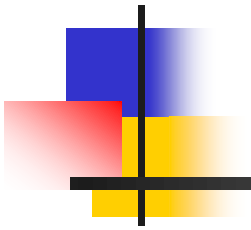
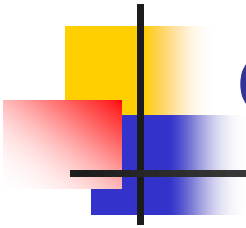


# INF2440 – Effektiv parallellprogrammering
# Uke 2, våren2014 - tidtaking

Arne Maus

OMS,

Inst. for informatikk



# Oppsummering – Uke1

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

# Tråder i Java (lett revidert og kompilerbar )

- 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();
                        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
```

# 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** | **1** | **2** | **20** | **200** | **2000** |
|---|---|---|---|---|---|
| Svar 1.gang<br>2. gang | 100 000<br>100 000 | 200000<br>159234 | 1290279<br>1706068 | 16940111<br>16459210 | 170127199<br>164954894 |
| Tap 1.gang<br>2. gang | 0 %<br>0% | 0%<br>20,4% | 35,5%<br>14,6% | 15,3%<br>17,7% | 14,9%<br>17,5% |

# Programmet som laget tabellen

kode for main-tråden
kode for trådene

```
import java.util.*;l
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)+ " millisek,");
             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
```

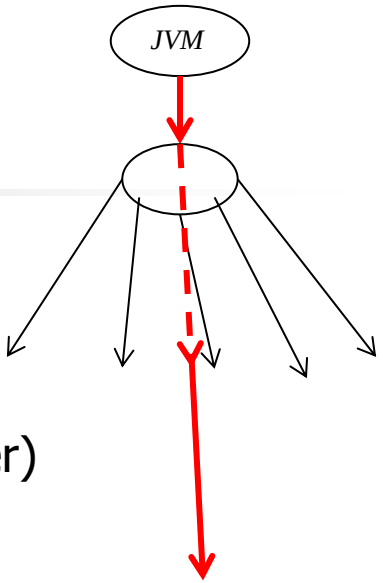
```
class Para implements Runnable{
                int ind;
                Para(int iind) { this.ind =ind;}

                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?

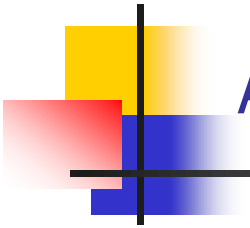
1. Kompleksitetsklasse:
   - O(1), O(logn), O(n), O(n*logn), O($n^{1,5}$ ), O($n^2$), …,NP
2. Størrelsen på data: n
   - Sorterer vi 100 eller 100 mil. tall?
   - Multipliserer vi to 4x4 matriser eller to 2000x2000 matriser?
3. Må vi synkronisere på delte data, må antall synkroniseringer være (minst) en orden lavere enn algoritmen pga. 'mye' overhead ved synkronisering.

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 > 0.2 sekund).**

# Plan for resten av Uke2

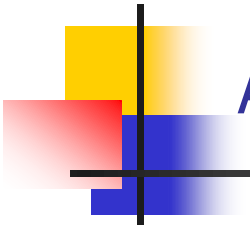



- I) Om å avslutte parallelle tråder
  - La dem bli ferdige med run-metoden,
    Hvordan teste at alle er ferdige ?
  - Synkronisert avslutning (Semaphore, CyclicBarrier)
  - new Thread – join() – avslutning
- II) Ulike synkroniseringsprimitiver
  - Vi skal bare lærte oss noen få - ett tilstrekkelig 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



# Avslutning med en CyclicBarrier

- En CyclicBarrier (cb= new CyclicBarrier (n+1))
  - Er tenkt som en ventested, en bom for n tråder (+ evt.  main) , alle må vente (sier cb.await() ) til sistemann ankommer køen, da kan alle fortsette.
  - Tråene er da ferdige og avslutter med å bli ferdige med sin 'run()-metode' og main forsetter og bruker deres resultat
  - 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



# Avslutning med en Semaphore

- En Semaphore (sf = new Semaphore(-n+1))
  - Administrerer (i dette tilfellet) –n+1 stk. **tillatelser**.
  - To sentrale primitiver:
    - sf.aquire() – ber om **en** tillatelse. Hvis det ikke er noen, 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.aquire() – 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.

# 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 ti 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

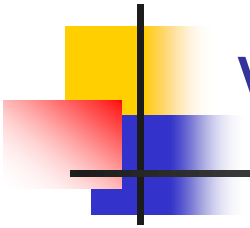
**Classes**

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

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

Eksempel på operasjoner i **AtomicIntegerArray:**

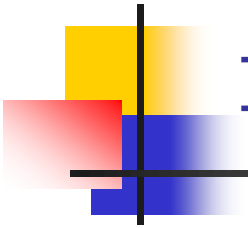
| | |
|---|---|
| int | **get**(int i) Gets the current value at position i. |
| int | **getAndAdd**(int i, int delta) Atomically adds the given value to the element at index i. |
| int | **getAndDecrement**(int i) Atomically decrements by one the element at index |
| void | **set**(int i, int newValue)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

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

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

# 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

```
M:\INF2440Para\FinnMax\FinnMaxokt2013>java FinnMaxMulti 10000 7

Kjøring:0, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     6.30 msek. , nanosek/n:  630.46
Max sekv = a:9853, paa:     0.28 msek. , nanosek/n:   28.38

Kjøring:1, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.57 msek. , nanosek/n:   56.87
Max sekv = a:9853, paa:     0.27 msek. , nanosek/n:   26.95

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

Kjøring:6, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.48 msek. , nanosek/n:   47.84
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.43

Median seq time:   0.014, median para time:   0.569,
 Speedup:    0.03, n = 10000
```

```
M:\INF2440Para\FinnMax\FinnMaxokt2013>java FinnMaxMulti 10000000 5

Kjøring:0, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    14.08 msek. , nanosek/n:    1.41
Max sekv = a:9999216, paa:     6.98 msek. , nanosek/n:    0.70

Kjøring:1, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     3.17 msek. , nanosek/n:    0.32
Max sekv = a:9999216, paa:     4.75 msek. , nanosek/n:    0.47

Kjøring:2, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     2.79 msek. , nanosek/n:    0.28
Max sekv = a:9999216, paa:     5.04 msek. , nanosek/n:    0.50

Kjøring:3, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     2.87 msek. , nanosek/n:    0.29
Max sekv = a:9999216, paa:     5.05 msek. , nanosek/n:    0.51

Kjøring:4, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     2.92 msek. , nanosek/n:    0.29
Max sekv = a:9999216, paa:     5.03 msek. , nanosek/n:    0.50

Median seq time:   5.052, median para time:   3.173,
 Speedup:    1.59, n = 10 000 000
```

n= 10 mill

```
M:\INF2440Para\FinnMax\FinnMaxokt2013>java -Xint FinnMaxMulti 10000000 5

Kjøring:0, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    67.24 msek. , nanosek/n:    6.72
Max sekv = a:9999216, paa:   179.40 msek. , nanosek/n:   17.94

Kjøring:1, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    64.00 msek. , nanosek/n:    6.40
Max sekv = a:9999216, paa:   175.12 msek. , nanosek/n:   17.51

Kjøring:2, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    51.42 msek. , nanosek/n:    5.14
Max sekv = a:9999216, paa:   176.23 msek. , nanosek/n:   17.62

Kjøring:3, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    64.95 msek. , nanosek/n:    6.49
Max sekv = a:9999216, paa:   173.17 msek. , nanosek/n:   17.32

Kjøring:4, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    60.11 msek. , nanosek/n:    6.01
Max sekv = a:9999216, paa:   185.84 msek. , nanosek/n:   18.58

Median seq time: 179.403, median para time:  64.950,
 Speedup:    2.76, n = 10 000 000
```

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

```
M:\INF2440Para\FinnMax>java FinnM 100000000 5
Kjoering:0, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 41.913504 millisek.
Max verdi sekvensiell i a:99989305, paa: 238.799921 millisek.

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

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

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

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


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

n= 100 mill

JIT-kompilering +optimalisering

Søppel-tømming

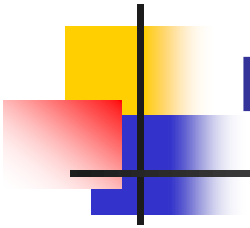
# Hva betyr dette for tidsmålingene

- 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
- Tidtagningen 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-1)/2]

# 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
- 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  (length-1)/2), som blir vår tidsmåling

```
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]+"millisec");
              }
              Arrays.sort(tidene);
              System.out.println("Median med svar:"+svar+", for trådene:"+tidene[(tidene.length-1)/2]+" millisec");
       } // 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
```



# Hva med operativsystemet:

- Linux og Windows har 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
```

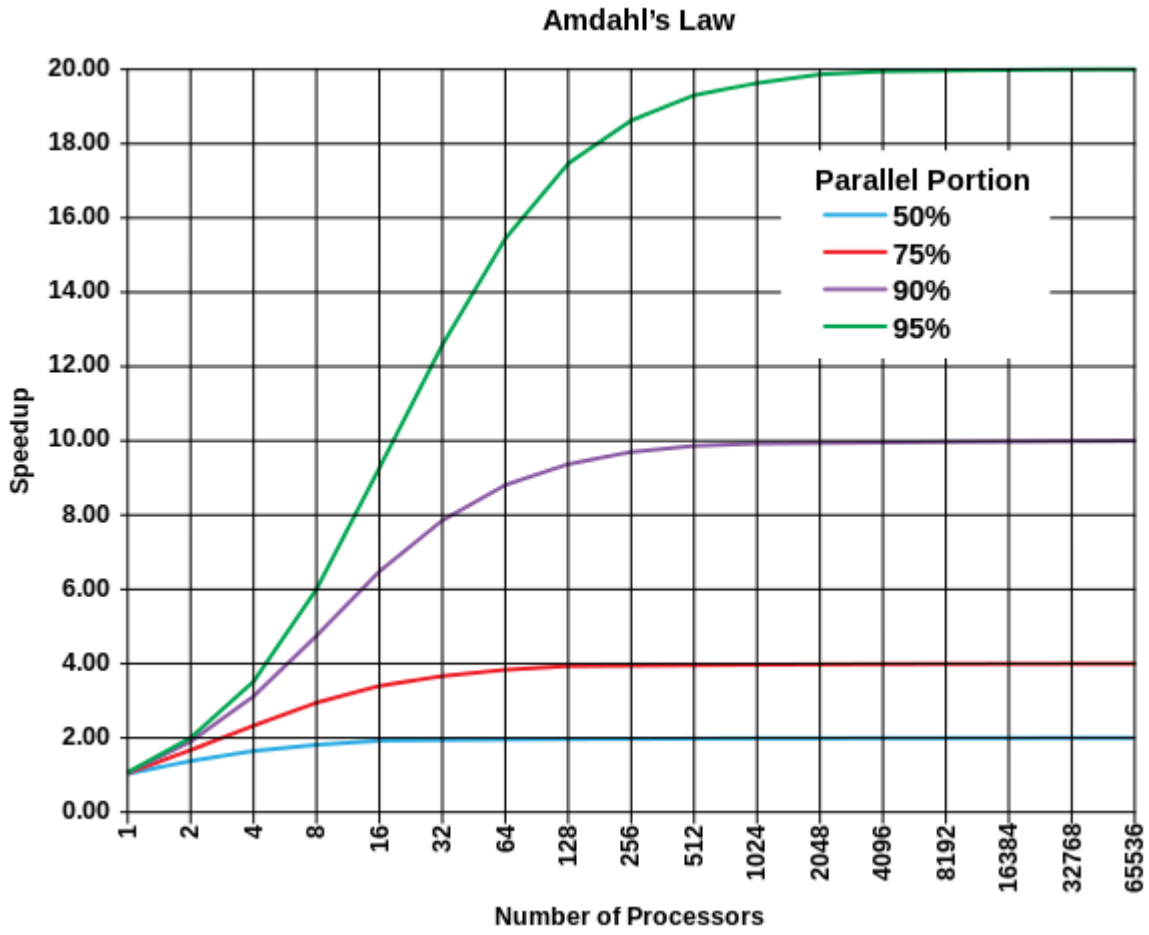
# Amdahl lov for parallelle beregninger

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

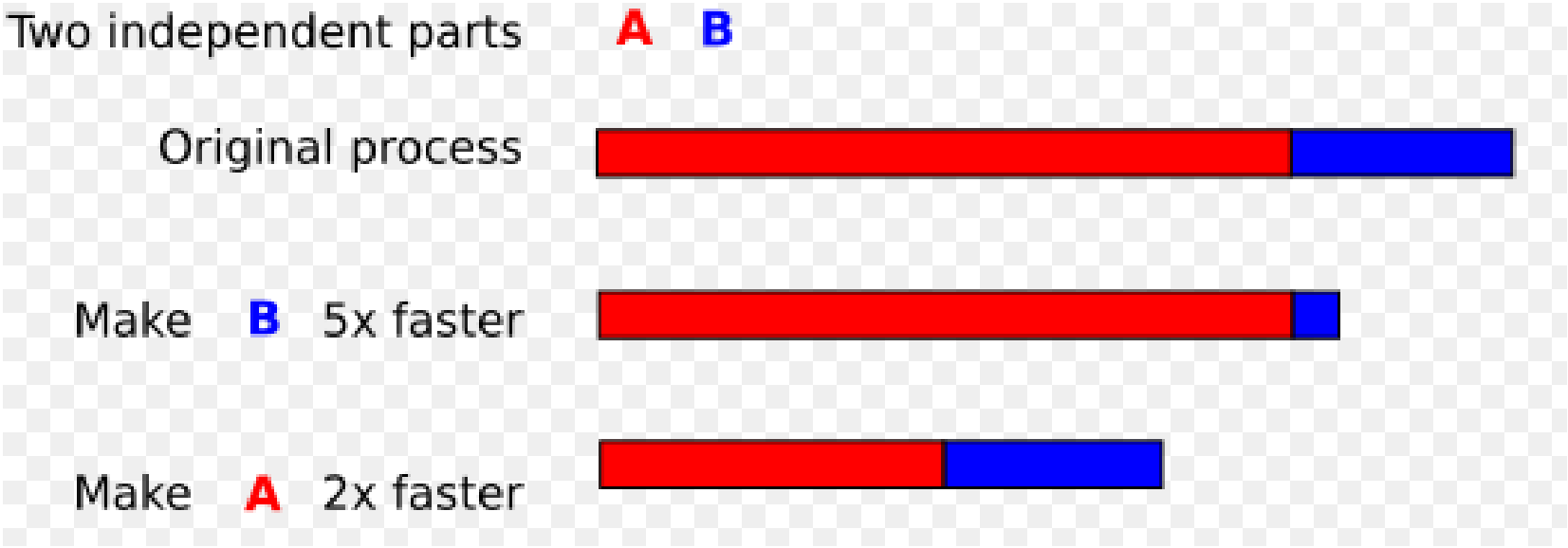
$$S = \frac{tid(sekvensiell)}{tid\ (parallell)} = \frac{1}{seq+p/k} = \frac{1}{1-p+p/k}$$

- 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 $\alpha$ andel sekvensiell kode, så er:

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

Fordi:

Parallell løsning: $a + b\ (a = sekvensiell\ tid, b = parallel\ tid)$

Sekvensiell løsning : $a + P * b$

Speedup er da:

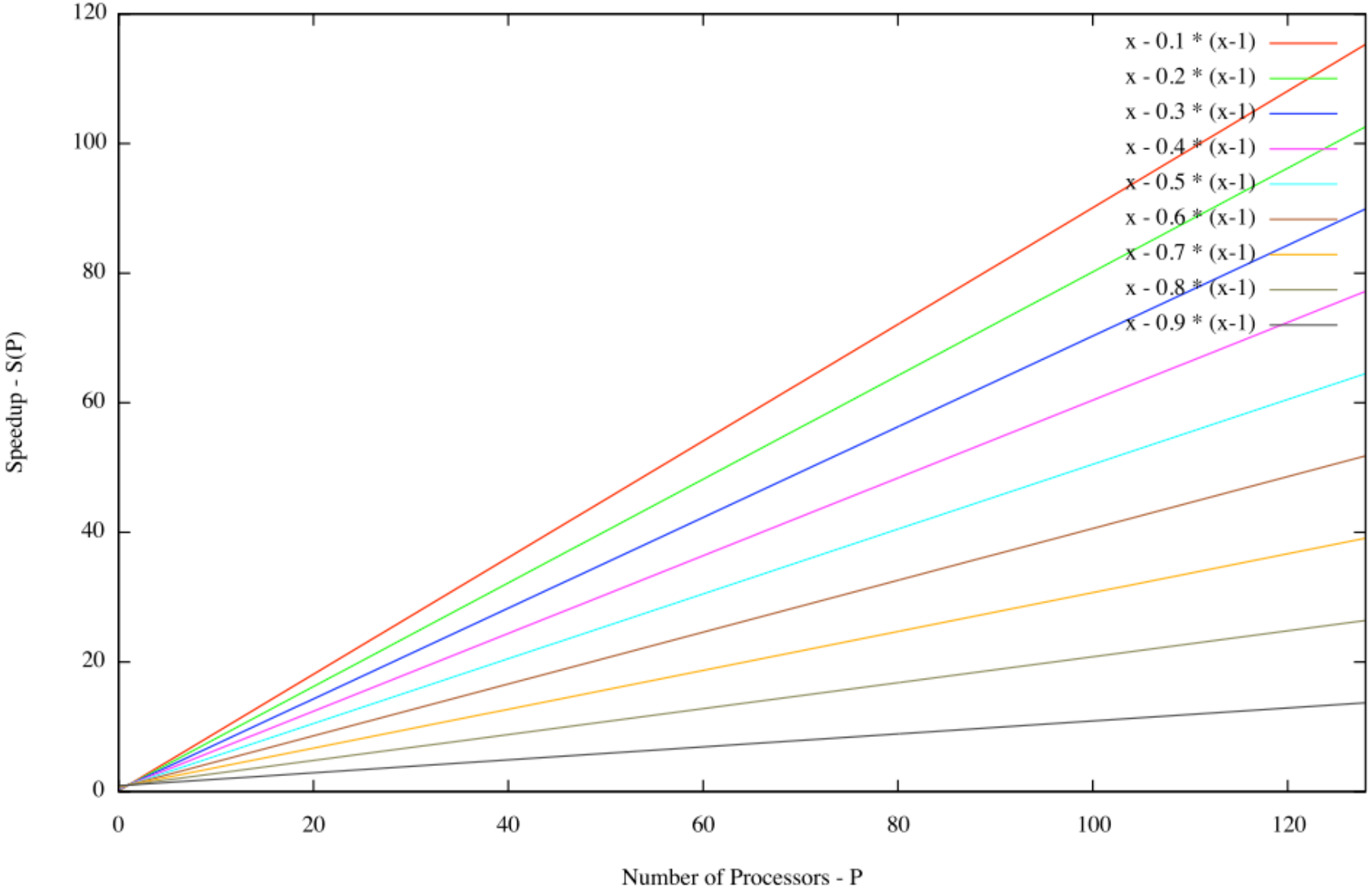
$(a + P * b)/(a + b\ )$, og definerer $\alpha = \frac{a}{a+b}$, så er:

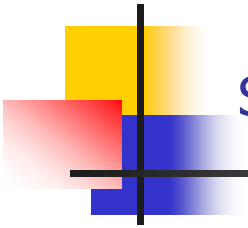
$$S(P) = \alpha + P * (1 - \alpha) = P\ - \alpha(P-1)$$

- Gustavson er mer optimistisk enn Amdahl, gir høyere speedup fordi han antar at med flere maskiner vil vi øke størrelsen på problemet.

  - «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 – 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 millisek.)

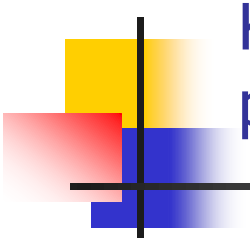
  Algoritmer vi skal jobbe med er mer av type 2 (Gustafson) enn type 3(Amdahl) men vi har alltid overhead, så små problemer løses lettest sekvensielt.

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

# Hva har vi sett på i Uke2

- 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, Operativsystem og søppeltømming
- IV) 'Lover' om Kjøretid
  - Amdahl lov
  - Gustafsons lov

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



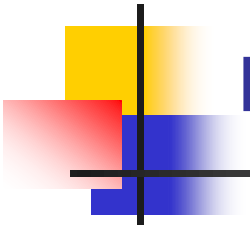
- Et problemet kunne være at når en av tråden lester opp et element i a[i]  (int = 4 byte), så er cache-linja 64 byte, så den får med seg flere elementer før og etter a[i].
- Disse 'andre' elementene er det andre tråder som skriver på.
- Vi skriver et testprogram (ParaArray) hvor 10 tråder med indeks : 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 i **ulike** elementer samtidig i en array går bra.
  - Dette skal vi bruke mye i kommende algoritmer.
  - (kan riktignok medføre noe ekstra eksekveringstid – det ser vi på senere)

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