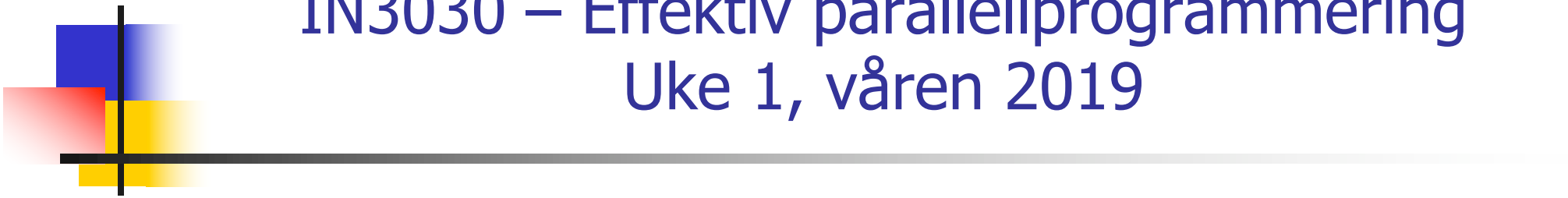


IN3030 – Effektiv parallellprogrammering
Uke 1, våren 2019

Eric Jul

Professor

PSE

Institutt for Informatikk

Litt om Eric

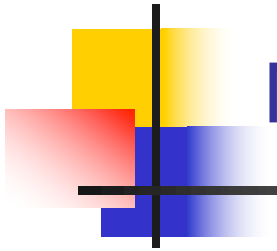
- **Ph.D. University of Washington, 1989**
- **Dansk-amerikaner**
- **Bor i Danmark – pendler til Oslo ca 3-4 gange/måned**
- **1989-2009: Førsteamauensis/professor Københavns Universitet**
- **2009-2015: Bell Labs, Dublin**
- **2016- Professor IFI**

En beslutning: hvilket språk?

- **Skal jeg forelese på dansk?**
- **Or English?**
- **Recording lectues: I will attempt BUT I make no promises – my recoding last Fall did not work ☹**
- **Lecture slides are in Norwegian (perhaps a little Danish intermixed in a few places...)**
- **Please keep up with the messages on the web site**
- **Use the Hjeplelærer ☺**
- **Ask questions when in doubt**

Bakgrunn IN3030

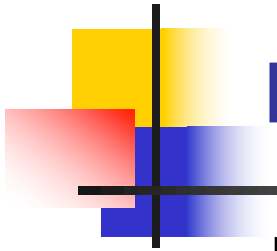
- **Kurs er fra 2014 – tidligere INF2440**
- **Lavet av Arne Maus – nu Emeritus**
- **Overtaket av Eric 2018**
- **I 2019: INF2440 -> IN3030**



Motivation for Parallelle Programmer

- **Maskiner kan bestå av flere CPU-er**
  - Hver CPU kan utføre programmer uavhengig av de andre CPUer
- **Hver CPU kan have flere kjerner**
  - Hver kjerne kan utføre programmer
- Vi vil gjerne utnytte disse muligheter for parallellisme

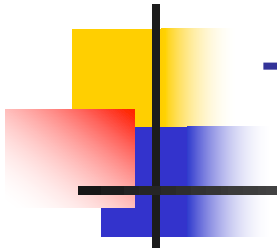
Hva vi skal lære om i dette kurset:



Lage parallelle programmer (algoritmer) som er:

- **Riktige**
  - Parallelle programmer er klart vanskeligere å lage enn sekvensielle løsninger på et problem.
- **Effektive**
  - dvs. raskere enn en sekvensiell løsning på samme problem

- Lære hvordan man parallelliserer et riktig, sekvensielt program + lage egne parallelle algoritmer som ikke bare er en slik parallellisering;
- Lære de mange problemene vi støter på og hvordan disse kan takles.
- Kurset er **empirisk** (med tidsmålinger), ikke basert på en teoretisk *modell* av parallelle beregninger,
- Vi oppfatter programmet som en god nok modell av det problemet vi skal løse. Vi trenger ingen modell av modellen.
- Presenterer en klassifikasjon av parallelle algoritmer (nytt).



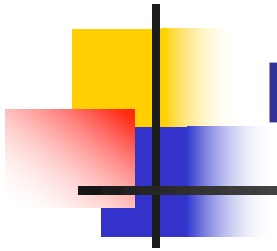
Tre grunner til å lage parallelle programmer

1) Skille ut aktiviteter som går **langsommere** i en egen tråd.
   - Eks: Tegne grafikk på skjermen, lese i databasen, sende melding på nettet. Asynkron kommunikasjon.
2) Av og til er det **lettere** å programmere løsningen som flere parallelle tråder. Naturlig oppdeling.
   - Eks: Kundesystem over nettet hvor hver bruker får en tråd.
   - Hele operativsystemet har mye parallellitet – 1572 aktive tråder i Windows 7 akkurat da denne foilen blev skrevet i 2017 – og 1921 aktive tråder i Mac OS X Sierra.
3) Vi ønsker **raskere** programmer, raskere algoritmer.
   - Eks: Tekniske beregninger, søking og sortering.

Dette kurset legger nesten all vekt på raskere algoritmer

IN3030 - et **nytt** kurs – og dog

- Noe vil også bli endret fra våren 2019 – spesielt da *Arne Maus*, som har utviklet kurs nu er emeritus – og *Eric* overtager.
- IN3030 er 90-95% baseret på INF2440.
- Planlagt fire obliger - den første legges ut 25. jan og vil få innleveringsfrist 9. feb. Så ca. 3-4 uker per oblig.
  - De individuelle innleveringer : Man kan samarbeide om algoritmer, men **ikke** ha helt lik eller delvis felles kode med andre.
- En oblig er ikke bare innlevering av ett eller flere parallelle programmer, men **også en liten rapport** om de testene man har gjort: hastighetsmålinger på disse for ulike størrelse av data med konklusjoner – f.eks speedup + O( ) og en forklaring på resultatene .
- Gruppetimer:
  - Jobbing med ukeoppgaver og obligene
- Nesten nytt kurs betyr at også dere selv vil være med på forme kurset, og at ikke alt vil være perfeekt.

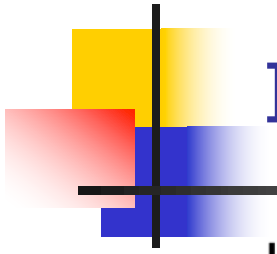


Pensum

- Ingen dekkende lærebok er funnet, men:
  - **Det som foreleses + oppgavene (obliger + ukeoppgaver) er pensum.**
  - **Bra bok:** Brian Goetz, T.Perlis, J. Bloch, J. Bobeer, D. Holms og Doug Lea::"Java Concurrency in practice", Addison Wesley 2006
  - Kap. **18 og 19** i A. Brunland, K. Hegna, O.C. Lingjærde, A. Maus:"Rett på Java" 3.utg. Universitetsforlaget, 2011.
  - I tillegg leses *fra en maskin på Ifi* kap 1 til 1.4, hele 2 og 3.1 til 3.7 (hopp over programeksemplene) i :

http://www.sciencedirect.com/science/book/9780124159938

(Hvis leses utenfor Ifi, så koster det $!)

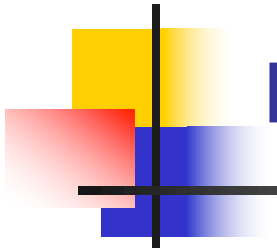


I dag – teori og praksis

- Ulike maskiner og kurs – hvor plasserer INF2440 seg?
- Begrunnelse for multikjerne CPU og parallelle løsninger/algoritmer.
- Parallelle løsninger på et problem er lengere (ofte minst dobbelt så lang kode) og (en god del) vanskeligere å lage enn en sekvensielt algoritme som løser samme problem.
- Den eneste grunnen til å lage parallelle algoritmer er at de går fortere enn samme sekvensielle algoritme – i alle fall for tilstrekkelig stor n (= antall data).
- Vi måler hvor-mange-ganger-fortere-det-går – speedup S:

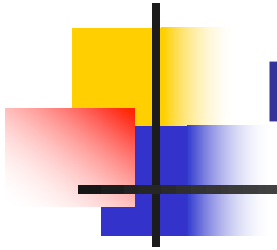
$$S = \frac{tid\ (sekvensiell\ algoritme)}{tid\ (parallell\ algoritme)}$$

- som da skal være > 1 , men vi skal også lage og teste programmer som har S < 1 og forklare hvorfor.



Lineær speedup ?

- Selvsagt ønsker vi lineær speedup – dvs. bruker vi *k* kjerner skulle det helst gå *k* ganger fortere enn med 1 kjerne.
- Meget sjelden at det kan oppnås (mer om det siden)
- Kan superlineær speedup oppnås?



Lineær speedup analogier

- Selvsagt ønsker vi lineær speedup, MEN:
- Nogle problemer er nemme at parallellisere: Eksempel: 10 personer kan plante 10 treer ca. 10 gange fortere end 1 person kan plante 10 treer.
- Andre problemer er vanskeligere at parallellisere: Eksempel: hvis 1 person kan samle et Lego-set på 10 timer, så vil det være vanskelig for 10 personer at samle det på 1 time – der er avhengigheter mellom delene.
- Andre problemer er nærmest umulige at parallellisere: Eksempel: hvis 1 kvinne kan lage et barn på 9 måneder, kan 9 kvinner så lage et barn på 1 måned?

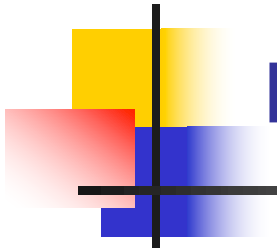
Flynns klassifikasjon av datamaskiner:

	Single Instruction	Multiple Instruction
Single Data	SISD : Enkeltkjerne CPU	MISD : Pipeline utførelse av instruksjoner i en CPU. Flere maskiner som av sikkerhetsgrunner utfører samme instruksjoner.
Multiple Data	SIMD : **GPU** – samme operasjon på mange elementer (en vektor) Det finnes også slike SSE – instruksjoner på Intel og AMDs CPU-er	**MIMD :** klynge av datamaskiner, og **Multikjerne CPU**

Eksempel på en SSSE 3-instruksjon (Intel i7)

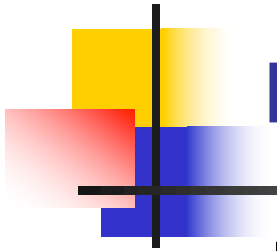
PHADDW, PHADDD	Packed Horizontal Add (Words or Doublewords)	takes registers A = [a0 a1 a2 …] and B = [b0 b1 b2 …] and outputs [a0+a1 a2+a3 … b0+b1 b2+b3 …]

- Det er mange registre (opp til 32) på en kjerne for bl.a. slike instruksjoner
- Et register kan sees på som en cache 0 – det tar bare én instruksjons-sykel å aksessere et register mot 3 sykler i cache1
- (i en 3 GHz CPU er en sykel 1/3 ns = 1/3 milliard dels sekund)



Dette kurset handler om tråder og effektivitet

- Hva er tråder
  - Se litt på maskinen
  - Se på operativsystemet
  - Hvordan skal vi oppfatte en tråd i et Java-program
  - Senere se på kompileringen og kjøring av Java-kode
- Hvordan måle effektivitet
  - Hvordan ta tiden på ulike deler av et program; både:
    - Den sekvensielle algoritmen
    - En eller flere parallelle løsninger
  - I dag: Enkel tidtaking
  - Neste gang: Bedre tidtaking
- Praktisk i dag
  - Standard måte å starte programmet med tråder

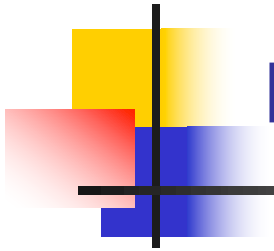


Flere mulige synsvinkler

Mange nivåer i parallellprogrammering:

1. Maskinvare
2. Programmeringsspråk
3. Programmeringsabstraksjon.
4. Hvilke typer problem egner seg for parallelle løsninger?
5. Empiriske eller formelle metoder for parallelle beregninger

INF2440: Parallellprogrammering av ulike algoritmer med tråder på multikjerne CPU i Java – empirisk vurdert ved tidsmålinger.

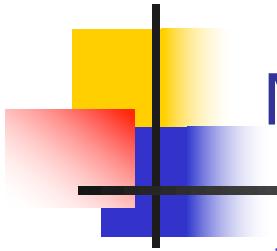


Learning-by-doing

Dette kurs er et Learning-by-doing kurs!

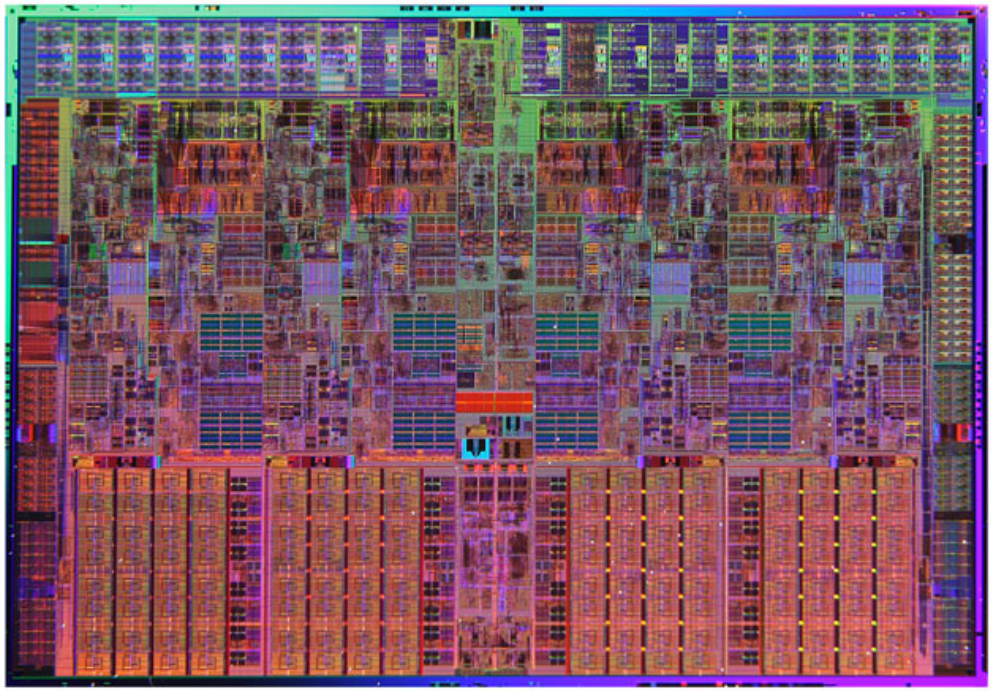
Utbyttet ditt er avhengig av DIN innsats!

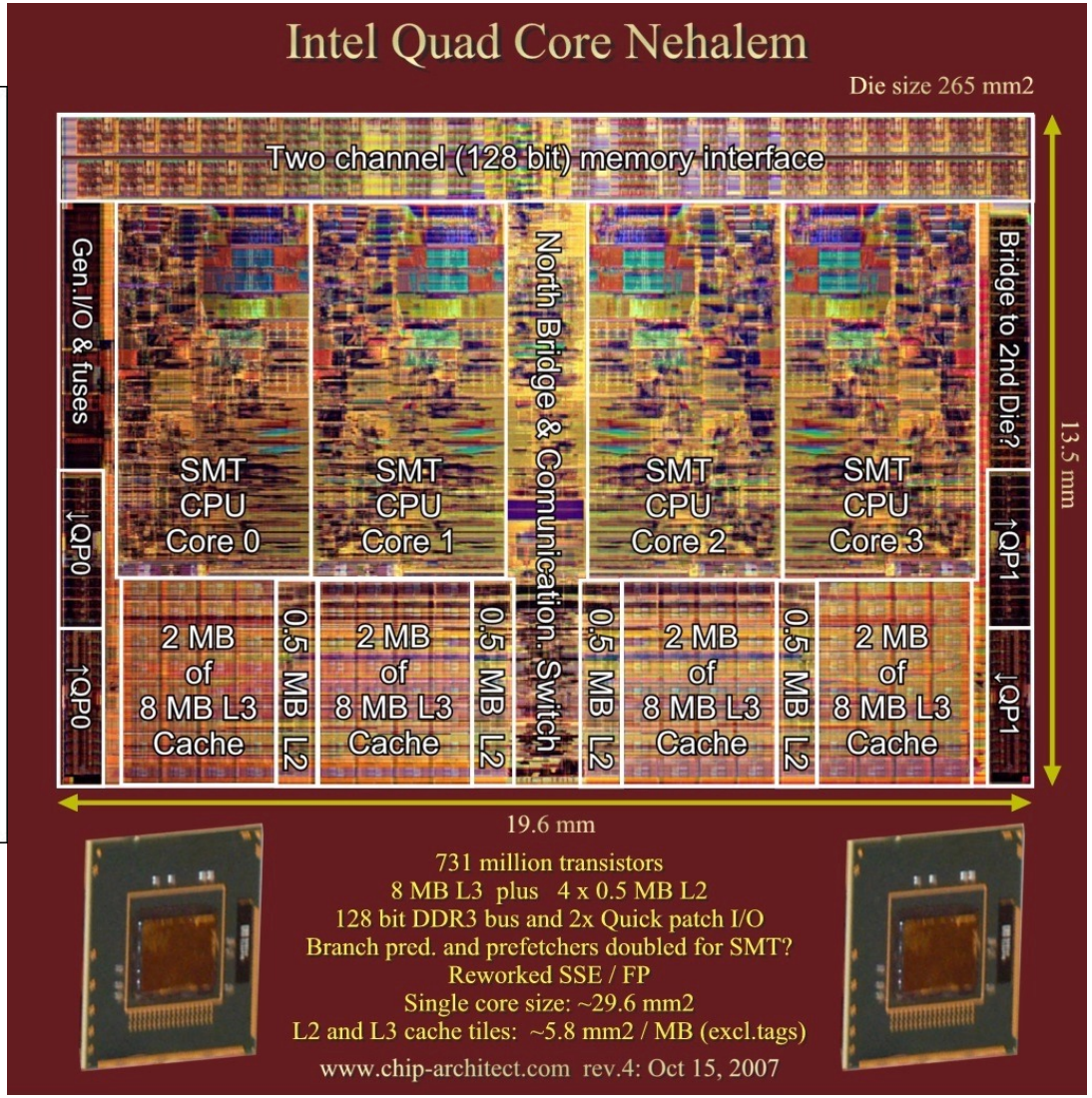
Maskinvare og språk for parallelle beregninger og Ifi-kurs



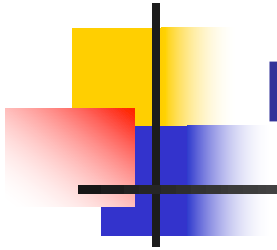
1. Klynger av datamaskiner - **INF3380**
   - jfr. Abelklyngen på USIT med ca 10 000 kjerner. 640 maskiner (noder), hver med 16 kjerner
   - C, C++, Fortran, MPI –de ulike programbitene i kjernene sender meldinger til hverandre
2. Grafikkkort GPU med 2000+ små-kjerner, **INF5063**
   - Eks Nvidia med flere tusen småkjerner (SIMD – maskin)
3. Multikjerne CPU (2-100 kjerner per CPU) **INF2440**
   - AMD, Intel, ARM, Mobiltelefoner,..
   - De fleste programmeringsspråk: Java, C, C++,..
4. Mange maskiner løst koblet over internett. **INF5510**
   - Planetlab – world-wide testbed
   - Emerald – språk til distribuert programmering
5. Teoretiske modeller for beregningene **INF4140**
   - PRAM modellen og formelle modeller (f.eks FSM)

Multikjerne - Intel Multicore Nehalam CPU





Mange ulike deler i en Multicore CPU – bla. en pipeline av maskininstruksjoner; kjernene holder på med 10 til 20 instruksjoner ***samtidig*** dvs. instruksjonsparallellitet

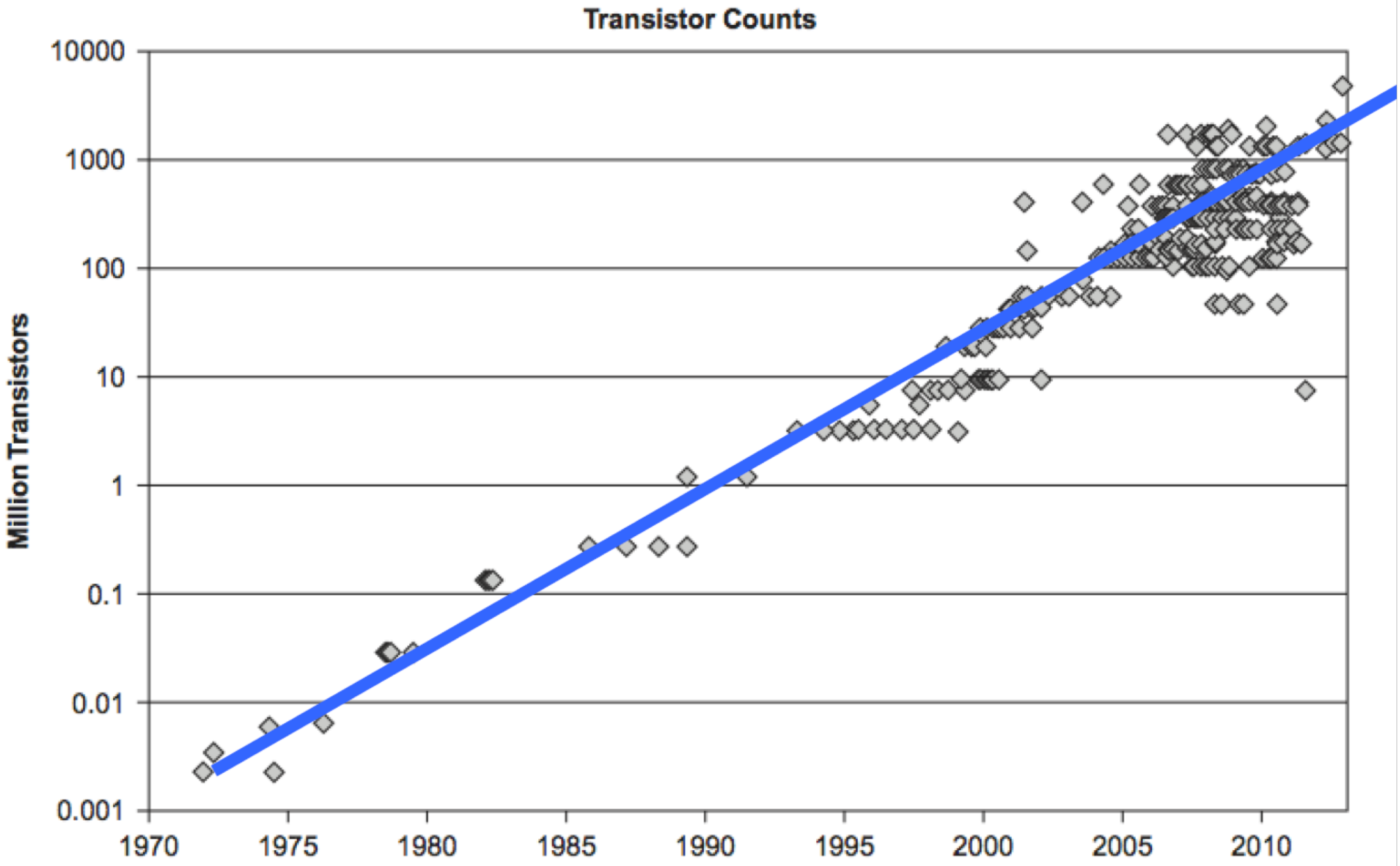


Hvorfor får vi multikjerne CPUer ?

- Hver 18-24 måned dobler antall transistorer vi kan få på en brikke: Moores lov
- Vi kan ikke lage raskere kretser fordi da vil vi ikke greie å luftkjøle dem (ca. 120 Watt på ca. 2x2 cm – varmere enn en rødglødende kokeplate). Og der er kvantemekaniske begrensninger også.
- Med f.eks dobbelt så mange transistorer ønsker vi oss egentlig en dobbelt så rask maskin, men det vi får er dessverre 'bare' dobbelt så mange CPU-kjerner.
- Med flere regnekverner (kjerner) må vi få opp hastigheten ved å lage parallelle programmer !

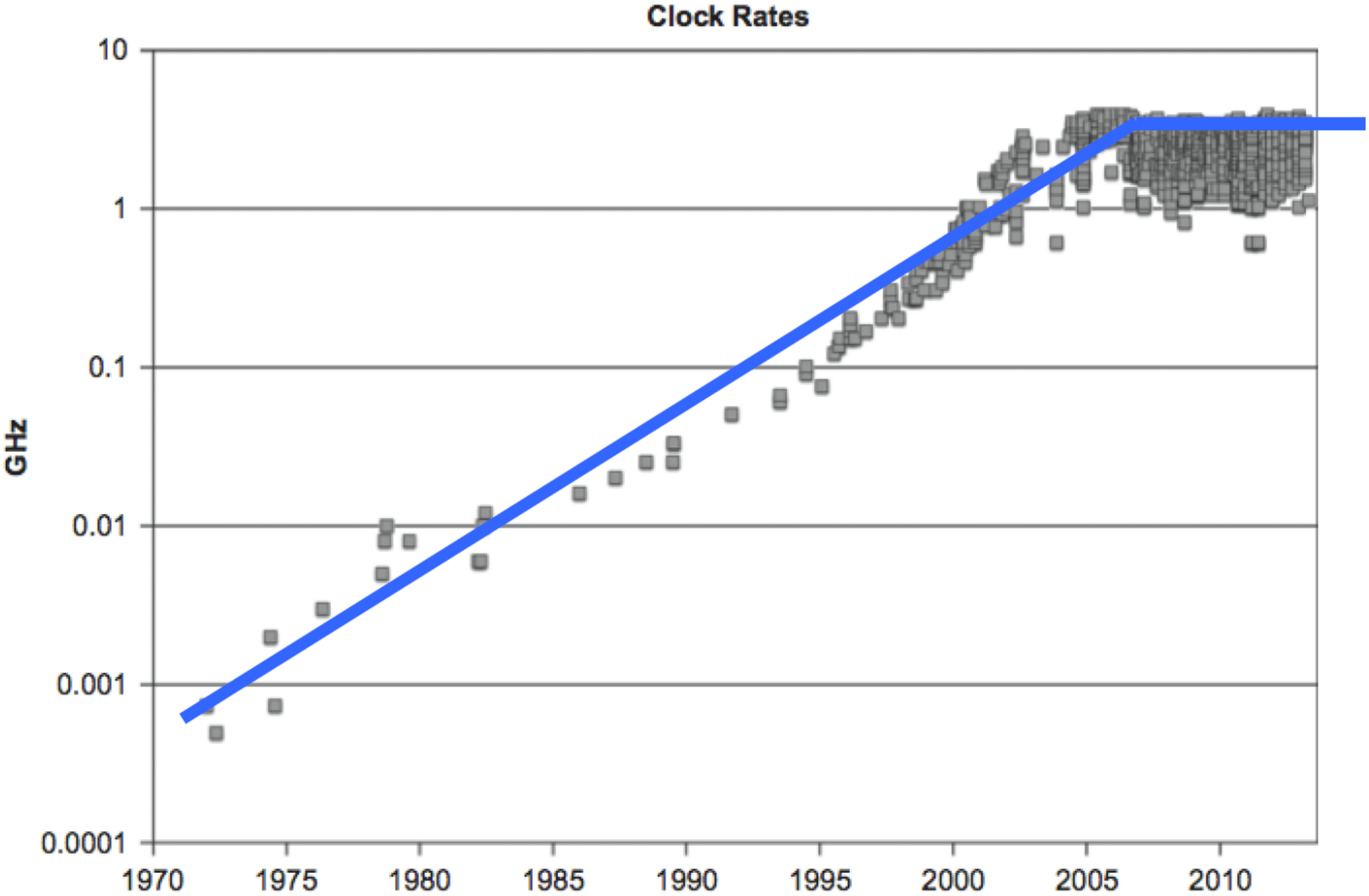
Transistors per Processor over Time

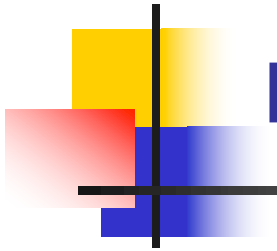
Continues to grow exponentially (Moore's Law)



Processor Clock Rate over Time

Growth halted around 2005

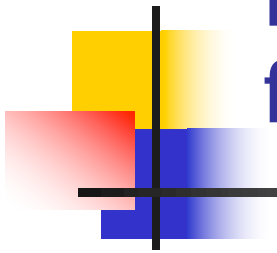




Helt avgjørede for oss – cache-hukommelse

- Hva er cache
  - Raskere (men også dyrere) hukommelse mellom hovedlageret og kjernene.
  - Vi må ha cache fordi det er så store hastighetsforskjeller mellom en CPU-kjerne og hovedlageret ('main memory')
  - Ofte nå 3-4 lag med cache hukommelser + et antall registere i kjernen (enda raskere enn cache-hukommelsene) som holder data eller intruksjoner
  - Når en kjerne trenger data eller en ny instruksjon (og den ikke har det i et register) leter den nedover i cache-hukommelsene. Først cache level 1 (L1), så L2 cachen, .. , før den går til hovedhukommelsen for data eller instruksjoner.
  - Det finns flere teknikker for å gjøre dette raskt (som pre-fetch , dvs at systemet henter neste data/instruksjon uten at kjernen eksplisitt har bedt om det)

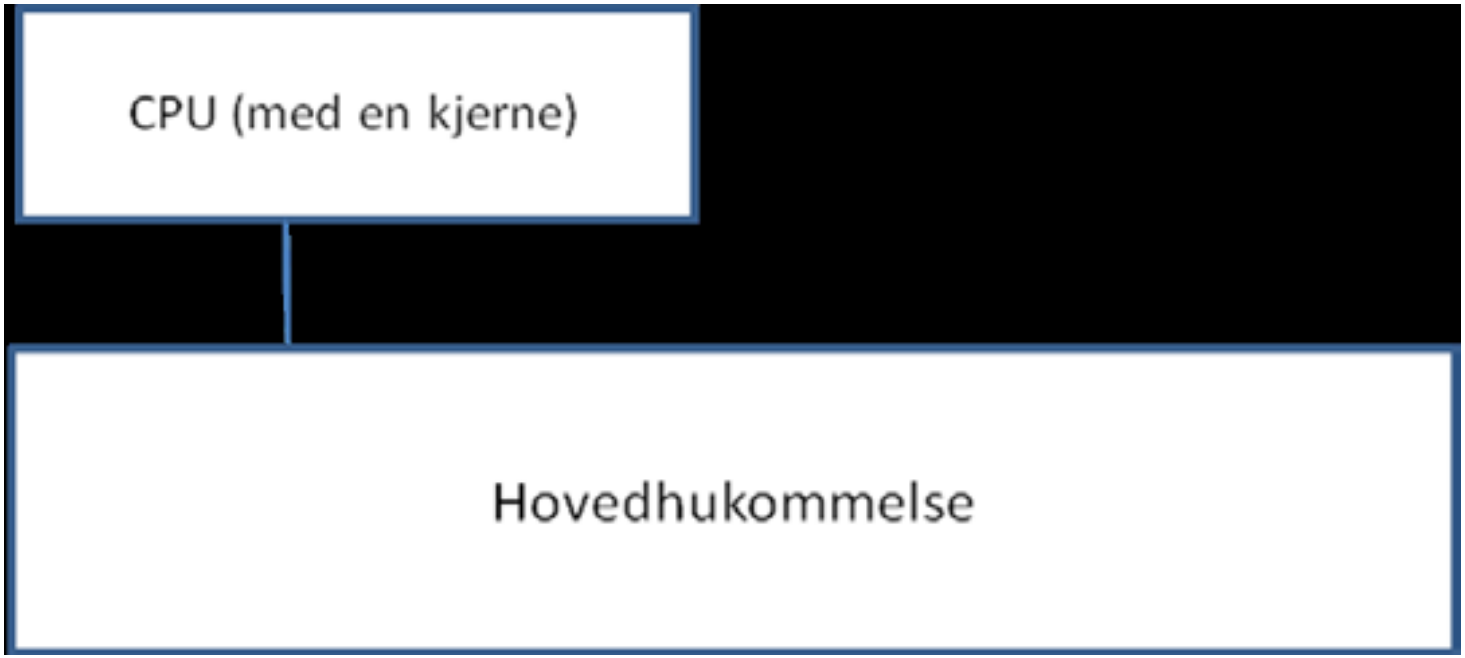
Hvordan tar vi hensyn til cache-systemet for å få raskere programmer?



- Vi ser bare på data-cachene (lite å hente på instruksjonene)
- Viktig å vite er at hver gang vi skal hente data i hovedlageret , får vi en cach-linje = 64 byte = f.eks 8 heltall (int)
- Det er svært begrenset plass i cachene, og en cach-linje som ikke har vært brukt på 'lenge' vil bli 'kastet ut'(overskrevet av en annen, nyere) cache-linje
- Slik er raskest:
  - Jobber på få data (korte deler av en array) 'lenge' av gangen – ikke hoppe rundt.
  - Helst gå forlengs eller baklengs gjennom data (arrayene) (i, i+1,.. eller: i, i-1,..)

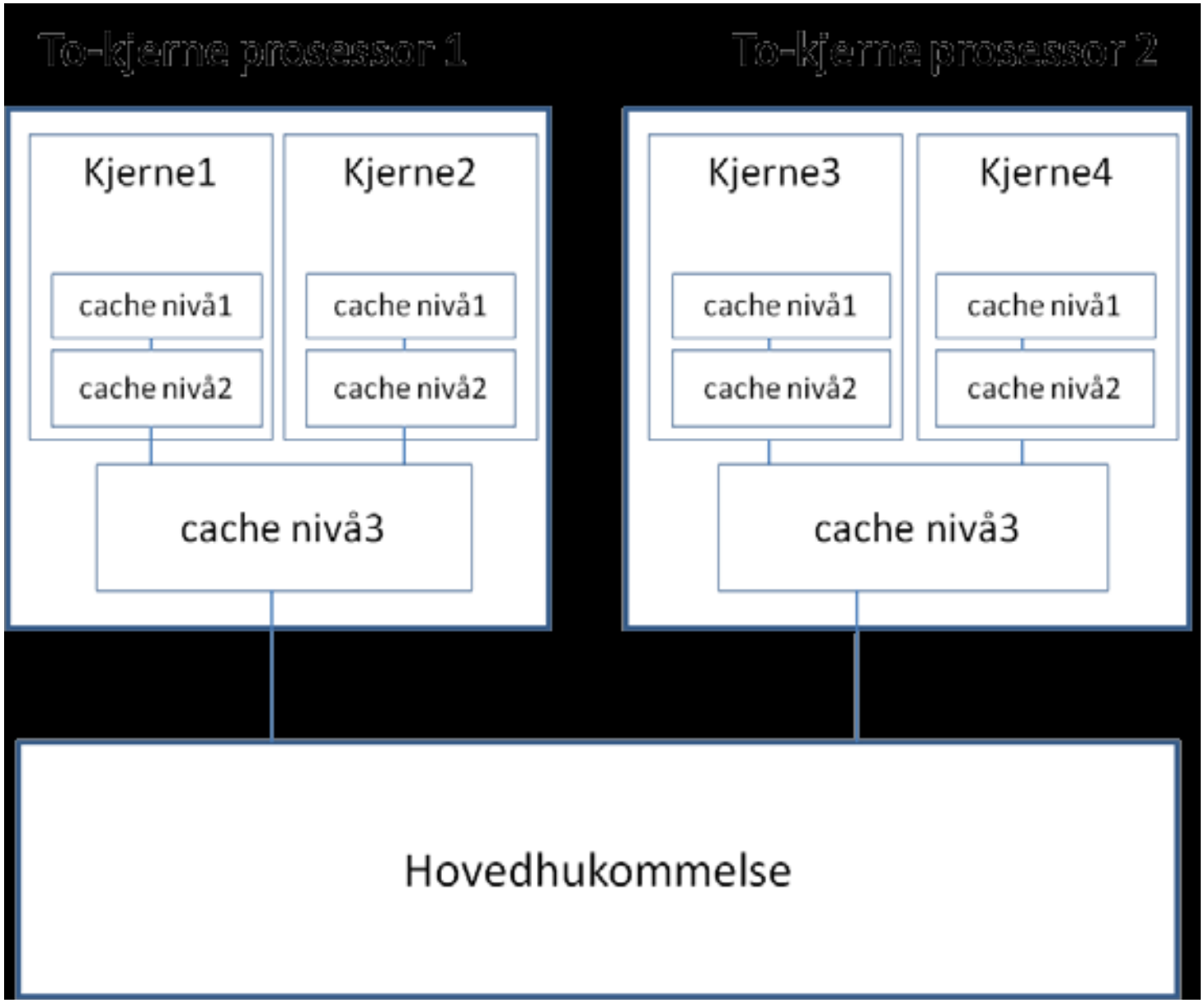
Vi må lage slike cache-vennlige programmer !

Maskin 1980 (uten cache)

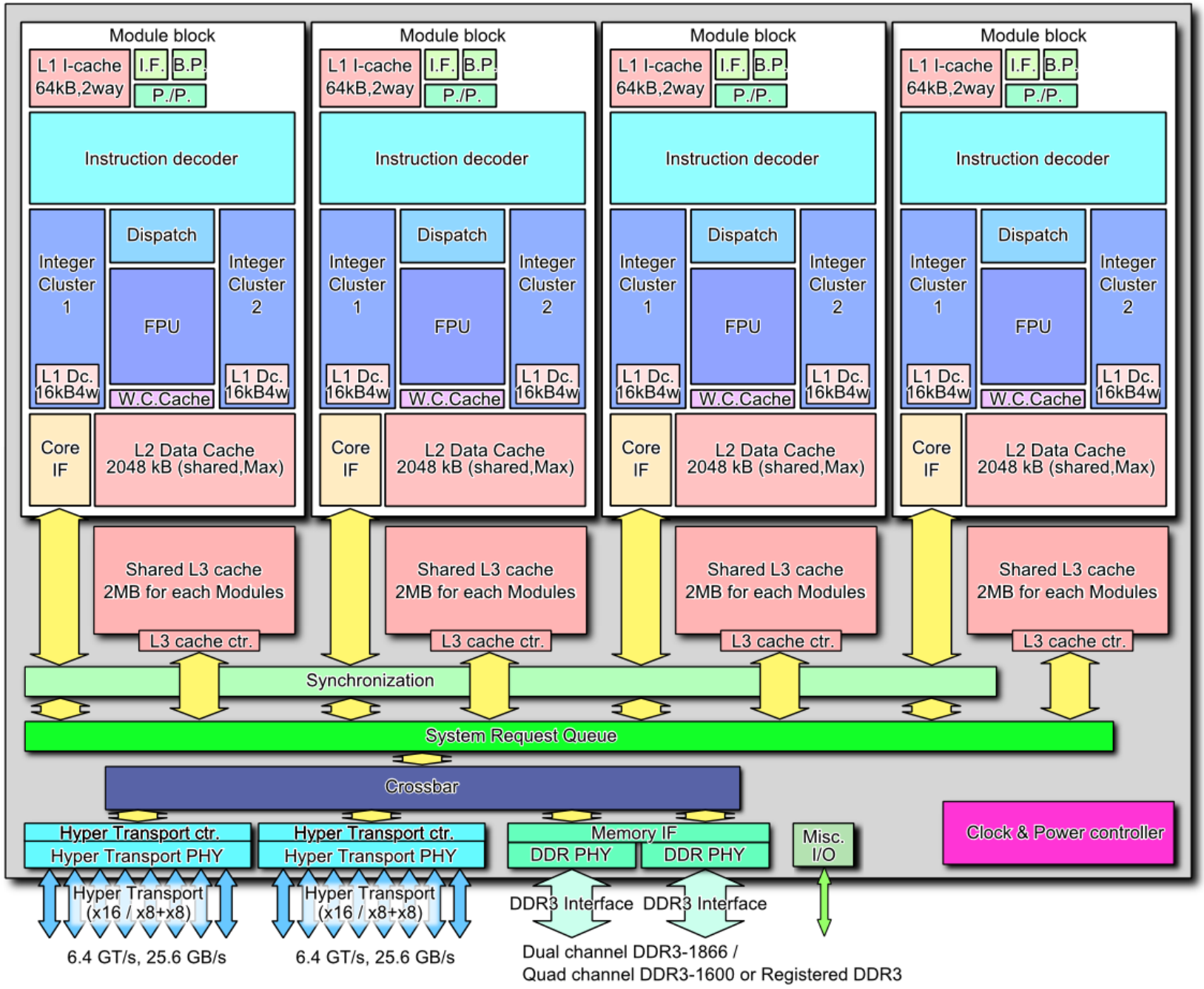


Figur 19.1 Skisse av en datamaskin i ca. 1980 hvor det bare var én beregningsenhet, en CPU, som leste sine instruksjoner og både skrev og leste data (variable) direkte i hovedhukommelsen. Intel 8080: 1 MHz CPU

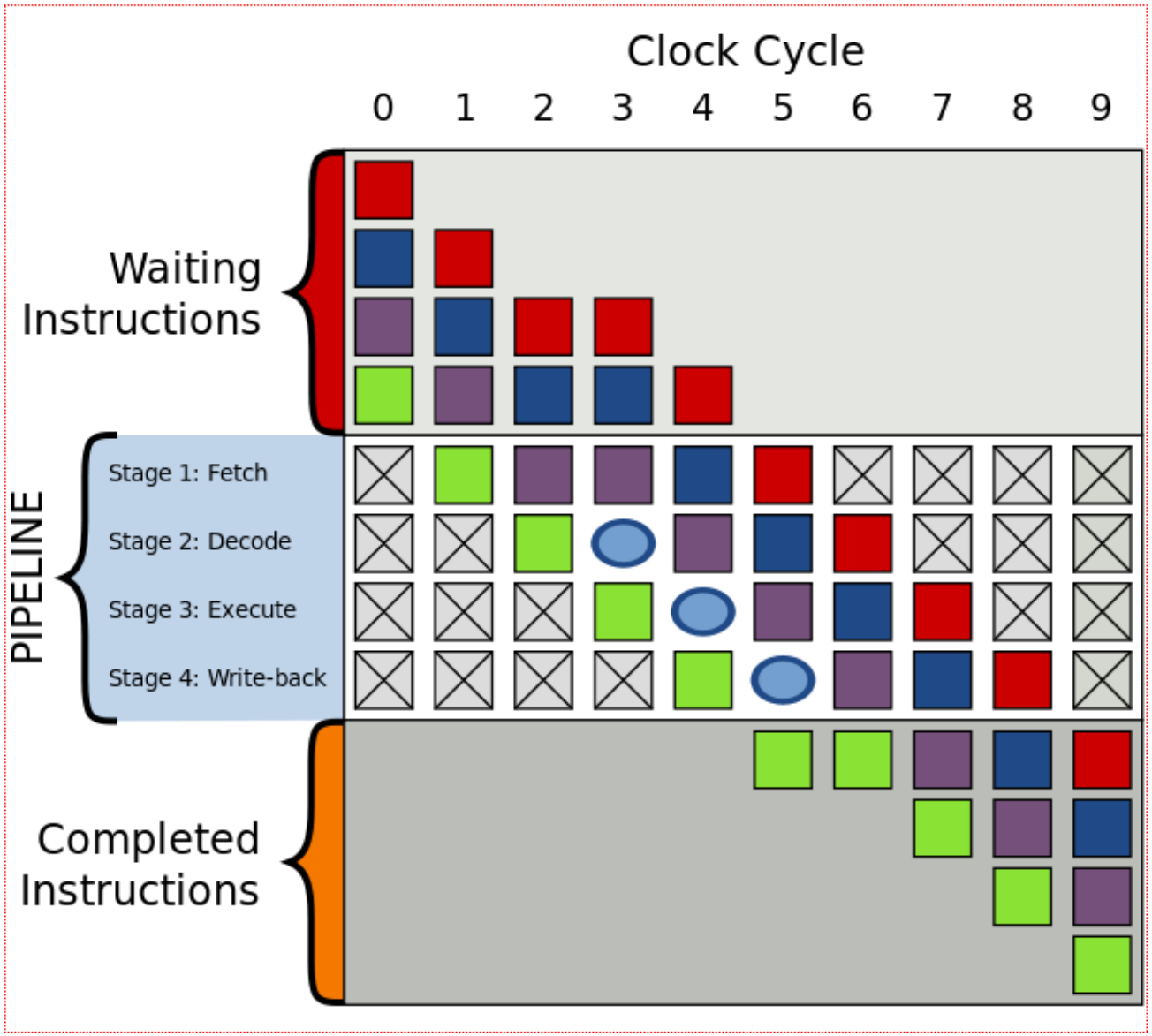
Maskin ca. 2010 med to dobbeltkjerne CPU-er



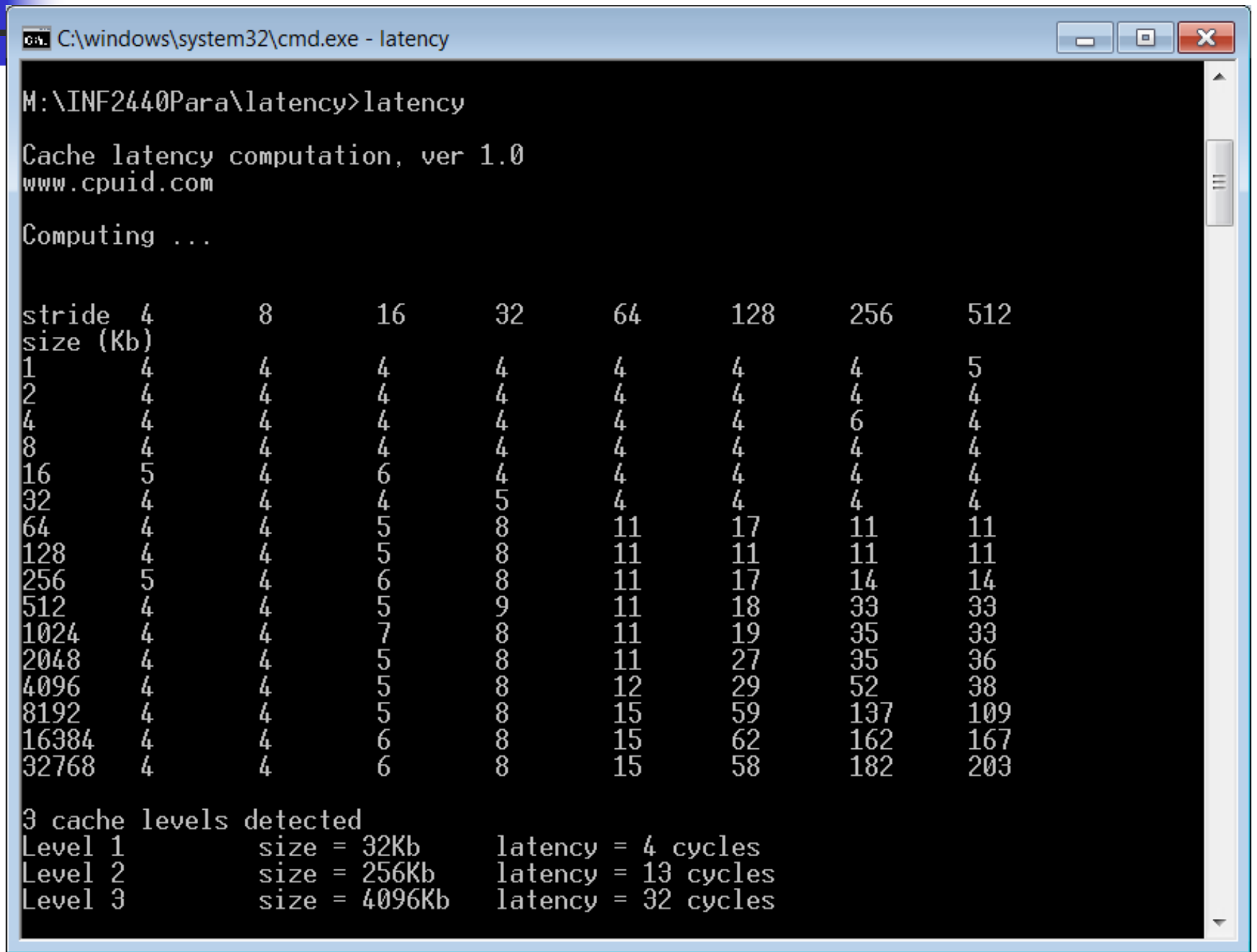
Hukommelses-systemet i en 4 kjerne CPU – mange lag og flere ulike beregningsmoduler i hver kjerne.:



Instruksjonsparallellitet i en CPU-kjerne. Pipeline – flere instruksjoner (her 4) utføres *samtidig* i raskest mulig rekkefølge.



Test av forsinkelse i data-cachene og hovedhukommelsen - latency.exe (fra CPUZ)



Oppsummering – ideen om at vi har uniform aksesstid i hukommelsen er helt galt

- Hukommelses-systemet i en multicore CPU ,Intel Core i5-459 3.3 GHz, – mange lag (typisk aksesstid i instruksjonssykler):

1. Registre i kjernen (1) – 8/32 registre
2. L1 cache (3-4) – 32 Kb
3. L2 cache (13) – 256 kb
4. L3 cache (32) – 8Mb
5. Hovedhukommelsen (virtuell hukommelse) (ca. 200) – 8-64 GB
6. Disken (15 000 000 roterende) = 5 ms – 1000 GB – 1-5 TB
   FlashDisk (ca 2 000 000 les, ca. 10 000 000 skriv) = ca. 1 ms

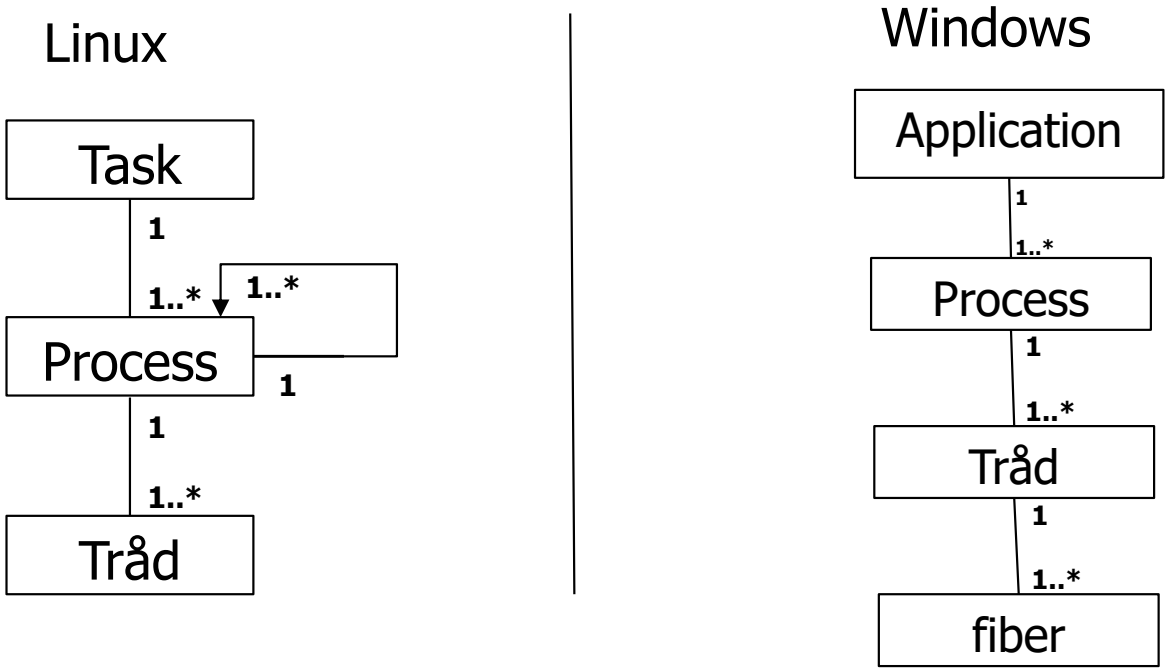
Vi kan ikke hente mer fra automatisk forbedring av hastigheten på våre programmer:

- **Ikke raskere maskiner** – luftkjølingsproblemet
- **Hovedhukommelsen - både *mye* langsommere enn CPU-ene (derfor cache), og det å sette stadig flere kjerner oppå en langsom hukommelse gir køer.**
- **Instruksjons-parallelliteten** i hver kjerne (pipelinen) er fult utnyttet – ikke mer å hente
- **Kompilatoren** – Java (etter ver 1.3) kompilerer videre til maskinkode og (etter ver 1.6) optimaliserer mye. JIT-kompilering. Ikke mulig å gjøre særlig mer effektiv

⇒ **Konklusjon** Skal vi ha raskere programmer, må vi som programmerere *selv* skrive parallelle løsninger på våre problemer.

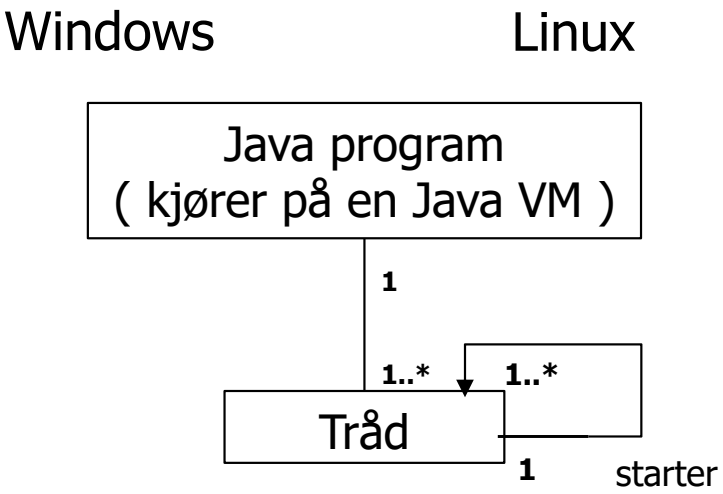
Operativsystemet og tråder

- De ulike operativsystemene (Linux, Windows) har ulike begreper for det som kjøres; mange nivåer (egentlig flere enn det som vises her)

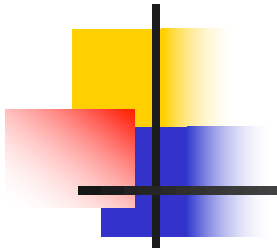


Heldigvis forenkler Java dette

Java forenkler dette ved å velge to nivåer



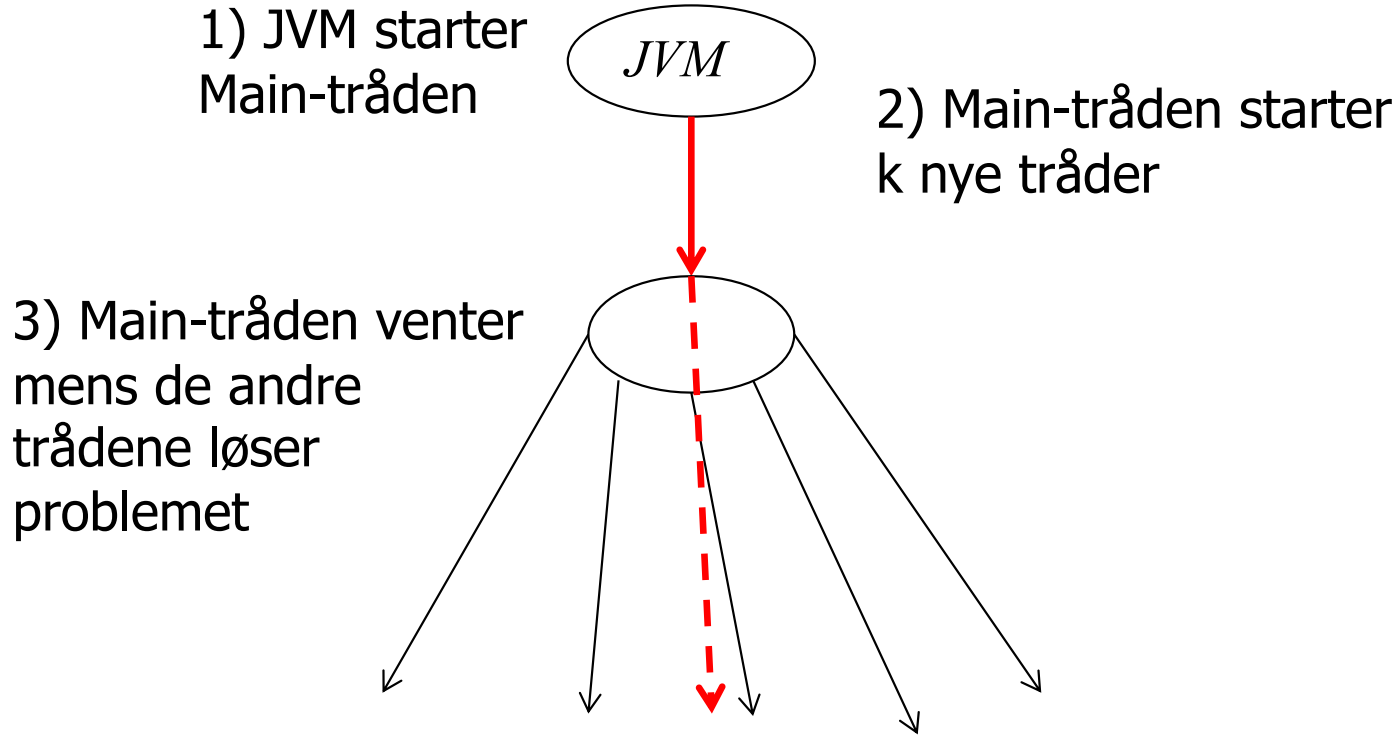
- **Alle trådene i et Java-program deler samme adresserom** (= samme plasser i hovedhukommelsen). Alle trådene kan lese og skrive i de variable (objektene) programmet har og ha adgang til samme kode (metodene i klassene).



Hva er tråder i Java ?

- I alle programmer kjører minst en tråd – main tråden (starter og kjører i public static void main).
- Main-tråden kan starte en eller flere andre, nye tråder.
- Enhver tråd som er startet, kan stoppes midlertidig eller permanent av:
  - Av seg selv ved kall på synkroniseringsobjekter hvor den må vente
  - Den er ferdig med sin kode (i metoden run), terminerer da
- Main-tråden og de nye trådene går i parallell ved at:
  - De kjører enten på hver sin kjerne
  - Hvis vi har flere tråder enn kjerner, vil klokka i maskinen sørge for at trådene av og til avbrytes og en annen tråd får kjøretid på kjernen.
- Vi bruker tråder til å parallellisere programmene våre

>java (også kalt JVM) starter main-tråden som igjen starter nye tråder



Tråder i Java er objekter av klassen Thread.

Konstruktør til Thread-klassen

Thread

`public Thread(Runnable target)`

Allocates a new `Thread` object. This constructor has the same effect as `Thread (null, target, gname)`, where `gname` is a newly generated name. Automatically generated names are of the form `"Thread-"+n`, where *n* is an integer.

Parameters:

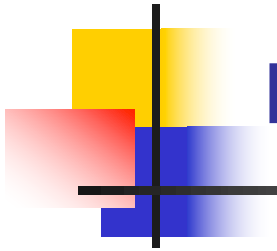
`target` - the object whose `run` method is invoked when this thread is started. If `null`, this classes `run` method does nothing.

- **`Runnable target`** er :
  - En klasse som implementerer grensesnittet 'Runnable'
- Det er en annen måte å starte er tråd hvor vi lager en subklasse av Thread (ikke fullt så fleksibel).

Tråder i Java

- Er én programflyt, dvs. en serie med instruksjoner som oppfører seg som ett vanlig, sekvensielt program – og kjører på én kjerne
- Det kan godt være (langt) flere tråder enn det er kjerner.
- En tråd er ofte implementert i form av en indre klasse i den klassen som løser problemet vårt (da får trådene greit aksess til **felles data**):

```
import java.util.concurrent.*;
class Problem { int [] fellesData ; // dette er felles, delte data for alle trådene
        public static void main(String [] args) {
                Problem p = new Problem();
                        p.utfoer();
                }
                void utfoer () { Thread  t = new Thread(new Arbeider());
                    t.start();
                 }

          class Arbeider implements Runnable {
               int i,lokalData; // dette er lokale data for hver tråd
                  public void run() {
                        // denne kalles når tråden er startet
                  }
           } // end indre klasse Arbeider
} // end class Problem
```

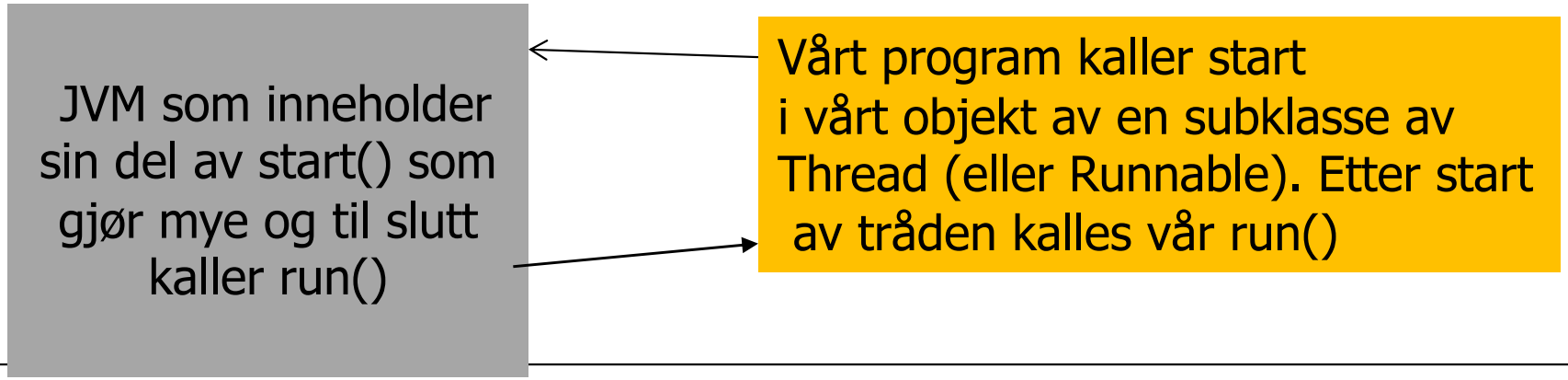


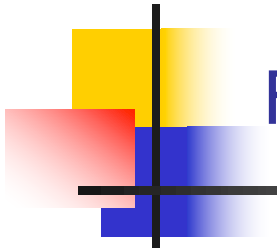
END OF LECTURE UKE1

- The first lecture, uke1, ended here, the remaining slides will be covered in the lecture in uke2 ☺

Tråder i Java

- En tråd er enten subklasse av Thread eller får til sin konstruktør et objekt av en klasse som implementerer Runnable.
- Poenget er at begge måtene inneholder en metode:
  - ’ public void run()’
- Vi kaller metoden start() i klassen Thread . Det sørger for at JVM starter tråden og at ’run()’ i vår klasse deretter kalles.





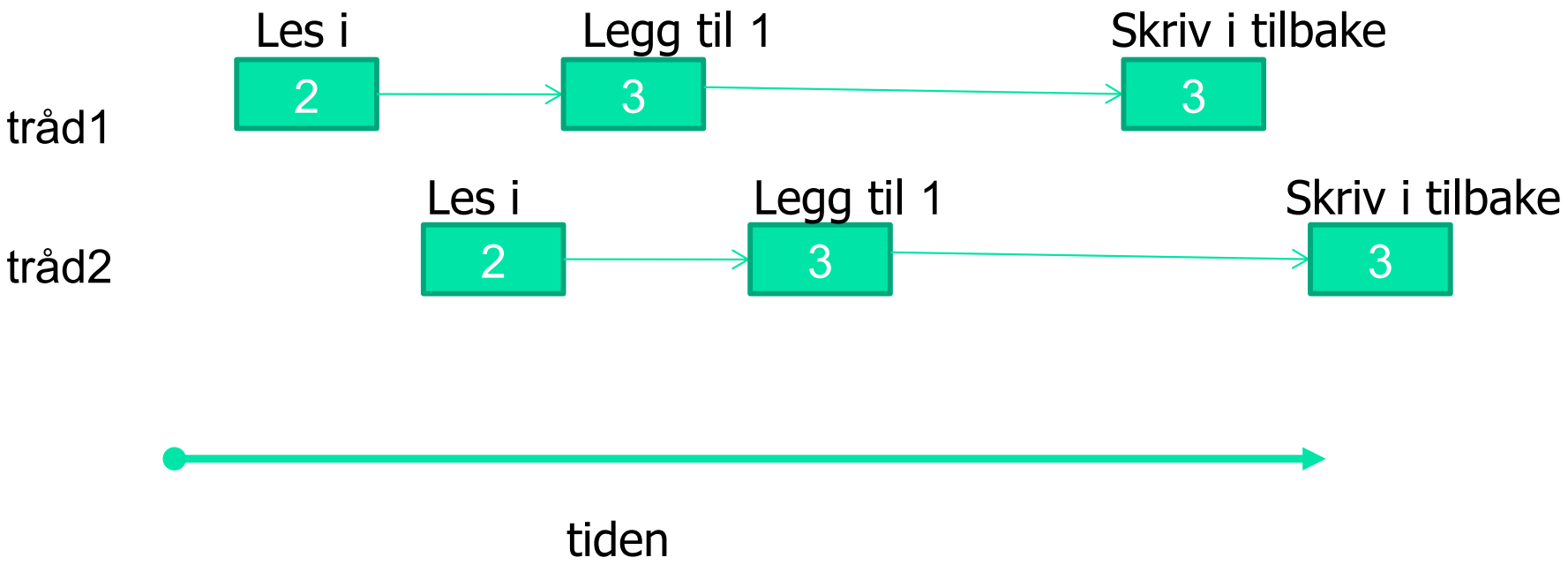
Flere problemer med parallellitet og tråder i Java

1. Operasjoner blandes (oppdateringer går tapt).
2. Oppdaterte verdier til felles data er ikke alltid synlig fra alle tråder (oppdateringer er ikke synlige når du trenger dem).
3. Synlighet har ofte med cache å gjøre.
4. The Java memory model (= hva skjer 'egentlig' når du kjører et Java-program).

- Vi må finne på 'skuddsikre' måter å programmere parallelle programmer
  - De er kanskje ikke helt tidsoptimale
  - Men de er lettere å bruke !!
  - Det er vanskelig nok likevel.
- Bare oversiktelige, 'enkle' måter å programmere parallelt er mulig i praksis

1) Ett problem i dag: operasjoner blandes ved samtidige oppdateringer

- Samtidig oppdatering - flere tråder sier gjentatte ganger: **i++ ;** der i er en felles int.
  - **i++** er 3 operasjoner: a) les i, b) legg til 1, c) skriv i tilbake
  - Anta i =2, og to tråder gjør i++
  - Vi kan få svaret 3 eller 4 (skulle fått 4!)
  - Dette skjer i praksis !



Test på i++; parallell

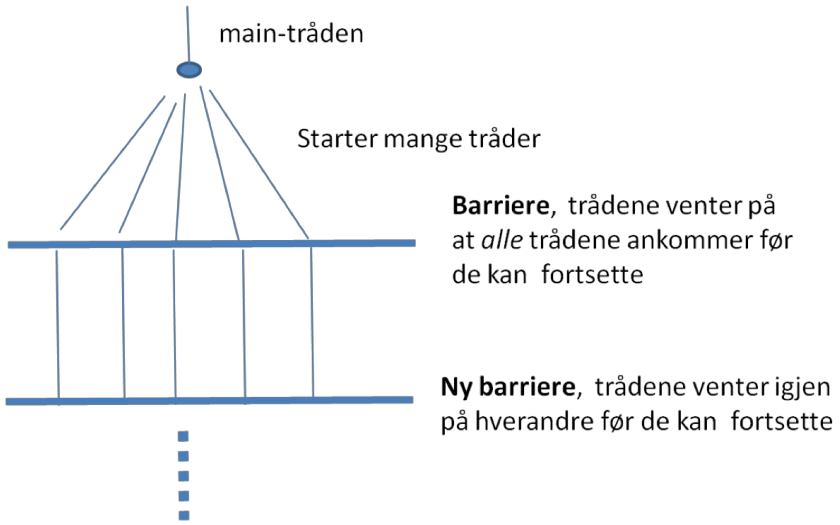
- Setter i gang **n tråder** (på en 2-kjerner CPU) som alle prøver å øke med 1 en felles variabel int i; 100 000 ganger uten synkronisering;

```
for (int j =0; j< 100000; j++) {
            i++;
}
```

- Vi fikk følgende feil - antall og %, (manglende verdier).
  Merk: Resultatene *varierer også mye* mellom hver kjøring :

Antall tråder n		**1**	**2**	**20**	**200**	**2000**
Svar	1.gang	100 000	200000	1290279	16940111	170127199
	2.gang	100 000	159234	1706068	16459210	164954894
Tap	1.gang	0 %	0%	35,5%	15,3%	14,9%
	2. gang	0%	20,4%	14,6%	17,7%	17,5%

Kommende program bruker CyclicBarrier. Hva gjør den?



- Man lager først ett, felles objekt `b` av klassen CyclicBarrier med et tall: `ant` til konstruktøren = det antall tråder den skal køe opp før alle trådene slippes fri 'samtidig'.
- Tråder (også main-tråden) som vil køe opp på en CyclicBarrier sier await() på den.
- De `ant-1` første trådene som sier await(), blir lagt i en kø.
- Når tråd nummer `ant` sier await() på `b`, blir alle trådene sluppet ut av køen 'samtidig' og fortsetter i sin kode.
- Det sykliske barriere objektet `b` er da med en gang klar til å være kø for nye, `ant` stk. tråder.

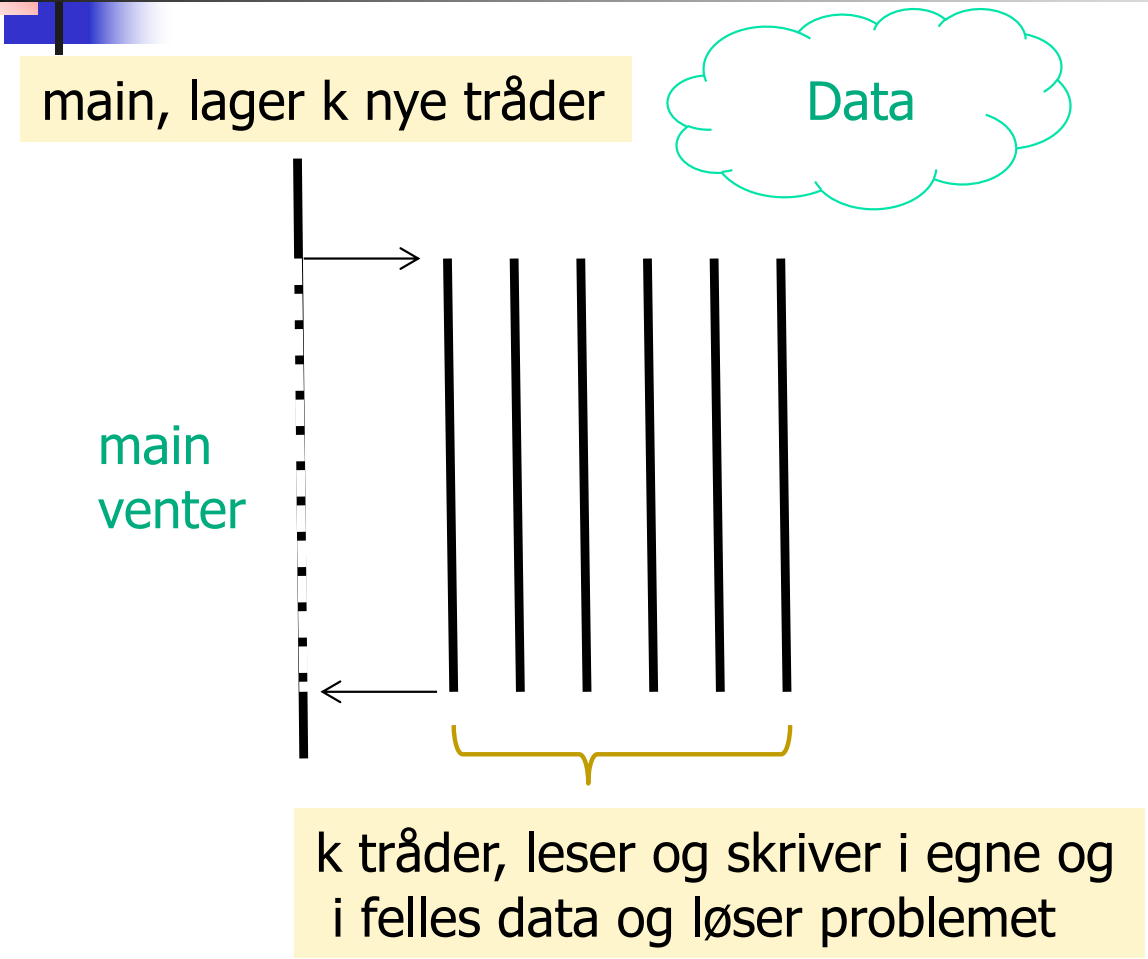
Praktisk: skal nå se på programmet som laget tabellen

```
import java.util.*;
import easyIO.*;
import java.util.concurrent.*;
/** Viser at manglende synkronisering på ett felles objekt gir feil – bare loesning 1) er riktig'*/

public class Parallell {
        int tall;                               // Sum av at 'antTraader' traader teller opp denne
        CyclicBarrier b ;                       // sikrer at alle er ferdige naar vi tar tid og sum
        int antTraader, antGanger ,svar; // Etter summering: riktig svar er:antTraader*antGanger

        // det kommer ialt 4 forsøk på å øke i, bare en av dem er riktig
        //synchronized void inkrTall(){ tall++;}      // 1) –OK fordi synkroniserer på ett objekt (p)
        void inkrTall() { tall++;}                    // 2) - feil

        public static void main (String [] args) {
                if (args.length < 2) {
                        System.out.println("bruk >java Parallell <antTraader> <n= antGanger>");
                }else{  int antKjerner = Runtime.getRuntime().availableProcessors();
                        System.out.println("Maskinen har "+ antKjerner + " prosessorkjerner.");
                        Parallell p =  new Parallell();
                        p.antTraader = Integer.parseInt(args[0]);
                        p.antGanger = Integer.parseInt(args[1]);
                        p.utfor();
        } } // end main
```

```
void utskrift (double tid) {
            svar = antGanger*antTraader;
            System.out.println("Tid "+antGanger+" kall * "+ antTraader+" Traader ="+
                                                    Format.align(tid,9,1)+ " millisek,");
             System.out.println(" sum:"+ tall +", tap:"+ (svar -tall)+" = "+
              Format.align( ((svar - tall)*100.0 /svar),12,6)+"%");

 } // end utskrift

  void  utfor () {   b = new CyclicBarrier(antTraader+1);       //+1, også main
                     long t = System.nanoTime();                 // start klokke

                     for (int j = 0; j< antTraader; j++) {
                           new Thread(new Para(j)).start();
                      }

                    try{   // main thread venter
                           b.await();
                     } catch (Exception e) {return;}
                     double tid = (System.nanoTime()-t)/1000000.0;
                     utskrift(tid);
    } // utfor
```

```
class Para implements Runnable{
        int ind;
        Para(int ind) { this.ind =ind;}

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

        //    void inkrTall() { tall++;}                    // 3) Feil - usynkronisert
        //    synchronized void inkrTall(){ tall++;}  // 4) Feil – kallene synkroniserer på
                                                          //      hvert sitt objekt
    } // end class Para
} // END class Parallell
```

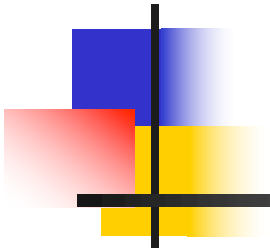
Husk: Vanligste oppsett av main-tråden + k tråder



Hver av trådene (main + k nye) er sekvensielle programmer.
Problemet er at de samtidig *ikke kan skrive* på felles data

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) men har også egne lokale data og metoder
- Stygg feil vi kan gjøre: Samtidig oppdatering (=skriving) på delte data (eks: i++)
  - Dette løses ved synkronisering .
  - Alle tråder som vil skrive må køes opp på **samme** synkroniseringsvariabel (som er et objekt) slik at bare én tråd slipper til av gangen.
  - ***Alle objekter*** kan nyttes som en synkroniseringsvariabel, og da kan vi bruke enten en synchronized metode,
  - eller objekter av spesielle klasser som:
    - CyclicBarrier
    - Semaphore (undervises senere)
  - De inneholder metoder som await(), som gjør at tråder venter.
- Senere vil vi lære å dele opp et problem i mindre biter til hver tråd og **ikke** synkronisere for mye (tar for mye tid)



IN3030 – Effektiv parallellprogrammering
Uke 1 2. del, våren 2019

Eric Jul

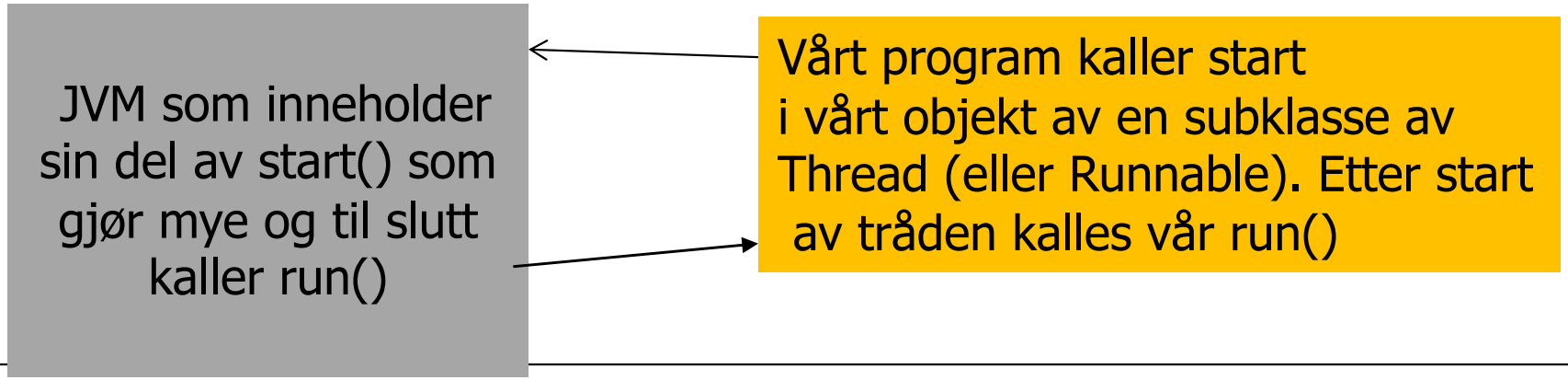
Professor

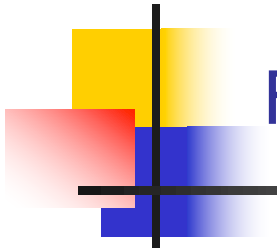
PSE

Institutt for Informatikk

Tråder i Java

- En tråd er enten subklasse av Thread eller får til sin konstruktør et objekt av en klasse som implementerer Runnable.
- Poenget er at begge måtene inneholder en metode:
  - ’ public void run()’
- Vi kaller metoden start() i klassen Thread . Det sørger for at JVM starter tråden og at ’run()’ i vår klasse deretter kalles.





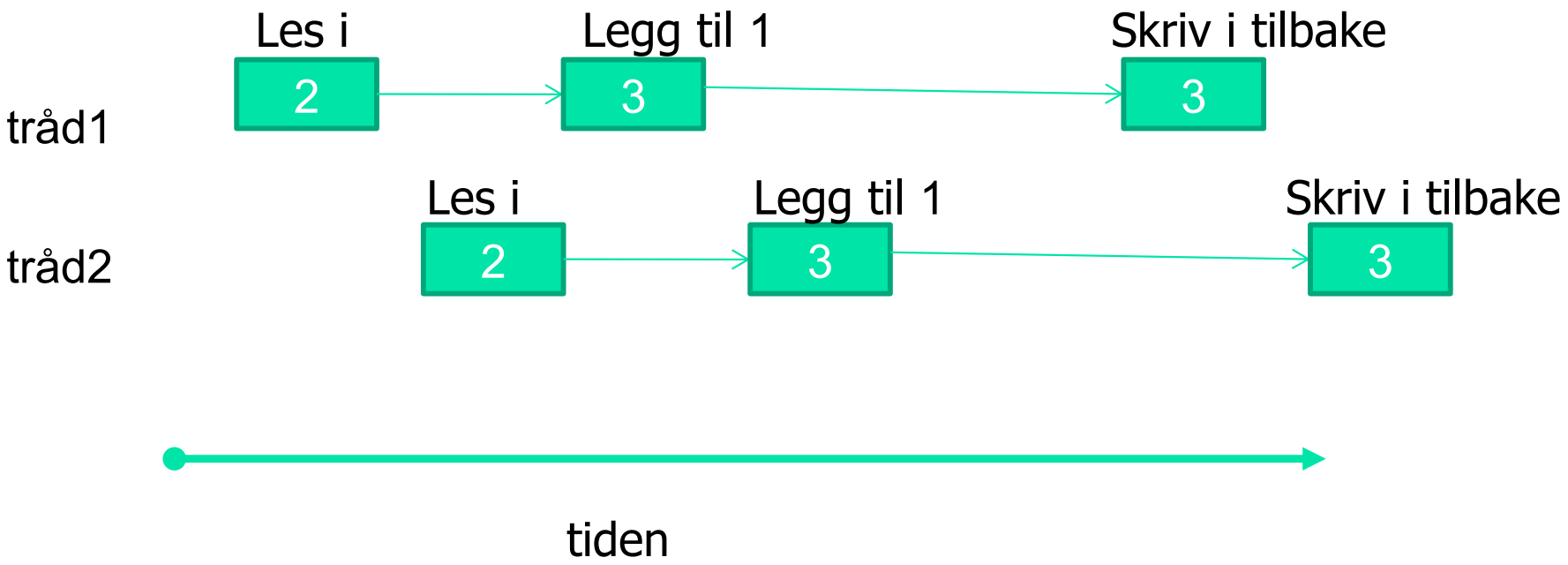
Flere problemer med parallellitet og tråder i Java

1. Operasjoner blandes (oppdateringer går tapt).
2. Oppdaterte verdier til felles data er ikke alltid synlig fra alle tråder (oppdateringer er ikke synlige når du trenger dem).
3. Synlighet har ofte med cache å gjøre.
4. The Java memory model (= hva skjer 'egentlig' når du kjører et Java-program).

- Vi må finne på 'skuddsikre' måter å programmere parallelle programmer
  - De er kanskje ikke helt tidsoptimale
  - Men de er lettere å bruke !!
  - Det er vanskelig nok likevel.
- Bare oversiktelige, 'enkle' måter å programmere parallelt er mulig i praksis

1) Ett problem i dag: operasjoner blandes ved samtidige oppdateringer

- Samtidig oppdatering - flere tråder sier gjentatte ganger: **i++ ;** der i er en felles int.
  - **i++** er 3 operasjoner: a) les i, b) legg til 1, c) skriv i tilbake
  - Anta i =2, og to tråder gjør i++
  - Vi kan få svaret 3 eller 4 (skulle fått 4!)
  - Dette skjer i praksis !



Test på i++; parallell

- Setter i gang **n tråder** (på en 2-kjerner CPU) som alle prøver å øke med 1 en felles variabel int i; 100 000 ganger uten synkronisering;

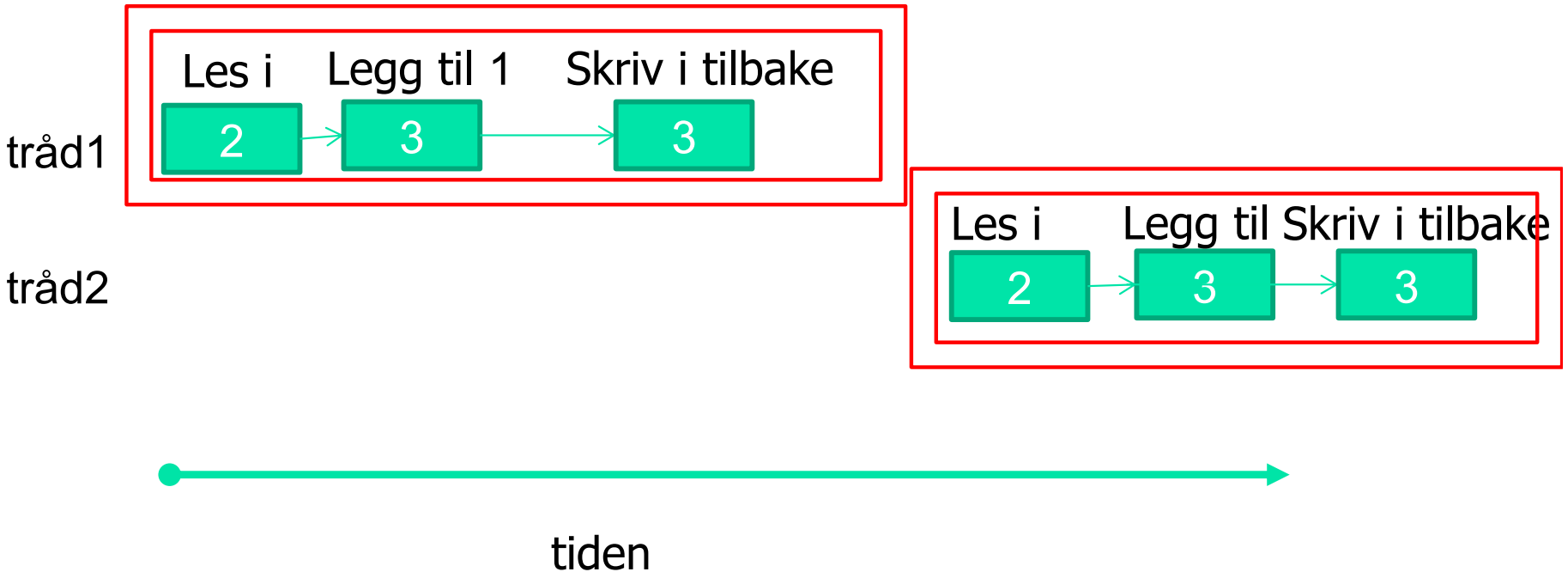
```
for (int j =0; j< 100000; j++) {
            i++;
}
```

- Vi fikk følgende feil - antall og %, (manglende verdier).
  Merk: Resultatene *varierer også mye* mellom hver kjøring :

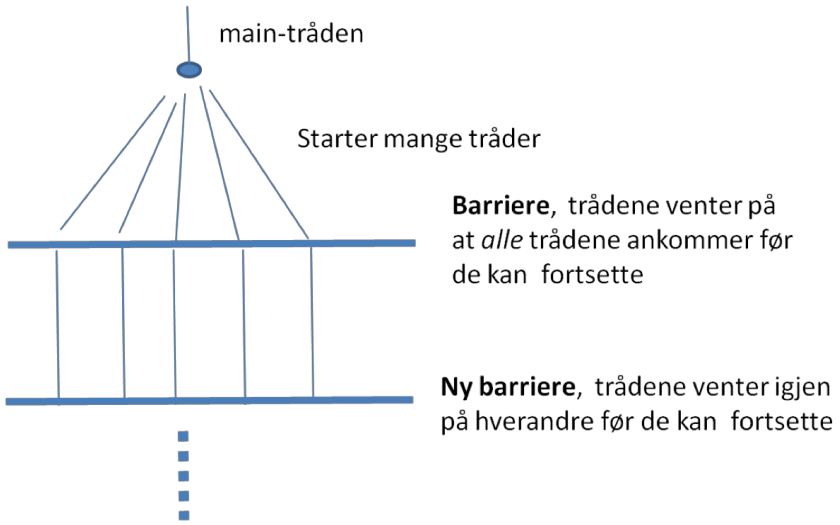
Antall tråder n		**1**	**2**	**20**	**200**	**2000**
Svar	1.gang	100 000	200000	1290279	16940111	170127199
	2.gang	100 000	159234	1706068	16459210	164954894
Tap	1.gang	0 %	0%	35,5%	15,3%	14,9%
	2. gang	0%	20,4%	14,6%	17,7%	17,5%

Synkronisering

- Løsning af problem med samtidig oppdatering
  - **i++** er 3 operasjoner: a) les i, b) legg til 1, c) skriv i tilbake
  - Critical section
  - Beskytt les-legg til-skriv i critical section

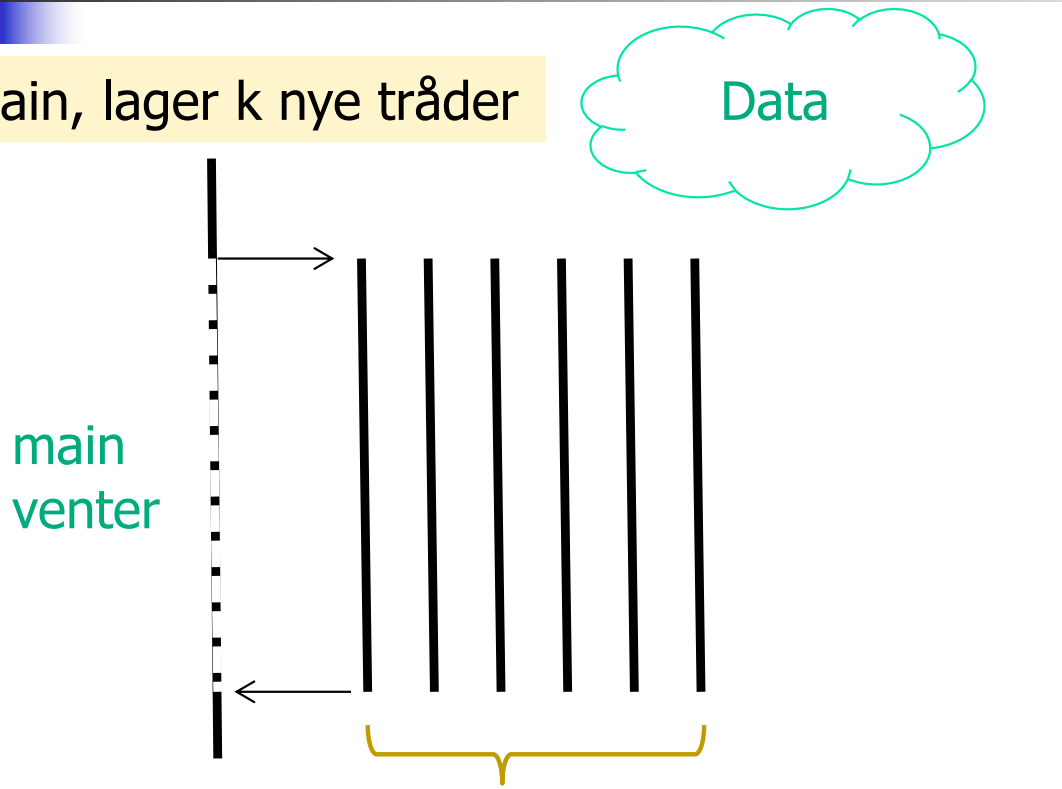


Kommende program bruker CyclicBarrier. Hva gjør den?

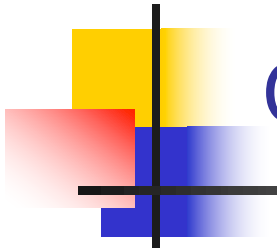


- Man lager først ett, felles objekt `b` av klassen CyclicBarrier med et tall: `ant` til konstruktøren = det antall tråder den skal køe opp før alle trådene slippes fri 'samtidig'.
- Tråder (også main-tråden) som vil køe opp på en CyclicBarrier sier await() på den.
- De `ant-1` første trådene som sier await(), blir lagt i en kø.
- Når tråd nummer `ant` sier await() på `b`, blir alle trådene sluppet ut av køen 'samtidig' og fortsetter i sin kode.
- Det sykliske barriere objektet `b` er da med en gang klar til å være kø for nye, `ant` stk. tråder.

Vanligste oppsett av main-tråden + k tråder

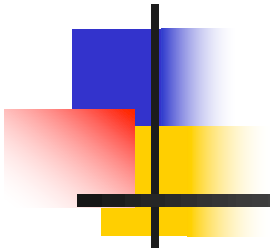


Hver av trådene (main + k nye) er sekvensielle programmer.
Synkronisering via cyclic barrier.
Problemet er at de samtidig *ikke kan skrive* på felles data.



Oppsummering – Uke1-uke2-del

- Stygg feil vi kan gjøre: Samtidig oppdatering (=skriving) på delte data (eks: i++)
  - Dette løses ved synkronisering
  - Alle tråder som vil skrive må køes opp på **samme** synkroniseringsvariabel (som er et objekt) slik at bare én tråd slipper til av gangen.
  - ***Alle objekter*** kan nyttes som en synkroniseringsvariabel, og da kan vi bruke enten en synchronized metode,
  - eller objekter av spesielle klasser som:
    - CyclicBarrier
    - Semaphore (undervises senere)
  - De inneholder metoder som await(), som gjør at tråder venter.
- Senere vil vi lære å dele opp et problem i mindre biter til hver tråd og **ikke** synkronisere for mye (tar for mye tid)

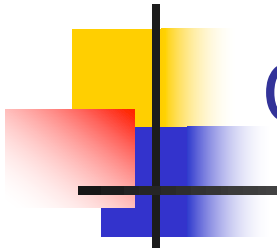


INF3030 – Effektiv parallellprogrammering
Uke 2 våren 2019
Synkronisering og tidstakning

Eric Jul

PSE,

Institutt 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)

Tråder i Java (lett revidert og kompilerbar)

- En tråd er én programflyt, dvs. en serie med instruksjoner som oppfører seg som ett program – og kjører på en kjerne
- Det kan godt være (langt) flere tråder enn det er kjerner.
- En tråd er ofte implementert i form av en indre klasse i den klassen som løser problemet vårt (da får de greit **felles data**):

```
import java.util.concurrent.*;
class Problem { int [] fellesData ; // dette er felles, delte data for alle trådene
                public static void main(String [] args) {
                  Problem p = new Problem();   // MÅ alltid lage ett objekt av den ytre klassen
                        p.utfoer();            // før det lages objekter av indre klasser
                 }
                 void utfoer () { Thread t = new Thread(new Arbeider());
                     t.start();
                  }

            class Arbeider implements Runnable {
                int i, lokalData; // dette er lokale data for hver tråd
                   public void run() {
                         // denne kalles når tråden er startet
                   }
    } // end indre klasse Arbeider
} // end class Problem
```

Flere tråder samtidig oppdatering av en variabel : i

- Alle trådene (1,2 20 , 200 og 2000) prøver samtidig å utføre i++ 100 000 ganger
- Vi skal se på programmet som produserte dette:

Antall tråder n Riktig svar:	**1** 100 000	**2** 200000	**20** 2000000	**200** 2000000	**2000** 200000000
Svar 1.gang 2. gang	100 000 100 000	200000 159234	1290279 1706068	16940111 16459210	170127199 164954894
Tap 1.gang 2. gang	0 % 0%	0% 20,4%	35,5% 14,6%	15,3% 17,7%	14,9% 17,5%

Programmet som laget tabellen

kode for main-tråden
kode for trådene

```
import java.util.*;I
import easyIO.*;
import java.util.concurrent.*;
/** Viser at manglende synkronisering på ett felles objekt gir feil – bare loesning 1) er riktig'*/

public class Parallell {
        int tall;                       // Sum av at 'antTraader' traader teller opp denne
        CyclicBarrier b ;               // sikrer at alle er ferdige naar vi tar tid og sum
        int antTraader, antGanger ,svar; // Etter summering: riktig svar er:antTraader*antGanger

        //synchronized void inkrTall(){ tall++;}      // 1) –OK fordi synkroniserer på samme objekt p
          void inkrTall() { tall++;}                  // 2) - feil

        public static void main (String [] args) {
                if (args.length < 2) {
                        System.out.println("bruk >java Parallell <antTraader> <n= antGanger>");
                }else{
                        int antKjerner = Runtime.getRuntime().availableProcessors();
                        System.out.println("Maskinen har "+ antKjerner + " prosessorkjerner.");
                        Parallell p =  new Parallell();
                         p.antTraader = Integer.parseInt(args[0]);
                        p.antGanger  = Integer.parseInt(args[1]);
                        p.utfor();
                 }
         } // end main
```

```
void utskrift (double tid) {
            svar = antGanger*antTraader;
            System.out.println("Tid "+antGanger+" kall * "+ antTraader+" Traader ="+
                                                    Format.align(tid,9,1)+ " ms,");
             System.out.println(" sum:"+ tall +", tap:"+ (svar -tall)+" = "+
            Format.align( ((svar - tall)*100.0 /svar),12,6)+"%");

 } // end utskrift

  void  utfor () {  // Denne kjøres bare av main-tråden
                    b = new CyclicBarrier(antTraader+1);       //+1, ogsaa main
                    long t = System.nanoTime();                // start klokke

                    for (int j = 0; j< antTraader; j++) {
                          new Thread( new Para(j) ).start();
                     }

                   try{   // main thread venter på at alle trådene er ferdige
                          b.await();
                    } catch (Exception e) {return;}
                   double tid = (System.nanoTime()-t)/1000000.0;
                   utskrift(tid);
    } // utfor
```

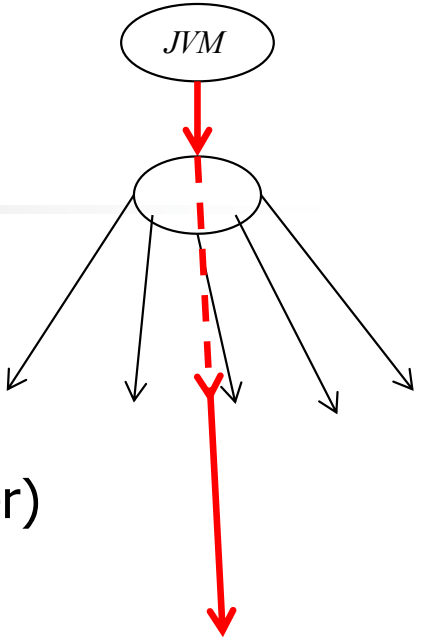
```
class Para implements Runnable{
                int ind;
                Para(int iind) { this.ind =ind;}

                public void run() { // Kjøres av hver tråd
                        for (int j = 0; j< antGanger; j++) {
                                inkrTall();
                          }
                          try {  // wait on all other threads + main
                                  b.await();
                          } catch (Exception e) {return;}
                   } // end run

                 //   void inkrTall() { tall++;}                    // 3) Feil - usynkronisert
                //    synchronized void inkrTall(){ tall++;}  // 4) Feil – kallene synkroniserer på
                                                                  //      hvert sitt objekt
    } // end class Para
} // END class Parallell
```

Hvilke typer problem egner seg for parallelle løsninger?

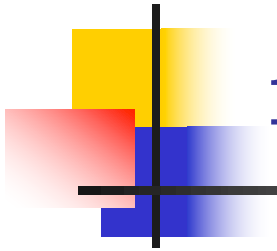
1. Kompleksitetsklasse:
   - O(1), O(logn), O(n), O(n*logn), O($n^{1,5}$), O(n^2), …,NP
2. Størrelsen på data: n
   - Sorterer vi 100 eller 100 million tall?
   - Multipliserer vi to 4x4 matriser eller to 2000x2000 matriser?
3. Må vi synkronisere på delte data, må antall synkroniseringer være (minst) en orden lavere enn algoritmen pga. 'mye' overhead ved synkronisering.
4. Å starte flere tråder tar ca. 3-5 millisekunder – den første tråden tar lengst tid.

Dette skal vi se på utover i kurset, med unntak av O(1) – (konstant eksekveringstid uavhengig av datamengden som klart **ikke** egner seg for parallellisering) kan det meste gis en mer effektiv parallell implementasjon ***hvis n er stor nok***
(eller sagt på en annen måte: **hvis kjøretiden er > 10 millisekunder).**

Plan for resten av Uke2

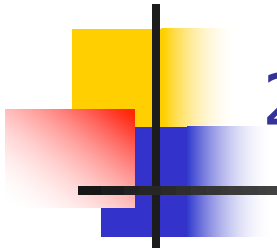


- I) Om å avslutte parallelle tråder
  - La dem bli ferdige med run-metoden, Hvordan teste at alle er ferdige ?
  - Synkronisert avslutning (Semaphore, CyclicBarrier)
  - new Thread – join() – avslutning
- II) Ulike synkroniseringsprimitiver
  - Vi skal bare lærte oss noen få - ett tilstrekkelig sett
- III) Hvor mye tid bruker parallelle programmer
  - JIT-kompilering
  - Overhead ved start
  - Synkronisering underveis i beregningene
  - Operativsystem og søppeltømming
- IV) 'Lover' om kjøretid
  - Amdahl lov
  - Gustafsons lov



1) Avslutning med en CyclicBarrier

- En CyclicBarrier (cb= new CyclicBarrier (n+1))
  - Er tenkt som et ventested, en bom/grind for et antall (i dette tilfellet for n+1) tråder - de n 'nye' trådene + main. Alle må vente når de sier sier cb.await() til sistemann ankommer køen, og **da** kan alle fortsette.
  - Trådene kan da være ferdige med en beregning kan selv avslutte med å bli ferdige med sin run() -kode. Main-tråden forsetter, og vet at de andre trådene er ferdige. Main-tråden kan da bruke resultatene fra trådene.
  - Den sykliske barrieren cb er da strakt klar til å køe nye n tråder som sier cb.await() , .. osv
  - cb.await() sies inne i en try-catch blokk



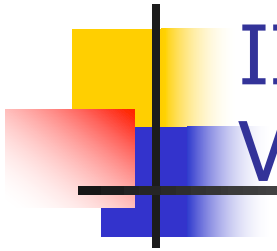
2) Avslutning med en Semaphore

- En Semaphore (sf = new Semaphore(-n+1))
  - Administrerer (i dette tilfellet) –n+1 stk. **tillatelser**.
  - To sentrale primitiver:
    - sf.acquire() – ber om **en** tillatelse. Antall tillatelser i sf blir da 1 mindre hvis antallet er >0. Hvis det ikke er noen ledig tillatelse, må tråden vente i en kø (inne i en try-catch blokk)
    - sf.release() – gir **én** tillatelse tilbake til semaforen sf. Ikke try-catch blokk (Den tillatelsen som gis tilbake behøver ikke vært 'fått' ved hjelp av aquire() ; den er bare et tall).
  - Avlutning med Semaphore sf:
    - Maintråden sier sf.acquire() – og må vente på at det er minst en tillatelse i sf.
    - Alle de n nye trådene sier sf.release() når de terminerer, og når den siste sier sf.release() blir det 1 tillatelse ledig og main fortsetter.
    - Ikke syklisk.

3) Avslutning med join() - enklest

- Logikken er her at i den rutinen hvor alle trådene lages, legges de også inn i en array. Main-tråden legger seg til å vente på den tråden som den har peker til skal terminere selv.
  Venter på alle trådene etter tur at de terminerer:

```
 // main –tråden i konstruktøren
Thread [] t = new Thread[n];
for (int i = 0; i < n; i++) {
   t[i] = new Thread (new Arbeider(..));
   t[i].start();
 }
  ……….
  // main vil vente her til trådene er ferdige
for(int i = 0; i < n; i++) {
       try{ t[i].join();
        }catch (Exception e){return;};
}     ……..
```



II) Mange ulike synkroniserings primitiver Vi skal bare lære noen få !

- java.util.concurrent

Classes

AbstractExecutorService
ArrayBlockingQueue
ConcurrentHashMap
ConcurrentLinkedDeque
ConcurrentLinkedQueue
ConcurrentSkipListMap
ConcurrentSkipListSet
CopyOnWriteArrayList
CopyOnWriteArraySet
CountDownLatch
CyclicBarrier
DelayQueue
Exchanger
ExecutorCompletionService
ecutor
eadPoolExecutor
ThreadPoolExecutor.AbortPolicy
ThreadPoolExecutor.CallerRunsPolicy
ThreadPoolExecutor.DiscardOldestPolicy
ThreadPoolExecutor.DiscardPolicy
Semaphore
SynchronousQueue
ThreadLocalRandom
ThrExecutors
ForkJoinPool
ForkJoinTask
ForkJoinWorkerThread
FutureTask
LinkedBlockingDeque
LinkedBlockingQueue
LinkedTransferQueue
Phaser
PriorityBlockingQueue
RecursiveAction
RecursiveTask
ScheduledThreadPoolEx

Interfaces

BlockingDeque
BlockingQueue
Callable
CompletionService
ConcurrentMap
ConcurrentNavigableMap
Delayed
Executor
ExecutorService
ForkJoinPool.ForkJoinWorkerThreadFacto
ry
ForkJoinPool.ManagedBlocker
Future
RejectedExecutionHandler
RunnableFuture
RunnableScheduledFuture
ScheduledExecutorService
ScheduledFuture
ThreadFactory
TransferQueue

java.util.concurrent.atomic

De har samme virkning (semantikk) som volatile variable (forklares senere), men kan gjøre mer sammensatte operasjoner. Mye raskere enn sychronized methods.

Classes

AtomicBoolean
AtomicInteger
AtomicIntegerArray
AtomicIntegerFieldUpdater
AtomicLong
AtomicLongArray
AtomicLongFieldUpdater
AtomicMarkableReference
AtomicReference
AtomicReferenceArray
AtomicReferenceFieldUpdater
AtomicStampedReference

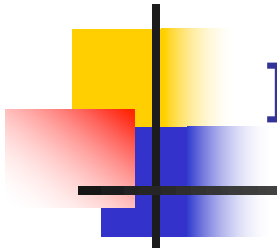
Eksempel på operasjoner i **AtomicIntegerArray:**

int	**get**(int i) Gets the current value at position i.
int	**getAndAdd**(int i, int delta) Atomically adds the given value to the element at index i.
int	**getAndDecrement**(int i) Atomically decrements by one the element at index
void	**set**(int i, int newValue) Sets the element at position i to the given value.

Vi skal bare lære ett fåtall av dette

- Her er de vi skal konsentrere oss om:
  - new Thread – join()
  - synchronized method
  - Semaphore – aquire() og release()
  - CyclicBarrier – await()
  - ExecutorService pool = Executors.newFixedThreadPool(k);
    med Futures - forklares senere
  - AtomicIntegerArray – get(), set(), getAndAdd(),..
  - ReentrantLock (i pakken: **java.util.concurrent.locks**)
  - volatile variable - forklares senere

- Alle de synkroniseringer vi trenger, kan gjøres med disse!
- De fleste andre har sine måter å gjøre det på, men man har neppe tid til å lære seg alle.
- Bedre å bli flink i et lite og tilstrekkelig sett av synkroniseringsprimitiver, enn halvgod i de fleste.

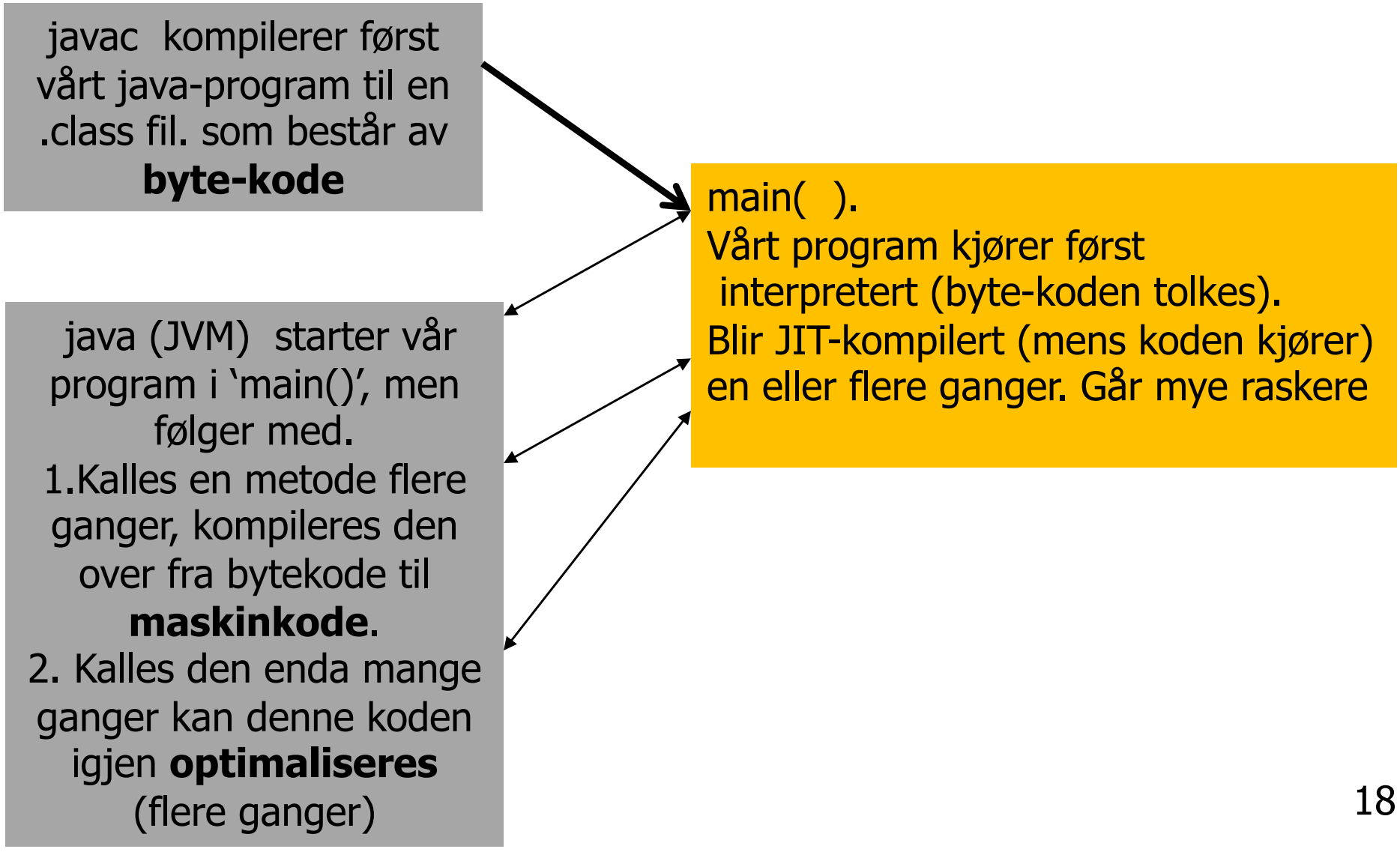


II) Tidtagning

- JIT –kompilering
  - Hvor mye betyr det egentlig
- Operativsystemet (Windows eller Linux)
  - Er de like raske?
- Søppeltømming i Java
  - Skjer under kjøring (med i tidene)

Tidsmålinger og JIT (Just In Time) -kompilering

- Tilbake til kompileringen av et Java-program:



Optimaliserng – ett eksempel

Original kode

```
class A {
  B b;
  public void newMethod() {
    y = b.get();
    ...do stuff...
    z = b.get();
    sum = y + z;
  }
}
class B {
   int value;
   final int get() {
      return value;
   }
}
```

1) Inline get

```
public void
newMethod() {
    y = b.value;
    ...do stuff...
    z = b.value;
    sum = y + z;
}
```

2) Fjern overflødige les

```
public  void
newMethod() {
    y = b.value;
    ...do stuff...
    z = y;
    sum = y + z;
}
```

3) Fjern overflødige variable

```
public  void
newMethod() {
    y = b.value;
    ...do stuff...
    y = y;
    sum = y + y;
}
```

4) Fjern død kode

```
public  void
newMethod() {
    y = b.value;
    ...do stuff...
    sum = y + y;
}
```

Mediantider for finnMax fra ukeoppgavene:

n= 10 000

Vi ser at kjøretidene (para) synker dramatisk fra 1.ste til neste kjøring.
Pga JIT-optimalisering

```
M:\INF3030Para\FinnMax>java FinnMaxMulti 10000 7

Kjøring:0, ant kjerner:8, antTråder:8
Max para = a:9853, paa:    13.24 msek. , nanosek/n: 1324.41
Max sekv = a:9853, paa:     0.13 msek. , nanosek/n:   12.59

Kjøring:1, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.20 msek. , nanosek/n:   20.22
Max sekv = a:9853, paa:     0.11 msek. , nanosek/n:   10.94

Kjøring:2, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.26 msek. , nanosek/n:   25.78
Max sekv = a:9853, paa:     0.11 msek. , nanosek/n:   11.18

Kjøring:3, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.21 msek. , nanosek/n:   21.39
Max sekv = a:9853, paa:     0.24 msek. , nanosek/n:   23.91

Kjøring:4, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.22 msek. , nanosek/n:   21.99
Max sekv = a:9853, paa:     0.20 msek. , nanosek/n:   19.74

Kjøring:5, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.25 msek. , nanosek/n:   25.00
Max sekv = a:9853, paa:     0.23 msek. , nanosek/n:   22.95

Kjøring:6, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.20 msek. , nanosek/n:   19.56
Max sekv = a:9853, paa:     0.21 msek. , nanosek/n:   20.52

Median seq time:   0.205, median para time:   0.250,
 Speedup:    0.82, n = 10 000
```

```
M:\INF3030Para\FinnMax>java  FinnMaxMulti 10000000 5

Kjøring:0, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    21.93 msek. , nanosek/n:    2.19
Max sekv = a:9999216, paa:     7.65 msek. , nanosek/n:    0.76

Kjøring:1, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     3.04 msek. , nanosek/n:    0.30
Max sekv = a:9999216, paa:     5.95 msek. , nanosek/n:    0.59

Kjøring:2, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     3.20 msek. , nanosek/n:    0.32
Max sekv = a:9999216, paa:     7.33 msek. , nanosek/n:    0.73

Kjøring:3, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     2.67 msek. , nanosek/n:    0.27
Max sekv = a:9999216, paa:     5.10 msek. , nanosek/n:    0.51

Kjøring:4, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     2.88 msek. , nanosek/n:    0.29
Max sekv = a:9999216, paa:     5.57 msek. , nanosek/n:    0.56

Median seq time:   5.945, median para time:   3.042,
 Speedup:    1.95, n = 10000000
```

n= 10 mill

```
M:\INF3030Para\FinnMax>java  -Xint FinnMaxMulti 10000000 5

Kjøring:0, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     53.13 msek. , nanosek/n:    5.31
Max sekv = a:9999216, paa:    144.08 msek. , nanosek/n:   14.41

Kjøring:1, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     44.94 msek. , nanosek/n:    4.49
Max sekv = a:9999216, paa:    144.86 msek. , nanosek/n:   14.49

Kjøring:2, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     33.83 msek. , nanosek/n:    3.38
Max sekv = a:9999216, paa:    137.45 msek. , nanosek/n:   13.75

Kjøring:3, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     53.63 msek. , nanosek/n:    5.36
Max sekv = a:9999216, paa:    136.90 msek. , nanosek/n:   13.69

Kjøring:4, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     50.09 msek. , nanosek/n:    5.01
Max sekv = a:9999216, paa:    137.71 msek. , nanosek/n:   13.77

Median seq time: 137.714, median para time:  50.088,
 Speedup:     2.75, n = 10000000
```

JIT-kompilering avslått :
> java –Xint
…..
n= 10 mill

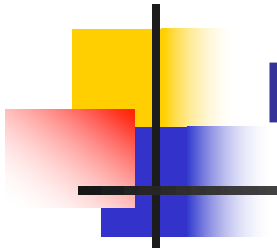
```
M:\INF3030Para\FinnMax>java FinnM 100000000 5
Kjoering:0, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 41.913504 ms.
Max verdi sekvensiell i a:99989305, paa: 238.799921 ms.

Kjoering:1, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 26.78024 ms.
Max verdi sekvensiell i a:99989305, paa: 235.431219 ms.

Kjoering:2, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 27.791271 ms.
Max verdi sekvensiell i a:99989305, paa: 248.066478 ms.

Kjoering:3, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 26.86283 ms.
Max verdi sekvensiell i a:99989305, paa: 236.013201 ms.

Kjoering:4, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 27.755575 ms.
Max verdi sekvensiell i a:99989305, paa: 223.535073 ms.


Median sequential time:236.013201, median parallel time:27.755575,
 n=  100000000, Speedup:  8.59
```

n= 100 mill

JIT-kompilering +optimalisering

Søppel-tømming

Hva betyr dette for tidsmålingene



- Første gangen vi gjører er tiden vi måler en sum av:
  - Først litt interpretering av bytekode
  - Så oversetting(kompilering) av hyppig brukte metoder til maskinkode
  - kjøring av resten av programmet dels i maskinkode.
- Andre gang vi kjører, kan følgende skje:
  - JVM finner at noen av maskinkompilerte metodene våre må optimaliseres ytterligere
  - Kjøretiden synker ytterligere
- Tredje gang er som oftest optimaliseringa ferdig, men ytterligere optimalisering kan bli gjort
- Tidtakningen vår må endres !
- Vi kjører det sekvensielle og parallelle programmet f.eks 9 ganger i en løkke , noterer alle kjøretider i to arrayer som så sorteres og vi velger medianverdien = a[a.length/2]
- Du får aldri samme svaret to ganger – mye variasjon !!

FinnMax, 3 ulike kjøringer (samme parametre , varierer antall tråder: 8, 16, 4)

Uke2>java FinnM 1000000 9
Kjøring:0**, ant kjerner:8, antTråder:8**
Max verdi parallell i a:999216, paa: 23.860968 ms.
Max verdi sekvensiell i a:999216, paa: 3.468803 ms.

Kjøring:1, ant kjerner:8, antTråder:8
Max verdi parallell i a:999216, paa: 0.311465 ms.
Max verdi sekvensiell i a:999216, paa: 0.549437 ms.
.....
Kjøring:8, ant kjerner:8, antTråder:8
Max verdi parallell i a:999216, paa: 0.422752 ms.
Max verdi sekvensiell i a:999216, paa: 0.532639 ms.

Median sequential time:0.52004,
median parallel time:0.429051,
Speedup: **1.26,** n = 1000000

Uke2>java FinnM 1000000 9
Kjøring:0, **ant kjerner:8, antTråder:16**
Max verdi parallell i a:999216, paa: 18.808946 ms.
Max verdi sekvensiell i a:999216, paa: 3.558043 ms.

Kjøring:1, ant kjerner:8, antTråder:16
Max verdi parallell i a:999216, paa: 1.847439 ms.
Max verdi sekvensiell i a:999216, paa: 0.453898 ms.
.....
Kjøring:8, ant kjerner:8, antTråder:16
Max verdi parallell i a:999216, paa: 0.502542 ms.
Max verdi sekvensiell i a:999216, paa: 0.471396 ms.

Median sequential time:0.509891,
median parallel time:0.646726,
Speedup: **0.90,** n = 1000000

Uke2>java FinnM 1000000 9
Kjøring:0, **ant kjerner:8, antTråder:4**
Max verdi parallell i a:999216, paa: 16.154151 ms.
Max verdi sekvensiell i a:999216, paa: 3.75507 ms.

Kjøring:1, ant kjerner:8, antTråder:4
Max verdi parallell i a:999216, paa: 1.280854 ms.
Max verdi sekvensiell i a:999216, paa: 0.520741 ms.

Kjøring:2, ant kjerner:8, antTråder:4
Max verdi parallell i a:999216, paa: 0.557136 ms.
Max verdi sekvensiell i a:999216, paa: 0.509191 ms.
.....
Kjøring:8, ant kjerner:8, antTråder:4
Max verdi parallell i a:999216, paa: 0.628527 ms.
Max verdi sekvensiell i a:999216, paa: 0.52354 ms.

Median sequential time:0.520741, median parallel time:0.628527,
Speedup: **0.88**, n = 1000000

«Aldri» samme resultatet to ganger

Uke2>java FinnM 1000000 9
ant kjerner:8, antTråder:8, n = 1mill

Med antall kjøringer for median = 9
1) Speedup: **0.68**, n = 1000000
2) Speedup: 0.96, n = 1000000
3) Speedup: 0.84, n = 1000000
4) Speedup: 0.71, n = 1000000
5) Speedup: 1.06, n = 1000000
6) Speedup: 1.26**,** n = 1000000

Med antall kjøringer for median = 21
7) Speedup: 1.00, n = 1000000
8) Speedup: 0.84, n = 1000000
9) Speedup: 0.88, n = 1000000
10) Speedup: **1.75**, n = 1000000
11) Speedup: 0.87, n = 1000000
12) Speedup: 1.11, n = 1000000
13) Speedup: 1.03, n = 1000000

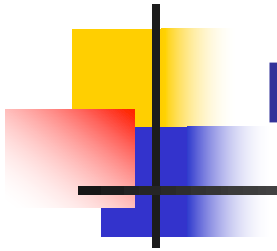
Konklusjon på JIT-kompilering

- JIT-kompilering kan skrues av med >java –Xint MittProg ..
  - Brukes bare for debugging
- JIT kompilering kan gi 10 til 30 ganger så rask eksekvering for liten n (en god del mer for stor n)
- Første, andre (og tredje) kjøring er tidsmessig sterkt misvisende
- Vi må:
  - Kjøre programmet i en løkke f.eks 9 (eller 7 eller 11) ganger
  - Legge tidene i hver sin array (sekvensielt og parallell tid)
  - Sortere arrayene
  - Ta ut medianen (element a.length/2), som blir vår tidsmåling

Dette måler tidene for 9 tråder kjørt **etter** hverandre

```
import java.util.concurrent.*;
import java.util.*;
class Problem2 { int [] fellesData  ; // dette er felles, delte data for alle trådene
       double [] tidene ;
       int ant, svar;
       public static void main(String [] args) {
              ( new Problem()).utfoer(args);
        }
        void utfoer (String [] args) {
                 ant = new Integer(args[0]);
                 fellesData = new int [ant];
                 tidene = new double[9];
                 for (int m = 0; m <9; m++) {
                                long tid = System.nanoTime();
                                Thread t = new Thread(new Arbeider());
                                 t.start();
                                 try{t.join();}catch (Exception e) {return;}
                                tidene[m] = (System.nanoTime() -tid)/1000000.0;
                                System.out.println("Tid for "+m + ", tråd:"+tidene[m]+« ms");
                 }
                 Arrays.sort(tidene);
                 System.out.println("Median med svar:"+svar+", for trådene:"+tidene[(tidene.length)/2]+" ms");
         } // end utfoer

         class Arbeider implements Runnable {
            int i,lokalData;    // dette er lokale data for hver tråd
                    public void run() {     int sum =0;
                                            for (int i = 0; i < ant; i++) sum +=fellesData[i];
                                            svar =sum;
                    }
          } // end indre klasse Arbeider
} // end class Problem
```

```
M:\INF3030Para\Powerpoint\Uke2>java Problem2  1000000

Tid for 0, tråd:22.26 ms
Tid for 1, tråd:  1.12ms
Tid for 2, tråd:  3.19ms
Tid for 3, tråd:  0.58ms
Tid for 4, tråd:  0.65ms
Tid for 5, tråd:  0.49ms
Tid for 6, tråd:  0.48ms
Tid for 7, tråd:  0.53ms
Tid for 8, tråd:  0.85ms

Median med svar:0, for trådene:0.65 ms
```



Hva med operativsystemet:

- Linux og Windows har om lag like rask implementasjon av Java og trådprogrammering,
- Dag Langmyhr testet to helt like maskiner med hhv. Linux og Windows, og resultatene tidsmessig (medianer) var nesten helt like, men
  - Ulike maskiner som Ifis store servere (diamant, safir,..) har en annen Linux og en noe langsommere ytelse for korte, trådbaserte programmer.

Hva med søppeltømming – garbage collection:

- Søppeltømming (=opprydding i lageret og fjerning av objekter vi ikke lenger kan bruke) kan slå til når som helst under kjøring:

```
Kjøring:2, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.35 msek. , nanosek/n:   35.07
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.36

Kjøring:3, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.57 msek. , nanosek/n:   56.87
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    0.66

Kjøring:4, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.43 msek. , nanosek/n:   43.47
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.33

Kjøring:5, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.49 msek. , nanosek/n:   49.20
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.36
```

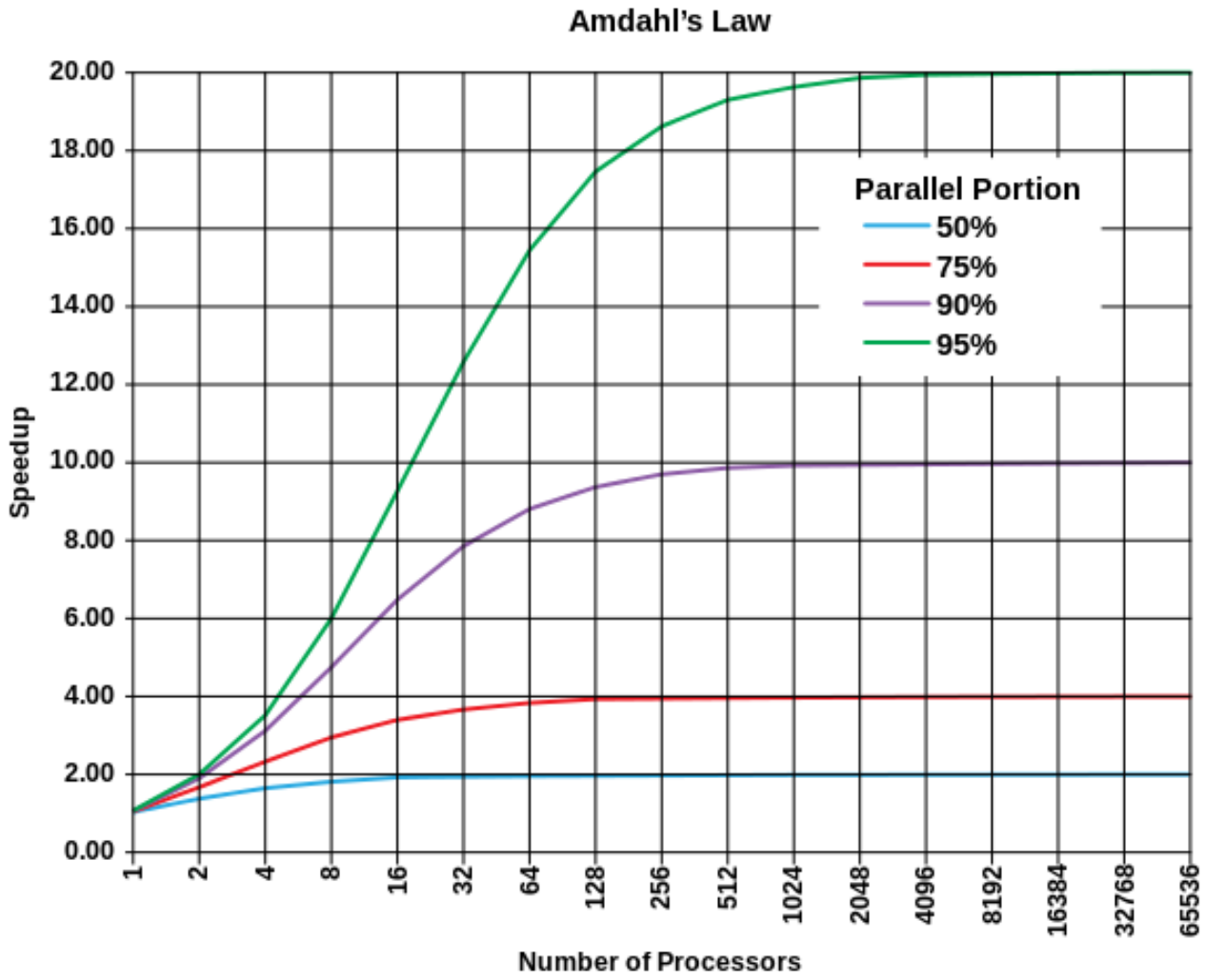
Amdahl lov for parallelle beregninger

- Amdahl lov: Har du **seq** andel sekvensiell kode og da **p** andel parallelliserbar kode i et parallelt program, **seq+p=1**, er den største speedup S du kan få med k kjerner:

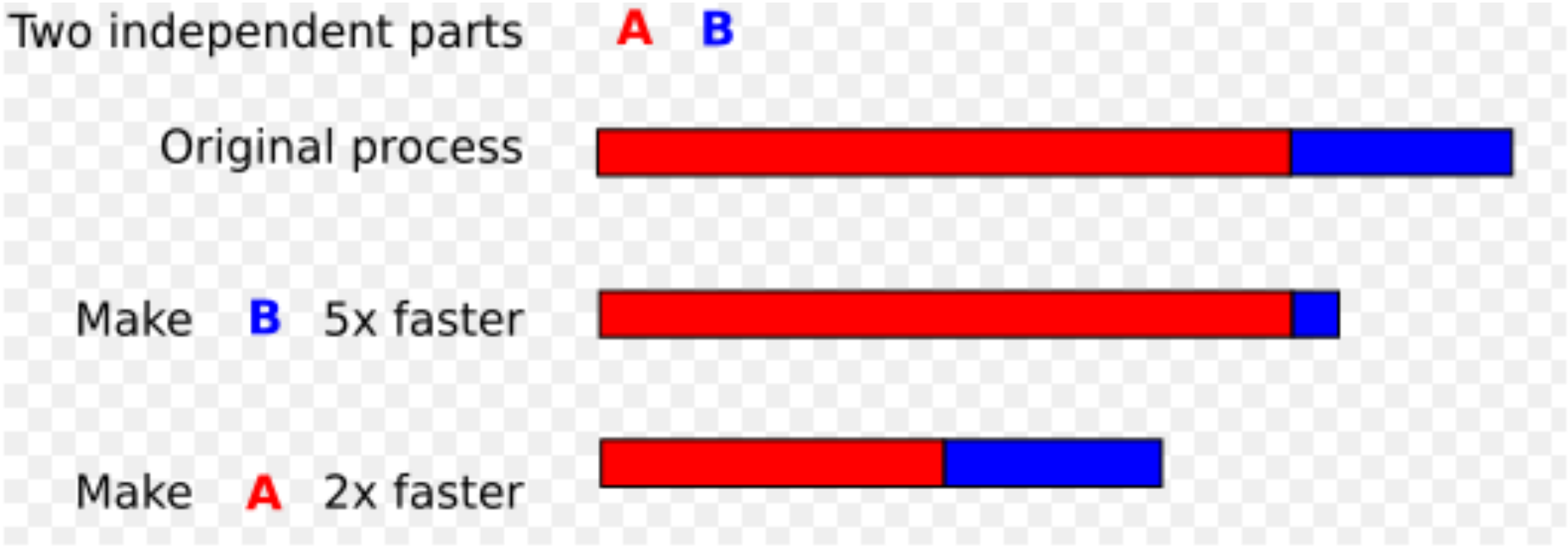
$$S = \frac{tid(sekvensiell)}{tid\ (parallell)} = \frac{1}{seq+p/k} = \frac{1}{1-p+p/k}$$

- Når k $\rightarrow\ \infty$, vil S $\rightarrow \frac{1}{1-p}$.
- Er p=0.9, så er S ≤ 10 uansett hvor mange kjerner du har, og har du 'bare' 50, er S = $\frac{1}{1-0.9+0,9/50}$ = 8,5.
- Amdahls lov er pessimistisk- antar fast størrelse på problemet
- «Hvis du først har brukt 10% av tida på en sekvensiell del, så kan resten av programmet ikke gå fortere enn 0.00 sekunder uansett hvor mange prosessorer du bruker på det. Dvs. at speedup ≤ 10»

Amdahl for ulike verdier av p



Amdahl – viktig å parallellisere største del



Gustafsons lov for parallelle beregninger

- La S være speedup, P antall kjerner og α være andel sekvensiell kode (tidsmessig), så er:

$$S(P) = P - \alpha\,(P-1)$$

Parallell løsning er: $a + b$ (a = sekvensiell tid, b = parallell tid)

Sekvensiell løsning er da: $a + P * b$

Speedup er da:

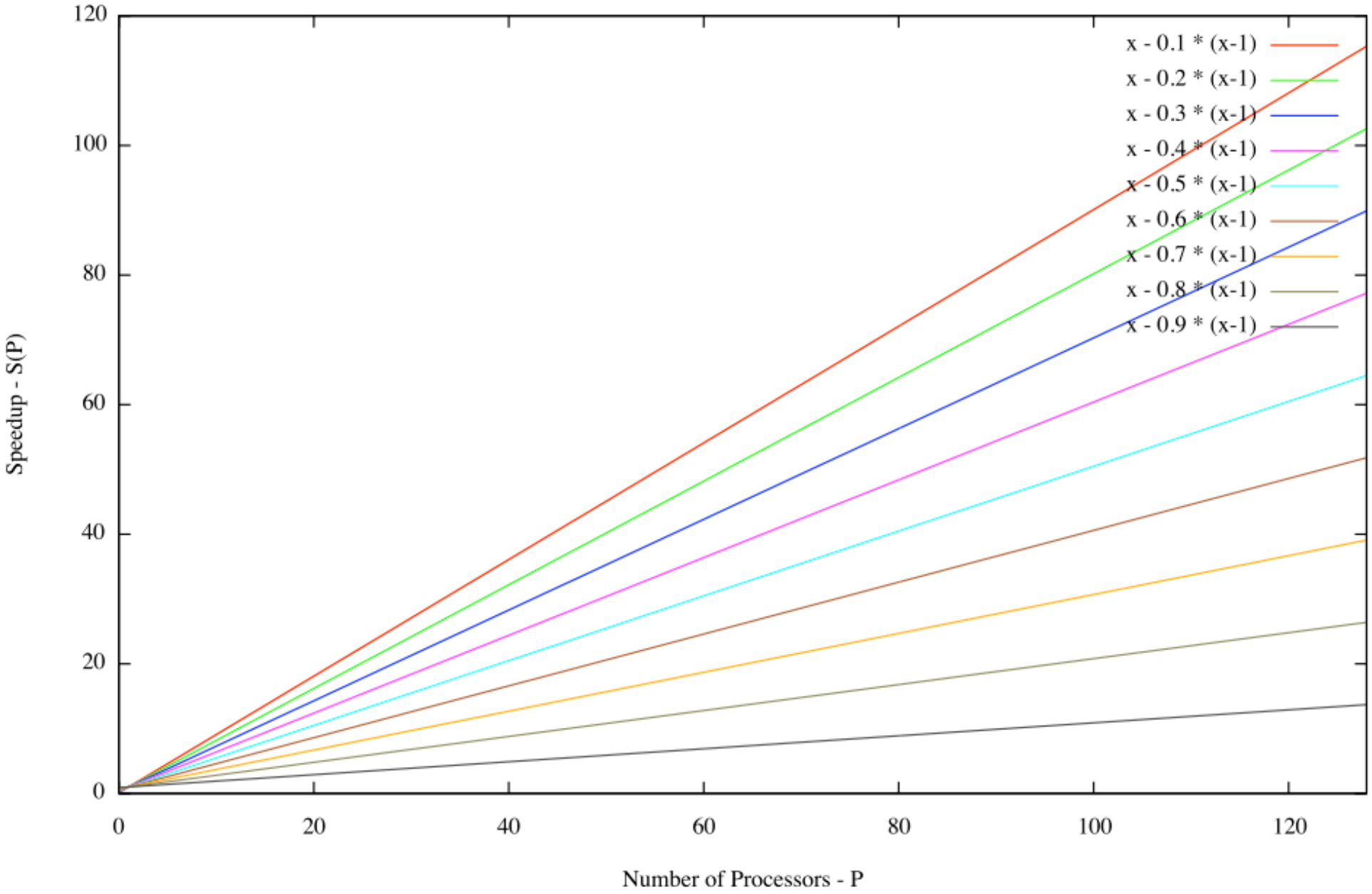
$\frac{a+P*b}{a+b}$, og har at $\alpha = \frac{a}{a+b}$ og da er:

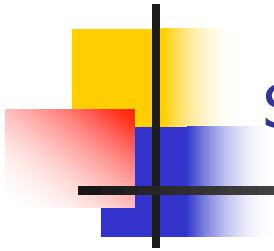
$$\mathbf{S}(\boldsymbol{P}) = \frac{a + P * b}{a + b} = \frac{a}{a + b} + P * \frac{b}{a + b} = \alpha + \mathrm{P} * \frac{b}{a + b}$$

$$= \alpha + \mathrm{P} * \frac{a+b-a}{a+b} = \alpha + \mathrm{P} * (1 - \alpha) = \mathbf{P} - \boldsymbol{\alpha}(\mathbf{P} - \mathbf{1})$$

- «Hvis du tidligere brukte 1 time på å løse et problem sekvensielt, vil du nå også bruke 1 time på å løse et større, mer nøyaktig problem parallelt, da med større speedup– for eksempel i meteorologi»

Gustafson's Law: $S(x) = x - \alpha(x - 1),$



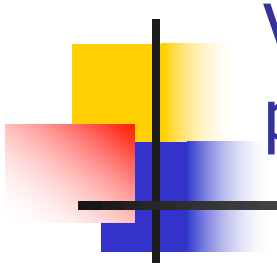


Sammenligning av Amdahl og Gustafson + egne betraktninger

- Amdahl antar at oppgaven er fast av en gitt lengde(n)
- Gustafson antar at du med parallelle maskiner løser større problemer (større n) og da blir den sekvensielle delen mindre.
- Min betraktning:
  1. En algoritme består av noen sekvensielle deler og noen parallelliserbare deler.
  2. Hvis de sekvensielle delene har lavere orden – f.eks O(log n), men de parallelle har en større orden – eks O(n) så vil de parallelle delene bli en stadig større del av kjøretida hvis n øker (Gustafson)
  3. Hvis de parallelle og sekvensielle delene har samme orden, vil et større problem ha samme sekvensielle andel som et mindre problem (Amdahl).
  4. I tillegg kommer alltid et fast overhead på å starte k tråder (1-4 ms.)

  Algoritmer vi skal jobbe med er mer av type 2 (Gustafson) enn type 3(Amdahl) men vi har alltid overhead, så små problemer løses best sekvensielt.

Konklusjon: For store problemer bør vi ha håp om å skalere nær lineært med antall kjerner hvis ikke vi får kø og forsinkelser når alle kjernene skal lese/skrive i lageret.



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

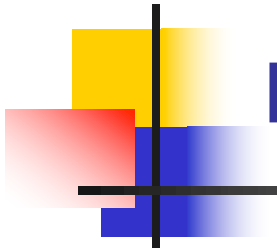
- Et problemet kunne være at når en av tråden lester opp et element i a[i] (int = 4 byte), så er cache-linja 64 byte, så den får med seg flere elementer før og etter a[i].
- Disse 'andre' elementene er det andre tråder som skriver på.
- Vi skriver et testprogram (ParaArray) hvor 10 tråder med indeks : 0,1,2,..,9 som øker hvert sitt element i en array tall[index] 100 000 ganger.

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%
 sum:100000000, tap:0 =   0.0%
```

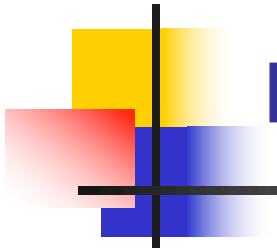


Konklusjon:

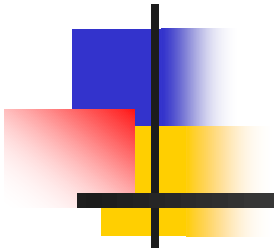
- Skriving samtidig i **ulike** elementer i en array går bra.
  - Dette skal vi bruke mye i kommende algoritmer.
  - (kan riktignok medføre litt ekstra eksekveringstid – det ser vi på senere)

- Men, skriving **samtidig i samme element** går galt!

Hva har vi sett på i Uke2

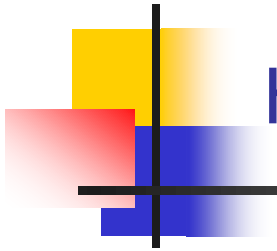


- Én stygg feil vi kan gjøre: Samtidig oppdatering (skriving) på delte data, på samme variabel (eks: i++)
- Samtidig skriving på en variabel ***må synkroniseres***:
  - Alle objekter kan nyttes som en synkroniseringsvariabel, og da kan vi bruke enten en synchronized metode (treigt)for å gjøre det,
  - eller objekter av spesielle klasser som:
    - CyclicBarrier
    - Semaphore (undervises Uke2)
    - Lock
  - De har metoder som await() eller lock(), som gjør at tråder evt. må vente.
- I) Tre måter å avslutte tråder vi har startet.
  - join(), Semaphor og CyclicBarrier.
- II) Mange 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



INF3030, Uke 3, våren 2019 – Regler for parallelle programmer, mer om cache og Matrise-multiplikasjon

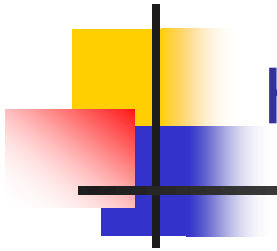
Arne Maus / Eric Jul
PSE,
Inst. for informatikk



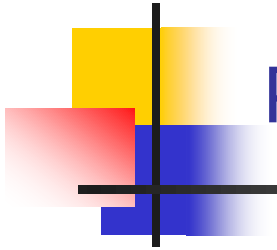
Hva har vi sett på i Uke2

- Én stygg feil vi kan gjøre: Samtidig oppdatering (skriving) på delte data, på samme variabel (eks: i++)
- Synkronisering er vanskelig!
- Samtidig skriving på en variabel ***må synkroniseres***:
  - Alle objekter kan nyttes som en synkroniseringsvariabel, og da kan vi bruke enten en synchronized metode (treigt) for å gjøre det,
  - eller objekter av spesielle klasser som:
    - CyclicBarrier
    - Semaphore
    - AtomicInteger
    - En av MANGE andre mulige

Hva har vi sett på i Uke2 (fortsatt)



- I) Tre måter å avslutte tråder vi har startet.
  - join(), Semaphore 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
  - Bruk av 'median-tider' av flere kjøinger (3,5,...,11)
- IV) 'Lover' om Kjøretid
  - Amdahl lov
  - Gustafsons lov (utsatt til uke3)
- Noen algoritmer følger Amdahl, andre (de fleste) følger Gustafson



Plan for uke 3

1. Gustafsons lov
2. Modell(er) for hvordan vi programmerer
3. Viktige regler om lesing og skriving på felles data.
4. Synlighetsproblemet
    1. Hvilke verdier ser ulike tråder som leser variable som en annen tråd skriver på?
5. Hvor fort kan JIT-kompilert kode gå?
6. Prefetch-mekanismen i elektronikken
7. Java har 'as-if sequential' semantikk
8. Effekten på eksekveringstid av cache
    1. Lese og skrive a[b[i]]=i;
9. Kommentarer til obligene og nå: Oblig 2 Matrisemultiplisering

Utledningen av Gustafson bruker flg. to regler

1) $$\frac{a+b}{a+b} = \frac{a}{a+b} + \frac{b}{a+b}$$

2) $$\frac{b}{a+b} = \frac{b+a-a}{a+b} = \frac{a+b}{a+b} - \frac{a}{a+b} = 1 - \frac{a}{a+b}$$

Gustafsons lov for parallelle beregninger

- La S være speedup, P antall kjerner og α være andel sekvensiell kode (tidsmessig), så er:

$$S(P) = P - \alpha\,(P-1)$$

Parallell løsning er: $a + b$ (a = sekvensiell tid, b = parallell tid)

Sekvensiell løsning er da: $a + P * b$

Speedup er da:

$\frac{a+P*b}{a+b}$, og har at $\alpha = \frac{a}{a+b}$ og da er:

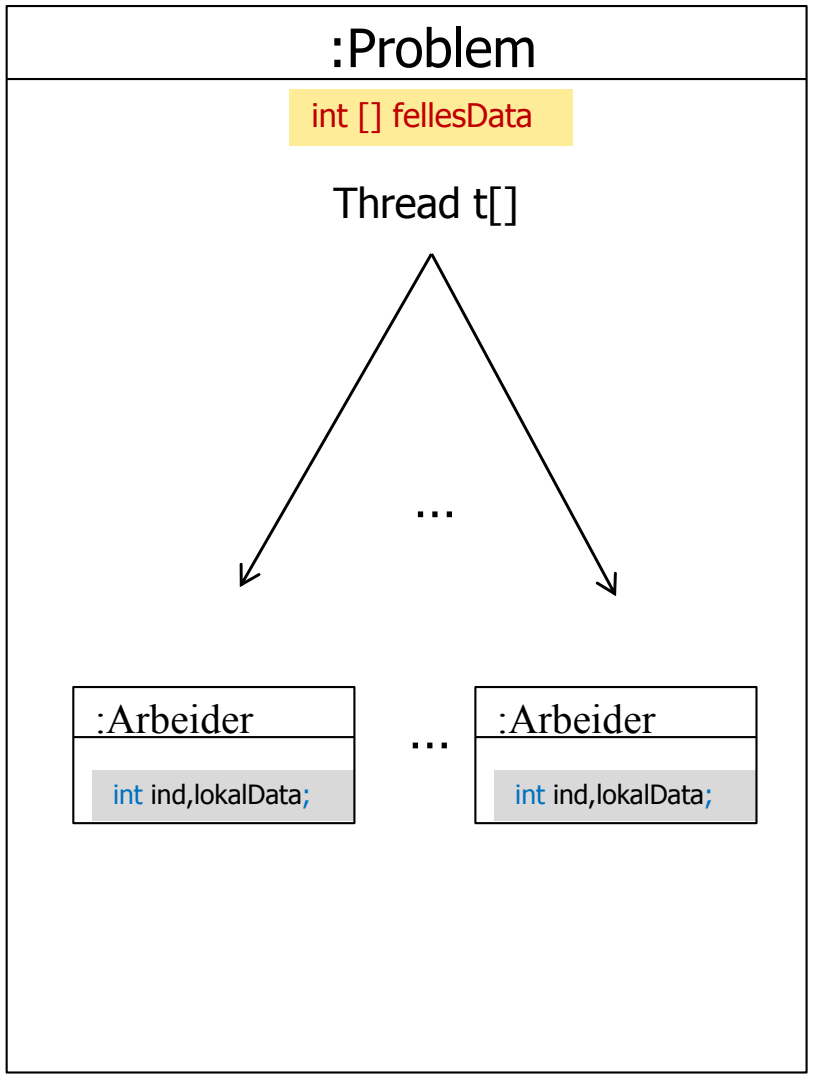
$$\mathbf{S}(\boldsymbol{P}) = \frac{a + P * b}{a + b} = \frac{a}{a + b} + P * \frac{b}{a + b} = \alpha + \mathrm{P} * \frac{b}{a + b}$$

$$= \alpha + \mathrm{P} * \frac{a+b-a}{a+b} = \alpha + \mathrm{P} * (1 - \alpha) = \mathbf{P} - \boldsymbol{\alpha}(\mathbf{P} - \mathbf{1})$$

- «Hvis du tidligere brukte 1 time på å løse et problem sekvensielt, vil du nå også bruke 1 time på å løse et større, mer nøyaktig problem parallelt, da med større speedup– for eksempel i meteorologi»

1) 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) {
            … // utfør sekvensiell kode med tidtaking
          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();
      }
      //.. metoder for sekvensielt  & parallell

      class Arbeider implements  Runnable {
        int ind, lokaleData; // lokale data
        //.. metoder for parallelt problem
         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)

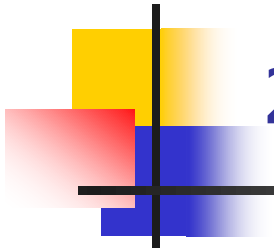
```
class ProblemX{
    <felles data>

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

    class ArbeiderTråd extends Thread{
       <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 når vi bruker parallell kode>;
        } // 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 INF3030 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 og 150 for matrise-multiplikasjon.
  - 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



2) 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 er å la en tråd i et 'annet' parallelt problem kalle på en parallell løsning som løsX. Blir fort for mange tråder og treigt. (dvs. ikke parallelliser inne i en allerede parallellisert kode)
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.

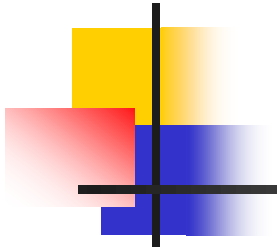
(fortsettes neste foil)

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. Hvis behovet for å ha en enkel seksvensiell kode oppstår midt under beregningene, kan alle trådene regne ut samme svar uten synkronisering seg imellom (skjer f.eks i parallell Quicksort)
8. Arbeider-trådene initierer bare lokale variable i sin konstruktør.
   8. Husk at objektet ikke er ferdig når konstruktøren kjører. Mye galt kan skje (se boka JCiP kap 3.2) hvis andre tråder får en peker til objektet før det er ferdig.
   9. Ingen kall til andre metoder i konstruktøren.
9. All handling i arbeider-trådene skjer i run() og i metoder kalt fra run().



Tre avgjørende prinsipper for 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

3) 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 & en øker b (100 000 ganger) og skriver ned verdiene av a og b i hver sin *lokale* arrayer: mA[] og mB[] (antså to sett av disse).

```
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;
 }
```

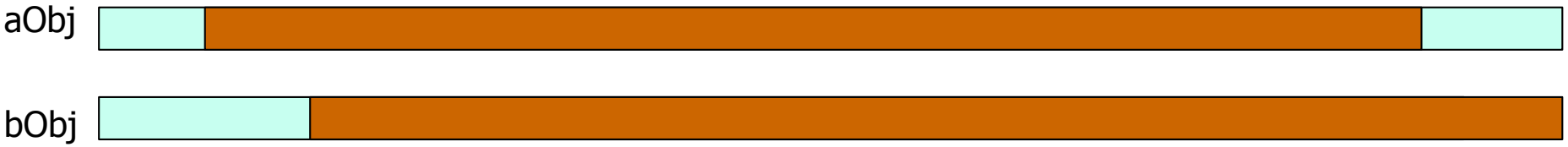
Ytre klasse SamLes med to indre klasser SkrivA og SkrivB

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


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



Leter utover i de to arrayene: mA[] i de to objektene aObj og bObj og starter utskrift ut når a-verdiene i er like og > 0, og skriver da ut de 10 neste verdiene av a og b i aObj og bObj

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

= like verdier i a og b

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 1458 til 1459, 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, 1462) sees aldri av 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 dette er synlig i annen tråd.

4) Skriving på nærliggende elementer i en array, kode.

```
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%
```

5) 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 går: 772 ganger fortere

JIT-kompilering ulike eksempler

n, ant. ganger	1	2	10	100	10000	100000	x bedre
for-løkke	0,3	0,15	0,03	0,018	0,009	0,007	42
metode-kall	2697	0,45	0,06	0,054	0,026	0,026	103 730
new int[100]	1,2	0,6	0,24	0,195	0,151	0,136	33
array copy med for-loop, n=100	1,8	1,5	2,64	2,500	1,177	0,188	9
System.arraycopy, n=100	5,7	0,3	0,15	0,126	0,072	0,064	89
new Thread med start & join()	3015	336	66,6	61,68	61,87	61,86	48
new C(int) og metodekall	2697	0,45	0,15	0,21	0,035	0,035	77 057
array read	0,3	0,3	0,06	0,036	0,012	0,012	25
Innstikk-sortering (n=100)	46,6	42,8	42,42	21,27	19,60	1,45	32

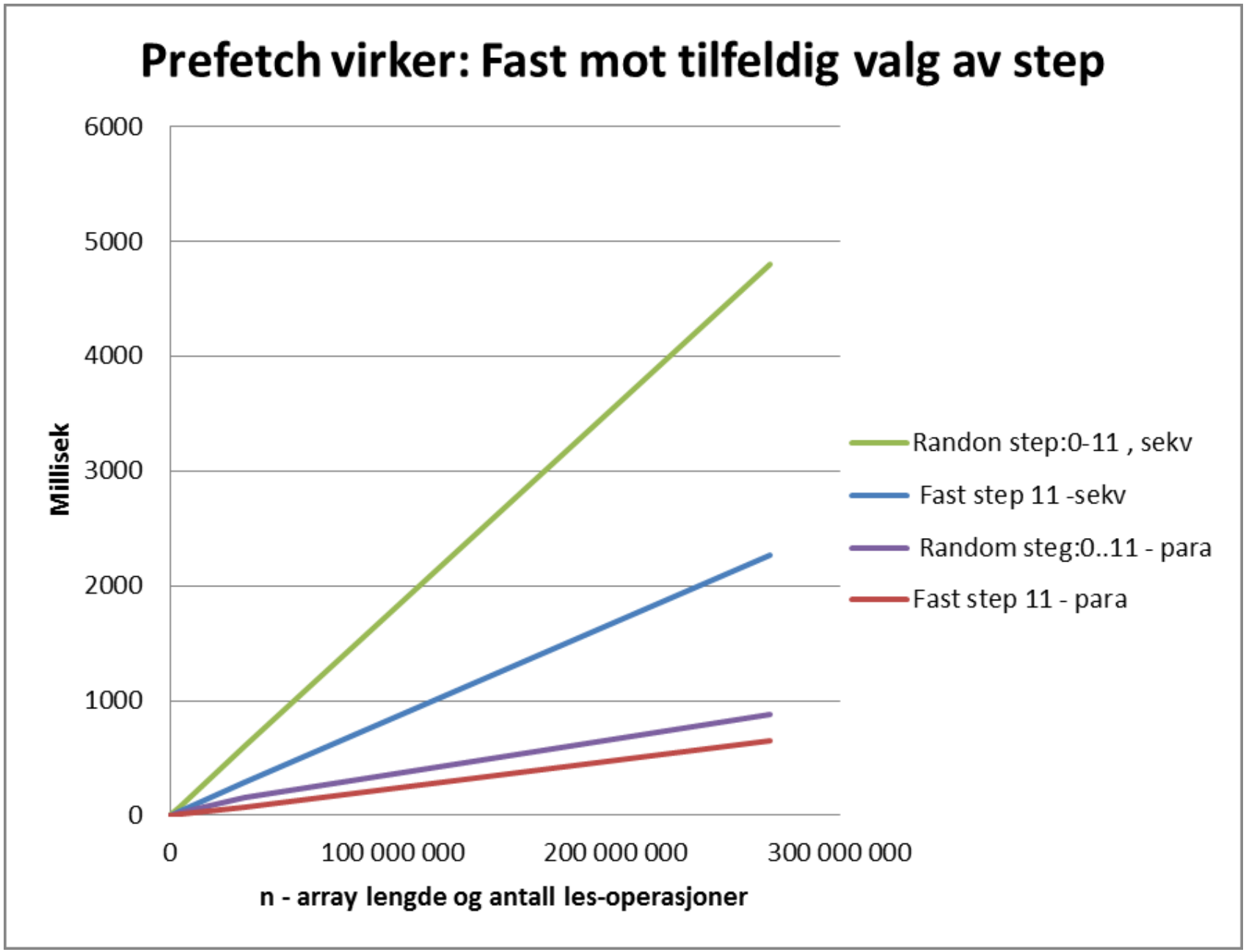
Tabell Gjennomsnitlige kjøretider i μs med Java 1.8.0 for ulike Java-konstruksjoner og et enkelt sorteringsprogram som funksjon av antall utførte ganger, Intel i7-7600 @3,4 Ghz, 8(4) kjerner.

5) Prefetch-mekanismen på brikken

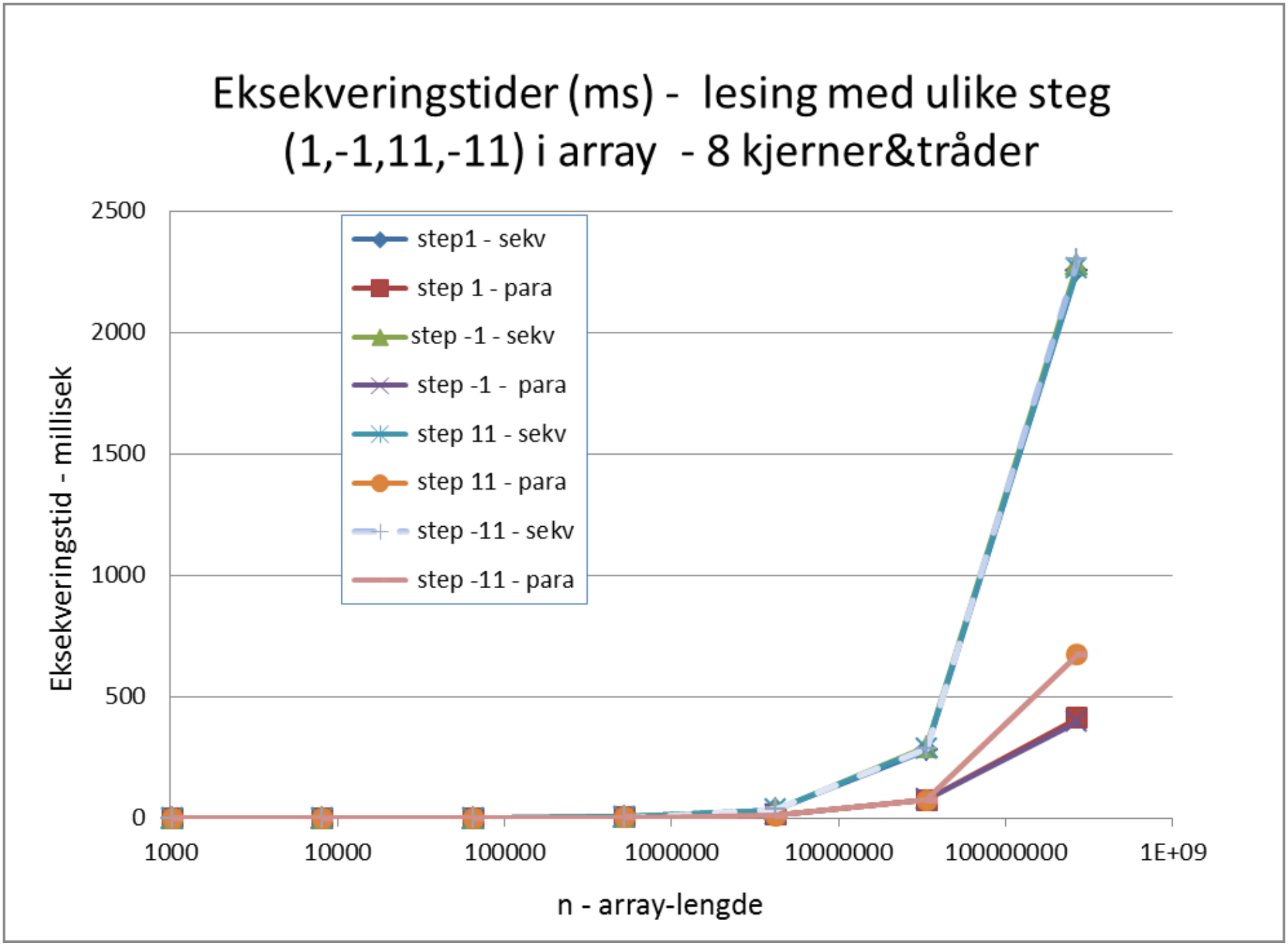
- Består i at hvis vi aksesserer (leser eller skriver) element i k i en array a[] og så element k+m, så prøver elektronikken å hente element a[k+2m], a[k+3m],..., ***før*** vi har bedt om det.
- Tillegget m kan være både positivt og negativt .
- Hvis elementene a[k+2m] ligger i samme cachelinje, går dette spesielt fort.
- Skrev testprogram for dette og testet m = 1, -1,11 og -11

```
for (int i=0;i < a.length; i++){
    // index = Math.abs((index+r.nextInt(step+1))%a.length);
    index = Math.abs((i+step)%a.length );
    sum += a[index];
}
```

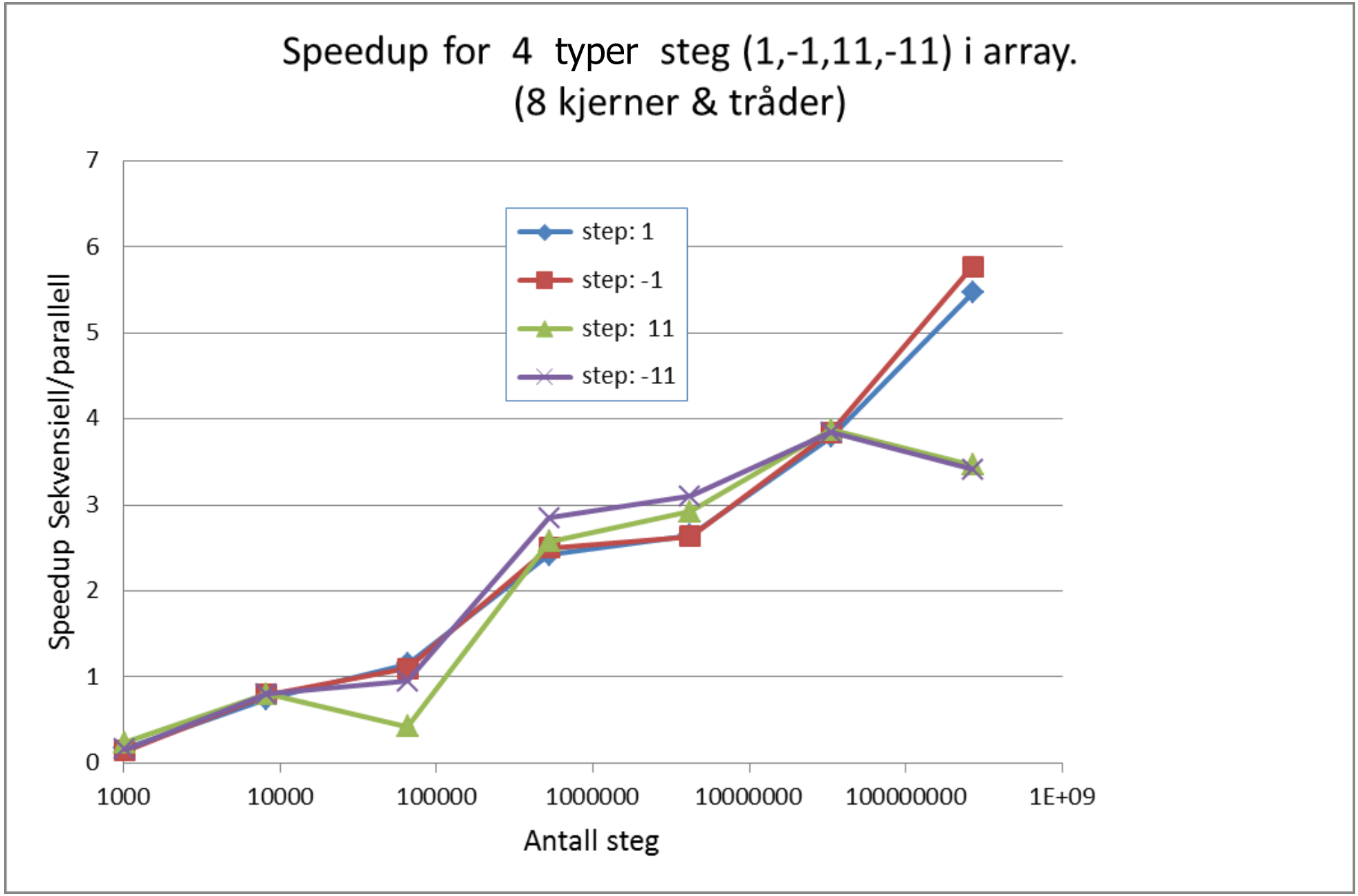
- Hva fant vi og hva kan vi slutte av det.
- Først grafer



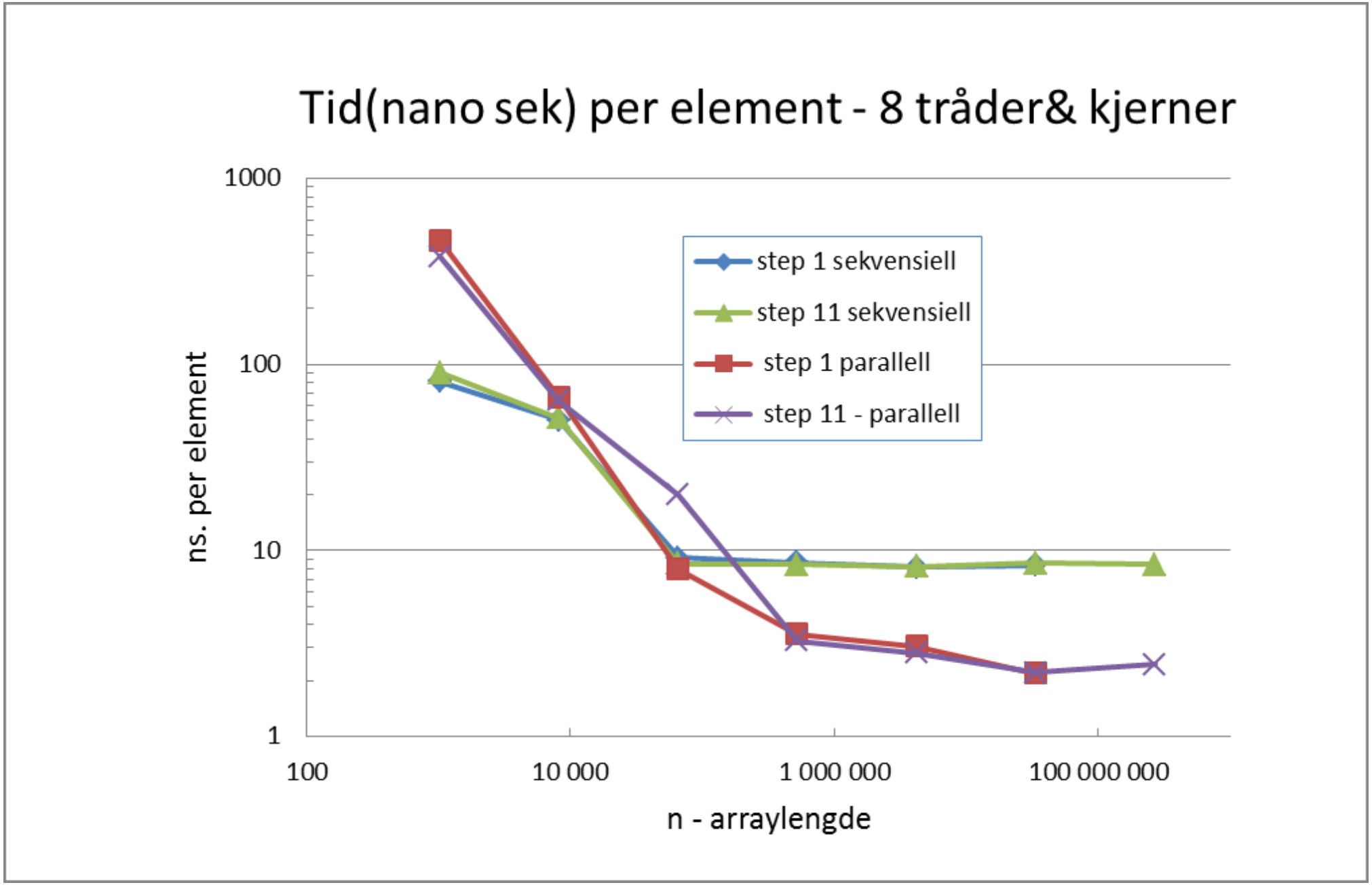
Konklusjon1: Fast steglengde er ca. dobbelt så raskt som tilfeldig (pga prefetch)



Konklusjon2: negative og positive tillegg er like raske.

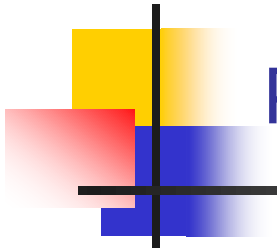


Konklusjon3: Spesielt steg 1,-1 har bra speedup pga. samme cachelinje



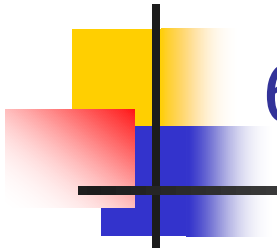
Konklusjon 4: Prosesseringstiden per element synker med økende n (JIT?)

Konklusjon 5: Per element tar det litt over 2*2,8 = ca. 6 instruksjoner å summere et element til en sum i parallell.



Prefetch-mekanismen hjelper en del

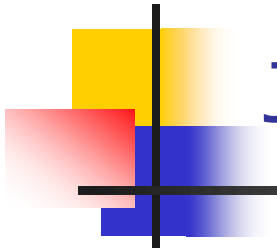
- Ikke så viktig som cache-systemet
- Ikke så viktig som JIT-kompilering
- men hjelper til og går på ingen måte i veien for de to viktigste mekanismene – ca. 2x raskere
- Programmet som laget data til disse grafene er laget av programmet Prefetch.java som er lagt ut på hjemmesida
- Grafene er laget i Excel (velg graftype:scatter diagram):
  - sett inn et slikt i regnearket og trykk så Select Data



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

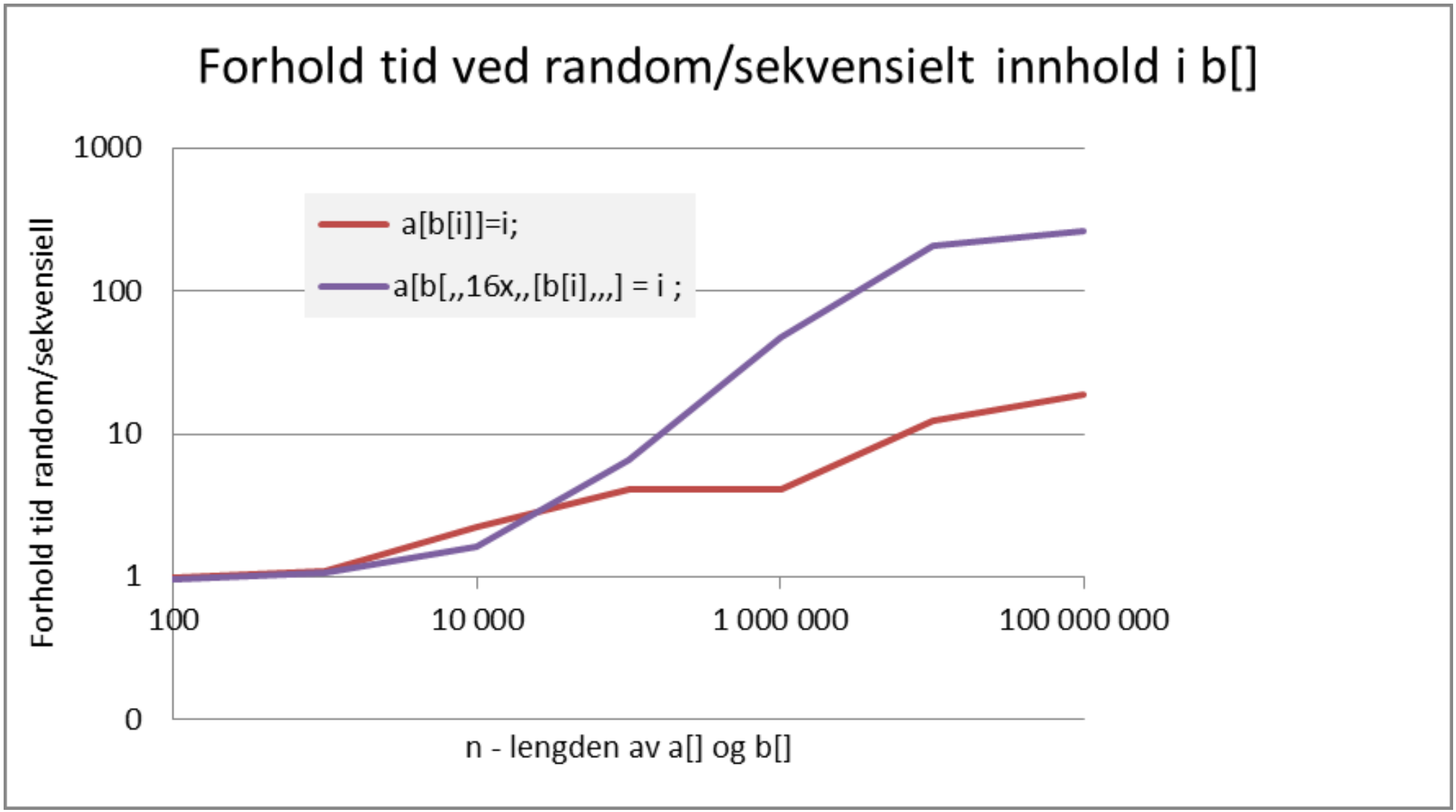


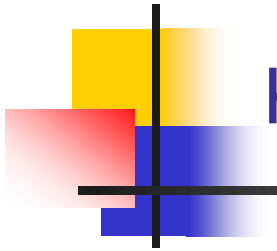
- Det vi utfører så i parallell i flere eksemplarer, er ikke akkurat den koden vi ser.
- **MEN**: Java lover deg at det som utføres gir akkurat samme resultat (på utskrift, fil, skjerm,..) som om programmet du har skrevet ble utført slavisk sekvensielt, en setning ad gangen slik det står i koden.
- Dette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

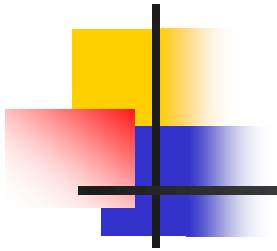
Hvor mange ganger tregere går random innhold i b[] enn b[] = 0,1,2,3,.. ?





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.



Matrix Multiplikation

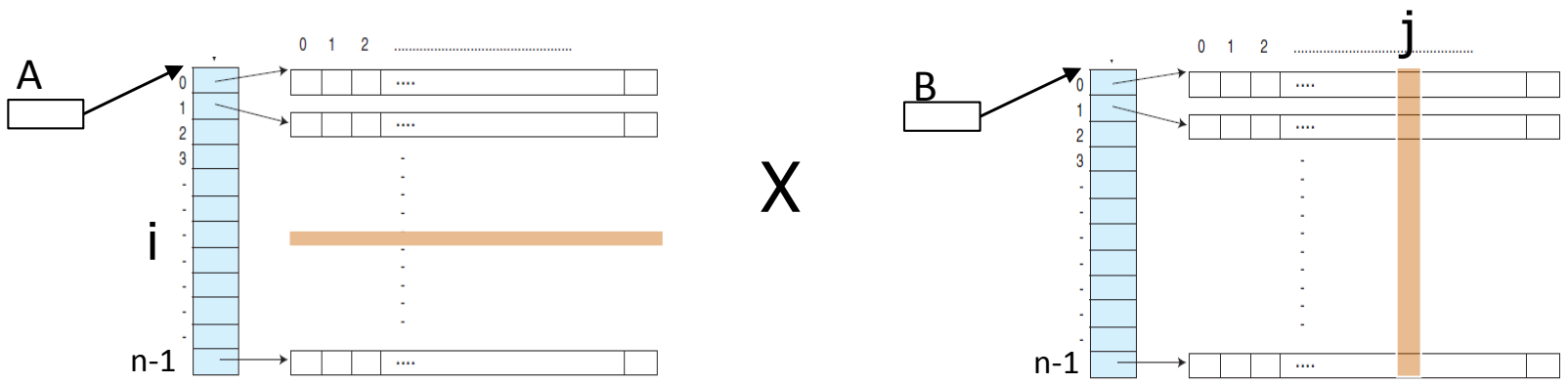
- Hvad er Matrix Multiplikation?
- Eksempel
- Effekten af cache

1) Ukeoppgaven denne uka, matrisemultiplisering

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

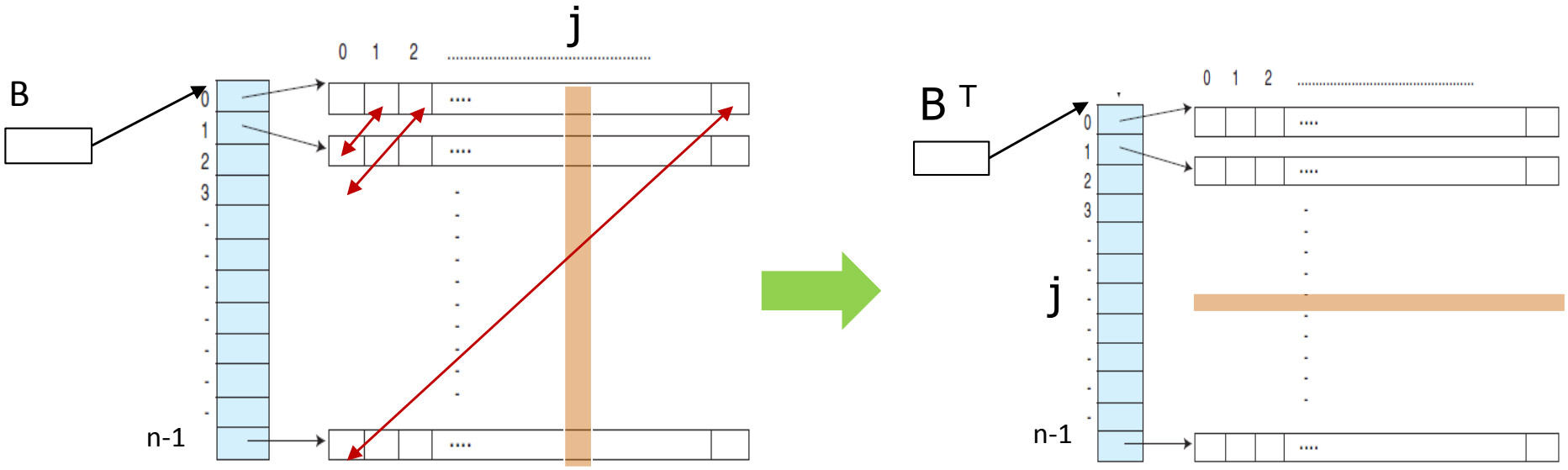
$$c[i][j] = \sum_{k=0}^{n-1} a[i][k] * b[k][j]$$

$C_{i,j} =$



Idé – transponer B (=bytte om rader og kolonner)

- Bytt om elementene i B ($B_{i,j}$ byttes om med $B_{j,i}$)
- Da blir kolonnene lagret radvis.

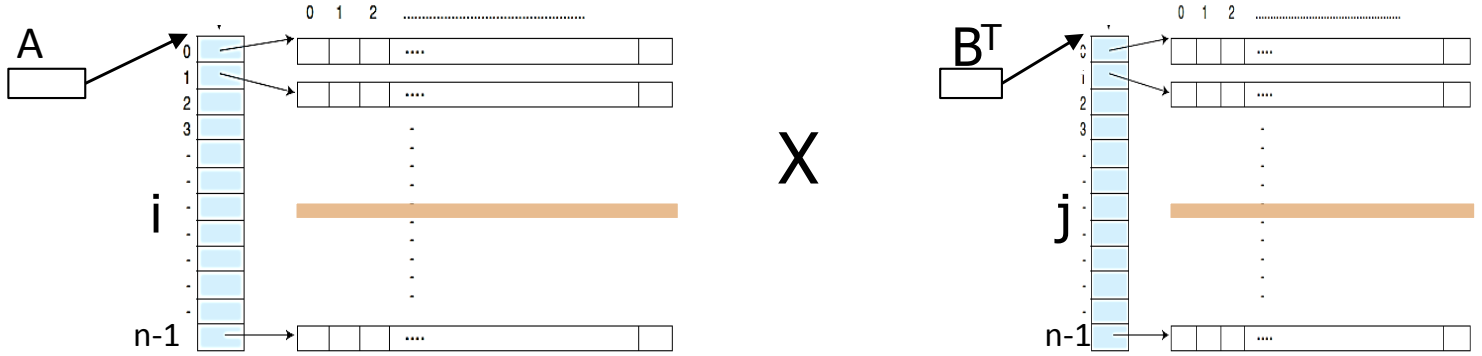


Begrunnelse: Det blir for mange cache-linjer (hver 64 byte) fra B i cachene

Vi har da at litt ny formel for C

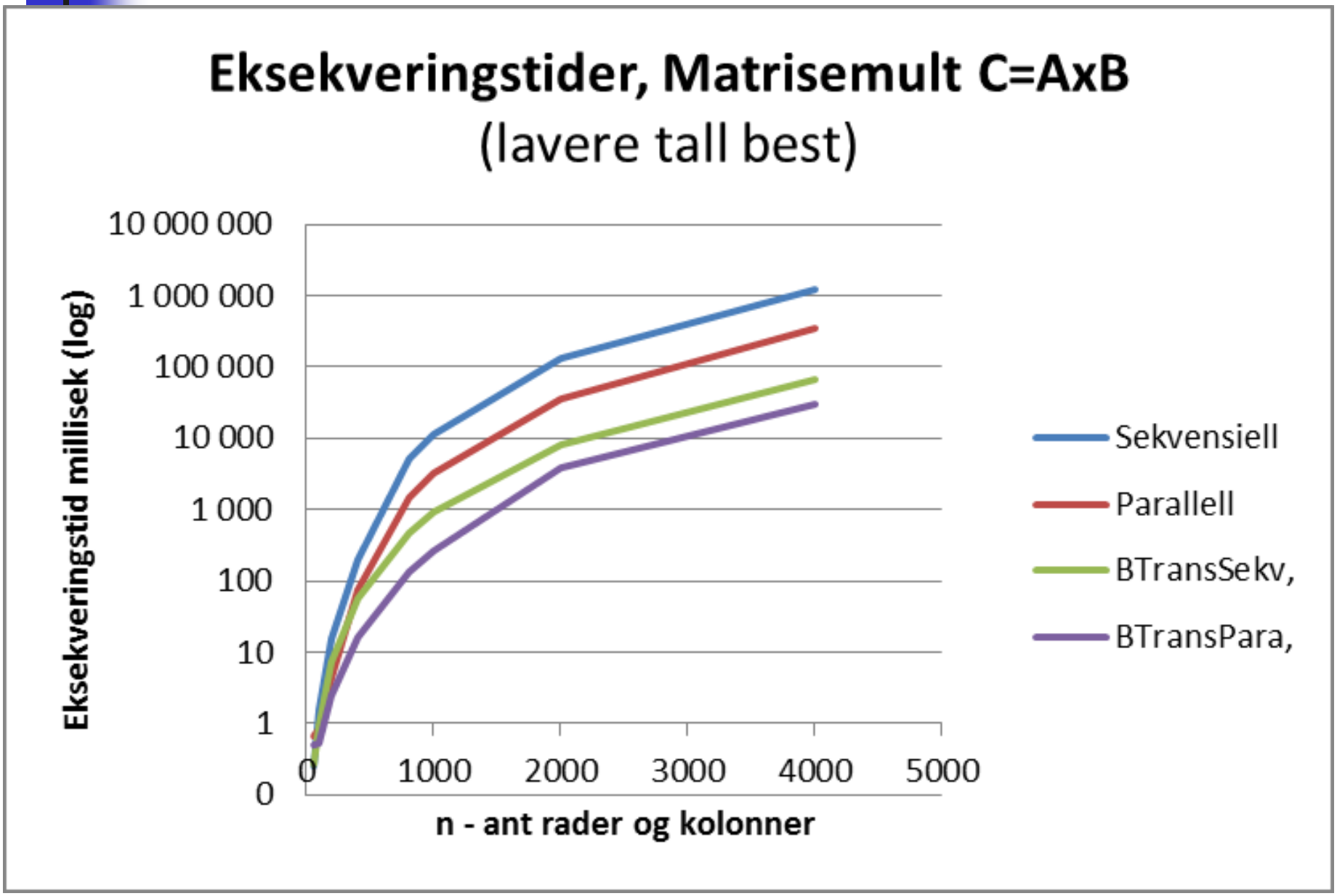
$$c[i][j] = \sum_{k=0}^{n-1} a[i][k] * b^T[j][k]$$

$C_{i,j} =$

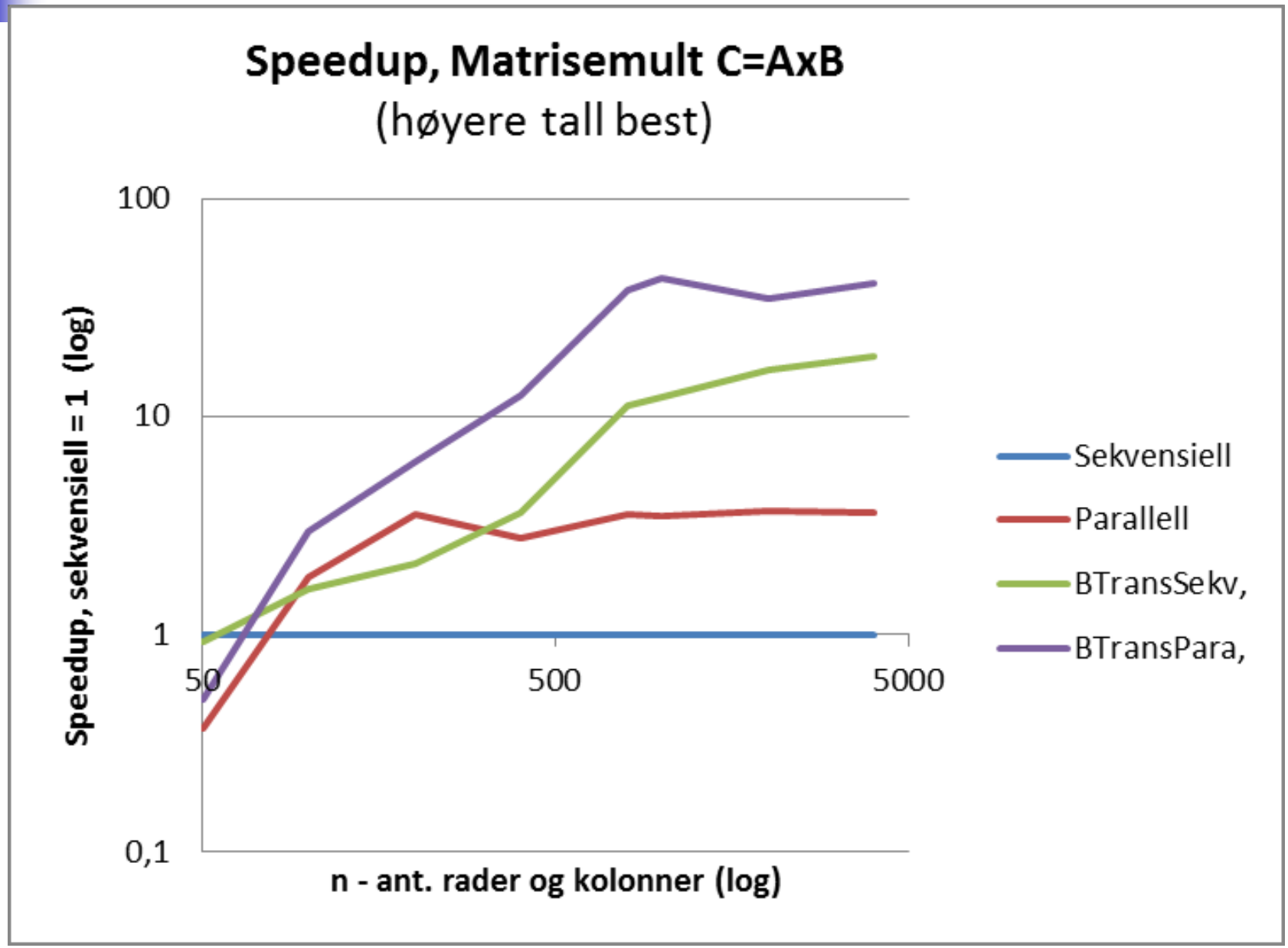


Vi får multiplisert to rader med hverandre – går det fortere ?

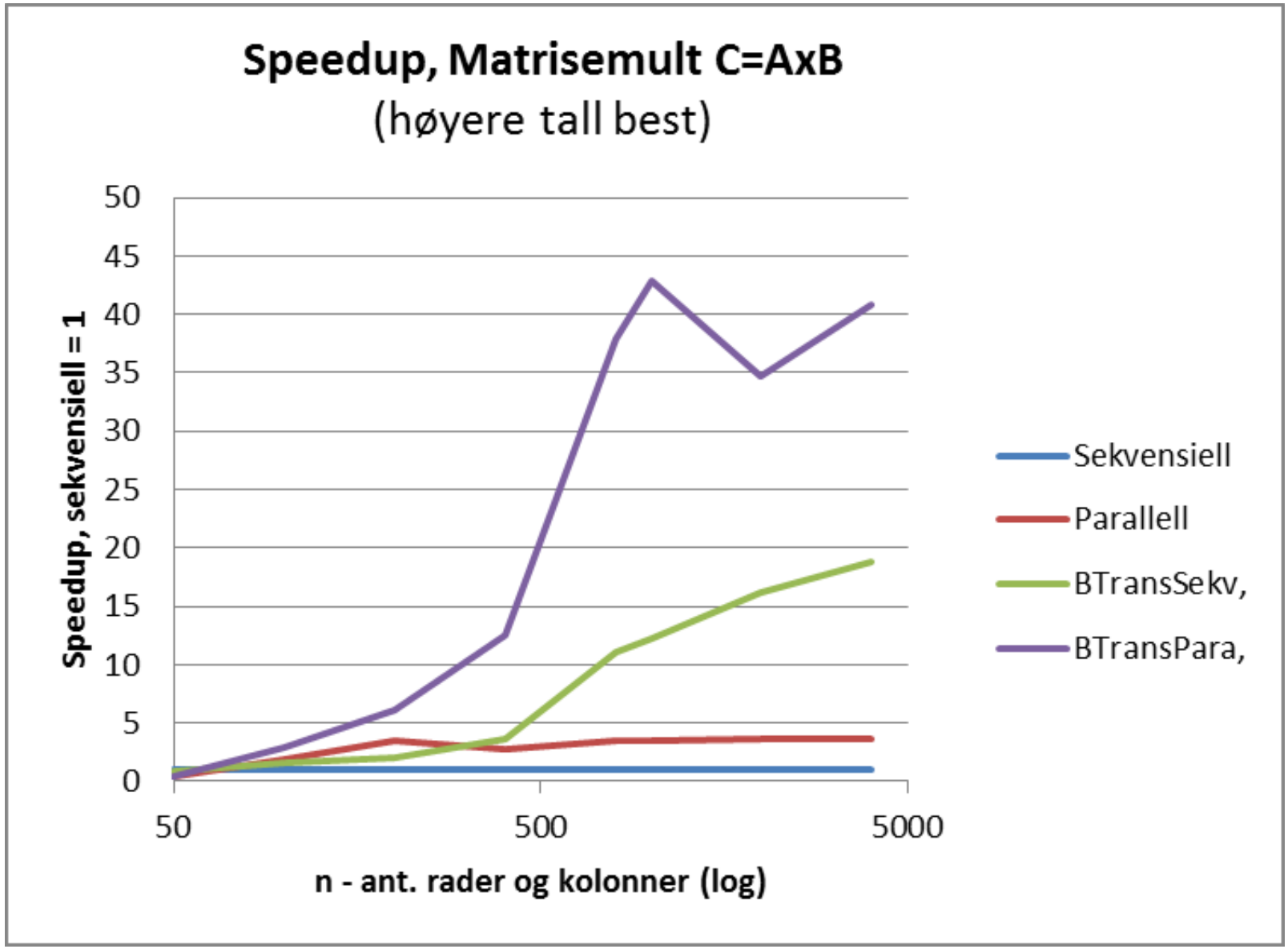
Kjøretider – i millisek. (y-aksen logaritmisk)



Kjøretidsresultater – Speedup , y-aksen logaritmisk



Speedup – med lineær y-akse

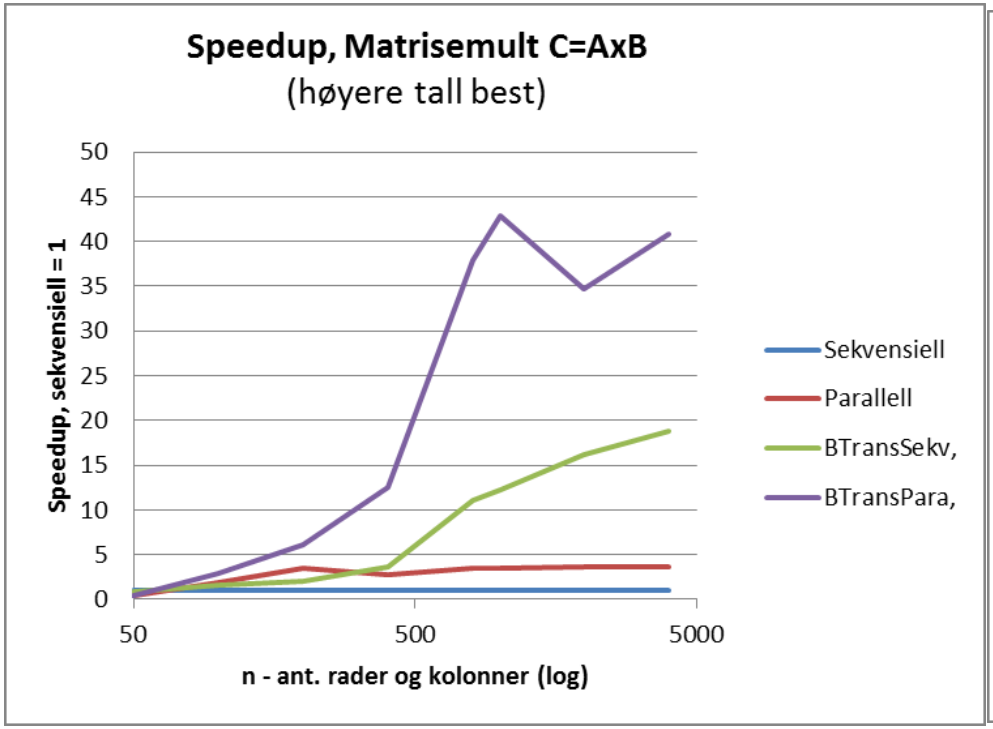


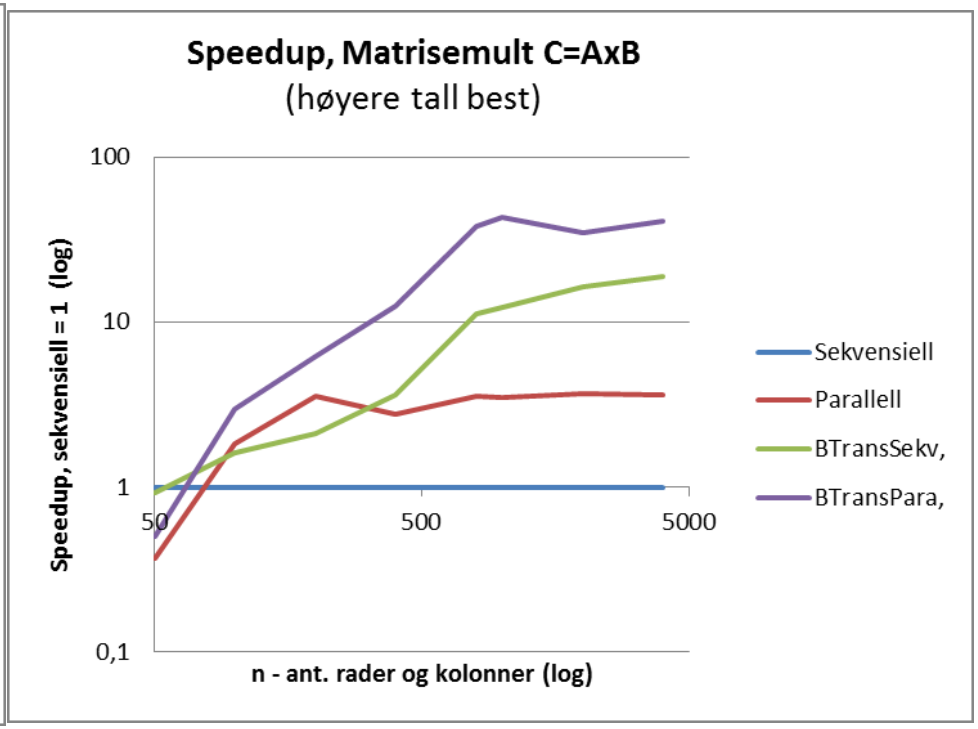
Konklusjon – Matrisemultiplikasjon

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

Speedup – Hvordan fremstille det?

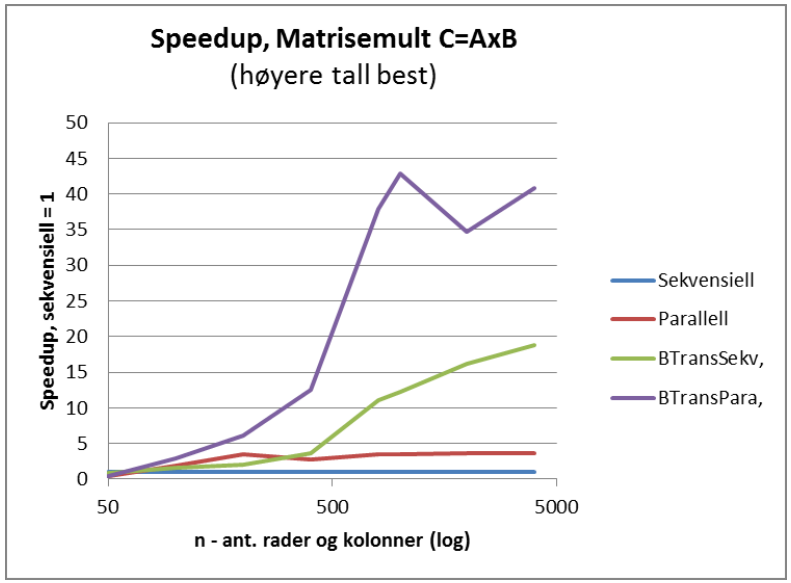
- Disse to kurvene er like ! Hvordan ?

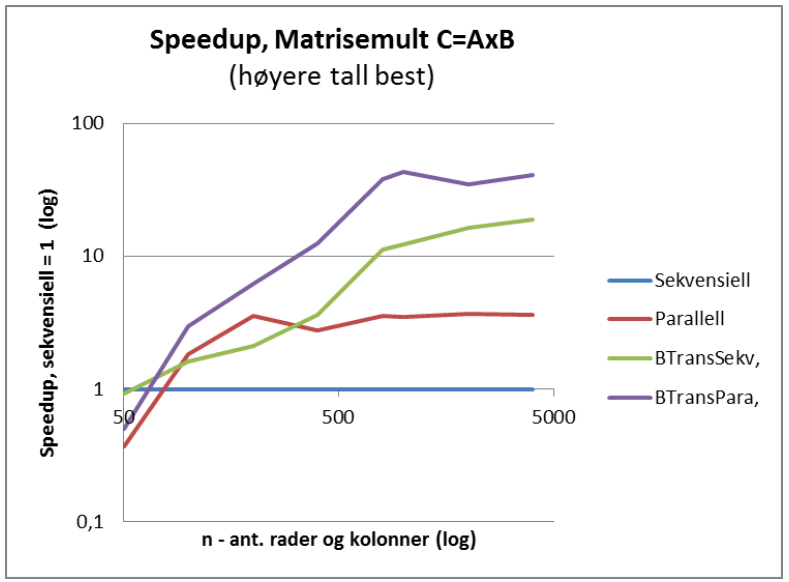


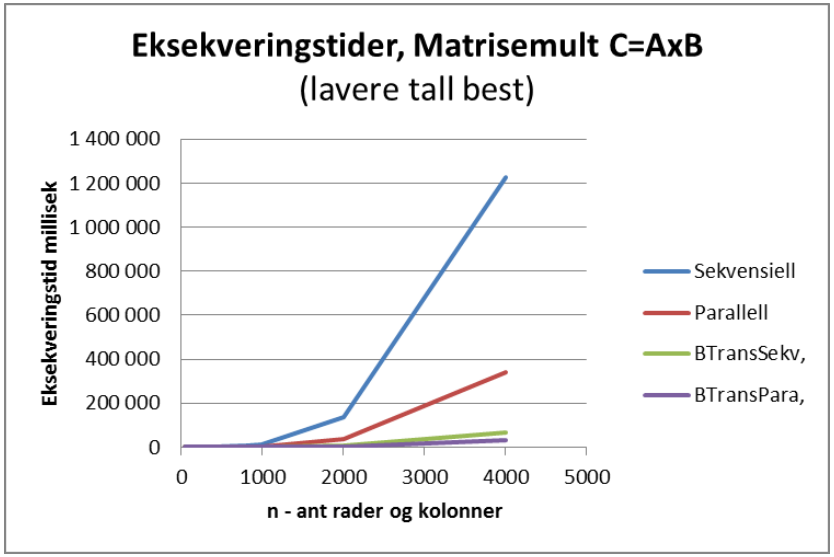


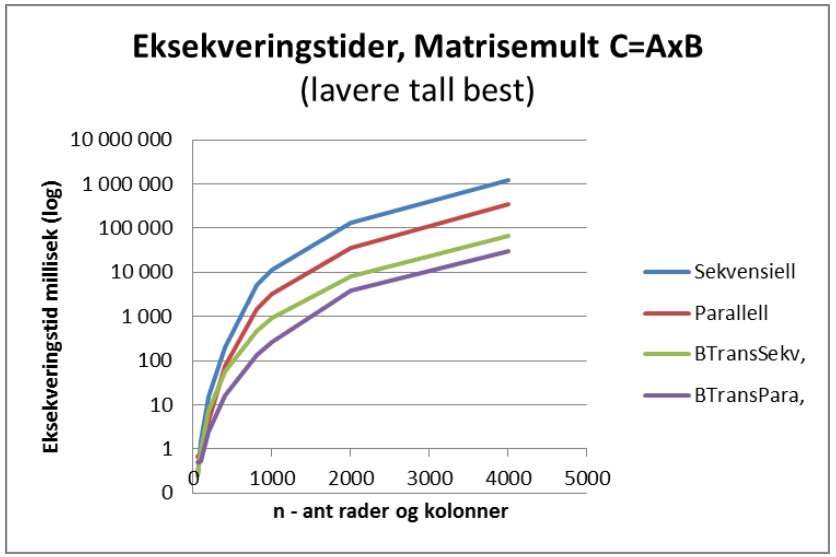
Hvordan framstille ytelse I

- Disse fire kurvene fremviser samme tall! Hvordan ?

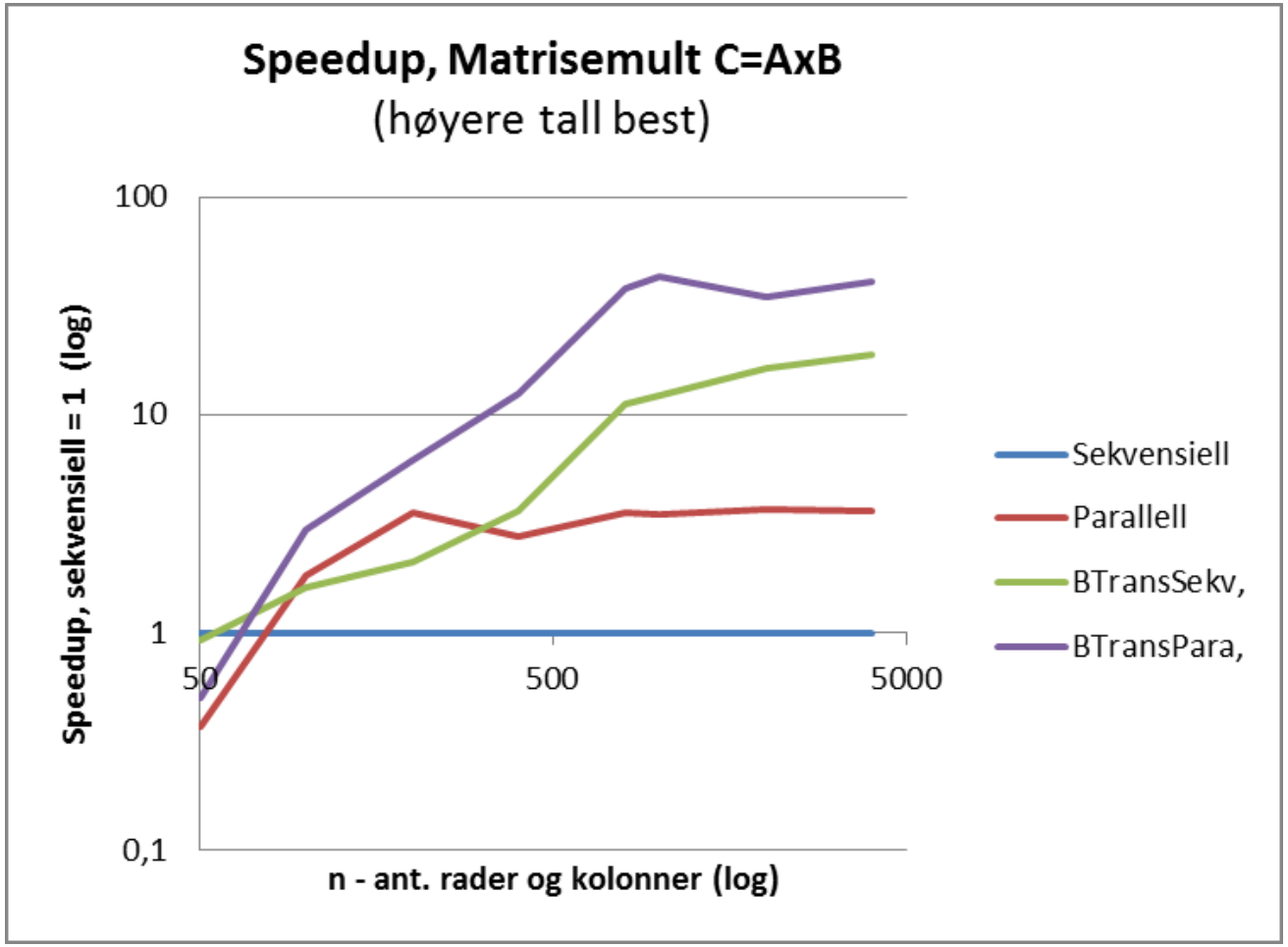






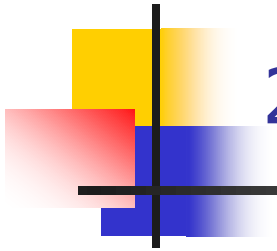


Både logaritmisk x- og y-akse



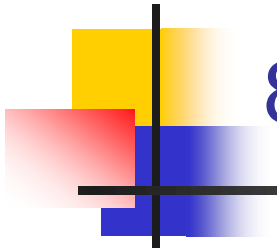
Fordel med log-akse er at den viser fram nøyaktigere små verdier, men vanskelig å lese nøyaktig mellom to merker på aksene.

2) PRAM modellen for parallelle beregninger



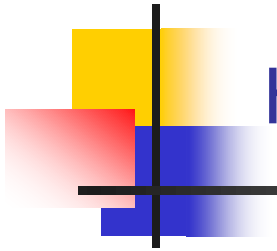
- PRAM (Parallel Random Access Memory) antar t ting:
  - Du har uendelig mange kjerner til beregningene
  - Enhver aksess i lageret tar like lag tid,
    - ignorerer f.eks problemet med cache-hukommelsen
  - Alle tråder går synkront, starter på samme tidspunkt
- Da blir mange algoritmer lette å beregne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

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



8) Kommentarer til : Oblig 1

- Oblig 1 har ikke spesifisert hvordan dere skal finne de ‘k’ største tall 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.



Hva her vi sett på i Uke3

1. Modell(er) for hvordan vi programmerer
2. Viktige regler om lesing og skriving på felles data.
3. Synlighetsproblemet
    1. Hvilke verdier ser ulike tråder som leser variable som en annen tråd skriver på?
4. Hvor fort kan JIT-kompilert kode gå?
5. Prefetch-mekanismen i elektronikken
6. Java har 'as-if sequential' semantikk (JIT-kompilering omformer koden mye, men den virker som vi skrev)
7. Effekten på eksekveringstid av cache
    1. Lese og skrive a[b[i]]=i;
    2. Radix-sortering sekvensiell, del 1 – mer kommer
    3. Det er ikke antall instruksjoner, med at data passer inn i cachene som teller!
8. Matrix Multiplikation
9. Kommentarer til obligene og nå:Oblig 1

```
import java.util.*;
import java.util.concurrent.*;
import easyIO.*;

class Prefetch{
        static CyclicBarrier vent,ferdig ;
        static int [] a;
        static int antTraader;
        static int antKjerner;
        static int[]lokalSum ;
        static int aLen, step;
        static int numIter;
        volatile boolean stop = false;
        static Prefetch p = new Prefetch();
        static String res;
        Random r;

        void intitier(int [] param, int step) {
                if ( vent == null) {
                        // bare initier forste gang - da er f.eks
vent == null
                         antKjerner =
Runtime.getRuntime().availableProcessors();
                         antTraader = antKjerner;
                         vent   = new
CyclicBarrier(antTraader+1); //+1, ogsaa main
                         ferdig = new
CyclicBarrier(antTraader+1); //+1, ogsaa main
                         lokalSum = new int [antTraader];
                         a = param;
                         r = new Random(123456);

             // start threads
             for (int i = 0; i< antTraader; i++)
                                  new Thread(new Para(i,
step)).start();
                  }
        } // end initier


        public static void main (String [] args) {
                if ( args.length != 4)   {
                        System.out.println("use: >java Prefetch
<length of array> <step 1,-1,9, -9> <num iterations> >
<resultatfil>");
            } else {
                    aLen = Integer.parseInt(args[0]);
                    step = Integer.parseInt(args[1]);
                    numIter = Integer.parseInt(args[2]);
                    res = args[3];
                    p = new Prefetch();
                    p.utforTest(aLen, step);

            }
```

```
          } // end main

          void println(String s) {
                  System.out.println(s);
                  Out r = new Out(res,true);
                  r.outln(s);
                  r.close();
          }// end println


    int SumSeq( int [] a, int step) {
                  int sum = 0;
                  int index  =0;

                  for (int i=0;i < a.length; i++){
                          //index = Math.abs((index+step)
%a.length ); // fast steg
                          index = Math.abs((index+r.nextInt(step+1))
%a.length) ;  // tilfeldig valg av steg: 0,..,step
                          sum += a[index];
                   }
               return sum;
           }// end SumSeq



         void  utforTest (int n, int step) {
             a= new int[n]; // okende lengde med okende antTraader

             int totalSum = -1;

                 Random r = new Random(1337);
                 for (int i =0; i< a.length;i++)
                    a[i] = Math.max(r.nextInt(10),0); // random fill
>=0

                  double [] seqTime = new double [numIter];
                  double [] parTime = new double [numIter];


              for (int j = 0; j < numIter; j++) {

                          long t ;
                          t = System.nanoTime();       // start
klokke

                           // try findin max numIter times

                           totalSum = SumPara(a,step);

                           t = (System.nanoTime()-t);
                            parTime[j] = t/1000000.0;
              System.out.println("\n-Tester prefetch-mekanismen ved
aa steppe i (1,-1,...) i en stor array");
```

```
                            System.out.println("Kjoring:"+j+", ant
kjerner:"+ antKjerner + ", antTraader:"
                                                + antTraader+",
step:"+step);
                            System.out.println("Sum verdi parallell
i a:"+totalSum +
                            ", paa: "+  parTime[j]+ " ms. " );

                            // sammenlign med sekvensiell utforing av
finnSum
                            t = System.nanoTime();

                            totalSum = SumSeq(a,step);

                            t = (System.nanoTime()-t);
                            seqTime[j] =t/1000000.0;

                        System.out.println("Sum verdi sekvensiell i
a:"+totalSum +", paa: "+
                                        seqTime[j]+ " ms."+",
step:"+step);
        } // end for j
        insertSort(seqTime, 0, numIter-1);
        insertSort(parTime, 0, numIter-1);

        println("\nMedian sequential time:"+seqTime[seqTime.length/
2]+
        "ms., median parallel time:"+ parTime[seqTime.length/2]
        + "ms.,\n step:"+step+", Speedup:"+
Format.align(seqTime[seqTime.length/2 ]/parTime[parTime.length/2],
6,2)
         +", n = "+n);

        stop = true;
        try{ // make all threads terminate
                                vent.await();
                } catch (Exception e) {}

         } // utfor

         int SumPara (int [] a, int step) {
                    // gjor synkronisering for traadene
                    intitier (a, step);
                    int totalSum = 0;

                    try{ // start all treads
                                vent.await();
                         } catch (Exception e) {return 0;}

                         try{ // wait for all treads to complete
                                ferdig.await();
                         } catch (Exception e) {return 0;}

                         // finn den storste max fra alle traadene
```

```
                        for (int i=0;i < antTraader;i++)
                           totalSum += lokalSum[i];

                     return totalSum;

         } // end finnSum


          class Para implements Runnable{
                     int ind;
                     int step;
                     Para(int i, int step) { ind =i; this.step =
step; r = new Random(123+i); }// konstruktor
                     Random r ;

                    void FindMinSum(int ind, int step){
                           int ant= a.length/antTraader;   // antall
elementer per traad
                           int num = ant;
                           if (ind == antTraader-1) num = ant +
(a.length % antTraader) ;
                           int start =  ant*ind;
                           int end    = start+num;
                           int index  =0;

                           int sum = 0;

                           for (int i=start;i < end; i++){
                                  index = Math.abs((index+step)
%a.length) ;           // fast steg
                                  //index =
Math.abs((index+r.nextInt(step+1))%a.length) ;  // tilfeldig valg av
steg 0,..,step
                                  sum += a[index];
                            }

                            lokalSum[ind] =sum;  // levÈr svar

               } // end FindLocalSum

                public void run() { // Her er det som kjores i
parallell:

                            while (! stop) {

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

                            if (! stop) {

                                         FindMinSum(ind,step);
```

```
                                                try {  // wait on all
other threads + main

ferdig.await();
                                                } catch (Exception e)
{return;}

                                           }// end stop thread

                                 } // while
                              } // end run
     } // end class Para

          /** sort double-array a to find median */
          void insertSort(double a[],int left, int right) {
                                      int i,k;
                                      double t;

                                      for (k = left+1 ; k <= right; k++)
{
                                              t = a[k] ;
                                              i = k;

                                              while ((a[i-1] ) > t ) {
                                                       a[i] = a[i-1];
                                                       if (--i == left)
break;
                                              }
                                              a[i] = t;
                                      } // end k
          } // end insertSort
}// END class Parallell
```

```
import java.util.concurrent.*;


/** Start >java SamLes <ant traader>
Viser at lesing av elementer skrevet av andre gÂr galt'*/

public class SamLes{
        int  a=0,b=0;               // Felles variable a,b
        CyclicBarrier sync, vent ;     // sikrer at begge starter
'samtidig' er ferdige f¯r utskrift
        int antTraader =2, antGanger ;
    SkrivA aObj;
    SkrivB bObj;

        void utskrift() {
           int num =20,j=0, max=4, aIndex,bIndex =0;

           System.out.println("               SkrivA
SkrivB");

           for (int i = 0; i < antGanger; i++) {
                       // skip uinteressante tilfeller
                            while (j < antGanger && (
                                  aObj.mA[j] == 0 || aObj.mB[j] ==
0||
                                  bObj.mA[j] == 0 || bObj.mB[j] ==
0)) j++;

                            aIndex = j;
                            while (bIndex < antGanger &&
                                 bObj.mA[bIndex] !=
aObj.mA[aIndex]) bIndex++;

                                    // skriv ut 10 interessante
tilfeller
                                    for (int k = 0; k < 10 &&
k+aIndex < antGanger && k+bIndex < antGanger; k++) {
                                                System.out.println(
                                                      "a.mA[" +
(k+aIndex) + "]= "+ aObj.mA[k+aIndex] +
                                                      "  a.mB["+
(k+aIndex)  +  "]= "+aObj.mB[k+aIndex] + "  | "+
                                                      " b.mA[" +
(k+bIndex) + "]= "+ bObj.mA[k+bIndex] +
                                                      "  b.mB["+
(k+bIndex)  + "]= "+bObj.mB[k+bIndex] );

                                    } // end k

                            System.out.println("--");
                            max--;
                            if (max < 0 ) return;
                            j = aIndex +10;
           } // end i
```

```
    } // end utskrift


        public static void main (String [] args) {
                if (args.length != 1) {
                        System.out.println(" bruk: java <ant
ganger oeke> ");
                 } else {
                        int antKjerner =
Runtime.getRuntime().availableProcessors();
                        System.out.println("Maskinen har "+
antKjerner + " prosessorkjerner.\n");
                        SamLes p =  new SamLes();
                        p.antGanger  = Integer.parseInt(args[0]);
                        p.utfor();
                 }
        } // end main

        void  utfor () {
            vent = new CyclicBarrier((int)antTraader+1); // vent
mellom algoritmer og tidtaging
        sync = new CyclicBarrier((int)antTraader);

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

                 try{
                   // main thread wenter pÂ 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 all other
threads + 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 all other
threads + main
                                                  vent.await();
                                         } catch (Exception e) {return;}
                              } // end run B
     } // end class Para


}// END class Parallell
```

```
import java.util.concurrent.*;
import java.util.*;
import easyIO.*;
// enklest mulig modell-kode for parallellitet,
// se Model2.java for tidtaking mm
class Tider2018_komp {
        Out ut;
        int n;

        void println(String s){
                System.out.println(s);
                ut.outln(s);
        }

     //  felles data og metoder A
      public static void main(String [] args) {
                  if ( args.length != 2) {
                                              System.out.println("use:
>java Tider2018_komp  <n>  <fil>");
               } else {
             Tider2018_komp p = new Tider2018_komp();
             p.utfoer(args);
                  }
      }

      int neste(int i){
                  return i+1;
          }

      void utfoer (String [] args) {
                  n = Integer.parseInt(args[0]);
                  int aLen = 100; //   Integer.parseInt(args[1]);
                  ut = new Out(args[1],true);  // Open with append
          int k=0;
          long m=0;
                  long t =0;
                  double d,f1,f2;
                  String s;
                  int [] ia,ib;
                  Thread t1,t2;
                  Thread [] tt = new Thread[2];
                   double [] db, db2;
                  Random r;
                  long[] tid = new long[n];



                  class C { int i;C(int i){ this.i =i;} int les ()
{ return i+10;} } // end C
          class D { int i;D(int i){ this.i =i;} int les () { return
i+11;} } // end D

          println("\nGjennomsnittstider per gang.\n");
          s ="for-loop"+Format.align(n,7)+"x             ";
```

```
                    k =0;


               t = System.nanoTime();
                      for (int j = 0 ; j <= n; j++ ) {

                        for (int i = 0; i<= 1000; i++) {
                             k+= i;
                        }
                      }
                      m = (System.nanoTime() -t);
                      if (m == 0) println("ERR m 0");
                      d = (double) (m/(n*1000.0));
                      println(s+Format.align(d,10,4)+"us.midlere verdi
" + n+" ganger"+k);

                //Metodekall ---------
                      s ="Metodekall                  ";
                      t = System.nanoTime();

                      for (int i = 0; i<=n; i++) {

                              k+=neste(k);
                      }
                      m = (System.nanoTime() -t);
                      if (m == 0) println("ERR m 0");
                      d = (double) (m/(n*1000.0));
                      println(s+Format.align(d,11,4)+"us.midlere verdi
" + n+" ganger");

             // Array -------------
                      s ="Array new len           ="+aLen+"" ;
                     t = System.nanoTime();
                      for (int i = 0; i<=n; i++) {

                         ia = new int[aLen];
                      }
                      m = (System.nanoTime() -t);
                      if (m == 0) println("ERR m 0");
                      d = (double) (m/(n*1000.0));
                      println(s+Format.align(d,10,3)+"us.midlere verdi
" + n+" ganger");


             // Array-copy ----------------------
                      s ="Array copy for-loop l=100   ";
                      ia = new int[aLen];
                      for (int i=0;i<aLen;i++) ia[i] = i+5;
                      ib = new int[aLen];
                       t = System.nanoTime();
                      for (int i = 0; i<=n; i++) {

                              for (int j = 0;j<aLen;j++)
                                 ib[j] = ia[j] ;
```

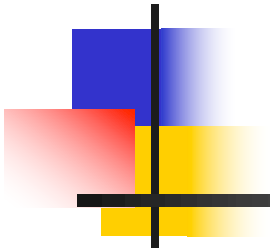
```
                    }
                    m = (System.nanoTime() -t);
                    if (m == 0) println("ERR m 0");
                    d = (double) (m/(n*1000.0));
                    println(s+Format.align(d,10,3)+"us.midlere verdi
" + n+" ganger");


                    // System.arraycopy ----------------------
                    s ="Array copy arraycopy l=100, ";
                    ia = new int[aLen];
                    for (int i=0;i<aLen;i++) ia[i] = i+5;
                    ib = new int[aLen];
           t = System.nanoTime();
                    for (int i = 0; i<=n; i++) {
               System.arraycopy(ia,0,ib,0,aLen);
                    }
                    m = (System.nanoTime() -t);
                    if (m == 0) println("ERR m 0");
                    d = (double) (m/(n*1000.0));
                    println(s+Format.align(d,10,3)+"us.midlere verdi
" + n+" ganger");

/*        // ------------ new thread ---------------
                    s ="new Thread() start & join   ";
           t = System.nanoTime();
                    for (int i = 0; i<=n; i++) {

                              (t1 = new Thread(new
Arbeider(3))).start();
                              try{t1.join();}catch (Exception e){}
                    }
                    m = (System.nanoTime() -t);
                    if (m == 0) println("ERR m 0");
                    d = (double) (m/(n*1000.0));
                    println(s+Format.align(d,10,3)+"us.midlere verdi
" + n+" ganger");
           System.gc();

*/


     // ------------ Two  new threadc  individually ---------------
                    s ="2sep new Thread() start & join ";
           t = System.nanoTime();
                    for (int i = 0; i<=n; i++) {

                               (t1 = new Thread(new
Arbeider(3))).start();
                               (t2 = new Thread(new
Arbeider(i))).start();

                                try{t1.join();t2.join();}catch
(Exception e){}
```

```
                    }
                    m = (System.nanoTime() -t);
                    if (m == 0) println("ERR m 0");
                    d = (double) (m/(n*1000.0));
                    println(s+Format.align(d,10,3)+"us.midlere verdi
" + n+" ganger");
          System.gc();

        // ------------ Two new new threads in array ---------------
                    s ="2 arr new Thread() start & join";
          t = System.nanoTime();
                    for (int i = 0; i<=n; i++) {

                            for (int j=0;j<2; j++) {
                                    (tt[j] = new Thread(new
Arbeider(3))).start();
                            }
               try{  for (int j=0;i<2; j++){ tt[j].join(); }}
               catch (Exception e){};

                    }
                    m = (System.nanoTime() -t);
                    if (m == 0) println("ERR m 0");
                    d = (double) (m/(n*1000.0));
                    println(s+Format.align(d,10,3)+"us.midlere verdi
" + n+" ganger");
          System.gc();




                    //---------------- new class-------------------
                    s ="new Class C+ metodekall     ";
                    t = System.nanoTime();
                    for (int i = 0; i<=n; i++) {

                             k= new C(k).les();
                    }
                    d = (double) ((System.nanoTime() -t)/(n*1000.0));
                    println(s+Format.align(d,10,3)+"us.midlere verdi
" + n+" ganger");
          m=0;
                    s ="new Class D+ metodekall   ";
                    for (int i = 0; i<=n; i++) {
                             t = System.nanoTime();
                                   k= new D(k).les();
                                    m += (System.nanoTime() -t);
                    }

                    if (m == 0) println("ERR m 0");
                    d = (double) (m/(n*1000.0));
                    println(s+Format.align(d,12,3)+"us.midlere verdi
" + n+" ganger");
```

```
                    s ="Array write                   ";

                    t = System.nanoTime();
                    for (int i = 0; i< n; i++) {

                            ia[i%aLen] =i;
                    }
                    m = (System.nanoTime() -t);
                    if (m == 0) println("ERR m 0");
                    d = (double) (m/(n*1000.0));
                    println(s+Format.align(d,10,5)+"us.midlere verdi
" + n+" ganger");


                    //Array les
                    s ="Array les                   ";
            t = System.nanoTime();

                    for (int i = 0; i<=n; i++) {
              // t = System.nanoTime();
                                    k = ia[i%aLen];
                           //  m += (System.nanoTime() -t);

                    }
                    m = (System.nanoTime() -t);
                    if (m == 0) println("ERR m 0");
                    d = (double) (m/(n*1000.0));
                    println(s+Format.align(d,12,5)+"us.midlere verdi
" + n+" ganger, k:"+k);


              //Convert double to long
                            s ="Double to long            ";
                             db = new double[n];
                              r = new Random(123);
                              m=0;

                             for (int i = 0; i<n; i++) {
                                     db [i]= r.nextDouble();
                             }
                            long ll=0;

                       t = System.nanoTime();
                            for (int i = 0; i<n; i++) {
                                     //t = System.nanoTime();
                                           ll =
Double.doubleToLongBits(db[i]);

                            }
                            m = (System.nanoTime() -t);
                            if (m == 0) println("ERR m 0");
                            d = (double) (m/(n*1000.0));
                    println(s+Format.align(d,14,5)+"us.midlere verdi
" + n+" ganger"+ll);
```

```
                    //Convert double to long RawBit
                                s ="Double to longRaw        ";
                                 db = new double[n];
                                  r = new Random(123);
                                  m=0;

                                 for (int i = 0; i<n; i++) {
                                         db [i]=
r.nextDouble();
                                 }
                                ll=0;

                            t = System.nanoTime();
                                   for (int i = 0; i<n; i++) {
                                           //t =
System.nanoTime();
                                                ll =
Double.doubleToRawLongBits(db[i]);
                                   }
                                   m = (System.nanoTime() -t);
                                   if (m == 0) println("ERR m 0");
                                   d = (double) (m/(n*1000.0));
                    println(s+Format.align(d,14,5)+"us.midlere verdi
" + n+" ganger"+ll);


               //Test simple insertsort
                     s ="simple insertsort1        ";
                     int len = Math.min(aLen,2000);
                     db = new double[len];
                     db2 = new double[len];
                     r = new Random(123);
                      for (int i = 0; i<len; i++) {
                               db [i]= r.nextDouble();
                               db2[i] = db[i];
                      }
           m=0;
                     for (int i = 0; i<=n; i++) {
                               System.arraycopy(db2,0,db,0,len);
                               t = System.nanoTime();
                                     insertSort1(db,0,len-1);
                                     m += (System.nanoTime() -t);
                     }
                     //m = (System.nanoTime() -t);
                     if (m == 0) println("ERR m 0");
                     d = (double) (m/(n*1000.0));
                     println(s+Format.align(d,12,3)+"us.midlere verdi
" + n+" ganger, len:"+len);

        ut.close();

      }
```

```
        class Arbeider implements Runnable {
                int ind;
                // lokale data og metoder B til hver traad
                       Arbeider (int in) {ind = in;}
                        public void run( ) {
                            ind++;
                        } // end run
      } // end indre klasse Arbeider

      public static void insertSort1(double a[],int left, int right)
{
           int i,k;
           double t;

           for (k = left ; k < right; k++) {
               t = a[k+1];
               i = k;

               while (i >= left && a[i] > t) {
                    a[i+1] = a[i];
                    i--;
               }
               a[i+1] = t;
         }
    } // end insertSort1


} // end class Tider2017
```

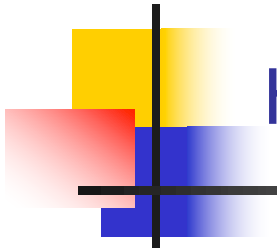


IN3030 Uke 4, v2019 – Om å samle parallelle svar, matrisemultiplikasjon og The Java Memory Model

Eric Jul

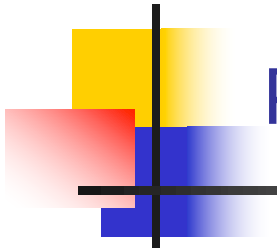
PSE,

Inst. for informatikk



Hva så vi på i uke 3

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



Plan for uke 4

I. Om å avlutte k tråder med å samle svarene fra disse. 1 gal + 5 riktige måter.

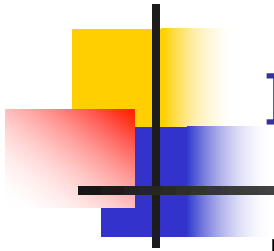
II. Vi bruker ikke PRAM modellen for parallelle beregninger

   I. Hva er PRAM og hvorfor er den ubrukelig for oss

III. Hva skjer egentlig i lageret (main memory) når vi kjører parallelle tråder - the Java Memory Model

IV. Hvorfor synkroniserer vi, og hva skjer da ?

V. Oblig 2: Parallell Matrice-multiplikasjon.



I) Hvordan samle data fra trådene til ett, felles 'svar'

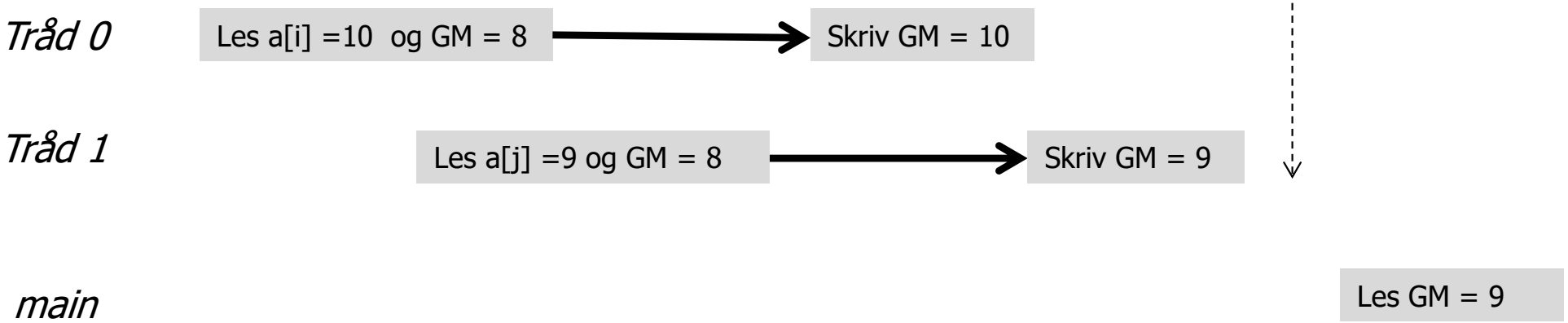
- Kursets mål: Riktig og Raskt - begge deler
- Skal se på ulike måter å løse flg. problem:
  - Vi har delt opp problemet vår i k deler på k kjerner/tråder og parallellisert det.
  - Hvordan kombinerer vi disse delresultatene til ett, felles og riktig svar?
  - Ser på problemet FinnMax og 6 «løsninger» på denne avslutningen:
    1. GAL – ingen synkronisering
    2. Synchronized metode for hvert element i a[]
    3. ReentrantLock beskyttet metode for hvert element i a[]
    4. Bare kall på samme ReentrantLock beskyttet metode hvis ny verdi > hittil største verdi.
    5. Alle trådene kaller samme ReentrantLock beskyttet metode med hver sin lokaleMaks når de er ferdige
    6. Tråd 0 finner max – de andre trådene venter

1) GAL – ingen synkronisering, koden i trådene:

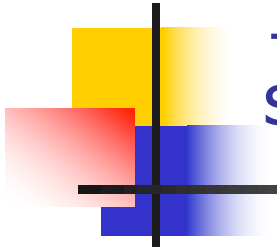
```
// Riktig oppdeling
ant= a.length/antTråder;   // antall elementer per tråd
start = ant*ind;
num = ant;
if (ind == antTråder-1) num = ant + (a.length % antTråder) ;
```

```
// GAL – IKKE synkronisert:
for (int i = 0; i< num; i++) {
      if (a[start+i] > globalMax) globalMax = a[start+i];
 }
```

- Mulig feilsituasjon, anta tråd 0 leser a[i] der GlobalMax ligger:



Hvor sannsynlig er dette? Skal vi ta sjansen ??



1) Nei – det er like raske og ALLTID riktige løsninger, og slike feil er farlige !

- Det er vanskelig å få denne gale måten til å feile, men med mye prøving og feiling (lure data) kan vi få den til å feile ca. ca. hver 100 000 gang.
- I Nets (tidl. Bankenes Betaligssentral) behandler de opp til 2 mill. transaksjoner hver dag.
- Vil vi lage programmer som kanskje feiler 20 ganger per dag?
- Hvor lette tror du slike programmer er å debugge (teste og rette) ?

2) Bruk av synchronized metode for alle elementene i a[]

```
 synchronized void updateGlobalMax1 (int val){
                        if (val > globalMax )  globalMax=val;
} // end updateGlobalMax1


……..
// i run-metoden i hver tråd:
  for (int i = 0; i< num; i++) {
            updateGlobalMax1(a[start+i]);
 }
```

- 100 % riktig svar alltid
- Det gjøres n kall på den synkroniserte metoden (n/k kall fra hver av de k trådene)
- synchronized tar mye tid !
- Elendig løsning: Regel 1: bruk ikke synchronized
- Regel 2: Hvis synchronized alikevel, så max 100 gange!

3) Bruk av ReentrantLock metode for alle elementene i a[]

```
 void updateGlobalMax2 (int val){
            lock.lock();
            try{
                 if (val > globalMax )  globalMax=val;
            } finally {  lock.unlock(); }
 } // end updateGlobalMax2
.....
// i run-metoden i hver tråd
 for (int i = 0; i< num; i++) {
        updateGlobalMax2(a[start+i]);
}
```

- 100 % riktig svar alltid
- Det gjøres n kall på lock metoden (n/k kall fra hver av de k trådene)
- Lock er raskere enn synchronized, men tar fortsatt mye tid !
- Elendig løsning

4) Bruk av ReentrantLock metode hver gang nytt element er større

```
void updateGlobalMax2 (int val){
            lock.lock();
            try{
                if (val > globalMax )  globalMax=val;
            } finally {  lock.unlock(); }
} // end updateGlobalMax2
.....
// i run-metoden i hver tråd
for (int i = 0; i< num; i++) {
        if (a[start+i] > globalMax ){
            updateGlobalMax2(a[start+i]);
 }}
```

- 100 % riktig svar alltid
- Det gjøres log n kall på lock metoden (logn/k kall fra hver av de k trådene)
- Lock er raskere enn synchronized, og logn kall er mye mindre enn n (eks. n= 1mill, log n =20) !
- Ikke helt bra løsning

5) lokal max (threadMax) i hver tråd, så oppdatering med lock - metode

```
// i run-metoden i hver tråd
for (int i = 0; i< num; i++) {
       if (a[start+i] > threadMax ) threadMax = a[start+i];
 }
  updateGlobalMax2(threadMax);
}
```

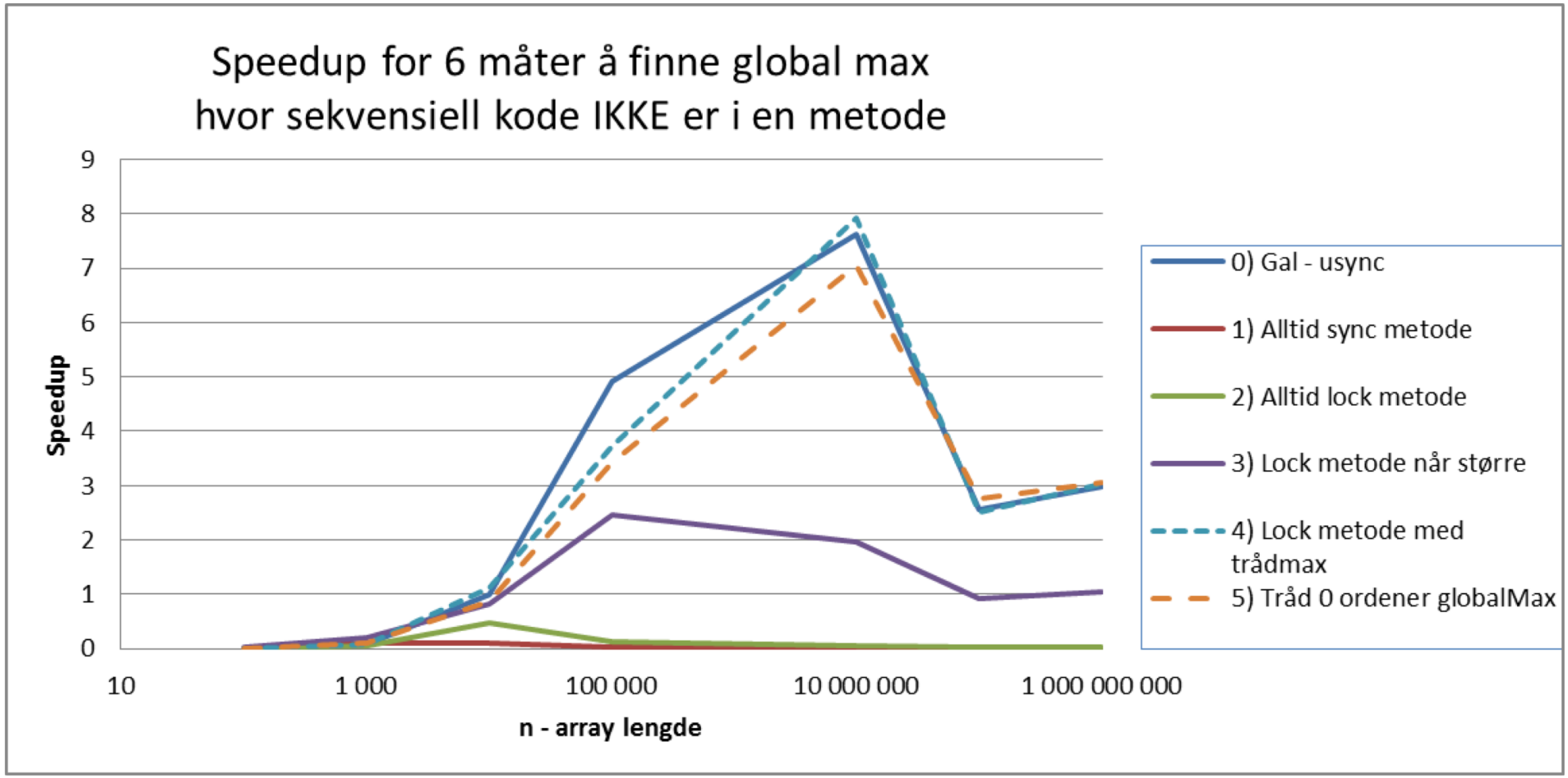
- 100 % riktig svar alltid
- Det gjøres k kall på lock-metoden (ett kall fra hver av de k trådene)
- Utmerket løsning

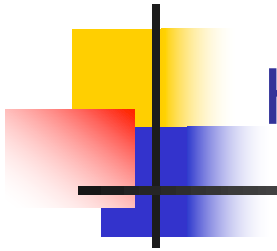
6) lokal max (threadMax) i hver tråd, så venting i ny CyclicBarrier (ventM5) , og tråd 0 ordner opp

```
// i hver tråd i run-metoden
    for (int i = 0; i < num; i++) {
          if (a[start+i] > threadMax ) threadMax = a[start+i];
     }
 lokalMax[ind] = threadMax;

 try {  // wait on all other threads
        ventM5.await();
} catch (Exception e) {return;}

if (ind == 0) {
     for (int j = 0; j < antTråder; j++){
            if (lokalMax[j] > globalMax) globalMax = lokalMax[j];
 } }
```

- Riktig og det gjøres k kall på en CylicBarrier(ett kall fra hver av de k trådene)
- Utmerket løsning

Speedup for n=100,1000,…,1 mrd med 8 tråder/kjerner





Hva lærer vi av dette ?

- En gal, to elendige, en dårlig og to gode løsninger
- Aldri bruk en gal metode (går alltid galt med i++ - programmet, sjelden galt her, men..)
- Det er alltid en eller flere like raske og riktige løsninger
- Antall synkroniseringer er helt avgjørende
  - n, log n eller k (k er et fast, lite tall)

Generelt: Det er langt flere alternativer i det parallelle programmet enn i det sekvensielle.

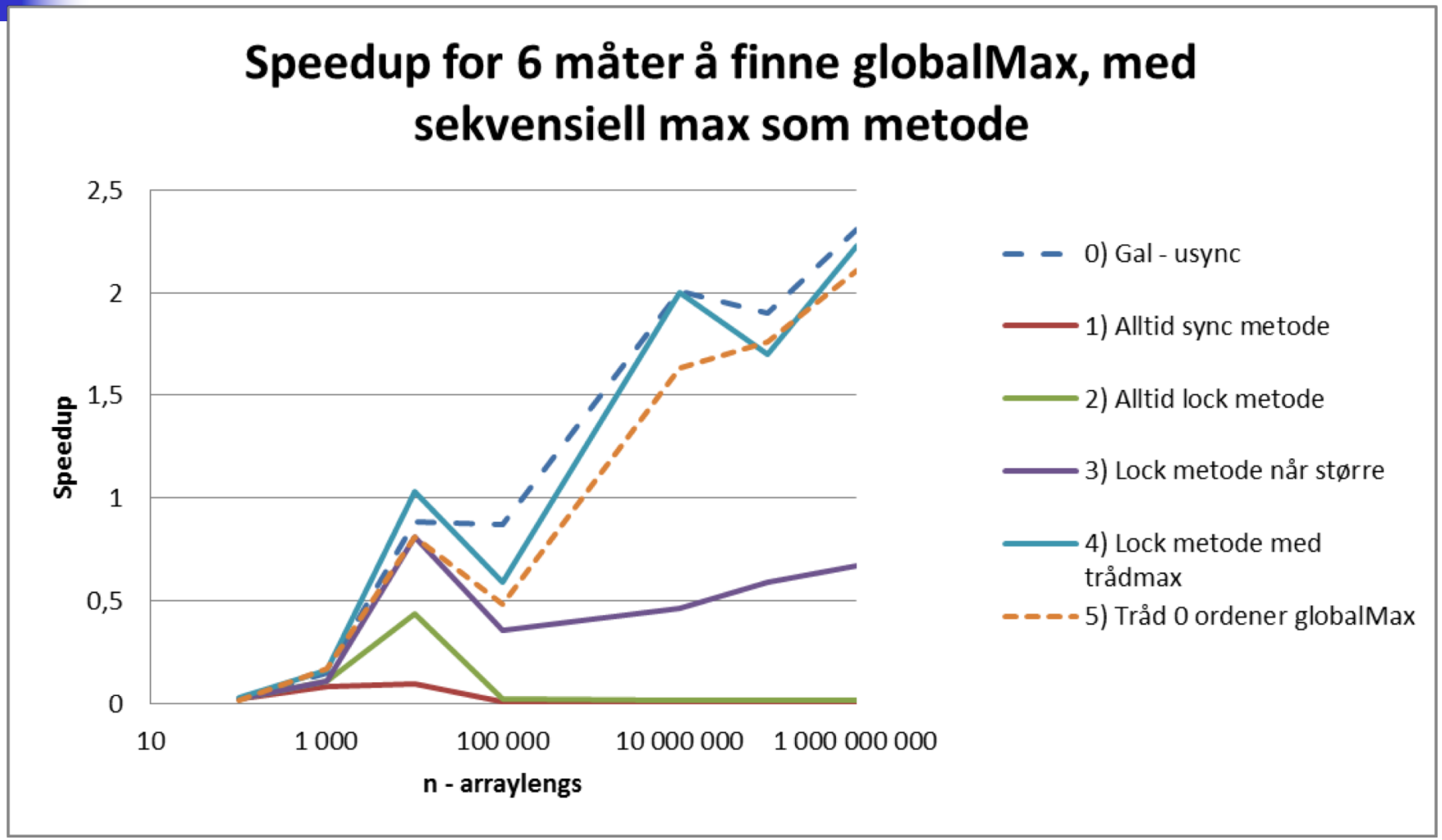
- Her har vi en løsning hvor vi bare lager trådene bare én gang.
- Vi kunne (langsommere) ha laget nye tråder hver gang og at main sa join() på dem – 5 nye alternativer.

Tillegg, hva om sekvensiell max lages som metode?

```
int RegnutMaxSeq(int [] a) {
      int seq =0;
       for (int i=0;i < a.length;i++){
              if (a[i] > seq) seq = a[i];
        }
        return seq;
 }//end RegnutMaxSeq
.....
 globalMaxSeq = RegnutMaxSeq(a);

// for (int i=0;i < a.length;i++){
//     if(a[i] > globalMaxSeq) globalMaxSeq = a[i];
// }
```

- Hvorfor skulle det gå fortere ??
- JIT-kompilering ??

Mye bedre JIT-kompilering av sekvensiell beregning (= raskere sekvensiell bergning); samme parallelle og samme form på kurvene, men da lavere speedup .



- Her ca. 3x raskere - bedre JIT kompilering av små metoder enn koden inline i en større metode.

Ny I DAG: Hvis vi ser på kjøretidene, så burde de kunne gjøres noe bedre !

```
// i run-metoden i hver tråd – noe langsom
for (int i = 0; i< num; i++) {
        if (a[start+i] > threadMax ) threadMax = a[start+i];
 }
  updateGlobalMax2(threadMax);
}
```

```
// bedre: i run-metoden i hver tråd - raskere
for (int i = fra; i< til; i++) {
        if (a[i] > threadMax ) threadMax = a[i];
 }
  updateGlobalMax2(threadMax);
}
```

Ny I DAG: Hvis vi ser på kjøretidene, så burde de kunne gjøres noe bedre med ny for-løkke + små metoder

Speedup går opp fra 1.6 til7,6
Kjøretider for N= 100 mil
- **ny** forløkke og liten metode for hver løkke:

```
Met:0, Para:     19.84 ms; Sekv:   32.343 ms., SUp:  1.6306
Met:1, Para:   4295.32 ms; Sekv:   32.343 ms., SUp:  0.0075
Met:2, Para:   3284.36 ms; Sekv:   32.343 ms., SUp:  0.0098
Met:3, Para:     26.03 ms; Sekv:   32.343 ms., SUp:  1.2427
Met:4, Para:     19.81 ms; Sekv:   32.343 ms., SUp:  1.6324
Met:5, Para:     20.04 ms; Sekv:   32.343 ms., SUp:  1.6141
```

Speedup går opp fra 1.6 til7,6
Kjøretider for N= 100 mil
- **gammel** forløkke og liten metode for hver løkke :

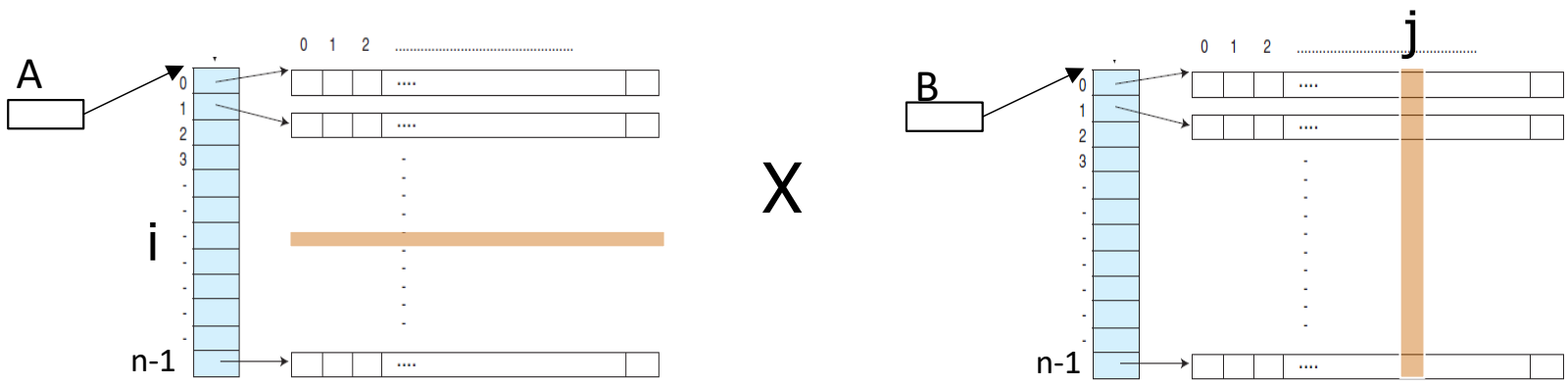
```
Met:0, Para:     20.82 ms; Sekv:   33.987 ms., SUp:  1.6326
Met:1, Para:   3956.23 ms; Sekv:   33.987 ms., SUp:  0.0086
Met:2, Para:   3094.19 ms; Sekv:   33.987 ms., SUp:  0.0110
Met:3, Para:     33.31 ms; Sekv:   33.987 ms., SUp:  1.0203
Met:4, Para:     21.40 ms; Sekv:   33.987 ms., SUp:  1.5881
Met:5, Para:     21.00 ms; Sekv:   33.987 ms., SUp:  1.6184
```

II) Oblig 2: forrige og denne uka, matrisemultiplisering

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

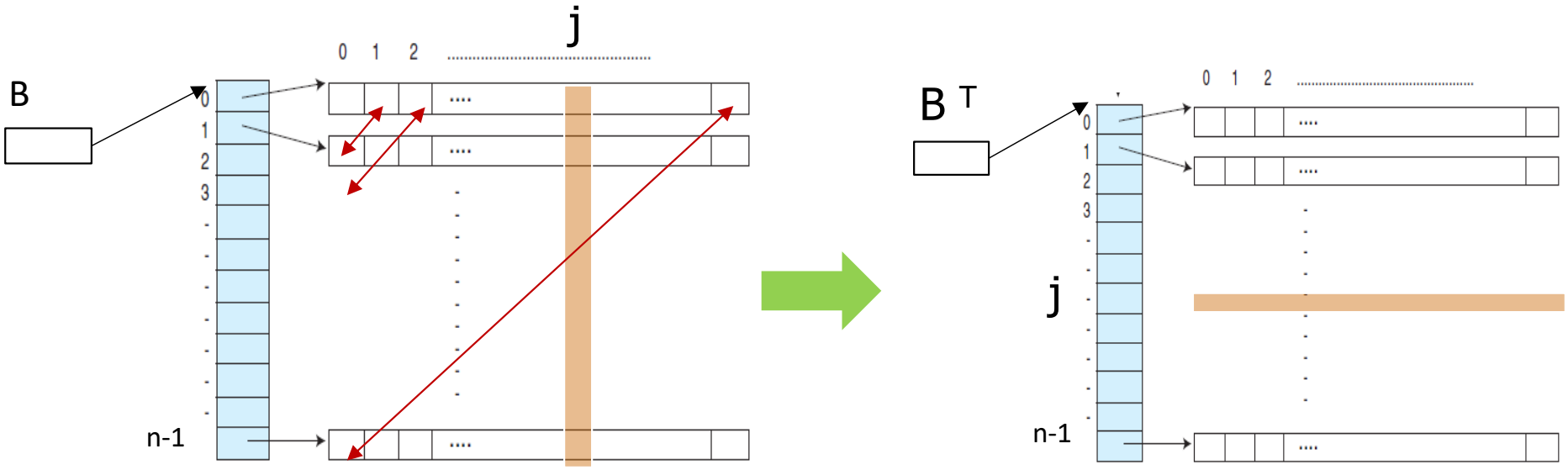
$$c[i][j] = \sum_{k=0}^{n-1} a[i][k] * b[k][j]$$

$C_{i,j} =$



Idé – transponer B (=bytte om rader og kolonner)

- Bytt om elementene i B ($B_{i,j}$ byttes om med $B_{j,i}$)
- Da blir kolonnene lagret radvis.

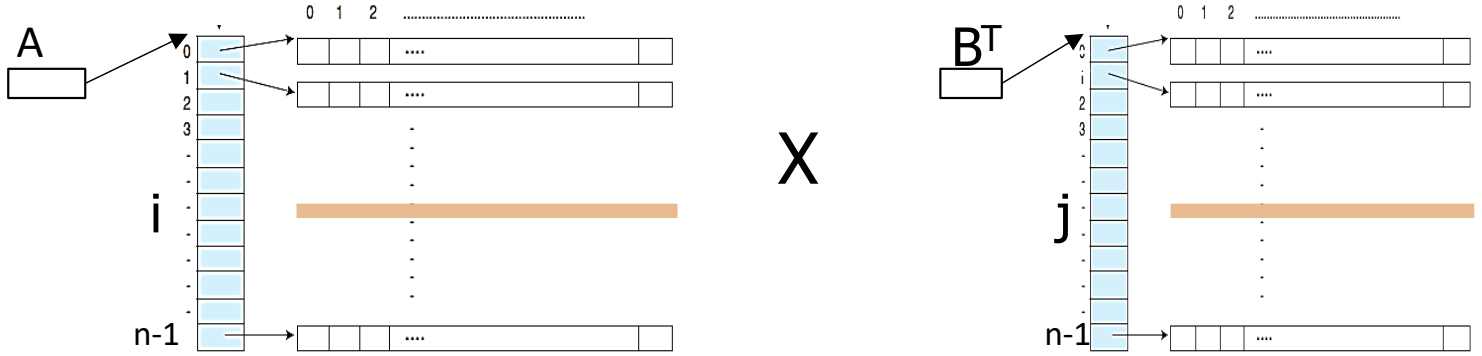


Begrunnelse: Det blir for mange cache-linjer (hver 64 byte) fra B i cachene

Vi har da at litt ny formel for C

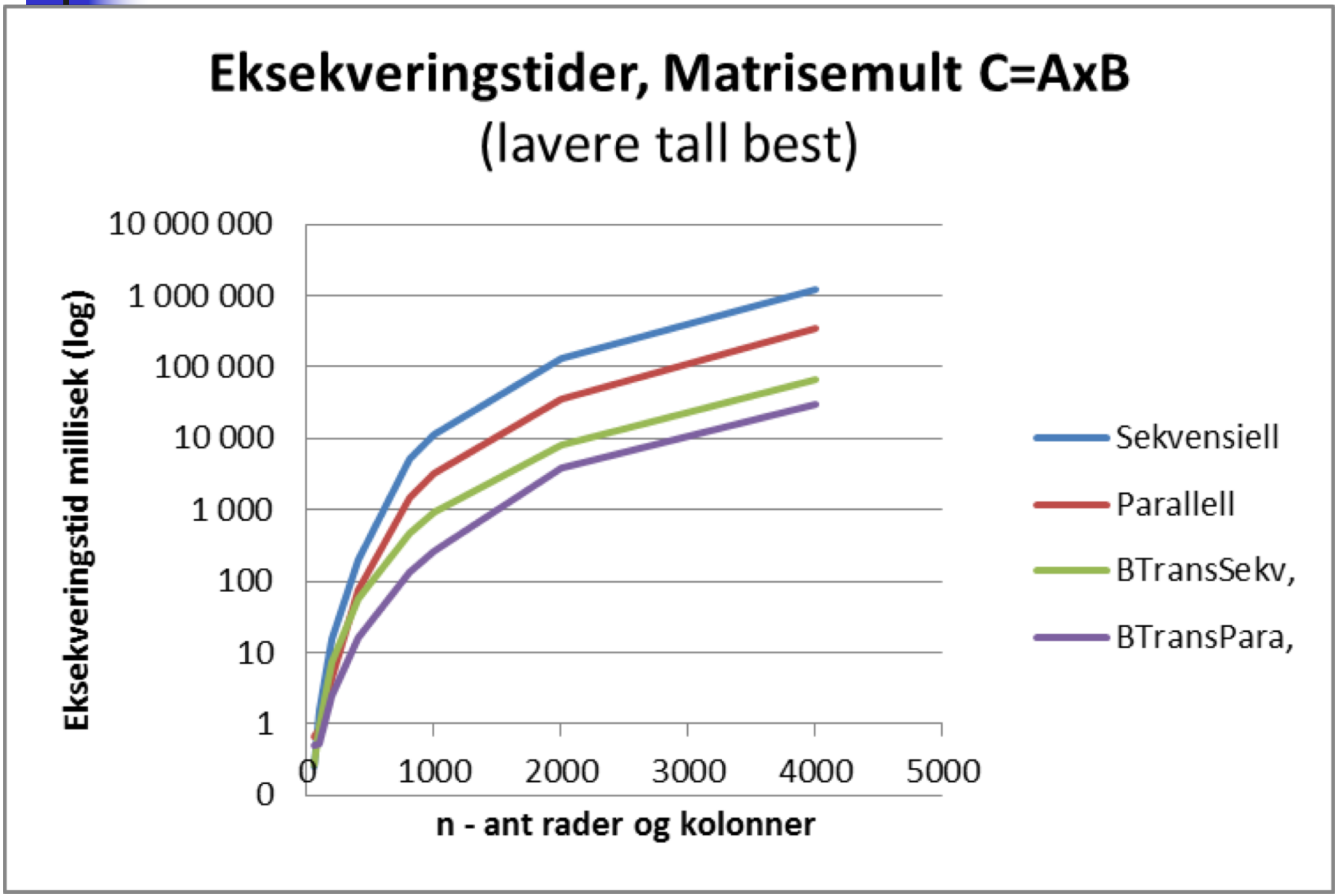
$$c[i][j] = \sum_{k=0}^{n-1} a[i][k] * b^T[j][k]$$

$C_{i,j} =$

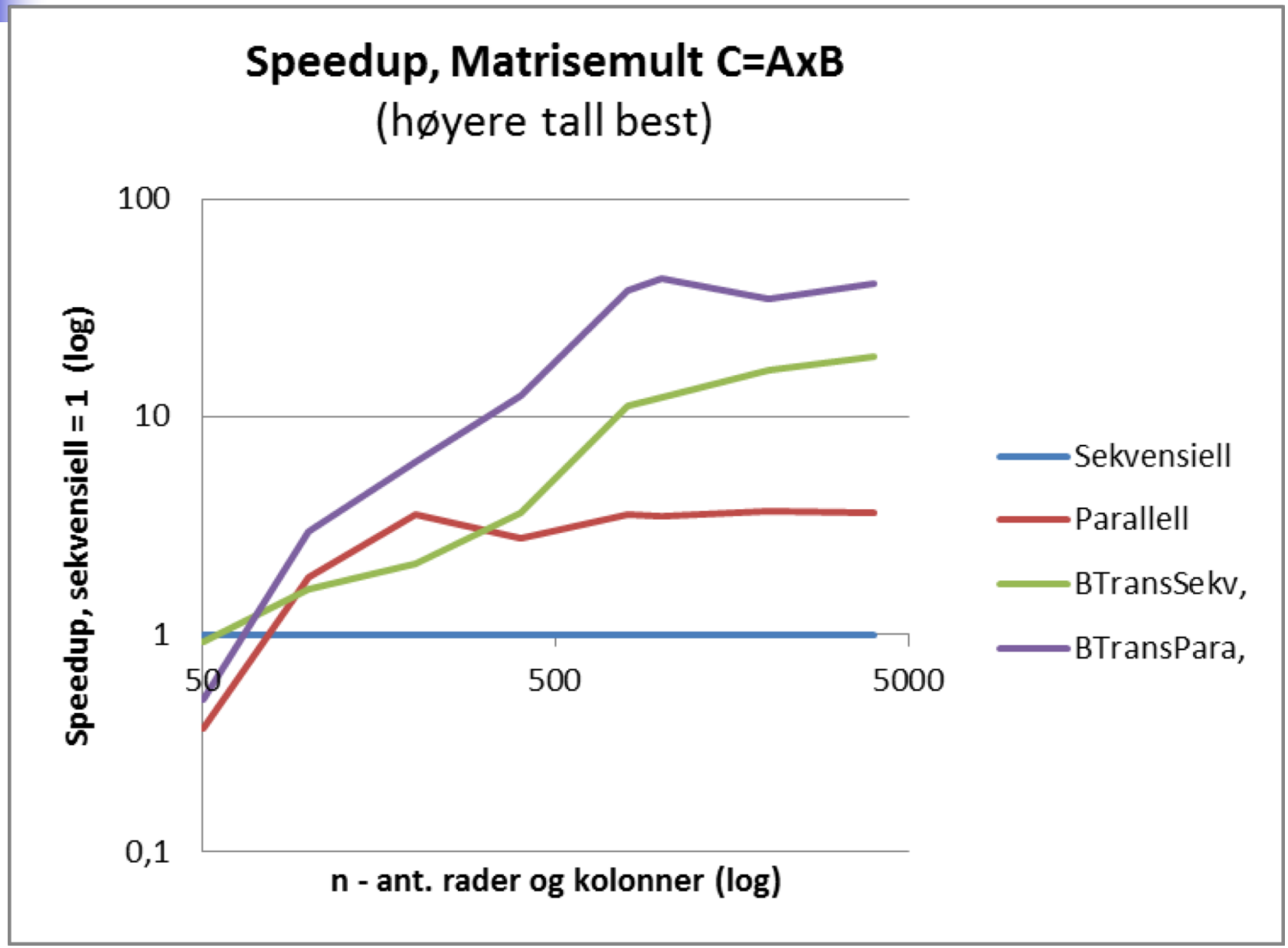


Vi får multiplisert to rader med hverandre – går det fortere ?

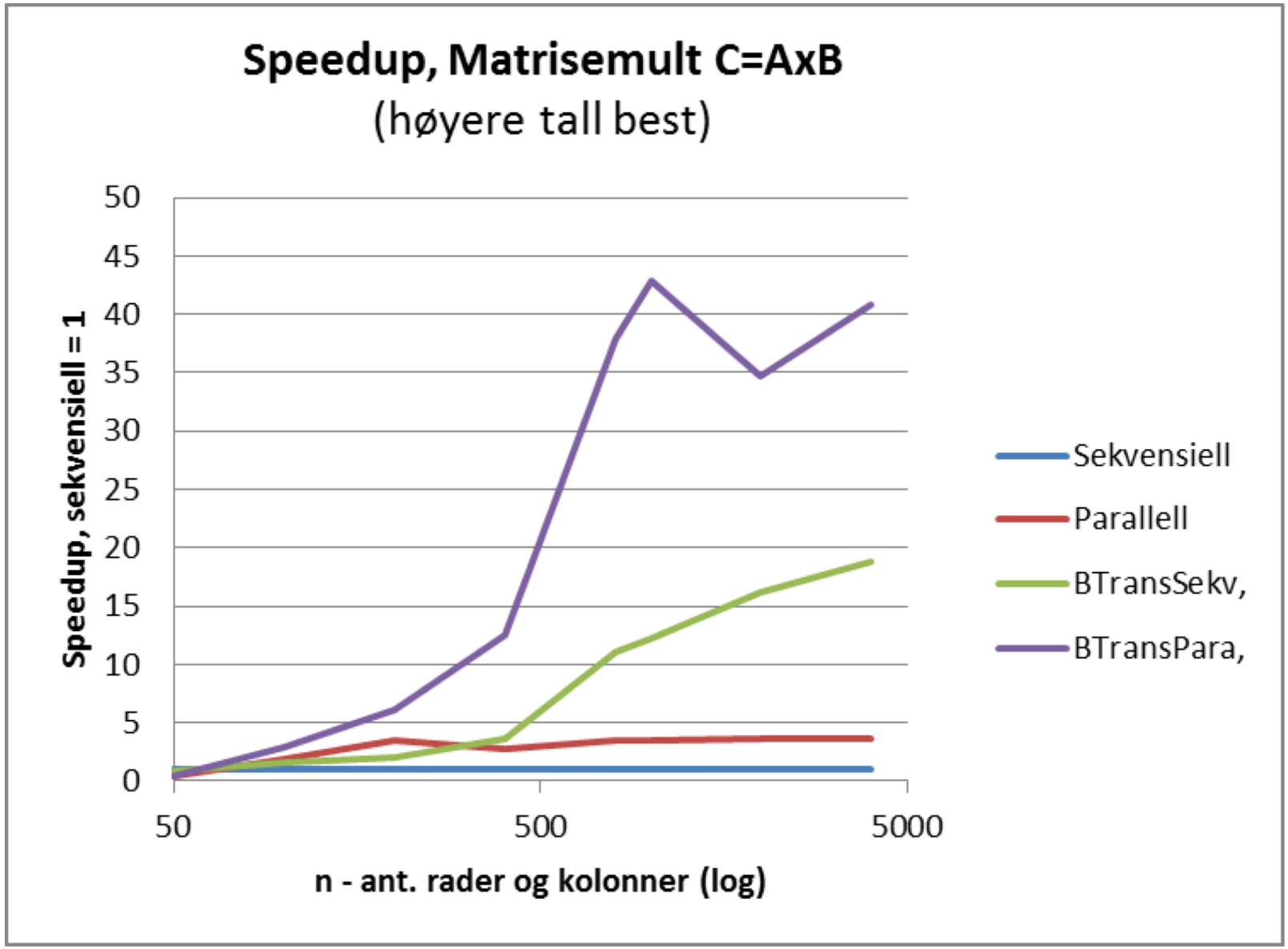
Kjøretider – i millisek. (y-aksen logaritmisk)



Kjøretidsresultater – Speedup , y-aksen logaritmisk



Speedup – med lineær y-akse

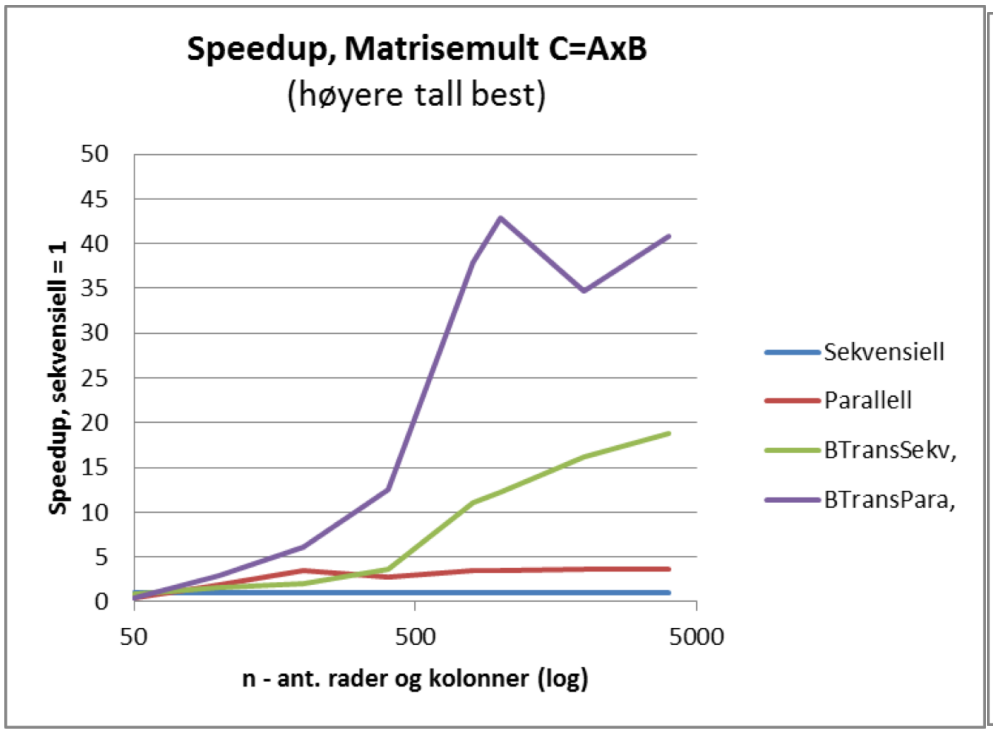


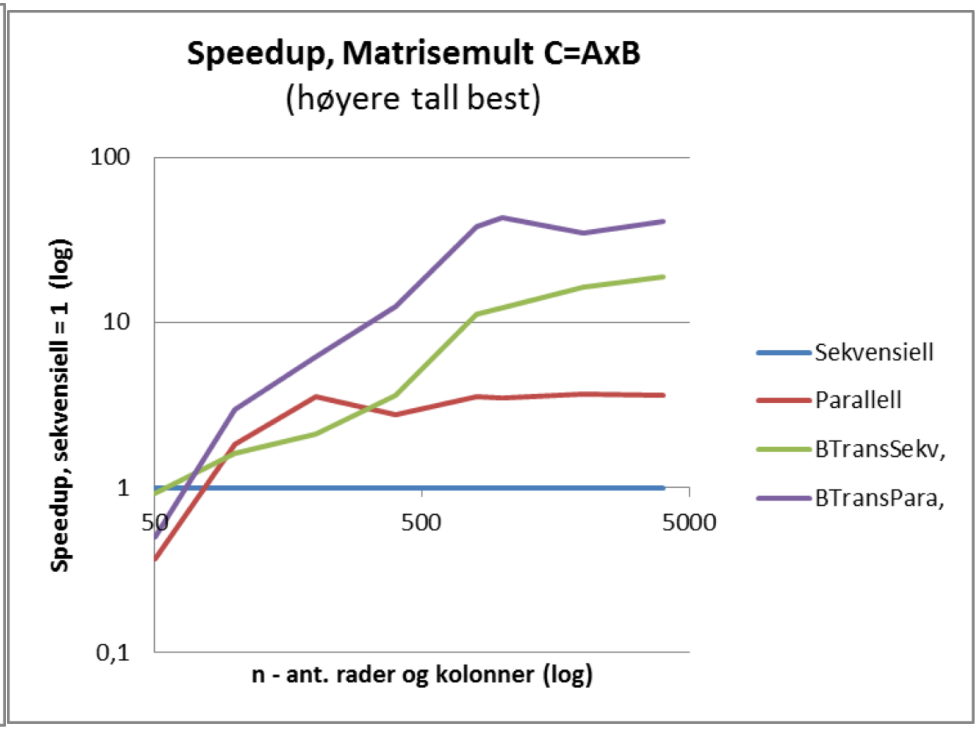
Konklusjon – Matrisemultiplikasjon

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

Speedup – Hvordan fremstille det?

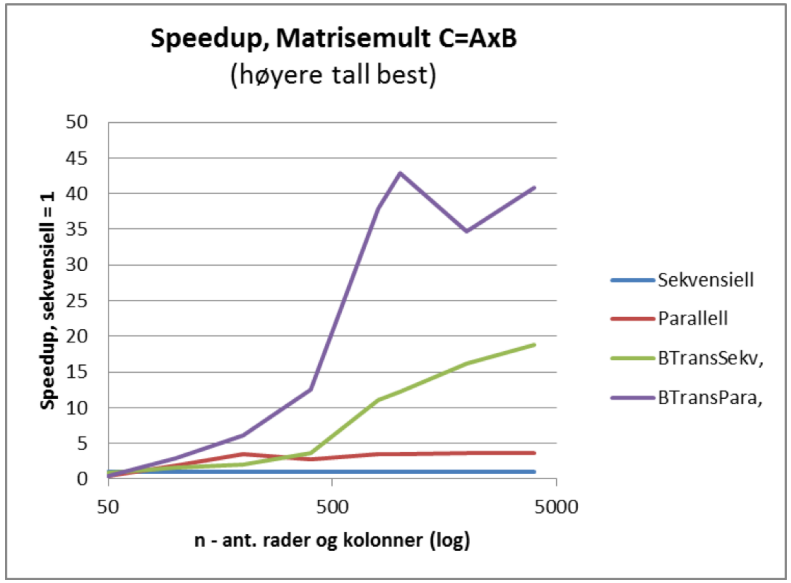
- Disse to kurvene er like ! Hvordan ?

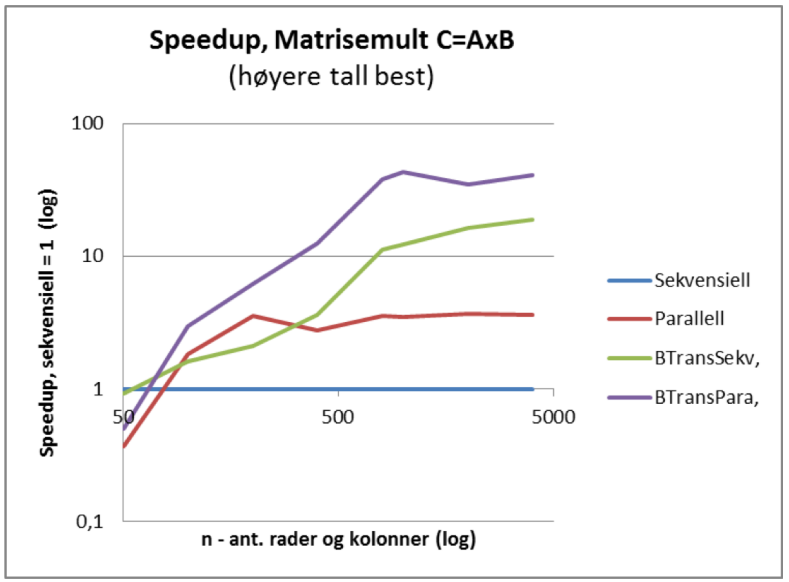


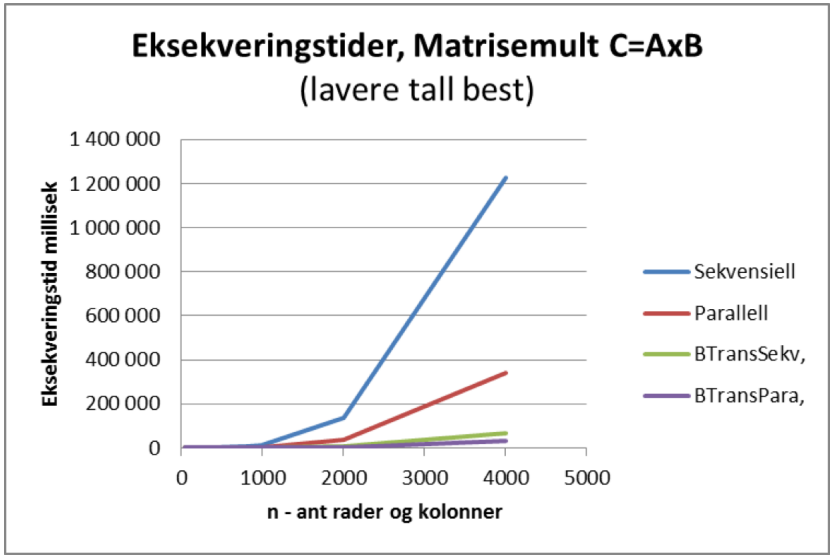


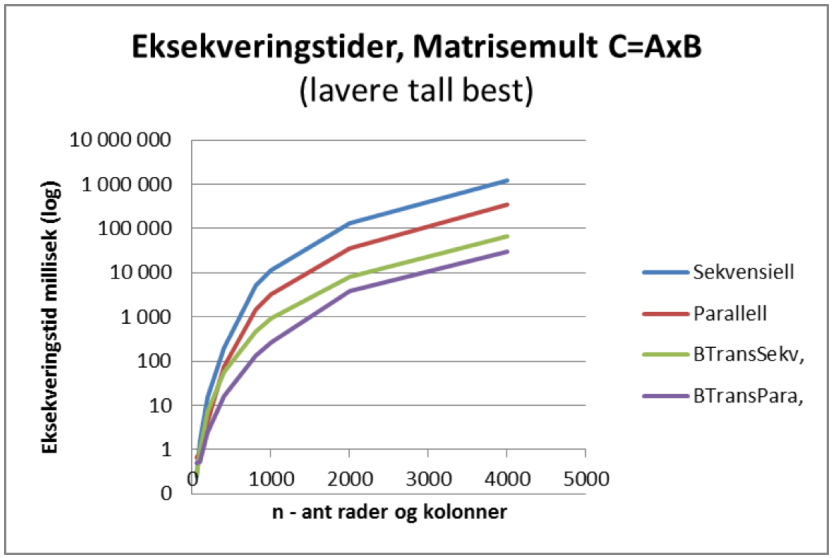
Hvordan framstille ytelse I

- Disse fire kurvene fremviser samme tall! Hvordan ?

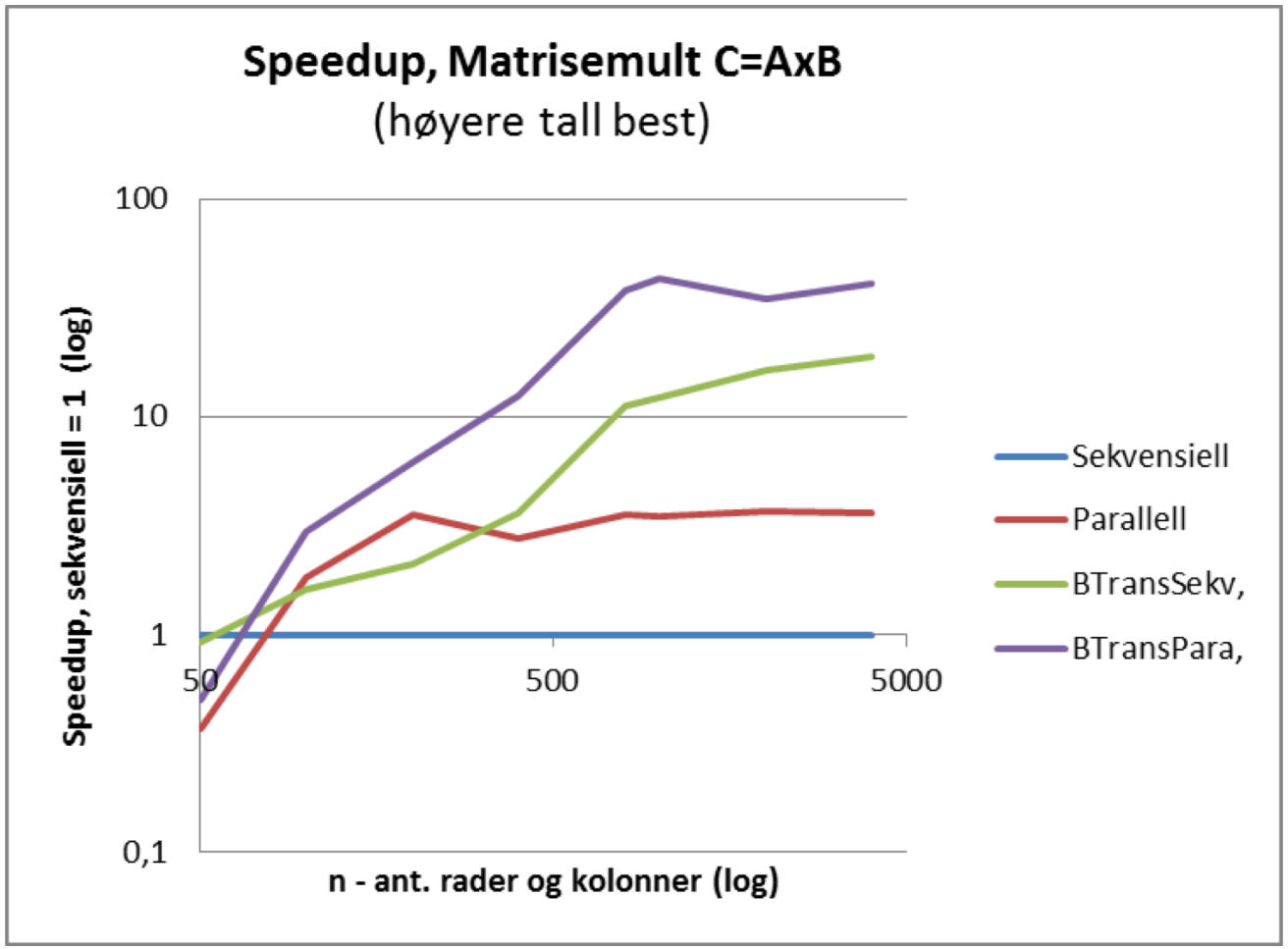




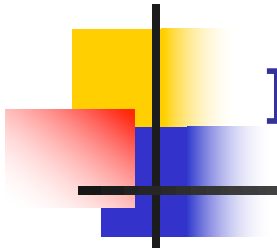




Både logaritmisk x- og y-akse



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



III) PRAM modellen for parallelle beregninger

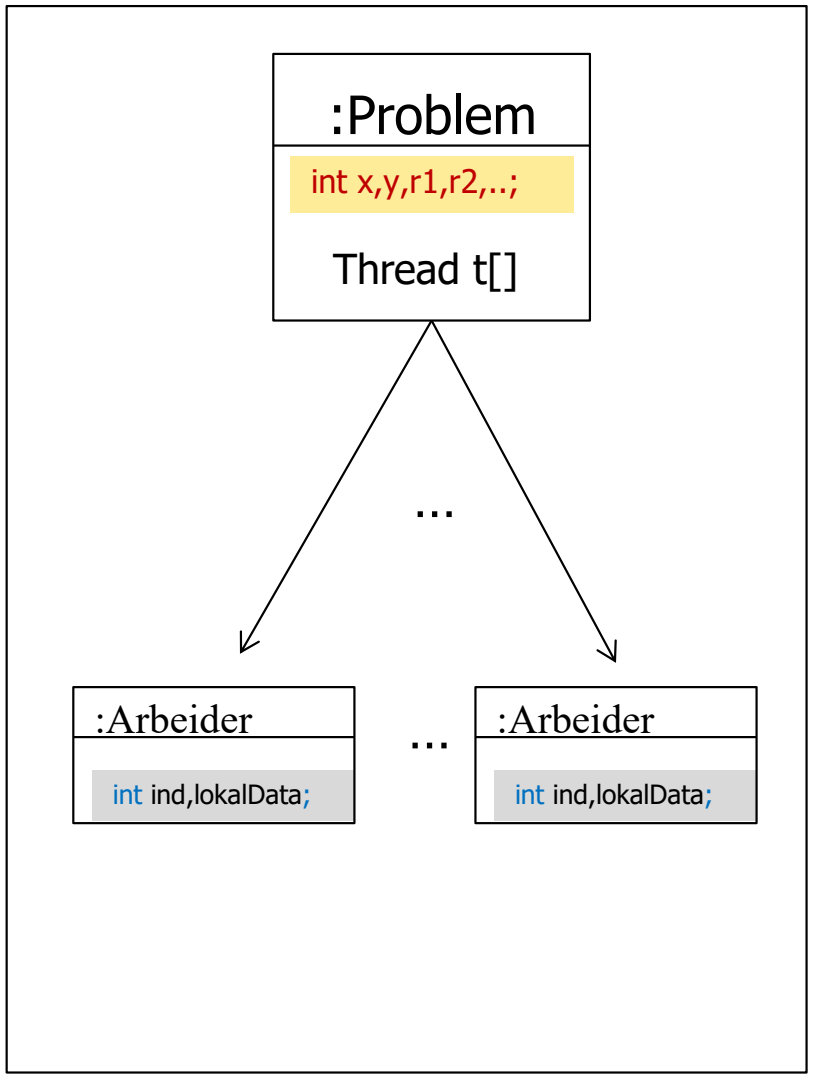
- PRAM (Parallel Random Access Memory) antar to ting:
  - Du har uendelig mange kjerner til beregningene
  - Enhver aksess i lageret tar like lag tid,
    - ignorerer f.eks fordelen med cache-hukommelsen
- Da blir mange algoritmer lette å beregne og programmere
- Problemet er at begge forutsetningene er helt gale.
- Det PRAM gjør er å telle antall instruksjoner utført

  Det har vi sett er helt feilaktig (Radix og Matrise-mult)
- Svært mange kurs og lærebøker er basert på PRAM

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

Noen kommentarer til modell for parallelle programmer

```
import java.util.concurrent.*;
class Problem { int x,y,r1,r2,..; //  felles data
      public static void main(String [] args) {
            Problem p = new Problem();
            p.utfoer(12);
      }
      void utfoer (int antT) {
          Thread [] t = new Thread [antT];
          for (int i =0; i< antT; i++)
              ( t[i] = new Thread(new Arbeider(i))).start();
           for (int i =0; i< antT; i++) t[i].join();
      }

      class Arbeider implements Runnable {
        int ind, lokaleData; // lokale data
          Arbeider (int in) {ind = in;)
              public void run() {
                    // kalles når tråden er startet
              } // end run
      } // end indre klasse Arbeider
} // end class Problem
```



Noen kommentarer til modell for parallelle programmer

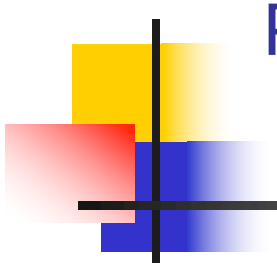
```
import java.util.concurrent.*;
class Problem { int [] a; int antTr=12;//  felles data
      public static void main(String [] args) {
            Problem p = new Problem();
            p.utfoer(antTr);
      }
     void utfoer (int antT) {
          Thread [] t = new Thread [antT];
           a = new int [10000000];  // + fyll a[] med tilfeldige tall
          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, fra,til,num; // lokale data
           Arbeider (int in) {ind = in; num= a.length/antTr;
                       fra =ind*num; til = (ind+1)*num;
                       if ( ind == antTr -1) til = a.length;
            }
             public void run() {
                    // kalles når tråden er startet
           } // end run
     } // end indre klasse Arbeider
} // end class Problem
```

IV) Utsatte operasjoner i JIT-kompilering

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

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

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

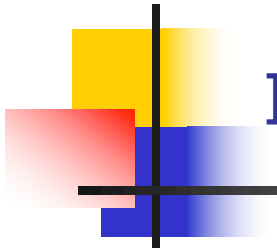


Repetisjon: 3 viktigste regler om lesing og skriving på felles data.

- Før (og etter) synkronisering på felles synkroniseringsvariabel gjelder :
  A. Hvis ingen tråder skriver på en felles variabel, kan alle tråder lese denne.
  B. To tråder må aldri skrive samtidig på en felles variabel (eks. i++)
  C. Hvis derimot én tråd skriver på en variabel må *bare* den tråden lese denne variabelen før synkronisering – ingen andre tråder må lese den før synkronisering.

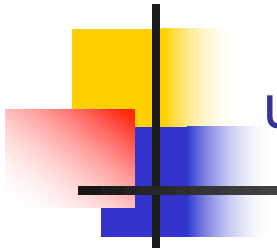
Muligens ikke helt tidsoptimalt, men enkel å følge – gjør det mulig/mye enklere å skrive parallelle programmer.

Regel C er i hovedsak 'problemet' i Java Memory Model



IV) Hva er en memory-modell og hvorfor trenger vi en!

- Vi har hittil sett at med flere tråder så:
  - Instruksjoner kan bli utsatt og byttet om
    - av CPU, kompilator, cache-systemet
  - Ulike tråder kan samtidig se ulike verdier på felles variabel
  - Programmet du skrev er optimalisert og har liten direkte likhet med det som eksekverer
- Vi må likevel ha en viss orden på (visse steder i eksekveringen):
  - På rekkefølgen av instruksjonene
  - Synlighet - når ser én tråd det en annen har skrevet ?
    - Maskinvare og hukommelse hvor alle-ser-det samme-alltid tar for lang tid
  - Atomære operasjoner
    - Når en operasjon først utføres, skal hele utføres uten at andre operasjoner blander seg inn og ødelegger (som i++)
- Ulike maskinvare-fabrikanter kan ulike memory-modeller, men Intel/AMD modellen er rask og understøttes av Java.



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

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

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

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

Initially, x == y == 0

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

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

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

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

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

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

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

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

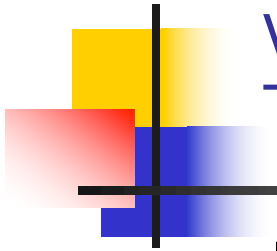
Initially, x == y == 0

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

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

Figure 12: Behavior that must be allowed

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



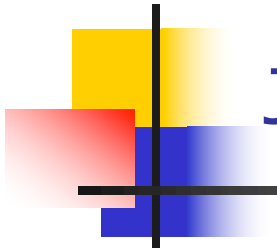
V) Effekten av synkronisering på samme synkroniseringsvariabel og The Java memory model (JMM) - pensum fortsetter

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

- Har den andre skrevet enda og har det kommet ned i lageret?

Svar: Vi lar begge (alle) trådene **synkronisere** på samme synkroniseringsvariabel (en CyclicBarrier eller en Semaphore,…):

- Da skjer følgende før noen fortsetter:
  1. Alle felles variable blir skrevet ned fra cachene og ned i lageret.
  2. Alle operasjoner som er utsatt, blir utført før noen av trådene slipper gjennom synkroniseringen.
  3. Følgelig ser alle trådene de samme verdiene på alle felles variable når de fortsetter etter synkroniseringa!



JMM og volatile variable – del 1

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

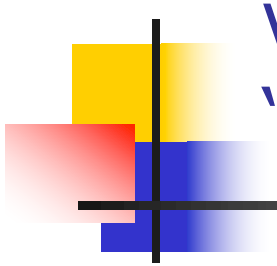
JMM og volatile variable – del 2

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

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

             if (! stop) {

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

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

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

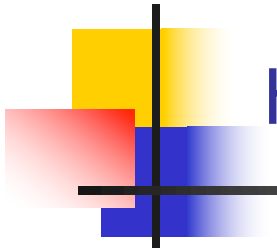
I ytre felles klasse:

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



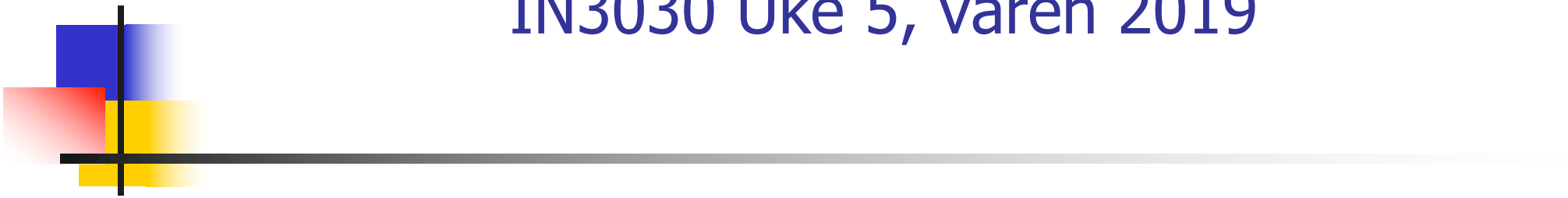
V) Synkronisering på samme synkroniseringsobjekt løser 'alle' problemer - løser synlighet og utsatte operasjoner.

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



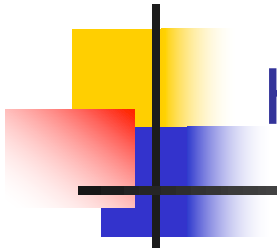
Hva her vi sett på i Uke4

I. Om å avlutte k tråder med å samle svarene fra disse.
   1 gal + 5 riktige måter + bedre for-løkke betyr mye!
II. Kommentarer om matrise-multiplikasjon
III. Vi bruker **ikke** PRAM modellen for parallelle beregninger
   I. Hva er PRAM og hvorfor er den ubrukelig for oss
IV. Å lage små metoder gir bedre, raskere JIT kompilering
V. Hva skjer egentlig i lageret (main memory) når vi kjører parallelle tråder - the Java Memory Model
VI. Hvorfor synkroniserer vi, og hva skjer da ?



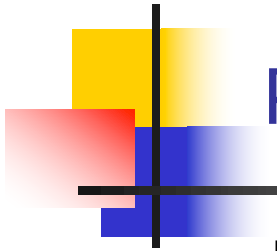
IN3030 Uke 5, våren 2019

Eric Jul
PSE
Inst. for informatikk



Hva så vi på i Uke4

1. Kommentarer om matrise-multiplikasjon
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
4. Hvorfor synkroniserer vi, og hva skjer da,
    1. Hvilke problemer blir løst ?



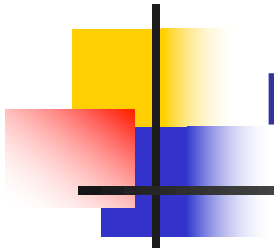
Plan for uke 5

Første time:

1. Oblig 1: comments
2. Oblig 2: Matrix multiplication
3. Modellkode2 -forslag for testing av parallell kode
4. Ulike løsninger på i++
5. Vranglås - et problem vi lett kan få (og unngå)

Annen time:

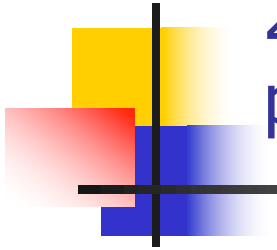
1. Ulike strategier for å dele opp et problem for parallellisering:
2. Om primtall – Eratosthenes Sil (ES)
3. Hvordan representere (ES) effektivt i maskinen



Reasons to fail oblig 1

Reasons to fail oblig 1:

- - Lacking tables in the report
- - Lacking explanation in the report
- - Lacking diagrams in the report
- - Not thread safe code
- - Too much synchronization

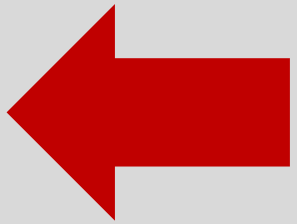


2) Modell-kode for tidssammenligning av (enkle) parallelle og sekvensiell algoritmer

- En god del av dere har laget programmer som virker for:
  - Kjøre både den sekvensielle og parallelle algoritmen
  - Greier å kjøre begge algoritmene 'mange' ganger for å ta mediantiden for sekvensiell og parallell versjon
  - Helst skriver resultatene ut på en fil for senere rapport-skriving
  - Dere kan slappe av nå, og se på min løsning
- For dere andre skal jeg gjennomgå min kode slik at dere har et skjelett å skrive kode innenfor
  - Det mest interessante i dette kurset er tross alt hvordan vi:
    - Deler opp problemet for parallellisering
    - Hvordan vi synkroniserer i en korrekt parallell løsning.

```
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import easyIO.*;
// file: Modell2.java
// Lagt ut  feb 2017 - Arne Maus, Ifi, UiO
// Som BARE et eksempel, er problemet med å øke fellesvariabelen i  n*antKjerner  ganger løst

class Modell2{// ****** Problemets FELLES DATA HER
              int i;
              final String navn = "TEST AV i++ med synchronized oppdatering";
               // Felles system-variable - samme for 'alle' programmer
               CyclicBarrier vent,ferdig, heltferdig ; // for at trådene og main venter på hverandre
               int antTraader;
               int antKjerner;
               int numIter ;          // antall ganger for å lage median (1,3,5,,)
               int nLow,nStep,nHigh;  // laveste, multiplikator, hoyeste n-verdi
               int n;                 // problemets størrelse
               String filnavn;
               volatile boolean stop = false;
               int med;
               Out ut;
                int [] allI;

               double [] seqTime ;
               double [] parTime ;
```

```
/** for også utskrift på fil */
synchronized void println(String s) {
        ut.outln(s);
        System.out.println(s);
 }

 /** for også utskrift på fil */
 synchronized void print(String s) {
         ut.out(s);
         System.out.print(s);
  }
  /** initieringen i main-tråden */
  void intitier(String args) {
        nLow = Integer.parseInt(args[0]);
        nStep = Integer.parseInt(args[1]);
        nHigh = Integer.parseInt(args[2]);
        numIter = Integer.parseInt(args[3]);
        seqTime = new double [numIter];
        parTime = new double [numIter];
        ut = new Out(args[4], true);

         antKjerner = Runtime.getRuntime().availableProcessors();
         antTraader = antKjerner;
         vent   = new CyclicBarrier(antTraader+1); //+1, også main
         ferdig = new CyclicBarrier(antTraader+1); //+1, også main
          heltferdig = new CyclicBarrier (2);       // main venter på tråd 0
          allI = new int [antTraader];
```

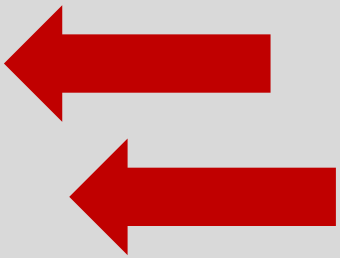


```
                // start trådene
                 for (int i = 0; i< antTraader; i++)
                        new Thread(new Para(i)).start();

     } // end initier

public static void main (String [] args) {
   if ( args.length != 5) {
     System.out.println("use: >java Modell2 <nLow> <nStep> <nHigh> <num iter> <fil>");
   } else {
                new Modell2().utforTest(args);
   }
} // end main
```

```
void  utforTest () {
        intitier();
        println("Test av  "+ navn+ "\n med "+
        antKjerner + " kjerner , og " + antTraader+" traader,  Median av:" + numIter+" iterasjon\n");
        println("\n     n      sekv.tid(ms)   para.tid(ms)    Speedup ");

        for (n = nHigh; n >= nLow; n=n/nStep) {
              for (med = 0; med < numIter; med++) {
                        long t = System.nanoTime();  // start tidtagning parallell
                        // Start alle trådene parallell beregning nå
                          try {
                             vent.await();   // start en parallell beregning
                             ferdig.await(); // vent på at trådene er ferdige
                          } catch (Exception e) {return;}
                          try {   heltferdig.await(); // vent på at tråd 0 har summert svaret
                          } catch (Exception e) {return;}

                           // her kan vi lese svaret

                          t = (System.nanoTime()-t);
                          parTime[med] =t/1000000.0;
                           println(« svaret er:» + i + «for n =» +n);

                           t = System.nanoTime(); // start tidtagning sekvensiell
                           //**** KALL PÅ DIN SEKVENSIELLE METODE  H E R ********
                           sekvensiellMetode (n,numIter);
                           t = (System.nanoTime()-t);
                           seqTime[med] =t/1000000.0;
             } // end for med
```



```
            println(Format.align(n,10)+
                    Format.align(median(seqTime,numIter),12,3)+
                    Format.align(median(parTime,numIter),15,3)+
                    Format.align(median(seqTime,numIter)/median(parTime,numIter),13,4));
        } // end n-loop
       exit();
 } // utforTest

/** terminate parallel threads*/
void exit() {
        stop = true;
        try { // start the other threads and they terminate
                    vent.await();
        } catch (Exception e) {return;}
        ut.close();
 }   // end exit

 /*** HER er din egen sekvensielle metode som  selvsagt IKKE ER synchronized, */
 void sekvensiellMetode (int n,int numIter){
         for (int j=0; j<n; j++){
                        i++;
         }
} // end sekvensiellMetode

/*** Her er evt. de parallelle metodene som ER synchronized  - treig*/
 synchronized void addI() {
         i++;
 }
```



```
class Para implements Runnable{
            int ind, minI=0, fra,til,num;
            Para(int i) { ind =i; } // konstruktor

    /*** HER er dine egen parallelle metoder som IKKE er synchronized */
    void  parallellMetode(int ind) {
          for (int j=0; j<n; j++){
                        minI++;
          }
        allI [ind] = minI;
    }

    void paraInitier(int n) {
          num = n/antTraader;
          fra = ind*num;
          til = (ind+1)*num;
          if (ind == antTraader-1) til =n;
          minI =0;
    }// end paraInitier
```

```
public void run() { // Her er det som kjores i parallell:
        while (! stop) {
            try {  // wait on all other threads + main
                vent.await();
             } catch (Exception e) {return;}
            if (! stop) {
                paraInitier(n);
                //**** KALL PÅ DINE PARALLELLE METODER  H E R ********
                  parallellMetode(ind); // parameter: traanummeret: ind

                  try{ // make all threads terminate
                        ferdig.await();
                  } catch (Exception e) {}
            } // end ! stop thread

              // tråd nr 0 adderer de 'numThreads' minI - variablene til en felles verdi
          if (ind == 0) { i =0;
              for (int j = 0; j < antTraader; j++) { i += allI[j]; }

              try {   heltferdig.await();  // si fra til main at tråd 0 har summert svaret
              } catch (Exception e) {return;}

          } // end tråd 0
      } // end while !stop
   } // end run

} // end class Para
```

Hvor lang tid tar et synchronized kall? Demoeks. hadde n synchroniced metode for all skriving til felles 'i'.

- Kjørte modell-koden for n=10 000 000 (3 ganger)

```
M:\INF2440Para\ModelKode>java Modell2 100 10 10000000 3 test-14feb.txt
Test av  TEST AV i++ med synchronized oppdatering
 med 8 kjerner , og 8 traader,  Median av:3 iterasjoner


    n       sekv.tid(ms)   para tid(ms)    Speedup
 10000000         6.704      11024.957       0.0006
  1000000         0.658       1084.411       0.0006
   100000         0.071         98.566       0.0007
    10000         0.007         10.927       0.0006
     1000         0.001          1.057       0.0010
      100         0.000          0.192       0.0018
```

- Svar: Et synchronized kall tar ca. 1000/(8*1000 000)ms = 0.15 μs = 150ns. = ca. 500 instruksjoner.

3) Finnes det alternativer & riktig kode?

- a) Bruk av ReentrantLock (import java.util.concurrent.locks.*;)

```
// i felledata-omraadet i omsluttende klasse
ReentrantLock laas = new ReentrantLock();
          ..........
 /*** HER skriver du eventuelle  parallelle metoder som ER synchronized */
  void addI() {
        laas.lock();
              i++;
        try{ laas.unlock();} catch(Exception e) {return;}
 } // end addI
```

- Kjøring:

```
M:\INF2440Para\ModelKode>java ModellAlt 1000000 5 test.txt
Test av  TEST AV i++ med ReentrantLock oppdatering
 med 8 kjerner , og 8 traader

Median of 5:  Sekv. tid:        0.70 ms, Para tid:       212.44 ms,
 Speedup:  0.003, n = 1000000
```

- 5x fortere enn synchronized !

b) Alternativ b til synchronized: Bruk av AtomicInteger

- Bruk av AtomicInteger (import java.util.concurrent.atomic.*;)

```
 // i felledata-omraadet i omsluttende klasse
AtomicInteger i = new AtomicInteger();
        .............
/*** HER skriver du eventuelle  parallelle metoder som ER synchronized */
 void addI() {
     i.incrementAndGet();
 } // end addI
```

- Kjøring:

```
M:\INF2440Para\ModelKode>java ModellAlt 1000000 5 test.txt
Test av  TEST AV i++ med AtomicInteger oppdatering
 med 8 kjerner , og 8 traader

Median of 5:  Sekv. tid:        0.66 ms, Para tid:       235.91 ms,
 Speedup:  0.003, n = 1000000
```

- **Konklusjon:** Både ReentrantLock og AtomicInteger er 5x fortere enn synchronized metoder + at all parallell kode kan da ligge i den parallelle klassen.

c) : Lokal kopi av i hver tråd og en synchronized oppdatering fra hver tråd til sist.

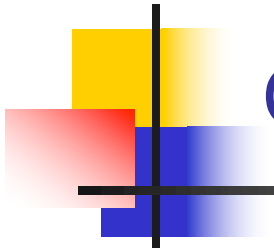
```
/*** HER skriver du eventuelle  parallelle metoder som ER synchronized */
        synchronized void addI(int tillegg) {
                i = i+ tillegg;
        } // end addI
        .......
         class Para implements Runnable{
                int ind;
                int minI=0;
              .....
                /*** HER skriver du parallelle metode som IKKE er synchronized */
               void  parallellMetode(int ind) {
                        for (int j=0; j<n; j++)
                                minI++;
               } // end parallellMeode

               public void run() {
                              ......
                               if (! stop) {
                                //**** KALL PÅ DIN PARALLELLE METODE  H E R ********
                                      parallellMetode(ind);
                                       addI(minI);
                                       try{ ......
```

Kjøring av alternativ C (lokal kopi først):

- Kjøring:

```
M:\INF2440Para\ModelKode>java ModellAlt 1000000 5 test.txt
Test av  TEST AV i++ forst i lokal i i hver traad, saa synchronized
oppdatering av i,  med 8 kjerner , og 8 traader

Median of 5:  Sekv. tid:        0.71 ms, Para tid:        0.47 ms,
 Speedup:  1.504, n = 1000000
```

- Betydelig raskere, ca. 500x enn alle de andre korrekte løsningene og noe raskere enn den sekvensielle løsningen
- Eneste riktige løsning som har speedup > 1.
- **Husk:** Ingen vits å lage en parallell algoritme hvis den sekvensielle er raskere.



Oppsummering

Løsning	kjøretid	Speedup
Sekvensiell	0,70 ms	1
Bare synchronized	1015,72 ms	0,001
ReentrantLock	212.44 ms	0,003
AtomicInteger	235,91 ms	0,003
Lokal kopi, så synchronized oppdatering 1 gang per tråd	0,47 ms	1,504

- Oppsummering:
  - Synkronisering av skriving på felles variable tar lang tid, og må minimeres (og synchronized er spesielt treg)
  - Selv den raskeste er 500x langsommere enn å ha lokal kopi av fellesvariabel int i , og så addere svarene til sist.

4) Vranglås: Rekkefølgen av flere synkroniseringer fra flere, ulike tråder – går det alltid bra?

- Anta at du har to ulike parallelle klasser A og B og at som begge bruker to felles synkroniseringsvariable: Semphorene 'vent' og 'ferdig' (begge initiert til 1).
- A og B synkroniserer seg *ikke* i samme rekkefølge:

```
A sier:
try{
   vent. acquire();
   ferdig. acquire();
} catch(Exception e)
{return;}

…gjør noe….

ferdig.release();
vent.release();
```

```
B sier:
try{
     ferdig. acquire();
     vent. acquire();
} catch(Exception e)
{return;}

…gjør noe….

vent.release();
ferdig.release();
```

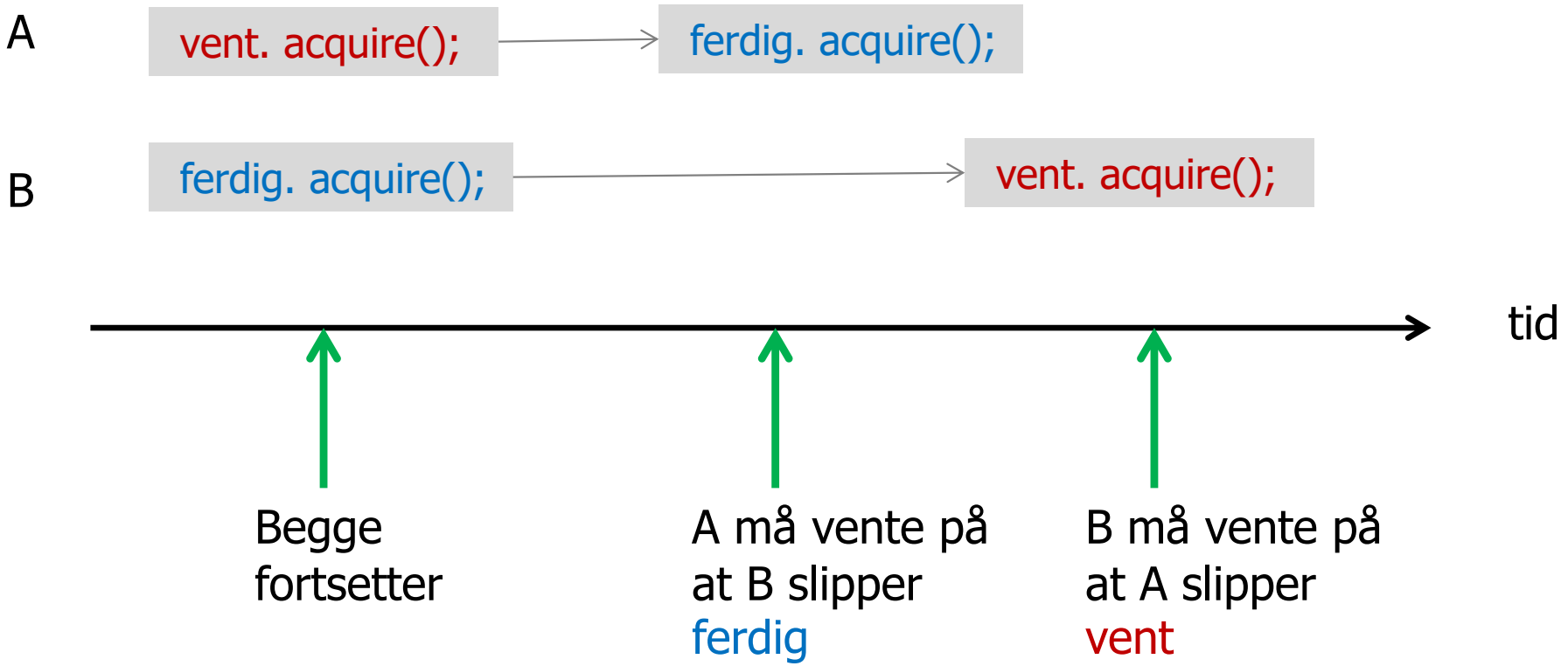
Ytre klasse VrangLaas med to indre klasser SkrivA og SkrivB

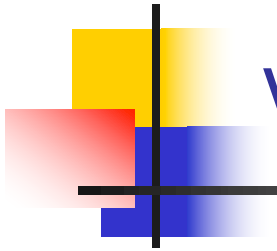
```
public class VrangLaas{
    int  a=0,b=0, antGanger;              // Felles variable a,b
    Semaphore ferdig, vent ;
    SkrivA aObj;
    SkrivB bObj;

     public static void main (String [] args) {
          if (args.length != 1) {
                System.out.println(" bruk: java <ant ganger oeke> );
          } else {
               int antKjerner = Runtime.getRuntime().
                                availableProcessors();
                System.out.println("Maskinen har "+ antKjerner +
                  " prosessorkjerner.\n");
                 VrangLaas p =  new VrangLaas();
                 p.antGanger  = Integer.parseInt(args[0]);
                 p.utfor();
           }
        } // end main

        void  utfor () {
              vent   = new Semaphore(1);
              ferdig = new Semaphore(1);
               (aObj =  new SkrivA()).start();
               (bObj =  new SkrivB()).start();
          } // utfor
```

```
class SkrivA extends Thread{
        public void run() {
            for (int j = 0; j<antGanger; j++) {
                    try {  // wait
                          vent.acquire();
                          ferdig.acquire();
                    } catch (Exception e) {return;}

                    a++;
                    System.out.println(" a: " +a);
                     vent.release();
                     ferdig.release();
            } // end j
      } // end run A
} // end class SkrivA

class SkrivB extends Thread{
        public void run() {
            for (int j = 0; j<antGanger; j++) {
                    try {  // wait
                        ferdig.acquire();
                        vent.acquire();
                    } catch (Exception e) {return;}

                    b++;
                    System.out.println(" b: " +b);
                     vent.release();
                     ferdig.release();
            } // end j
      } // end run B
  } // end class SkrivB
} // end class VrangLaas
```

Vranglås – del 2

- Dette kan gi såkalt vranglås (deadlock) ved at begge trådene venter på at den andre skal bli gå videre.
- Hvis operasjonene blandes slik går det galt (og det skjer også i praksis!)

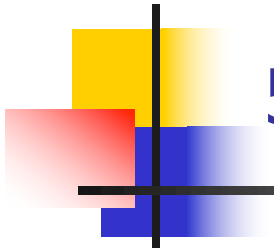




Vranglås - løsning

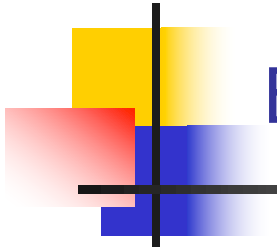
- A og B venter på hverandre til evig tid – programmet ditt henger!
- Løsning: Følg disse enkle regler i hele systemet (fjerner **all** vranglås):

1. Hvis du skal ha flere synkroniserings-objekter i programmet, så må de sorteres i en eller annen rekkefølge.
2. Alle tråder som bruker to eller flere av disse, må be om å få vente på dem (s.acquire(),..) i samme rekkefølge som de er sortert !
3. I hvilken rekkefølge disse synkroniserings-objektene slippes opp (s. release(),..) har mer med hvem av de som venter man vil slippe løs først, og er ikke så nøye; gir ikke vranglås.

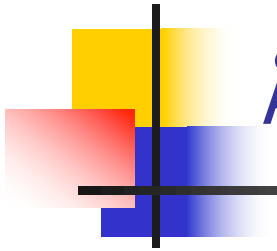


5) Om å parallelliser et problem

- 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, og lar 'hele' koden virke på hver av disse data-delene (en del til hver tråd).
  - 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
- Også mulig å dele opp koden:
  - Alternativ Oblig3 i INF1000: Beregning av Pi (3,1415..) med 17 000 sifre med tre ArcTan-rekker
  - Primtalls-faktorisering av store tall N for kodebrekking:
    - $N = p_1 * p_2$

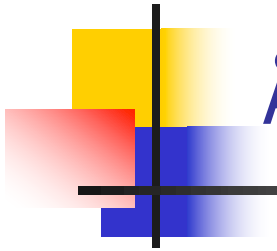


End of First Hour of Lecture



Å dele opp 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 CyclickBarrier-synkronisering mellom hver av disse delene + en synkronisert avslutning (join(), ..).
- Eks:
  - finnMax – hadde ett slikt steg: for (int i = 0 …n-1) -løkke
  - MatriseMult hadde ett slikt steg med trippel-løkke
  - Flere steg mulig: Eksempler senere i kurs (Radix)



Å dele opp data – del 2

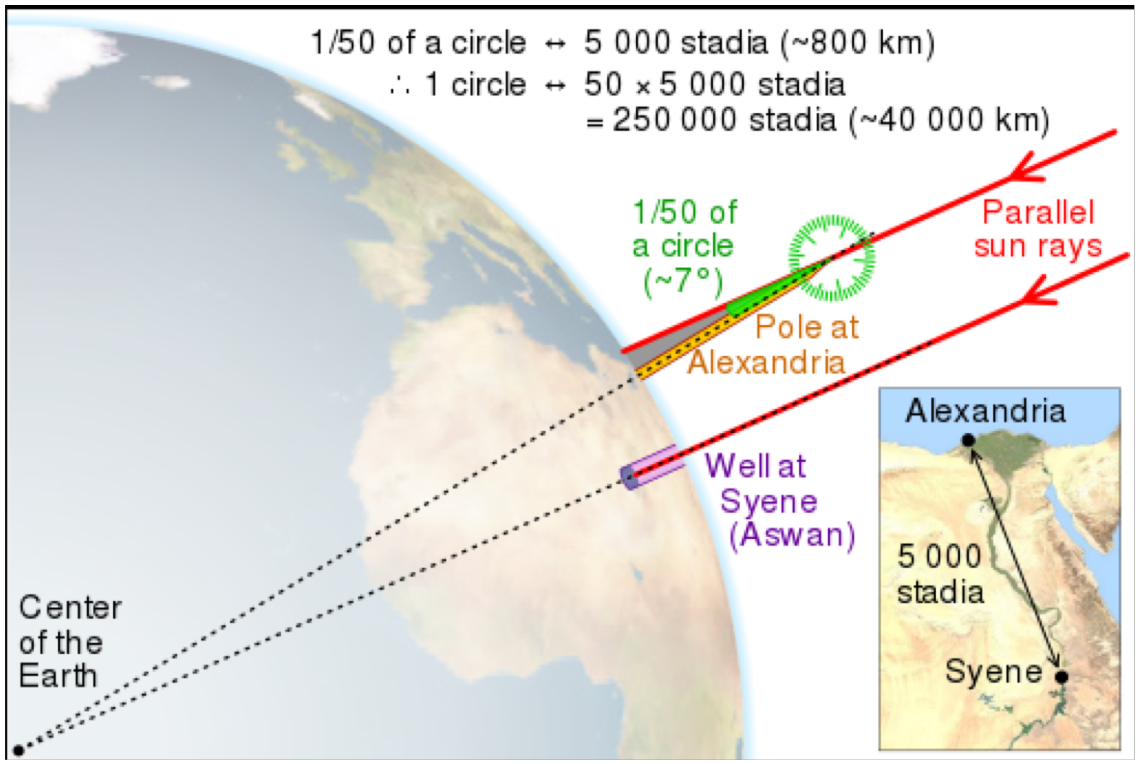
- For å planlegge parallellisering av ett slikt steg må vi finne:
  - Hvilke data i problemet er lokale i hver tråd?
  - Hvilke data i problemet er felles/delt mellom trådene?
- Viktig for effektiv parallell kode.
  - Hvordan deler vi opp felles data (om mulig)
  - Kan hver tråd beregne hver sin egen,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

5) Om primtall – og om Eratosthenes sil (oblig 3)

- 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, og noen mener det fortsatt)
- Ethvert tall N > 1 lar seg faktorisere som et produkt av primtall:
  - $N = p_1 * p_2 * p_3 * \ldots\ldots\ldots * p_k$
  - Denne faktoringen er entydig (pånær rækkefølge); dvs. den eneste faktoriseringen av N – gjøres entydig hvis tall i faktoriseringen sorteres
  - Hvis det bare er ett tall i denne faktoriseringen, er N selv et primtall

Litt mer om Eratosthenes

Eratosthenes, matematikker, laget også et estimat på jordas radius som var < 1,5% feil, grunnla geografi som fag, fant opp skuddårsdagen + at han var sjef for Biblioteket i Alexandria (den tids største forskningsinstitusjon).



2 måter å lage primtall

Ønsker at finne alle primtal $p_i < N$

- Lage en tabell over alle de primtallene vi trenger
  - Eratosthene sil

- Dividere alle tall < N med alle oddetall < $\sqrt{N}$?
  - Divisjonsmetoden

Hvordan lage og lagre primtall

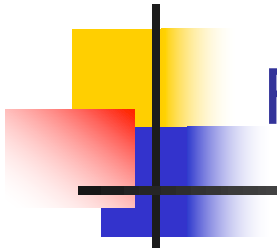
- A) Med Eratosthenes sil:

```
Z:\INF2440Para\Primtall>java PrimtallESil 2000000000
 max primtall m:2000000000
Genererte alle primtall <= 2000000000  paa     18 949 millisek
med Eratosthenes sil  og det største primtallet er:1999999973
```

- Med gjentatte divisjoner

```
Z:\INF2440Para\Primtall>java PrimtallDiv 2000000000
Genererte alle primtall <=2000000000  paa   1 577 302  millisek med
 divisjon , og det største primtallet er:1999999973
```

- Å lage primtallene p og finne dem ved divisjon (del på alle oddetall < SQRT(p) p = 2,3,4,..) er ca. 100 ganger langsommere enn Eratosthenes avkryssings-tabell (kalt Eratosthenes sil).

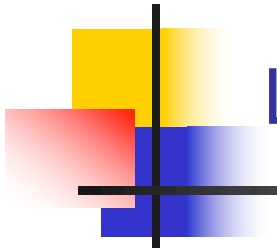


Finne primtall -- Eratosthenes sil

- Hvordan?

Å lage og lagre primtall (Erotosthenes sil)

- Som en bit-tabell (1- betyr primtall, 0-betyr ikke-primtall)
  - Påfunnet i jernalderen 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)



Litt mer om Eratothenes sil

- Vi representerer ikke partallene på den tallinja som det krysses av på fordi vi vet at 2 er et primtall (det første) og at alle andre partall er ikke-primtall.
- Har vi funnet et nytt primtall p, for eksempel. 5, starter vi avkryssingen for dette primtallet først for tallet p*p (i eksempelet: 25), men etter det krysses det av for p*p+2p, p*p+4p,.. (i eksempelet 35,45,55,…osv.). Grunnen til at vi kan starte på p*p er at alle andre tall t < p*p slik det krysses av i for eksempel Wikipedia-artikkelen har allerede blitt krysset av andre primtall < p.
- Det betyr at for å krysse av og finne alle primtall < N , behøver vi bare å krysse av på denne måten for alle primtall p ≤ sqrt(N). Dette sparer svært mye tid.

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,.):

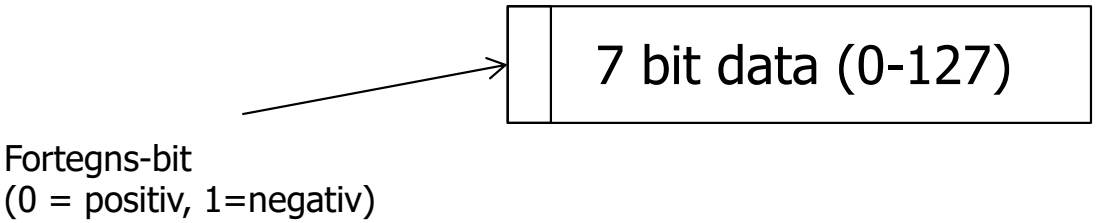
1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

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

1) Hvordan bruke 8 eller 7 bit i en **byte-array** for å representere primtallene

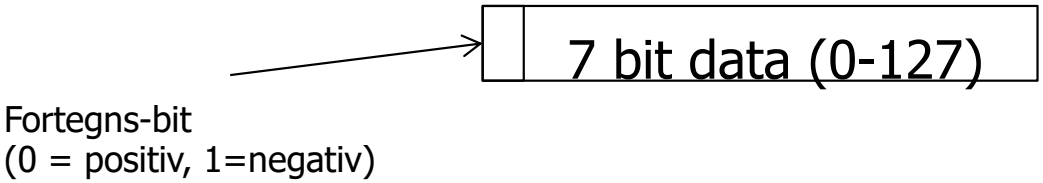
En byte = 8 bit heltall:



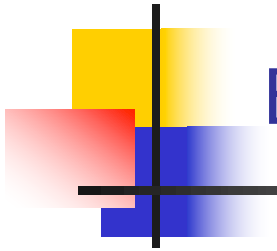
- Vi representer alle oddetallene (1,3,5,,,) som ett bit (0= ikke-primtall, 1 = primtall)
- Bruke alle 8 bit :
  - Fordel: mer kompakt lagring og litt raskere(?) adressering
  - Ulempe: Kan da ikke bruke verdien i byten direkte (f.eks som en indeks til en array), heller ikke +,-,* eller /-operasjonene på verdien
- Bruke 7 bit:
  - Fordel: ingen av ulempene med 8 bit
  - Ulempe: Tar litt større plass og litt langsommere(?) adressering

1) Hvordan representere 8 (eller 7) bit i en byte-array

byte = et 8 bit heltall



- Bruker alle 8 bitene til oddetallene:
  - Anta at vi vil sjekke om tallet ***k*** er et primtall, sjekk først om ***k*** er 2, da ja, hvis det er et partall (men ikke 2) da nei – ellers sjekk så tallets bit i byte-arrayen
    - Byte nummeret til ***k*** i arrayen er da:
      - Enten: ***k***/16, eller: ***k*** >>>4 (shift 4 høyreover uten kopi av fortegns-bitet er det samme som å dele med 16)
    - Bit-nummeret er i denne byten er da enten (***k*** %16)/2 eller (***k*** &15)>>1
  - Hvorfor dele på 16 når det er 8 bit
    - fordi vi fjernet alle partallene – egentlig 16 tall representert i første byten, for byte 0: tallene 0-15
  - Om så å finne bitverdien – se neste lysark.



Bruke 7 bit i hver byte i arrayen

- Anta at vi vil sjekke om tallet ***k*** er et primtall sjekk først om ***k*** er 2, da ja, ellers hvis det er et partall (men ikke 2) da nei – ellers:
- Sjekk da tallets bit i byte-arrayen
  - Byte nummeret til ***k*** i arrayen er da: ***k***/14
  - Bit-nummeret er i denne byten er da: (***k*** %14)/2
- Nå har vi byte nummeret og bit-nummeret i den byten. Vi kan da ta AND (&) med det riktige elementet i en av de to arryene som er oppgitt i skjelett-koden og teste om svaret er 0 eller ikke.
- Hvordan sette alle 7 eller 8 bit == 1 i alle byter)
  - 7 bit: hver byte settes = 127 (men bitet for 1 settes =0)
  - 8 bit: hver byte settes = -1 (men bit for 1 settes = 0)

- Konklusjon: bruk 8 eller 7 bit i hver byte (valgfritt) i Oblig2

2) Faktorisering av et tall M i sine primtallsfaktorer

- Vi har laget og lagret ved hjelp av Erotosthanes sil alle (unntatt 2) 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 ==0, så er p_i en av faktorene til M. Vi forsetter så med å faktorisere ett mindre tall 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 $p_i \leq \sqrt{M}$ som dividerer M med rest == 0, så er M selv et primtall.

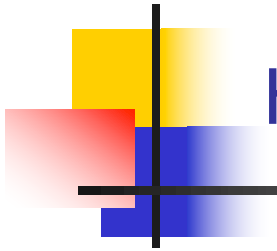
Hvordan parallellisere faktorisering ?

1. Gjennomgås neste uke - denne uka viktig å få på plass en effektiv sekvensiell løsning med om lag disse kjøretidene for N = 2 mill:

```
M:\INF2440Para\Primtall>java PrimtallESil 2000000
 max primtall m:2 000 000
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
```

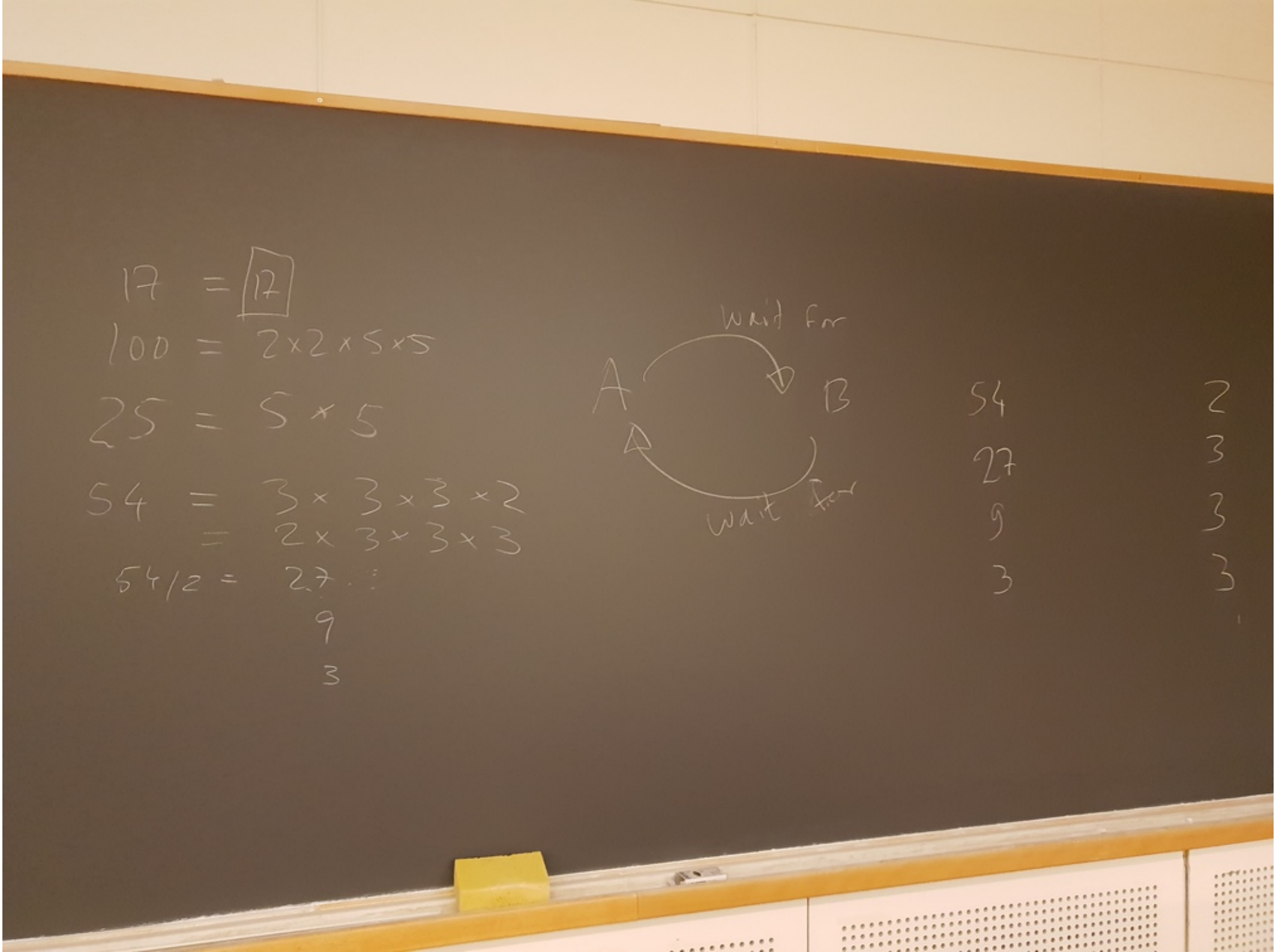
Faktorisering av store tall med 18-19 desimale sifre

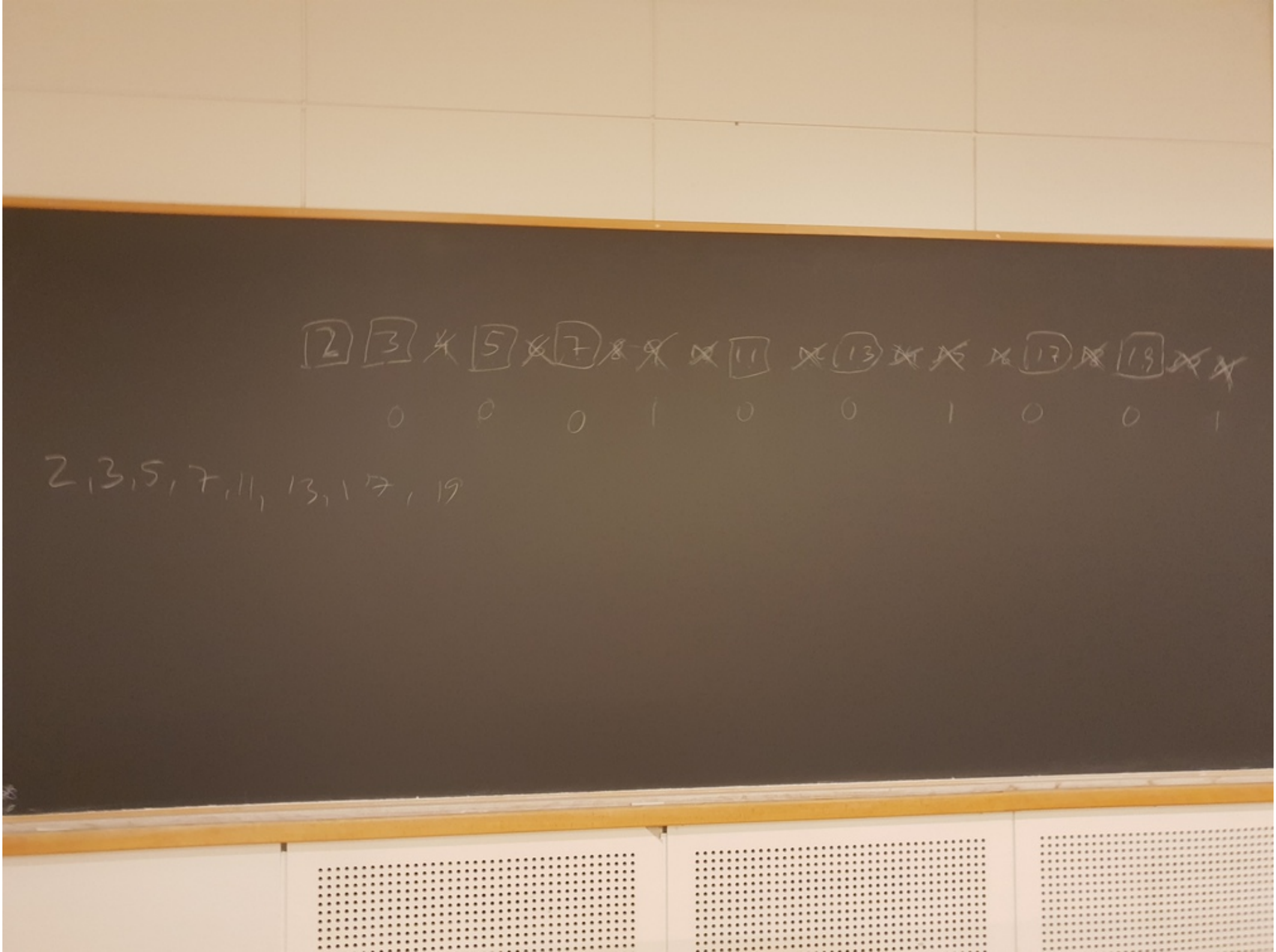
```
Uke5>java PrimtallESil 2140000000
max primtall m:2 140 000 000
bitArr.length:133 750 001
Genererte primtall <= 2 140 000 000  paa    11030.36 millisek
med Eratosthenes sil ( 0.00010530 millisec/primtall)
 antall primtall < 2 140 000 000 er: 104 748 779, dvs:  4.89% ,
 og det største primtallet er: 2 139 999 977

4 579 599 999 999 999 900 = 2*2*3*5*5*967*3673*19421*221303
4 579 599 999 999 999 901 = 457959999999999901
4 579 599 999 999 999 902 = 2*228979999999999951
4 579 599 999 999 999 903 = 3*31*13188589*3733758839
4 579 599 999 999 999 904 = 2*2*2*2*2*19*71*106087842846553
4 579 599 999 999 999 905 = 5*7*1308457142857142 83
.............
4 579 599 999 999 999 997 = 11*41632727272727272 7
4 579 599 999 999 999 998 = 2*121081*18911307306679
4 579 599 999 999 999 999 = 3*17*19*6625387*713333333
100 faktoriseringer beregnet paa:  333481.4427ms
 dvs:   3334.8144ms. per faktorisering

 largestLongFactorizedSafe: 4 579 599 841 640 001 173= 2139999949*2139999977
```



Hva har vi sett på i uke 5

1. Modell2-kode for sammenligning av kjøretider på (enkle) parallelle og sekvensielle algoritmer.
2. Hvordan lage en parallell løsning – ulike måter å synkronisere skriving på felles variable
3. Vranglås - et problem vi lett kan få (og unngå)
4. Ulike strategier for å dele opp et problem for parallellisering:
5. Hvorfor lage en avkryssingstabell over alle oddetall for å finne alle primtall (Eratosthenes sil) – steg 1 i Oblig 2


17 = 17
100 = 2×2×5×5
25 = 5×5
54 = 3×3×3×2
= 2×3×3×3
54/2 = 27
9
3
A
B
Wait for
Wait for
54 2
27 3
9 3
3 3


2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
0 0 0 1 0 0 1 0 0 1
2, 3, 5, 7, 11, 13, 17, 19

Summary of IN3030 lecture 2019-02-22

- Clarification of Oblig2
- About parallelizing algorithms
- Parallelizing Erastophenes sieve
- More about syncrhonzation methods in Java.
- More about running threads, yield, join
- Cooks and Servers (Kokke og kelnere) (demo with plates)
- Three solutions to Cooks and servers: (including code)
    - Busy waiting
    - Solution with synchronized and a common queue
    - Solution with Monitors and Conditions in Java

INF3030 Uke 6, våren 2019

Eric Jul
PSE
Inst. for informatikk

Å dele opp 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 CyclickBarrier-synkronisering mellom hver av disse delene + en synkronisert avslutning (join(), ..).
- Eks:
  - finnMax – hadde ett slikt steg: for (int i = 0 …n-1) -løkke
  - MatriseMult hadde ett slikt steg med trippel-løkke

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 'mye' tid)

Dette skal vi se på i uke 6

1. Kort om noen mulige måter å parallellisere Eratosthenes Sil og faktoriseringen i Oblig2 (mer neste uke).
2. **Concurrency:** Tre måter å programmere monitorer i Java eksemplifisert med tre løsninger av problemet: Kokker og Kelnere (eller generelt: produsenter og konsumenter)
    1. med sleep()og aktiv polling.
    2. med synchronized methods , wait>() og notify(),..
    3. med Lock og flere køer (Condition-køer)

2) Kortfattet om parallellisere Eratosthenes Sil (mer neste uke)

- Vi har M store tall som skal faktoriseres og skal nå lage n = $\sqrt{M}$ primtall
- Del opp noe:
  a. Tall-linjen (oddetallene) vi skal krysse av med –dvs. tallene mellom 3 og $\sqrt{M}$ i k deler (like store?) – kryss av hver del med **alle** primtallene vi har
     a. Men vi starter jo med bare to primtall :2 og 3 - alle?
  b. Dele opp primtallene vi skal krysse av med – f.eks
     a. Hver tråd tar neste primtall vi har funnet (3,5,..)
     b. Vi deler opp de første primtallene til tråd0, neste til tråd1,..
- Last-ballansering ?
  - Like mye arbeid til hver tråd – gir god speedup
- Er alle disse alternativene riktige ? (to skriver samtidig?)
- Skal noe kopieres lokalt og så samles når trådene er ferdige?

3) Flere (synkroniserings-) metoder i klassen Thread.

- **getName()** Gir navnet på tråden (default: Thread-0, Thread-1,..)
- **join():** Du venter på at en tråd terminerer hvis du kaller på dens join() metode.
- **sleep(t):** Den nå kjørende tråden sover i minst 't' millisek.
- **yield():** Den nå kjørende tråden pauser midlertidig og overlater til en annen tråd å kjøre på den kjernen som den første tråden ble kjørt på..
- **notify():** (arvet fra klassen Object, som alle er subklasse av). Den vekker opp **én** av trådene som venter på låsen i inneværende objekt. Denne prøver da en gang til å få det den ventet på.
- **notifyAll():** (arvet fra klassen Object). Den vekker opp **alle de** trådene som venter på låsen i inneværende objekt. De prøver da alle en gang til å få det de ventet på.
- **wait():** (arvet fra klassen Object). Får nåværende tråd til å vente til den enten blir vekket med notify() eller notifyAll() for dette objektet.

Et program som tester join(),yield() og getname() :

```
import java.util.ArrayList;

public class YieldTest {
    public static void main(String args[]){
        ArrayList<YieldThread> list = new ArrayList<YieldThread>();
        for (int i=0;i<20;i++){
            YieldThread et = new YieldThread(i+5);
            list.add(et);
            et.start();
        }
        for (YieldThread et:list){
            try {
                et.join();
            } catch (InterruptedException ex) { }
}}} // end class YielsTest

class YieldThread extends Thread{
   int stopCount;
    public YieldThread(int count){
        stopCount = count;
    }
    public void run(){
        for (int i=0;i<30;i++){
            if (i%stopCount == 0){
                System.out.println(«Stopper thread: "+getName());
                yield();
            }
}}} // end class YieldThread
```

```
M:\INF3030Para\Yield>java YieldTest
Stopper thread: Thread-0
Stopper thread: Thread-0
Stopper thread: Thread-5
Stopper thread: Thread-5
Stopper thread: Thread-9
Stopper thread: Thread-9
Stopper thread: Thread-12
Stopper thread: Thread-4
Stopper thread: Thread-3
Stopper thread: Thread-3
Stopper thread: Thread-2
Stopper thread: Thread-2
Stopper thread: Thread-2
Stopper thread: Thread-1
Stopper thread: Thread-1
Stopper thread: Thread-1
Stopper thread: Thread-3
Stopper thread: Thread-19
Stopper thread: Thread-18
Stopper thread: Thread-18
Stopper thread: Thread-17
Stopper thread: Thread-16
Stopper thread: Thread-16
```

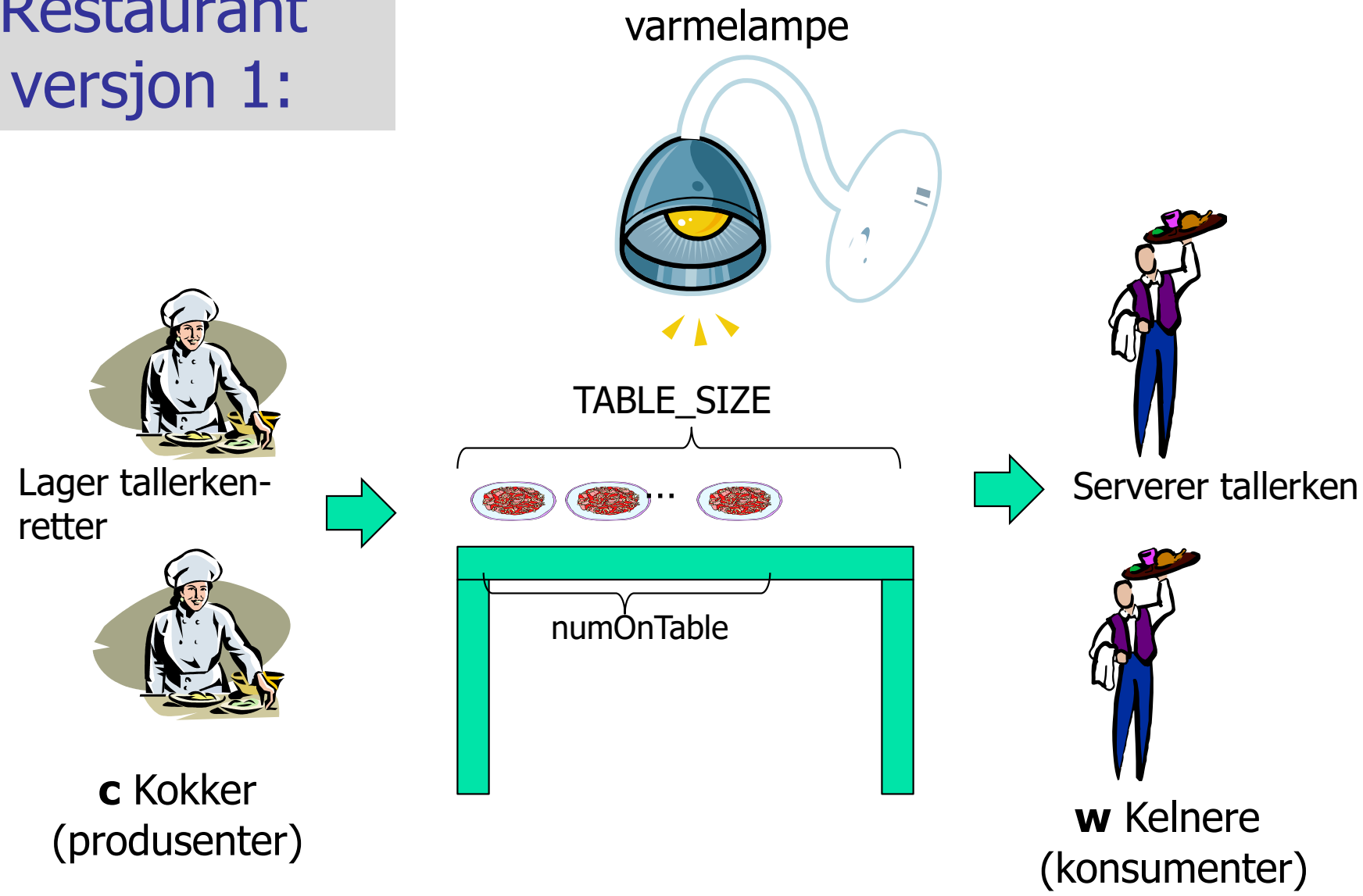
Vi ser at de 20 trådene gir fra seg kontrollen et ulike antall ganger > 0

```
Stopper thread: Thread-4
Stopper thread: Thread-4
Stopper thread: Thread-15
Stopper thread: Thread-12
Stopper thread: Thread-14
Stopper thread: Thread-14
Stopper thread: Thread-13
Stopper thread: Thread-13
Stopper thread: Thread-11
Stopper thread: Thread-11
Stopper thread: Thread-10
Stopper thread: Thread-8
Stopper thread: Thread-7
Stopper thread: Thread-6
Stopper thread: Thread-0
Stopper thread: Thread-6
Stopper thread: Thread-10
Stopper thread: Thread-15
Stopper thread: Thread-17
Stopper thread: Thread-19
Stopper thread: Thread-7
Stopper thread: Thread-8
```

Problemet vi nå skal løse: En restaurant med kokker og kelnere og med et varmebord hvor maten står

- Vi har **c** Kokker som lager mat og
  w Kelnere som server maten (tallerkenretter)
- Mat som kokkene lager blir satt fra seg på et **varmebord** (med TABLE_SIZE antall plasser til tallerkener)
- Kokkene kan ikke lage flere tallerkener hvis varmebordet er fullt
- Kelnerne kan ikke servere flere tallerkener hvis varmebordet er tomt
- Det skal lages og serveres NUM_TO_BE_MADE antall tallerkener

Restaurant versjon 1:



3) Om monitorer og køer (tre eksempler på concurrent programming). Vi løser synkronisering mellom to ulike klasser.

- **Først** en aktivt pollende (masende) løsning med synkroniserte metoder (Restaurant1.java).
  - Aktiv venting i en løkke i hver Kokk og Kelner
    + at de er i køen på å slippe inn i en synkronisert metode
- **Så** en løsning med bruk av monitor slik det var tenkt fra starten av i Java (Restaurant2.java).
  - Kokker og Kelnere venter i samme wait()-køen
    + i køen på å slippe inn i en synkronisert metode.
- **Til siste** en løsning med monitor med Lock og Condition-køer (flere køer – en per ventetilstand (Restaurant9.java)
  - Kelnere og Kokker venter i hver sin kø
    + i en køen på å slippe inn i de to metoder beskyttet av en Lock

.

Felles for de tre løsningene

```
import java.util.concurrent.locks.*;
class Restaurant {

  Restaurant(String[] args) {
     <Leser inn antall Kokker, Kelnere og  antall retter>
      <Oppretter Kokkene og Kelnerne og starter dem>
   }

    public static void main(String[] args) {
          new Restaurant(args);
       }
    } // end main
} // end class Restaurant

class HeatingTable{  // MONITOR
     int numOnTable  = 0,
         numProduced = 0,
         numServed=0;
     final int MAX_ON_TABLE  =3;
     final int NUM_TO_BE_MADE;
     // Invarianter:
     //   0 <= numOnTable  <=  MAX_ON_TABLE
     //          numProduced  <= NUM_TO_BE_MADE
     //          numServed      <= NUM_TO_BE_MADE

    < + ulike data i de tre eksemplene>
```

```
public xxx boolean putPlate(Kokk c)
   <Leggen tallerken til på bordet
   (true) ellers (false) Kokk må vente)
} // end put

public xxx boolean getPlate(Kelner w)  {
   <Hvis bordet tomt (false) Kelner venter
      ellers (true) - Kelner tar da en
      tallerken og serverer den>
  } // end get
} // end class HeatingTable

class Kokk extends Thread {
    HeatingTable tab;
    int ind;
    public void run() {
          while/if (tab.putPlate(..))
         < Ulik logikk i eksemplene>
    }
} // end class Kokk

class Kelner extends Thread {
    HeatingTable tab;
    int ind;
    public void run() {
          while/if (tab.getPlate()){
           <ulik logikk i eksemplene>
        }
} // end class Kelner
```

Invariantene på felles variable

- Invariantene må **alltid holde** (være sanne) utenfor monitor-metodene.
- Hva er de felles variable her:
  - MAX_ON_THE_TABLE
  - NUM_TO_BE_MADE
  - numOnTable
  - numProduced
  - numServed = numProduced – numOnTable
- Invarianter:
  1. **0 ≤ numOnTable ≤ TABLE_SIZE**
  2. **numProduced ≤ NUM_TO_BE_MADE**
  3. **numServed ≤ numProduced**

Invariantene viser 4 tilstander vi må ta skrive kode for

Invarianter:

0 <= numOnTable <= MAX_ON_TABLE

numServed <= numProduced <= NUM_TO_BE_MADE

1. **numOnTable == MAX_ON_TABLE**

   → **Kokker venter**

2. **0 == numOnTable**

   → **Kelnere venter**

3. **numProduced == NUM_TO_BE_MADE**

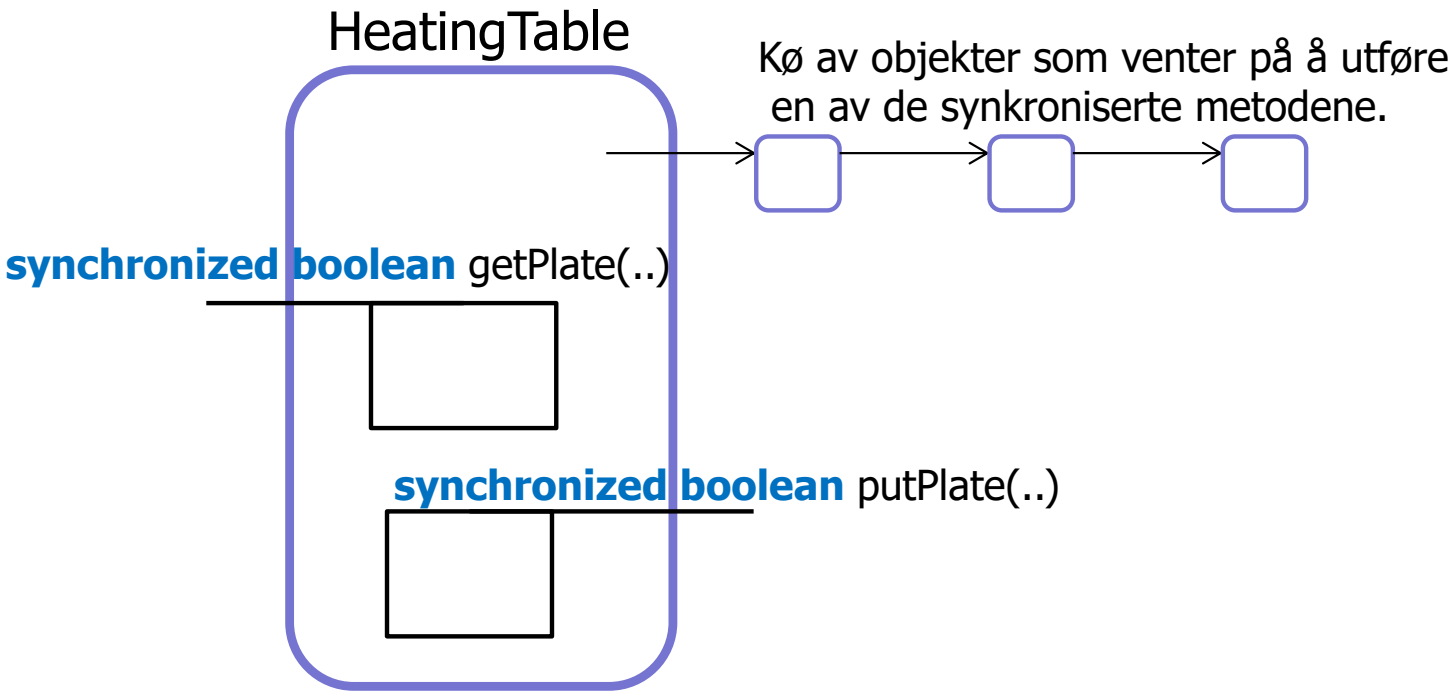
   → **Kokkene ferdige**

4. **numServed == NUM_TO_BE_MADE**

   → **Kelnerene ferdige**

Først en aktivt pollende (masende) løsning med synkroniserte metoder (Restaurant1.java).

- Dette er en løsning med **en kø**, den som alle tråder kommer i hvis en annen tråd er inne i en synkronisert metode i samme objekt.
- Terminering ordnes i hver kokk og kelner (i deres run-metode)
- Den køen som nyttes er en felles kø av alle aktive objekter som evt. samtidig prøver å kalle en av de to synkroniserte metodene **get** og **put.** Alle objekter har en slik kø.



Restaurant løsning 1

```
class Kokk extends Thread {
....
    public void run() {
       try {
          while (tab.numProduced < tab.NUM_TO_BE_MADE) {
            if (tab.putPlate(this) ) {
               // lag neste tallerken
            }
            sleep((long) (1000 * Math.random()));
          }
       } catch (InterruptedException e) {}
       System.out.println("Kokk "+ind+" ferdig: " );
   }
} // end Kokk

class Kelner extends Thread {
.............
     public void run() {
          try {
            while ( tab.numServed< tab.NUM_TO_BE_MADE) {
                 if ( tab.getPlate(this)) {
                    // server tallerken
                 }
                 sleep((long) (1000 * Math.random()));
               }
          } catch (InterruptedException e) {}
            System.out.println("Kelner " + ind+" ferdig");
     }
} // end Kelner
```

```
synchronized boolean  putPlate(Kokk c) {
      if (numOnTable == TABLE_SIZE) {
                return false;
       }
       numProduced++;
      // 0 <= numOnTable  <  TABLE_SIZE
      numOnTable++;
      // 0 < numOnTable  <=  TABLE_SIZE
      System.out.println("Kokk no:"+c.ind+",
        laget  tallerken no:"+numProduced);
     return true;
  } // end putPlate

synchronized boolean  getPlate(Kelner w) {
     if  (numOnTable == 0)     return false;
     // 0 < numOnTable  <=  TABLE_SIZE
     numServed++;
     numOnTable--;
     // 0 <= numOnTable <  TABLE_SIZE
     System.out.println("Kelner no:"+w.ind+
     ",  serverte  tallerken no:"+numServed);
    return true;
  }
```

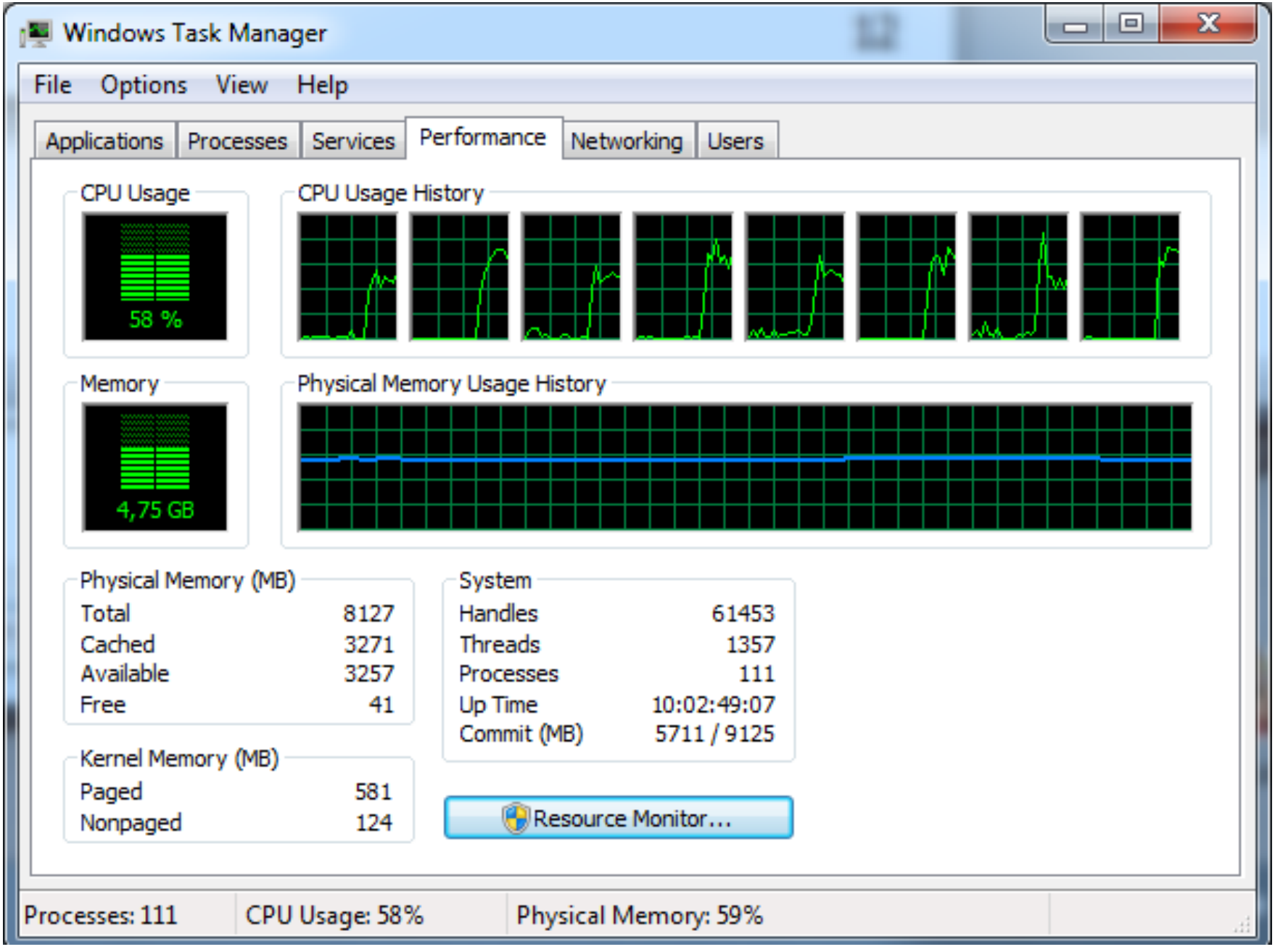
```
M:Restaurant1>java Restaurant1 11 8 8
Kokk no:8, laget  tallerken no:1
Kokk no:4, laget  tallerken no:2
Kokk no:6, laget  tallerken no:3
Kelner no:5, serverte  tallerken no:1
Kokk no:3, laget  tallerken no:4
Kelner no:2, serverte  tallerken no:2
Kokk no:1, laget  tallerken no:5
Kelner no:5, serverte  tallerken no:3
Kelner no:4, serverte  tallerken no:4
Kokk no:7, laget  tallerken no:6
Kokk no:2, laget  tallerken no:7
Kelner no:7, serverte  tallerken no:5
Kokk no:4, laget  tallerken no:8
Kelner no:3, serverte  tallerken no:6
Kelner no:3, serverte  tallerken no:7
Kokk no:1, laget  tallerken no:9
Kelner no:2, serverte  tallerken no:8
Kokk no:6, laget  tallerken no:10
Kokk no:3, laget  tallerken no:11
Kokk 8 ferdig:
```

```
Kelner no:8, serverte  tallerken no:9
Kelner no:7, serverte  tallerken no:10
Kelner no:6, serverte  tallerken no:11
Kokk 3 ferdig:
Kokk 5 ferdig:
Kelner 1 ferdig
Kokk 1 ferdig:
Kokk 4 ferdig:
Kelner 5 ferdig
Kokk 7 ferdig:
Kelner 2 ferdig
Kokk 2 ferdig:
Kelner 4 ferdig
Kelner 3 ferdig
Kelner 7 ferdig
Kelner 6 ferdig
Kokk 6 ferdig:
Kelner 8 ferdig
```

Problemer med denne løsningen er aktiv polling

- Alle Kokke- og Kelner-trådene går aktivt rundt å spør:
  - Er der mer arbeid til meg? Hviler litt, ca.1 sec. og spør igjen.
  - Kaster bort mye tid/maskininstruksjoner.
- Spesielt belastende hvis en av trådtypene (Produsent eller Konsument) er klart raskere enn den andre,
  - Eks . setter opp 18 raske Kokker som sover bare 1 millisek mot 2 langsomme Kelnere som sover 1000 ms.
  - I det tilfellet tok denne aktive ventingen/masingen 58% av CPU-kapasiteten til 8 kjerner
- Selv etter at vi har testet i run-metoden at vi kan greie en tallerken til, må vi likevel teste på om det går OK
  - En annen tråd kan ha vært inne og endret variable
- Utskriften må være i get- og put-metodene. Hvorfor?

Løsning1 med 18 raske Kokker (venter 1 ms) og 2 langsomme Kelnere (venter 1000 ms). Kokkene stresser maskinen med stadige mislykte spørsmål hvert ms. om det nå er plass til en tallerken på varmebordet . CPU-bruk = 58%.



Løsning 2: Javas originale opplegg med monitorer og **to køer**.

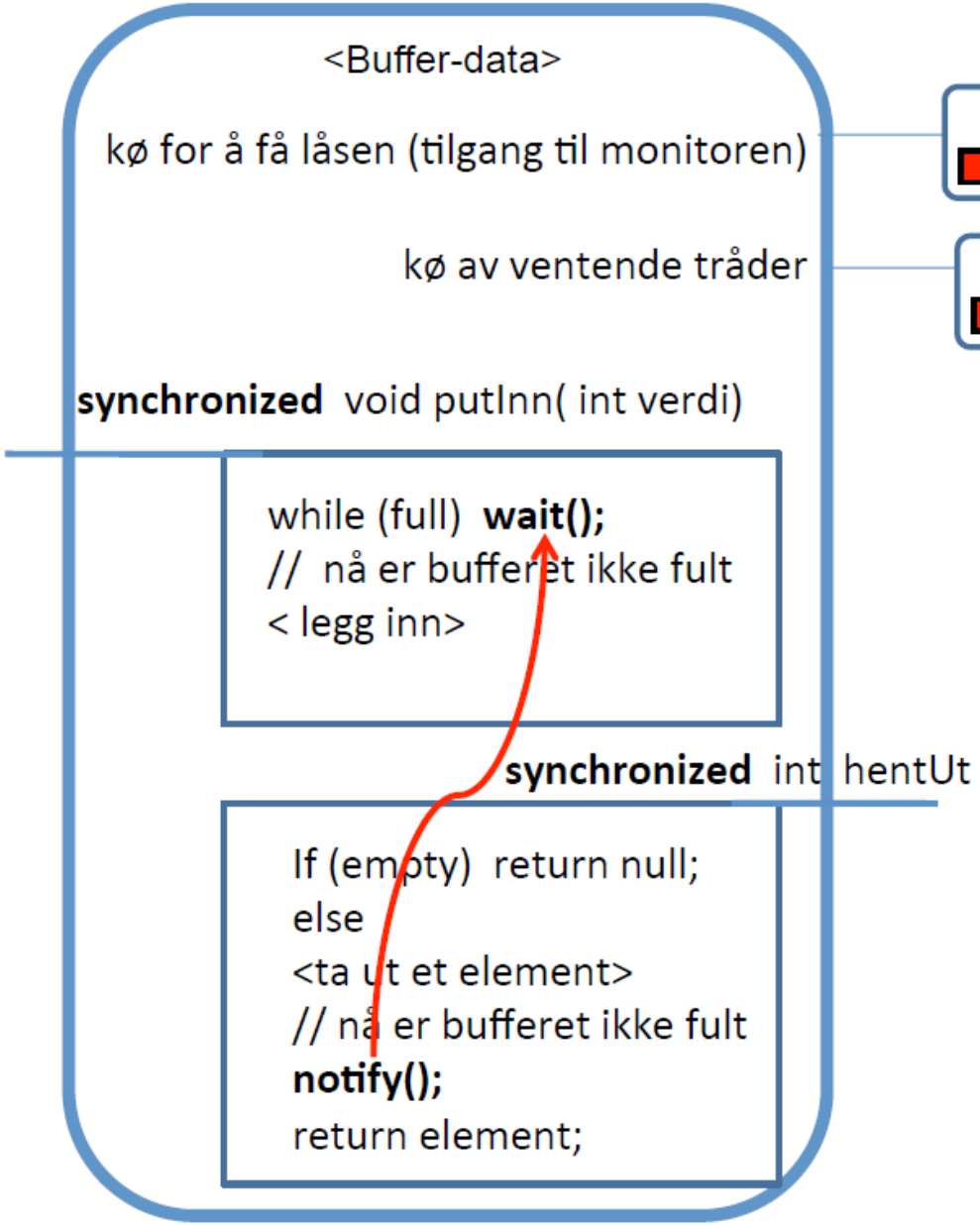
- Den originale Java løsningen med synkroniserte metoder og en rekke andre metoder og følgende innebygde metoder:
- **sleep(t):** Den nå kjørende tråden sover i 't' millisek.
- **notify():** (arvet fra klassen Object som alle er subklasse av). Den vekker opp **en** av trådene som venter på låsen i inneværende objekt. Denne prøver da en gang til å få det den ventet på.
- **notifyAll():** (arvet fra klassen Object). Den vekker opp **alle de** trådene som venter på låsen i inneværende objekt. De prøver da alle en gang til å få det de ventet på.
- **wait():** (arvet fra klassen Object). Får nåværende tråd til å vente til den enten blir vekket med notify() eller notifyAll() for dette objektet.

Å lage parallelle løsninger med en Java 'monitor'

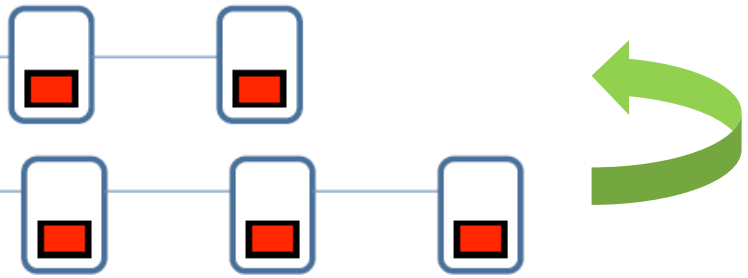
- En Java-monitor er et objekt av en vilkårlig klasse med synchronized metoder
- Det er to køer på et slikt objekt:
  - En kø for de som venter på å komme inn/fortsette i en synkronisert metode
  - En kø for de som her sagt **wait()** (og som venter på at noen annen tråd vekker dem opp med å si notify() eller notifyAll() på dem)
    - wait() sier en tråd inne i en synchronized metode.
    - notify() eller notifyAll() sies også inne i en synchronized metode.

Monitor-ideen er sterkt inspirert av Tony Hoare (mannen bak Quicksort)

To køer i en basal Java monitor:



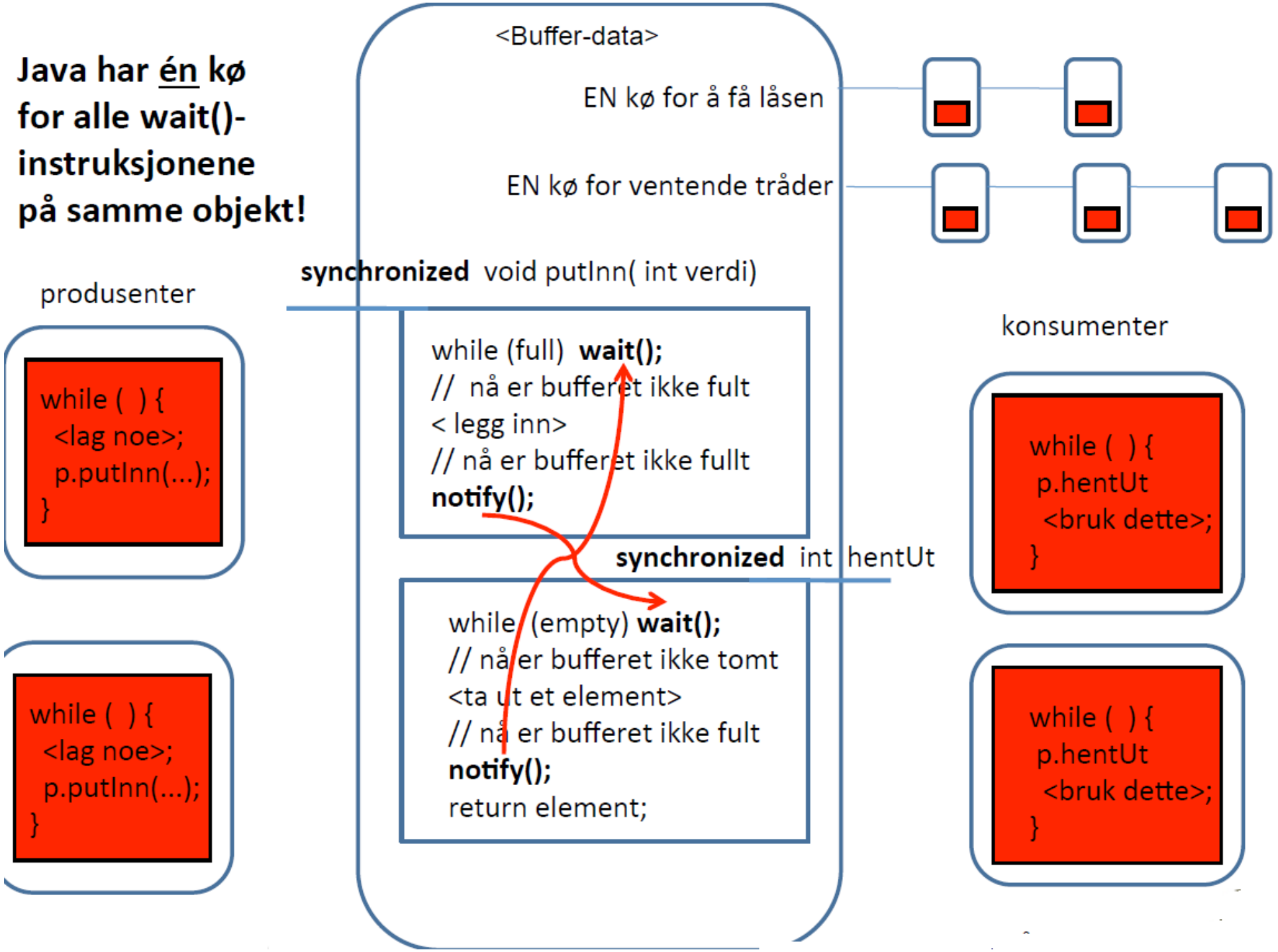
En kø av ventende tråder
på hele monitoren



notify()

En kø av ventende tråder på
"wait"-instruksjoner
(wait-set).
Startes av notify () og/eller
notifyAll()
Legges da i den andre køen
(først ? (Nei, ingen garanti))
Derfor er det nødvendig med
"while ..."

Java har <u>én</u> kø for alle wait()-instruksjonene på samme objekt!

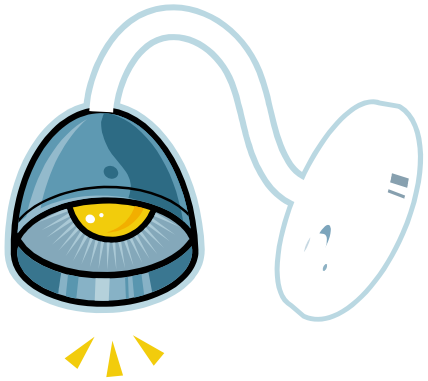


Pass på å unngå vranglås

Restauranten (2):

numProduced ≤ NOS_TO_BE_MADE

c Kokker (producers)



numServed = = numProduced - numOnTable

w Kelnere (consumers)

En kø: Kokker og Kelnere venter samme wait-køen

Løsning 2, All venting er inne i synkroniserte metoder i en to køer.

- All venting inne i to synkroniserte metoder
- Kokker and Kelnere venter på neste tallerken i wait-køen
- Vi må vekke opp alle i wait-køen for å sikre oss at vi finner en av den typen vi trenger (Kokk eller Kelner) som kan drive programmet videre
- Ingen testing på invariantene i run-metodene

Begge løsninger 2) og 3):
run-metodene prøver en gang til hvis siste operasjon lykkes:

Kokker:

```
public void run() {
        try {
            while (tab.putPlate(this)) {
                sleep((long) (1000 * Math.random()));
            }
        } catch (InterruptedException e) {}
        // Denne Kokken er ferdig
    }
```

Kelnere:

```
public void run() {
        try {
            while (tab.getPlate(this) ){
                sleep((long) (1000 * Math.random()));
            }
        } catch (InterruptedException e) {}
          // Denne Kelneren er ferdig
    }
```

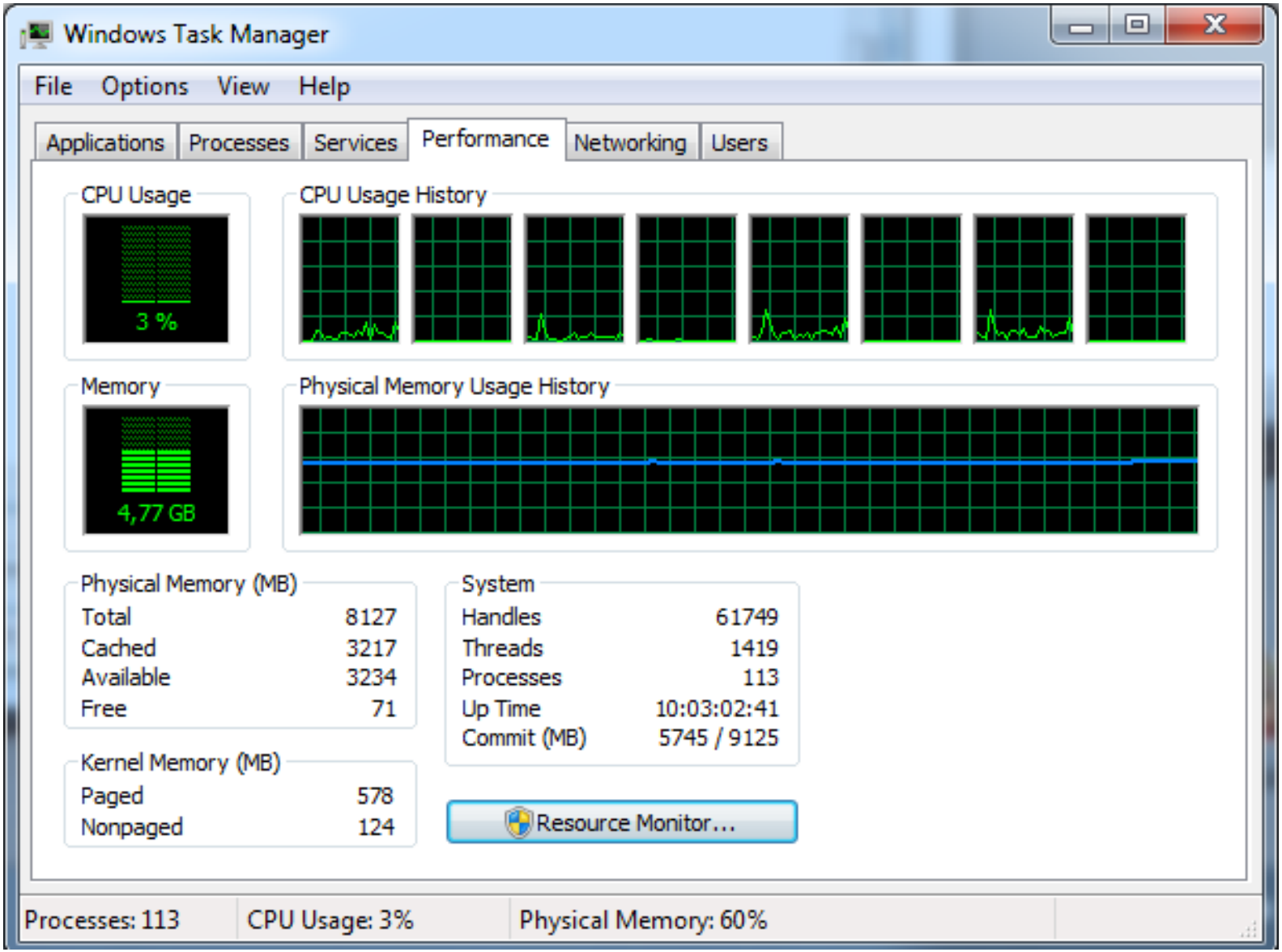
Løsning2

```
public synchronized boolean putPlate (Kokk c) {
    while (numOnTable == TABLE_SIZE &&
           numProduced <   NUM_TO_BE_MADE) {
        try { // The while test holds here meaning  that a Kokk should
              // but can not make a dish, because the table is full
              wait();
        } catch (InterruptedException e) { // Insert code to handle interrupt }
    }
    // one or both of the loop conditions are now false
    if (numProduced < NUM_TO_BE_MADE) {
        // numOnTable < TABLE_SIZE
        // Hence OK to increase numOnTable
        numOnTable++;
        // numProduced < NUM_TO_BE MADE
        // Hence OK to increase numProduced:
        numProduced++;
        // numOnTable > 0 , Wake up a waiting
        // waiter, or all if
        //numProduced ==NUM_TO_BE_MADE
        notifyAll();  // Wake up all waiting
        if (numProduced ==
            NUM_TO_BE_MADE) {
            return false;
        } else{ return true; }
    } else {
        // numProduced ==
        //NUM_TO_BE_MADE
        return false;}
} // end putPlate
```

Løsning2

```
public synchronized boolean getPlate (Kelner w) {
      while (numOnTable == 0 && numProduced < NUM_TO_BE_MADE ) {
            try { // The while test holds here the meaning that the table
                  // is empty and there is more to serve
                  wait();
            } catch (InterruptedException e) { // Insert code to handle interrupt }
       }
      //one or both of the loop conditions are now false
      if (numOnTable > 0) {
            // 0 < numOnTable <= TABLE_SIZE
            // Hence OK to decrease numOnTable:
            numOnTable--;
            // numOnTable < TABLE_SIZE
            // Must wake up a sleeping Kokker:
            notifyAll();   // wake up all queued Kelnere and Kokker
            if (numProduced == NUM_TO_BE_MADE && numOnTable == 0) {
                  return false;
            }else{ return true;}
      } else { // numOnTable == 0 && numProduced == NUM_TO_BE_MADE
            return false;}
} // end getPlate
```

Løsning2 med 18 raske Kokker (venter 1 ms) og 2 langsomme Kelnere (venter 1000 ms). Kokkene stresser ikke maskinen med stadige mislykte spørsmål , men venter i kø til det er plass til en tallerken til på varmebordet . CPU-bruk = 3%.



End of second lecture, uke 6

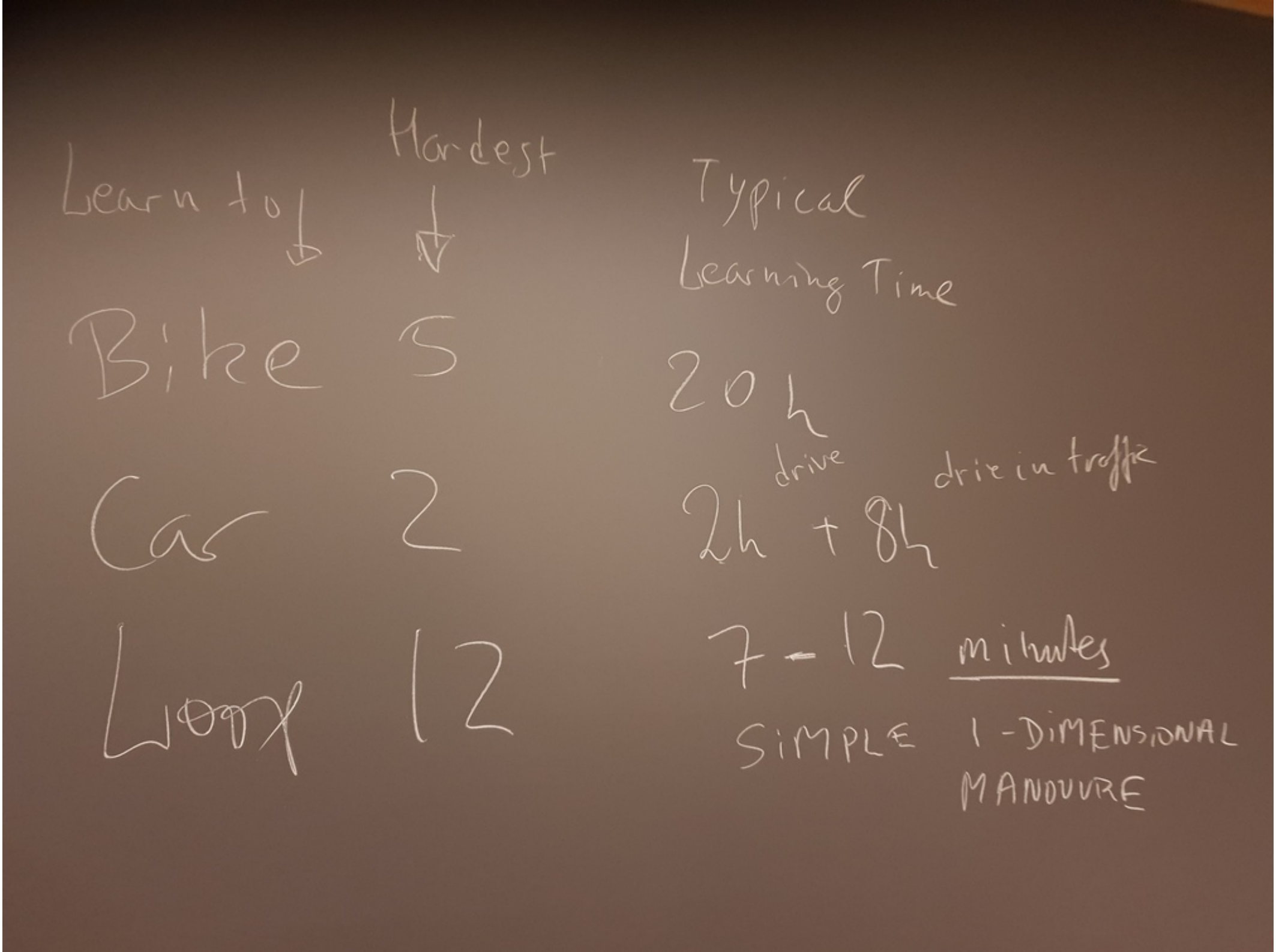
Reduce synchs between threads

1. Lose concurrency

2. Overhead SLOW!

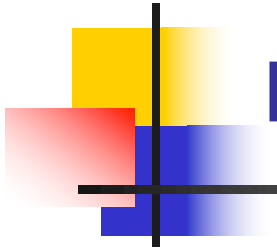
Transpose $\Theta(N^2)$

Multiply $\Theta(N^3)$


Learn to
Hardest
Typical Learning Time
Bike 5
20 h
Car 2
drive
2h + 8h
drive in traffic
Loop 12
7 - 12 minutes
SIMPLE 1-DIMENSIONAL MANOUVRE

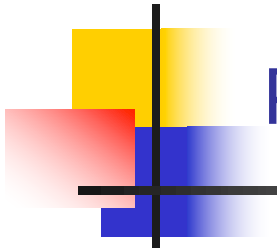
INF3030 Uke 7, våren 2019

Eric Jul
PSE
Inst. for informatikk



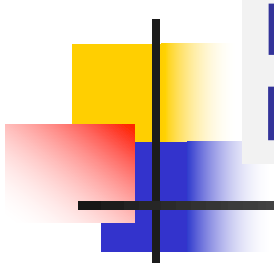
Hva så vi på i uke 6

1. Eratosthenes sil
2. Kokker og Kelnere
3. Concurrency: De første to av tre måter å programmere monitorer i Java eksemplifisert med to løsninger av problemet: Kokker og Kelnere (eller generelt: produsenter og konsumenter)
    1. med sleep() og aktiv polling.
    2. med monitor, synchronized methods , wait>() og notify(),..
    3. med Lock og flere køer (Condition-køer) (I dag!)



Plan for uke 7

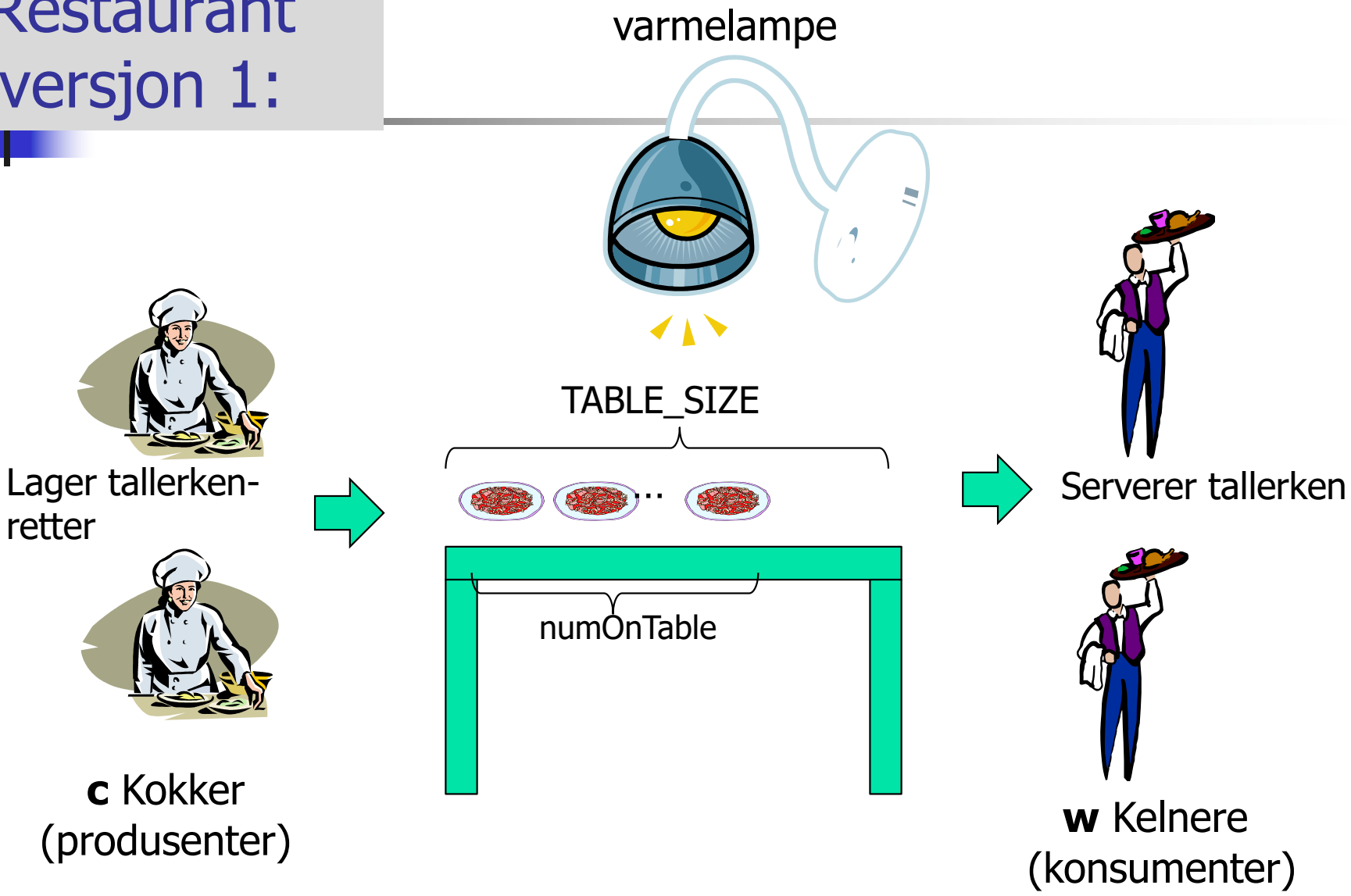
1. Piazza – use it – if you are OK with it. NOT REQUIRED.
2. Mere om synkronisering og kokkene og kelnere
3. Sil igjen
4. Hvilken orden O() har Eratosthenes Sil ?
5. Faktorisering av tall
6. Alternativer for å parallellisere: Eratosthenes Sil
7. Generelt om last-balansering ved parallellisering med eksempel.



Problemet vi nå skal løse: En restaurant med kokker og kelnere og med et varmebord hvor maten står

- Vi har **c** Kokker som lager mat og
  w Kelnere som server maten (tallerkenretter)
- Mat som kokkene lager blir satt fra seg på et **varmebord** (med TABLE_SIZE antall plasser til tallerkener)
- Kokkene kan ikke lage flere tallerkener hvis varmebordet er fullt
- Kelnerne kan ikke servere flere tallerkener hvis varmebordet er tomt
- Det skal lages og serveres NUM_TO_BE_MADE antall tallerkener

Restaurant versjon 1:



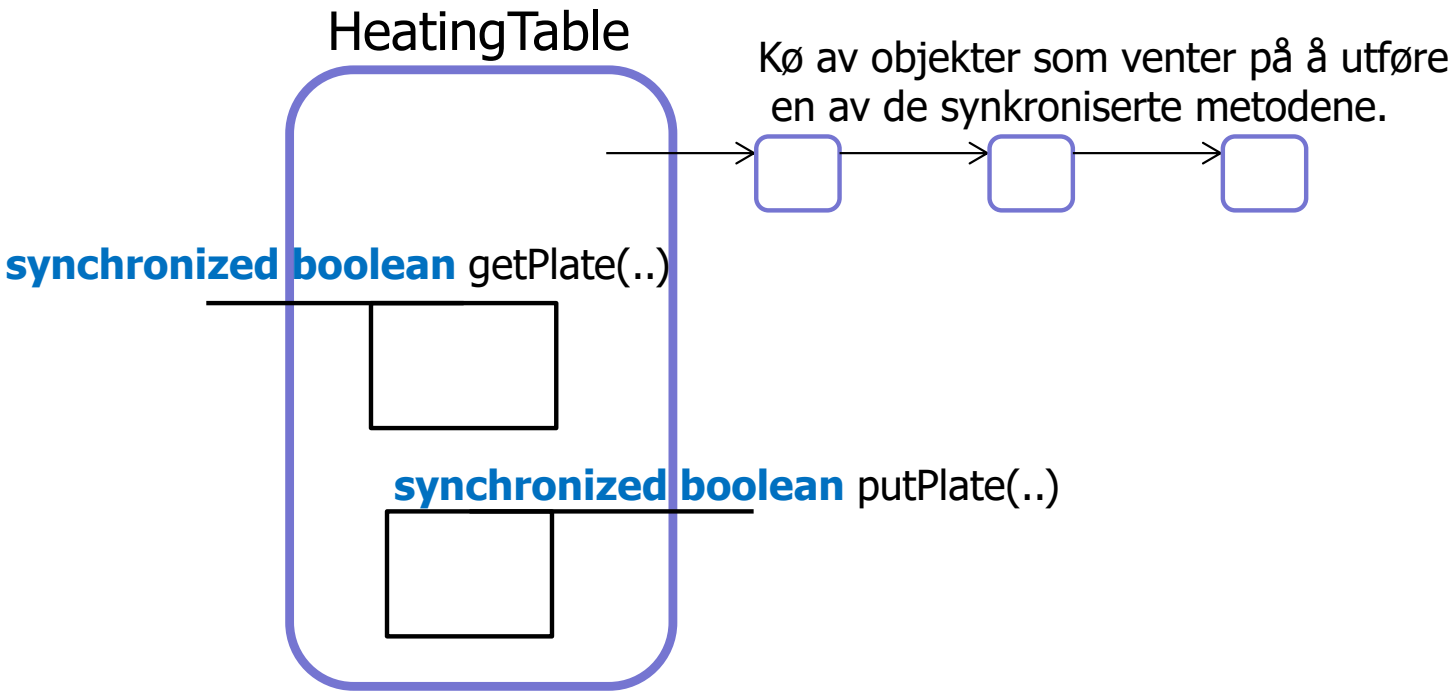
3) Om monitorer og køer (tre eksempler på concurrent programming). Vi løser synkronisering mellom to ulike klasser.

- **Først** en aktivt pollende (masende) løsning med synkroniserte metoder (Restaurant1.java).
  - Aktiv venting i en løkke i hver Kokk og Kelner
    + at de er i køen på å slippe inn i en synkronisert metode
- **Så** en løsning med bruk av monitor slik det var tenkt fra starten av i Java (Restaurant2.java).
  - Kokker og Kelnere venter i samme wait()-køen
    + i køen på å slippe inn i en synkronisert metode.
- **Til siste** en løsning med monitor med Lock og Condition-køer (flere køer – en per ventetilstand (Restaurant9.java)
  - Kelnere og Kokker venter i hver sin kø
    + i en køen på å slippe inn i de to metoder beskyttet av en Lock

.

Først en aktivt pollende (masende) løsning med synkroniserte metoder (Restaurant1.java)

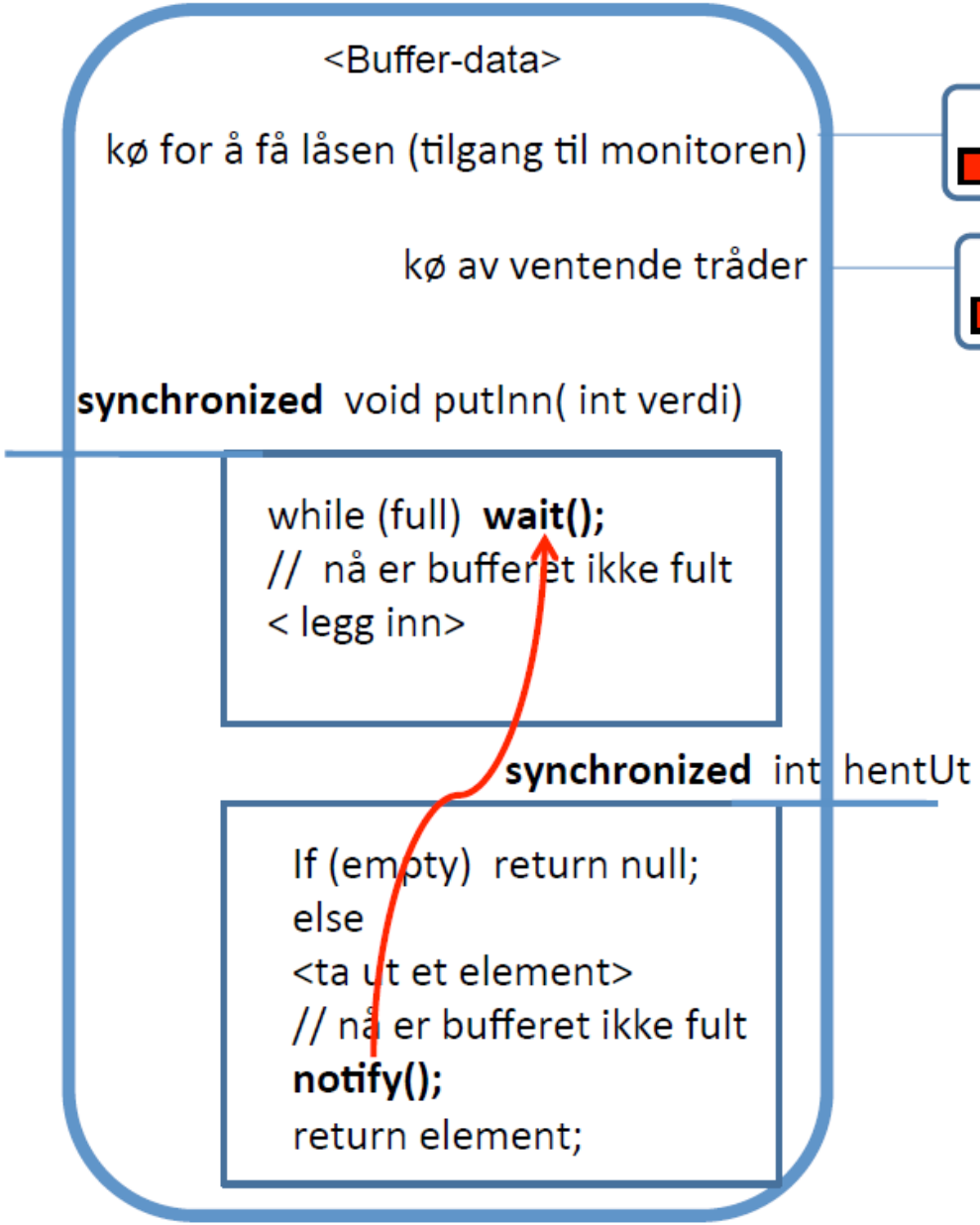
- Dette er en løsning med **en kø**, den som alle tråder kommer i hvis en annen tråd er inne i en synkronisert metode i samme objekt.
- Terminering ordnes i hver kokk og kelner (i deres run-metode)
- Den køen som nyttes er en felles kø av alle aktive objekter som evt. samtidig prøver å kalle en av de to synkroniserte metodene **get** og **put.** Alle objekter har en slik kø.



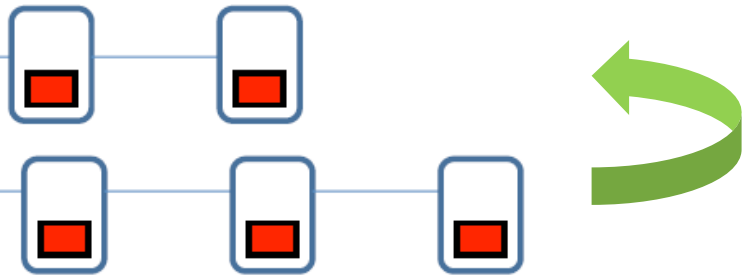
Løsning 2: Javas originale opplegg med monitorer og **to køer**.

- Den originale Java løsningen med synkroniserte metoder og en rekke andre metoder og følgende innebygde metoder:
- **sleep(t):** Den nå kjørende tråden sover i 't' millisek.
- **notify():** (arvet fra klassen Object som alle er subklasse av). Den vekker opp **en** av trådene som venter på låsen i inneværende objekt. Denne prøver da en gang til å få det den ventet på.
- **notifyAll():** (arvet fra klassen Object). Den vekker opp **alle de** trådene som venter på låsen i inneværende objekt. De prøver da alle en gang til å få det de ventet på.
- **wait():** (arvet fra klassen Object). Får nåværende tråd til å vente til den enten blir vekket med notify() eller notifyAll() for dette objektet.

To køer i en basal Java monitor:

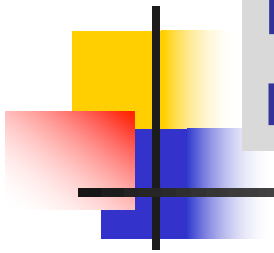


En kø av ventende tråder
på hele monitoren



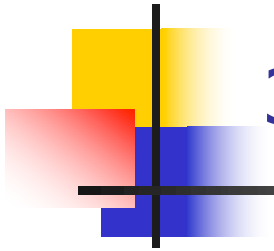
notify()

En kø av ventende tråder på
"wait"-instruksjoner
(wait-set).
Startes av notify () og/eller
notifyAll()
Legges da i den andre køen
(først ? (Nei, ingen garanti))
Derfor er det nødvendig med
"while ..."



Løsning 2, All venting er inne i synkroniserte metoder i en to køer.

- All venting inne i to synkroniserte metoder
- Kokker and Kelnere venter på neste tallerken i wait-køen
- Vi må vekke opp alle i wait-køen for å sikre oss at vi finner en av den typen vi trenger (Kokk eller Kelner) som kan drive programmet videre
- Ingen testing på invariantene i run-metodene



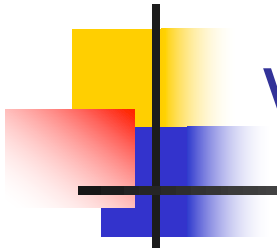
3) En parallell løsning med Doug Lea's Conditions.

- Bruker to køer:
  - En for Kokker som venter på en tallerkenplass på bordet
  - En for Kelnere som venter på en tallerken
- Da trenger vi ikke vekke opp alle trådene, **Bare en** i den riktige køen.
  - Kanskje mer effektivt
  - Klart lettere å programmere

```
final Lock lock = new ReentrantLock();
final Condition notFull  = lock.newCondition();     // kø for Kokker
final Condition notEmpty = lock.newCondition(); // kø for Kelnere

public boolean putPlate (Kokker c) throws InterruptedException {
     lock.lock();
      try {while (numOnTable == MAX_ON_TABLE && numProduced < NUM_TO_BE_MADE){
                notFull.await(); // waiting for a place on the table
             }
            if (numProduced < NUM_TO_BE_MADE) {
                  numProduced++;
                  numOnTable++;;
                  notEmpty.signal(); // Wake up a waiting Kelner to serve
                  if (numProduced == NUM_TO_BE_MADE) {
                       //  I have  produced the last plate,
                      notEmpty.signalAll();   // tell Kelnere to stop waiting, terminate
                      notFull.signalAll();       // tell Kokker to stop waiting and terminate
                      return false;
                   }
                 return true;
            } else {  return false;}
      } finally {
         lock.unlock();
  } } // end putPlate
```

```
public boolean getPlate (Kelnerw) throws InterruptedException {
    lock.lock();
    try {
        while (numOnTable == 0 && numProduced < NUM_TO_BE_MADE ) {
            notEmpty.await();      // This Kelner is waiting for a plate
        }
        if (numOnTable > 0) {
            numOnTable--;
            notFull.signal();  // Signal to one Kokk in the Kokker's waiting queue
            return true;
        } else {
            return false;}
    } finally {
        lock.unlock();
    }
} // end getPlate
```

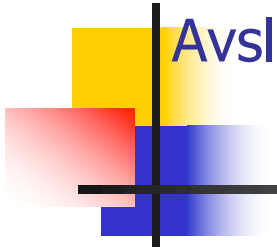
En Kelner eller en Kokk blir signalisert av to grunner:
- for å behandle (lage eller servere) en tallerken til
- ikke mer å gjøre, gå hjem (tøm begge køene)



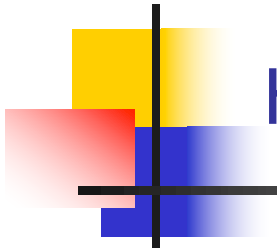
Vurdering av de tre løsningene

- **Løsning 1**: Enkel, men kan ta for mye av CPU-tiden. Særlig når systemet holder av andre grunner å gå i metning vil typisk en av trådene da bli veldig treige, og da tar denne løsningen plutselig ½-parten av CPU-tiden.
- **Løsning 2:** God, men vanskelig å skrive
- **Løsning 3:** God, nesten like effektiv som løsning 2 og lettere å skrive.

Avsluttende bemerkninger til Produsent-Konsument problemet



- Invarianter brukes av alle programmerere (ofte ubevisst)
  - program, loop or metode (sekvensiell eller parallell)
  - Å si dem eksplisitt hjelper på programmeringen
- HUSK: synchronized/lock virker bare når alle trådene synkroniserer på samme objektet.
  - Når det skjer er det **sekvensiell tankegang** mellom wait/signal
- Når vi sier notify() eller wait() på en kø, vet vi ikke:
  - Hvilken tråd som starter
  - Får den tråden det er signalisert på kjernen direkte etter at den som sa notify(), eller ikke ??. Ikke definert
- Debugging ved å spore utførelsen (trace) – System.out.println(”..”)
  - Skriving utenfor en Locket/synkronisert metode/del av metode, så lag en:
    - synchronized void println(String s) {System.out.println(s);}
  - Ellers kan utskrift bli blandet eller komme i gal rekkefølge.



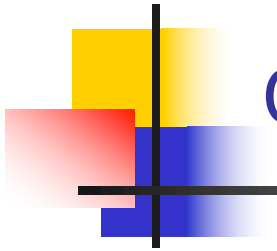
Hoare's Original Monitors and Conditions

- **Review of Hoare's article**

Om sil og faktorisering

- Eratosthenes sil
  - Vi krysser av i en bit-tabell. Husk det er 'bare' ***ikke-primtallene*** vi finner ved avkryssingen (primtallene er de som står igjen etter all nødvendig avkryssing)
  - (Det er ikke mulig å effektivt finne primtallene direkte! Ingen formel for primtallene!)

- Faktorisering av M:
  - Vi dividerer M med alle primtall < N i E-Silen for $N = \sqrt{M}$
  - Finner vi en faktor f i M , fortsetter vi med faktorisering av M' = M/f;



Quick Review of Erastophene's Sieve

- **Review of crossing off**

Hvor mange ganger krysser vi av i Eratosthenes Sil for alle primtall < N = $\sqrt{M}$?

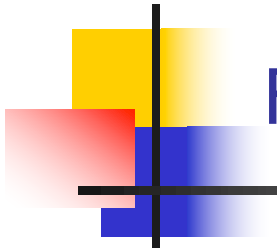
- Hvordan vet vi hvor mange kryss som settes med ES?
- Se på faktoriseringa:

```
sekv: 3999999999999999899 = 107*37383177570093457
sekv: 3999999999999999900 = 2*2*3*5*5*89*1447*1553*66666667
sekv: 3999999999999999901 = 19*2897*72670457642207
sekv: 3999999999999999902 = 2*49965473*40027640687
sekv: 3999999999999999903 = 3*101*241*54777261958561
sekv: 3999999999999999904 = 2*2*2*2*2*1061*117813383600377
sekv: 3999999999999999905 = 5*7*13*8791208791208791
sekv: 3999999999999999906 = 2*3*3*17*31*421674045962471
sekv: 3999999999999999907 = 14521697*275449900931
sekv: 3999999999999999908 = 2*2*11*168409*539811357523
sekv: 3999999999999999909 = 3*2713*491460867428431
sekv: 3999999999999999910 = 2*5*399999999999999991
sekv: 3999999999999999911 = 167*659*3581*86629*117163
sekv: 3999999999999999912 = 2*2*2*3*7*7*4127*669869*1230349
sekv: 3999999999999999913 = 239*16736401673640167
```

- Vi ser at de største tallene har (2+6+3+..+2) ulike faktorer som ikke er 2, er = 3,5 i snitt – si 4.
- Hver slik ulike faktor i tallet t < $\sqrt{M}$ tilsvarer 1 avkryssing av p_i (eller en mindre faktor p_j) < $\sqrt[2]{N}$ (=$\sqrt[4]{M}$)

Hvilken orden har Eratosthnes Sil – algoritmen (N)

- Vi krysser av for alle primtall < $\sqrt{N}$ - hvor mange er det?
  - svar: omlag: $\frac{\sqrt{N}}{\log(\sqrt{N})}$
- Hvor mange kryss settes av p_i mellom p_i^2 og N ?
  - svar: omlag $\frac{N-pi^2}{2}$ / (2* p_i)< $\frac{N}{4}$
- Et øvre estimat for (antall primtall)* (antall kryss per primtall)=
  - O(N) = $\frac{\sqrt{N}}{\log(\sqrt{N})} * \frac{N}{4} = \frac{N\sqrt{N}}{\log(\sqrt{N})} = \frac{2*N\sqrt{N}}{\log(N)} = \frac{\boldsymbol{N\sqrt{N}}}{\boldsymbol{log(N)}}$



Factorizing Numbers

- **How to factorize number**

Om å parallelliser Sil og Faktorisering, **Riktig** og **Raskere**

- Eratosthenes sil
  - Vi krysser av i en bit-tabell.
  - Hvordan gjøre dette i parallell
    - Hva det er vi deler opp, hva gjør hver tråd?
    - Skal vi kopiere noen data til hver tråd?
    - Hva er felles data og hva er lokale data i hver tråd.
- Faktorisering av M:
  - Vi dividerer M med alle primtall i E-Silen fra 0:$\sqrt{M}$
    - Parallelliseringen av faktorisering: Hvordan dele opp :
      - Primtallene ?
      - Tallene fra 0: M ?
    - Skal vi kopiere noen data til hver tråd?
    - Hva er felles data og hva er lokale data i hver tråd.

Riktig og Raskere – Eratosthenes Sil

- Mulig problem
  - E-sil : Hvis to tråder setter av kryss samtidig i en byte, kan det da gå galt?
    - byte[i] = byte[i] & xxxx; // problem ?
    - Hvordan blir dette utført i kjernen
    - Må vel da synkronisere ?

- Raskere:
  - Antall synkroniseringer må ikke bli for stort
    - Antall tråder ganger (evt * 2,3,4) : Helt OK
    - Log (N) ganger, evt log(N)*ant tråder: OK,
    - $\sqrt{N}$ tja ? (hvor stor %-andel er $\sqrt{N}$ av N, N=100, N=10 000)
    - **ikke** :N = $\sqrt{M}$ eller M ganger

Alternativ 1 for Eratosthenes Sil

- Dele opp primtallene mellom trådene så hver tråd tar noen hver og krysser av i hele Sila:
  - Ulempe: 3 og 5 genererer langt de fleste kryssene (lastbalansering)
    - Kunne la tråd$_0$ ta 3, tråd$_1$ ta 5 , osv
  - Problem: to ulike primtall krysser av ulike steder i ***samme*** byte – kan da vel miste ett av kryssene (eks: 3 kryssser 51, 7 krysser 49) Avkryssing er 3 opersjoner: Last opp i register , &-operasjon, lagre tilbake
  - To primtall krysser av for det samme tallet (eks. 7 og 5 krysser av 105) – da taper vi vel (?) ingen ting om ett av kryssene går tapt.
- Hvis alle trådene har sin kopi av Sila, skulle dette gå OK.
  - Må da tilslutt 'slå sammen' disse k stk byte-arrayene .
  - Tiden det tar å slå sammen disse vil kanskje bli 'lang' ??

Alternativ 2 for Eratosthenes Sil:

- Dele opp tallene i Sila : tallene 0:N mellom trådene,
  - Kan ikke dele helt likt – skillet må gå mellom to byter
    - Hvorfor (?)
- Hver tråd krysser av med alle aktuelle primtall på sin del av Sila.
- Problem 1: Hvor starter avkryssingen av primtall p i hver tråds (første) byte.
- Problem 2: Hvordan vet trådene som har området: *low..high* hvilke primtall den skal krysse av for??
- Hvilke er det ? Svar : alle primtall < $\sqrt{N}$ eller $\sqrt{high}$
- Hvordan lage og lagre dem først?
  - Tre alternativer:
    - Tråd$_0$ lager dem først (sekvensiell del – jfr. Amdahls lov)
    - Alle trådene lager de samme primtallene < $\sqrt{N}$ i en lokal kopi
    - Se på dette rekursivt som et problem som kan parallelliseres
      (dvs . Sekvensiell fase : Først alle primtall < $\sqrt{\sqrt{N}}$ som så lager primtall < $\sqrt{N}$)

Grep 2 – vi lager noen ekstra data

- I FinnMax-problemet hadde vi først:
  - en lokal variabel: lokalMax som hver tråd lokalt oppdaterte til de var ferdige; og så kalte hver tråd en global synkronisert metode som evt. satte ny verdi i globalMax
- Kan vi prøve noe tilsvarende her. I løsning a) trenger alle trådene primtallene < $\sqrt{N}$.
- Skisse - enten:

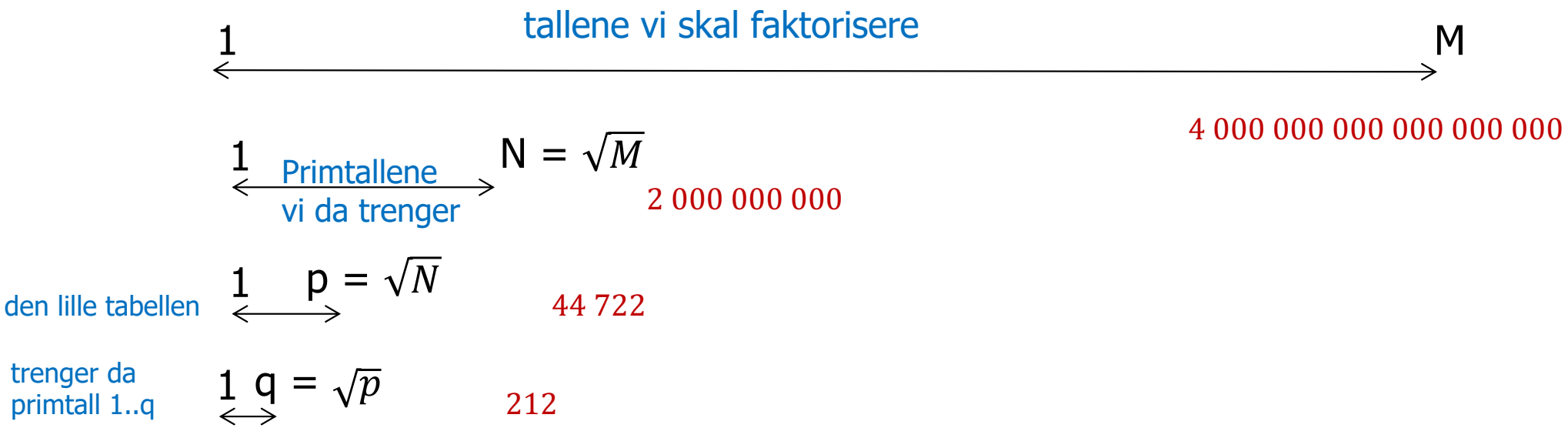
  a1) Tråd-0 lager disse primtallene først i tabellen mens de andre trådene venter, eller
  a2) Alle trådene lager hver sin lokale tabell over disse primtallene
  a3) Enten a1) eller a2), så krysser hver tråd deretter av i sin del av bit-arrayen for alle disse primtallene.

  N.B. Hvis a1) skal tråd-0 bare bruke disse 'små' primtallene til først å krysse av i området opp til $\sqrt{N}$, ikke i hele sitt område.

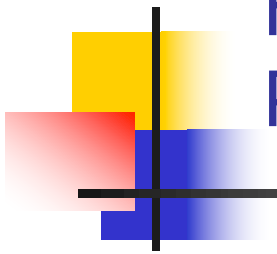
Billig å lage en slik liten start-tabell for trådene i alt. a)

- Eks: skal vi parallelt krysse av for primtall < 2 milliarder, trenger denne tabellen plass til $\sqrt{2\ mrd} = 44722$ bit/16= 2796 byter for å finne de primtallene p < 44722.

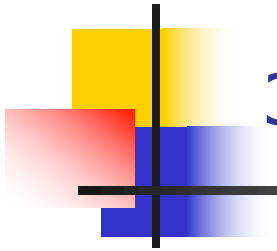
(og i den lille tabellen trenger vi bare krysse av med primtall q < $\sqrt{44722} = 212$ for å finne dem. Billig start på å parallellisere det å finne alle primtall < 2000 000 000).



Faktorisering alt 3:
Primtallene 0:N deles likt ut en etter en etter tur

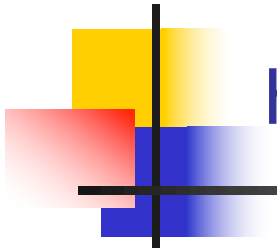


- Alle trådene tar hver k'te primtall
- Anta at k=4
  - $Tråd_0$ tar: 3, 13,...
  - $Tråd_1$ tar: 5, 17,...
  - $Tråd_2$ tar: 7, 19,...
  - $Tråd_3$ tar: 11, 23,...
- Fordeler
  - Med de andre alternativene: Hvis vi finner f.,eks at 13 er en faktor, så har tråd 1,2 og 3 ikke mer å gjøre (og $tråd_0$ gjør resten av arbeidet)
- Ulempe
  - Hvor raskt er det stadig for en tråd å finne neste k'te primtall i tabellen
  - Kan vi organisere primtallene slik at det går raskere ?



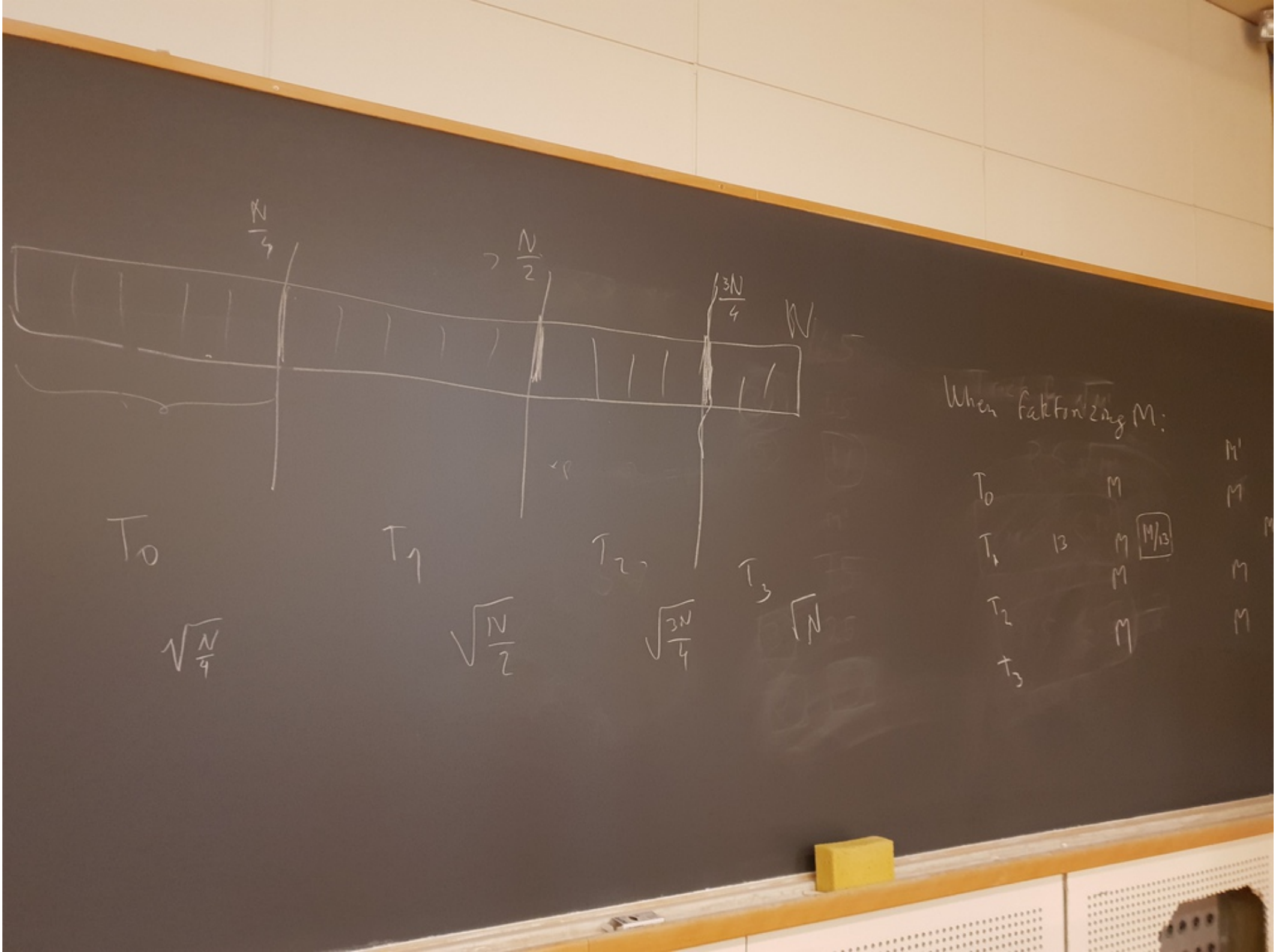
3) Generelt om last-balansering og 'ekstra' lokale data

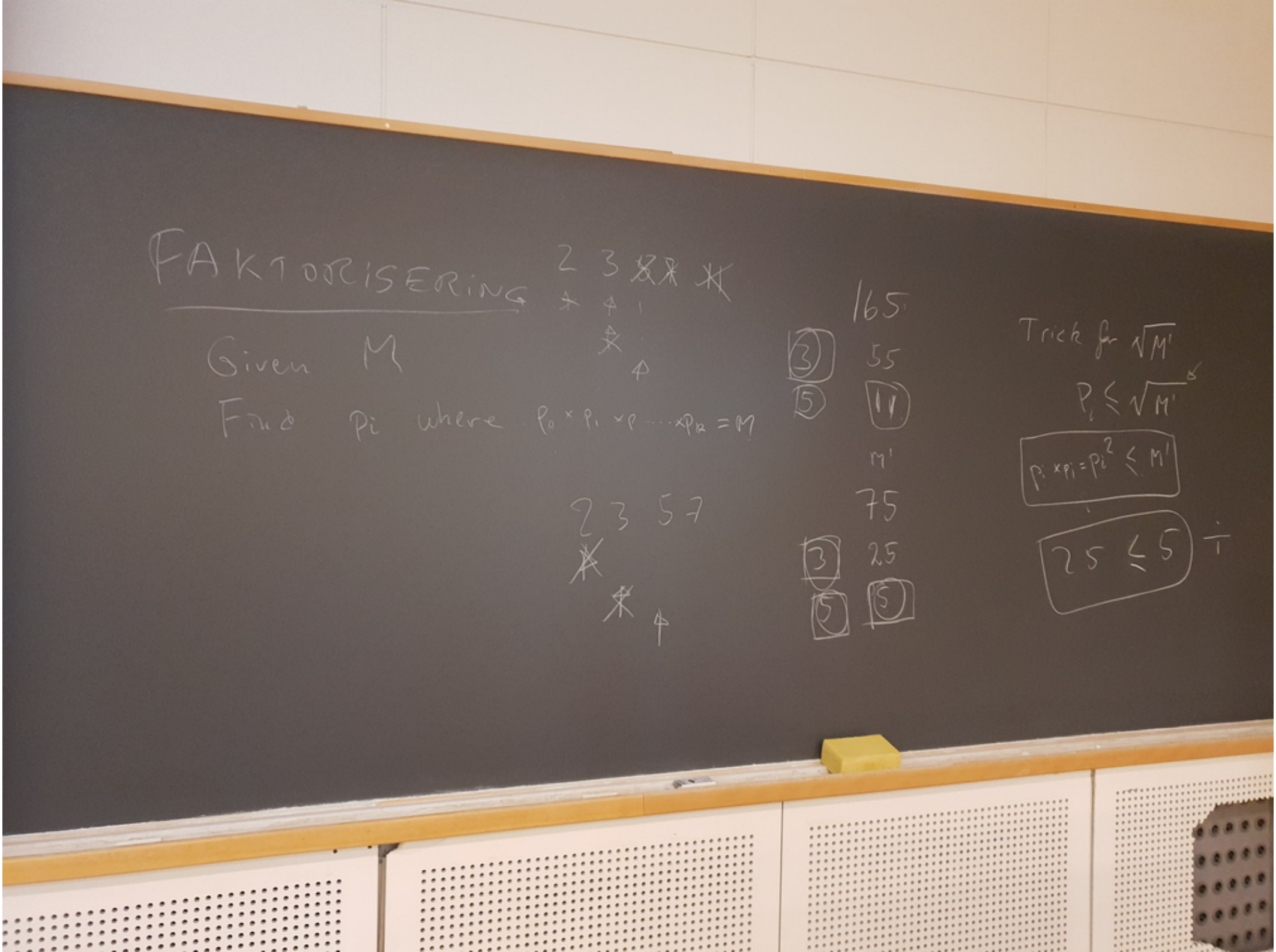
- Vitsen er at alle trådene har om lag like mye å gjøre (tidsmessig), eller sagt på en annen måte:
  - Vi prøver å optimalisere slik at den tråden som fullfører sist gjør det raskest mulig.
- Når vi skal parallellisere er det (nesten) alltid en array eller matrise vi skal regne/gjøre noe med:
  - Vi kan enten dele opp selve arrayen (i like store deler 1/k – del til hver av de k trådene) – dvs. dele opp **indeksene**
    - radvis eller kolonnevis hvis det er en matrise,
  - eller vi kan dele opp **verdiene** i arrayen slik at tråd-0 får de 1/k minste verdiene, tråd-1 de neste 1/k verdiene,.. osv, eller
  - vi kan estimere arbeidsmengden og vi 1/k-del av den til hver av trådene, eller …
  - ? .



Hva har vi sett på i uke 7

1. Piazza – please use it
2. Mere om synkronisering og kokkene og kelnere – demo ved tavle ☺
3. Erastostenes sil review
4. Hvilken orden O() har Eratosthenes Sil ?
5. Faktorisering – med eksempler
6. Hvordan lage en parallell løsning – ulike måter å synkronisere skriving på felles variable – med eksempel
7. Ulike strategier for å dele opp et problem for parallellisering


N/4
N/2
3N/4
N
T0
T1
T2
T3
√N/4
√N/2
√3N/4
√N
When factoring M:
M'
T0 M M
T1 13 M M/13
M
T2 M
T3


FAKTORISERING
Given M
Find Pi where P0 × P1 × P ···· × Pn = M
2 3
2 3 5 7
165
3
55
5
11
M'
75
3
25
5
5
Trick for √M'
Pi ≤ √M'
Pi × Pi = Pi² ≤ M'
25 ≤ 5

Piazza

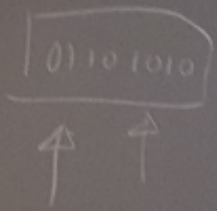
2 3 5 7 11 13 17 19 23

N'

561

3 187

11 17



Simple parallelization of sieve

N

T_0 : 2, 3 2 13 23

T_1 : 5, 7 3 17

T_2 : 11, 13 11 19 $\sqrt{N}$

Monitors: An Operating System Structuring Concept

C.A.R. Hoare
The Queen's University of Belfast

This paper develops Brinch-Hansen's concept of a monitor as a method of structuring an operating system. It introduces a form of synchronization, describes a possible method of implementation in terms of semaphores and gives a suitable proof rule. Illustrative examples include a single resource scheduler, a bounded buffer, an alarm clock, a buffer pool, a disk head optimizer, and a version of the problem of readers and writers.

Key Words and Phrases: monitors, operating systems, scheduling, mutual exclusion, synchronization, system implementation languages, structured multiprogramming
CR Categories: 4.31, 4.22

This paper is based on an address delivered to IRIA, France, May 11, 1973. Author's address: Department of Computer Science, The Queen's University of Belfast, Belfast BT7 1NN, Northern Ireland.

1. Introduction

A primary aim of an operating system is to share a computer installation among many programs making unpredictable demands upon its resources. A primary task of its designer is therefore to construct resource allocation (or scheduling) algorithms for resources of various kinds (main store, drum store, magnetic tape handlers, consoles, etc.). In order to simplify his task, he should try to construct separate schedulers for each class of resource. Each scheduler will consist of a certain amount of local administrative data, together with some procedures and functions which are called by programs wishing to acquire and release resources. Such a collection of associated data and procedures is known as a *monitor*; and a suitable notation can be based on the *class* notation of SIMULA67 [6].

```
monitorname: monitor
  begin . . . declarations of data local to the monitor;
    procedure procname (. . . formal parameters . . .);
      begin . . . procedure body . . . end;
    . . . declarations of other procedures local to the monitor;
    . . . initialization of local data of the monitor . . .
  end;
```

Note that the procedure bodies may have local data, in the normal way.

In order to call a procedure of a monitor, it is necessary to give the name of the monitor as well as the name of the desired procedure, separating them by a dot:

```
monitorname.procname(. . . actual parameters . . .);
```

In an operating system it is sometimes desirable to declare several monitors with identical structure and behavior, for example to schedule two similar resources. In such cases, the declaration shown above will be preceded by the word **class**, and the separate monitors will be declared to belong to this class:

```
monitor 1, monitor 2: classname;
```

Thus the structure of a class of monitors is identical to that described for a data representation in [13], except for addition of the basic word *monitor*. Brinch-Hansen uses the word *shared* for the same purpose [3].

The procedures of a monitor are common to all running programs, in the sense that any program may at any time attempt to call such a procedure. However, it is essential that only one program at a time actually succeed in entering a monitor procedure, and any subsequent call must be held up until the previous call has been completed. Otherwise, if two procedure bodies were in simultaneous execution, the effects on the local variables of the monitor could be chaotic. The proce-

dures local to a monitor should not access any nonlocal variables other than those local to the same monitor, and these variables of the monitor should be inaccessible from outside the monitor. If these restrictions are imposed, it is possible to guarantee against certain of the more obscure forms of time-dependent coding error; and this guarantee could be underwritten by a visual scan of the text of the program, which could readily be automated in a compiler.

Any dynamic resource allocator will sometimes need to delay a program wishing to acquire a resource which is not currently available, and to resume that program after some other program has released the resource required. We therefore need: a "wait" operation, issued from inside a procedure of the monitor, which causes the calling program to be delayed; and a "signal" operation, also issued from inside a procedure of the same monitor, which causes exactly one of the waiting programs to be resumed immediately. If there are no waiting programs, the signal has no effect. In order to enable other programs to release resources during a wait, a wait operation must relinquish the exclusion which would otherwise prevent entry to the releasing procedure. However, we decree that a signal operation be followed immediately by resumption of a waiting program, without possibility of an intervening procedure call from yet a third program. It is only in this way that a waiting program has an absolute guarantee that it can acquire the resource just released by the signalling program without any danger that a third program will interpose a monitor entry and seize the resource instead.

In many cases, there may be more than one reason for waiting, and these need to be distinguished by both the waiting and the signalling operation. We therefore introduce a new type of "variable" known as a "condition"; and the writer of a monitor should declare a variable of type condition for each reason why a program might have to wait. Then the wait and signal operations should be preceded by the name of the relevant condition variable, separated from it by a dot:

```
condvariable.wait;
condvariable.signal;
```

Note that a condition "variable" is neither true nor false; indeed, it does not have any stored value accessible to the program. In practice, a condition variable will be represented by an (initially empty) queue of processes which are currently waiting on the condition; but this queue is invisible both to waiters and signallers. This design of the condition variable has been deliberately kept as primitive and rudimentary as possible, so that it may be implemented efficiently and used flexibly to achieve a wide variety of effects. There is a great temptation to introduce a more complex synchronization primitive, which may be easier to use for many purposes. We shall resist this temptation for a while.

As the simplest example of a monitor, we will design a scheduling algorithm for a single resource, which is dynamically acquired and released by an unknown number of customer processes by calls on procedures

```
procedure acquire;
procedure release;
```

A variable[1]

```
busy:Boolean
```

determines whether or not the resource is in use. If an attempt is made to acquire the resource when it is busy, the attempting program must be delayed by waiting on a variable

```
nonbusy:condition
```

which is signalled by the next subsequent release. The initial value of busy is false. These design decisions lead to the following code for the monitor:

```
single resource:monitor
begin busy:Boolean;
    nonbusy:condition;
  procedure acquire;
    begin if busy then nonbusy.wait;
          busy := true
    end;
  procedure release;
    begin busy := false;
        nonbusy.signal
    end;
  busy := false; comment initial value;
end single resource
```

Notes

1. In designing a monitor, it seems natural to design the procedure headings, the data, the conditions, and the procedure bodies, in that order. All subsequent examples will be designed in this way.
2. The acquire procedure does not have to retest that busy has gone false when it resumes after its wait, since the release procedure has guaranteed that this is so; and as mentioned before, no other program can intervene between the signal and the continuation of exactly one waiting program.
3. If more than one program is waiting on a condition, we postulate that the signal operation will reactivate the longest waiting program. This gives a simple neutral queuing discipline which ensures that every waiting program will eventually get its turn.
4. The single resource monitor simulates a Boolean semaphore [7] with *acquire* and *release* used for *P* and *V* respectively. This is a simple proof that the monitor/condition concepts are not in principle less powerful than semaphores, and that they can be used for all the same purposes.

[1] As in PASCAL [15], a variable declaration is of the form: ⟨*variable identifier*⟩:⟨*type*⟩;

2. Interpretation

Having proved that semaphores can be implemented by a monitor, the next task is to prove that monitors can be implemented by semaphores.

Obviously, we shall require for each monitor a Boolean semaphore "*mutex*" to ensure that the bodies of the local procedures exclude each other. The semaphore is initialized to 1; a *P*(*mutex*) must be executed on entry to each local procedure, and a *V*(*mutex*) must usually be executed on exit from it.

When a process signals a condition on which another process is waiting, the signalling process must wait until the resumed process permits it to proceed. We therefore introduce for each monitor a second semaphore "*urgent*" (initialized to 0), on which signalling processes suspend themselves by the operation *P*(*urgent*). Before releasing exclusion, each process must test whether any other process is waiting on *urgent*, and if so, must release it instead by a *V*(*urgent*) instruction. We therefore need to count the number of processes waiting on *urgent*, in an integer "*urgentcount*" (initially zero). Thus each exit from a procedure of a monitor should be coded:

if *urgentcount* > 0 **then** *V*(*urgent*) **else** *V*(*mutex*)

Finally, for each condition local to the monitor, we introduce a semaphore "*condsem*" (initialized to 0), on which a process desiring to wait suspends itself by a *P*(*condsem*) operation. Since a process signalling this condition needs to know whether anybody is waiting, we also need a count of the number of waiting processes held in an integer variable "*condcount*" (initially 0). The operation "*cond.wait*" may now be implemented as follows (recall that a waiting program must release exclusion before suspending itself):

```
condcount := condcount + 1;
if urgentcount > 0 then V(urgent) else V(mutex);
P(condsem);
comment This will always wait;
condcount := condcount - 1
```

The signal operation may be coded:

```
urgentcount := urgentcount + 1;
if condcount > 0 then {V(condsem); P(urgent)};
urgentcount := urgentcount - 1
```

In this implementation, possession of the monitor is regarded as a privilege which is explicitly passed from one process to another. Only when no one further wants the privilege is *mutex* finally released.

This solution is not intended to correspond to recommended "style" in the use of semaphores. The concept of a condition-variable is intended as a substitute for semaphores, and has its own style of usage, in the same way that while-loops or coroutines are intended as a substitute for jumps.

In many cases, the generality of this solution is unnecessary, and a significant improvement in efficiency is possible.

1. When a procedure body in a monitor contains no **wait or signal, exit** from the body can be coded by a simple *V*(*mutex*), since *urgentcount* cannot have changed during the execution of the body.
2. If a *cond.signal* is the last operation of a procedure body, it can be combined with monitor exit as follows:

```
    if condcount > 0 then V(condsem)
else if urgentcount > 0 then V(urgent)
                        else V(mutex)
```

3. If there is no other wait or signal in the procedure body, the second line shown above can also be omitted.
4. If *every* signal occurs as the last operation of its procedure body, the variables *urgentcount* and *urgent* can be omitted, together with all operations upon them. This is such a simplification that O-J. Dahl suggests that signals should always be the last operation of a monitor procedure; in fact, this restriction is a very natural one, which has been unwittingly observed in all examples of this paper.

Significant improvements in efficiency may also be obtained by avoiding the use of semaphores, and by implementing conditions directly in hardware, or at the lowest and most uninterruptible level of software (e.g. supervisor mode). In this case, the following optimizations are possible.

1. *urgentcount* and *condcount* can be abolished, since the fact that someone is waiting can be established by examining the representation of the semaphore, which cannot change surreptitiously within noninterruptible mode.
2. Many monitors are very short and contain no calls to other monitors. Such monitors can be executed wholly in noninterruptible mode, using, as it were, the common exclusion mechanism provided by hardware. This will often involve *less* time in noninterruptible mode than the establishment of separate exclusion for each monitor.

I am grateful to J. Bezivin, J. Horning, and R.M. McKeag for assisting in the discovery of this algorithm.

3. Proof Rules

The analogy between a monitor and a data representation has been noted in the introduction. The mutual exclusion on the code of a monitor ensures that procedure calls follow each other in time, just as they do in sequential programming; and the same restrictions are placed on access to nonlocal data. These are the reasons why the same proof rules can be applied to monitors as to data representations.

As with a data representation, the programmer may associate an invariant $\mathcal{I}$ with the local data of a monitor, to describe some condition which will be true of this data before and after every procedure call. $\mathcal{I}$ must also be made true after initialization of the data, and before *every* wait instruction; otherwise the next following procedure call will not find the local data in a state which it expects.

With each condition variable b the programmer may associate an assertion B which describes the condition under which a program waiting on b wishes to be resumed. Since other programs may invoke a monitor procedure during a wait, a waiting program must ensure that the invariant $\mathcal{I}$ for the monitor is true beforehand. This gives the proof rule for waits:

$\mathcal{I}\ \{b.wait\}\ \mathcal{I}\&B$

Since a signal can cause immediate resumption of a waiting program, the conditions $\mathcal{I}\&B$ which are expected by that program must be made true before the signal; and since B may be made false again by the resumed program, only $\mathcal{I}$ may be assumed true afterwards. Thus the proof rule for a signal is:

$\mathcal{I}\&B\{b.signal\}\mathcal{I}$

This exhibits a pleasing symmetry with the rule for waiting.

The introduction of condition variables makes it possible to write monitors subject to the risk of deadly embrace [7]. It is the responsibility of the programmer to avoid this risk, together with other scheduling disasters (thrashing, indefinitely repeated overtaking, etc. [11]). Assertion-oriented proof methods cannot prove absence of such risks; perhaps it is better to use less formal methods for such proofs.

Finally, in many cases an operating system monitor constructs some "virtual" resource which is used in place of actual resources by its "customer" programs. This virtual resource is an abstraction from the set of local variables of the monitor. The program prover should therefore define this abstraction in terms of its concrete representation, and then express the intended effect of each of the procedure bodies in terms of the abstraction. This proof method is described in detail in [13].

4. Example: Bounded Buffer

A bounded buffer is a concrete representation of the abstract idea of a sequence of portions. The sequence is accessible to two programs running in parallel: the first of these (the producer) updates the sequence by appending a new portion x at the end; and the second (the consumer) updates it by removing the first portion. The initial value of the sequence is empty. We thus require two operations:

(1) *append*(*x*:*portion*);

which should be equivalent to the abstract operation

sequence := *sequence* ∩ ⟨*x*⟩;

where ⟨*x*⟩ is the sequence whose only item is x and ∩ denotes concatenation of two sequences.

(2) *remove*(**result** *x*:*portion*);

which should be equivalent to the abstract operations

x := *first*(*sequence*); *sequence* := *rest*(*sequence*);

where *first* selects the first item of a sequence and *rest* denotes the sequence with its first item removed. Obviously, if the sequence is empty, *first* is undefined; and in this case we want to ensure that the consumer waits until the producer has made the sequence nonempty.

We shall assume that the amount of time taken to produce a portion or consume it is large in comparison with the time taken to append or remove it from the sequence. We may therefore be justified in making a design in which producer and consumer can both update the sequence, but not simultaneously.

The sequence is represented by an array:

buffer :**array** 0..*N* − 1 **of** *portion*;

and two variables:

(1) *lastpointer*:0..*N* − 1;

which points to the buffer position into which the next append operation will put a new item, and

(2) *count*:0..*N*;

which always holds the length of the sequence (initially 0).

We define the function

seq (*b*,*l*,*c*) =$_{df}$ **if** *c* = 0 **then** *empty*
else *seq*(*b*,*l*⊖1,*c*−1) ∩⟨*b*[*l*⊖1]⟩

where the circled operations are taken modulo N. Note that if $c \neq 0$,

first(*seq*(*b*,*l*,*c*)) = *b*[*l*⊖*c*]

and

rest(*seq*(*b*,*l*,*c*)) = *seq*(*b*,*l*,*c*−1)

The definition of the abstract sequence in terms of its concrete representation may now be given:

sequence =$_{df}$ *seq*(*buffer*, *lastpointer*, *count*)

Less formally, this may be written

sequence =$_{df}$ ⟨*buffer*[*lastpointer*⊖*count*],
buffer[*lastpointer*⊖*count*⊕1],
. . . ,
buffer[*lastpointer*⊖1]⟩

Another way of conveying this information would be by an example and a picture, which would be even less formal.

The invariant for the monitor is:

$0 \leq count \leq N\ \&\ 0 \leq lastpointer \leq N - 1$

There are two reasons for waiting, which must be represented by condition variables:

nonempty:*condition*;

means that the count is greater than 0, and

nonfull:*condition*;

means that the count is less than N.

With this constructive approach to the design [8], it is relatively easy to code the monitor without error.

```
bounded buffer:monitor
  begin buffer:array 0..N − 1 of portion;
        lastpointer:0..N − 1;
        count:0..N;
        nonempty,nonfull:condition;
    procedure append(x:portion);
      begin if count = N then nonfull.wait;
            note 0 ≤ count < N;
            buffer[lastpointer] := x;
            lastpointer := lastpointer ⊕ 1;
            count := count+1;
            nonempty.signal
      end append;
    procedure remove(result x:portion);
      begin if count = 0 then nonempty.wait;
            note 0 < count ≤ N;
            x := buffer[lastpointer⊖count];
            nonfull.signal
      end remove;
    count := 0; lastpointer := 0;
  end bounded buffer;
```

A formal proof of the correctness of this monitor with respect to the stated abstraction and invariant can be given if desired by techniques described in [13]. However, these techniques seem not capable of dealing with subsequent examples of this paper.

Single-buffered input and output may be regarded as a special case of the bounded buffer with $N = 1$. In this case, the array can be replaced by a single variable, the *lastpointer* is redundant, and we get:

```
iostream:monitor
begin buffer:portion;
      count:0..1;
      nonempty,nonfull:condition;
  procedure append(x:portion);
    begin if count = 1 then nonfull.wait;
      buffer := x;
      count := 1;
      nonempty.signal
    end append;
  procedure remove(result x:portion);
    begin if count = 0 then nonempty.wait;
      x := buffer;
      count := 0;
      nonfull.signal
    end remove;
  count := 0;
end iostream;
```

If physical output is carried out by a separate special purpose channel, then the interrupt from the channel should simulate a call of *iostream.remove*(x); and similarly for physical input, simulating a call of *iostream.append*(x).

5. Scheduled Waits

Up to this point, we have assumed that when more than one program is waiting for the same condition, a signal will cause the longest waiting program to be resumed. This is a good simple scheduling strategy, which precludes indefinite overtaking of a waiting process.

However, in the design of an operating system, there are many cases when such simple scheduling on the basis of first-come-first-served is not adequate. In order to give a closer control over scheduling strategy, we introduce a further feature of a conditional wait, which makes it possible to specify as a parameter of the wait some indication of the priority of the waiting program, e.g.:

```
busy . wait (p);
```

When the condition is signalled, it is the program that specified the lowest value of p that is resumed. In using this facility, the designer of a monitor must take care to avoid the risk of indefinite overtaking; and often it is advisable to make priority a nondecreasing function of the time at which the wait commences.

This introduction of a "scheduled wait" concedes to the temptation to make the condition concept more elaborate. The main justifications are:

1. It has no effect whatsoever on the *logic* of a program, or on the formal proof rules. Any program which works without a scheduled wait will work with it, but possibly with better timing characteristics.
2. The automatic ordering of the queue of waiting processes is a simple fast scheduling technique, except when the queue is exceptionally long—and when it is, central processor time is not the major bottleneck.
3. The maximum amount of storage required is one word per process. Without such a built-in scheduling method, each monitor may have to allocate storage proportional to the number of its customers; the alternative of dynamic storage allocation in small chunks is unattractive at the low level of an operating system where monitors are found.

I shall yield to one further temptation, to introduce a Boolean function of conditions:

```
condname.queue
```

which yields the value true if anyone is waiting on *condname* and false otherwise. This can obviously be easily implemented by a couple of instructions, and affords valuable information which could otherwise be obtained only at the expense of extra storage, time, and trouble.

A trivially simple example is an *alarmclock* monitor, which enables a calling program to delay itself for a stated number n of time-units, or "*ticks*". There are two entries:

```
procedure wakeme (n:integer);
procedure tick;
```

The second of these is invoked by hardware (e.g. an interrupt) at regular intervals, say ten times per second. Local variables are

```
now:integer;
```

which records the current time (initially zero) and

```
wakeup:condition;
```

on which sleeping programs wait. But the *alarmsetting* at which these programs will be aroused is known at the time when they start the wait; and this can be used to determine the correct sequence of waking up.

```
alarmclock:monitor
begin now:integer;
    wakeup:condition;
  procedure wakeme(n:integer);
    begin alarmsetting:integer;
      alarmsetting := now + n;
      while now < alarmsetting do wakeup.wait(alarmsetting);
      wakeup.signal;
      comment In case the next process is due to wake up at the
      same time;
    end;
  procedure tick;
    begin now := now + 1;
      wakeup.signal
    end;
  now := 0
end alarmclock
```

In the program given above, the next candidate for wakening is actually woken at every tick of the clock. This will not matter if the frequency of ticking is low enough, and the overhead of an accepted signal is not too high.

I am grateful to A. Ballard and J. Horning for posing this problem.

6. Further Examples

In proposing a new feature for a high level language it is very difficult to make a convincing case that the feature will be both easy to use efficiently and easy to implement efficiently. Quality of implementation can be proved by a single good example, but ease and efficiency of use require a great number of realistic examples; otherwise it can appear that the new feature has been specially designed to suit the examples, or vice versa. This section contains a number of additional examples of solutions of familiar problems. Further examples may be found in [14].

6.1 Buffer Allocation

The bounded buffer described in Section 4 was designed to be suitable only for sequences with small portions, for example, message queues. If the buffers contain high volume information (for example, files for pseudo offline input and output), the bounded buffer may still be used to store the *addresses* of the buffers which are being used to hold the information. In this way, the producer can be filling one buffer while the consumer is emptying another buffer of the same sequence. But this requires an allocator for dynamic acquisition and relinquishment of *buffer addresses*. These may be declared as a type

```
type bufferaddress = 1..B;
```

where B is the number of buffers available for allocation.

The buffer allocator has two entries:

```
procedure acquire (result b:bufferaddress);
```

which delivers a free *buffer address* b; and

```
procedure release(b:bufferaddress);
```

which returns a *bufferaddress* when it is no longer required. In order to keep a record of free buffer addresses the monitor will need:

```
freepool:powerset bufferaddress;
```

which uses the *PASCAL* powerset facility to define a variable whose values range over all sets of *buffer addresses*, from the empty set to the set containing all *buffer addresses*. It should be implemented as a *bitmap* of B consecutive bits, where the ith bit is 1 if and only if i is in the set. There is only one condition variable needed:

```
nonempty:condition
```

which means that *freepool* $\neq$ *empty*. The code for the allocator is:

```
buffer allocator:monitor
begin freepool:powerset bufferaddress;
    nonempty:condition;
  procedure acquire (result b:bufferaddress);
    begin if freepool = empty then nonempty.wait;
      b := first(freepool);
      comment Any one would do;
      freepool := freepool - {b};
      comment Set subtraction;
    end acquire;
  procedure release(b:bufferaddress);
    begin freepool := freepool - {b};
      nonempty.signal
    end release;
  freepool := all buffer addresses
end buffer allocator
```

The action of a producer and consumer may be summarized:

```
producer: begin b:bufferaddress; . . .
            while not finished do
              begin bufferallocator.acquire(b);
                . . . fill buffer b . . .;
                bounded buffer.append(b)
              end; . . .
          end producer;
consumer: begin b:bufferaddress; . . .
            while not finished do
              begin bounded buffer.remove(b);
                . . . empty buffer b . . .;
                buffer allocator.release(b)
              end; . . .
          end consumer;
```

This buffer allocator would appear to be usable to share the buffers among several streams, each with its own producer and its own consumer, and its own instance of a bounded buffer monitor. Unfortunately,

when the streams operate at widely varying speeds, and when the freepool is empty, the scheduling algorithm can exhibit persistent undesirable behavior. If two producers are competing for each buffer as it becomes free, a first-come-first-served discipline of allocation will ensure (apparently fairly) that each gets alternate buffers; and they will consequently begin to produce at equal speeds. But if one consumer is a 1000 lines/min printer and the other is a 10 lines/min teletype, the faster consumer will be eventually reduced to the speed of the slower, since it cannot forever go faster than its producer. At this stage nearly all buffers will belong to the slower stream, so the situation could take a long time to clear.

A solution to this is to use a scheduled wait, to ensure that in heavy load conditions the available buffers will be shared reasonably fairly between the streams that are competing for them. Of course, inactive streams need not be considered, and streams for which the consumer is currently faster than the producer will never ask for more than two buffers anyway. In order to achieve fairness in allocation, it is sufficient to allocate a newly freed buffer to that one among the competing producers whose stream currently owns fewest buffers. Thus the system will seek a point as far away from the undesirable extreme as possible.

For this reason, the entries to the allocator should indicate for what stream the buffer is to be (or has been) used, and the allocator must keep a count of the current allocation to each stream in an array:

```
count:array stream of integer;
```

The new version of the allocator is:

```
bufferallocator:monitor
  begin freepool:powerset bufferaddress;
      nonempty:condition
      count:array stream of integer;
    procedure acquire(result b:bufferaddress; s:stream);
      begin if freepool = empty then nonempty.wait(count[s]);
        count[s] := count[s] + 1;
        b := first(freepool);
        freepool := freepool - {b}
      end acquire;
    procedure release(b:bufferaddress; s:stream)
      begin count[s] := count[s] - 1;
        freepool := freepool - {b};
        nonempty·signal
      end;
    freepool := all buffer addresses;
    for s:stream do count[s] := 0
  end bufferallocator
```

Of course, if a consumer stops altogether, perhaps owing to mechanical failure, the producer must also be halted before it has acquired too many buffers, even if no one else currently wants them. This can perhaps be most easily accomplished by appropriate fixing of the size of the bounded buffer for that stream and/or by ensuring that at least two buffers are reserved for each stream, even when inactive. It is an interesting comment on dynamic resource allocation that, as soon as resources are heavily loaded, the system must be designed to fall back toward a more static regime.

I am grateful to E.W. Dijkstra for pointing out this problem and its solution [10].

6.2 Disk Head Scheduler

On a moving head disk, the time taken to move the heads increases monotonically with the distance traveled. If several programs wish to move the heads, the average waiting time can be reduced by selecting, first, the program which wishes to move them the shortest distance. But unfortunately this policy is subject to an instability, since a program wishing to access a cylinder at one edge of the disk can be indefinitely overtaken by programs operating at the other edge or the middle.

A solution to this is to minimize the frequency of change of direction of movement of the heads. At any time, the heads are kept moving in a given direction, and they service the program requesting the nearest cylinder in that direction. If there is no such request, the direction changes, and the heads make another sweep across the surface of the disk. This may be called the "elevator" algorithm, since it simulates the behavior of a lift in a multi-storey building.

There are two entries to a disk head scheduler:

(1) *request*(*dest*:*cylinder*);

where

```
type cylinder = 0..cylmax;
```

which is entered by a program just *before* issuing the instruction to move the heads to cylinder *dest*.

(2) *release*;

which is entered by a program when it has made all the transfers it needs on the current cylinder.

The local data of the monitor must include a record of the current headposition, *headpos*, the current direction of *sweep*, and whether the disk is *busy*:

```
headpos:cylinder;
direction:(up, down);
busy:Boolean
```

We need two conditions, one for requests waiting for an *upsweep* and the other for requests waiting for a *downsweep*:

```
upsweep, downsweep:condition
```

```
dischead:monitor
begin headpos:cylinder;
    direction:(up, down);
    busy:Boolean;
    upsweep,downsweep:condition;
  procedure request(dest:cylinder);
    begin if busy then
      {if headpos < dest ∨ headpos = dest & direction = up
        then upsweep·wait(dest)
        else downsweep.wait(cylmax-dest)};
      busy := true; headpos := dest
    end request;
```

```
  procedure release;
    begin busy := false;
      if direction = up then
        {if upsweep.queue then upsweep.signal
                          else {direction := down;
                                downsweep.signal}}
      else if downsweep.queue then downsweep.signal
                              else {direction := up;
                                    upsweep.signal}
    end release;
    headpos := 0; direction := up; busy := false
end dischead;
```

6.3 Readers and Writers

As a more significant example, we take a problem which arises in on-line real-time applications such as airspace control. Suppose that each aircraft is represented by a record, and that this record is kept up to date by a number of "writer" processes and accessed by a number of "reader" processes. Any number of "reader" processes may simultaneously access the same record, but obviously any process which is updating (writing) the individual components of the record must have exclusive access to it, or chaos will ensue. Thus we need a class of monitors; an instance of this class local to *each* individual aircraft record will enforce the required discipline for that record. If there are many aircraft, there is a strong motivation for minimizing local data of the monitor; and if each read or write operation is brief, we should also minimize the time taken by each monitor entry.

When many readers are interested in a single aircraft record, there is a danger that a writer will be indefinitely prevented from keeping that record up to date. We therefore decide that a new reader should not be permitted to start if there is a writer waiting. Similarly, to avoid the danger of indefinite exclusion of readers, all readers waiting at the end of a write should have priority over the next writer. Note that this is a very different scheduling rule from that propounded in [4], and does not seem to require such subtlety in implementation. Nevertheless, it may be more suited to this kind of application, where it is better to read stale information than to wait indefinitely!

The monitor obviously requires four local procedures:

startread	entered by reader who wishes to read.
endread	entered by reader who has finished reading.
startwrite	entered by writer who wishes to write.
endwrite	entered by writer who has finished writing.

We need to keep a count of the number of users who are reading, so that the last reader to finish will know this fact:

```
readercount:integer
```

We also need a *Boolean* to indicate that someone is actually writing:

```
busy:Boolean;
```

We introduce separate conditions for readers and writers to wait on:

```
OKtoread, OKtowrite:condition;
```

The following annotation is relevant:

```
OKtoread ≡ ¬ busy
OKtowrite ≡ ¬ busy & readercount = 0
invariant: busy ⟹ readercount = 0
```

```
class readers and writers:monitor
  begin readercount:integer;
      busy:Boolean;
      OKtoread, OKtowrite:condition;
    procedure startread;
      begin if busy ∨ OKtowrite.queue then OKtoread.wait;
        readercount := readercount + 1;
        OKtoread.signal;
        comment Once one reader can start, they all can;
      end startread;
    procedure endread;
      begin readercount := readercount − 1;
        if readercount = 0 then OKtowrite.signal
      end endread;
    procedure startwrite;
      begin
        if readercount ≠ 0 ∨ busy then OKtowrite.wait
        busy := true
      end startwrite;
    procedure endwrite;
      begin busy := false;
        if OKtoread.queue then OKtoread.signal
                          else OKtowrite.signal
      end endwrite;
    readercount := 0;
    busy := false;
  end readers and writers;
```

I am grateful to Dave Gorman for assisting in the discovery of this solution.

7. Conclusion

This paper suggests that an appropriate structure for a module of an operating system, which schedules resources for parallel user processes, is very similar to that of a data representation used by a sequential program. However, in the case of monitors, the bodies of the procedures must be protected against re-entrance by being implemented as critical regions. The textual grouping of critical regions together with the data which they update seems much superior to critical regions scattered through the user program, as described in [7, 12]. It also corresponds to the traditional practice of the writers of operating system supervisors. It can be recommended without reservation.

However, it is much more difficult to be confident about the condition concept as a synchronizing primitive. The synchronizing facility which is easiest to use is probably the conditional *wait* [2, 12]:

```
wait(B);
```

where B is a general Boolean expression (it causes the given process to wait until B becomes true); but this may be too inefficient for general use in operating systems,

because its implementation requires re-evaluation of the expression B after every exit from a procedure of the monitor. The condition variable gives the programmer better control over efficiency and over scheduling; it was designed to be very primitive, and to have a simple proof rule. But perhaps some other compromise between convenience and efficiency might be better. The question whether the signal should always be the last operation of a monitor procedure is still open. These problems will be studied in the design and implementation of a pilot project operating system, currently enjoying the support of the Science Research Council of Great Britain.

Another question which will be studied will be that of the disjointness of monitors: Is it possible to design a separate isolated monitor for each kind of resource, so that it will make sensible scheduling decisions for that resource, using only the minimal information about the utilization of that resource, and using no information about the utilization of any resource administered by other monitors? In principle, it would seem that, when more knowledge of the status of the entire system is available, it should be easier to take decisions nearer to optimality. Furthermore, in principle, independent scheduling of different kinds of resource can lead to deadly embrace. These considerations would lead to the design of a traditional "monolithic" monitor, maintaining large system tables, all of which can be accessed and updated by any of the procedures of the monitor.

There is no a priori reason why the attempt to split the functions of an operating system into a number of isolated disjoint monitors should succeed. It can be made to succeed only by discovering and implementing good scheduling algorithms in each monitor. In order to avoid undesirable interactions between the separate scheduling algorithms, it appears necessary to observe the following principles:

1. Never seek to make an optimal decision; merely seek to avoid persistently pessimal decisions.
2. Do not seek to present the user with a virtual machine which is better than the actual hardware; merely seek to pass on the speed, size, and flat unopiniated structure of a simple hardware design.
3. Use preemptive techniques in preference to non-preemptive ones where possible.
4. Use "grain of time" [9] methods to secure independence of scheduling strategies.
5. Keep a low variance (as well as a low mean) on waiting times.
6. Avoid fixed priorities; instead, try to ensure that every program in the system makes reasonably steady progress. In particular, avoid indefinite overtaking.
7. Ensure that when demand for resources outstrips the supply (i.e. in overload conditions), the behavior of the scheduler is satisfactory (i.e. thrashing is avoided).
8. Make rules for the correct and sensible use of monitor calls, and assume that user programs will obey them. Any checking which is necessary should be done not by a central shared monitor, but rather by an algorithm (called "user envelope") which is local to each process executing a user program. This algorithm should be implemented at least partially in the hardware (e.g. base and range registers, address translation mechanisms, capabilities, etc.).

It is the possibility of constructing separate monitors for different purposes, and of separating the scheduling decisions embodied in monitors from the checking embodied in user envelopes, that may justify a hope that monitors are an appropriate concept for the structuring of an operating system.

Acknowledgments. The development of the monitor concept is due to frequent discussions and communications with E.W. Dijkstra and P. Brinch-Hansen. A monitor corresponds to the "secretary" described in [9], and is also described in [1, 3].

Acknowledgment is also due to the support of IFIP WG.2.3., which provides a meeting place at which these and many other ideas have been germinated, fostered, and tested.

Received February 1973; revised April 1974

References

1. Brinch-Hansen, P. Structured multiprogramming. *Comm. ACM 15*, 7 (July 1972), 574–577.
2. Brinch-Hansen, P. "A comparison of two synchronizing concepts," *Acta Information 1* (1972), 190–199.
3. Brinch-Hansen, P. *Operating System Principles*. Prentice-Hall, Englewood Cliffs, N.J., 1973.
4. Courtois, P. J., Heymans, F., Parnas, D.L. Concurrent control with readers and writers. *Comm. ACM 14*, 10 (Oct. 1971), 667–668.
5. Courtois, P.J., Heymans, F., Parnas, D.L. Comments on [2]. *Acta Informatica 1* (1972), 375–376.
6. Dahl, O.J. Hierarchical program structures. In *Structured Programming*, Academic Press, New York, 1972.
7. Dijkstra, E.W. Cooperating Sequential Processes. In *Programming Languages* (Ed. F. Genuys), Academic Press, New York, 1968.
8. Dijkstra, E.W. A constructive approach to the problem of program correctness. *BIT 8* (1968), 174–186.
9. Dijkstra, E.W. Hierarchical ordering of sequential processes. In *Operating Systems Techniques*, Academic Press, New York, 1972.
10. Dijkstra, E.W. Information streams sharing a finite buffer. *Information Processing Letters 1*, 5 (Oct. 1972), 179–180.
11. Dijkstra, E.W. A class of allocation strategies inducing bounded delays only. Proc AFIPS 1972 SJCC, Vol. 40, AFIPS Press, Montvale, N.J., pp. 933–936.
12. Hoare, C.A.R. Towards a theory of parallel programming. In *Operating Systems Techniques*, Academic Press, New York, 1972.
13. Hoare, C.A.R. Proof of correctness of data representations. *Acta Informatica* 1 (1972), 271–281.
14. Hoare, C.A.R. A structured paging system. *Computer J. 16*, 3 (1973), 209–215.
15. Wirth, N. The programming language PASCAL. *Acta Informatica 1*, 1 (1971), 35–63.

IN3030 Uke 8, v2019

Eric Jul
PSE,
Inst. for informatikk

Hva har vi sett på i uke 7

1. Piazza – please use it
2. Mere om synkronisering og kokkene og kelnere – demo ved tavle ☺
3. Erastostenes sil review
4. Hvilken orden O() har Eratosthenes Sil ?
5. Faktorisering – med eksempler
6. Hvordan lage en parallell løsning – ulike måter å synkronisere skriving på felles variable – med eksempel
7. Ulike strategier for å dele opp et problem for parallellisering

Hva skal vi se på i uke 8

- Perspektiv: Utvikling i CPU-speed, minne, nettverk, herunder effekten av Moore's Law
- Et sitat om tidsforbruk ved faktorisering
- Om en feil i Java 7 (også i Java 8)ved tidtaking
- Hvordan parallellisere rekursive algoritmer
- Gå ikke i 'direkte oversettelses-fella'
  - eksemplifisert ved Kvikk-sort, 3 ulike løsninger
- Hvor lang tid tar de ulike mekanismene vi har i Java 6 og 8?

Perspektiv: Utvikling i CPU, minne, nettverk og mere om Moore's law

- 1980 CPU: Intel 8080 8-bit processor
  - «int»: one byte
  - «long»: two bytes
  - CPU clock frequency: 1 MHz
  - Memory: 64 kilobytes
  - Data transmission 1kbit/s == 125 bytes/s
- A look at Moore's law

Newer Machines

- 2018: Intel i7
  - up-to 3.2 GHz – 8x cores
  - 8 Gbyte Memory
  - 1 Gbit/s network

Performance 1980 vs 2018

- 2018: Intel i7
  - 8 x 3.2 GHz vs. 1 MHz ~ factor 25,600
  - 8 Gbyte Memory vs. 64 kbytes ~ factor 125,000
  - 1 Gbit/s network vs 1 kbit/s ~ factor 1,000,000

- ☺

What is the only thing that has NOT changed?

- Ping: sending an empty message from Oslo to Seattle
- How long does this take?

Round-trip ping Time

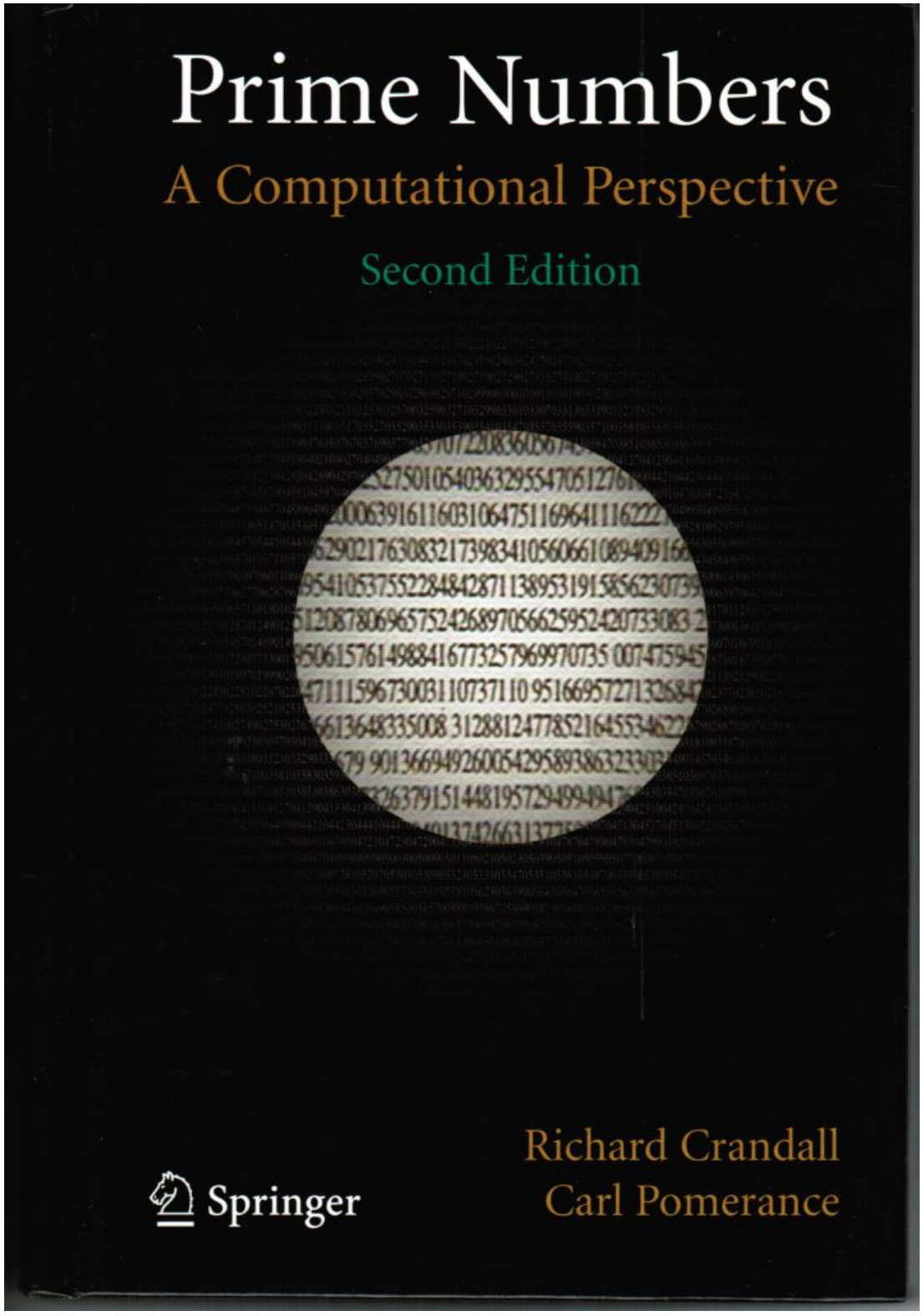
- 2019: Approximately 180-200 ms
- 1988: How much slower?
  - Same time?
  - 10x longer?
  - 100x longer?
  - 1000x longer?
  - 10,000x longer?

Latency has not changed: limited by the *incredibly* slow

- How fast is the speed of light?
- When a 3 GHz core executes one cycle, how far does light travel?
- How far does light travel in 1 ns?

Sitat fra denne boka (fra 2005) om hvor lang tid det tar å finne ut om et 19-sifret tall er primtall ved divisjon.
Denne boka hevder 1 døgn, vi skriver program som gjør det på ca. 1-3 sek – eller : 24*60*60 = 86 400 x fortere

Og selv om boka deler med all oddetall $<\sqrt{N}$ og ikke bare primtallene $<\sqrt{N}$, skulle det bare gå ca. 10x langsommere (fordi ca 5-10% av alle oddetall $<\sqrt{N}$, er primtall).



3.1.3 Practical considerations

It is perfectly reasonable to use trial division as a primality test when n is not too large. Of course, "too large" is a subjective quality; such judgment depends on the speed of the computing equipment and how much time you are willing to allow a computer to run. It also makes a difference whether there is just the occasional number you are interested in, as opposed to the possibility of calling trial division repeatedly as a subroutine in another algorithm. On a modern workstation, and very roughly speaking, numbers that can be proved prime via trial division in one minute do not exceed 13 decimal digits. In one day of current workstation time, perhaps a 19-digit number can be resolved. (Although these sorts of rules of thumb scale, naturally, according to machine performance in any given era.)

Minner om at vi kan greie å faktorisere 19-sifrete tall på < 0.3 sek. sekvensielt !
IKKE på en dag !

AntTraader:16, AntKjerner:8 , tider i millisec for 100 elementer

	n	sekv	para	Speedup	sekv/elm	para/elm
EratosthesSil	2000000000	6626.54	4618.58	1.43		
Faktorisering	2000000000	18906.73	10973.91	1.72	189.0673	109.7391
Total tid	2000000000	**25533.31**	15568.38	1.64		
EratosthesSil	200000000	419.42	186.86	2.24		
Faktorisering	200000000	4621.13	2572.63	1.80	46.2113	25.7263
Total tid	200000000	5077.76	2745.52	1.85		
EratosthesSil	20000000	22.90	17.78	1.29		
Faktorisering	20000000	496.74	335.88	1.48	4.9674	3.3588
Total tid	20000000	519.54	353.66	1.47		
EratosthesSil	2000000	1.97	2.01	0.98		
Faktorisering	2000000	77.35	70.68	1.09	0.7735	0.7068
Total tid	2000000	79.37	72.54	1.09		
EratosthesSil	200000	0.23	0.21	1.09		
Faktorisering	200000	8.32	18.93	0.44	0.0832	0.1893
Total tid	200000	8.54	19.14	0.45		

Vi finner også store primtall

```
para: 3999999999999972 = 2*2*3*199*16750418760469
para: 3999999999999973 = 3999999999999973
para: 3999999999999974 = 2*11*11*149*613*1809663731
para: 3999999999999975 = 3*5*5*13*53*754717*1025641
para: 3999999999999976 = 2*2*2*499999999999997
para: 3999999999999977 = 7*571285714285711
para: 3999999999999978 = 2*3*3*67*1229*131321*205507
para: 3999999999999979 = 17*631*60539*61595143
para: 3999999999999980 = 2*2*5*109*18348623853211
para: 3999999999999981 = 3*331*487*82714525291
para: 3999999999999982 = 2*73281367*272920673
para: 3999999999999983 = 3999999999999983
para: 3999999999999984 = 2*2*2*2*3*7*19*23*739*1187*310559
```

De to store 19 sifrete tallene her 9..973 og 9...983, multipliserer vi dem sammen får vi et 38 sifret tall. Slike meget store tall (128 bit) = to store primtall multipliert med hverandre brukes som krypteringsnøkler.

2) Merkelig feil (?) i Java 7 (også i Java 8) tidtaking ?

- Av og til blir kjøretidene == 0 – hvorfor ?
- Eller veldig få nanoSekunder – hvorfor ?

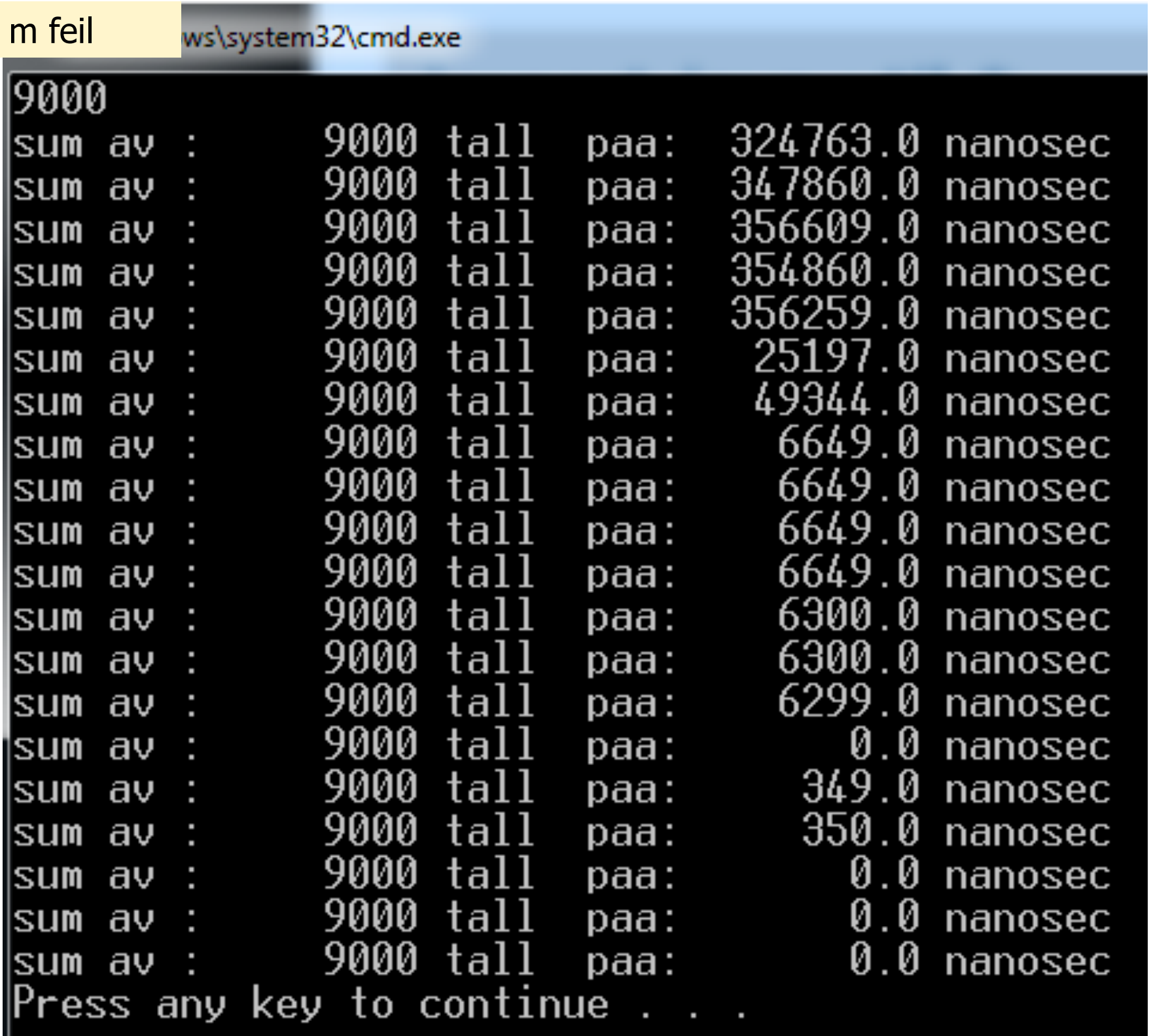
```
import java.util.*;

class FinnSumFeil{
  public static void main(String[] args){
      if (args.length != 1) {
            System.out.println ("use: >java  FinnSumFeil n" );
    }else {
          int len = new Integer(args[0]).intValue();
          FinnSumFeil fs =new FinnSumFeil();

          for (int k = 0; k < 20; k++){
                int [] arr = new int[len];
                for (int i = 0; i < arr.length; i++){ arr[i] = i;   }

                long start  = System.nanoTime();
                long sum = fs.summer(arr);
                long timeTaken = System.nanoTime() - start ;

                // System.out.printf(", SUM: %12d",  sum );
                System.out.printf(" %2d) sum av:%9d tall paa:%10d nanosec%n",
                     (k+1), len,timeTaken);
   }}
} // end main

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

Kjøres (java7):
>java FinnSumFeil n

og kjører man dette med n=9000 med Java 7
ser du de ulike skrittene i JIT-kompilering og feil.

Fjern kommentaren på den første printf-setninga og alt blir OK.

Java 7 m feil


ws\system32\cmd.exe
9000
sum av : 9000 tall paa: 324763.0 nanosec
sum av : 9000 tall paa: 347860.0 nanosec
sum av : 9000 tall paa: 356609.0 nanosec
sum av : 9000 tall paa: 354860.0 nanosec
sum av : 9000 tall paa: 356259.0 nanosec
sum av : 9000 tall paa: 25197.0 nanosec
sum av : 9000 tall paa: 49344.0 nanosec
sum av : 9000 tall paa: 6649.0 nanosec
sum av : 9000 tall paa: 6649.0 nanosec
sum av : 9000 tall paa: 6649.0 nanosec
sum av : 9000 tall paa: 6649.0 nanosec
sum av : 9000 tall paa: 6300.0 nanosec
sum av : 9000 tall paa: 6300.0 nanosec
sum av : 9000 tall paa: 6299.0 nanosec
sum av : 9000 tall paa: 0.0 nanosec
sum av : 9000 tall paa: 349.0 nanosec
sum av : 9000 tall paa: 350.0 nanosec
sum av : 9000 tall paa: 0.0 nanosec
sum av : 9000 tall paa: 0.0 nanosec
sum av : 9000 tall paa: 0.0 nanosec
Press any key to continue . . .

Java 7 feilen

Dear Java Developer,

Thank you for reporting this issue.

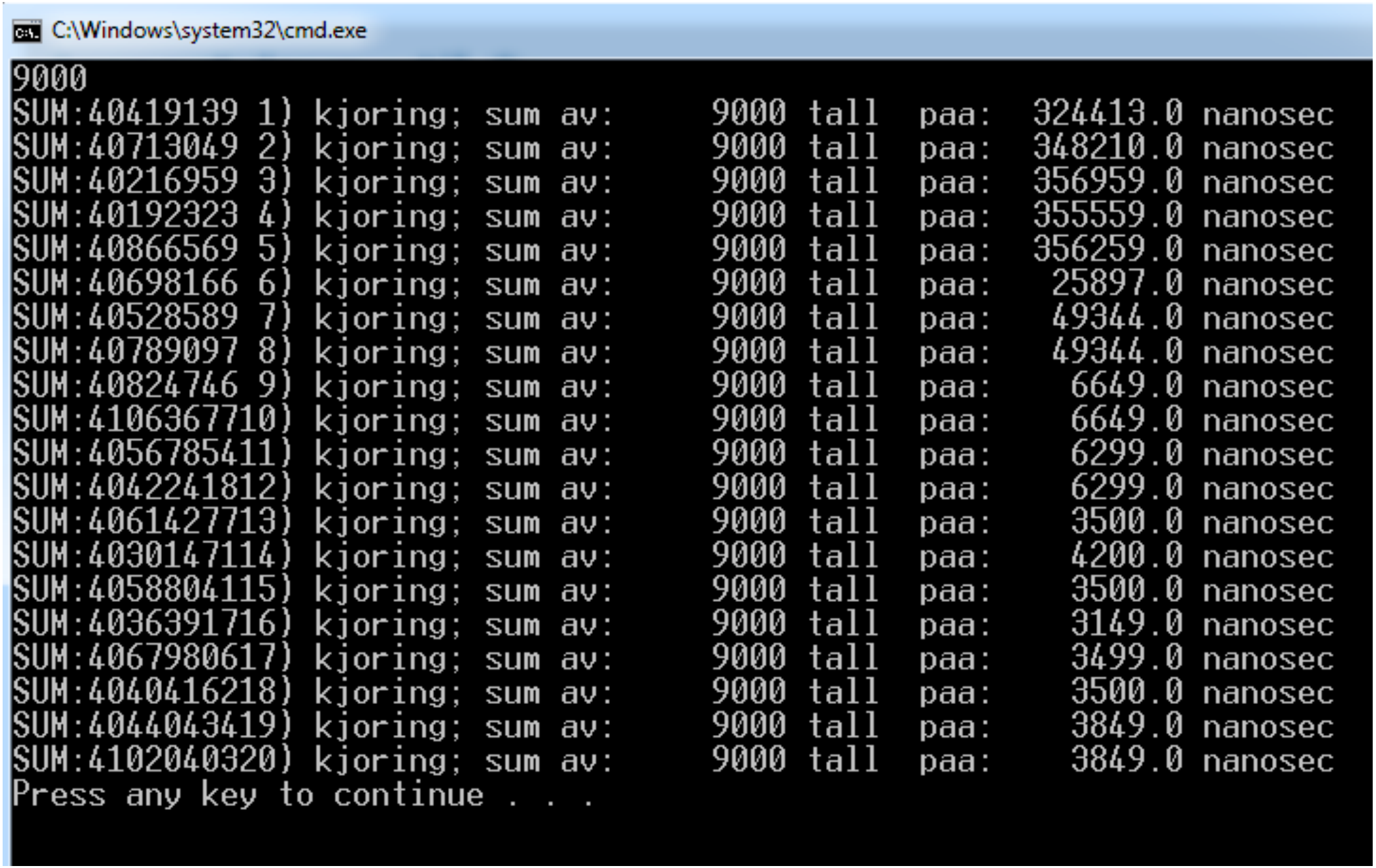
We have determined that **this report is a new bug** and have entered the bug into our bug tracking system under Bug Id: 9006037. You can look for related issues on the Java Bug Database at http://bugs.sun.com.

We will try to process all newly posted bugs in a timely manner, but we make no promises about the amount of time in which a bug will be fixed. If you just reported a bug that could have a major impact on your project, consider using one of the technical support offerings available at Oracle Support.

Thanks again for your submission!

Regards,
Java Developer Support

Java 8 (2015) uten utskrift
av sum – OK , n= 9000



```
import java.util.*;
import easyIO.*;
class FinnSumFeil{
  public static void main(String[] args){
      if (args.length != 1) {
           System.out.println ("use: >java  FinnSumFeil n" );
    }else {
         int len = new Integer(args[0]).intValue();
         FinnSumFeil fs =new FinnSumFeil();

         for (int k = 0; k < 20; k++){
              int [] arr = new int[len];
              for (int i = 0; i < arr.length; i++){ arr[i] = i;   }

              long start  = System.nanoTime();
              long sum = fs.summer(arr);
              long timeTaken = System.nanoTime() - start ;

              // System.out.printf(", SUM: %12d",  sum );
              System.out.printf(" %2d) sum av:%9d tall paa:%10d nanosec%n",
                   (k+1), len,timeTaken);
   }}
} // end main

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

Java 8 kjøres:
>java FinnSumFeil n

og kjører man dette med n= først n= 9000 og n= 1800000 ser du de ulike skrittene i JIT-kompilering og feil.

Fjern kommentaren på den første printf-setninga og alt blir OK.

Java 8 (2017) uten utskrift av sum
n = 9 000

Problemet løst ??

```
M:\INF2440Para\FinnSumFeil>java FinnSumFeil 9000
 1) sum of:     9000 numbers  on :    100047 nanosec
 2) sum of:     9000 numbers  on :     98244 nanosec
 3) sum of:     9000 numbers  on :    115369 nanosec
 4) sum of:     9000 numbers  on :    101549 nanosec
 5) sum of:     9000 numbers  on :    100948 nanosec
 6) sum of:     9000 numbers  on :    101550 nanosec
 7) sum of:     9000 numbers  on :    101249 nanosec
 8) sum of:     9000 numbers  on :    101249 nanosec
 9) sum of:     9000 numbers  on :     17726 nanosec
10) sum of:     9000 numbers  on :     17726 nanosec
11) sum of:     9000 numbers  on :     17726 nanosec
12) sum of:     9000 numbers  on :     35753 nanosec
13) sum of:     9000 numbers  on :     17726 nanosec
14) sum of:     9000 numbers  on :     17726 nanosec
15) sum of:     9000 numbers  on :     51976 nanosec
16) sum of:     9000 numbers  on :     15923 nanosec
17) sum of:     9000 numbers  on :      3305 nanosec
18) sum of:     9000 numbers  on :      5107 nanosec
19) sum of:     9000 numbers  on :      2704 nanosec
20) sum of:     9000 numbers  on :      2703 nanosec
```

Java 8 (2017) uten utskrift av sum
n = 1 800 000:

```
M:\INF2440Para\Powerpoint\Uke8\>java FinnSumError 1800000
 1) sum of:  1800000 numbers  on :   2778189 nanosec
 2) sum of:  1800000 numbers  on :   1943261 nanosec
 3) sum of:  1800000 numbers  on :         0 nanosec
 4) sum of:  1800000 numbers  on :    575648 nanosec
 5) sum of:  1800000 numbers  on :         0 nanosec
 6) sum of:  1800000 numbers  on :         0 nanosec
 7) sum of:  1800000 numbers  on :         0 nanosec
 8) sum of:  1800000 numbers  on :         0 nanosec
 9) sum of:  1800000 numbers  on :         0 nanosec
10) sum of:  1800000 numbers  on :       301 nanosec
11) sum of:  1800000 numbers  on :         0 nanosec
12) sum of:  1800000 numbers  on :         0 nanosec
13) sum of:  1800000 numbers  on :         0 nanosec
14) sum of:  1800000 numbers  on :         0 nanosec
15) sum of:  1800000 numbers  on :         0 nanosec
16) sum of:  1800000 numbers  on :         0 nanosec
17) sum of:  1800000 numbers  on :       301 nanosec
18) sum of:  1800000 numbers  on :       301 nanosec
19) sum of:  1800000 numbers  on :         0 nanosec
20) sum of:  1800000 numbers  on :         0 nanosec
```

Java 8 **med utskrift av SUM** (2017):

```
M:\INF2440Para\Powerpoint\Uke8\FinnSumFeil>java FinnSumError 1800000
, SUM: 1619999100000  1) sum of:  1800000 numbers  on :   2589511 nanosec
, SUM: 1619999100000  2) sum of:  1800000 numbers  on :   2028886 nanosec
, SUM: 1619999100000  3) sum of:  1800000 numbers  on :    532683 nanosec
, SUM: 1619999100000  4) sum of:  1800000 numbers  on :    514356 nanosec
, SUM: 1619999100000  5) sum of:  1800000 numbers  on :    523070 nanosec
, SUM: 1619999100000  6) sum of:  1800000 numbers  on :    527276 nanosec
, SUM: 1619999100000  7) sum of:  1800000 numbers  on :    634534 nanosec
, SUM: 1619999100000  8) sum of:  1800000 numbers  on :    740591 nanosec
, SUM: 1619999100000  9) sum of:  1800000 numbers  on :    647153 nanosec
, SUM: 1619999100000 10) sum of:  1800000 numbers  on :    563330 nanosec
, SUM: 1619999100000 11) sum of:  1800000 numbers  on :    600884 nanosec
, SUM: 1619999100000 12) sum of:  1800000 numbers  on :    605692 nanosec
, SUM: 1619999100000 13) sum of:  1800000 numbers  on :    693120 nanosec
, SUM: 1619999100000 14) sum of:  1800000 numbers  on :    850552 nanosec
, SUM: 1619999100000 15) sum of:  1800000 numbers  on :    538692 nanosec
, SUM: 1619999100000 16) sum of:  1800000 numbers  on :    541697 nanosec
, SUM: 1619999100000 17) sum of:  1800000 numbers  on :    662475 nanosec
, SUM: 1619999100000 18) sum of:  1800000 numbers  on :    636938 nanosec
, SUM: 1619999100000 19) sum of:  1800000 numbers  on :    552212 nanosec
, SUM: 1619999100000 20) sum of:  1800000 numbers  on :    687112 nanoseco
```

```
import java.util.*;

class FinnSumFeil{
  public static void main(String[] args){
      if (args.length != 1) {
            System.out.println ("use: >java  FinnSumFeil n" );
    }else {
          int len = new Integer(args[0]).intValue();
          FinnSumFeil fs =new FinnSumFeil();

          for (int k = 0; k < 20; k++){
                int [] arr = new int[len];
                for (int i = 0; i < arr.length; i++){ arr[i] = i;   }

                long start  = System.nanoTime();
                long sum = fs.summer(arr);
                long timeTaken = System.nanoTime() - start ;

                // System.out.printf(", SUM: %12d",  sum );
                System.out.printf(" %2d) sum av:%9d tall paa:%10d nanosec%n",
                     (k+1), len,timeTaken);
   }}
} // end main

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

kjøres:
>java FinnSumFeil n

og kjører man dette med n=1800000
ser du de ulike skrittene i JIT-kompilering og feil.

Fjern kommentaren på den første printf-setninga og alt blir OK.

Java 8 med interpretert bytekode
(uten sum skrevet ut) - OK

```
M:\INF2440Para\FinnSumFeil>java -Xint FinnSumError 1800000
 1) sum of:  1800000 numbers  on :  18404490 nanosec
 2) sum of:  1800000 numbers  on :  17398008 nanosec
 3) sum of:  1800000 numbers  on :  17483333 nanosec
 4) sum of:  1800000 numbers  on :  17422344 nanosec
 5) sum of:  1800000 numbers  on :  17262808 nanosec
 6) sum of:  1800000 numbers  on :  17439469 nanosec
 7) sum of:  1800000 numbers  on :  17648576 nanosec
 8) sum of:  1800000 numbers  on :  17532606 nanosec
 9) sum of:  1800000 numbers  on :  19139071 nanosec
10) sum of:  1800000 numbers  on :  18521061 nanosec
11) sum of:  1800000 numbers  on :  20507885 nanosec
12) sum of:  1800000 numbers  on :  18735276 nanosec
13) sum of:  1800000 numbers  on :  17826138 nanosec
14) sum of:  1800000 numbers  on :  18006705 nanosec
15) sum of:  1800000 numbers  on :  17587587 nanosec
16) sum of:  1800000 numbers  on :  17801803 nanosec
17) sum of:  1800000 numbers  on :  17911764 nanosec
18) sum of:  1800000 numbers  on :  17501661 nanosec
19) sum of:  1800000 numbers  on :  17815322 nanosec
20) sum of:  1800000 numbers  on :  17526898 nanosec
```

Konklusjon om optimaliseringsfeil

- Sannsynligvis gjøres følgende feil:
  - sum trenges ikke/brukes ikke, og fjernes
  - For å optimaliser mer flyttes kallet på summer()-metoden over kallet på de to kallene på System.nanoTime()
- Riktig går det når 'sum' (og tiden) brukes i utskrift, og da byttes det ikke om på kallene.
  - Fjern kommentaren i utskriften, dvs:

```
long start  = System.nanoTime();
long sum = fs.summer(arr);
long timeTaken = System.nanoTime() - start ;

System.out.printf(", SUM: %12d",  sum );
```

- Dette er 'eneste' grepet jeg har funnet hittil for å unngå denne feilen i tillegg til :
  - >java –Xint Mittprogram….
- Følgelig ikke rettet riktig i Java 8

2) Hvordan ikke gå i den rekursive fella!

- Vi skal nå gå gjennom hvordan vi behandler rekursjon og parallellisering av programmer med rekursjon
- Først en 'teoretisk' PRAM-lignende parallell løsning
  - som går ca. 1000x langsommere enn en sekvensiell versjon.
- Så skal vi se på to forbedringer som gjør at den uhyre treige løsningen vår tilslutt har en speedup på ca. 4
- Hele problemet bunner i de tallene vi presenterer i dag :

  - Å starte en ny tråd med vent på terminering tar: **92**μs (snitt mange ganger)
  - Å gjøre et metodekall tar: **0.016** μs (snitt mange ganger)

Om rekursiv oppdeling av et problem

- Svært mange problemer kan gis en (sekvensiell og parallell) rekursiv løsning:
  - De fleste søkeproblemer
    - Del søkebunken rekursivt opp i disjunkte deler og søk (i parallell) i hver bunke.
  - Mange sorterings-algoritmer som QuickSort, Flettesortering, og venstreRadix-sortering er definert rekursivt
  - Oblig4 - den konvekse innhyllinga
- Skal nå bruke en ny formulering av QuickSort som eksempel og gi den 3 ulike løsninger:
  A. Ren oversettelse av rekursjonen til tråder
  B. Med to tråder for hvert nivå inntil vi bruker InnstikkSortering
  C. Med en ny tråd for hver nivå og avslutting av tråder når lengden < LIMIT (si 50 000) – deretter vanlig rekursjon

Generelt om rekursiv oppdeling av a[] i to deler

```
void Rek (int [] a, int left, int right) {
        <del opp omradet a[left..right] >
        int deling = partition (a, left,right);

        if (deling - left  > LIMIT ) Rek (a,left,deling-1);
        else <enkel løsning>;
        if (right - deling > LIMIT) Rek (a,deling, right);
        else  <enkel løsning>
}
```

```
void Rek(int [] a, int left, int right) {
        <del opp omradet a[left..right]>
        int deling = partition (a, left,right);
        Thread t1 = null, t2= null;

        if (deling - left > LIMIT )  t1 = new Thread (a,left,deling-1);
        else <enkel løsning>;
        if (right - deling > LIMIT) t2 = new Thread (a,deling, right);
        else  <enkel løsning>
        try{ if (t1!=null)t1.join();
              if (t2!=null)t2.join();} catch(Exception e){};
}
```

Her noe stilisert, skal egentlig ha new Thread(new(Arbeider (a,left,deling-1))+ run-metode med samme innhold som Rek

```
void Rek (int [] a, int left, int right) {
            <del opp omradet a[left..right] >
            int deling = partition (a, left,right);

            if (deling - left  > LIMIT ) Rek (a,left, deling -1);
            else <enkel løsning>;
            if (right - deling > LIMIT) Rek (a, deling , right);
            else  <enkel løsning>
}
```

A

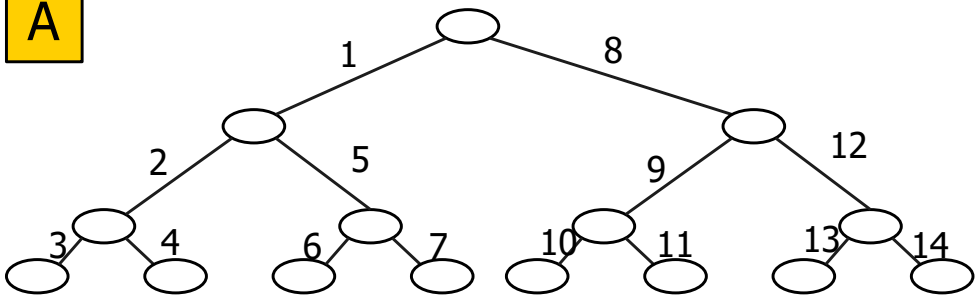
```
void Rek(int [] a, int left, int right) {
            <del opp omradet a[left..right]>
            int deling = partition (a, left,right);
            Thread t1 = null, t2=null;

            if (deling- left > LIMIT )
                (t1 = new Thread (a,left, deling -1)).start();
            else <enkel løsning>;
            if (right - deling> LIMIT)
                 (t2 = new Thread (a, deling , right)).start();
            else  <enkel løsning>
            try{ if (t1!=null) t1.join();
                  if (t2!=null) t2.join();}
           catch(Exception e){return;};
}
```

B

Oppdeling med **to** tråder per nivå i treet:

- Når ventes det i den rekursive løsningen
  - Har det betydning for rekkefølgen av venting ?
- Når ventes det i den parallelle løsningen A?
  - Har rekkefølgen på venting på t1 og t2 betydning?
- Antar at kall på Rek tar T millisek.
- Hvor lang tid tar A og B
- Hvilken er raskest ?

A

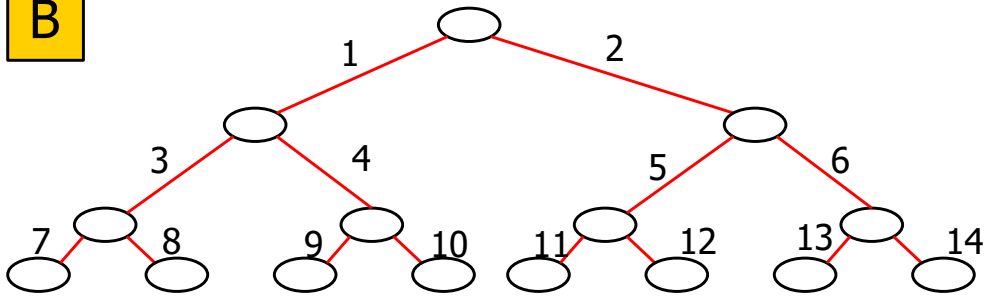


Rekursjon

Tråd

Dybde først - sekvensiell

B



Rekursjon

Tråd

Bredde først - parallell

Oppdeling med **en tråd** per nivå i treet:

- Hvorfor virker dette ?
- To alternativ løsning med 1 tråd
  - Har det betydning for rekkefølgen av venting ?
- Når ventes det i C-løsningen?
  - Har rekkefølgen på venting på t1 betydning?
- Når ventes det i D-løsningen?
- Antar at kall på Rek tar T millisek.
- Hvor lang tid tar C
- Hvor lang tid tar D

Hvilken er klart raskest:
C eller D?

D – er raskest fordi både høyre og venstre gren startes før man venter.

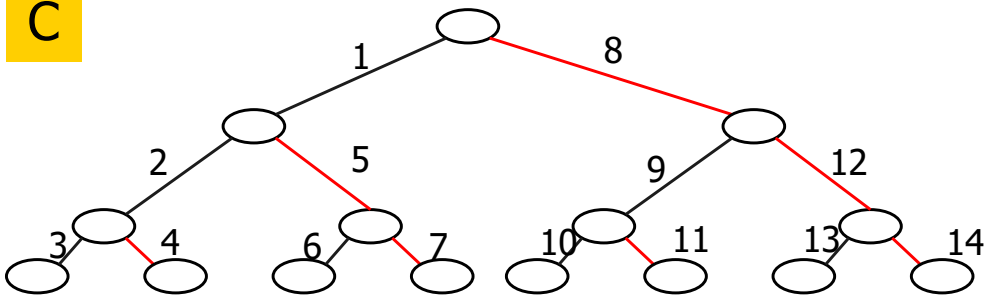
C

```
void Rek(int [] a, int left, int right) {
        <del opp omradet a[left..right]>
        int deling = partition (a, left,right);
        Thread t1;

        if (deling - left > LIMIT )
                Rek (a,left,deling-1);
        else <enkel løsning>;
        if (right - deling > LIMIT)
                t1 = new Thread (a,right,deling-1);
        else  <enkel løsning>
         try{t1.join();} catch(Exception e){};}
```

D

```
void Rek(int [] a, int left, int right) {
        <del opp omradet a[left..right]>
        int deling = partition (a, left,right);
        Thread t1;

        if (deling - left > LIMIT )
                t1 = new Thread (a,left,deling-1);
        else <enkel løsning>;
        if (right - deling > LIMIT)
                Rek (a,deling, right);
        else  <enkel løsning>
        try{t1.join();} catch(Exception e){};}
```

C

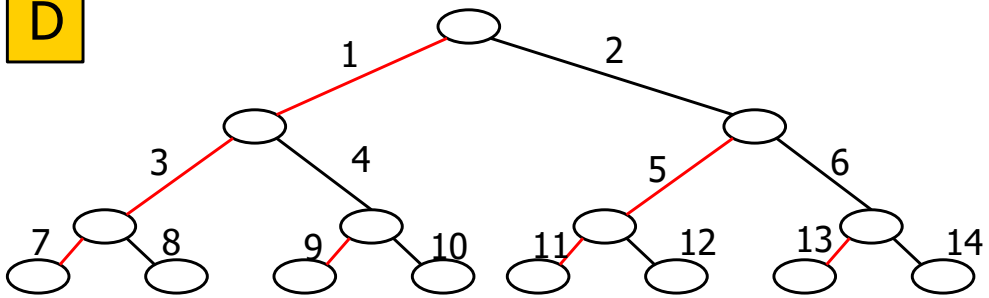


Rekursjon

Tråd

Dybde først - sekvensiell

D



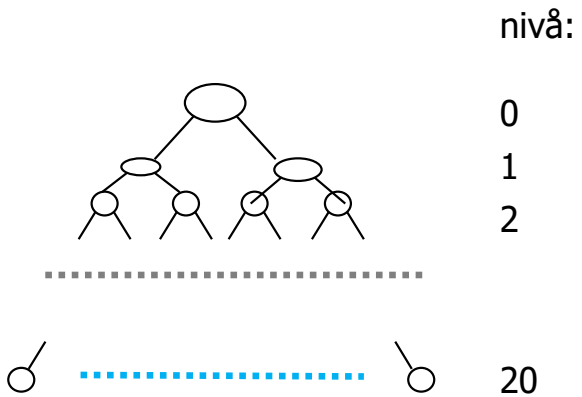
Rekursjon

Tråd

Bredde først - parallell

Hvor mange kall gjør vi i en rekursiv løsning?

- Anta Quicksort av n $=2^k$ tall
  (k= 10 ⇒ n = 1000, k= 20 ⇒ n= 1 mill)
- Kalltreet vil på første nivå ha 2 lengder av 2^{19}, på neste: 4 = 2^2 hver med 2^{18} og helt ned til nivå 20, hvor vi vil ha $2^{20+1}-1$ kall hver med 1 = 2^0 element.
- I hele kalltreet gjør vi altså 2 millioner -1 kall for å sortere 1 mill tall !
- Bruker vi innstikksortering for n < 32 = 2^5 så får vi 'bare' $2^{20-5+1} = 2^{15+1} -1$= **65 535** kall.
- Metodekall tar først: **2 us** men så **0.02** μs og kan også optimaliseres bort (og gis speedup >1)
- Å lage en tråd og starte den opp tar først : ca.**3000** μs, men så ca. **62** μs for de neste trådene (med start() og join())

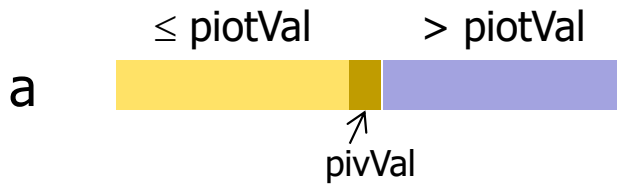


Vi kan IKKE bare erstatte rekursive kall med nye tråder i en rekursiv løsning !

A) Sekvensiell kvikksort – ny og enklere kode

```
// sekvensiell Kvikksort
void quicksortSek(int[] a, int left, int right) {
        int piv = partition (a, left, right); // del i to
        int piv2 = piv-1, pivotVal = a[piv];
        while (piv2 > left  && a[piv2] == pivotVal) {
             piv2--; // skip like elementer i midten
         }

        if ( piv2-left > 16) quicksortSek(a, left,  piv2);
            else insertSort(a,left, piv2);
        if ( right-piv >16) quicksortSek(a, piv + 1, right);
           else  insertSort(a, piv+1, right);
} // end quicksort
```

Etter: partition(a,left,right)




```
// del opp a[] i to: smaa og storre
int partition (int [] a, int left, int right) {
     int pivVal = a[(left + right) / 2];
     int index = left;
    // plasser pivot-element helt til høyre
    swap(a, (left + right) / 2, right);

     for (int i = left; i < right; i++) {
          if (a[i] <= pivVal) {
               swap(a, i, index);
               index++;
          }
      }
      swap(a, index, right); // sett pivot tilbake
      return index;
 } // end partition

 void swap(int [] a, int left, int right) {
       int temp = a[left];
       a[left] = a[right];
       a[right] = temp;
} // end swap
```

B) En parallell kode (modellert etter A)

```
void Rek(int [] a, int left, int right) {
            <del opp omradet a[left..right]>
            int deling = partition (a, left,right);
            Thread t1 = null, t2=null;

            if (deling- left > LIMIT )
                 (t1 = new Thread (a,left, deling -1)).start();
            else insertSort(a,left, deling -1);
            if (right - deling> LIMIT)
                  (t2 = new Thread (a, deling , right)).start();
            else  insertSort(a, deling , right);
            try{ if (t1!=null)t1.join();
                  if (t2!=null)t2.join();} catch(Exception e){};
}
```

B) Ren kopi av rekursiv løsning: Katastrofe

```
M:>java QuickSort 100 10 100000 1 uke9.txt
Test av  TEST AV QuickSort
 med 8 kjerner ,   Median av:1 iterasjoner, LIMIT:2


      n      sekv.tid(ms)   para.tid(ms)    Speedup
   100 000      34.813      41310.276       0.0008
    10 000       0.772        735.838       0.0010
     1 000       0.078         66.007       0.0012
       100       0.009          3.491       0.0026
```

Konklusjon:

- For store n speeddown på > 1000
- Kunne ikke kjøre for n > 100 000 pga. trådene tok for stor plass

Hva med en passe LIMIT = 32 ?

```
>java QuickSort 100 10 1000000 1 uke9.

Test av  TEST AV QuickSort
med 8 kjerner ,   Median av:1 iterasjoner, LIMIT:32


      n        sekv.tid(ms)    para.tid(ms)     Speedup
  1000000        89.181        41789.076         0.0021
   100000         8.118          823.432         0.0099
    10000         3.021           55.845         0.0541
     1000         0.060            3.463         0.0173
      100         0.006            0.302         0.0185
```

Konklusjon:

- Mye bedre, men fortsatt 100 x langsommere enn sekvensiell(N = 100 000)
- Greide nå n= 1 mill (fordi færre tråder)
- Fortsatt håpløst dårlig pga. for mange tråder
- Trenger ny idé : Bruk sekvensiell løsning når n < 50 000 ? BIG_LIMIT

Skisse av ny løsning

```
void Rek(int [] a, int left, int right) {
    if ( right – left  < BIG_LIMIT) quicksort (a, left,right);
    else {
            <del opp omradet a[left..right]>
            int deling = partition (a, left,right);
            Thread t1 = null, t2=null;

            //if (deling- left > LIMIT )
                (t1 = new Thread (a,left, deling -1)).start();
            //else insertSort(a,left, deling -1);
            //if (right - deling> LIMIT)
                 (t2 = new Thread (a, deling , right)).start();
            //else  insertSort(a, deling , right);
            try{ if (t1!=null)t1.join();
                 if (t2!=null)t2.join();} catch(Exception e){};
    }
}
```

Generere nye tråder bare i toppen av rekusjonstreet

- Kan da også stryke kode om LIMIT (vil ikke bli utført) i parallell kode
- Bruken av insertSort gjøres i (sekv) quicksort(..)

Kjøreeksempel med BIG_LIMIT og LIMIT

```
>java QuickSort 100 10 100000000 1 uke9.txt
Test av  TEST AV QuickSort med BIG_LIMIT
 med 8 kjerner ,   Median av:1 iterasjoner,
                   LIMIT:32, BIG_LIMIT:50000


      n       sekv.tid(ms)    para.tid(ms)     Speedup
 100000000    12042.708        3128.675        3.8491
  10000000     1090.252         277.264        3.9322
   1000000       92.958          32.640        2.8480
    100000        7.682           5.198        1.4777
     10000        0.616           0.737        0.8356
      1000        0.051           0.117        0.4354
       100        0.013           0.015        0.8636
```

BIG_LIMIT = 100 000

```
>java QuickSort 100 10 100000000 1 uke9.txt
Test av  TEST AV QuickSort med BIG_LIMIT
 med 8 kjerner ,   Median av:1 iterasjoner, LIMIT:32,
BIG_LIMIT:100000


      n       sekv.tid(ms)    para.tid(ms)     Speedup
 100000000    12110.344        2967.764        4.0806
  10000000     1084.587         277.454        3.9091
   1000000       93.428          32.078        2.9125
    100000        7.894           7.828        1.0085
     10000        0.815           0.617        1.3224
      1000        0.072           0.101        0.7153
       100        0.013           0.015        0.8410
```

N= 200 mill – 20 , BIG_LIMIT = 100 000

```
>java -Xmx6000m QuickSort 20 10 200000000 3 uke9.txt
Test av  TEST AV QuickSort med BIG_LIMIT
 med 8 kjerner ,   Median av:3 iterasjoner, LIMIT:32,
BIG_LIMIT:100000


      n       sekv.tid(ms)    para.tid(ms)     Speedup
 200000000    26091.361        5685.715        4.5889
  20000000     2352.854         598.071        3.9341
   2000000      197.353          61.362        3.2162
    200000       17.003           9.944        1.7099
     20000        1.328           1.332        0.9966
      2000        0.111           0.114        0.9754
       200        0.016           0.016        1.0217
        20        0.000           0.000        1.0000
```

Konklusjon om å parallellisere rekursjon

- Antall tråder må begrenses !
- I toppen av treet brukes tråder (til vi ikke har flere og kanskje litt mer)
- I resten av treet bruker vi sekvensiell løsning i hver tråd!
- Viktig også å kutte av nedre del av treet (her med insertSort) som å redusere treets størrelse drastisk (i antall noder)
- Vi har for n = 100 00 gått fra:

n	sekv.tid(ms)	para.tid(ms)	Speedup	
100000	34.813	41310.276	0.0008	Ren trådbasert
100000	8.118	823.432	0.0099	Med insertSort
100000	7.682	5.198	1.4777	+ Avkutting i toppen

- Speedup > 1 og ca. 10 000x fortere enn ren oversettelse.

Hvor lang tid tar egentlig ulike mekanismer i Java ? (fra Petter A. Busteruds master-oppgave: Invesigating Different Concurrency Mechanisms in Java)

- Dette er målinger på en relativt gammel CPU :
  - Intel Pentium M 760@ 2GHz – single core
  - 2 GB DDR2 RAM @ 533 MHz
- Windows 7 32bit
- Linux Mint 13, 32bit
- Det er de relative hastighetene som teller
- Petter målte bl.a. :
  - kalle en metode
  - lage et nytt objekt (new C()) + kalle en metode
  - lage en Tråd (new Thread + kalle run())
  - Lage en ExecutorService + legge inn tråder
  - Tider på int[] og trådsikre AtomicIntegerArray

Generelt er Linux 15-20% raskere enn Windows (ikke optimalisert)

Hvor lang tid tar egentlig ulike mekanismer i Java ? (fra Petter A. Busteruds master-oppgave: Investigating Different Concurrency Mechanisms in Java)

A) Metodekall på Windows og Linux (ikke optimalisert)

MethodCall	Number of Method Calls					
	1	2	4	8	16	32
First Call	4.75	4.75	4.75	4.75	4.75	4.75
Additional Calls	-	2.24	2.05	1.96	1.96	1.96
Total Run Time	4.75	6.71	10.7	18.5	34.1	65.4

Table 6.2: Overhead using Methods on Windows (in μs)

MethodCall	Number of Method Calls					
	1	2	4	8	16	32
First Call	4.54	4.54	4.55	4.54	4.54	4.54
Additional Calls	-	1.61	1.49	1.45	1.43	1.42
Total Run Time	4.54	6.08	8.95	14.6	26.0	48.5

Table 6.3: Overhead using Methods on Linux (in μs)

Generelt er kan vi gjøre mer enn 500-700 metodekall per millisek.

Å lage et nytt objekt av en klasse (new) og kalle en metode i objektet (μs)

ClassCall	Number of Classes					
	1	2	4	8	16	32
Create the Object	759	768	774	774	773	777
First Method Call	11	11	11	11	11	11
Additional Calls	-	4	3	2	2	2
Total Run Time	770	788	802	817	848	914

Table 6.4: Overhead using Classes on Windows (in μs)

Konklusjon:

- Verken det å kalle en metode eller å lage et objekt tar særlig lang tid
- Mye blir optimalisert videre
- Forskjellen mellom første og andre kall er at det sannsynligvis er blitt JIT-kompilert til maskinkode, men ikke optimalisert

Lage en tråd (new Thread()) og kalle run() - µs

- **First `Run()` Call**: Executing *`thread`*`.start()`, starting the object's `Run` method to be called in that separately executing thread.
- **Additional Threads**: Identical to "Create the First Thread", for the rest of the specified number of threads.
- **Additional `Run()` Calls**: Identical to "First `Run()` Call", but for the additional threads.
- **Total Thread Create**: Total cost of creating all Threads, First Thread + All Additional Threads.
- **Total `Run()` Call**: Total cost of the First `Run()` Call + All Additional `Run()` Calls.

ThreadCall	Number of Threads					
	1	2	4	8	16	32
Create the First Thread	759	758	764	767	766	772
First `Run()` Call	839	835	836	815	820	846
Additional Threads	-	37	27	24	22	22
Additional `Run()` Calls	-	188	178	180	170	168
Total Thread Create	-	797	850	935	1101	1456
Total `Run()` Call	-	1039	1388	2118	3425	6050
Total Run Time	1614	1841	2244	3062	4554	7533

Table 6.6: Overhead using Threads on Windows (in μs)

Lage en ExecutorService

- **Create ExecutorService**: Is the set-up time for the ExecutorService, here we create a fixed `ThreadPool` with the specified number of threads as a parameter. In addition create a `Future` to provide the ability to check when *computations* (calls) are completed.
- **First Submit**: The very first submit done to the `ThreadPool`, with the additional overhead for the system to assign the submitted work to an available Thread in the Pool.
- **Additional Submits**: Every additional submit done to the `ThreadPool`.
- **Total Submit**: Total cost of the First Submit + All Additional Submits.

ExecutorCall	Number of Threads in Pool					
	1	2	4	8	16	32
Create ExecutorService	4881	4861	4874	4870	4835	4957
First Submit	2026	2045	2030	2061	2042	2067
Additional Submits	-	231	238	212	203	214
Total Submit	-	2282	2756	3565	5142	8699
Total Run Time	6915	7169	7694	8534	10123	13734

Table 6.8: Overhead using ExecutorService on Windows (in μs)

Lage og bruke en vanlig int [] a (IKKE trådsikker) mot AtomicIntegerArray (trådsikre heltall)

`int[]`	Number: N					
	10	100	1000	10^4	10^5	10^6
GetAndSet N-times	0.119	0.288	1.96	22.6	225	2357
Increment N-times	0.339	0.540	2.22	21.2	208	2182
Decrement N-times	0.347	0.547	2.24	21.1	208	2177

Table 6.13: Overhead using Array of `int` on Windows (in μs)

`AtomicIntegerArray`	Number: N					
	10	100	1000	10^4	10^5	10^6
Create Array[N]	138	139	138	140	173	251
Get N-times	1.37	3.03	19.2	163	688	2951
Set N-times	4.37	5.92	19.2	167	1173	3860
Increment N-times	8.57	11.8	42.4	341	2323	5768
Decrement N-times	8.60	11.9	42.3	360	2365	5805

Table 6.15: Overhead using AtomicIntegerArray on Windows (in μs)

Oppsummering om kjøretider – java 6:

- Metodekall tar svært liten tid: **2-5** μs og kan også optimaliseres bort (og gis speedup >1)
- Å lage et objekt av en klasse (og kalle en metode i det) tar liten tid: ca.**785** μs
- Å lage en tråd og starte den opp tar en del tid: ca.**1500** μs, men lite, ca. **180** μs for de neste trådene (med start())
- Å lage en en tråd-samling og legge tråder (og Futures) opp i den tar ca. **7000** μs for første tråd og ca. **210** μs for neste tråd.
- Å bruke trådsikre heltall (AtomicInteger og AtomicIntegerArray) mot vanlige int og int[] tar vanligvis **10-20 x** så lang tid , men for mye bruk (> 10^6) tar det bare ca. 2x tid (det siste er sannynligvis en cache-effekt)

Sequential Quicksort

Number of Elements	Sorting Time
1 000	104 μs
2 500	288 μs
5 000	607 μs
7 500	973 μs
10 000	1318 μs
25 000	3632 μs
50 000	7980 μs
75 000	12214 μs
100 000	16980 μs

Figure 6.2: Quicksort Sorting Times

En parallell sorterings-løsning må alltid sammenlignes med hvor mye vi kunne ha sortert sekvensielt på denne tida!

Nye målinger i Java 8 i 2017 (programmet: Tider.java)

```
//Metodekall
 s ="Metodekall";
 t = System.nanoTime();
      k=neste(k); // metodekall 1 gang
 d = (double) ((System.nanoTime() -t)/1000.0);
 println(s+Format.align(d,10,4)+"us. forste gang");

  t = System.nanoTime();
 for (int i = 0; i<n; i++) {
      k+=neste(k); // metodekall, k ganger
 }
 d = (double) ((System.nanoTime() -t)/(n*1000.0));
 println(s+Format.align(d,10,4)+"us.snitt " + n+" ganger");
```

```
// Koden de andre tilfellene som ble testet
ia = new int[100];                                  // new Array

(t1 = new Thread(new Arbeider(3))).start();  // new Thread start&join()
try{t1.join();}catch (Exception e){}

k= new C(k).les();                                 // lage Object C + metodekall
k= new D(k).les();                                 // lage Object D+ metodekall

ia[i%100] =i;                                       // skriv i array
k = ia[i%55];                                       // les fra array
```

Hva med i 2017 (Java 8)– har kjøretidene endret seg ?

	Tid første (us)	Tid neste (us)	Snitt tid 10 (us)	Snitt tid 100000 (us)	Speedup
Metodekall	2.1	0.15	0.06	0.016	131
new int[100]	0.3	0.13	0.27	0.12	2.5
new Thread() start()+join()	3086.0	73.0	123.0	62.5	49
new C() + metodekall	3392.0	0.6	0.12	0.03	113066
new D() + metodekall	3025.0	0.9	0.15	0.04	84 500
Ett Array write/read	0.30	0.15	0.06	0.013	23

Konklusjoner: 1) Ulik speedup. Best i Java8: å lage objekter, og metodekall
2) relativt lite speedup på arrayer
Det er vår program som blir raskere, IKKE JVM (java) som tolker det bedre.
De på det å lage objekter av en klasse C, gjør ikke noe for klasse D-objekter

Sammenlignet med Petters tider?

	Java8: Tid første (us)	Java8 Snitt tid neste 10 (us)	Java 6: Tid første (us)	Java6: Snitt tid 16 (us) Uten join()	
Metodekall	2.1	0.06	4.75	1.96	
new int[100]	0.3	0.27			
new Thread() start()+join()	3086.0	123.0	1814	284	
new C() + metodekall	3392.0	0.12	770	53	
new D() + metodekall	3025.0	0.15	770	53	
Ett Array write/read	0.30	0.06		0.27	

Konklusjoner:

1) Ulik speedup – Java 8 er langt raskere enn Java6 på metodekall og objekter

2) relativt lite speedup på arrayer (husk maskinen hans var nesten 1/2 av hastigheten til dagens maskiner)

Hva så vi på i uke 8

- Perspektiv: Utvikling i CPU-speed, minne, nettverk, herunder effekten av Moore's Law – og af lysets langsomme hastighet!
- Et sitat om tidsforbruk ved faktorisering
- Om en feil i Java 7 (også i Java 8)ved tidtaking
- Hvordan parallellisere rekursive algoritmer
- Gå ikke i 'direkte oversettelses-fella'
  - eksemplifisert ved Kvikk-sort, 3 ulike løsninger
- Hvor lang tid tar de ulike mekanismene vi har i Java 6 og 8?

The experts look ahead

Cramming more components onto integrated circuits

With unit cost falling as the number of components per circuit rises, by 1975 economics may dictate squeezing as many as 65,000 components on a single silicon chip

By Gordon E. Moore

Director, Research and Development Laboratories, Fairchild Semiconductor division of Fairchild Camera and Instrument Corp.

The future of integrated electronics is the future of electronics itself. The advantages of integration will bring about a proliferation of electronics, pushing this science into many new areas.

Integrated circuits will lead to such wonders as home computers—or at least terminals connected to a central computer—automatic controls for automobiles, and personal portable communications equipment. The electronic wristwatch needs only a display to be feasible today.

But the biggest potential lies in the production of large systems. In telephone communications, integrated circuits in digital filters will separate channels on multiplex equipment. Integrated circuits will also switch telephone circuits and perform data processing.

Computers will be more powerful, and will be organized in completely different ways. For example, memories built of integrated electronics may be distributed throughout the machine instead of being concentrated in a central unit. In addition, the improved reliability made possible by integrated circuits will allow the construction of larger processing units. Machines similar to those in existence today will be built at lower costs and with faster turn-around.

The author

Dr. Gordon E. Moore is one of the new breed of electronic engineers, schooled in the physical sciences rather than in electronics. He earned a B.S. degree in chemistry from the University of California and a Ph.D. degree in physical chemistry from the California Institute of Technology. He was one of the founders of Fairchild Semiconductor and has been director of the research and development laboratories since 1959.

Present and future

By integrated electronics, I mean all the various technologies which are referred to as microelectronics today as well as any additional ones that result in electronics functions supplied to the user as irreducible units. These technologies were first investigated in the late 1950's. The object was to miniaturize electronics equipment to include increasingly complex electronic functions in limited space with minimum weight. Several approaches evolved, including microassembly techniques for individual components, thin-film structures and semiconductor integrated circuits.

Each approach evolved rapidly and converged so that each borrowed techniques from another. Many researchers believe the way of the future to be a combination of the various approaches.

The advocates of semiconductor integrated circuitry are already using the improved characteristics of thin-film resistors by applying such films directly to an active semiconductor substrate. Those advocating a technology based upon films are developing sophisticated techniques for the attachment of active semiconductor devices to the passive film arrays.

Both approaches have worked well and are being used in equipment today.

The establishment

Integrated electronics is established today. Its techniques are almost mandatory for new military systems, since the reliability, size and weight required by some of them is achievable only with integration. Such programs as Apollo, for manned moon flight, have demonstrated the reliability of integrated electronics by showing that complete circuit functions are as free from failure as the best individual transistors.

Most companies in the commercial computer field have machines in design or in early production employing integrated electronics. These machines cost less and perform better than those which use "conventional" electronics.

Instruments of various sorts, especially the rapidly increasing numbers employing digital techniques, are starting to use integration because it cuts costs of both manufacture and design.

The use of linear integrated circuitry is still restricted primarily to the military. Such integrated functions are expensive and not available in the variety required to satisfy a major fraction of linear electronics. But the first applications are beginning to appear in commercial electronics, particularly in equipment which needs low-frequency amplifiers of small size.

Reliability counts

In almost every case, integrated electronics has demonstrated high reliability. Even at the present level of production—low compared to that of discrete components—it offers reduced systems cost, and in many systems improved performance has been realized.

Integrated electronics will make electronic techniques more generally available throughout all of society, performing many functions that presently are done inadequately by other techniques or not done at all. The principal advantages will be lower costs and greatly simplified design—payoffs from a ready supply of low-cost functional packages.

For most applications, semiconductor integrated circuits will predominate. Semiconductor devices are the only reasonable candidates presently in existence for the active elements of integrated circuits. Passive semiconductor elements look attractive too, because of their potential for low cost and high reliability, but they can be used only if precision is not a prime requisite.

Silicon is likely to remain the basic material, although others will be of use in specific applications. For example, gallium arsenide will be important in integrated microwave functions. But silicon will predominate at lower frequencies because of the technology which has already evolved around it and its oxide, and because it is an abundant and relatively inexpensive starting material.

Costs and curves

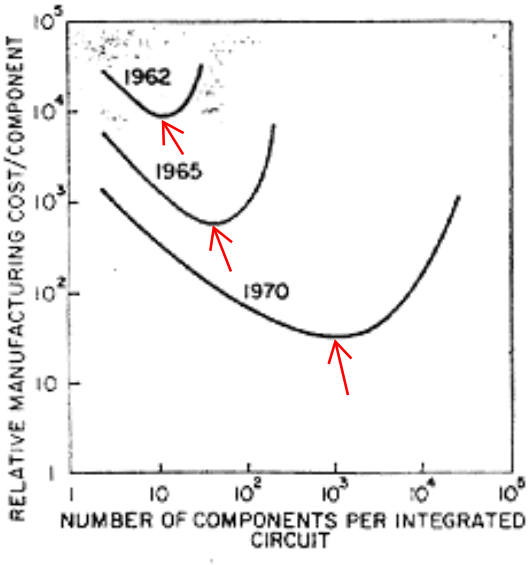
Reduced cost is one of the big attractions of integrated electronics, and the cost advantage continues to increase as the technology evolves toward the production of larger and larger circuit functions on a single semiconductor substrate. For simple circuits, the cost per component is nearly inversely proportional to the number of components, the result of the equivalent piece of semiconductor in the equivalent package containing more components. But as components are added, decreased yields more than compensate for the increased complexity, tending to raise the cost per component. Thus there is a minimum cost at any given time in the evolution of the technology. At present, it is reached when 50 components are used per circuit. But the minimum is rising rapidly while the entire cost curve is falling (see graph below). If we look ahead five years, a plot of costs suggests that the minimum cost per component might be expected in circuits with about 1,000 components per circuit (providing such circuit functions can be produced in moderate quantities.) In 1970, the manufacturing cost per component can be expected to be only a tenth of the present cost.

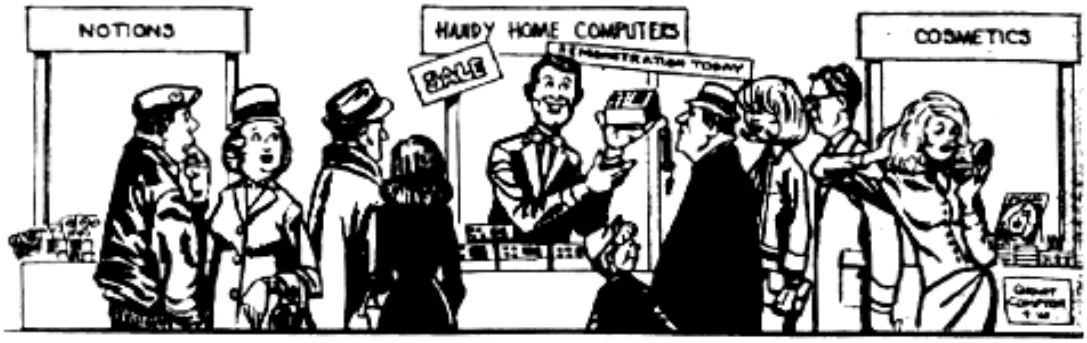
The complexity for minimum component costs has increased at a rate of roughly a factor of two per year (see graph on next page). Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000.

I believe that such a large circuit can be built on a single wafer.

Two-mil squares

With the dimensional tolerances already being employed in integrated circuits, isolated high-performance transistors can be built on centers two thousandths of an inch apart. Such



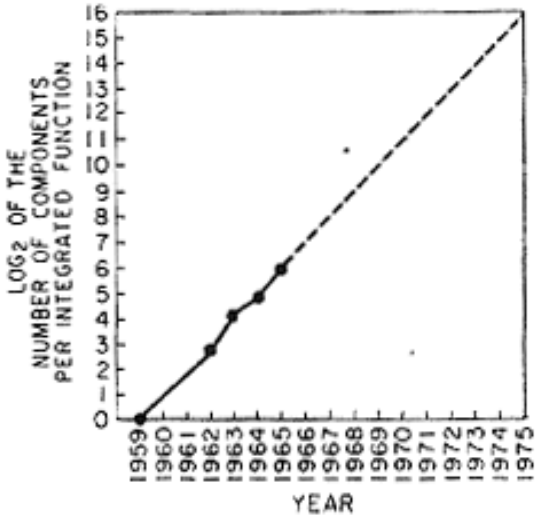


a two-mil square can also contain several kilohms of resistance or a few diodes. This allows at least 500 components per linear inch or a quarter million per square inch. Thus, 65,000 components need occupy only about one-fourth a square inch.

On the silicon wafer currently used, usually an inch or more in diameter, there is ample room for such a structure if the components can be closely packed with no space wasted for interconnection patterns. This is realistic, since efforts to achieve a level of complexity above the presently available integrated circuits are already underway using multilayer metalization patterns separated by dielectric films. Such a density of components can be achieved by present optical techniques and does not require the more exotic techniques, such as electron beam operations, which are being studied to make even smaller structures.

Increasing the yield

There is no fundamental obstacle to achieving device yields of 100%. At present, packaging costs so far exceed the cost of the semiconductor structure itself that there is no incentive to improve yields, but they can be raised as high as is economically justified. No barrier exists comparable to the thermodynamic equilibrium considerations that often limit yields in chemical reactions; it is not even necessary to do any fundamental research or to replace present processes. Only the engineering effort is needed.



In the early days of integrated circuitry, when yields were extremely low, there was such incentive. Today ordinary integrated circuits are made with yields comparable with those obtained for individual semiconductor devices. The same pattern will make larger arrays economical, if other considerations make such arrays desirable.

Heat problem

Will it be possible to remove the heat generated by tens of thousands of components in a single silicon chip?

If we could shrink the volume of a standard high-speed digital computer to that required for the components themselves, we would expect it to glow brightly with present power dissipation. But it won't happen with integrated circuits. Since integrated electronic structures are two-dimensional, they have a surface available for cooling close to each center of heat generation. In addition, power is needed primarily to drive the various lines and capacitances associated with the system. As long as a function is confined to a small area on a wafer, the amount of capacitance which must be driven is distinctly limited. In fact, shrinking dimensions on an integrated structure makes it possible to operate the structure at higher speed for the same power per unit area.

Day of reckoning

Clearly, we will be able to build such component-crammed equipment. Next, we ask under what circumstances we should do it. The total cost of making a particular system function must be minimized. To do so, we could amortize the engineering over several identical items, or evolve flexible techniques for the engineering of large functions so that no disproportionate expense need be borne by a particular array. Perhaps newly devised design automation procedures could translate from logic diagram to technological realization without any special engineering.

It may prove to be more economical to build large

systems out of smaller functions, which are separately packaged and interconnected. The availability of large functions, combined with functional design and construction, should allow the manufacturer of large systems to design and construct a considerable variety of equipment both rapidly and economically.

Linear circuitry

Integration will not change linear systems as radically as digital systems. Still, a considerable degree of integration will be achieved with linear circuits. The lack of large-value capacitors and inductors is the greatest fundamental limitations to integrated electronics in the linear area.

By their very nature, such elements require the storage of energy in a volume. For high Q it is necessary that the volume be large. The incompatibility of large volume and integrated electronics is obvious from the terms themselves. Certain resonance phenomena, such as those in piezoelectric crystals, can be expected to have some applications for tuning functions, but inductors and capacitors will be with us for some time.

The integrated r-f amplifier of the future might well consist of integrated stages of gain, giving high performance at minimum cost, interspersed with relatively large tuning elements.

Other linear functions will be changed considerably. The matching and tracking of similar components in integrated structures will allow the design of differential amplifiers of greatly improved performance. The use of thermal feedback effects to stabilize integrated structures to a small fraction of a degree will allow the construction of oscillators with crystal stability.

Even in the microwave area, structures included in the definition of integrated electronics will become increasingly important. The ability to make and assemble components small compared with the wavelengths involved will allow the use of lumped parameter design, at least at the lower frequencies. It is difficult to predict at the present time just how extensive the invasion of the microwave area by integrated electronics will be. The successful realization of such items as phased-array antennas, for example, using a multiplicity of integrated microwave power sources, could completely revolutionize radar.

Gordon E. Moore, Co-founder
Intel Corporation

Progress In Digital Integrated Electronics

Complexity of integrated circuits has approximately doubled every year since their introduction. Cost per function has decreased several thousand-fold, while system performance and reliability have been improved dramatically. Many aspects of processing and design technology have contributed to make the manufacture of such functions as complex single chip microprocessors or memory circuits economically feasible. It is possible to analyze the increase in complexity plotted in Figure 1 into different factors that can, in turn, be examined to see what contributions have been important in this development and how they might be expected to continue to evolve. The expected trends can be recombined to see how long exponential growth in complexity can be expected to continue.

A first factor is the area of the integrated structures. Chip areas for some of the largest of the circuits used in constructing Figure 1 are plotted in Figure 2. Here again, the trend follows an exponential quite well, but with significantly lower slope than the complexity curve. Chip area tor maximum complexity has increased by a factor of approximately 20 from the first planar transistor in 1959 to the 16,384-bit charge-coupled device memory chip that corresponds to the point plotted for 1975, while complexity, according to the annual doubling law, should have increased about 65,000-fold. Clearly much of the increased complexity had to result from higher density of components on the chip, rather than from the increased area available through the use of larger chips.

Figure 1 Approximate component count for complex integrated circuits vs. year of Introduction.

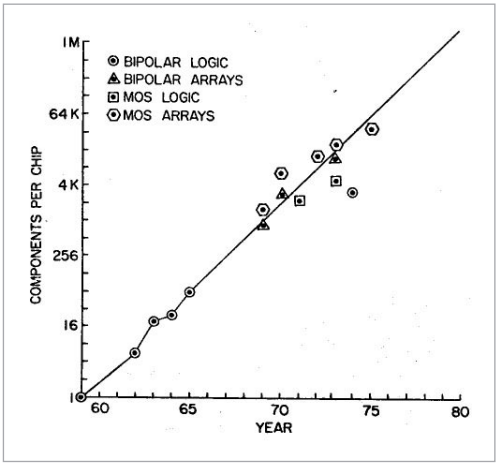
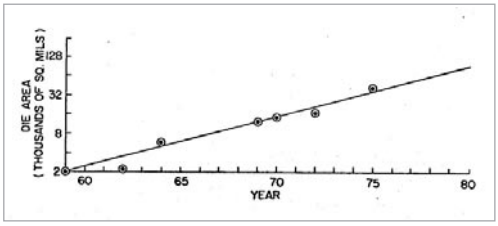


Figure 2 Increase in die area for most complex integrated devices commercially available.



Density was increased partially by using finer scale microstructures. The first integrated circuits of 1961 used line widths of 1 mil (~25 micrometers) while the 1975 device uses 5 micrometer lines. Both line width and spacing between lines are equally important in improving density. Since they have not always been equal,

G. E. Moore, "Progress in Digital Integrated Electronics." © 1975 IEEE. Reprinted, with permission, from Technical Digest 1975. International Electron Devices Meeting, IEEE, 1975, pp. 11-13.

Figure 3 Device density contribution from the decrease in line widths and spacings.

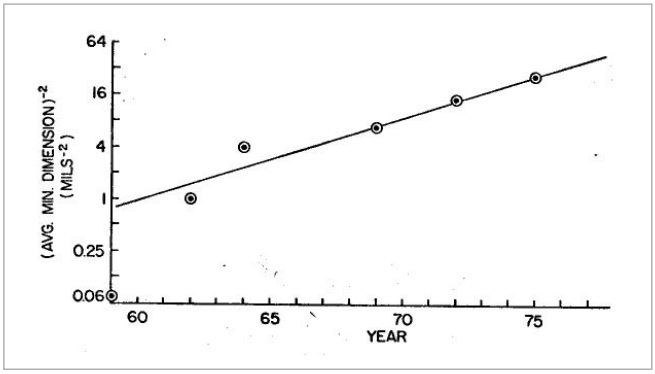
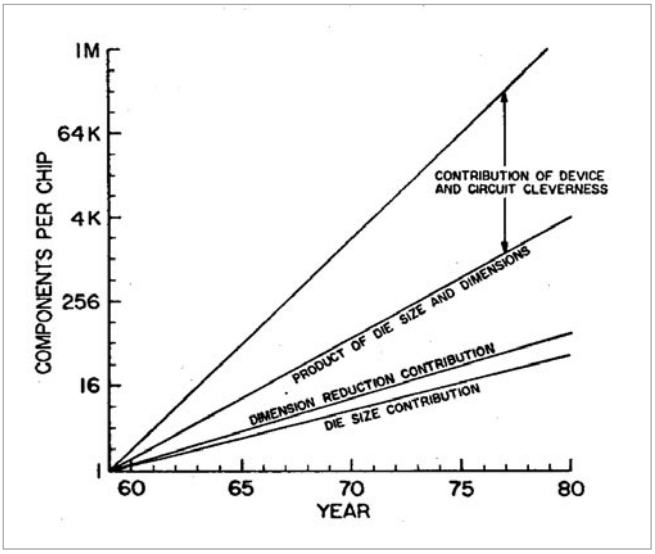


Figure 4 Decomposition of the complexity curve into various components.



the average of the two is a good parameter to relate to the area that a structure might occupy. Density can be expected to be proportional to the reciprocal of area, so the contribution to improve density vs. time from the use of smaller dimensions is plotted in Figure 3.

Neglecting the first planar transistor, where very conservative line width and spacing was employed, there is again a reasonable fit to an exponential growth. From the exponential approximation represented by the straight line in Figure 3, the increase in density from this source over the 1959-1975 period is a factor of approximately 32.

Combining the contribution of larger chip area and higher density resulting from geometry accounts for a 640-fold increase in complexity, leaving a factor of about 100 to account for through 1975, as is shown graphically in Figure 4. This factor is the contribution of circuit and device advances to higher density. It is noteworthy that this contribution to complexity has been more important than either increased chip area or finer lines. Increasingly the surface areas of the integrated devices have been committed to components rather than to such inactive structures as device isolation and interconnections, and the components themselves have trended toward minimum size, consistent with the dimensional tolerances employed.

Can these trends continue?

Extrapolating the curve for die size to 1980 suggests that chip area might be about 90,000 sq. mils, or the equivalent of 0.3 inches square. Such a die size is clearly consistent with the 3 inch wafer presently widely used by the industry. In fact, the size of the wafers themselves have grown about as fast as has die size during the time period under consideration and can be expected to continue to grow. Extension to larger die size depends principally upon the continued reduction in the density of defects. Since the existence of the type of defects that harm integrated circuits is not fundamental, their density can be reduced as long as such reduction has sufficient economic merit to justify the effort. I see sufficient continued merit to expect progress to continue for the next several years. Accordingly, there is no present reason to expect a change in the trend shown in Figure 2.

With respect to dimensions, in these complex devices we are still far from the minimum device sizes limited by such fundamental considerations as the charge on the electron or the atomic structure of matter. Discrete devices with sub-micrometer dimensions show that no basic problems should be expected at least until the average line width and

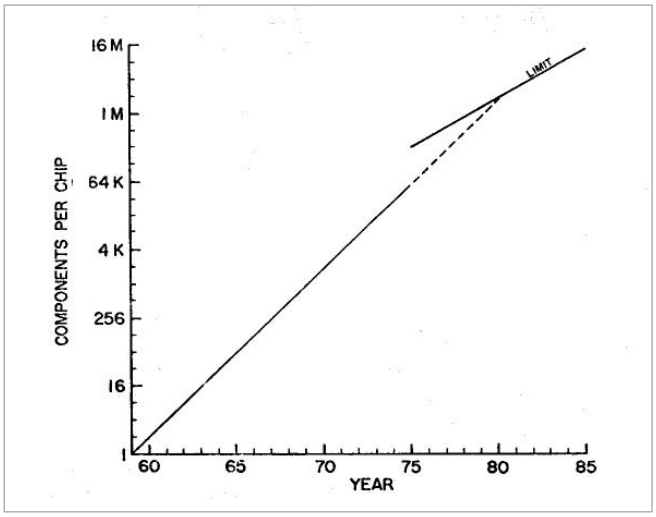
spaces are a micrometer or less. This allows for an additional factor of improvement at least equal to the contribution from the finer geometries of the last fifteen years. Work in non-optical masking techniques, both electron beam and X-ray, suggests that the required resolution capabilities will be available. Much work is required to be sure that defect densities continue to improve as devices are scaled to take advantage of the improved resolution. However, I see no reason to expect the rate of progress in the use of smaller minimum dimensions in complex circuits to decrease in the near future. This contribution should continue along the curve of Figure 3.

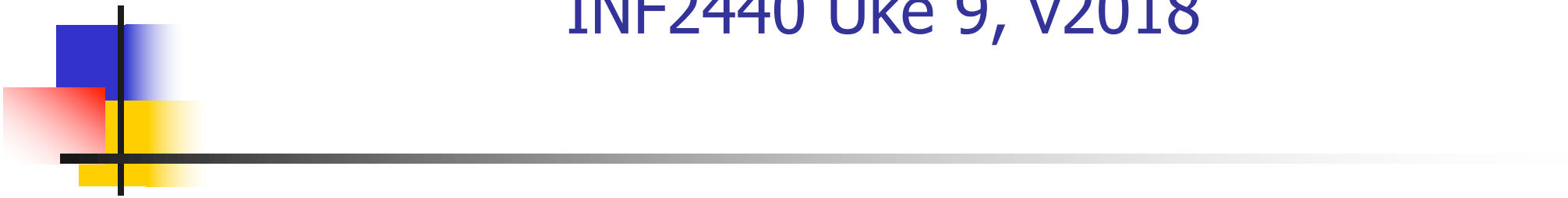
With respect to the factor contributed by device and circuit cleverness, however, the situation is different. Here we are approaching a limit that must slow the rate of progress. The CCD structure can approach closely the maximum density practical. This structure requires no contacts to the components within the array, but uses gate electrodes that can be at minimum spacing to transfer charge and information from one location to the next. Some improvement in overall packing efficiency is possible beyond the structure plotted as the 1975 point in Figure 1, but it is unlikely that the packing efficiency alone can contribute as much as a factor of four, and this only in serial data paths. Accordingly, I am Inclined to suggest a limit to the contribution of circuit and device cleverness of another factor of four in component density.

With this factor disappearing as an important contributor, the rate of increase of complexity can be expected to change slope in the next few years as shown in Figure 5. The new slope might approximate a doubling every two years, rather than every year, by the end of the decade.

Even at this reduced slope, integrated structures containing several million components can be expected within ten years. These new devices will continue to reduce the cost of electronic functions and extend the utility of digital electronics more broadly throughout society.

Figure 5 Projection of the complexity curve reflecting the limit on increased density through invention.





INF2440 Uke 9, v2018

Eric Jul
PSE
Inst. for informatikk

Hva så vi på i uke 8

- Perspektiv: Utvikling i CPU-speed, minne, nettverk, herunder effekten av Moore's Law – og af lysets langsomme hastighet!
- Et sitat om tidsforbruk ved faktorisering
- Om en feil i Java 7 (også i Java 8)ved tidtaking
- Hvordan parallellisere rekursive algoritmer
- Gå ikke i 'direkte oversettelses-fella'
  - eksemplifisert ved Kvikk-sort, 3 ulike løsninger
- Hvor lang tid tar de ulike mekanismene vi har i Java 6 og 8?

Hva skal vi se på i uke 9

- Oblig3: Primes
- Next obligs tentative dates: 4: 29/-12/4, 5: 26/4-10/5
- Concurrency & synkronisering – et eksempel
  - Feil v/gjenbruk av cyclic barrier – eksamen v18
- Parallellisering av Bubblesort

Concurrency & synkronisering

- Spørsmål fra eksamen våren 2018

Parallel Bubblesort

- Spørsmål fra eksamen våren 2018

Information

INF2440 Spring 2018 written exam

Date and time: June 11th, 2018 at 09:00.

Duration: 4 hours

Material allowed: All written material is allowed. No electronic aid is allowed. The lecture slides are attached as a PDF.

1 Question 1.1: Java Thread Creation

In a Java program, the programmer must create and start every thread that the program uses. Is this right?
Select an alternative:

- Yes, the program must create and start all threads.
- Yes, the program must create and start all threads in main().
- No, the first thread is created automatically and is automatically started and set to execute the main() method. The remaining threads must be created and started by one of the other threads in the program.
- No, the programmer must merely provide the code for the threads. The compiler will then automatically generate the threads at compile time and ensure that they are started when the program runs.
- No, all threads are started automatically.

Maximum marks: 4

2 Question 1.2: Java Threads Features

Which of the following statements are true for Java Threads that are created in the same program? (More than one answer is possible.)
Select one or more alternatives:

- Java threads can execute in parallel independently of one another on multi-core machines.
- If there are more threads than available cores, the program cannot be run.
- Java threads execute at the same speed because each thread is assigned to its own core .
- Java threads can access the local variable of one another.
- Java threads reside in a common part of memory and can access static variables declared in the class containing the threads.
- Java threads always execute one at a time.

Maximum marks: 6

3 Question 1.3: Java Synchronized

Can we ensure that only one thread at a time is executing a method by prefixing the method with the

synchronized keyword.
Select an alternative:

- Yes, if more than one thread calls the method, the Java system ensures that only one thread is executing the method at a time.
- Yes, except for the initial thread running main() -- it is always allowed to execute the method even if another thread is running the method at the same time.
- No, inside the method, we must add a synchronization of some kind, e.g., CyclicBarrier.
- Only if the method return type is void.

Maximum marks: 5

4 Question 1.4: Performance of Java Threads

Which of the following statements is closest to reality?
Select an alternative:

- Java threads can be created and run so quickly that even if we are just doing two or three integer additions, it is worth doing them in a separate thread because the thread will execute quickly on another core.
- Java threads take a lot of time to create and to synchronize so in general a thread should not be created unless it can do a lot of work, e.g., at least many thousands of additions.
- Java Threads are cheap to create as long as there are less threads than cores on the machine.
- Java threads are expensive to create but it becomes cheaper and cheaper to create threads the more threads a program creates.

Maximum marks: 5

i Question 2: Cyclic Barriers

In this question, you should consider the program Problem.java attached as a PDF-document.

5 Question 2.1: Cyclic Barrier - termination with two threads

When the program is called with the parameters "2 0", will the program always terminate? Explain your answer.

Fill in your answer here

Format

Words: 0

Maximum marks: 10

6 Question 2.2: Cyclic Barrier - output with two threads

When the program is called with the parameters "2 0", will the program *always* print the same text? Explain your answer.

Fill in your answer here

Format

Words: 0

Maximum marks: 10

7 Question 2.3 Cyclic Barrier: Termination with four threads

When the program is called with the parameters "2 2", will the program always terminate? Explain your answer.

Fill in your answer here

Format | B I U | Words: 0

Maximum marks: 12

8 Question 2.4: Cyclic Barrier: Output with four threads

When the program is called with the paramters "2 2", will the program always print the same text? If so, show the output. If not, give two different examples of what the program prints.

Fill in your answer here

Format | B I U | Words: 0

Maximum marks: 12

i Question 3: Finding Prime Deserts

This question is based on Erathostenes Sieve that finds prime numbers up to a given number N. You should already be quite familiar with the sieve as you have implemented it in Oblig 3 (attached as PDF).

This question is about finding so-called Prime Deserts. A Prime Desert is an interval [A, B] where A < B, A and B

are prime numbers, and there is no prime between A and B. Examples of such deserts are [2, 3], [7, 11], [23, 29], and [337, 347]. The size of a Prime Desert is defined as B-A-1, i.e., the number of integers strictly between A and B. A is called the start point of the Prime Desert and B is called the end point.

You are to write a Java program that generates a list of Prime Deserts. The list should be sorted so that the intervals come in ascending order, i.e., the starts points are in ascending order. The first Prime Desert is [2, 3] and thereafter, the next Prime Desert should be larger than the previous one throughout the list. Furthermore, you should find all Prime Deserts where the end point no greater than N, but you should exclude from the list any Prime Desert whose size is not greater than the size of the previous Prime Desert in the list. For example, that means that the start of the list is: [2,3] ,[3, 5], [7, 11], [23, 29], [89, 97],... i.e., [5, 7], [11, 13], [13, 17], [17, 19] and more are left out because their size is not greater than the previous Prime Desert in the list. To be sure, you may not leave out a Prime Desert, if it is larger than the previous in the list and smaller than the next one; for example, you cannot have a list that starts with [2, 3] followed by [23, 29].

You need not write the entire program as you can assume that you already have the code that you wrote for Oblig 3. You are welcome to base your code on Modell2 from the lectures in Week 5 (uke5).

9 Question 3.1: Sequential Program for finding Prime Deserts

Write a sequential Java program that finds the list of Prime Deserts specified in the introduction to Question 3.

Fill in your answer here

```
1
```

Maximum marks: 20

10 Question 3.2: Parallel Program to find Prime Deserts

Write a parallel Java program that finds the list of Prime Deserts specified in the introduction to Question 3. Strive to achieve a speedup over the sequential version.

Fill in your answer here

Maximum marks: 40

Question 4: Bubblesort

Bubblesort is a sorting algorithm that sorts, e.g., an integer array A, by comparing two neighboring elements and exchaning them, if the first one is greater than the second. The sorting consists of a number of passes. Each pass goes thru the array one element at a time and performs the exchange, if necessary. At the end of pass k, the k'th largest number is in place at the top of the array, so the next pass need not consider the top k elements of the array.
A sequential bubblesort of an integer array is given as a PDF file.

11 Question 4.1 Parallelizing Bubblesort

How can Bubblesort be parallelized? Describe the design of a solution.

Fill in your answer here

Format

Words: 0

Maximum marks: 20

12 Question 4.2: Parallel Bubblesort

Write a Java program implementing your design of a parallel version of Bubblesort from Question 4.1. Strive to have a speedup over the sequential version. Put any added explanation that you have as comments in the code.

Fill in your answer here

```
1
```

Maximum marks: 40

13 Question 5.1: Recursion and Parallelism

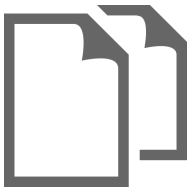
There are many ways of parallelizing a recurvise algorithm. Of the following statements, which of them is a good idea (you may select more than one):

Select one or more alternatives:

- When the recursive calls form a tree structure, it is a good idea to spawn and use threads at the bottom of the tree only.
- A simple way to parallelize a recursive program is to have one thread per recursive call. This almost always leads to achieving a good speedup.
- It is important to limit the number of threads.
- When the recursive calls form a tree structure, it is a good idea to spawn new thread at the top of the tree and limit the number of threads at the bottom of the tree.
- Recursive calls are relatively expensive compared to creating and running a thread.
- Recursive calls are several orders of magnitude cheaper than creating and running a thread.
- When the recursive calls form a tree structure, it is a good idea to replace branches at the lower levels of the tree with simple sequential solutions of the problem.
- It is important to balance the number of threads and the amount of work done by each thread.

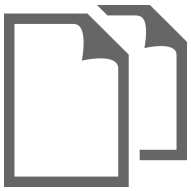
Maximum marks: 16

Question 12
Attached



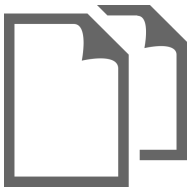
```
static void bubbleSort(int[] arr) {
   int n = arr.length;
   int temp;
   for (int i=0; i < n; i++){
      for (int j=1; j < (n-i); j++){
          if(arr[j-1] > arr[j]){
             //swap elements
             temp = arr[j-1];
             arr[j-1] = arr[j];
             arr[j] = temp;
          }

      }
   }
}
```

Question 12
Attached



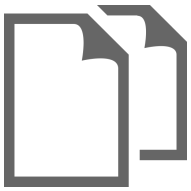
```
static void bubbleSort(int[] arr) {
   int n = arr.length;
   int temp;
   for (int i=0; i < n; i++){
      for (int j=1; j < (n-i); j++){
          if(arr[j-1] > arr[j]){
             //swap elements
             temp = arr[j-1];
             arr[j-1] = arr[j];
             arr[j] = temp;
          }

      }
   }
}
```

Question 5
Attached



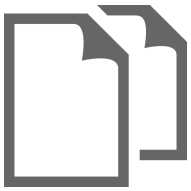
```
import java.util.concurrent.*;
class Problem {
   // felles data og metoder A
   static int num = 2;
   static int extra = 2;
   static CyclicBarrier b;

   public static void main(String [] args) {
      Problem p = new Problem();
      num = Integer.parseInt(args[0]);
      extra = Integer.parseInt(args[1]);
      b = new CyclicBarrier(num);

      p.utfoer(num+extra); // extra threads
      System.out.println(" Main TERMINATED");
   } // end main

   void utfoer (int antT) {
      Thread [] t = new Thread [antT];
      for (int i =0; i< antT; i++)
         ( t[i] = new Thread(new Arbeider(i))).start();
     try {
        for (int i =0; i< antT; i++) t[i].join();
     } catch(Exception e) {}
   } // end utfoer

   class Arbeider implements Runnable {
      // lokale data og metoder B
      int ind;

      void sync() {
         try{
            b.await();
         } catch (Exception e) { return;}
      }

      public Arbeider (int in) {ind = in;};

      public void run() {
         sync();
         System.out.println("A"+ind);
         sync();
         System.out.println("B"+ind);
      } // end run
   } // end indre klasse Arbeider
} // end class Problem
```

Question 5
Attached



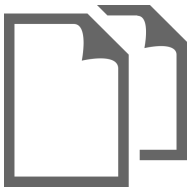
```
import java.util.concurrent.*;
class Problem {
   // felles data og metoder A
   static int num = 2;
   static int extra = 2;
   static CyclicBarrier b;

   public static void main(String [] args) {
      Problem p = new Problem();
      num = Integer.parseInt(args[0]);
      extra = Integer.parseInt(args[1]);
      b = new CyclicBarrier(num);

      p.utfoer(num+extra); // extra threads
      System.out.println(" Main TERMINATED");
   } // end main

   void utfoer (int antT) {
      Thread [] t = new Thread [antT];
      for (int i =0; i< antT; i++)
         ( t[i] = new Thread(new Arbeider(i))).start();
     try {
        for (int i =0; i< antT; i++) t[i].join();
     } catch(Exception e) {}
   } // end utfoer

   class Arbeider implements Runnable {
      // lokale data og metoder B
      int ind;

      void sync() {
         try{
            b.await();
         } catch (Exception e) { return;}
      }

      public Arbeider (int in) {ind = in;};

      public void run() {
         sync();
         System.out.println("A"+ind);
         sync();
         System.out.println("B"+ind);
      } // end run
   } // end indre klasse Arbeider
} // end class Problem
```

Question 6
Attached



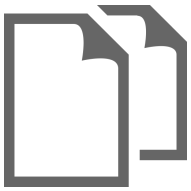
```
import java.util.concurrent.*;
class Problem {
   // felles data og metoder A
   static int num = 2;
   static int extra = 2;
   static CyclicBarrier b;

   public static void main(String [] args) {
      Problem p = new Problem();
      num = Integer.parseInt(args[0]);
      extra = Integer.parseInt(args[1]);
      b = new CyclicBarrier(num);

      p.utfoer(num+extra); // extra threads
      System.out.println(" Main TERMINATED");
   } // end main

   void utfoer (int antT) {
      Thread [] t = new Thread [antT];
      for (int i =0; i< antT; i++)
         ( t[i] = new Thread(new Arbeider(i))).start();
     try {
        for (int i =0; i< antT; i++) t[i].join();
     } catch(Exception e) {}
   } // end utfoer

   class Arbeider implements Runnable {
      // lokale data og metoder B
      int ind;

      void sync() {
         try{
            b.await();
         } catch (Exception e) { return;}
      }

      public Arbeider (int in) {ind = in;};

      public void run() {
         sync();
         System.out.println("A"+ind);
         sync();
         System.out.println("B"+ind);
      } // end run
   } // end indre klasse Arbeider
} // end class Problem
```

Question 6
Attached



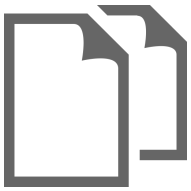
```
import java.util.concurrent.*;
class Problem {
   // felles data og metoder A
   static int num = 2;
   static int extra = 2;
   static CyclicBarrier b;

   public static void main(String [] args) {
      Problem p = new Problem();
      num = Integer.parseInt(args[0]);
      extra = Integer.parseInt(args[1]);
      b = new CyclicBarrier(num);

      p.utfoer(num+extra); // extra threads
      System.out.println(" Main TERMINATED");
   } // end main

   void utfoer (int antT) {
      Thread [] t = new Thread [antT];
      for (int i =0; i< antT; i++)
         ( t[i] = new Thread(new Arbeider(i))).start();
     try {
        for (int i =0; i< antT; i++) t[i].join();
     } catch(Exception e) {}
   } // end utfoer

   class Arbeider implements Runnable {
      // lokale data og metoder B
      int ind;

      void sync() {
         try{
            b.await();
         } catch (Exception e) { return;}
      }

      public Arbeider (int in) {ind = in;};

      public void run() {
         sync();
         System.out.println("A"+ind);
         sync();
         System.out.println("B"+ind);
      } // end run
   } // end indre klasse Arbeider
} // end class Problem
```

Question 7
Attached



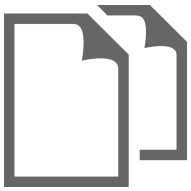
```
import java.util.concurrent.*;
class Problem {
   // felles data og metoder A
   static int num = 2;
   static int extra = 2;
   static CyclicBarrier b;

   public static void main(String [] args) {
      Problem p = new Problem();
      num = Integer.parseInt(args[0]);
      extra = Integer.parseInt(args[1]);
      b = new CyclicBarrier(num);

      p.utfoer(num+extra); // extra threads
      System.out.println(" Main TERMINATED");
   } // end main

   void utfoer (int antT) {
      Thread [] t = new Thread [antT];
      for (int i =0; i< antT; i++)
         ( t[i] = new Thread(new Arbeider(i))).start();
     try {
        for (int i =0; i< antT; i++) t[i].join();
     } catch(Exception e) {}
   } // end utfoer

   class Arbeider implements Runnable {
      // lokale data og metoder B
      int ind;

      void sync() {
         try{
            b.await();
         } catch (Exception e) { return;}
      }

      public Arbeider (int in) {ind = in;};

      public void run() {
         sync();
         System.out.println("A"+ind);
         sync();
         System.out.println("B"+ind);
      } // end run
   } // end indre klasse Arbeider
} // end class Problem
```

Question 7
Attached



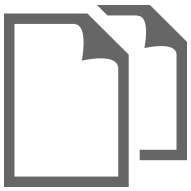
```
import java.util.concurrent.*;
class Problem {
   // felles data og metoder A
   static int num = 2;
   static int extra = 2;
   static CyclicBarrier b;

   public static void main(String [] args) {
      Problem p = new Problem();
      num = Integer.parseInt(args[0]);
      extra = Integer.parseInt(args[1]);
      b = new CyclicBarrier(num);

      p.utfoer(num+extra); // extra threads
      System.out.println(" Main TERMINATED");
   } // end main

   void utfoer (int antT) {
      Thread [] t = new Thread [antT];
      for (int i =0; i< antT; i++)
         ( t[i] = new Thread(new Arbeider(i))).start();
     try {
        for (int i =0; i< antT; i++) t[i].join();
     } catch(Exception e) {}
   } // end utfoer

   class Arbeider implements Runnable {
      // lokale data og metoder B
      int ind;

      void sync() {
         try{
            b.await();
         } catch (Exception e) { return;}
      }

      public Arbeider (int in) {ind = in;};

      public void run() {
         sync();
         System.out.println("A"+ind);
         sync();
         System.out.println("B"+ind);
      } // end run
   } // end indre klasse Arbeider
} // end class Problem
```

Question 8
Attached



```
import java.util.concurrent.*;
class Problem {
   // felles data og metoder A
   static int num = 2;
   static int extra = 2;
   static CyclicBarrier b;

   public static void main(String [] args) {
      Problem p = new Problem();
      num = Integer.parseInt(args[0]);
      extra = Integer.parseInt(args[1]);
      b = new CyclicBarrier(num);

      p.utfoer(num+extra); // extra threads
      System.out.println(" Main TERMINATED");
   } // end main

   void utfoer (int antT) {
      Thread [] t = new Thread [antT];
      for (int i =0; i< antT; i++)
         ( t[i] = new Thread(new Arbeider(i))).start();
     try {
        for (int i =0; i< antT; i++) t[i].join();
     } catch(Exception e) {}
   } // end utfoer

   class Arbeider implements Runnable {
      // lokale data og metoder B
      int ind;

      void sync() {
         try{
            b.await();
         } catch (Exception e) { return;}
      }

      public Arbeider (int in) {ind = in;};

      public void run() {
         sync();
         System.out.println("A"+ind);
         sync();
         System.out.println("B"+ind);
      } // end run
   } // end indre klasse Arbeider
} // end class Problem
```

Question 8
Attached



```
import java.util.concurrent.*;
class Problem {
   // felles data og metoder A
   static int num = 2;
   static int extra = 2;
   static CyclicBarrier b;

   public static void main(String [] args) {
      Problem p = new Problem();
      num = Integer.parseInt(args[0]);
      extra = Integer.parseInt(args[1]);
      b = new CyclicBarrier(num);

      p.utfoer(num+extra); // extra threads
      System.out.println(" Main TERMINATED");
   } // end main

   void utfoer (int antT) {
      Thread [] t = new Thread [antT];
      for (int i =0; i< antT; i++)
         ( t[i] = new Thread(new Arbeider(i))).start();
     try {
        for (int i =0; i< antT; i++) t[i].join();
     } catch(Exception e) {}
   } // end utfoer

   class Arbeider implements Runnable {
      // lokale data og metoder B
      int ind;

      void sync() {
         try{
            b.await();
         } catch (Exception e) { return;}
      }

      public Arbeider (int in) {ind = in;};

      public void run() {
         sync();
         System.out.println("A"+ind);
         sync();
         System.out.println("B"+ind);
      } // end run
   } // end indre klasse Arbeider
} // end class Problem
```

Suggested solutions INF2440 Spring 2018 Written Exam

Eric Jul

Question 1.1 the correct alternative is the last one.

Question 1.2 alternative 4 and 6 are correct, the rest are wrong.

Question 1.3 the third alternative is the correct one.

Question 1.4 the third alternative is the correct one.

Question 2.1

Yes, it will terminate. The program will generate two threads that will wait for each other at each call of sync because the cyclic barrier is initialized to two also. So first they will meet once at the barrier then when released, meet there again.

Question 2.2

No, the two threads created will synchronize at the cyclic barrier, but when released from the barrier, they compete and either one can proceed to the print statement first, so output can both be:

A0

A1

B1

B0

Main TERMINATED

Or

A1

A0

B1

B0

Main TERMINATED

Or two others similar where B1 and B0 are interchanged.

Note: Some have forgotten “ Main TERMINATED”

Question 2.3

No.

One example of where the program does not terminate:

Say thread 1, is VERY slow, then one execution sequence could be that thread 0 gets to the cyclic barrier first, then thread 2 whereafter they both print A0, A2. In the meantime thread 3 gets to the cyclic barrier the first time and blocks.

Then thread 0 gets to the cyclic barrier the second time and thereby releasing thread 0 and thread 3 from the barrier. Thread 3 prints A3 and then blocks at the barrier the second time.

Then thread 0 prints B0 and is done.

Then thread 2 reaches the barrier the second time – and since thread 3 is waiting there, they are both released.

Then thread 2 prints B2 and is done.

Then thread 3 prints B3 and is done.

Because thread 1 is very slow, only now does it reach the barrier the first time. As there is no other thread at the barrier, thread 1 blocks at the barrier. Because the three other threads are done, thread 1 will never proceed past the barrier.

Thus the program does not terminate.

Note: many conclude that the program terminates because, if the threads are executing at about the same speed, then, in most cases, the program WILL terminate. However, they overlook that we CANNOT assume that the threads execute at the same speed. Here is a version of the program where one of the threads is substantially slower than the other – by inserting a 2 second delay into one of the threads – and so it will, with a very high probability NOT terminate:

```
import java.util.concurrent.*;
class Problem {
  // felles data og metoder A
  static int num = 3;
  CyclicBarrier b = new CyclicBarrier(num);

  public static void main(String [] args) {
    Problem p = new Problem();
    p.utfoer(num+1); // num+1 == 4
    System.out.println(" Main TERMINATED");
  } // end main

  void utfoer (int antT) {
    Thread [] t = new Thread [antT];
    for (int i =0; i< antT; i++)
      ( t[i] = new Thread(new Arbeider(i))).start();
```

```
    try {
      for (int i =0; i< antT; i++) t[i].join();
    } catch(Exception e) {}
  } // end utfoer

  class Arbeider implements Runnable {
    // lokale data og metoder B
    int ind;

    void sync() {
      try{
        b.await();
      } catch (Exception e) { return;}
    }

    public Arbeider (int in) {ind = in;};

    public void run() {
      // kalles naar traaden er startet
      if (ind == 1) try {
        TimeUnit.SECONDS.sleep(2);
      } catch (Exception e) { return;};
      sync();
      System.out.println("A"+ind);
      sync();
      System.out.println("B"+ind);
    } // end run
  } // end indre klasse Arbeider
} // end class Problem
```

Question 2.4

The output is not always the same.

Here are two examples:

```
A3
A2
A0
B0
B3
B2
```

Where after the program blocks forever—thread one is waiting at the cyclic barrier while all the other threads have finished and so thread one will wait forever.

```
A0
A1
```

```
A2
A3
B0
B1
B2
B3
 Main TERMINATED
```

Where after the program terminates.

Note: many forget to include the printout from the main thread: “ Main TERMINATED”

Question 3.1

Attached program contains a possible sequential solution.

Note: Some lost points because they did not pay attention to getting the first deserts correct and/or the final desert.

Question 3.2

Attached program contains a possible parallel solution. It works by splitting the primes into parts, one for each thread, with approximately the same number of primes in each part. Then each thread find the prime deserts presents in its part – with due care to find deserts that span across parts – this is done by adjusting the parts to start and end at a prime – and by including the last prime of one part as the first prime of the next part. Each thread finds a list of prime deserts of monotonically increasing length within its part. When all threads are done, the main thread then combines the lists of deserts starting with the first part and including only those deserts that are strictly larger than the previous desert.

Censor notes:

- Some do not start the threads.
- Some do not initialize their lists.
- A few have problems synchronizing the threads at the end.
- Some have problems dividing up the primes so that all primes are included.
- Some have problems finding two deserts where the end point of the first is the same as the start point of the next AND this point is on the boundary between two parts allocated to two different threads. For example [2, 3] and [3, 5]
- Some use ArrayList that is operated on by multiple threads; it is NOT thread safe (in contrast to arrays where multiple threads can operate on disjunct parts of the array).
- Some have problems combining the results from each array at the end.

- Some have confusing representations of deserts; this is not a problem – except for those than managed to confuse themselves and thus make programming mistakes because of the confusing representation.

Question 4.1

Bubblesort can be parallelized in more than one way.

Method using parallel passes.

One way is to use K threads where K is 1-2 x the number of cores.

Each thread starts a pass of the Bubblesort algorithm, i.e., thread T0 bubbles from 0 to N, thread T1 from 0 to N-1, and so on. Once a thread has reached to index J in the array, the following thread can bubblesort up to J-1. However, the next thread must NOT touch anything from J and up. Because the threads execute asynchronously, T1 could be faster than T0 and get to – and even pass – element J.

One simple solution is to have a lock for every element (except the last). Then a thread Tm that wants to compare and, possibly, exchange elements K and K+1 will lock element K to prevent T(m+1) from accessing element K or any element thereafter. After the compare and, possibly, exchange Tm unlocks element K, continues on to K+1, which is then locked and so forth.

However, this solution is cost-prohibitive in that the number of synchronizations is the same as the number of comparisons. This can be reduced by locking R elements at a time and having at most one thread working on the R elements in such a segment. One suitable choice for R could be to set R to N/K/10, so that the number of synchronizations is limited to K*10 per pass and at least 9 out of 10 segments are free, so that there will be many segments available and thus less chance of a core being free and no thread for it. With K threads there can then be K threads simultaneously bubbling thru the array each in a segment. If one thread is a little faster than the others, it will be delayed at the next lock. If K is, say, 2x the number of cores, it is likely that there will be a thread that is able to execute – the faster threads will be delayed and the slower given a chance to barge ahead.

Each thread enters a new pass by locking an entry region.

In the entry region, it grabs the next pass number, grabs the lock of the first segment, and thereafter releasing the lock for the entry region. This ensures that threads are started in the order of the pass numbers. We identify the pass numbers by the number of elements that must be treated in the pass, i.e., the first pass is N because it need to bubble thru all elements from 0 to N-1 for a total of N elements.

For locking the entry region and each segment, we use a semaphore initialized to one.

Each thread then bubbles thru the first segment. At the end of the segment, it grabs the lock of the next segment, performs the check, and possible exchange, on the two

elements that span the two segments, then releasing the lock for the first segment. Thereafter it repeats this for the next segment until the last segment. Note that the number of segment decreases by one for every R passes. (The segment size could be reduced toward the end as to keep all cores active but then the synchronization overhead may become excessive.)

An alternative version is to split the array into K segments, one for each thread. Each thread, k, await that the thread, k-1, for the previous segment has done one pass, at which time k-1 performs the final check, and possible exchange with the last element of its segment, with the first element of the k'th segment. Thereafter it releases the k'th thread to perform one pass of the k'th segment. The k-1'th then goes back to the start of its segment and awaits that the k-2'th segment has completed another pass. Thread 0 can start a new pass at any time and does not wait for the previous (non-existent) pass. An extra lock is necessary between two segment, k-1, and k, as to ensure that the thread k-1 does not complete a pass before thread k has started the previous pass. . (The segment size could be reduced toward the end as to keep all cores active but then the synchronization overhead may become excessive.)

Both of these algorithms are faithful to the bubblesort algorithm – and both can achieve speedup for the parallel version.

Censor notes: many have merely divided the array into segment and then bubblesorted each segment using a thread.

- Some have then bubblesorted the entire array sequentially. This is a poor solution because the major part of the work is in the final bubblesort and thus little speedup is achieved. Actually, a speeddown is achieved in most cases.
- Some have then mergesorted the segments, which gives reasonable speedup but that is because mergesorting is much faster than bubblesort. This solution is not faithful to bubblesort.
- Some have then concatenated the segments two by two and then bubblesorted the doublesized segments using half as many threads. And thereafter concatenated and bubblesorted again until there is only one segment that is bubblesorted sequentially. This solution is faithful to bubblesort and so is worth more points than a non-faithful solution. But it is still not achieving any speedup because the last bubblesorts still has to do up to N/2 passes often taking ¾ of the time of the sequential version. So a speeddown results easily.

Censor notes: Across all solutions, there are problems with getting the boundaries right and also the actions required at the boundaries between sections. And in the merging, some cannot merge properly.

Question 4.2

The attached program contains a parallel version of Bubblesort using the parallelization described in 4.1.

Question 5.1 the correct alternatives are: 2,3,5,7,8. The rest are wrong.

Attached programs:

```
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

///---------------------------------------------------------
//     Fil: PrimeDesertPara.java
//     Implements Sequential and Parallel Prime Desert finding
//     written by: Eric Jul, University of Oslo, 2018
//
//     Cloned from:
//     Fil: EratosthenesSil.java
//     implements bit-array (Boolean) for prime numbers
//     written by:  Arne Maus , Univ of Oslo, 2013, 2015
//
//
//---------------------------------------------------------

/**
* Implements the bitArray of length 'bitLen' [0..bitLen ]
*   1 - true (is prime number)
*   0 - false
*  can be used up to 2 G Bits (integer range)
*/
public class PrimeDesertPara
{
   byte [] bitArr ;
   int bitLen;
   final  int [] bitMask ={1,2,4,8,16,32,64,128};
   final  int [] bitMask2 ={255-1,255-2,255-4,255-8,255-16,255-32,255-64,255-
128};
   CyclicBarrier readyToGo, allDone;


   PrimeDesertPara (int max) {
      bitLen = max;
      bitArr = new byte [(bitLen/16)+1];
      setAllPrime();
      generatePrimesByEratosthenes();
   }

   public static void main(String args[]) {
      PrimeDesertPara sil;
```

```
      if ( args.length != 2)  {
         System.out.println("use: >java PrimeDesertPara <Max>
<threadCount>");
         System.exit(0);
      }
      int num = Integer.parseInt(args[0]);
      int threadCount = Integer.parseInt(args[1]);
      if (!((num >= 5) && (threadCount >= 2))) {
         System.out.println("Bad parameters: Max must be at least 5 and
threadCount at least 2");
         System.exit(0);
      };

      // Here we generate the sil sequentially as we assumme that you already
have a parallel
      // version of the sil generation from Oblig 3
      long silTime = System.nanoTime();
      sil = new PrimeDesertPara(num);
      silTime = System.nanoTime() - silTime;
      System.out.println("Sil sequential generation time: " +
(silTime/1000000.0) + " ms");

      // now do the Prime Desert calculations
      sil.doit(sil, num, threadCount);

   } // end main()

   void doit(PrimeDesertPara sil, int num, int threadCount) {
      ArrayList<int[]> deserts = new ArrayList<int[]>();
      readyToGo = new CyclicBarrier(threadCount+1); // includes main() thread
      allDone = new CyclicBarrier(threadCount+1);

      // Sequential version of PrimeDesert
      long seqTime = System.nanoTime();  // Start sequential timing
      int[] desert = new int[2];
      desert[0] = 2;
      desert[1] = 3;
      deserts.add(desert);
      int desertLength = (3-2);
      int nextSP, nextEP, lastP;
      lastP = sil.lastPrime();
      nextSP = 3;
      nextEP = sil.nextPrime(nextSP);
      while (nextEP <= lastP) {
         if ((nextEP-nextSP) > desertLength) {
            desert = new int[2];
            desert[0] = nextSP;
            desert[1] = nextEP;
            deserts.add(desert);
            desertLength = nextEP-nextSP;
         }
         nextSP = nextEP;
         nextEP = sil.nextPrime(nextSP);
      };
      seqTime = System.nanoTime()-seqTime;
```

```
      System.out.println("Prime Desert Sequential time " +
(seqTime/1000000.0) + " ms\n");

      sil.printDeserts(deserts);



      // Parallel version

      long paraTime = System.nanoTime();

      // Create a desert list for each thread
      ArrayList<ArrayList<int[]>> desertLists = new
ArrayList<ArrayList<int[]>>();

      for (int i = 0; i < threadCount; i++) {
         desertLists.add(new ArrayList<int[]>());
      }

      // start threads
      int largestPrime = sil.lastPrime();
      int chunkSize = largestPrime/threadCount;
      int lastThread = threadCount - 1;
      for (int i = 0; i < threadCount; i++) {
         int from = i*chunkSize;
         int fromPrime = sil.nextPrime(from);
         int upto;
         if (i < lastThread) {upto = sil.nextPrime(from + chunkSize);} else
upto = largestPrime;
         System.out.println("Starting thread "+i+" from "+fromPrime+" upto
"+upto);
         new Thread(new Para(i, threadCount, sil, fromPrime, upto,
desertLists.get(i))).start();
      }

      try {
         readyToGo.await(); // await all threads ready to execute
      } catch (Exception e) {return;}

      // Now the threads are doing their thing

      try {
         allDone.await(); // await all worker threads DONE
      } catch (Exception e) {return;}

      // Combine results
      int currentLength = 0;
      int len;
      ArrayList<int[]> currentList;
      ArrayList<int[]> combinedList = new ArrayList<int[]>();
      for (int i = 0; i < threadCount; i++) {
         //System.out.println("Desert List " + i);
         currentList = desertLists.get(i);
         //sil.printDeserts(currentList);
         for (int j = 0; j < currentList.size(); j++) {
            len = currentList.get(j)[1] - currentList.get(j)[0];
            if (len > currentLength) {
```

```
                currentLength = len;
                combinedList.add(currentList.get(j));
             }
          }
       }


       paraTime = System.nanoTime()-paraTime;

       System.out.println("Prime Desert Parallel time " + (paraTime/1000000.0)
+ " ms\nSpeedup "+ seqTime*1.0/paraTime);

       sil.printDeserts(combinedList);

       for (int i = 0; i < threadCount; i++) {
          System.out.println("Desert List " + i);
          sil.printDeserts(desertLists.get(i));
       }

    }

    void printDeserts(ArrayList<int[]> deserts) {
       for (int i = 0; i < deserts.size(); i++) {
          int len = deserts.get(i)[1] - deserts.get(i)[0];
          System.out.println(" "+i+": ["+deserts.get(i)[0]+",
"+deserts.get(i)[1]+"] length: "+len);
       }
       System.out.println("------------------");
    }


    void setAllPrime() {
       for (int i = 0; i < bitArr.length; i++) {
          bitArr[i] = -1 ;     // alt    ( byte)255;
       }
    }

    void setNotPrime(int i) {
       bitArr[i/16] &= bitMask2[(i%16)>>1];
    }

    boolean isPrime (int i) {
       if (i == 2 ) return true;
       if ((i&1) == 0) return false;
       else  return (bitArr[i>>4] & bitMask[(i&15)>>1]) != 0;
    }

    ArrayList<Long> factorize (long num) {
       ArrayList <Long> fakt = new ArrayList <Long>();
       int maks = (int) Math.sqrt(num*1.0) +1;
       int pCand =2;

       while (num > 1 & pCand < maks) {
          while ( num % pCand == 0){
             fakt.add((long) pCand);
             num /= pCand;
          }
```

```
      pCand = nextPrime(pCand);
      // maks = (int) Math.sqrt(num*1.0) +1;
   }

   if (pCand>=maks) fakt.add(num);
   return fakt;
} // end factorize


int nextPrime(int i) {
   // returns next prime number after number 'i'

   int k;
   if (i < 2) return 2;
   if (i == 2) return 3;
   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

int lastPrime() {
   int j = ((bitLen>>1)<<1) -1;
   while (! isPrime(j) ) j-=2;
   return j;
} // end lastPrime

long largestLongFactorizedSafe () {
   long l;
   int i,j = ((bitLen>>1)<<1) -1;
   while (! isPrime(j) ) j -= 2;
   i = j-2;
   while (! isPrime(i) ) i -= 2;
   return (long)i*(long)j;
} // end largestLongFactorizedSafe


    void printAllPrimes(){
          for ( int i = 2; i <= bitLen; i++)
           if (isPrime(i)) System.out.println(" "+i);
    } // end printAllPrimes

    int numberOfPrimesLess(int n){
          int num = 2;  // we know 2 and 3 are primes
          int p ;

          for (p=3 ; p < n;p = nextPrime(p) ){
                num++;
          }
          return num;
    } // end numberOfPrimesLess


     void generatePrimesByEratosthenes() {
                int m = 3, m2=6,mm =9;     // next prime
                setNotPrime(1);      // 1 is not a prime
```

```
                    while ( mm < bitLen) {
                                m2 = m+m;
                                for ( int k = mm; k < bitLen; k +=m2){
                                      setNotPrime(k);
                                }
                                m = nextPrime(m);
                                mm= m*m;
                          }
              } // end generatePrimesByEratosthenes

   class Para implements Runnable {
      int ind, from, upto, threadCount;
      PrimeDesertPara sil;
      ArrayList<int[]> myDeserts;

      Para(int in, int c, PrimeDesertPara sil, int from, int upto,
ArrayList<int[]> myDeserts) {
         ind = in;
         threadCount = c;
         this.sil = sil;
         this.from = from;
         this.upto = upto;
         this.myDeserts = myDeserts;
      } // konstruktor

      public void run() { // Her er det som kjores i parallell:

      try {
         readyToGo.await(); // await all threads ready to execute
      } catch (Exception e) {return;}

      int[] desert = new int[2];
      desert[0] = from;
      desert[1] = sil.nextPrime(from);
      int desertLength = desert[1] - desert[0];
      myDeserts.add(desert);

      int nextSP, nextEP, lastP;
      lastP = upto;
      nextSP = desert[1];
      nextEP = sil.nextPrime(nextSP);
      while (nextEP <= lastP) {
         if ((nextEP-nextSP) > desertLength) {
            desert = new int[2];
            desert[0] = nextSP;
            desert[1] = nextEP;
            myDeserts.add(desert);
            desertLength = nextEP-nextSP;
         }
         nextSP = nextEP;
         if (ind < threadCount) {
            nextEP = sil.nextPrime(nextSP);
         } else if (nextEP < lastP) {
            nextEP = sil.nextPrime(nextSP);
         } else nextEP = lastP + 1;
      };
```

```
      // Done

      try {
         allDone.await(); // await all threads done
      } catch (Exception e) {return;}

      } // end run
   } // end class Para

} // end class PrimeDesert
```

```
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import java.util.concurrent.atomic.*;

///---------------------------------------------------------
//     Fil: BubblesortPara.java
//     Implements Sequential and Parallel BubbleSort
//     written by: Eric Jul, University of Oslo, 2018
//
//---------------------------------------------------------


public class BubblesortPara
{
   CyclicBarrier readyToGo, allDone;
   Semaphore lockEntryRegion;
   Semaphore[] sems;
   AtomicInteger passId;
   int[] arr;
   int segmentLength;


   BubblesortPara (int max) {

   } // end constructor BubblesortPara

   public void fillArr(int[] arr) {
      int seed = 123;
      int n = arr.length;
      Random r = new Random(seed);
      for (int i = 0; i < n; i++){
         arr[i] = r.nextInt(n);
      }
   } // end fillArr

   public void bubblesort(int[] arr) {
      int n = arr.length;
      int temp;
      for (int i=0; i < n; i++){
         for (int j=1; j < (n-i); j++){
            if(arr[j-1] > arr[j]){
               //swap elements
               temp = arr[j-1];
               arr[j-1] = arr[j];
               arr[j] = temp;
            }
         }
      }
   }

   public static void main(String args[]) throws InterruptedException {
      BubblesortPara b;
      final int min =1000;
```

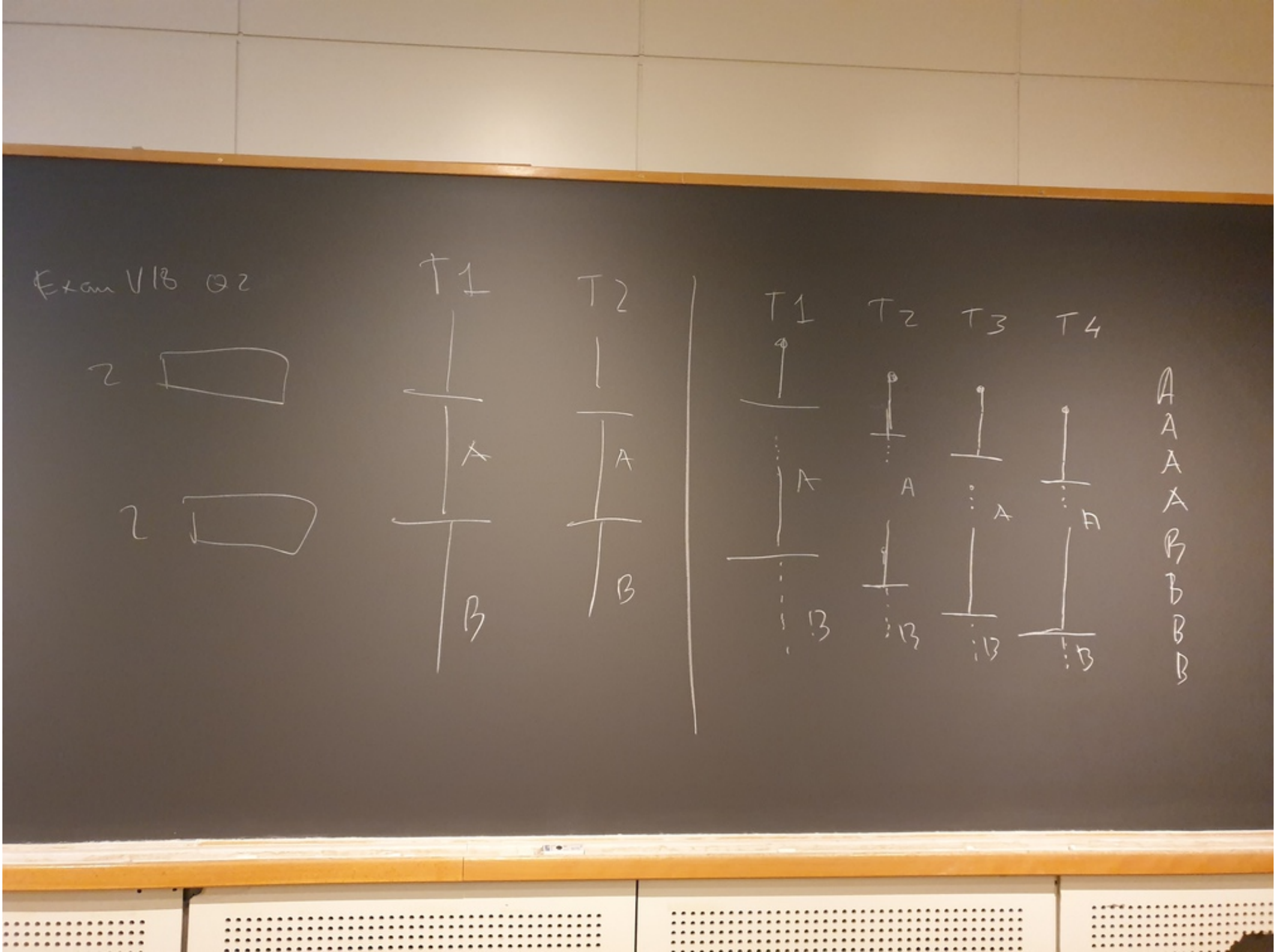
```
      if ( args.length != 2)  {
         System.out.println("use: >java BubblesortPara <Max> <threadCount>");
         System.exit(0);
      }
      int num = Integer.parseInt(args[0]);
      int threadCount = Integer.parseInt(args[1]);
      if (!((num >= min) && (threadCount >= 2))) {
         System.out.println("Bad parameters: Max must be at least "+min+" and
threadCount at least 2");
         System.exit(0);
      };

      b = new BubblesortPara(num);

      b.doit(b, num, threadCount);

   } // end main()

   void doit(BubblesortPara b, int num, int threadCount) throws
InterruptedException {
      arr = new int[num];

      // Sequential version of Bubblesort
      b.fillArr(arr);
      long seqTime = System.nanoTime();  // Start sequential timing
      b.bubblesort(arr);
      seqTime = System.nanoTime()-seqTime;

      System.out.println("Bubblesort Sequential time " + (seqTime/1000000.0)
+ " ms\n");

      // Parallel version
      b.fillArr(arr); // reset the array to original content.

      long paraTime = System.nanoTime();
      readyToGo = new CyclicBarrier(threadCount+1); // includes main() thread
      allDone = new CyclicBarrier(threadCount+1);
      lockEntryRegion = new Semaphore(1, true);

      // Divide the array into threadCount*10 segments
      // Create a Semaphore to protect each segment
      int segmentCount = threadCount * 10;
      segmentLength = num/segmentCount;
      //System.out.println("Segment length: "+segmentLength+"  segment count:
"+segmentCount);
      sems = new Semaphore[segmentCount];
      passId = new AtomicInteger(num);

      for (int k = 0; k < segmentCount; k++) sems[k] = new Semaphore(1,
true);

      // fill array.
      b.fillArr(arr);

      // start threads
      int lastThread = threadCount - 1;
      for (int i = 0; i < threadCount; i++) {
```

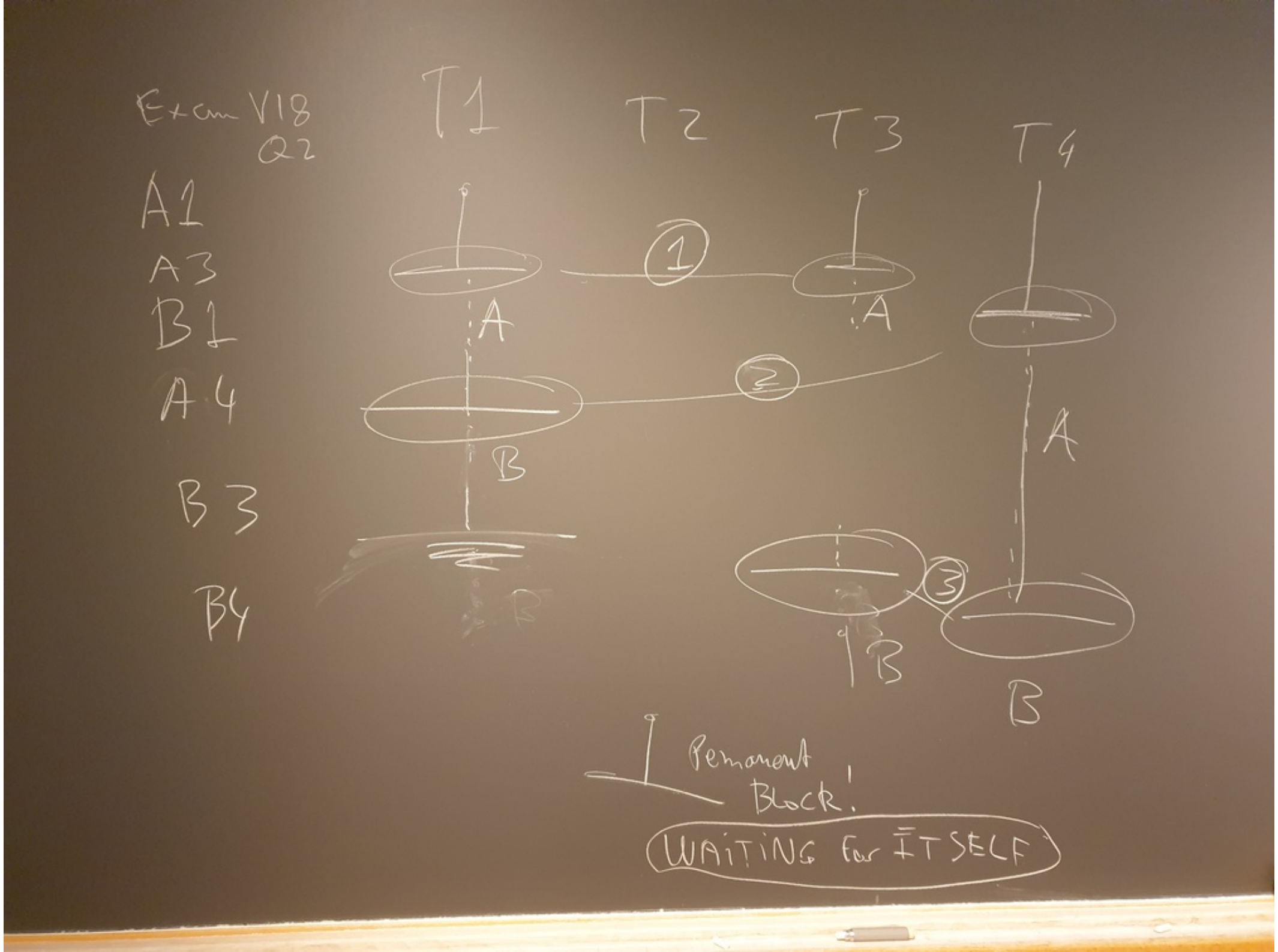
```
         //System.out.println("Starting thread "+i);

         new Thread(new Para(i, b, num)).start();
      }

      try {
         readyToGo.await(); // await all threads ready to execute
      } catch (Exception e) {return;}

      // Now the threads are doing their thing

      try {
         allDone.await(); // await all worker threads DONE
      } catch (Exception e) {return;}

      // Combine results


      paraTime = System.nanoTime()-paraTime;

      System.out.println("Bubblesort Parallel time " + (paraTime/1000000.0) +
" ms\nSpeedup "+ seqTime*1.0/paraTime);



   }

   class Para implements Runnable {
      int ind;
      BubblesortPara b;
      int myPassId;
      int num, currentSeg, myLastSeg, temp;

      Para(int in, BubblesortPara b, int num) {
         ind = in;
         this.b = b;
         this.num = num;
      } // konstruktor

      public void run() { // Her er det som kjores i parallell:

         try {
            readyToGo.await(); // await all threads ready to execute
         } catch (Exception e) {return;}

         // ********************** Thread code for parallel part
************************
         int currentSeg;
         //System.out.println("T"+ind);
         while (passId.get() > 1) {
            // enter entry region and grab next available pass
            try {
               lockEntryRegion.acquire();
            } catch (Exception e) {
               return;
            }
```

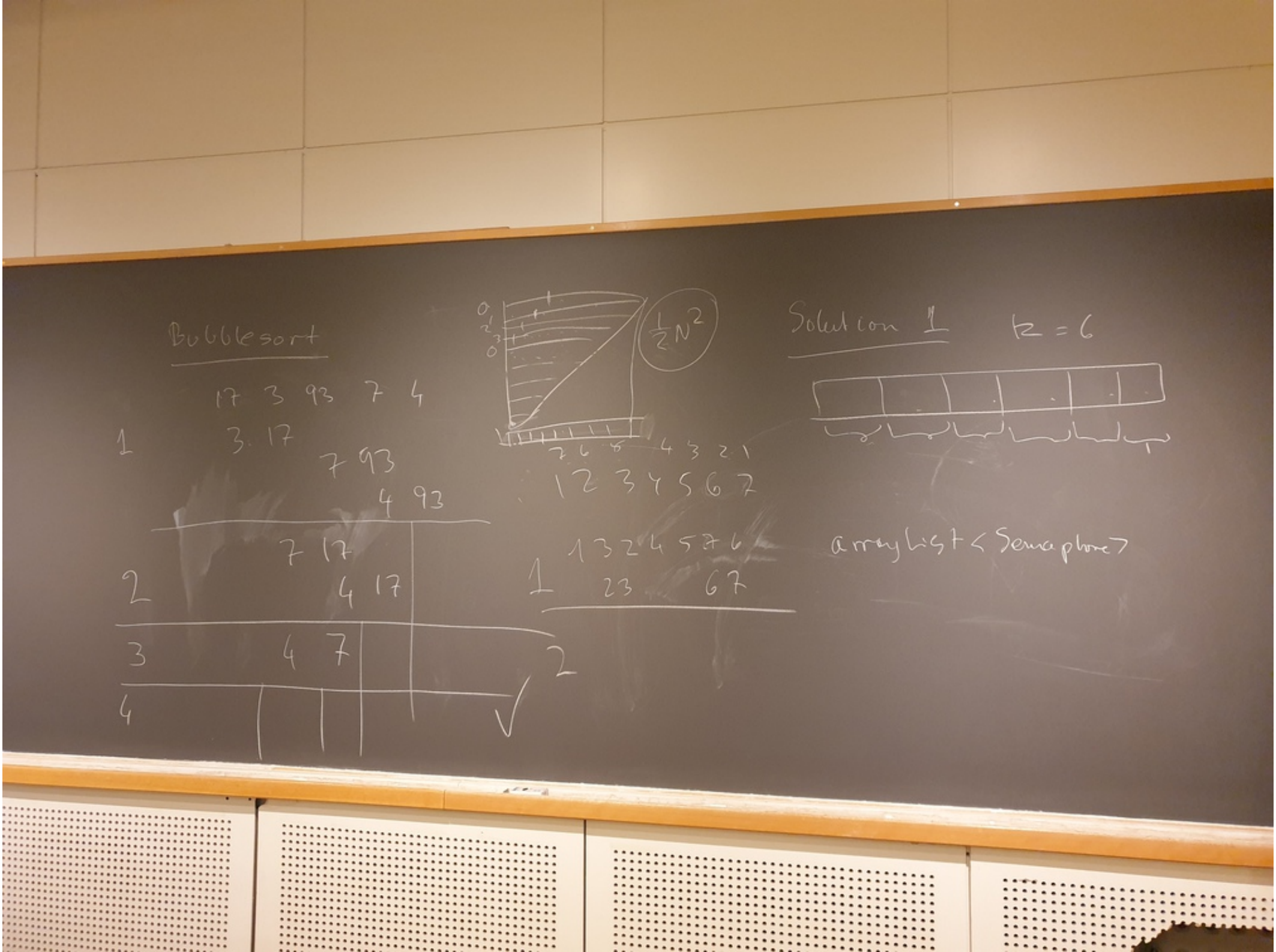
```
            myPassId = passId.getAndDecrement();
            //System.out.println("T"+ind+" got myPassId: "+myPassId);
            if (myPassId <= 1) {
                // others have done the work, so quit
                lockEntryRegion.release();
                break;
            }
            currentSeg = 0;
            try {
               sems[currentSeg].acquire();
            } catch (Exception e) {
               return;
            }

            //System.out.println("Seg length "+segmentLength);

            lockEntryRegion.release();
            // end of entry region - protected by lockEntryRegion

            // Now repeatedly bubble thru the segments
            myLastSeg = (myPassId-1)/segmentLength;

            for (int s = 0; s <= myLastSeg; s++) {
               int j;
               // for each segment do the bubbling
               int segEnd = (s+1)*segmentLength-1;
               if (segEnd >= myPassId) segEnd = myPassId-1;
               //System.out.println("T"+ind+" starting seg "+s+" segEnd:
"+segEnd+" myLastSeg "+myLastSeg);
               for (j = s*segmentLength+1; j <= segEnd; j++) {
                  if (arr[j-1] > arr[j]) {
                      // swap elements
                      temp = arr[j-1];
                      arr[j-1] = arr[j];
                      arr[j] = temp;
                  }
               }
               if (s < myLastSeg) {
                  // Must handle overlap with next segment at boundary, so
must lock both segments

                  try {
                     sems[s+1].acquire();
                  } catch (Exception e) {
                     return;
                  }

                  if (arr[j-1] > arr[j]) {
                      // swap elements
                      temp = arr[j-1];
                      arr[j-1] = arr[j];
                      arr[j] = temp;
                  }
               }
               // done with this segment so release lock
               //System.out.println("T"+ind+" releasing seg "+s);
               sems[s].release();
```

```
            } // for each segment
         }

         // ************************ Thread specific code done
*****************************

         try {
            allDone.await(); // await all threads done
         } catch (Exception e) {return;}

      } // end run
   } // end class Para

} // end class BubblesortPara
```

```
import java.util.concurrent.*;
class Problem {
   // felles data og metoder A
   static int num = 3;
   CyclicBarrier b = new CyclicBarrier(num);

   public static void main(String [] args) {
      Problem p = new Problem();
      p.utfoer(num+1); // num+1 == 4
      System.out.println(" Main TERMINATED");
   } // end main

   void utfoer (int antT) {
      Thread [] t = new Thread [antT];
      for (int i =0; i< antT; i++)
         ( t[i] = new Thread(new Arbeider(i))).start();
     try {
        for (int i =0; i< antT; i++) t[i].join();
     } catch(Exception e) {}
   } // end utfoer

   class Arbeider implements Runnable {
      // lokale data og metoder B
      int ind;

      void sync() {
         try{
            b.await();
         } catch (Exception e) { return;}
      }

      public Arbeider (int in) {ind = in;};

      public void run() {
         // kalles naar traaden er startet
         sync();
         System.out.println("A");
         sync();
         System.out.println("B");
      } // end run
   } // end indre klasse Arbeider
} // end class Problem
```


Exam VIB Q2
2
2
T1
T2
A
A
B
B
T1
T2
T3
T4
A
A
A
A
B
B
B
B
A
A
A
A
B
B
B
B


Exam V18
Q2
T1
T2
T3
T4
A1
A3
B1
A4
B3
B4
1
2
3
A
A
A
B
B
B
B
Permanent
Block!
WAITING for ITSELF


Bubblesort
17 3 93 7 4
1 3 17
7 93
4 93
7 17
2 4 17
3 4 7
4
2
$\frac{1}{2}N^2$
1 2 3 4 5 6 7
1 3 2 4 5 7 6
1 23 67
Solution 1
k = 6
arrayList<Semaphore>

Solution 2

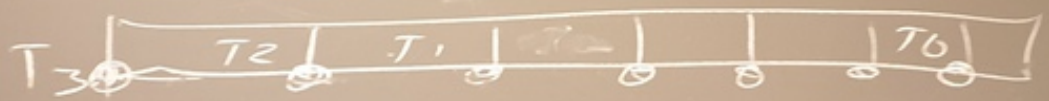
One thread per pass

+ N semaphores

$k = 2 \times C + 1$

N

100



```
static void bubbleSort(int[] arr) {
   int n = arr.length;
   int temp;
   for (int i=0; i < n; i++){
      for (int j=1; j < (n-i); j++){
          if(arr[j-1] > arr[j]){
             //swap elements
             temp = arr[j-1];
             arr[j-1] = arr[j];
             arr[j] = temp;
          }

      }
   }
}
```

```
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import java.util.concurrent.atomic.*;

///---------------------------------------------------------
//     Fil: BubblesortPara.java
//     Implements Sequential and Parallel BubbleSort
//     written by: Eric Jul, University of Oslo, 2018
//
//---------------------------------------------------------


public class BubblesortPara
{
   CyclicBarrier readyToGo, allDone;
   Semaphore lockEntryRegion;
   Semaphore[] sems;
   AtomicInteger passId;
   int[] arr;
   int segmentLength;


   BubblesortPara (int max) {

   } // end constructor BubblesortPara

   public void fillArr(int[] arr) {
      int seed = 123;
      int n = arr.length;
      Random r = new Random(seed);
      for (int i = 0; i < n; i++){
         arr[i] = r.nextInt(n);
      }
   } // end fillArr

   public Boolean checkSorted(int[] arr) {
      int n = arr.length;
      Boolean result = true;
      for (int i=1; i < n; i++) {
         if (arr[i-1] > arr[i]) {
            result = false;
            break;
         }
      }
      return result;
   }

   public void bubblesort(int[] arr) {
      int n = arr.length;
      int temp;
      for (int i=0; i < n; i++){
         for (int j=1; j < (n-i); j++){
            if(arr[j-1] > arr[j]){
```

```
                //swap elements
                temp = arr[j-1];
                arr[j-1] = arr[j];
                arr[j] = temp;
             }
          }
       }
   }

   public static void main(String args[]) throws
InterruptedException {
      BubblesortPara b;
      final int min =1000;

      if ( args.length != 2)      {
         System.out.println("use: >java BubblesortPara <Max>
<threadCount>");
         System.exit(0);
      }
      int num = Integer.parseInt(args[0]);
      int threadCount = Integer.parseInt(args[1]);
      if (!((num >= min) && (threadCount >= 2))) {
         System.out.println("Bad parameters: Max must be at least
"+min+" and threadCount at least 2");
         System.exit(0);
      };

      b = new BubblesortPara(num);

      b.doit(b, num, threadCount);

   } // end main()

   void doit(BubblesortPara b, int num, int threadCount) throws
InterruptedException {
      arr = new int[num];

      // Sequential version of Bubblesort
      b.fillArr(arr);
      long seqTime = System.nanoTime();  // Start sequential timing
      b.bubblesort(arr);
      seqTime = System.nanoTime()-seqTime;

      System.out.println("Bubblesort Sequential time " + (seqTime/
1000000.0) + " ms\n");

      // Parallel version
      b.fillArr(arr); // reset the array to original content.

      long paraTime = System.nanoTime();
      readyToGo = new CyclicBarrier(threadCount+1); // includes
main() thread
      allDone = new CyclicBarrier(threadCount+1);
      lockEntryRegion = new Semaphore(1, true);
```

```
      // Divide the array into threadCount*10 segments
      // Create a Semaphore to protect each segment
      int segmentCount = threadCount * 10;
      segmentLength = num/segmentCount;
      //System.out.println("Segment length: "+segmentLength+"
segment count: "+segmentCount);
      sems = new Semaphore[segmentCount];
      passId = new AtomicInteger(num);

      for (int k = 0; k < segmentCount; k++) sems[k] = new
Semaphore(1, true);

      // fill array.
      b.fillArr(arr);

      // start threads
      int lastThread = threadCount - 1;
      for (int i = 0; i < threadCount; i++) {

         //System.out.println("Starting thread "+i);

         new Thread(new Para(i, b, num)).start();
      }

      try {
         readyToGo.await(); // await all threads ready to execute
      } catch (Exception e) {return;}

      // Now the threads are doing their thing

      try {
         allDone.await(); // await all worker threads DONE
      } catch (Exception e) {return;}

      // Combine results


      paraTime = System.nanoTime()-paraTime;

      System.out.println("Bubblesort Parallel time " + (paraTime/
1000000.0) + " ms\nSpeedup "+ seqTime*1.0/paraTime);

      System.out.println("Check of monoticity: " +
b.checkSorted(arr));
   }

   class Para implements Runnable {
      int ind;
      BubblesortPara b;
      int myPassId;
      int num, currentSeg, myLastSeg, temp;

      Para(int in, BubblesortPara b, int num) {
```

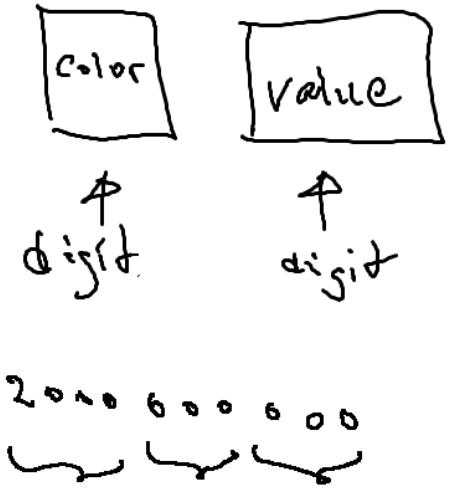
```
         ind = in;
         this.b = b;
         this.num = num;
      } // konstruktor

      public void run() { // Her er det som kjores i parallell:

         try {
            readyToGo.await(); // await all threads ready to execute
         } catch (Exception e) {return;}

         // ************************ Thread code for parallel part
*************************
         int currentSeg;
         //System.out.println("T"+ind);
         while (passId.get() > 1) {
            // enter entry region and grab next available pass
            try {
               lockEntryRegion.acquire();
            } catch (Exception e) {
               return;
            }
            myPassId = passId.getAndDecrement();
            //System.out.println("T"+ind+" got myPassId:
"+myPassId);
            if (myPassId <= 1) {
                // others have done the work, so quit
                lockEntryRegion.release();
                break;
            }
            currentSeg = 0;
            try {
               sems[currentSeg].acquire();
            } catch (Exception e) {
               return;
            }

            //System.out.println("Seg length "+segmentLength);

            lockEntryRegion.release();
            // end of entry region - protected by lockEntryRegion

            // Now repeatedly bubble thru the segments
            myLastSeg = (myPassId-1)/segmentLength;

            for (int s = 0; s <= myLastSeg; s++) {
               int j;
               // for each segment do the bubbling
               int segEnd = (s+1)*segmentLength-1;
               if (segEnd >= myPassId) segEnd = myPassId-1;
               //System.out.println("T"+ind+" starting seg "+s+"
segEnd: "+segEnd+" myLastSeg "+myLastSeg);
               for (j = s*segmentLength+1; j <= segEnd; j++) {
                  if (arr[j-1] > arr[j]) {
```

```
                        // swap elements
                        temp = arr[j-1];
                        arr[j-1] = arr[j];
                        arr[j] = temp;
                    }
                }
                if (s < myLastSeg) {
                    // Must handle overlap with next segment at
boundary, so must lock both segments

                    try {
                       sems[s+1].acquire();
                    } catch (Exception e) {
                       return;
                    }

                    if (arr[j-1] > arr[j]) {
                        // swap elements
                        temp = arr[j-1];
                        arr[j-1] = arr[j];
                        arr[j] = temp;
                    }
                }
                // done with this segment so release lock
                //System.out.println("T"+ind+" releasing seg "+s);
                sems[s].release();
             } // for each segment
          }

          // ************************ Thread specific code done
*****************************

          try {
             allDone.await(); // await all threads done
          } catch (Exception e) {return;}

       } // end run
    } // end class Para

} // end class BubblesortPara
```

Radix sorting

{ 170 061 512 503 693 703 154 275 765 087 897 677 908 509

Sorting Cards



0 ~~170~~
1 ~~061~~
2 ~~512~~
3 ~~503~~ 693 703
4 ~~154~~
5 ~~275~~ 765
6
7 ~~897~~ 677
8 ~~908~~
9 ~~509~~

FIRST PASS

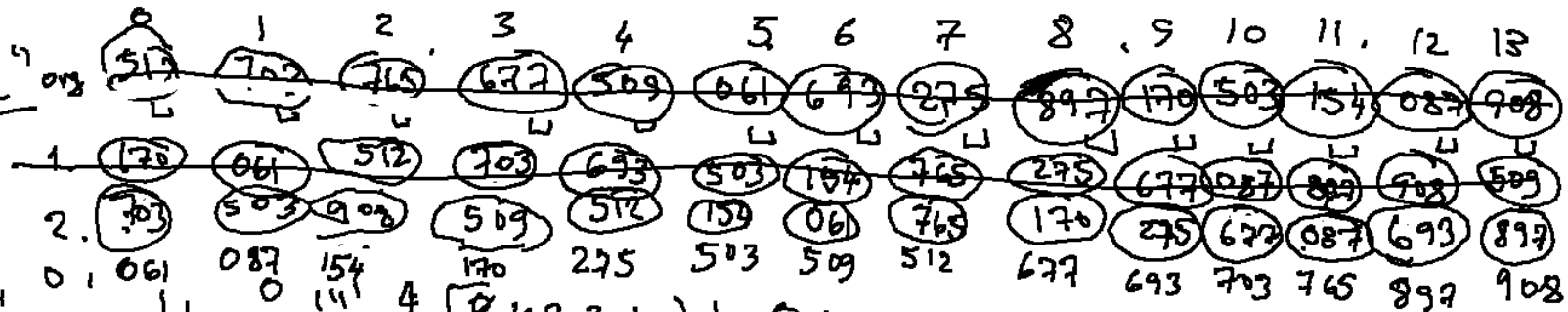
0 ~~503~~ 703 ~~908 509~~
1 ~~512~~
2
3
4
5 ~~154~~
6 ~~061~~ ~~765~~
7 ~~170~~ ~~275~~ ~~677~~
8
9 ~~693~~ ~~897~~

Second pass

0 061
1 154 170
2 275
3
4
5 503 509 512
6 677 693
7 703 765
8 897
9 908

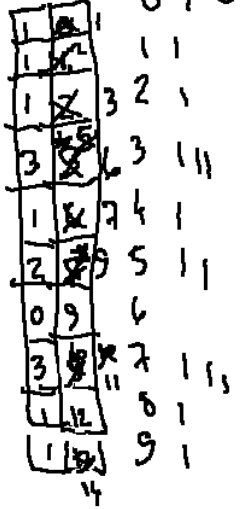
That's Pass

Making Radix "inplace" ons

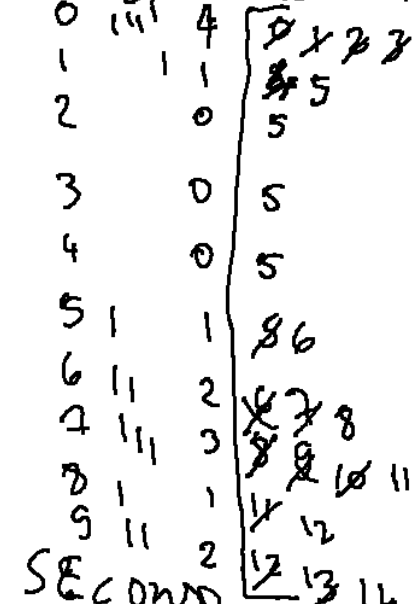


Steps

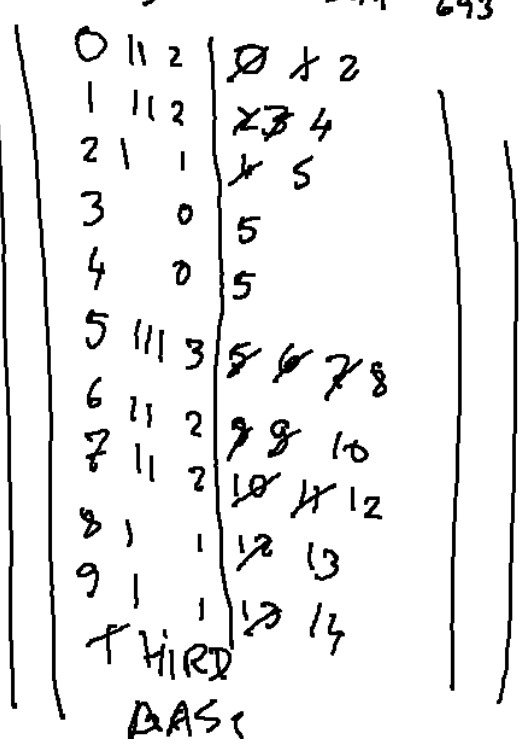
1. Find max of the array
   Compute # of digits
2. Count digit values
3. Place elements
4. Repeat for each pass



FIRST PASS



SECOND PASS



THIRD PASS

```
// Radix sort Java implementation
import java.io.*;
import java.util.*;

class Radix {
    // A utility function to get maximum value in arr[]
    static int getMax(int arr[], int n)
    {
        int mx = arr[0];
        for (int i = 1; i < n; i++)
            if (arr[i] > mx)
                mx = arr[i];
        return mx;
    }

    // A function to do counting sort of arr[] according to
    // the digit represented by exp. (eg. 300 is represented
by 100)
    static void countSort(int arr[], int n, int exp)
    {
        int output[] = new int[n]; // output array
        int i;
        int count[] = new int[10];
        Arrays.fill(count,0);

        // Store count of occurrences in count[]
        for (i = 0; i < n; i++)
            count[ (arr[i]/exp)%10 ]++;

        // Change count[i] so that count[i] now contains
        // actual position of this digit in output[]
        for (i = 1; i < 10; i++)
            count[i] += count[i - 1];

        // Build the output array
        for (i = n - 1; i >= 0; i--)
        {
            output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];
            count[ (arr[i]/exp)%10 ]--;
        }

        // Copy the output array to arr[], so that arr[] now
        // contains sorted numbers according to curent digit
        for (i = 0; i < n; i++)
            arr[i] = output[i];
    }

    // The main function to that sorts arr[] of size n using
    // Radix Sort
    static void radixsort(int arr[], int n)
```

```
    {
        // Find the maximum number to know number of digits
        int m = getMax(arr, n);

        // Do counting sort for every digit. Note that instead
        // of passing digit number, exp is passed. exp is 10^i
        // where i is current digit number
        for (int exp = 1; m/exp > 0; exp *= 10)
            countSort(arr, n, exp);
    }

    // A utility function to print an array
    static void print(int arr[], int n)
    {
        for (int i=0; i<n; i++)
            System.out.print(arr[i]+" ");
    }


    /*Driver function to check for above function*/
    public static void main (String[] args)
    {
        int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
        int n = arr.length;
        radixsort(arr, n);
        print(arr, n);
    }
}
```

Uke 12 – IN3030 v2019

Eric Jul

PSE-gruppa

Ifi, UiO

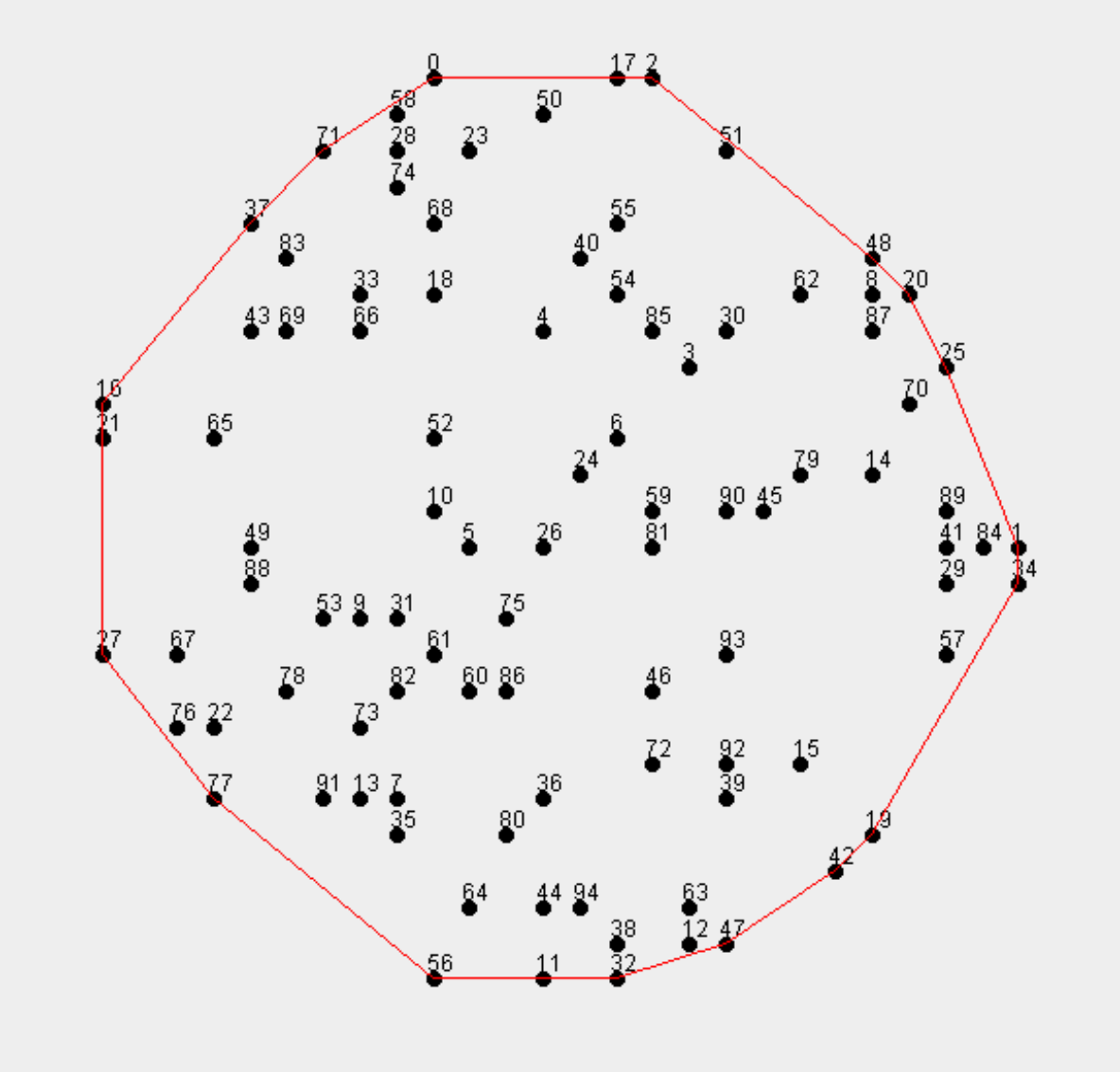
Oblig 5
Konvekse Innhyllinga
Introduksjon

Den konvekse innhyllinga til n punkter – Oblig 5

- Hva er det, definisjon
  - Hvordan ser den ut
- Hva brukes den til?
- Hvordan finner vi den?

1) Oblig 4, problemstilling

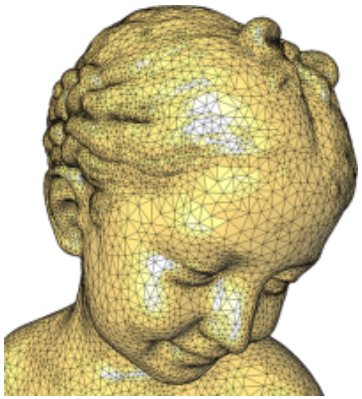
- Vi skal finne den konvekse innhyllinga til n punkter p i xy-planet.
- Her er 95 tilfeldige punkter og deres innhylling:
- en rekke med linjer fra punkt til punkt i mengden slik at alle andre punkter er på 'innsida' av denne mangekanten.
- Mangekanten er konveks, dvs alle indre vinkler er ≤ 180°
- Altså **ikke** linje 19-57 og 57-34.



. Alle punktene på en slik innhylling er med: Altså: linje 16-21 og 21-27, **ikke** linje direkte: 16-27.

Hva bruker vi den konvekse innhyllinga til?

- Innhyllinga er en helt nødvendig første steg i flere-stegs algoritmer innen :
  - Spillgrafikk (modellerering av flater , mennesker, ansikter, hus, borger, terreng, .. osv)
  - Kartografi
    - Høydekart over landskap
    - Sjøkart
    - volumberegninger innen olje-prospektering.



- Er f.eks. starten på en Delaunay-triangulering av punktene.

De etterfølgende figurer er laget i Geogebra . Anbefales sterkt (gratis) – last ned: http://geogebra.no/

Først en enkel geometrisk sats, I

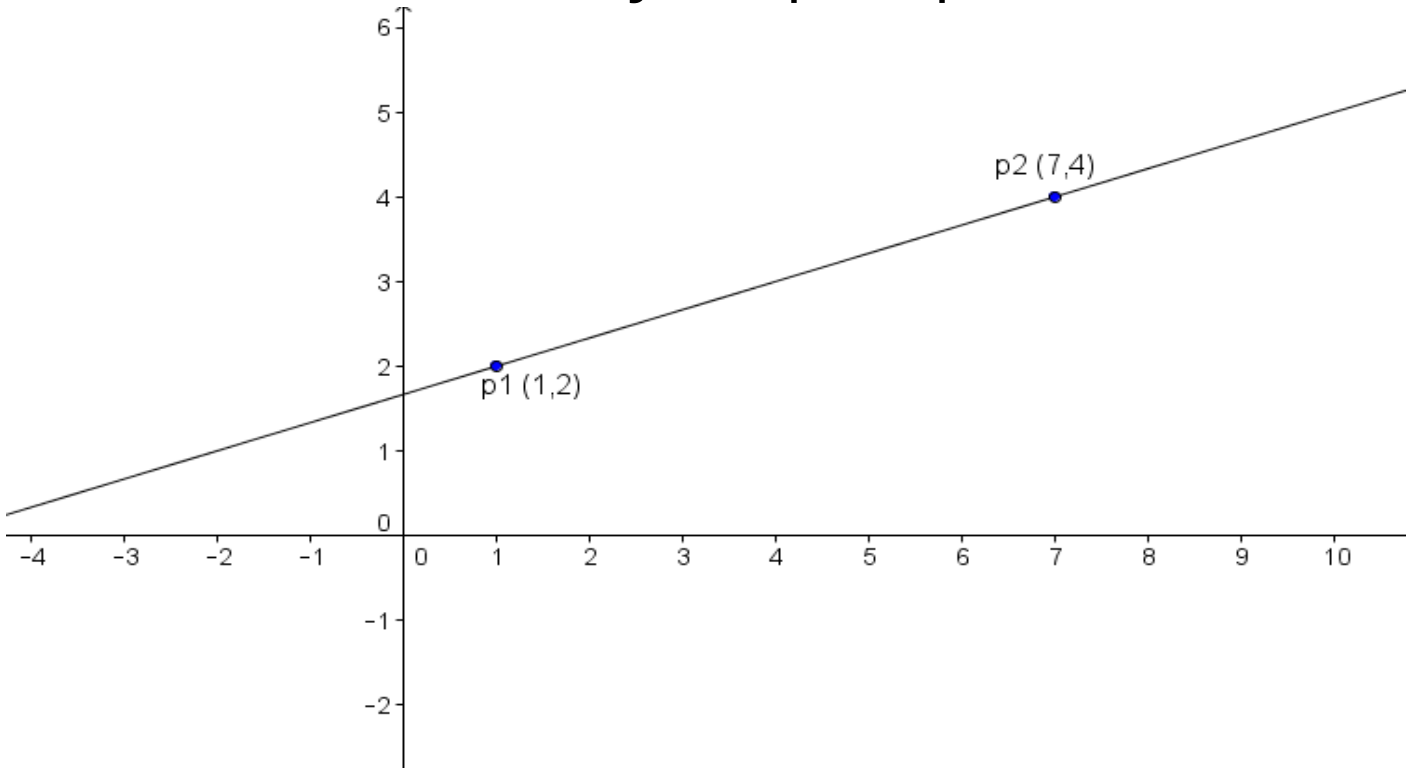
Ligningen for en linje

Enhver linje **fra** et punkt p1 (x1,y1) **til** p2(x2,y2) kan skrives på formen (trivielt forskjellig fra slik Geogebra gjør det):

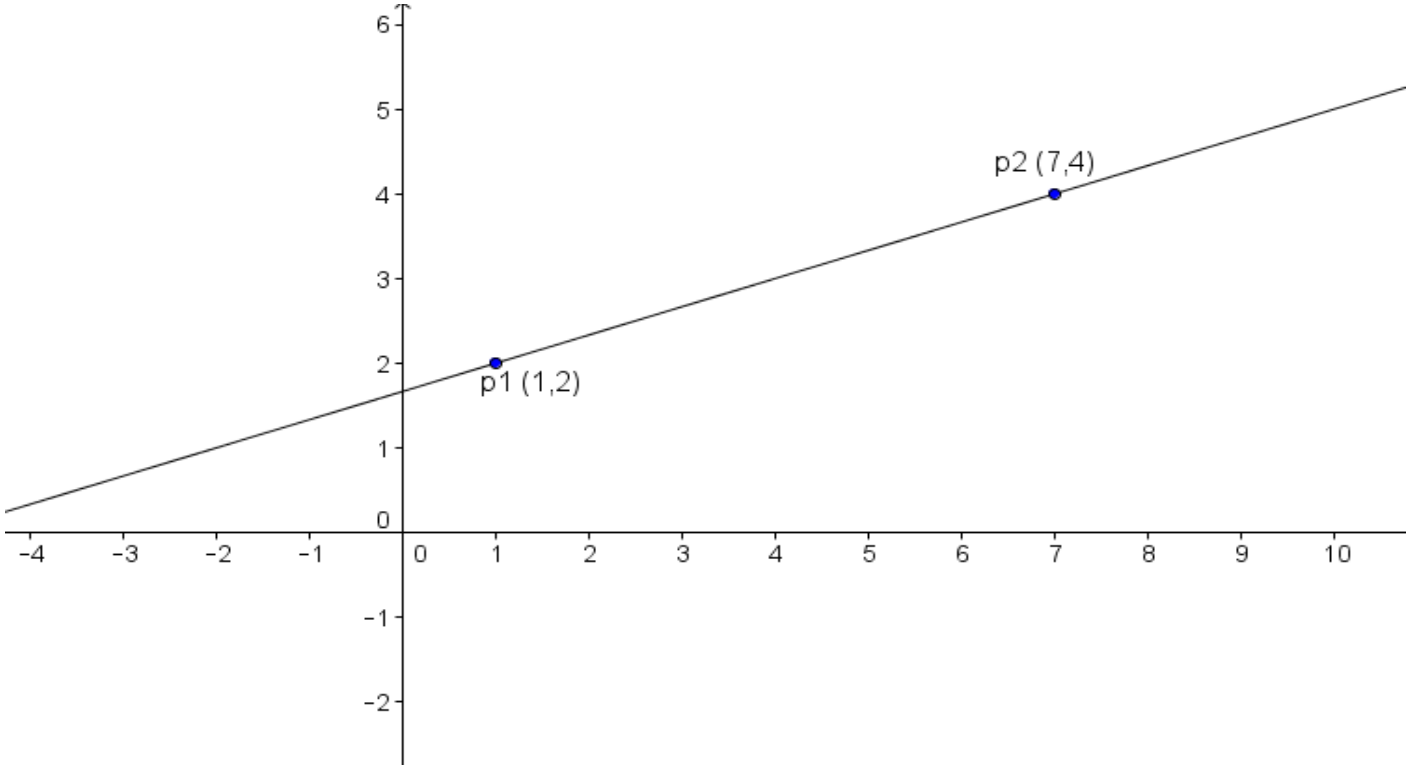
$$\boldsymbol{ax + by + c = 0}$$

Hvor: $a = y1 - y2,\ b = x2 - x1\ og\ c = y2 * x1 - y1 * x2.$

Merk at dette er en rettet linje *fra* p1 *til* p2.



Først en enkel geometrisk sats, II



Figur2. En linje fra p1 (1,2) til p2 (7,4) har da linjeligningen:

$$a = y1 - y2, \quad b = x2 - x1 \quad , \quad c = y2 * x1 - y1 * x2.$$

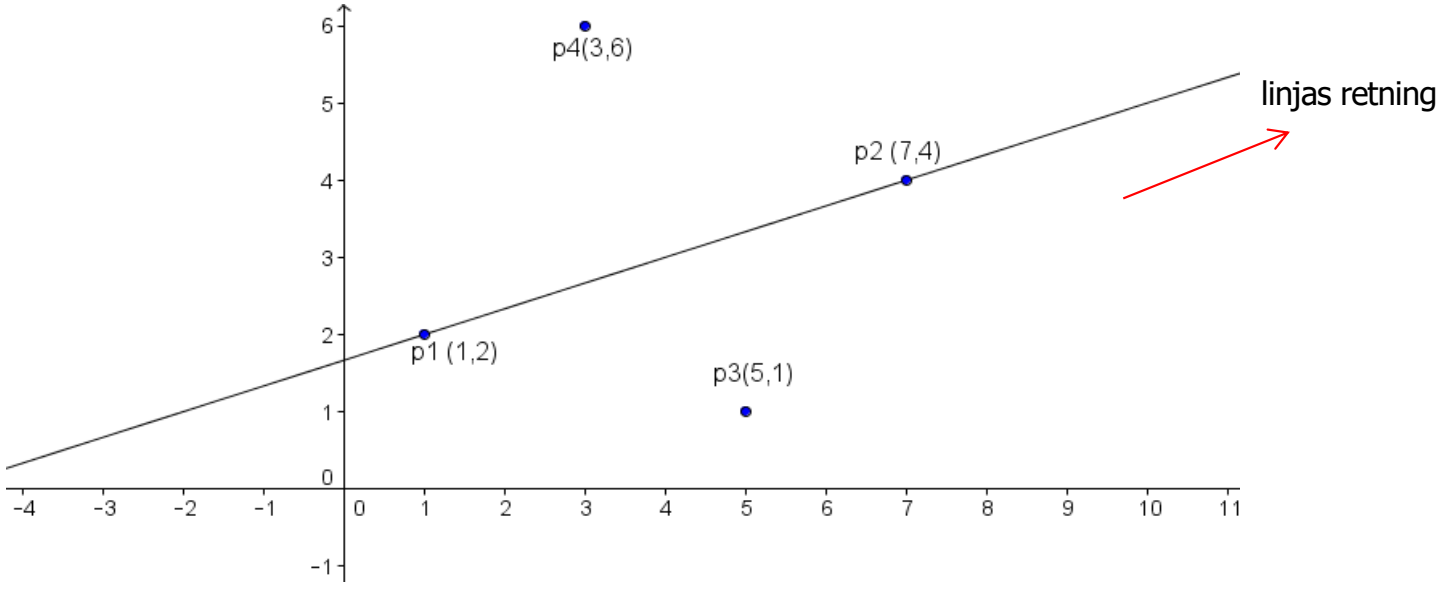
$$(2-4)x + (7-1)y + (4*1 - 2*7) = 0;\ dvs:\ \mathbf{-2x + 6y - 10 = 0}$$

Avstanden fra et punkt til en linje, I

- Setter vi ethvert punkt på p(px,py) linja, vil linjeligninga gi 0 som svar (per definisjon):

$$\boldsymbol{a} * \boldsymbol{px} + \boldsymbol{b} * \boldsymbol{py} + \boldsymbol{c} = \boldsymbol{0}$$

- Setter vi inn et punkt som **ikke** er på linja (p4 eller p3) vil vi få et tall som er :
  - negativt (<0) hvis punktet er til høyre for linja, sett i linjas retning: p1 til p2
  - positivt (>0) hvis punktet er til venstre for linja, sett i linjas retning: p1 til p2



Avstanden fra et punkt til en linje II

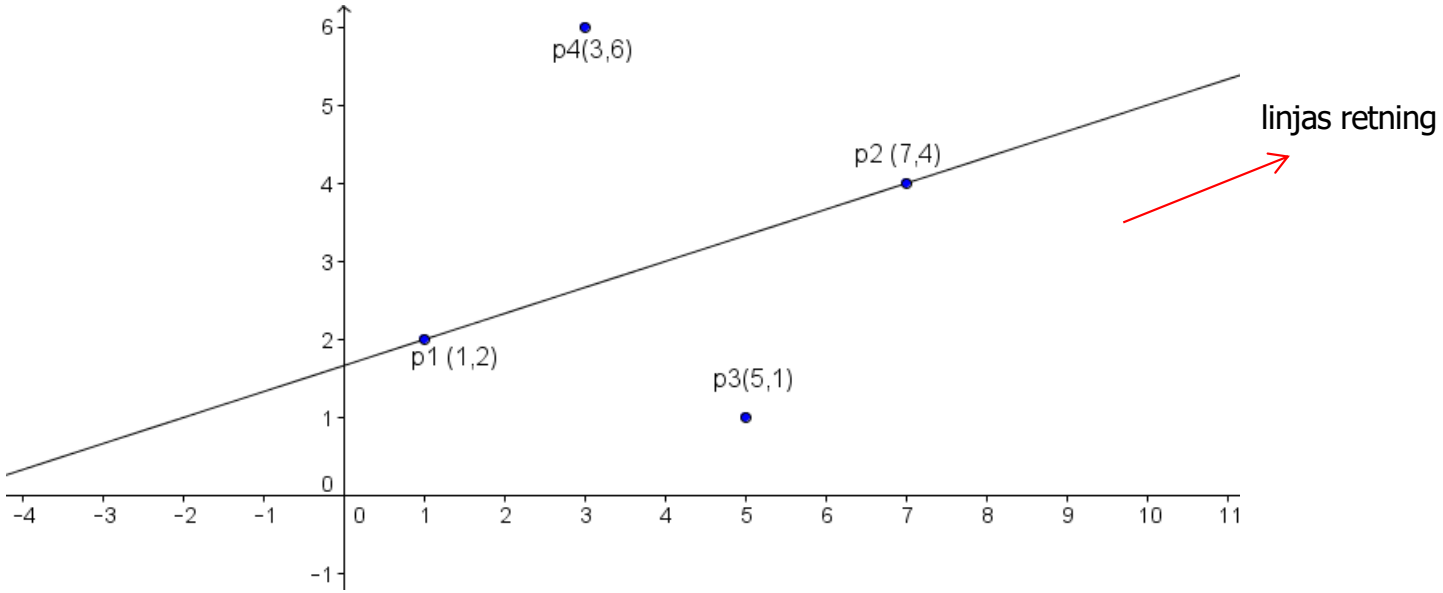
- Avstanden fra et punkt (x,y)til en linje (vinkelrett ned på linja) er :

$$d = \frac{ax + by + c}{\sqrt{a^2 + b^2}}$$

- Jo lenger fra linja punktene et desto større negative og positive tall blir det.

Setter inn **p3** (5,1) i linja p1-p2: -2x+6y-10 = 0, får vi

d= : $\frac{-2*5+6*1-10}{\sqrt{40}} = \frac{-14}{6{,}32} = -2{,}21..$



En linje deler da planet i to: Punktene til høyre og til venstre for linja (sett fra rettet linje fra p1 til p2)

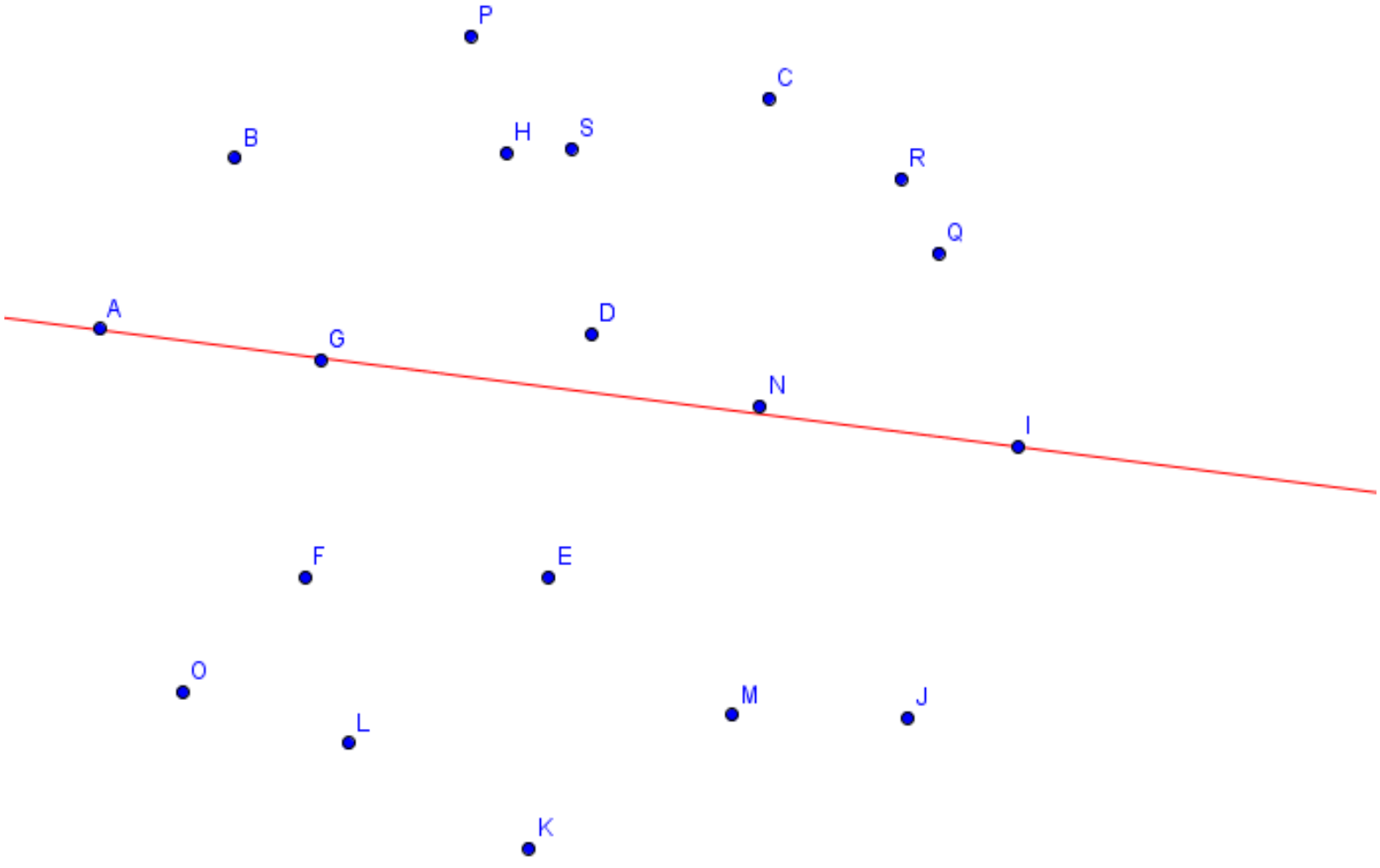
- Vi er nå interessert i de punktene som ligger lengst fra én gitt linje (til høyre for den) $\boldsymbol{ax + by + c = 0}$ i en stor punktmengde.

- Kan da avstandsformelen forenkles – gjøres raskere ?

$$d = \frac{ax + by + c}{\sqrt{a^2 + b^2}}$$

To observasjoner:

- Punktene med minst og størst x-verdi (A og I) ligger på den konvekse innhyllinga
- De punktetene som ligger lengst fra (positivt og negativt) enhver linje p1-p2, er to punkter på den konvekse innhyllinga.(P og K)



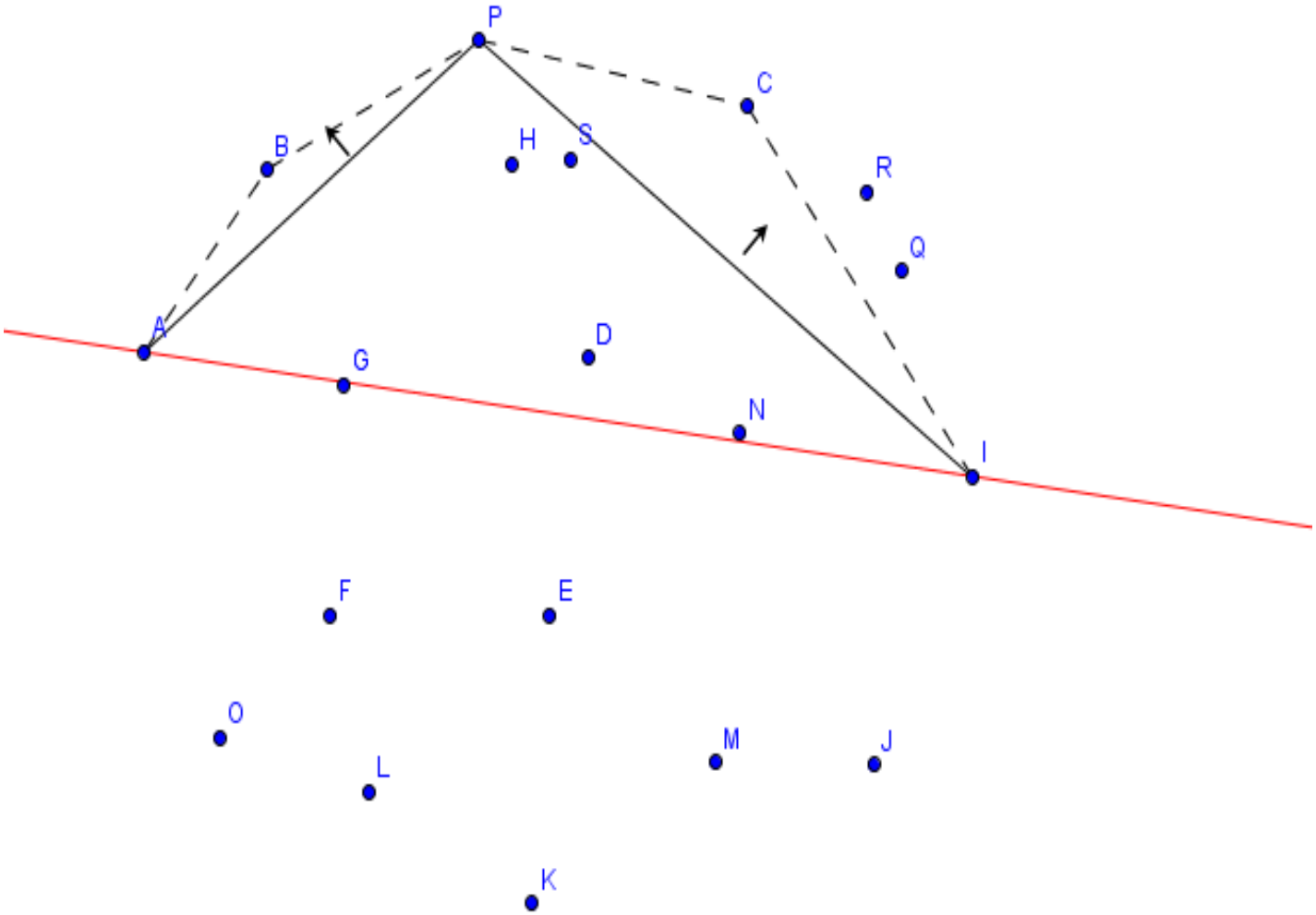
Vi skal etter starten av algoritmen bare se på det punktet som ligger i mest negativ avstand fra linja (dvs mest til-høyre for linja)

Algoritmen for å finne den konvekse innhyllinga sekvensielt

1. Trekk linja mellom de to punktene vi vet er på innhyllinga fra maxx -minx (I - A).
2. Finn punktet med størst negativ (kan være 0) avstand fra linja (i fig 4 er det P). Flere punkter samme avstand, velg vi bare ett av dem.
3. Trekk linjene fra p1 og p2 til dette nye punktet p3 på innhyllinga (neste lysark: I-P og P-A).
4. Fortsett rekursivt fra de to nye linjene og for hver av disse finn nytt punkt på innhyllinga i størst negativ avstand (≤ 0).
5. Gjenta pkt. 3 og 4 til det ikke er flere punkter på utsida av disse linjene.
6. Gjenta steg 2-5 for linja minx-maxx (A-I) og finn alle punkter på innhyllinga under denne.

Rekursiv løsning: Finn først P (mest neg. 'avstand' fra I-A)

Trekk så I-P og finn C, Trekk så I-C , og finn R. trekk så I-R og finn Q. Finner så intet 'over' R-C eller C-P. Trekker P-A og finner så B over. Ferdig.



Problemer dere vil møte i den rekursive, sekvensielle løsningen I

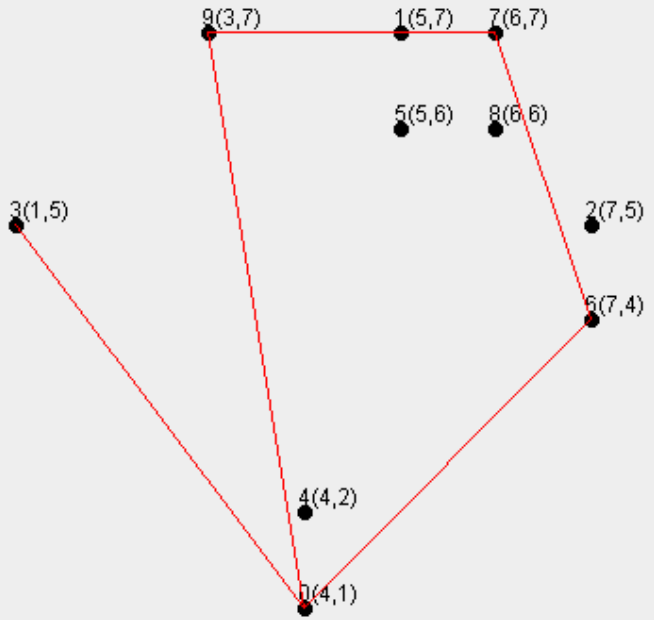
- **Hvordan representere et punkt p_i ?**
  - Med indeksen 'i' (ikke med koordinatene x og y) ?
- **Debugging** (alle gjør feil først) av et grafisk problem vil vi ha tegnet ut punktene og vårt beste forsøk på konvekse innhylling.

Klassen TegnUt (hvis n < 250)
Brukes slik fra main-tråden:

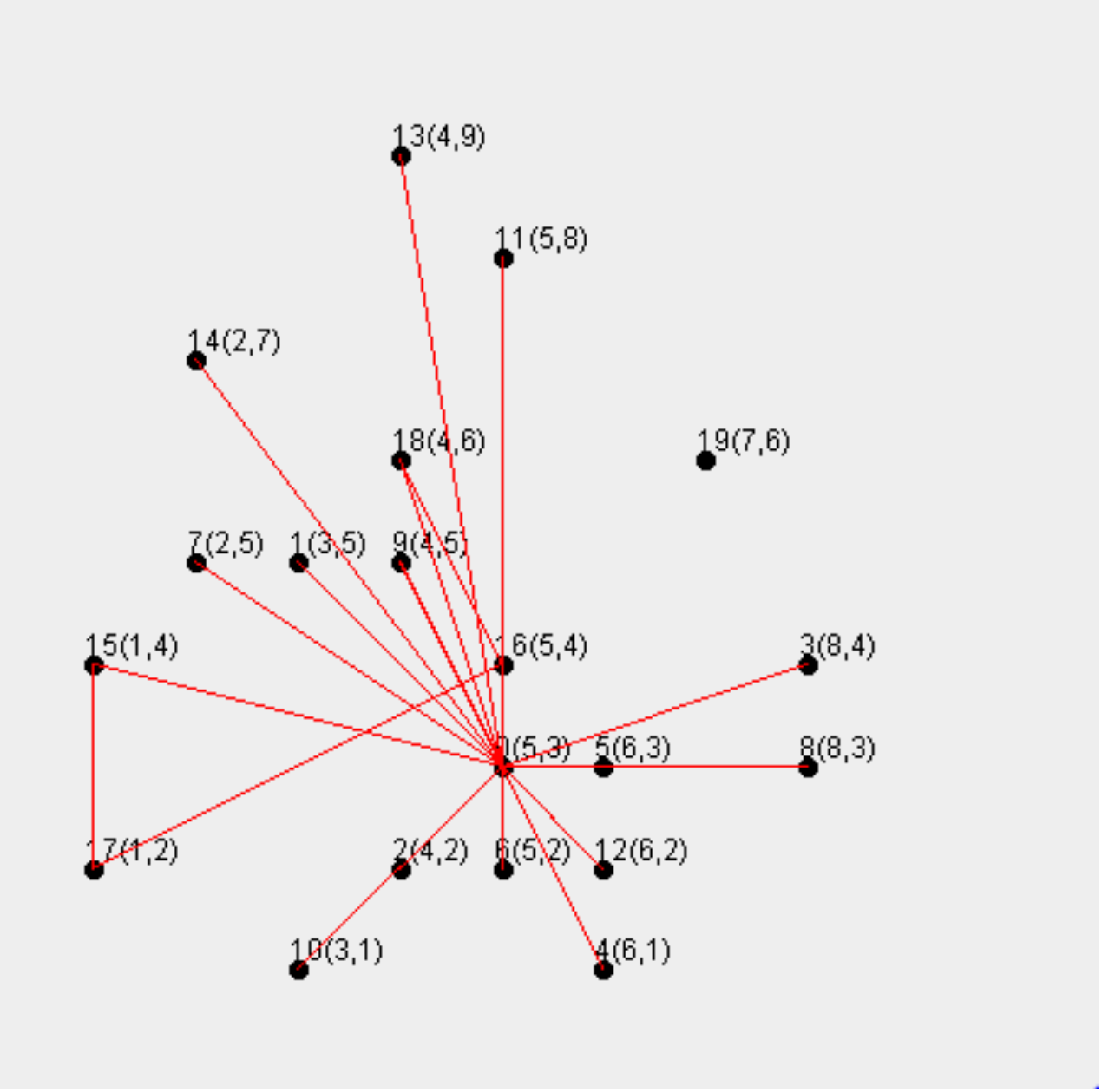
```
TegnUt tu = new TegnUt (this, koHyll);
```

a) TegnUt tegner ut punktene og innhyllinga i en IntList koHyll. Skrives trivielt om av deg hvis du bruker ArrayList. 'this' er en peker til main-objektet.
b) TegnUt antar at main-objektet er et objekt av klassen Oblig5.
c) Ikke nødvendigvis 'proff' kode i klassen TegnUt

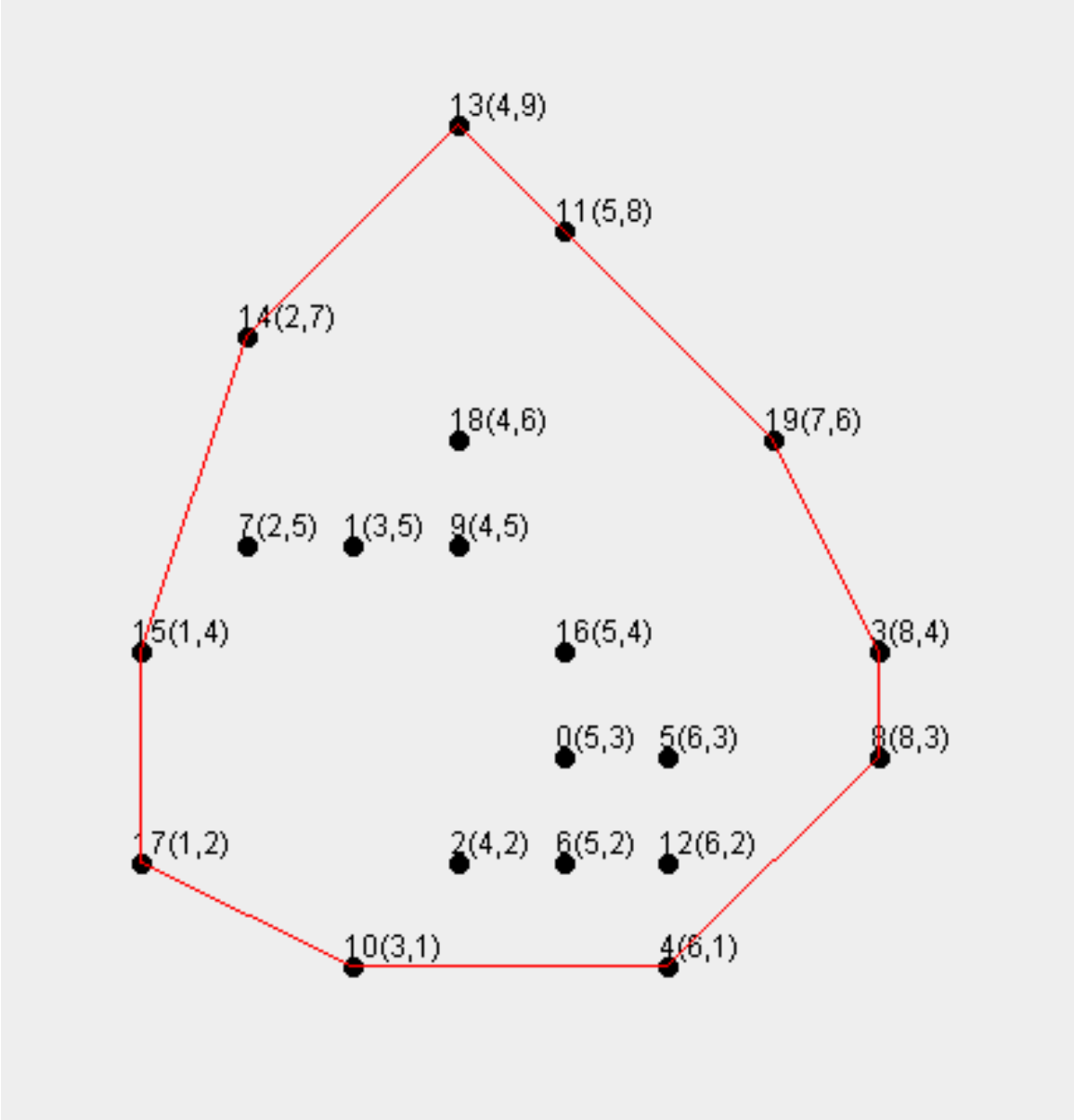
«Litt» feil:



Mye feil:


13(4,9)
11(5,8)
14(2,7)
18(4,6)
19(7,6)
7(2,5)
1(3,5)
9(4,5)
15(1,4)
16(5,4)
3(8,4)
5(6,3)
8(8,3)
17(1,2)
2(4,2)
12(6,2)
10(3,1)
4(6,1)

Riktig

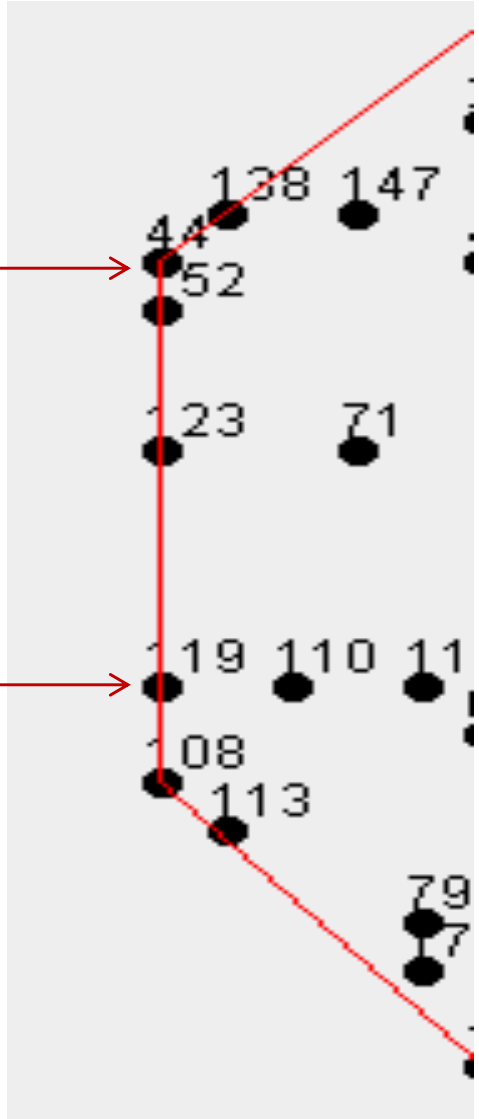

13(4,9)
11(5,8)
14(2,7)
18(4,6)
19(7,6)
7(2,5)
1(3,5)
9(4,5)
15(1,4)
16(5,4)
3(8,4)
0(5,3)
5(6,3)
8(8,3)
17(1,2)
2(4,2)
6(5,2)
12(6,2)
10(3,1)
4(6,1)

Problemer dere vil møte i den rekursive, sekvensielle løsningen II

- **Finne punktene på den konvekse innhyllinga i riktig rekkefølge**?
  - Tips: Du bruker to metoder for det sekvensielle tilfellet:
    - sekvMetode() som finner minx, maxx og starter rekursjonen. Starter rekursjonen med to kall på den rekursive metoden, først på maxx-minx, så minx-maxx:
    - sekvRek (int p1, int p2, int p3, IntList m)
      som, inneholder alle punktene som ligger på eller under linja p1-p2, og p3 er allerede funnet som det punktet med størst negativ avstand fra p1-p2.
      IntList m er en mengde punkter som ligger over (til høyre for) linje p1-p2
  - Du kan la sekvRek legge inn ett punkt: p3 i innhyllinga-lista, men hvor i koden er det ?
  - Når legges minx inn i innhyllingslista ?

Problemer dere vil møte i den rekursive, sekvensielle løsningen III

- **Få med alle punktene på innhyllinga hvor flere/mange ligger på samme linje (i avstand = 0), og få dem i riktig rekkefølge.**
- Tips:
  - Husk at når du finner at største negative avstand er = 0 må du ikke inkluderer p1 eller p2 som mulig nytt punkt (de er allerede funnet)
  - Si at du har funnet p1=44 og p2=119. Du bør da bare være interessert i å finne de punktene som ligger mellom p1 og p2 på linja (52 og 123), og da må du teste om nytt punkt p3 har både y og x-koordinater *mellom* tilsvarende koordinater for p1 og p2.
  - Da finner du ett av punktene (si: 123) med kall på sekRek over linja p1-p2 (44-119). Gjenta rekursivt (over 44-123) og 123-119) til det ikke lenger er noen punkter mellom nye p1 og p2.
  - Punktene videre nedetter linja (f.eks. 119-108) finnes av rekursjon tilsvarende som for 44 og 119.



INF3030 Uke 14, v2019

Eric Jul
PSE,
Inst. for informatikk

Resten av INF3030 – v2019

- (uke 13 – ingen forelesning)
- Denne forelesningen (uke14)
  - Mer om hvordan parallellisere ulike problemer
- 3. mai
  - Ingen forelesning – jobbar med Oblig
- 10. mai
  - Ingen forelesning – Oblig 5 deadline!
- 23. mai – review av kurs, perspektiv (uke 17)
  - Litt om eksamen
  - Gjennomgang av tidligere eksamen
- 7. juni, 4 timer eksamen
  - Tillatt å ha med all skriftlig materiale
  - Ha med utskrift av alle forelesningene (definerer pensum) og Obligene med dine løsninger

Hva skal vi se på i Uke14

I) Om program, samtidige kall og synlighet av data

- Når en tråd (main eller en av de andre) aksesserer data, hvilke er det.
- Kan de parallelle trådene og main kalle samme metode samtidig. Kan de kalle metoder fra en sekvensiell løsning? Hva skjer da ?
- Hvor mange stack-er (stabler) har vi; hva er det, og hvilke har vi?

II) Mer om to store programmer, og hvordan nå den siste av dem kan parallelliseres.

- Delaunay triangulering – de beste trekantene!
  - Brukes ved kartlegging, oljeleting, bølgekraftverk,..
  - Spill-grafikk: ved å gi tekstur, farge og glatte overflater på gjenstander, personer, våpen osv.
  - Er egentlig flere algoritmer etter hverandre. Skissering av hvordan disse kan parallelliseres.

III) Parallellisering av Oblig 5 - den konvekse innhyllinga (DKI)

- Hva er forventet O() – kompleksitet av DKI
- To strategier for parallellisering - roadmap
- Idag : rekursiv parallellisering
- Hva kan ventes av speedup på oblig 5

II) Om program, samtidige kall og synlighet av data

- Når en tråd (main eller en av de andre) aksesserer data, hvilke er det?
- Kan vi stoppe en annen tråd ?
  - Hvis ikke vi, hvem da?
- Hvor mange stack-er har vi, hva er det og hvilke har vi?
- Kan flere tråder kalle samme metode samtidig.
  - Hva skjer da ?

Når en tråd (main eller en av de andre) aksesserer (lesere/skriver) data, hvilke data er det?

- Svar: Det vanlige skopet (utsynet til deklarasjoner):
  - først lokale variable og i metoden og parametre
  - så variable og metoder i klassen man er inne i
  - Evt. så i en eller flere ytre klasser
- Dette gjelder alltid uansett hvilken tråd som kaller metoden
  - Kallstedet har sitt skop
  - Utførelsesstedet har sitt skop.

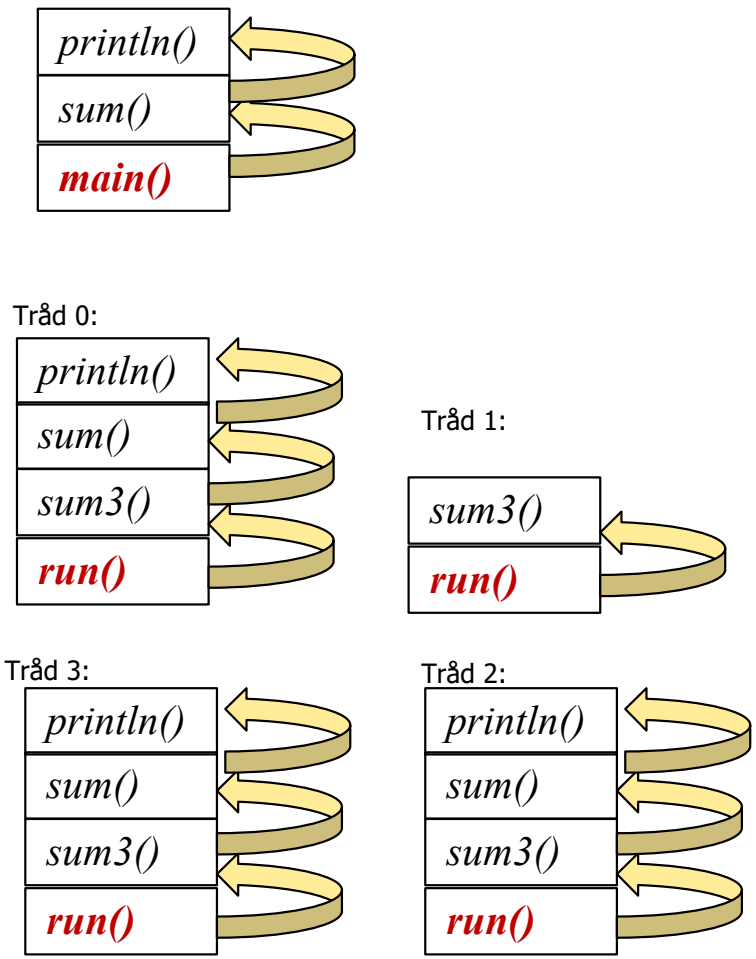
```
import java.util.concurrent.*;
class Problem {
    int [] a = new int[100]; // fyll med verdier
    long s;
     public static void main(String [] args) {
          Problem p = new Problem();
          p.utfoer(12);
     }
     void utfoer (int antT) {
          Thread [] t = new Thread [antT];
          for (int i =0; i< antT; i++)
              ( t[i] = new Thread(new Arbeider(i))).start();
           long s = sum(a);
           long t = t[0].sum2( t[0].b);

           for (int i =0; i< antT; i++) t[i].join();
     }
     long sum (int a[]) { s=0 ;
         for (int i = 0; i<a.length;i++) s += a[i];
         return s;
     }

     class Arbeider implements Runnable {
              int ind;
               int [] b= new int[200]; // fyll b[]
               long sum2 (int a[]) {long s=0 ;
                          for (int i = 0; i<a.length;i++) s += a[i];
                   return s;
               }
               long sum3(int [] c) { return sum(c);}

              Arbeider (int in) {ind = in;)
                    public void run(int ind) {
                       long p = sum(a);
                       long q = sum2(b);
                       long r = sum3(a);
                     } // end run
        } // end indre klasse Arbeider
} // end class Problem
```

Kan vi stoppe en annen tråd ? Hvis ikke vi, hvem da?

- Helst ingen, forbudt
  - I java 1.0 var det riktignok metoder som:
    - stop() **Deprecated**. This method is inherently unsafe. Stopping a thread with Thread.stop causes it to unlock all of the monitors that it has locked ….
    - suspend() **Deprecated**. This method has been deprecated, as it is inherently deadlock-prone.
  - Disse vil bli fjernet og skaper bare problemer – **ikke** bruk dem!
- Bare trådene kan stoppe seg selv
  - midlertidig ved synkronisering
  - ved å gå ut av siste setning i sin main() eller run() metode
- En tråd kan starte andre tråder, men de må stoppe/terminere seg selv

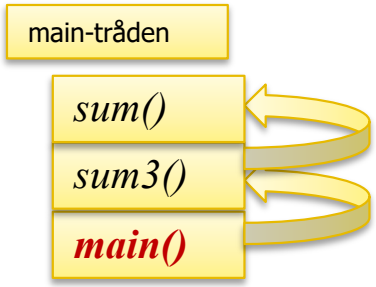
Hva er en stack (stabel)? Hvor mange har vi og hvilke er det?

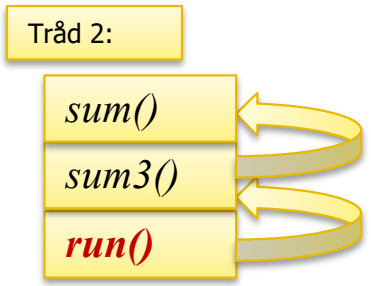
- En stabel er et dataområde som holder de metodene (med deres lokale variable og parametre) som er kalt.
  - En metode kan kalle en annen metode,
    - Da legger det nye metodeobjektet seg oppå det som kalte,..osv
    - Det hele er som en tallerken-stabel
  - Det er bare den metoden som er på toppen av en stabel som gjør noe nå.
    - De lenger nede venter bare på at den som ligger rett over den skal returnere,...osv
- Hvert trådobjekt (også objektet med main) har hver sin stabel.
  - nederst i hver av disse stablene ligger hhv. main() og run()
- Java er et multi-stabel språk (ikke alle språk er det)
- Stakkene ender seg hele tiden etter som metoder kalles og returnerer



Kan flere tråder kalle samme metode samtidig. Hva skjer da ?

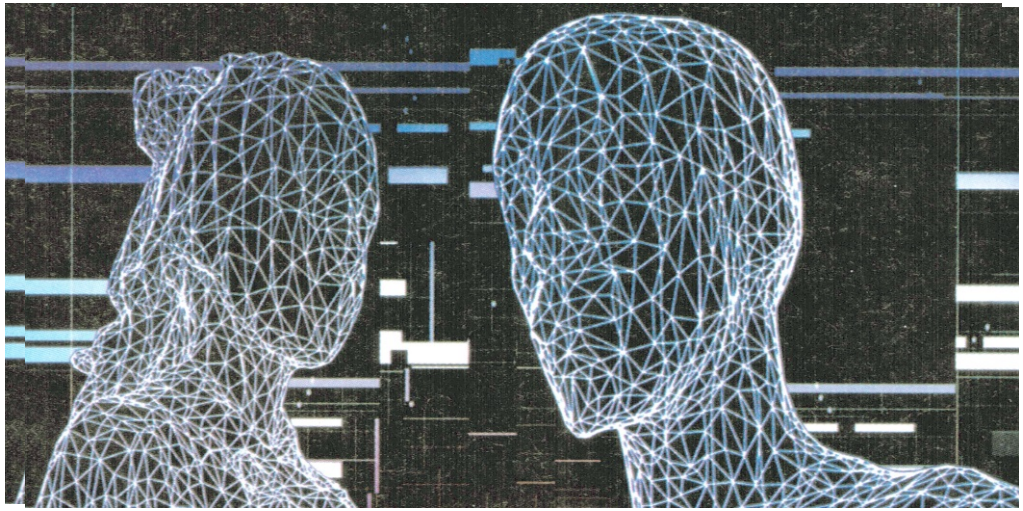
- Anta at både main-tråden og tråd 2 kalle metoden sum3() som igjen kaller sum().
- Da legger det på hver av stablene en metode-objekt av hhv. sum3() og oppå det hvert sitt sum()-objekt
- **Svaret er JA**. Samtidige metodekall til samme metode skaper ingen problemer.
- Metodene 'bare ligger der' og kan kalles av alle andre metoder som har utsyn til dem, enten:
  - Gjennom sitt skop på kallstedet
  - Eller via pekere

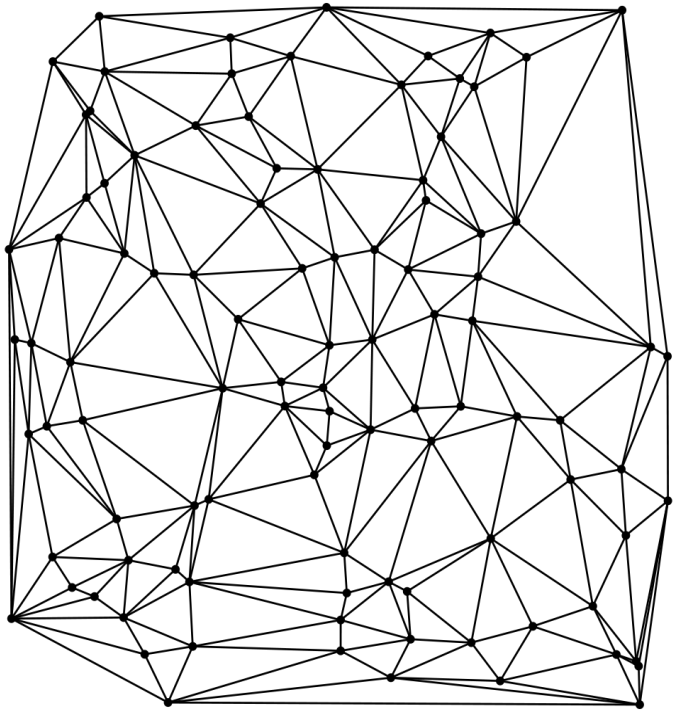




II) Triangulering – å lage en flate ut fra noen målinger

- Av og til vil vi representere noen målinger i 'naturen' og lage en kunstig, kontinuerlig flate:
  - Oljeleting – topp/bunn av oljeførende lag
  - Kart – fjell og daler, sjøkart
  - Grafiske figurer:
    - Personer, våpen, hus,..
- (x,y) er posisjonen, mens z er høyden
- Vi kan velge mellom :
  - Firkanter – det er vanskelige flater i en firkant (vridde)
  - Trekanter – best, definerer et rett plan
- Rette plan kan lettest glattes for å få jevne overganger til naboflater.





Mye av dette arbeidet hviler på:

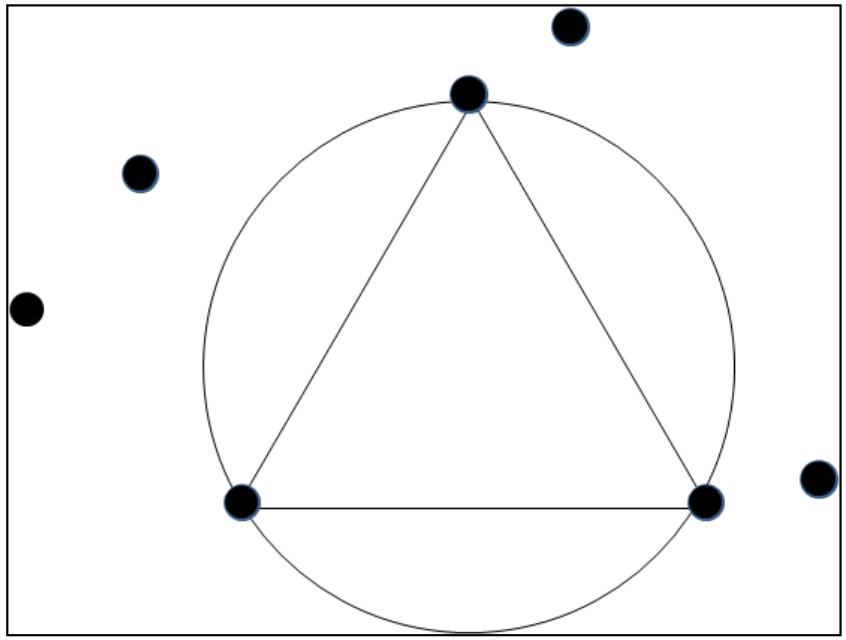
- En Delaunay flatemodell Arne Maus laget i 1980-84 på NR (Kartografi). Simula og Fortran 77. Solgt bla. til et bølgekraftverkprosjekt (Sintef) og Oljedirektoratet.
  - Maus, Arne. (1984). Delaunay triangulation and the convex hull of n points in expected linear time. *BIT 24,* 151-163.
- Masteroppgave: Jon Moen Drange: « Parallell Delaunay-triangulering i Java og CUDA.» Ifi, UiO, mai 2010
  - A. Maus og J.M. Drange: «All closest neighbours are proper Delaunay edges generalized, and its application to parallel algorithms», NIK 2010, Gjøvik, Tapir, 2010
- Masteroppgave: Peter Ludvik Eidsvik: «PRP-2014: Parallelle, faseoppdelte rekursive algoritmer» Ifi, UiO, mai 2014
- Masteroppgave: Erik Thune Lund: «Implementing High-Performance Delaunay Triangulation in Java.», Ifi, des. 2014
- Arne Maus, ny Delaunay-algoritme 2014-2016

Delaunay triangulering (1934)

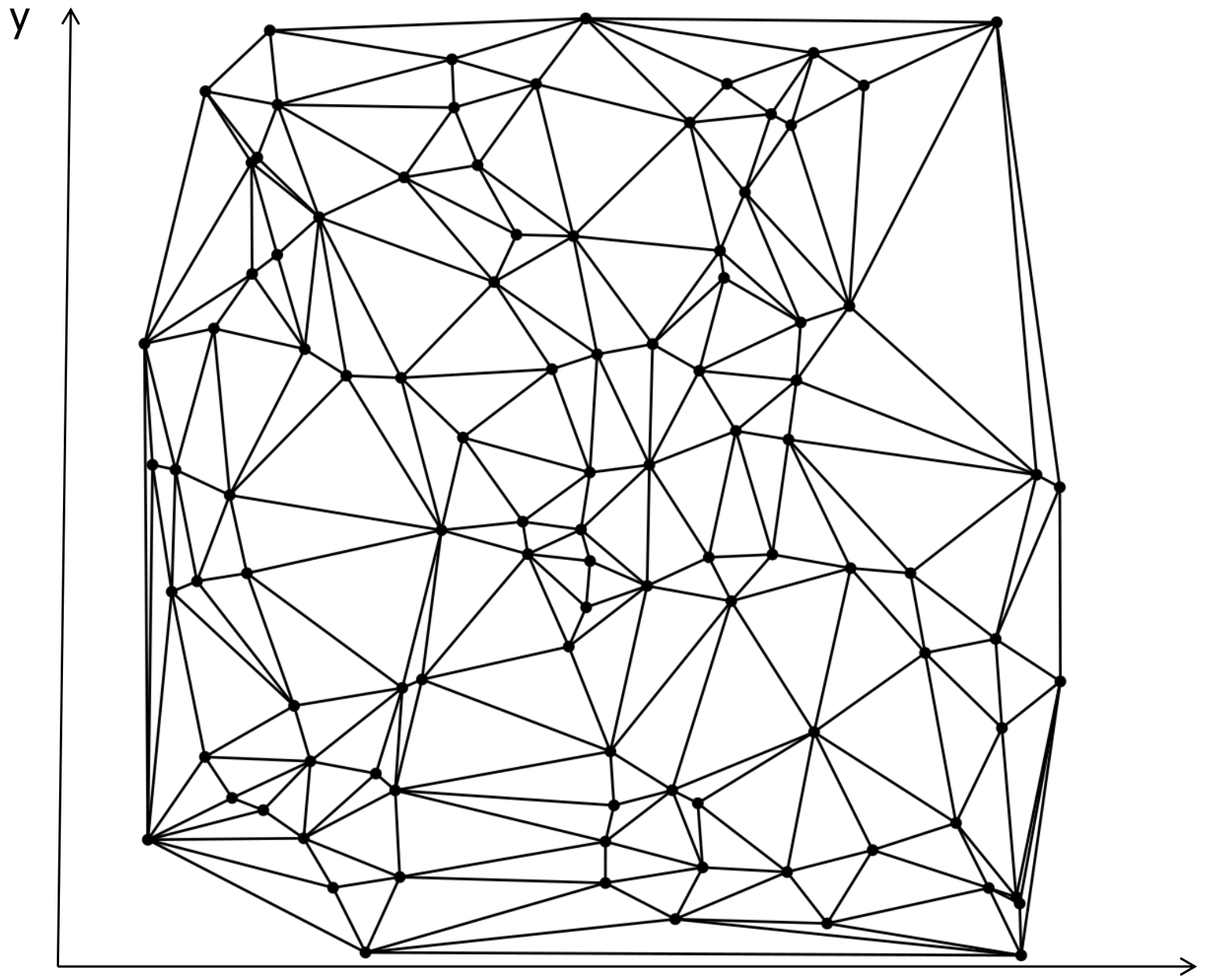
Boris Nikolaevich Delaunay 1890 – 1980, russisk fjellklatrer og matematiker
(etterkommer etter en fransk offiser som ble tatt til fange under Napoleons invasjon av Russland, 1812)

Vi har n punkter i planet
Forbind disse punktene med hverandre med et trekantnett slik at:

- Ingen linjer (trekantsider) krysser hverandre
- Man lager de 'beste' trekantene (maksimerer den minste vinkelen, dvs. færrest lange og tynne trekanter)
- **Def:** Den omskrevne sirkelen for tre de hjørnene i enhver trekant inneholder ingen av de (andre) punktene i sitt indre

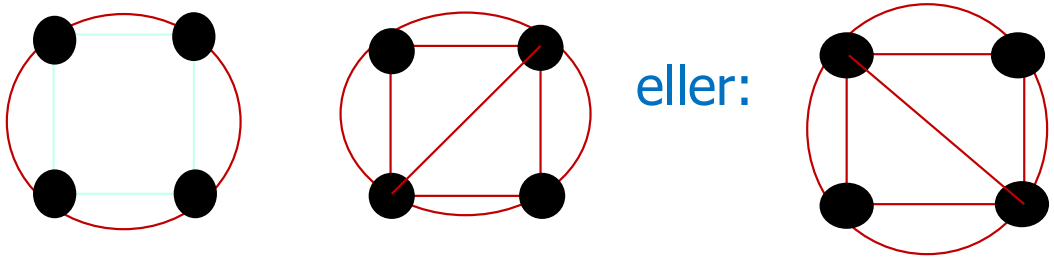


Delaunay triangulering av 100 tilfeldige punkter

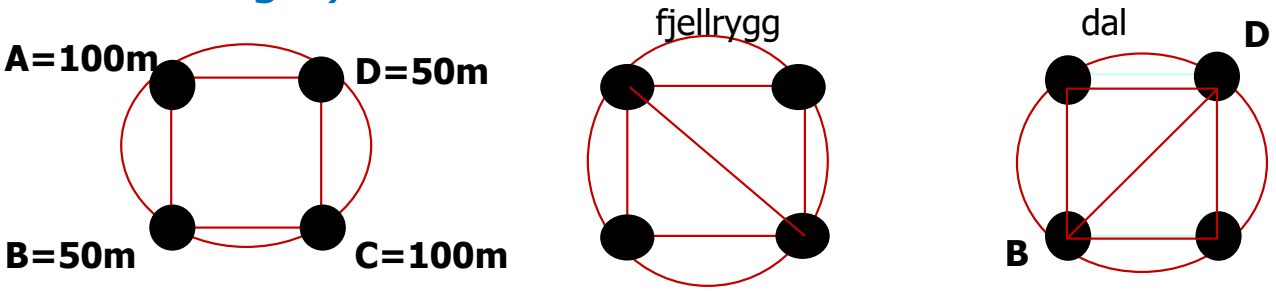


Noen egenskaper ved Delaunay triangulering (DT)

- Hvis ikke alle punktene ligger på en linje, er en DT entydig med følgende spesialtilfelle:
  - Hvis ***4 eller flere*** punkter ligger på sirkelen, må vi finne en regel om hvilke av trekanter vi skal velge (kosirkularitet) – her eks. 4::



  - Anta at hjørnene i er målinger av høyder i terrenget. Går det en dal fra B til D eller en fjellrygg A til C ? (ikke avgjørbart uten å se på terrenget).



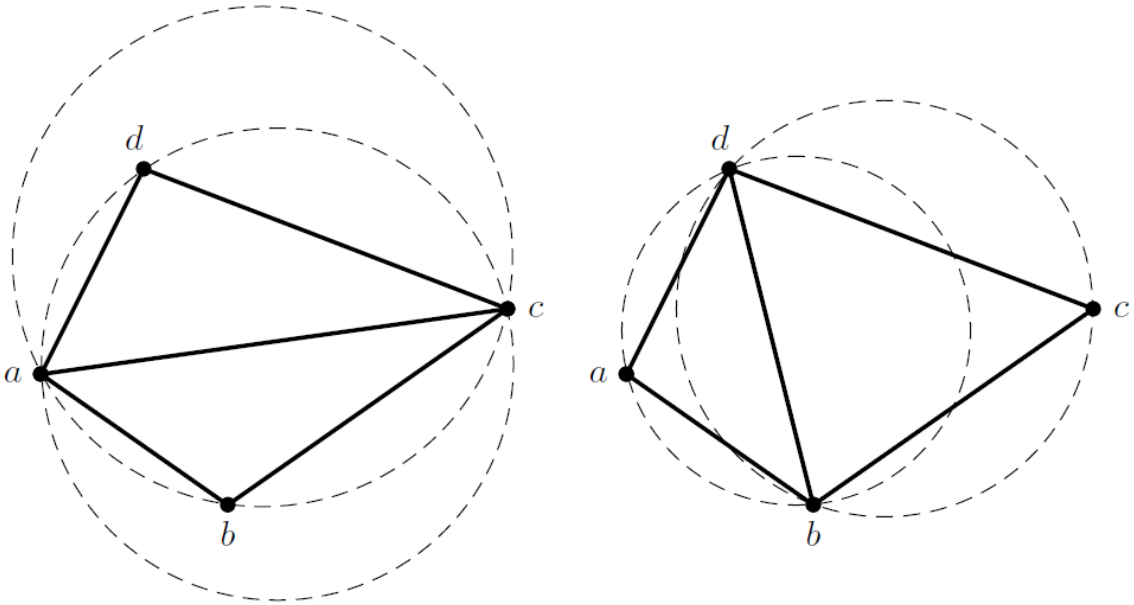
  - Vi velger bare én konsekvent – f.eks siden fra (minst y, minst x) til (størst y, størst x) – dvs. linjen B-D. Dette kan generaliseres.

Noen flere egenskaper ved Delaunay triangulering

- I snitt har et indre punkt P (et punkt som ikke er på den konvekse innhyllninga) seks naboer, men antall naboer kan variere mellom 2 og n-1 (men vanligvis 4-12)
  - Et punkt P har f.eks n-1 naboer, hvis n-1 av punktene ligger på en sirkel og P er et indre punkt i sirkelen.
  - Et indre punkt P har tre naboer hvis f.eks vi har 4 punkter, og P ligger inne i en trekant av de tre andre punktene.
  - Et punkt i hjørnet på innhyllinga kan ha 2 naboer.
- Alle punktene må følgelig ha en fleksibel liste av punkter som er dets naboer (dvs. de punktene den danner trekanter med)
- For at vi kan vite hvilke trekanter et punkt deltar i, må denne nabolista sorteres (mot klokka) – eller genereres i den rekkefølgen.
  - Grunnen til dette er at noen algoritmer finner naboene i en 'tilfeldig' rekkefølge
- Har vi k punkter $n_1,n_2,..n_k$ i en sortert naboliste for punktet P vil n_i-n_{i+1} være en trekant og n_k-n_1 også være en av de k trekantene P deltar i.
- Enhver indre trekantside deltar i to trekanter.

Delaunay algoritmer; mange & få gode

- De aller første for å lage en DT (Delaunay Trekant) ABC:
  - Velg et punkt A, prøv alle mulige (n-1) av B_i, og igjen for hver av B_i –ene: alle mulige (n-2) valg av C_j . Test så om A B_i C_j tilfredstiller sirkel-kriteriet.
  - Å finne én DT tar da $O(n^2)$ tid og finne alle DT tar **$O(n^3)$** tid !
- I kurset INF 4130 undervises en flippingsalgoritme som i verste tilfellet er $O(n^2)$.



Delaunay – algoritmer her (i prinsippet O(n)):

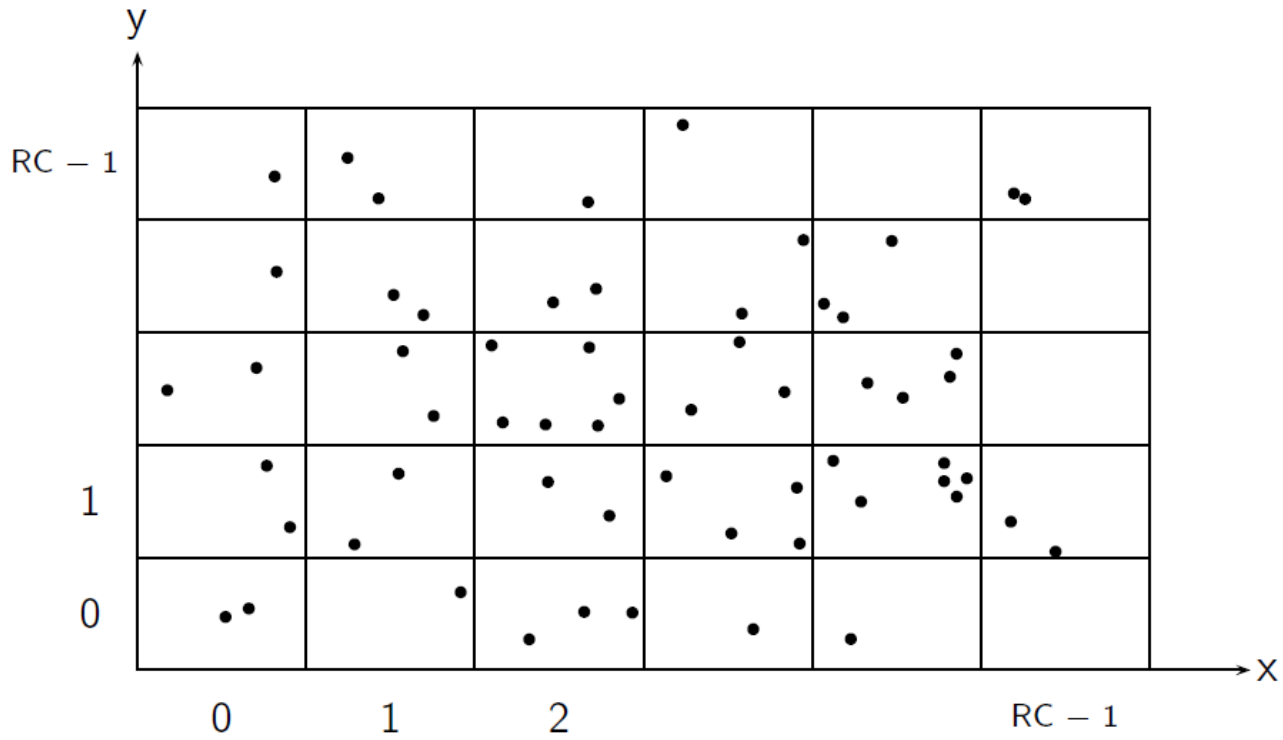
- Her skal vi konsentrere oss om to algoritmer som er O(n):
  - a) Konveks innhylling + sirkelutvidelser fra kjent linje AB i en DT
  - b) Konveks innhylling + Nærmeste nabo(er) + sirkelutvidelser fra kjent linje AB i en DT for resten
- Konveks innhylling, a) er Oblig5
- Rask kode som løser a) og b) er ca. 2000 LOC (Lines Of Code). Vil her bare skissere algoritmen for b).

Stegene i en Delaunay triangulering

1. Sorter punktene i et rutenett (bokser).
   - For lettere å finne punkter nær et annet punkt
2. Finn den konvekse innhyllinga
   - Dette avgrenser søk og er kjente trekantsider i en DT
3. Foreta selve trekanttrekkinga:
   1. Finn neste C ut fra kjent trekantside AB
   2. Sortér naboer for et punkt rundt et punkt (retning mot klokka)
4. Spesielt problem: hvordan måle en vinkel nøyaktig og raskt ?
5. Bør vi bruke heltall eller flyttall som x,y i posisjonen til punktene?
   1. Svar: Heltall – ellers blir vinkelberegning umulig på små vinkler + at de fleste oppmålinger (som GPS) er i heltall
   2. (alle punktene bør ha heltalls-koordinater som er partall)

Delaunay – triangulering har følgende delalgoritmer

1) Sorter datapunktene i bokser (ca. 2- 10 punkter i hver boks):

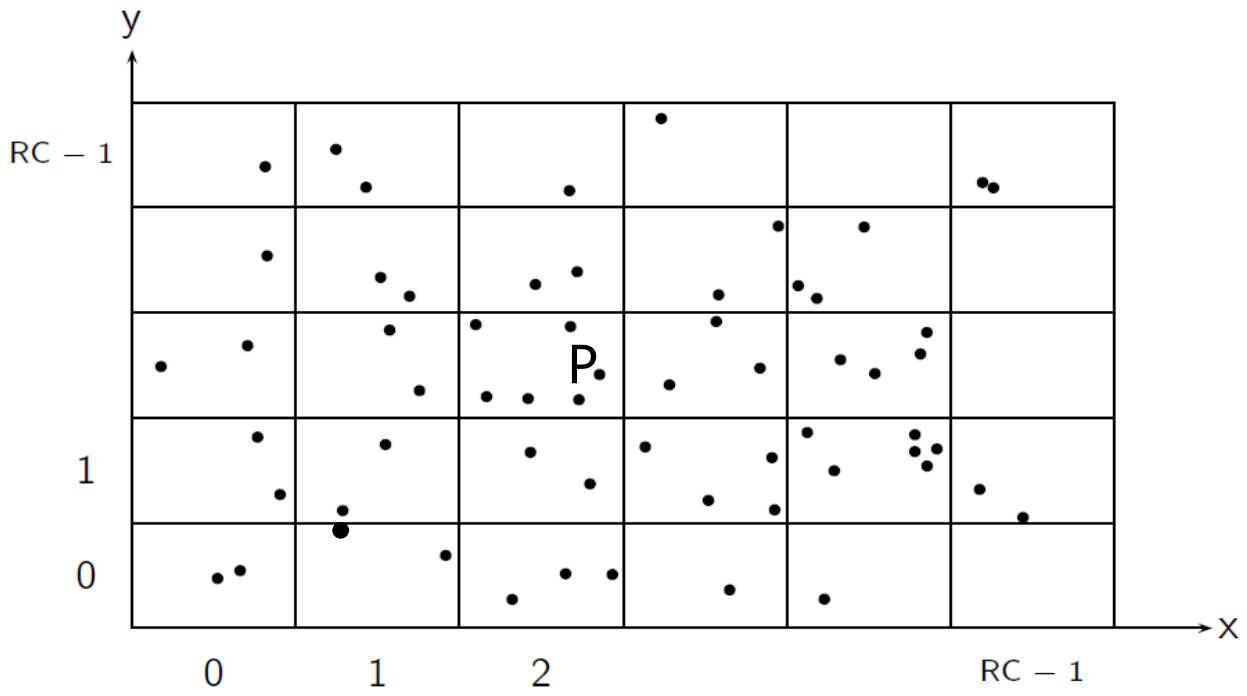


Med 1 000 000 punkter blir dette en 316x316 rutenett med 10 punkter i hver boks. Datastruktur forteller hvilke punkter det er i en boks.

Bruk f.eks Radix-sortering og senere parallell Radix .

Hvordan/hvorfor bruke en boks-oppdeling

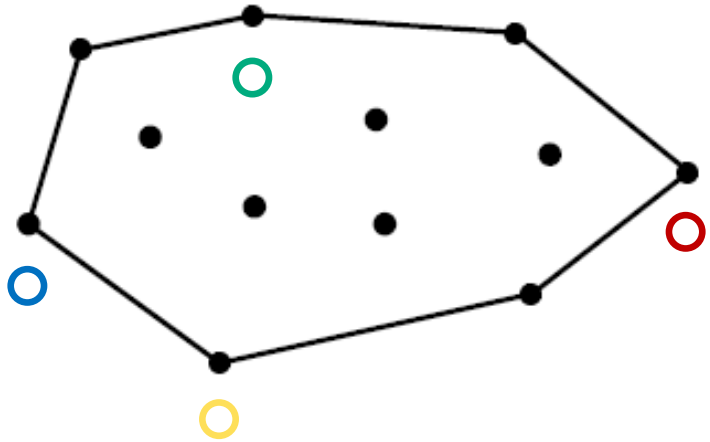
- Si vi skal finne interessante punkter i nærheten av et punkt P:



- Man slår først opp i boksen som P er i, men hvis man der finner punkter som ligger i en avstand hvor 'nærmere' punkter kan ligge i en annen boks, ser man på alle punkter i en 3x3 boksene rundt P (evt. 5x5, osv)

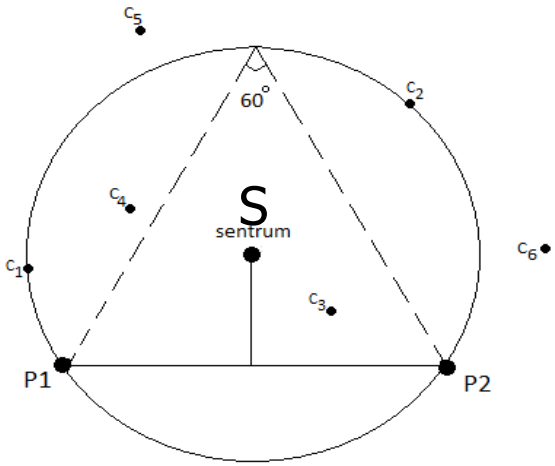
Finn den konvekse innhyllinga – Oblig 5

- Dette avgrenser søk og er kjente trekantsider i en DT



- Vi vet at fire (i spesielle tilfeller :tre eller to) punkter er med på den konvekse innhyllinga:
  - max x, min x, max y , **min y**
- Foreleses senere i dag som Oblig 5 uten bokser (
  - Bokser er raskere enn mengder i IntList, men med bokser ville ha blitt en for stor Oblig 5

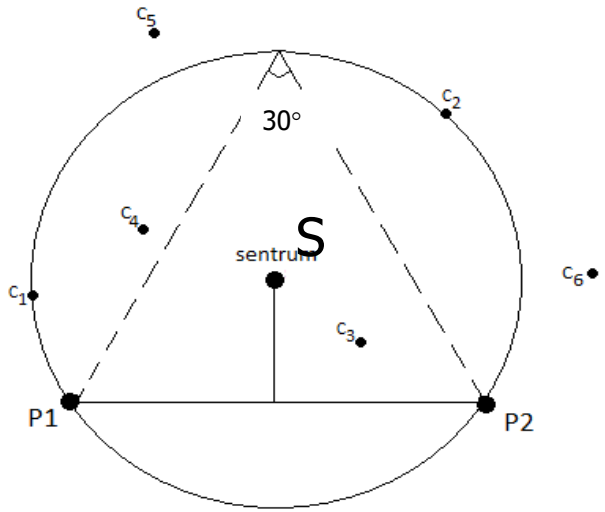
Foreta selve trekanttrekkinga - 1:



- Gitt DK (Kantside i Delaunay trekant) P1-P2.
  Skal finne punkt C slik
  at P1P2C er en
  DT (Delaunay Trekant).
- Lager så midtnormalen på P_1P_2, og finner så sentrum S slik at alle perfeirivinklene på sirkelen med sentrum S har en vinkel på 60°.
- Åpner alle bokser rundt S slik at vi finner alle punkter C som er inne i sirkelen og over P_1P_2.
- Alle punkter inne i en slik sirkel har større ∠ p1Cp2 enn de på utsida
- Finner den C av disse med størst vinkel P_1CP_2.
- Når man finner at ABC er en trekant, noteres det i A at C er ny nabo, men bør man notere ?
  - i C at A og B da er nye naboer; (eller i B at A er ny nabo til B) ?
  - Nei, fordi ……

Foreta selve trekanttrekkinga - 2:

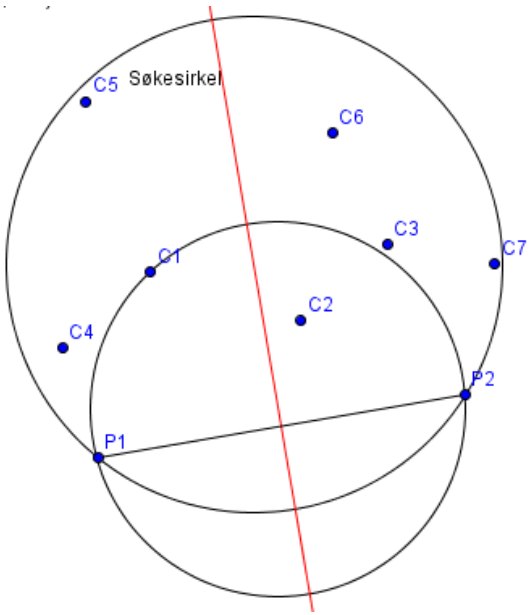
- Var det ingen punkter inne i sirkelen, må sentrum S skyves til vi har laget en f.eks ca. dobbelt så stor sirkel.

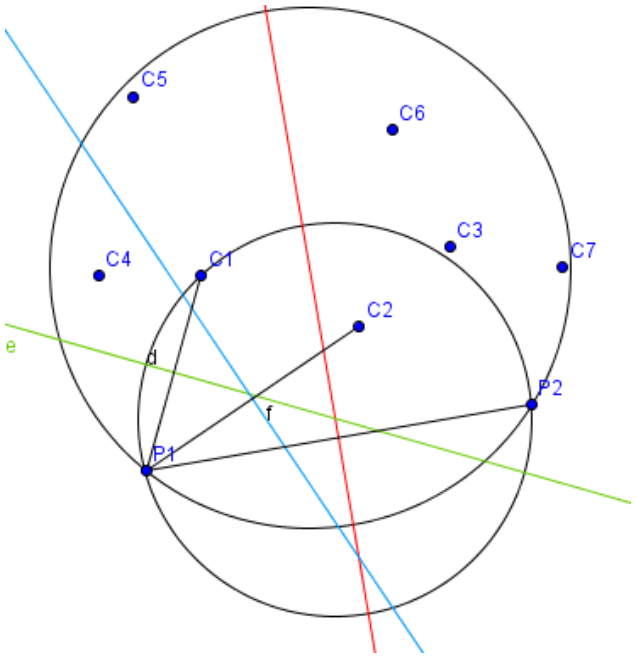


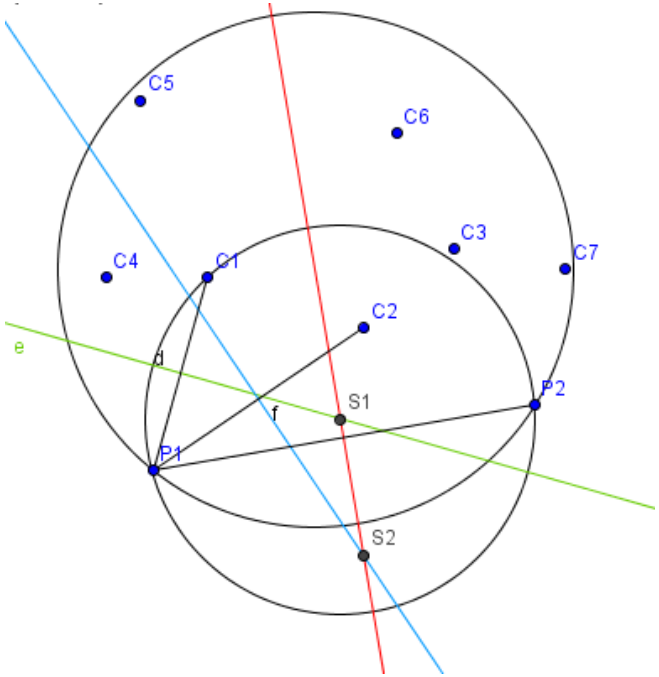
- Åpner alle bokser rundt ny S slik at vi finner alle punkter C som er inne i sirkelen og over P1-P2.
- Finner den C med størst $\angle$ P1CP2.
- Hvis det fortsatt ikke er punkter inne i **ny større sirkel**, gjør vi den enda større,.., åpner nye bokser til vi får minst ett punkt C inne i sirkelen.

Hvordan: Finne den C med størst $\angle P_1CP_2$.

- Alle punkter inne i en søkesirkel med sentrum på midtnormalen p1-p2 har større $\angle P_1CP_2$ enn de utafor sirkelen.
- Trekker midtnormalen på p1-p2 (rød); og etter tur alle linjer fra p1 til C1, C2,.. inne i søkesirkelen.
- For hver slik linje p1-ci, finn midtnormalen på denne (blå og grønn). De to midtnormalene (rød fra p1-p2) og fra p1-c, skjærer hverandre i sirkelen som akkurat er sentrum i den omskrevne sirkelen for disse tre punktene:
- Velg punktet Ci som har sirkelsenter Si lengst negativt fra linja p1-p2



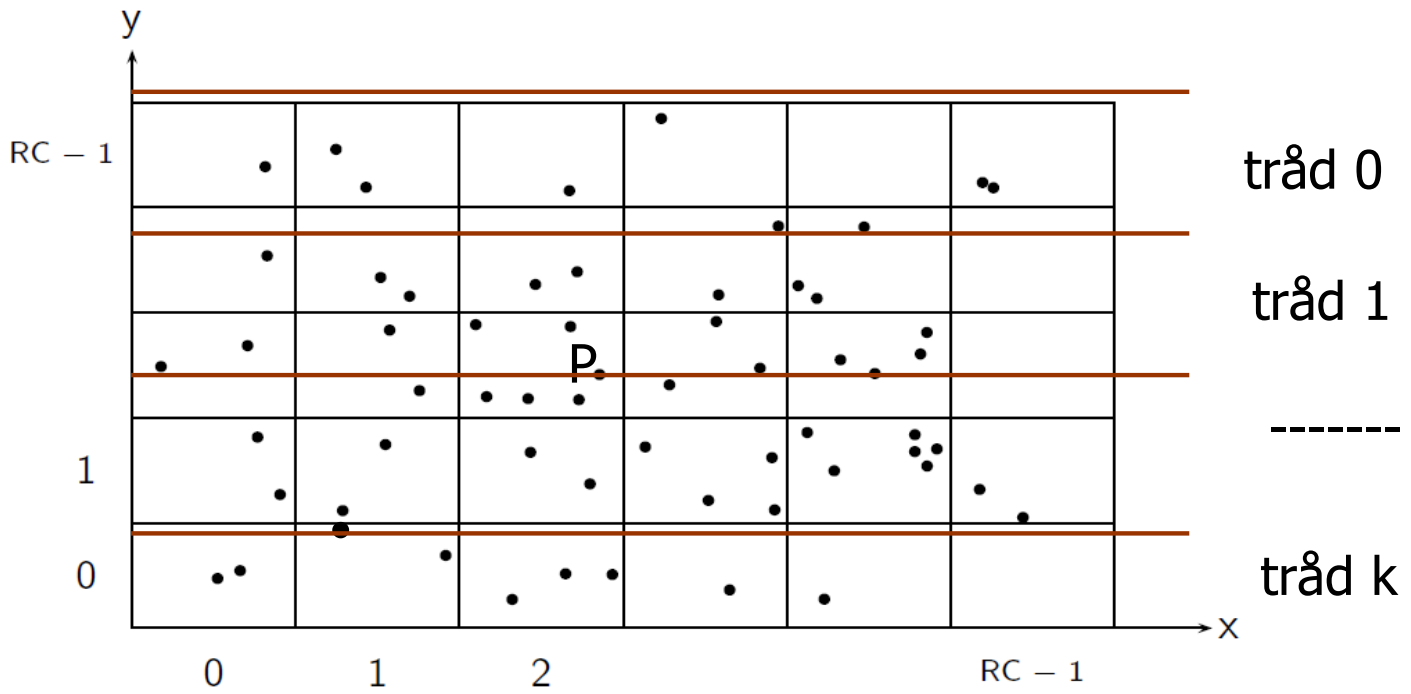




- I figurene over ser vi at C2 har størst $\angle P_1CP_2$. fordi S2 har større negativ avstand fra p1-p2 enn S1.

Parallellisere trekanttrekkinga - 1

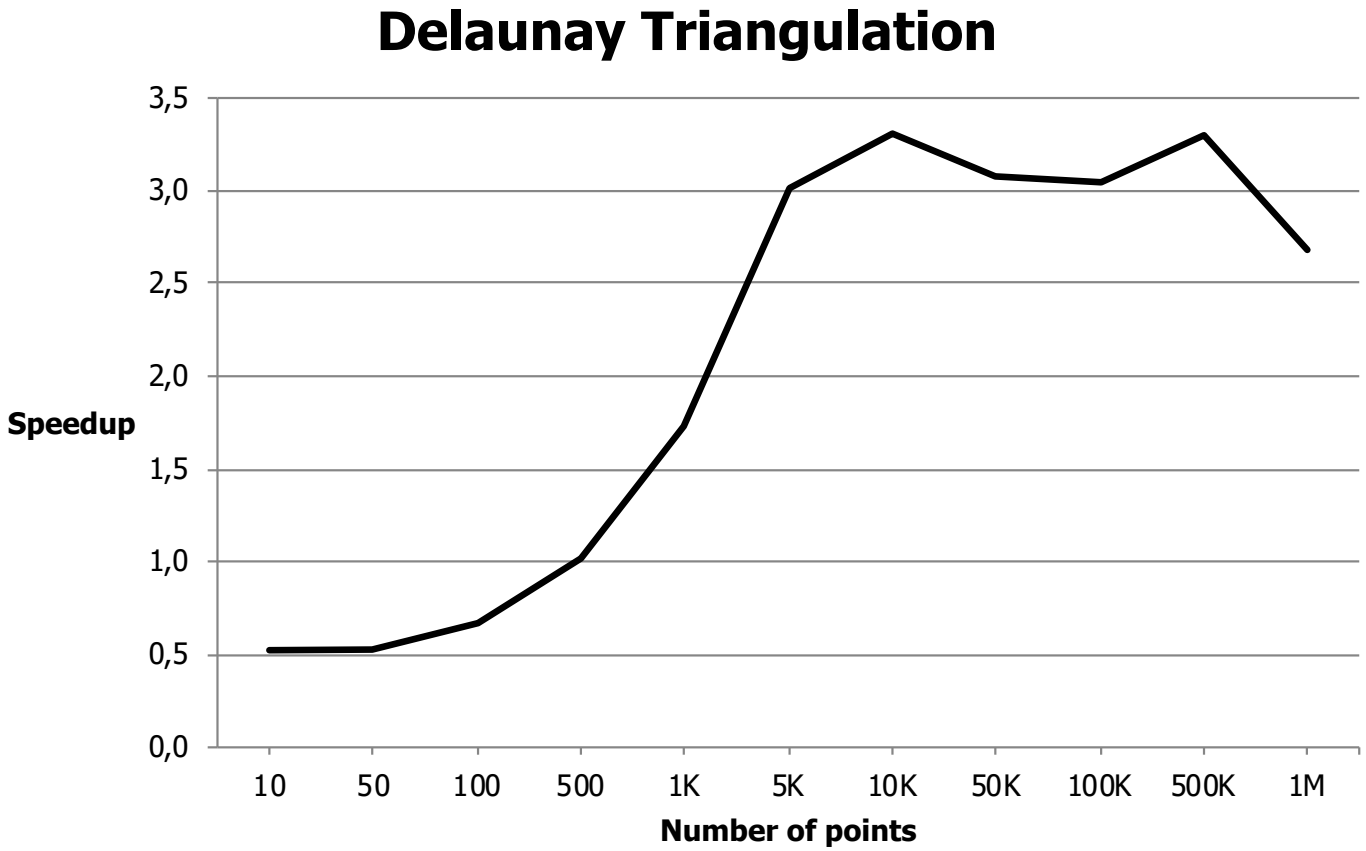
- Hvilke delte data har vi – dvs. data som to tråder muligens kan skrive samtidig i?
  - Svar: Koordinatene for de n punktene, lista over den konveks innhyllinga og særlig nabolista for hvert punkt.
- Deler punktmengden radvis per tråd (se for deg langt flere bokser):



Parallellisering av sirkel-metoden – 2.

- Opplagt parallellisering er å dele punktene (både de på innhyllinga og de indre punktene) likt mellom trådene.
- Problemene oppstår når en tråd_i 'eier' A, men enten ikke eier både B eller C (og vi har funnet ny DT: ABC):
  - Løsning: A noterer bare nye naboer hos seg selv fordi en DT er entydig, og den samme trekanten ABC blir også funnet av de trådene som eier B og C, og da notert i de punktene.
  - Dette er også mest effektivt i den sekvensielle løsningen !
- Da bryter vi ikke noen av reglene for parallellisering:
  - Trådene kan all lese samtidig data det ikke skrives på.
  - Hvis en tråd skriver, må ingen av de andre trådene lese det før etter synkronisering.

Speedup med bruk av JavaPRP på DT-problemet - 1



Graf 6: *Viser speedup vi får ved å beregne Delaunay triangulering i parallell, generert av Java PRP. Dataene baserer seg på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz, med 8 kjerner (4 fysiske med hyperthreading).*

Speedup for håndkodet parallell Delaunay-algoritme -2

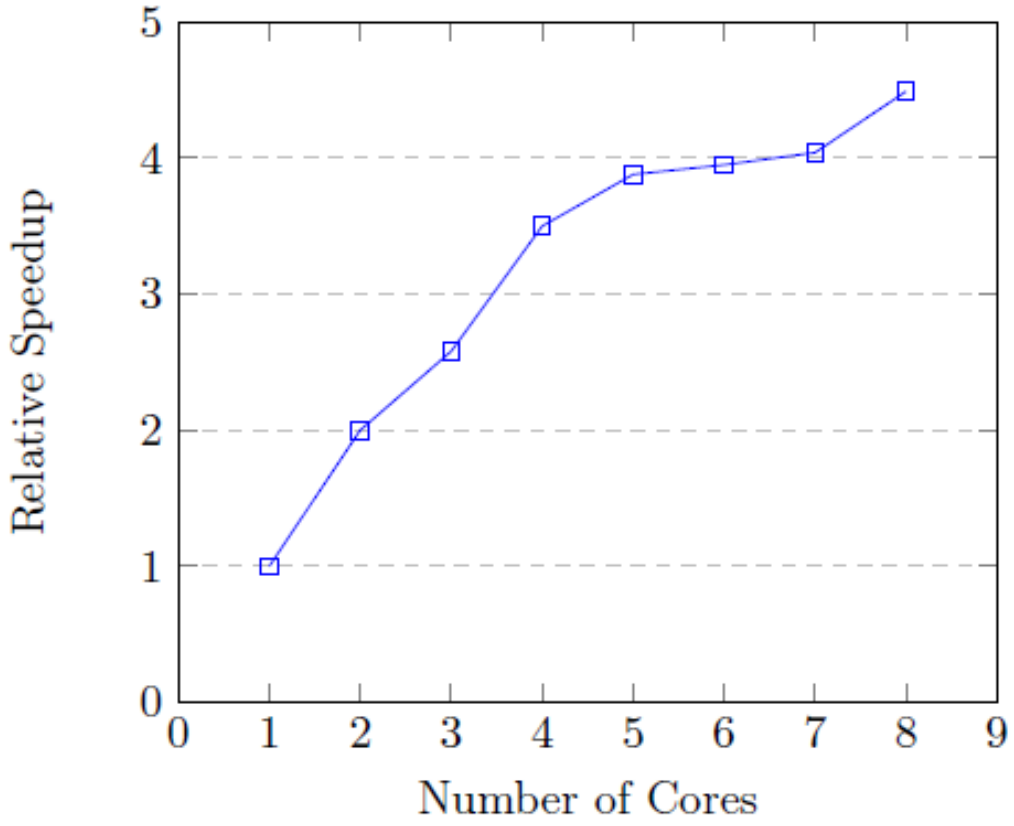
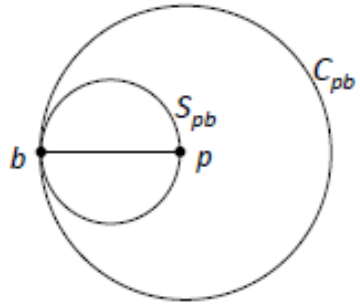


Figure 7.3: Shows the average speedup for different numbers of cores when trian gulating 10 million points.

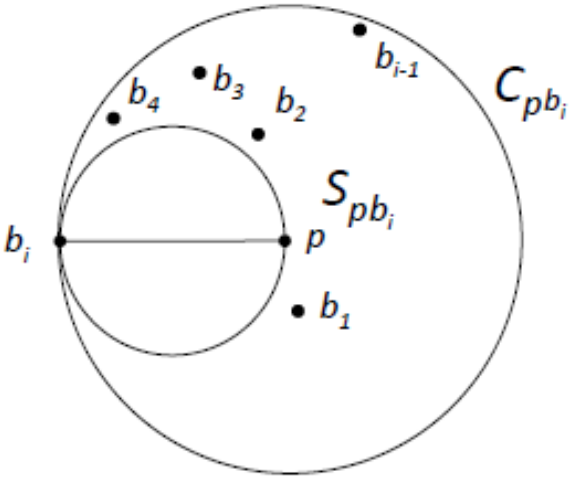
Alternativ metode for å finne trekantnaboer

- Sats: Enhver linje som er diameter i en sirkel uten andre indre punkter i sirkelen, er en DK, en gyldig trekantside i DT.

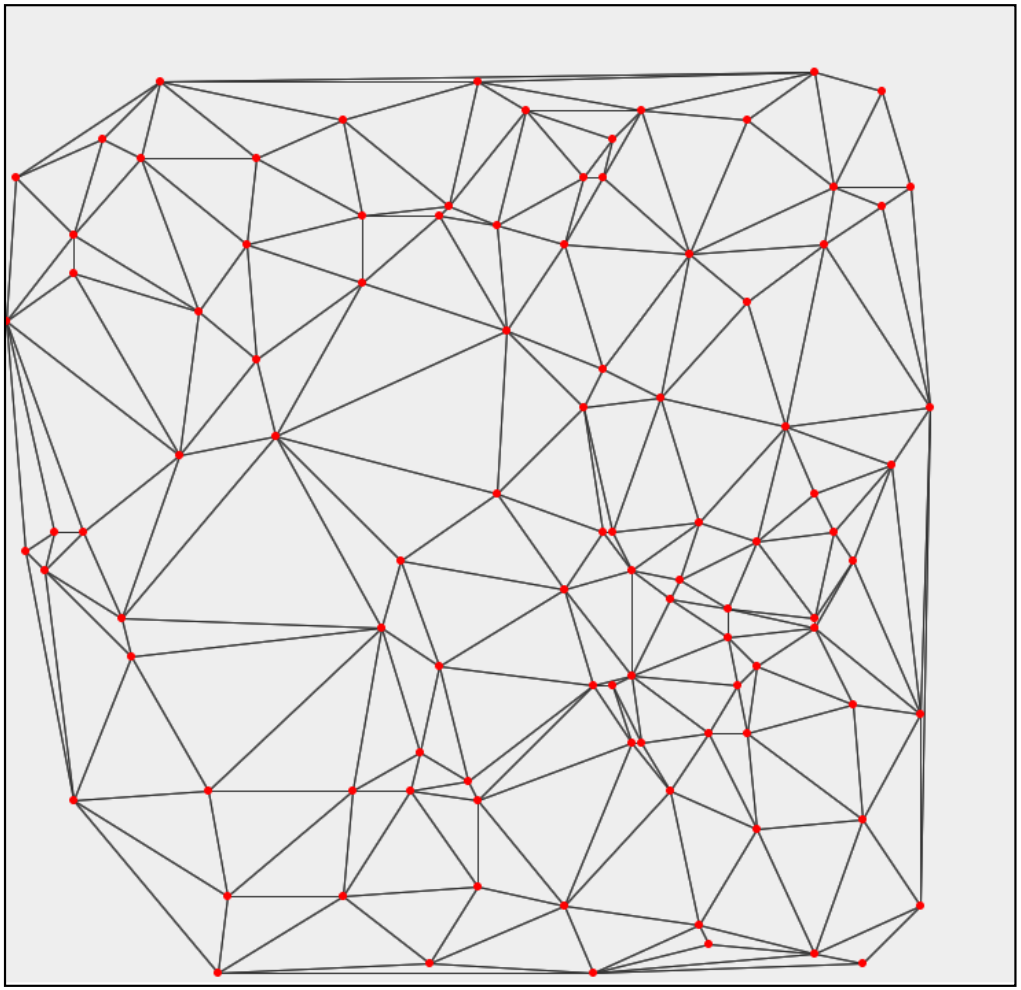


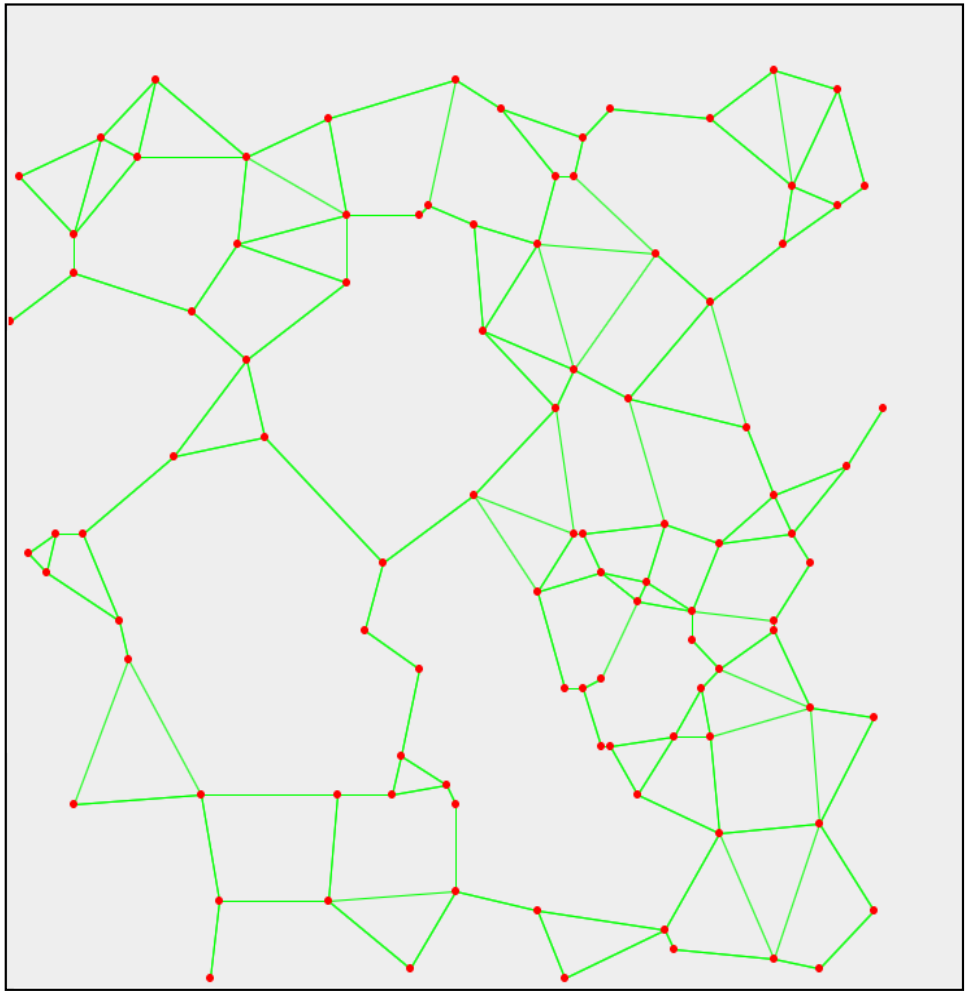
- Linjen fra ethvert punkt p til dens *nærmeste* nabo b er derfor en DK

- Også de m nærmeste naboene til p er DT-trekantsider hvis sirkelen p – 'nær nabo' b_i ikke inneholder noen av de andre punkter som ligger nærmere p – dvs. b_j hvor j <i

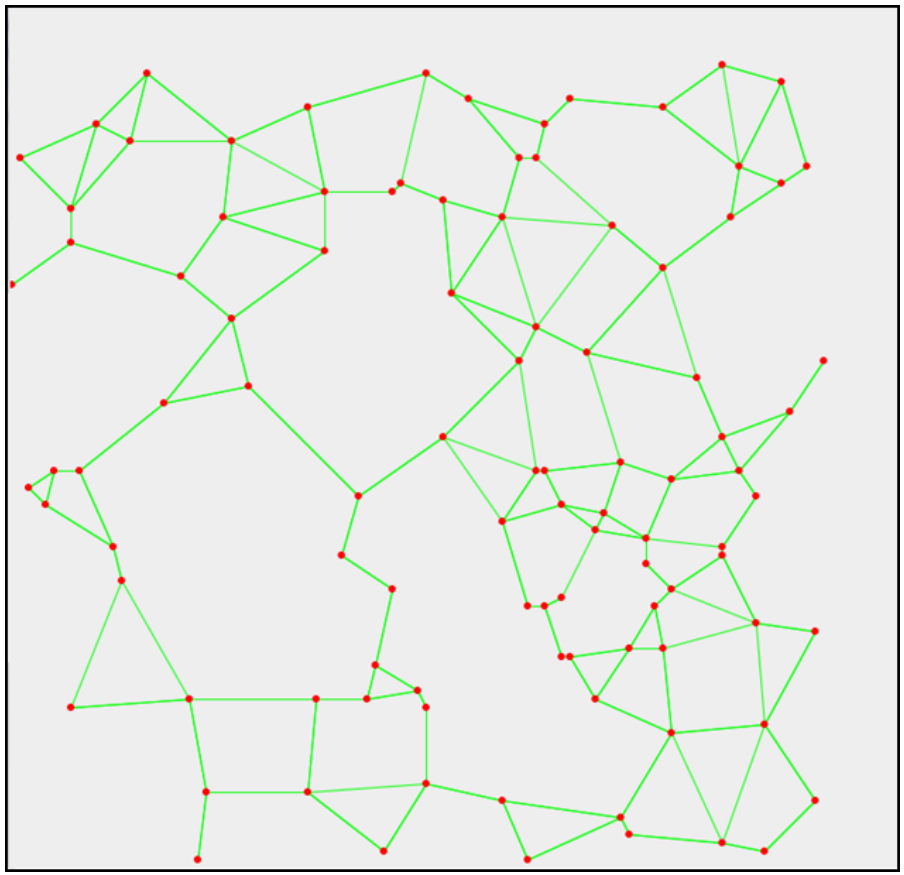


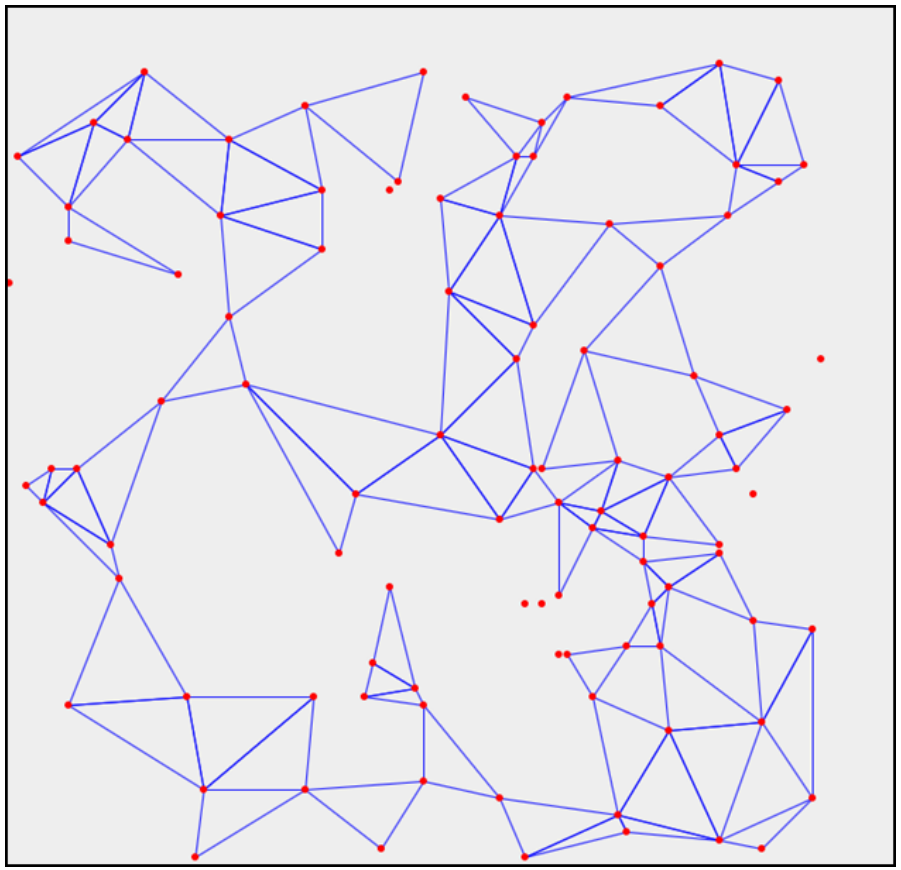
Fullstendig DT og de trekantsidene man finner med 10 nærmeste naboer (man kan finne ca. 70% av alle DK med denne metoden ved å teste de 25 nærmeste naboer)





Trekantsider og trekanter funnet ned 10 nærmeste naboer-metoden (resten fylles ut med Sirkel-metoden)





Noe speedup for nærmeste nabo-metoden, sekvensielt

Antall punkter	Kjøretider(ms) for		Forbedring
	Sekvensiell Delaunay triangulering **uten** DT fra naboer	Sekvensiell Delaunay triangulering **med** DT fra naboer	
10	2	2	1,23
50	6	5	1,10
100	7	8	0,85
500	17	16	1,09
1K	30	27	1,11
5K	129	118	1,09
10K	251	215	1,16
50K	1 376	1 136	1,21
100K	2 640	2 280	1,16
500K	15 417	14 084	1,09
1M	30 578	30 515	1,00

Oppsummering om parallellisering av Delaunay triangulering

- Mange metoder og alternativer
- Hvert 'stort' delproblem kan 'lett' gis en parallell løsning
- En god sekvensiell (og da parallell) løsning krever god innsikt i selve problemet.
- Parallelliseringa krever innsikt i særlig hva er felles data og hvordan IKKE synkronisere for mye på disse, men i hovedsak la parallelliteten gå i usynkroniserte parallelle faser etterfulgt av litt sekvensiell kode.

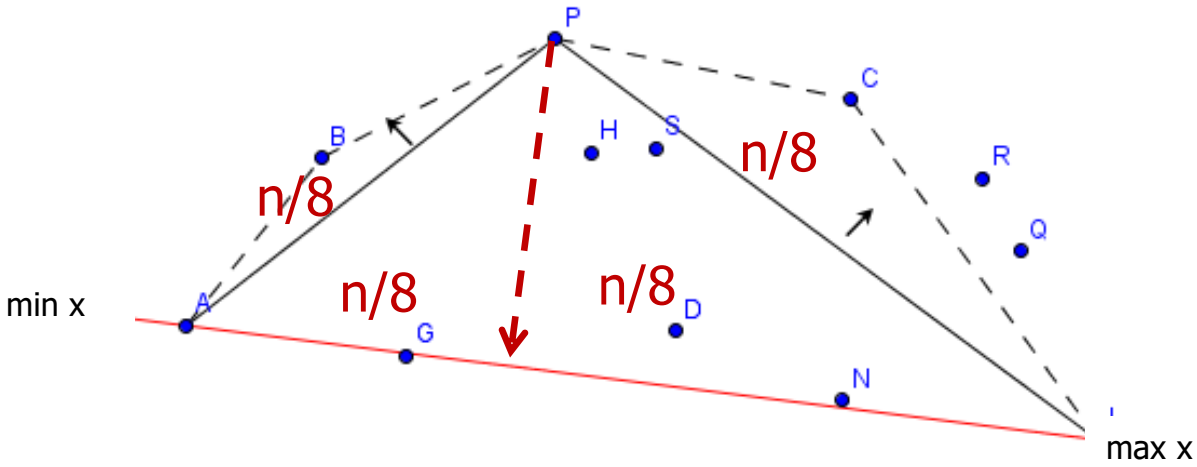
Oblig 5 – den konvekse innhyllinga (KoHyll) tips:

- Litt repetisjon av den sekvensielle algoritmen
  - Problemet er at vi må finn flere parametere før vi kan starte den sekvensielle, rekursive løsningen.
- Forventet kjøretid og 'verste-tilfelle' kjøretid for n punkter
- Hvilken speedup kan vi forvente
  - 2015
  - 2016
  - 2017 ?
- Hvordan løser vi sekvensielt og parallelliserer vi Oblig 5

Algoritmen for å finne den konvekse innhyllinga sekvensielt

1. Trekk linja mellom de to punktene vi vet er på innhyllinga fra maxx -minx (I - A).
2. Finn punktet med størst negativ (kan være 0) avstand fra linja (i fig 4 er det P). Flere punkter samme avstand, velg vi bare ett av dem.
3. Trekk linjene fra p1 og p2 til dette nye punktet p3 på innhyllinga (neste lysark: I-P og P-A).
4. Fortsett rekursivt fra de to nye linjene og for hver av disse finn nytt punkt på innhyllinga i størst negativ avstand (≤ 0).
5. Gjenta pkt. 3 og 4 til det ikke er flere punkter på utsida av disse linjene.
6. Gjenta steg 2-5 for linja minx-maxx (A-I) og finn alle punkter på innhyllinga under denne.

Forventet kjøretid med n tilfeldige punkter = O(n)



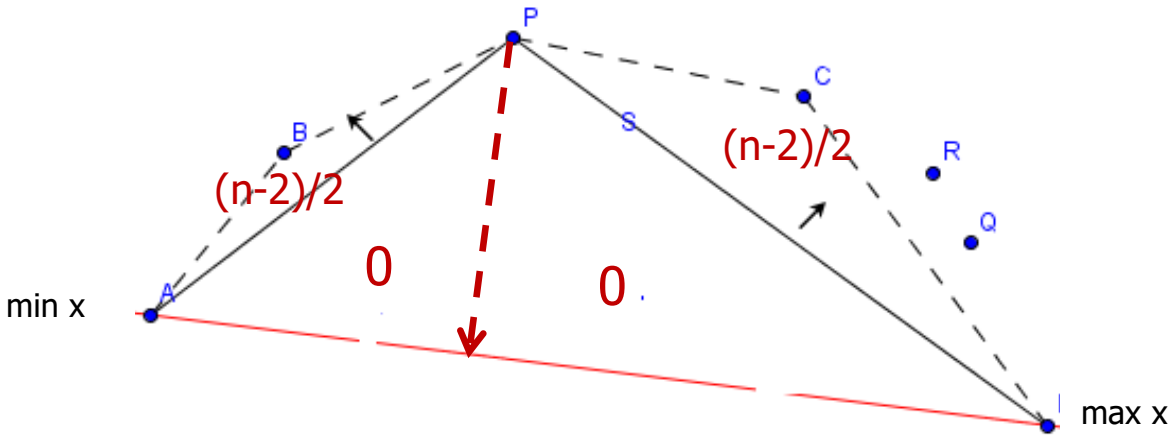
Ny mengde:

1. Finne max og min x – lete-mengde : **n**
2. Finne mengden over/under min-max og P : **n** n/2
3. Finne 2 punkter: C og B : n/2 *2 = **n**, **n/8**
4. Finne 4 punkter max-C,C-P,P-B og B-min :n/8 *4: **n/2** n/32
5. Finne 8 nye punkter : n/32 *8 : **n/4** n/128
6. Finne 16 nye punkter n/ 128 *16: **n/8** n /512
7. + ...

Sum start = **n** + Alle 'over' max-min= 2 n + n/2 + n/4 +n/8+.. = **3n**

+ sum alle 'under' min-max = **3 n**, Totalsum = n + 2*3n= **7 n = O(n)**

Verste tilfellet kjøretid – alle punktene er på innhyllinga, f.eks alle punktene på et kvadrat eller en sirkel



Ny mengde:

1. Finne max x og min x – lete-mengde : **n**
2. Finne mengden over/under min-max og P : **n** (n-2)
3. Finne 2 punkter: C og B : (n-2)/2 *2 = **n-2,** **(n-4)**
4. Finne 4 punkter :(n-4)/4 *4: **n-4** (n-8)
5. Finne 8 nye punkter (n-2^3): (n-2^4)
6. Dette fortsetter til n - $2^k \leq 1$, dvs. k= $\log_2 n$ ganger

Totalsum =

$\sum_{i=1}^{\log n}(n - 2^i) = \sum_{i=1}^{\log n} n - \sum_{i=1}^{\log n} 2^i$ = nlog n – n ≤ n $\log_2 n$ = **O(n $\log_2 n$)**

Generelt om parallellisering

A. Starten tar først **n** steg for å finne minx &maxxx
B. Så trenger vi nye **n** steg for å finne mengdene til høyre og til venstre for linja minx-maxx
C. Siden hele algoritmen tar **7n** steg, må A og B parallelliseres, ellers får vi ikke god speedup (Amdahls lov)

Alle algoritmer startet i alle fall med A.

Roadmap – to mulig gode parallelliseringer

- Metode 1: tilsavarende fullParalellQuicksort:
  - Del punktene i k = antKjerne*overbook (=2,4,8,..)
  - Kjør sekvensiell algoritme for DKI på hver mengde
  - Du har da k stk. 'små' DKI-er:
    - Hva er minx-maxx for alle ??
    - Hvordan skjøte dem sammen??
    - Har du da den DKI for alle punktene du leter etter ??

  Foreleses ikke

- Metode 2: Parallelliser rekursivt den sekvensielle koden.
  - Foreleses idag.

III) Parallellisering av oblig 5

Stegene i oblig 5:

- Vi jobber hele tiden med *linjer* & *mengder* (IntList) av de tallene som er kandidater til å bli valgt til nytt punkt på KoHyll og da som et endepunkt på en ny linje.
- Starten er litt vanskelig:

1) Finn minx og maxx – trivielt å parallellisere (som oblig1)
   - Alle tallene i x[] og y[] –arrayen- Vi har da første linje.
   - Parallellisere : Trivielt jfr. FinnMax/Oblig 1
2) Trekk linjen minx-maxx, finn nå:
   1) To mengder : Alle til venstre for (eller på) minx-maxx og alle til høyre for(eller på) minx-maxx ,
   2) Samtidig som 2.1 – finner punktet P4 = mest negativ avstand fra minx-maxx, og P3 = mest pos avstand .

   Hvordan parallellisere 2)?

   En IntList eller flere ? (en IntList for hver tråd ?)

Avstanden fra et punkt til en linje II

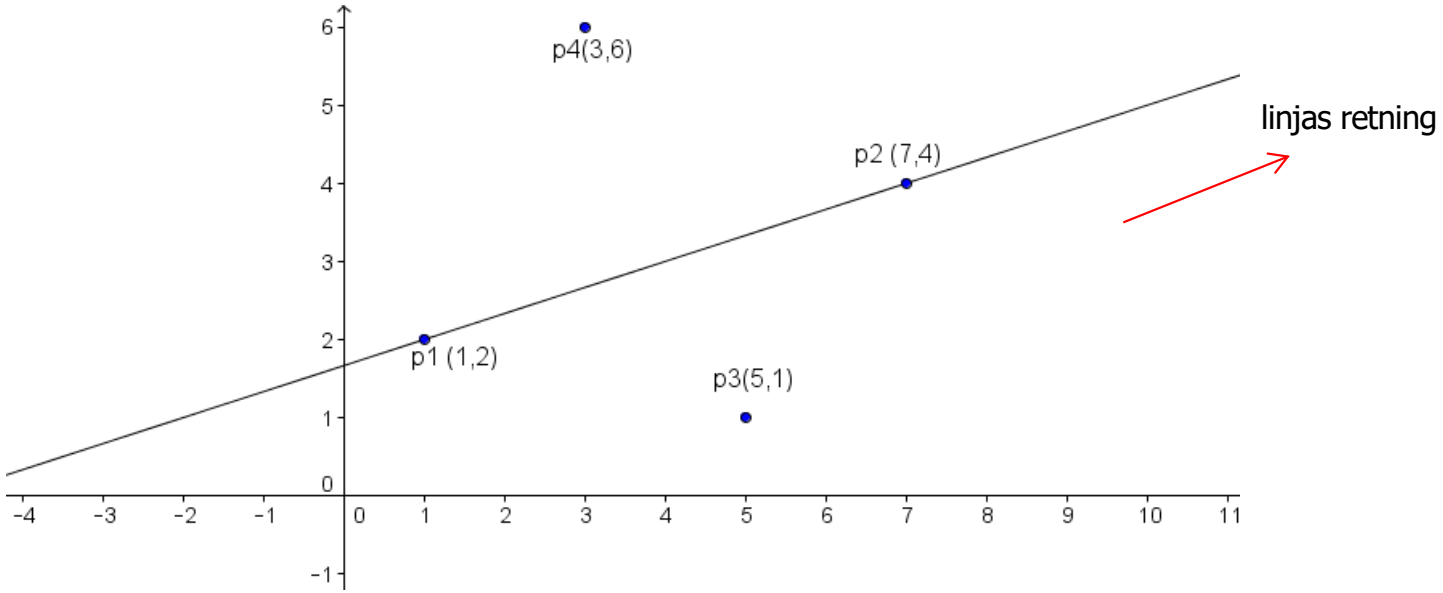
- Avstanden fra et punkt til en linje (vinkelrett ned på linja) er :

$$d = \frac{ax + by + c}{\sqrt{a^2 + b^2}}$$

- Jo lenger fra linja punktene et desto større negative og positive tall blir det.

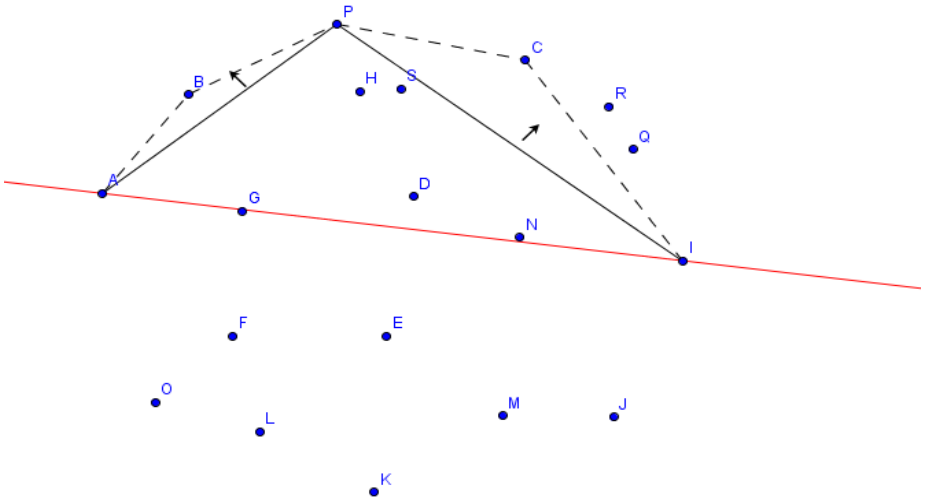
Setter inn **p3** (5,1) i linja p1-p2: -2x+6y-10 = 0, får vi

d= : $\frac{-2*5+6*1-10}{\sqrt{40}} = \frac{-14}{6{,}32} = -2{,}21..$



Hvordan parallellisere denne rekursive finningen av stadig nye punkter på KoHyll.

- Lett, men vi vet at vi *ikke* bare kan erstatte alle rekursive kall med new Thread.
- Bare nye tråder ned til et visst nivå, dvs. må ha med en ny parameter som telles ned med 1 for hvert kall, og nye tråder bare hvis denne er >0.
- Spørsmålet som gjenstår:
  - Når legger vi punktene vi finner inn i KoHyll ?
  - IKKE i den rekkefølgen vi finner dem (p1,p2,p3) fordi det er feil.
  - Vi kan riktignok sortere punktene etterpå, men…



Hvordan legge inn de punktene vi finner på KoHyll.

Sekvensielt:

- Først legges P_1 inn, så alle punktene vi finner rekursivt mellom P_1 og P_3 , men *ikke* P_3 (hvorfor ikke?).
- Tilsvarende så alle punktene på P_3-P_2 ; først P_3 , (men *ikke* P_2 tilsist) inn i KoHyll (hvorfor ikke; samme spørsmål som over) .

Parallelt:

- Lar seg også gjøre, men da bruker vi standard-trikset ved parallellisering: Vi kopierer noen sentrale data til hver tråd.
- Hva kopierer vi her ?
  - Hver tråd får sin egen IntList = sin del av KoHyll
  - Disse lokale KoHyll fra hver tråd må så 'settes sammen' til én felles KoHyll etter hvert. Tilsammen får vi en array av IntList-er
  - Hvordan : også mer tips neste uke.
- Helst ikke lage en Locked metode for å legge inn punktene i en sentral KoHyll fordi der er O($\sqrt{n}$) punkter på KoHyll (mange).

Parametre til sentral rekursiv metode

- Nå et vanskelig valg, hva er parametrene til en rekursiv metode som ut fra en mengde og tre punkter finner KoHyll .

Sekvensiell:

- Inn: En InList m. og int p1,p2,p3 og 'gammel' KoHyll
- Til videre kall: En ny (mindre) mengde m2 og nytt pkt m4
  En ny (mindre) mengde m3 og nytt pkt m5
- Ut : Oppdatert KoHyll

Parallell (tråd og rekursiv metode):

- Inn: En array av IntList, p1,p2,p3, en tråd-lokal IntList minKoHyll, et nivå (for å avslutte generering av tråder)
- Til videre kall: En ny (mindre) mengde m2 og nytt pkt m4, nivå-1
  En ny (mindre) mengde m3 og nytt pkt m5 ,nivå-1
  en IntList over minKoHyll funnet så langt.
- Ut: Oppdatert mynKoHyll ett nivå opp (den tråd/metode som kalte denne) og til sist da felles, global KoHyll.

Hva så vi på i Uke14

I) Om program, samtidige kall og synlighet av data

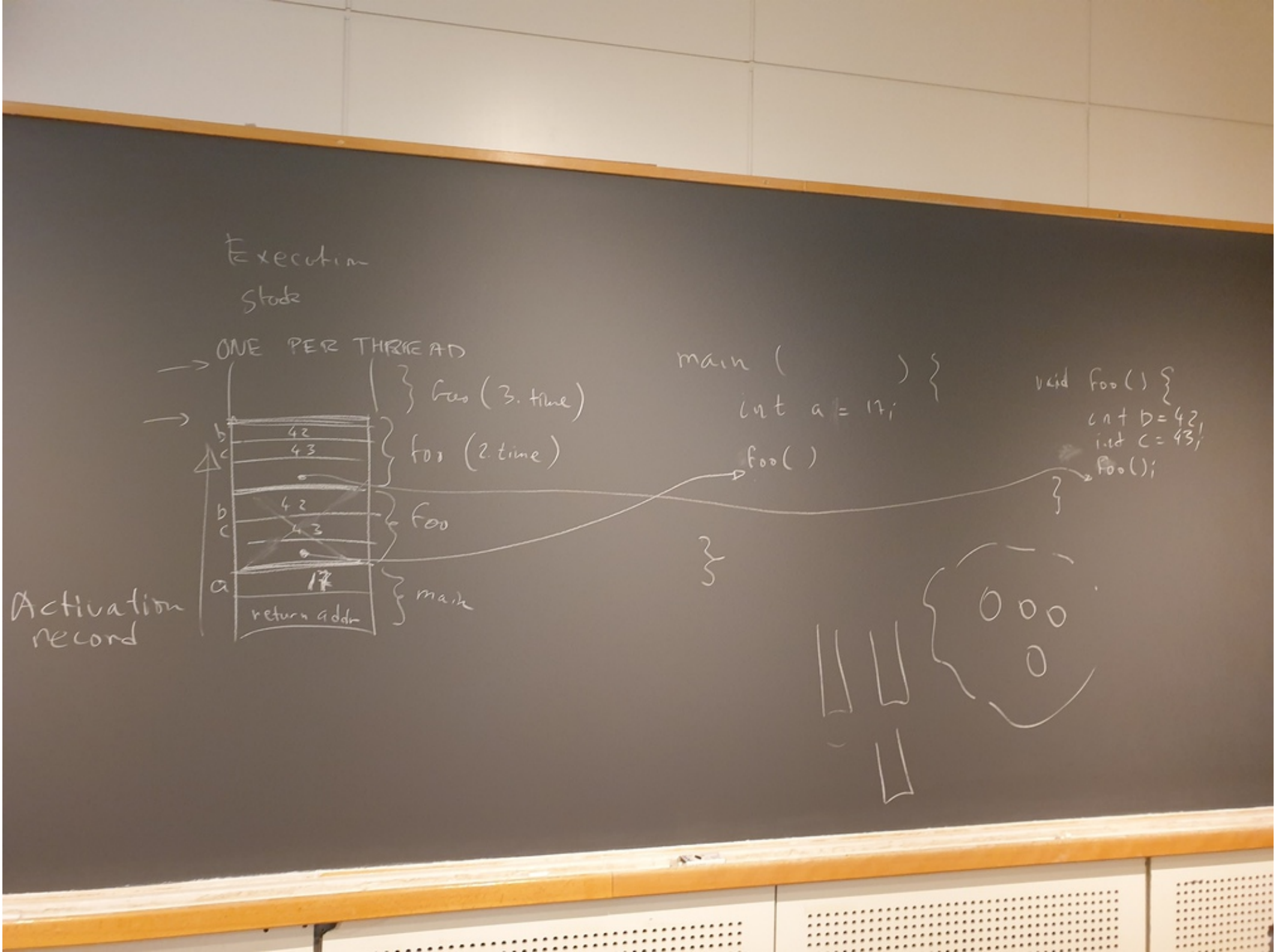
- Når en tråd (main eller en av de andre) aksesserer data, hvilke er det.
- Kan de parallelle trådene og main kalle samme metode samtidig. Kan de kalle metoder fra en sekvensiell løsning? Hva skjer da ?
- Hvor mange stack-er (stabler) har vi, og hvilke ?

II) Mer om et stort program, og hvordan den siste fasen i den kan parallelliseres.

- Delaunay triangulering – de beste trekantene!
  - Brukes ved kartlegging, oljeleting, bølgekraftverk,..
  - Spill-grafikk: ved å gi tekstur, farge og glatte overflater på gjenstander, personer, våpen osv.
  - Er egentlig to algoritmer etter hverandre (KoHyll + trekanttrekking)
  - Sekvensiell og parallell løsning av trekanttrekkinga.

III) Parallellisering av oblig 5 - den konvekse innhyllinga

- To strategier for parallellisering - roadmap
- Hva kan ventes av speedup på oblig 5
- De ulike stegene i oblig 5 og hvordan de alle kan parallelliseres rekursivt


Execution Stack
ONE PER THREAD
Foo (3. time)
foo (2. time)
Foo
main
b
c
42
43
a
17
return addr
Activation record
main ( )
int a = 17;
foo( )
void Foo ( )
int b = 42;
int c = 43;
Foo();

Agenda

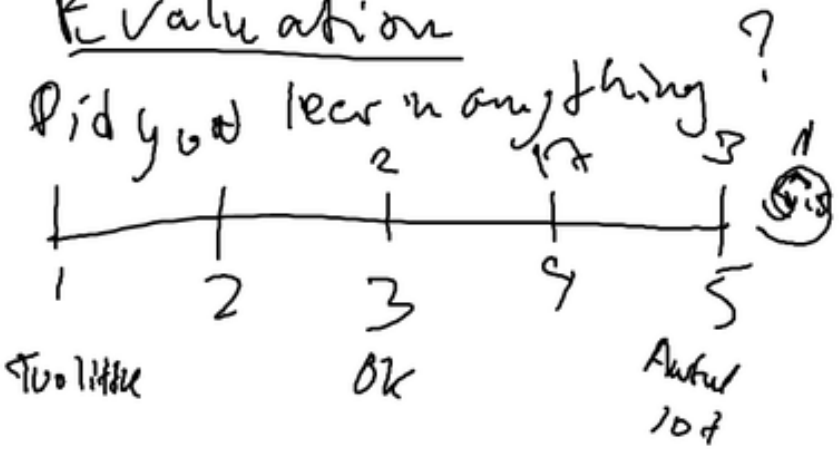
- Review
- Threads & Synchronization
- Review of Exam Spring 2018
- Evaluation

June 7th 9am

- Threads in Java
- Synchronizers
  - Cyclic Barriers
  - Semaphores
- Parallelization - Granularity
  - Divide & Conquer
  - Recursion / Threads
- Caching
  - Moores Law
  - Speed of Light.
    - 300 000 km/s
- DiT

Evaluation

Did you learn anything?



Was it hard?