene vil skrive samtidig i samme felles byter. Synkroniserings-løsning: Alle trådene har kopi av bit-arrayen. Disse summeres sammen (logisk eller: |) i parallell til sist:
  b) La tråd-0 eie de N/k første bitene i bit arrayen, tråd-1 de N/k neste bit-ene,…, og krysse av med alle relevante primtall der.
     a) Problem: Vi må justere disse grensene mellom hva hver tråd eier, så de har hele byter, og at alle bytene er eid av en tråd (litt fiklete). Da oppnår vi at ingen av trådene skriver i samme byte.
- Jeg valgte alternativ b) i min løsning

Parallellisere EratosthenesSil – del2

- Uansett valg av a) eller b) har vi følgende problem:
  - For å generere kryss for N tall, trenger alle trådene å vite alle primtall p< $\sqrt{N}$.
    - For å løse et problem må vi altså allerede ha løst det?
    - svar:NEI:
  - Løsning: Vi lar f.eks tråd-0 først generere primtallene p < $\sqrt{N}$ nederst i bit-arrayen, før vi starter alle trådene (også tråd-0) i parallell og finner resten. Da oppnår vi at alle trådene bare leser i felles byter som ingen skriver i (de nederste bytene for primtallene < $\sqrt{N}$)
- Løsning b) med tillegget vi gjør her, medfører at ingen andre tråder leser eller skriver i en byte som en annen tråd skriver på. Vi har da et kappløpsfritt program med lite synkronisering – bare 2 barrier-synkroniseringer.

1) Faktorisering av et tall M i sine primtallsfaktorer sekvensielt

- Vi har laget og lagret ved hjelp av Erotosthanes sil alle primtall < N i en bit-array over alle odde-tallene.
  - 1 = primtall, 0=ikke-primtall
  - Vi har krysset ut de som ikke er primtall
- Hvordan skal vi så bruke dette til å faktorisere et tall M < N*N ?
- **Svar:** Divider M med alle primtall $p_i < \sqrt{M}$ (p_i =2,3,5,..), og hver gang en slik divisjon M/p_i har rest ==0, så er p_i en av faktorene til M. Vi forsetter så med å faktorisere M'=M/p_i.
- Faktoriseringen av M = p_i *..* p_k er da produktet av alle de primtall som dividerer M uten rest.
- HUSK at en p_i kan forekommer flere ganger i svaret. eks: 20= 2*2*5, 81 = 3*3*3*3, osv
- Finner vi ingen faktorisering av M', dvs. ingen $p_i < \sqrt{M'}$ som dividerer M' med rest == 0, så er M' selv et primtall.

Faktorisering sekvensielt - kjøretider

1. Dere har vel har på plass en effektiv sekvensiell løsning med om lag disse kjøretidene for N = 2 mill:

```
M:\INF2440Para\Primtall>java PrimtallESil 2000000
 max primtall m:2000000
Genererte primtall <= 2000000  paa        15.56 millisek
med Eratosthenes sil (  0.00004182 millisek/primtall)
......
3999998764380 = 2*2*3*5*103*647248991
3999998764381 = 37*108108074713
3999998764382 = 2*271*457*1931*8363
3999998764383 = 3*19*47*1493093977
3999998764384 = 2*2*2*2*2*7*313*1033*55229
3999998764385 = 5*13*59951*1026479
3999998764386 = 2*3*3*31*71*100964177
3999998764387 = 1163*1879*1830431
3999998764388 = 2*2*11*11*17*23*293*72139
 100 faktoriseringer beregnet paa:     422.0307ms -
 dvs:      4.2203ms. per faktorisering
```

Faktorisering i parallell – del1

- Vi nå laget N primtall i bit-arrayen vår og skal kunne faktorisere alle tall M <N*N.
- Grovt sett har vi her også to alternativer med k tråder:
  - a) «Annen-hver-idéen»: La tråd-0 dividere M med 3, så det k'te primtallet, så de 2k-te primtallet,..; tråd-1 med 5, det (k+1)te primtallet,… osv.
    - problem: Ikke like lett å effektivt hoppe slik og finne det neste k-primtallet.
    - Ikke plass til å lage alle disse k mengdene av primtall først – en mengde for hver tråd.
  - b) Del opp alle primtall < N slik: Gi de som er mindre enn N/k til tråd-0, de mellom (N/k)+1 og 2(N/k) til tråd-1,.. osv.
    - La hver tråd så finne de faktorer i M (= en av sine primtall) som er inne i sitt område.
  - Både a) og b): Hver tråd kan enten ha en egen ArrayList for faktorene som finnes, eller legge dem inn i en felles ArrayList via en synkronisert metode (ArrayList er ikke trådsikker).

Faktorisering i parallell – del 2

- Jeg valgte selv løsning b) – det at hver tråd forsøket å faktorisere M med ca. 1/k-del av primtallene.
  - Liten ulempe : det er flest primtall for tråd-0 (mellom 1 og N/k). Noe ujevnt arbeid mellom trådene.
  - Få synkroniseringer: en Barrier inne i algoritmen etter at alle trådene har faktorisert med sine primtall + en for avslutningen

Faktorisering i parallell – del 3

- Sært problem for både a) og b):
  - Ingen av trådene vet om den resten den har funnet: M' etter å ha delt ned M med sine faktorer er en faktor. Hvis M'>1, er det da et primtall eller noe som kan bli er ytterligere faktorisert av de andre trådene?
  - Løsning: Vi må etter at vi har synkronisert på at alle trådene er ferdige, gange sammen de faktorene vi har fått fra alle trådene i FakProd. Hvis FakProd == M, har vi funnet alle faktorene i M. Hvis ikke er M/FakProd den siste faktoren i M (og selv et stort primtall >N).

Eks: La N=20 og anta at vi har 4 tråder. Tråd-0 skal sjekke for alle primtallene i tallområdet 1-4 , tråd-1 har: 5-9,.., og tråd-3 :15-20.

La **M** = 322 , da vil tråd-0 finne 2 som faktor, tråd-1 finner 7, og ingen av de andre trådene finner noen faktorer. Da er FakProd =2*7 =14 (!= 322), og 322/14 **= 23** er en faktor og **322= 2*7*23**

Ikke akseptabel løsning på Oblig2

- Siden oblig2 består i å parallelliser faktoriseringen av 100 tall, så kunne man tenke seg at man:
  - Parallelliserte dannelsen av Eratosthenes Sil og så:
  - kjøre den sekvensielle faktoriseringen av de 100 tallene i parallell, slik at tråde 0 tok de 100/k første tallene, tråd-1 de neste 100/k tallene,..osv
  - Da har man ikke parallellisert de to algoritmene, men den første + Oblg2, ikke dette var ikke meningen med oppgaven.
  - En slik parallellisering av Oblig2 vil ikke bli godkjent selv om den tar like kort (eller kortere) tid.

Hva har vi sett på i uke 8:

1. En effektiv Threadpool ?
   - Executors.newFixedThreadPool
2. Mer om effektivitet og JIT-kompilering !
3. Om et problem mellom long og int
4. Presisering av det å faktorisere ethvert tall M
5. Om parallellisering av
   - Laging og lagring av Eratosthenes Sil
     - To alternativer
   - Faktorisering av ethvert tall M < N*N
     (finne de primtallene < N som ganget sammen gir M).
     - To alternativer
6. Utsettelse av innleveringsfristen for Oblig2 en uke.