IN3030 – Effektiv parallellprogrammering
Uke 2, våren 2020

Eric Jul

Professor

PSE

Institutt for Informatikk

Oppsummering – Uke1

- Why did we get multicore computers? – Moore’s law.
- Speed-up central:

$$S = \mathit{Time(Sequential\ algorithm)} \,/\, \mathit{Time(Parallel\ algorithm)}$$

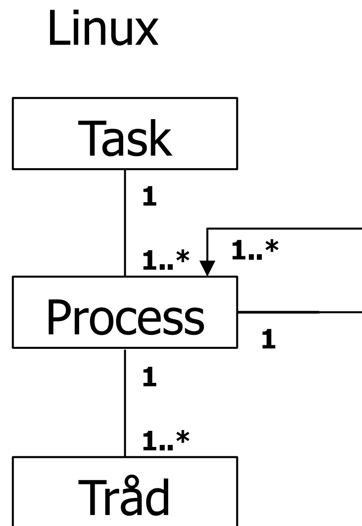
Today

- Threads in Java
- Concurrent Update problem
- Synchronization

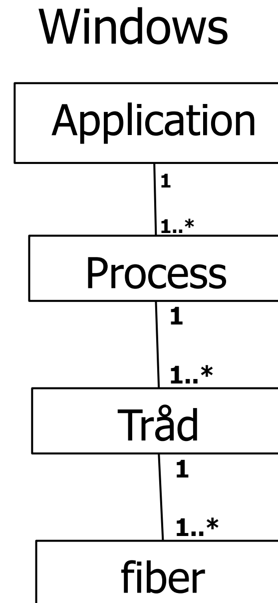
Operativsystemet og tråder

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

Linux

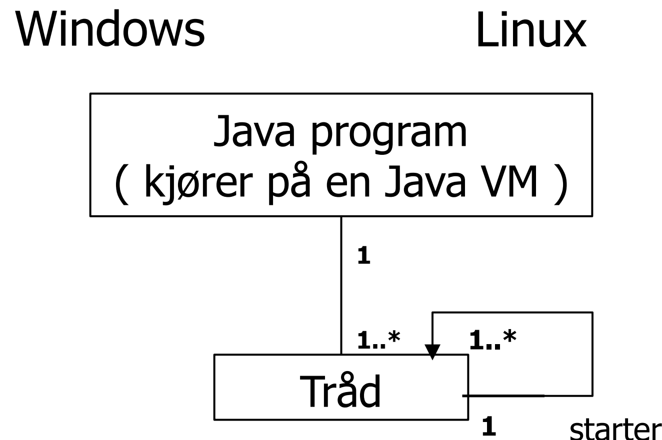


Windows



Heldigvis forenkler Java dette

Java forenkler dette ved å velge to nivåer

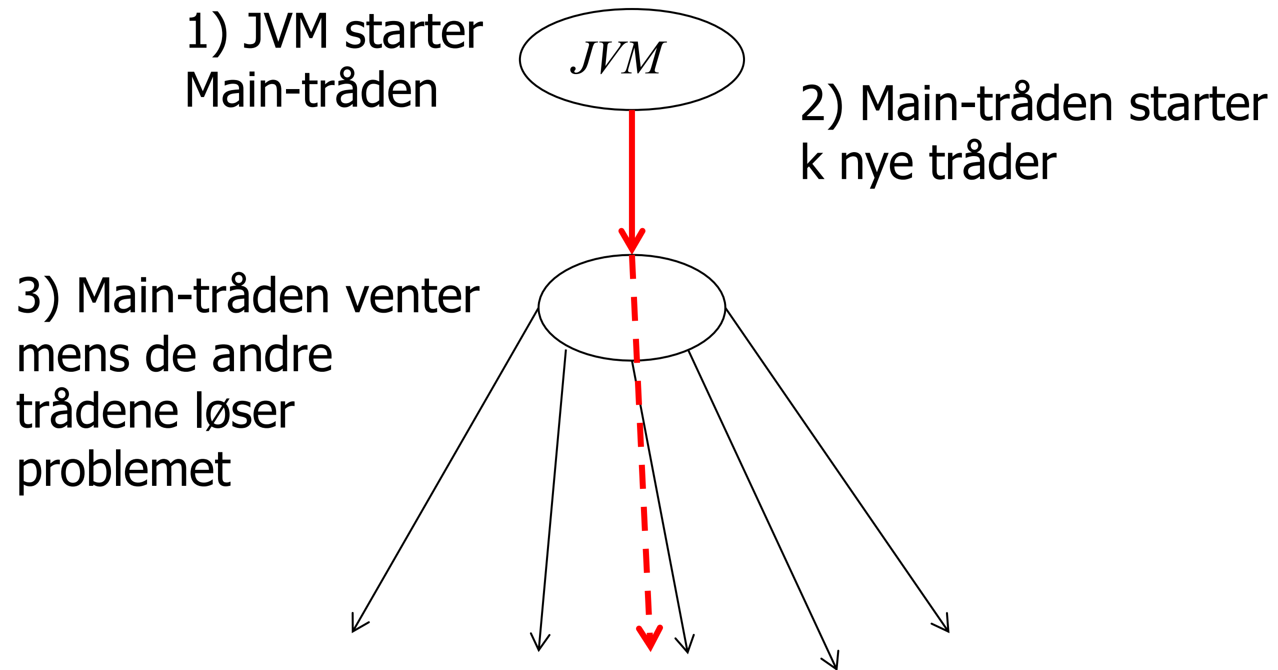


- **Alle trådene i et Java-program deler samme adresserom** (= samme plasser i hovedhukommelsen). Alle trådene kan lese og skrive i de variable (objektene) programmet har og ha adgang til samme kode (metodene i klassene).

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
```

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.

JVM som inneholder sin del av start() som gjør mye og til slutt kaller run()

Vårt program kaller start i vårt objekt av en subklasse av Thread (eller Runnable). Etter start av tråden kalles vår run()

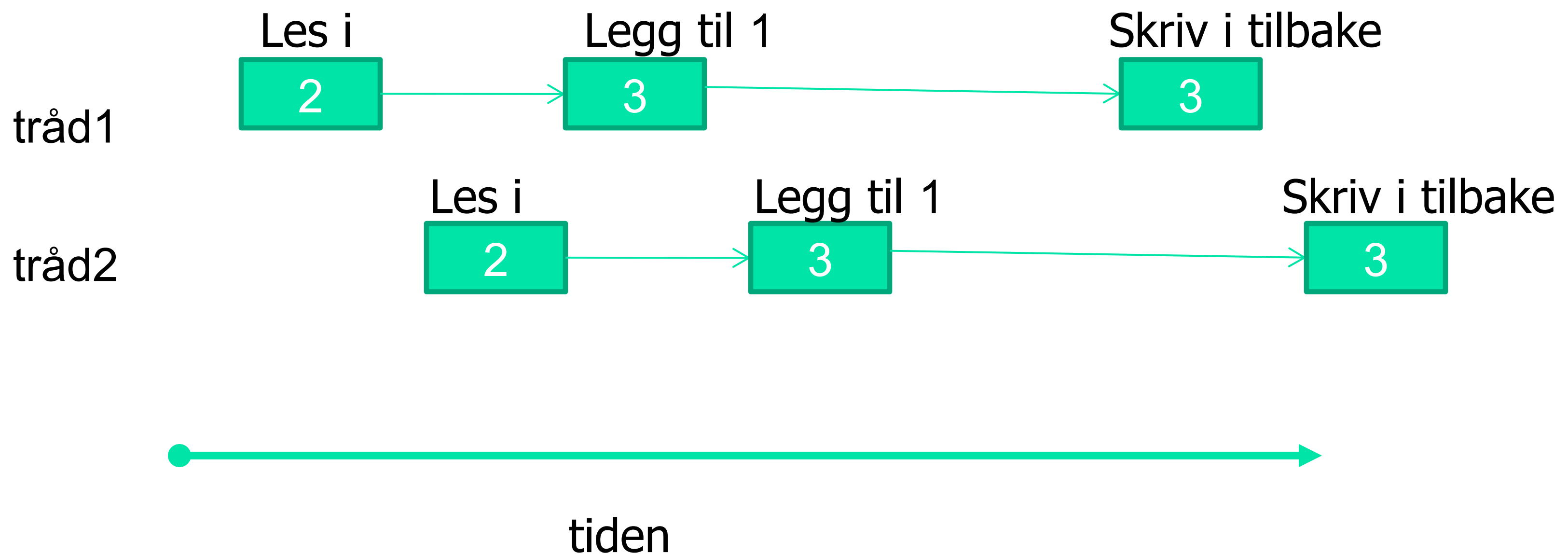
Flere problemer med parallellitet og tråder i Java

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;

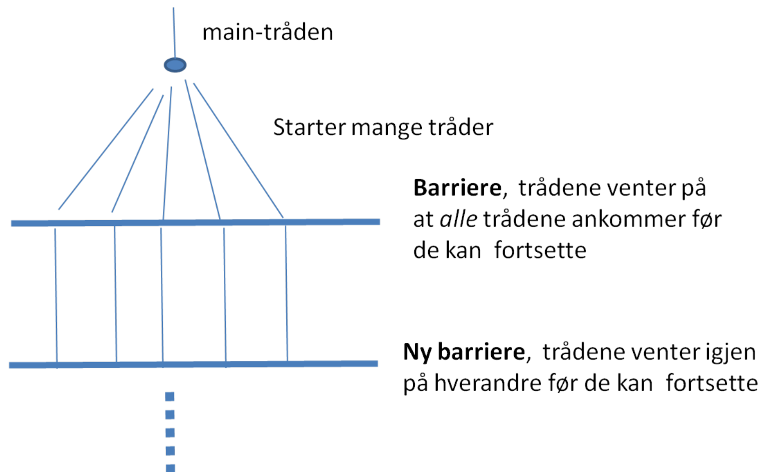
```
for (int j =0; j< 100000; j++) {
        i++;
}
```

- 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 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%

- Prevent concurrent update problem

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.

End of uke2 first lecture

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 I alt 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
```

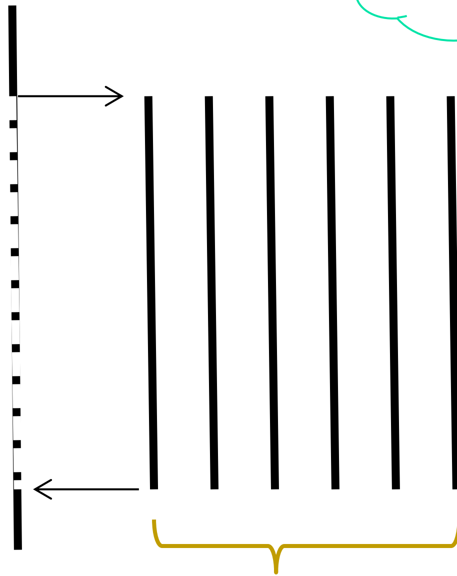
```
class Para implements Runnable{
        int ind;
        Para(int ind) { this.ind =ind;}

        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

main, lager k nye tråder

Data

main
venter



k tråder, leser og skriver i egne og
i felles data og løser problemet

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

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:

javac kompilerer først vårt java-program til en .class fil. som består av **byte-kode**

main( ).
Vårt program kjører først
interpretert (byte-koden tolkes).
Blir JIT-kompilert (mens koden kjører) en eller flere ganger. Går mye raskere

java (JVM) starter vår program i 'main()', men følger med.
1.Kalles en metode flere ganger, kompileres den over fra bytekode til **maskinkode**.
2. Kalles den enda mange ganger kan denne koden igjen **optimaliseres** (flere ganger)

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
```

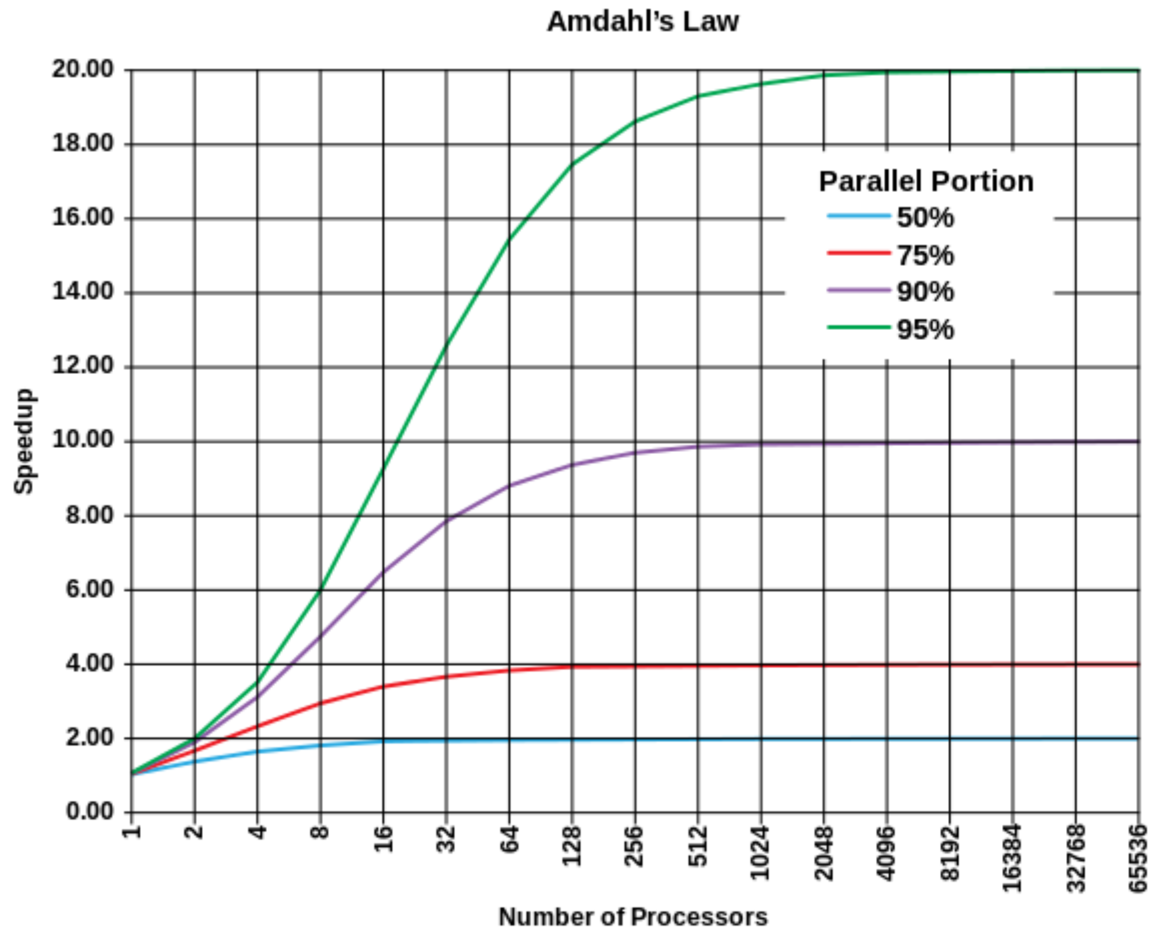
Amdahl lov for parallelle beregninger

- 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

Two independent parts **A** **B**

Original process

Make **B** 5x faster

Make **A** 2x faster

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: $a + P * b$

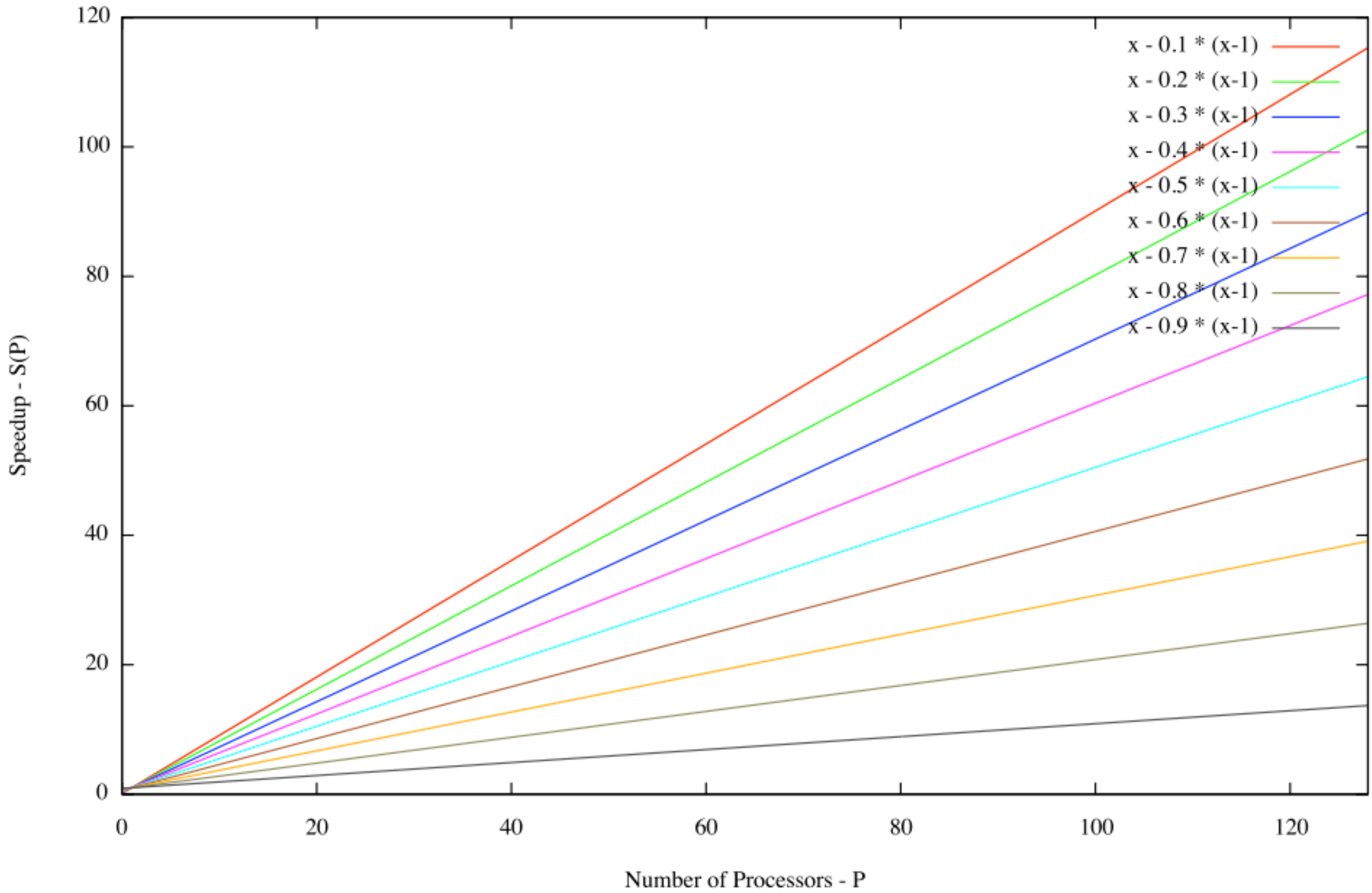
  Speedup er da:

  $\frac{a+P*b}{a+b}$, 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: $S(x) = x - \alpha(x-1),$
x - 0.1 * (x-1)
x - 0.2 * (x-1)
x - 0.3 * (x-1)
x - 0.4 * (x-1)
x - 0.5 * (x-1)
x - 0.6 * (x-1)
x - 0.7 * (x-1)
x - 0.8 * (x-1)
x - 0.9 * (x-1)
Speedup - S(P)
Number of Processors - P
0
20
40
60
80
100
120

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%
```

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!