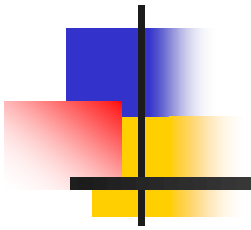


INF2440 Uke 13, v2014

Arne Maus
OMS,
Inst. for informatikk

Hva så vi på i Uke 12

I) Om «siffere» i Om Oblig 3

II) Om optimalisering av Oblig2 – nye tall med Java8

III) Java8

- Forbedringer
  - Feilen med tidtaking
  - Raskere Jit-kompilering
- Nytt:
  - Metoder som parametere til metoder (closure, lamda)
  - Akkumulatorer i `java.util.concurrent`

IV) Goldbachs hypotese – uløst problem siden 1742

- Løsning av en av Goldbachs problemer
- Parallellisering av denne

V) Datastrukturen i Oblig3 – ulike alternativer

Fra hjemmesida til INF2440:

Vennligst flere utfyller denne:

2. apr. 2014 11:08

Underveisevaluering av INF2440 Rediger

Vennligst fyll ut underveiseevalueringa av INF2440 (**få** spørsmål)

31. mar. 2014 09:04

BARE ni av dere har svart – vennligst ta bryet med et 5 min. svar.

+ Ingen gruppe eller forelesninger 14.-20. april (påske)

Hva skal vi se på i Uke13

I) Om vindus- (GUI) programmering i Java

- Tråder ! Uten at vi vet om det !

II) Hvordan pakke inn en parallell algoritme slik at andre kan bruke den.

- Kan du gi din Oblig4 til en venn eller en ny jobb som ikke har peiling på parallellitet og si at her er en metode som sorterer 3-10x fortere enn Arrays.sort() ?
- Nødvendige endringer til algoritmene, effektivitet !
- Fordelinger av tall vi skal sortere.
- Fornuftig avslutning av programmet ditt (hva med trådene)
- Brukervennlig innpakking !
- Dokumentasjon

III) Litt om løsning på prog1 og prog2 – finn én sum for alle n < 2 mrd. og alle (antall) Goldbach-summer for alle partall < 1 . mill. sekvensielt og parallelt.

.

Tråder og GUI

- (Nesten) alle vindu-systemer er basert på at det startes **én** egen tråd
  - Forsøk på å bruke flere tråder ender ofte i vranglås-situasjoner
- Java bruker: JFC/Swing component architecture (forbedret versjon av awt)
- At Java starter en tråd når vi starter opp, står det lite/intet om i dokumentasjonen (men leter man, finner man den)
- Da har vi to tråder :
  - main-tråden
  - the event dispatching thread , som har en helt egen modell for hendelser (muse-klikk, inntasting, osv) – INF1010-stoff
- Det beste er at man lar main-tråden **død/terminere** etter at vi har startet swing.
- swing er ikke det eneste som starter opp tråder (vær mistenksom om rare ting skjer)
- Fra API-dokumentasjonen: "In general Swing is not thread safe. All Swing components and related classes, unless otherwise documented, must be accessed on the event dispatching thread."

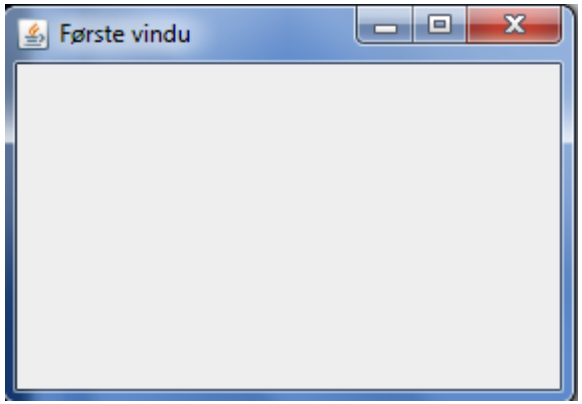
```
Import javax.swing.*;
class ForsteVindu extends JFrame {
      static int a =0;
     // Konstruktør. Lager og viser fram et vindu Rett På Java s.251.
    ForsteVindu() {
            setTitle("Første vindu");
            setSize(200, 200);
            a = 5;
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            setVisible(true);
    }

   public static void main(String[] args) {
              System.out.println("1) a="+a);

               // Lag et vindu som setter a =5
               SwingUtilities.invokeLater     (new Runnable() {
                        public void run() {
                          System.out.println("2) a="+a);
                           new ForsteVindu();
                          System.out.println("3) a="+a);
                        }
               } // end Runnable
              );
              System.out.println("4) a="+a);
    } // end main
}
```

Utskrift fra programmet: Her ser vi at vi har to tråder

- Vinduet som kommer opp



- I kommando-vinduet (merk rekkefølgen og verdien a)

main-tråden terminerer

```
Z:\INF2440Para\GrafiskSort>java ForsteVindu
1) a=0
4) a=0
2) a=0
3) a=5
```

To andre greie skjemaer for å starte swing:

```
public class MyApp implements Runnable {
    public void run() {
        // Invoked on the event dispatching
        //  thread. Construct and show GUI.
    }

  public static void main(String[] args) {
    SwingUtilities.invokeLater(new MyApp(args));
    }
}
```

javax.swing

Class JFrame

java.lang.Object
  java.awt.Component
    java.awt.Container
      java.awt.Window
        java.awt.Frame
          javax.swing.JFrame

JFrame har 23 egne metoder og arver ca. 410 metoder fra superklassene!

```
public class MyApp {
    MyApp(String[] args) {
        // Invoked on the event dispatching thread.
        //  Do any initialization here.
    }

    public void show() {
        // Show the UI.
    }

    public static void main(final String[] args) {
        // Schedule a job for the event-dispatching
        //  thread: creating and showing this
        // application's GUI.
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new MyApp(args).show();
            }
        });
    }
}
```

II) Hvordan pakke inn en parallell algoritme slik at andre kan bruke den?

- Vi ønsker:
  - a) Brukervennlighet
  - (Korrekte programmer)
  - b) Effektivitet
  - c) Opprydding
- Hvordan lage et bibliotek av parallelle algoritmer så de kan brukes i ordinære sekvensielle programmer uten at brukeren aner noe om parallelle programmer.
- c) Opprydding: Husk å rydde opp etter oss etter at programmet terminerer:
  - Vi ønsker ikke en rekke tråder som venter på ett eller annet når main-tråden terminerer!
  - For mange stadig nye tråder som ikke terminerer kan også 'kvele' et program (hukommelsen fylles opp med søppel)

II-a) Hvordan pakke inn en parallell algoritme slik at andre kan bruke den?

- Brukervennlighet – hvordan skal en 'naiv' bruker få adgang til vår fanatastisk raske, parallelle metode X
  - Eks: Parallell Quicksort
- Svar: Pakk den inn i en klasse: Sorting
- Brukeren sier enten – viktig valg:
  - **A)** `Sortering.quicksort(a);`
    - A- Quicksort er da en statisk metode.
  - **B)** `Sorting s = new Sorting(); s.quicksort(a);`
  - **C)** Eller: `new Sorting().quicksort(a);`
    - B,C - Quickssort er da en ikke-statisk metode, men en objekt-metode.
- Effektivitet og et rimelig krav om opprydding kan avgjøre hva vi velger: A,B eller C.

Effektivitet 1: Innpakking av vår fantastiske, parallelle quicksort.

- Både A (static) , B og C (klasse-metode) er mulig, men:
- Problemet er overhead fra å starte opp trådene + JIT
  - Ca. 2-3 millisekunder (ms) å lage ca. 4-8 tråder.
  - Viktigere: JIT-kompilering ?
- A) Static: **`Sort.quicksort(a)`**
  - 1. Da vil vi hver gang måtte starte nye tråder.
  - 2. Må vi da også begynne JIT-kompileringen om igjen?
    - den kan jo gi 50x- 200x raskere kode ?
- B) **`Sorting s = new Sorting();`**
  - Og hver gang en bruker skal sortere sier hun: **`s.quicksort(a);`**
  - Da får vi bare en gangs laging av trådene og hva med JIT-kompileringen?
- C) **`new Sorting().quicksort(a);`**
  - Overhead som A) – nye tråder hver gang og blir JIT-bevart ?

Lager først et enklere eksempel med sekvensiell kode

- Summer n (=8000) tall
- Ser bare på kjøretider, og konkluderer over til parallelle metoder.

```
import java.util.*;
class StaticA{
  public static void main(String[] args){
        if (args.length !=1 ) {
                System.out.println("bruk: >java StaticA <length of array>");
        } else {
                int len = Integer.parseInt(args[0]);
                for(int k = 0; k < 35; k++){
                   int[] arr = new int[len];
                    Random r = new Random();
                    for(int i = 0; i < arr.length; i++){
                                arr[i] = r.nextInt(len-1);
                    }
                long start  = System.nanoTime();
                long sum = A.summer(arr);
                long timeTaken = System.nanoTime() - start ;

            System.out.println((k+1)+") s="+sum+"  paa:"+timeTaken+" nanosek");
         } // end k
      }// end else
   } // end main
} // end StaticA

class A {
    static long summer(int [] arr){
            long sum = 0;
            for(int i = 0; i < arr.length; i++){
                sum += arr[i];
            }
       return sum;
     } // end summer
} // end class A
```

Statisk metode i class A:

```
A.summer(arr);
```

```
D:\INF2440Para\Static-ABC>java StaticA 8000
1) s=32167274   paa: 634787 nanosek
2) s=32741772   paa: 147822 nanosek
3) s=31728253   paa: 173228 nanosek
4) s=32436477   paa: 149361 nanosek
5) s=32033085   paa: 146667 nanosek
6) s=31704132   paa: 143973 nanosek
7) s=32068126   paa: 155905 nanosek
8) s=31757102   paa: 155521 nanosek
9) s=31592331   paa: 150902 nanosek
10) s=32202192  paa: 142432 nanosek
11) s=32105388  paa: 152056 nanosek
12) s=32093118  paa: 151671 nanosek
13) s=31760040  paa: 155906 nanosek
14) s=32046068  paa: 149361 nanosek
15) s=31988108  paa: 226352 nanosek
16) s=32124653  paa: 190552 nanosek
17) s=31934687  paa: 159370 nanosek
18) s=31939602  paa: 165914 nanosek
19) s=31813594  paa: 186702 nanosek
20) s=31785897  paa:1822365 nanosek
21) s=32285940  paa:    3464 nanosek
22) s=32275599  paa:    3080 nanosek
23) s=32151546  paa:    3849 nanosek
24) s=31862056  paa:    3464 nanosek
25) s=31671287  paa:    3849 nanosek
26) s=31840318  paa:    3465 nanosek
27) s=32067825  paa:    3465 nanosek
```

Statisk metode i class A:

```
A.summer(arr);
```

```
import java.util.*;

class ClassB{
  public static void main(String[] args){
        if (args.length !=1 ) {
                System.out.println("bruk: >java ClassB <length of array>");
        } else { B b = new B();
                int len = Integer.parseInt(args[0]);
                for(int k = 0; k < 35; k++){
                   int[] arr = new int[len];
                    Random r = new Random();
                    for(int i = 0; i < arr.length; i++){
                                arr[i] = r.nextInt(len-1);
                    }
                long start  = System.nanoTime();
                long sum = b.summer(arr);
                long timeTaken = System.nanoTime() - start ;

            System.out.println((k+1)+") s="+sum+"  paa:"+timeTaken+" nanosek");
         } // end k
      }// end else
   } // end main
} // end ClassB

class B {
      long summer(int [] arr){
            long sum = 0;
            for(int i = 0; i < arr.length; i++){
                sum += arr[i];
            }
       return sum;
     } // end summer
} // end class B
```

Objekt- metode i class B

```
B b = new B();
    …….
    b. summer(arr);
```

```
D:\INF2440Para\Static-ABC>java ClassB 8000
1) s=31927404  paa: 137813 nanosek
2) s=31805822  paa: 156676 nanosek
3) s=32078058  paa:1661454 nanosek
4) s=32109706  paa:   3464 nanosek
5) s=32196482  paa   :3080 nanosek
6) s=32044765  paa:   3080 nanosek
7) s=31661648  paa:   3850 nanosek
8) s=31516447  paa:   3465 nanosek
9) s=31815483  paa:   3079 nanosek
10) s=31846239  paa:  3465 nanosek
11) s=31752719  paa:  3465 nanosek
12) s=31878804  paa:  3464 nanosek
13) s=31941947  paa:  5004 nanosek
14) s=31810101  paa:  3465 nanosek
15) s=31730506  paa:  3464 nanosek
16) s=31826011  paa:  3465 nanosek
17) s=31976545  paa:  3464 nanosek
18) s=31900289  paa:  5774 nanosek
19) s=32008971  paa:  5004 nanosek
20) s=31801909  paa:  5389 nanosek
21) s=31535942  paa:  6544 nanosek
22) s=32033130  paa:  4620 nanosek
23) s=31692354  paa:  5004 nanosek
24) s=31878940  paa:  3464 nanosek
25) s=32073671  paa:  3465 nanosek
26) s=31876982  paa:  3465 nanosek
27) s=31778065  paa:  4619 nanosek
```

Objekt- metode i class B

```
B b = new B();
……
b. summer(arr);
```

```
import java.util.*;

class ClassC{
  public static void main(String[] args){
        if (args.length !=1 ) {
                System.out.println("bruk: >java ClassC <length of array>");
        } else {
                int len = Integer.parseInt(args[0]);
                for(int k = 0; k < 35; k++){
                   int[] arr = new int[len];
                    Random r = new Random();
                    for(int i = 0; i < arr.length; i++){
                                arr[i] = r.nextInt(len-1);
                    }
                long start  = System.nanoTime();
                long sum = new C().summer(arr);
                long timeTaken = System.nanoTime() - start ;

            System.out.println((k+1)+") s="+sum+"  paa:"+timeTaken+" nanosek");
         } // end k
      }// end else
   } // end main
} // end ClassC

class C {
      long summer(int [] arr){
           long sum = 0;
           for(int i = 0; i < arr.length; i++){
               sum += arr[i];
           }
      return sum;
    } // end summer
} // end class C
```

Objekt- metode i class C

```
new C().summer(arr);
```

```
D:\INF2440Para\Static-ABC>java ClassC 8000
1) s=32087203  paa:651339 nanosek
2) s=31544151  paa:156291 nanosek
3) s=32044431  paa:158986 nanosek
4) s=32121418  paa:271006 nanosek
5) s=32022808  paa:  46194 nanosek
6) s=32046616  paa:   6159 nanosek
7) s=32040178  paa:   6544 nanosek
8) s=32010535  paa:   7699 nanosek
9) s=32175599  paa:   7699 nanosek
10) s=32414924  paa:   6544 nanosek
11) s=32087937  paa:   3465 nanosek
12) s=32184513  paa:   3464 nanosek
13) s=32247691  paa:   5005 nanosek
14) s=32041925  paa:   4620 nanosek
15) s=32015535  paa:   5004 nanosek
16) s=32083860  paa:   3849 nanosek
17) s=31979593  paa:   5390 nanosek
18) s=31661029  paa:   4620 nanosek
19) s=31688502  paa:   5004 nanosek
20) s=31983432  paa:   5005 nanosek
21) s=32031745  paa:   3464 nanosek
22) s=31954441  paa:   5004 nanosek
23) s=32008609  paa:   5004 nanosek
24) s=32380944  paa:   5004 nanosek
25) s=31992932  paa:   4235 nanosek
26) s=31929029  paa:   3464 nanosek
27) s=32065607  paa:   3079 nanosek
```

Objekt- metode i class C

```
new Sorting().quicksort(a);
```

Konklusjon om statiske eller objekt-metoder

- Omtrent samme forbedring uansett hvordan vi deklarerer og bruker metoden 'summer()'
- JIT-kompileringen ga ca. 200x raskere kjøretid og alle alternativene brukte til sist ca. 3400 nanosek. på å summere 8000 tall.
- Når vi bruker noe i en klasse som har statiske variabler eller statiske metoder, med blir det opprettet et Klasseobjekt som er der under hele kjøringen av programmet
- Når vi har klasser med objekt-metoder og objekt-variable, skilles det mellom data og metoder.
- Data er egentlig objektene, mens metodene ligger bare ett sted.
- Når et man kaller en metode i et objekt, får 'bare' metodene en ekstra parameter som forteller hvor data-klumpen (objektet) er.

Konklusjon: Metodene JIT kompileres hver gang de brukes og ligger bare ett sted under 'hele' tiden under programmets levetid.

Hva med parallell kode – ville da tidene være like?

- Prog A : statisk: Sorting.quicksort(): Måtte sannsynligvis startet k tråder hver gang vår statiske quicksort ble kalt.
- ProgB : b = new Sorting(); b.quicksort();
  Her lager vi trådene bare en gang, og bare metodekallet utføres for hver sortering.
- ProgC: new Sorting().quicksort(); - her må også vi starte alle trådene hver gang vi skal sortere + at vi må først lage et nytt objekt.

Konklusjon: Metode B er å foretrekke – noe bedre speedup for store n, og speeedup >1 for noe lavere verdi av n,

- men vil måtte ha en egen exit()-metode for å rydde opp og fjerne trådene.
- Ved A og C kan og må vi fjerne objektene før sorteringa avsluttes.

(Vi skal senere se på hva JIT-kompilering gjør med:
new Thread(…);)

Hva med å lage nye tråder hver gang (A og C) eller bare en gang (B)

- Lager et program som lager 8 tråder og kjører de 20 ganger med JIT-kompilering?
- Trådene gjør intet (øker en variabel med 1 for ikke å bli optimalisert bort)

```
D:\INF2440Para\ThreadTest>java ThreadTest
Starting 8 threadsLooping 20 times.
 Time in us per Thread:
Loop #1 : 220.194375 us
Loop #2 : 129.585625 us
Loop #3 : 134.3975 us
Loop #4 : 169.0915 us
Loop #5 : 179.485375 us
Loop #6 : 184.970875 us
Loop #7 : 890.0645 us
Loop #8 : 133.338875 us
Loop #9 : 502.3665 us
Loop #10 : 354.73625 us
Loop #11 : 180.977 us
Loop #12 : 178.715375 us
Loop #13 : 759.853375 us
Loop #14 : 149.891875 us
Loop #15 : 561.50525 us
Loop #16 : 178.23425 us
Loop #17 : 181.07325 us
Loop #18 : 173.229875 us
Loop #19 : 187.954375 us
Loop #20 : 161.488625 us
Average time: 280 us per Thread
```

Konklusjon: Å starte en tråd og vente på at den avslutter join() blir JIT-kompilert

– med f.eks: 1 tråd generert 600 ganger kan vi få speedup på 80 (inkluderer da JIT-kompilering),

-men speedup på 8 tråder hver gang 60 ganger får vi speedup på ca. 4.

- new Thread(..) start og avslutning (join) kan per tråd ta mindre enn 50 us (100 tråder 800 ganger) med speedup 4.8.

```
class ThreadTestB {
static int s;
    public static void main(String[] args) {
        int  numberofthreads = 100, numberofloops   = 800;
        double min = 2000000000, max =0, speedup, timeused;
        long timestart, timeend, timeavg = 0;
        System.out.println("Starting " + numberofthreads + " threads" +
                           "Looping " + numberofloops + " times");

        for (int i = 0; i < numberofloops; i++) {
            timestart = System.nanoTime();
            Thread[] t = new Thread[numberofthreads];

            for (int j = 0; j < numberofthreads; j++)
                (t[j] = new Thread( new ExThread())).start();

            try {for (int k = 0;k< numberofthreads;k++) t[k].join();
            }catch (Exception e) return;}

            timeend = System.nanoTime();
             timeused = ((timeend - timestart)/(1000.0*numberofthreads));
            if (min > timeused) min = timeused;
            if (max < timeused) max = timeused;
            timeavg += timeused ;
            System.out.println("Loop #" + (i+1) + " :
               " + ((timeend - timestart)/(1000.0*numberofthreads) )+ " us");
        }
        timeavg = (timeavg / numberofloops);
        System.out.println("\nAverage time: " + timeavg + " us, max:"+max+
                           ", min:"+min+", speedup:"+ (max/min));
    } }
```

```
class ExThread implements Runnable
{
    public void run() {
        try {
              ThreadTestB.s++;
        }catch (Exception e) {return;}
}}
```

IIb) Nødvendige endringer/tillegg til algoritmen

- Hva er mest typisk med parallelle algoritmer?
- Svar For små verdier av n er de virkelig langsomme!

```
>java QuickSort 100 10 100000000
      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
```

- Løsning: Vi må bruke **den sekvensielle** versjonen av algoritmen under en viss grense – eks: n < 50 000.
- For sortering har vi også en algoritme: Innstikksortering, som er klart raskere enn alle andre sorteringer; n <50

Skisse 1 av algoritmen

```
static void quicksort( int [] a) {
      if  ( n < INSERT_LIMIT) innstikkSortering (a);
      else if( n < PARA_LIMIT) sekvensiellQuicksort(a)
      else {

      <gjør parallell quickSort>

      }
} // end quicksort
```

Vi låner ett triks til fra Arrays.sort() – spesielle fordelinger av tallene vi skal sortere

- Arrays.sort var tidligere en grei, litt langsom implementasjon av Quicksort – si 200 LOC (Lines Of Code)
- Nå, i alle fall fom. Java 1.6 er den på ca. 1200 LOC
- De har lagt inn en del spesiell kode for å dekke spesielle fordelinger
- Noen fordelinger er enkle(re) å sortere:
  - Anta at a[] er sortert stigende ($a[i] \leq a[i+i] \ \forall \ i < n-1$)
  - Anta at a[] er synkende sortert ($a[i] \geq a[i+i] \ \forall \ i < n-1$)
  - Det er mange like verdier blant de n tallene.
  - (andre fordelinger kan være spesielt vanskelige – særlig for Radix-sortering)
- Vi skal lage spesialløsninger for de to første av disse, den tredje har vi allerede løst (mer om noen foiler)

Hvordan sjekke om forlengs og baklengs sortert ? -og hva gjør vi da ?

```
static void quicksort(int [] a) {
        if (! sortert (a) )
        else if (! baklengsSortet(a) ) {
        else if ( n < INSERT_LIMIT) innstikkSortering (a);
        else if( n < PARA_LIMIT) sekvensiellQuicksort(a)
        else {

                <gjør parallell quickSort>

                }
} // end quicksort
```

sortert() og baklengsSortert ()

```
boolean sortert (int [] a) {
    int t = a[0], neste;
    for( int i = 1; i < a.length; i++) {
       neste = a[i];
       if (t > neste ) return false;
       else t=neste;
    }
    return true;
}// end sortert
```

```
// IKKE denne – hvorfor denne som //
brukes av Arrays.sort ?
boolean sortert (int [] a) {
     for( int i = 1; i < a.length; i++) {
        if (a[i-1] > a[i]) return false;
     }
    return true;
}// end langsommere sortert
```

```
boolean baklengsSortert(int [] a) {
    int t = a[0], neste, i;
    for( i = 1; i < a.length; i++) {
       neste = a[i];
       if (t < neste ) break;
       else t=neste;
    }
    if ( i < a.length) return false;
    // reverse all elements in a[]
    int temp, slutt = a.length-1,
         stopp = a.length/2;;
    for( i = 0; i < stopp; i++) {
            temp = a[i];
            a[i] = a[slutt];
            a[slutt--] = temp;
      }
      return true;
} // end baklengsSortert
```

Vi går gjennom arrayen a[] to ekstra ganger ?

- Fail-fast

Hvordan sikre oss mot mange like elementer?

```
void quicksortSek(int[] a, int left, int right) {
    if (right-left < LIMIT) insertSort (a,left,right);
    else {  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 >0) quicksortSek(a, left,  piv2);
            if ( right-piv >1) quicksortSek(a, piv + 1, right);
         }
} // end quicksort
```

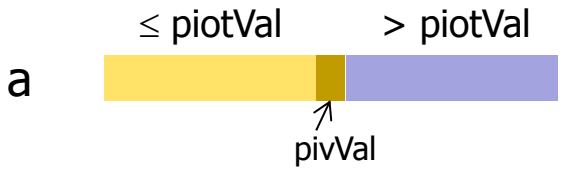
```
// del opp a[] i to: smaa og storre
int partition (int [] a, int left, int right) {
    int pivVal = a[(left + right) / 2];
    int index = left;
    // plasser pivot-element helt til høyre
    swap(a, (left + right) / 2, right);

    for (int i = left; i < right; i++) {
        if (a[i] <= pivVal) {
                    swap(a, i, index);
                    index++;
        }
     } // end for
     swap(a, index, right); // sett pivot tilbake
     return index;
} // end partition
```

```
// Hvorfor ikke denne – kortere kode ??
void quicksortSek2(int[] a, int left, int right) {
    if (right-left < LIMIT) insertSort (a,left,right);
    else {  int piv = partition (a, left, right); // del i to

    if ( piv-left >1) quicksortSek(a, left,  piv2);
    if ( right-piv >1) quicksortSek(a, piv + 1, right);
  }
} // end quicksort
```

Etter: partition(a,left,right)



Hva betyr fordelinga,
Forsøk1 : Sekv. og para med while-løkke og piv2

```
a = new int[n];
Random r = new Random (123);
for (int i =0; i<n; i++) a[i] = r.nextInt(n);
```

n	sekv.tid(ms)	para.tid(ms)	Speedup
20000000	2058.949	612.130	3.3636
2000000	248.645	73.054	3.4036
200000	22.083	11.696	1.8881
20000	1.868	1.619	1.1537
2000	0.126	0.138	0.9114
200	0.007	0.008	0.8262

```
// Forsøk 1: fyll a[] med 0,1,2,...,9, 0,1  både sekvensiell og paralle
for (int i =0; i<n; i++) a[i] = i%10;
```

n	sekv.tid(ms)	para.tid(ms)	Speedup
20000000	216.149	203.474	1.0623
2000000	17.205	22.083	0.7791
200000	1.658	6.363	0.2605
20000	0.175	0.177	0.9881
2000	0.020	0.019	1.0182
200	0.007	0.009	0.7600

Mye raskere algoritmer sekv. og para. (men speedup litt langsom)

Hva betyr fordelinga:
Forsøk2 : Sekv. og para **uten** while-løkke og piv2

```
a = new int[n];
Random r = new Random (123);
for (int i =0; i<n; i++) a[i] = r.nextInt(n);
```

n	sekv.tid(ms)	para.tid(ms)	Speedup
2000000	248.645	73.054	3.4036
200000	22.083	11.696	1.8881
20000	1.868	1.619	1.1537
2000	0.126	0.138	0.9114
200	0.007	0.008	0.8262
20	0.001	0.001	1.0010

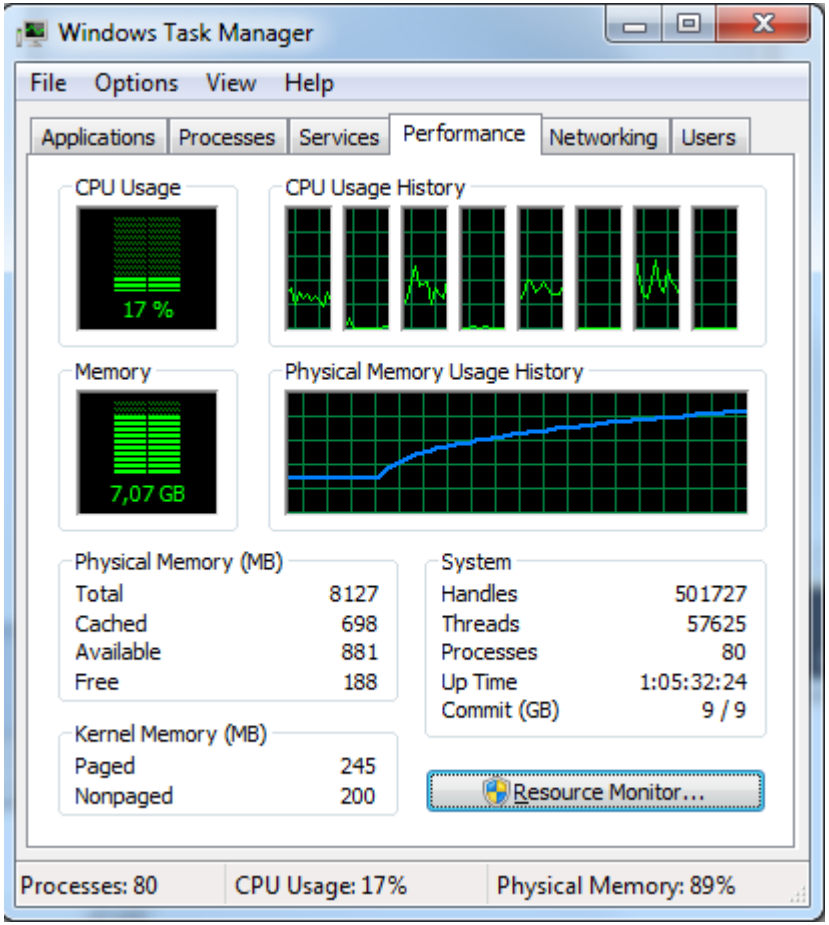
```
// Forsøk 2: fyll a[] med 0,1,2,...,9, 0,1  både sekvensiell og para
for (int i =0; i<n; i++) a[i] = i%10;
// Største n =20 000 det virket for:
```

n	sekv.tid(ms)	para.tid(ms)	Speedup
20000	61.692	60.320	1.0227
2000	0.679	0.663	1.0238
200	0.013	0.014	0.9000
20	0.000	0.001	0.5000

Forsøk 3: n= 200 000, Random for sekvensiell, 0,1,2..,9 for parallell – begge **uten** while-løkke og piv2.

Memory like før :

java.lang.OutOfMemoryError: unable to create new native thread



Forsøk 3: java.lang.OutOfMemoryError: unable to create new native thread

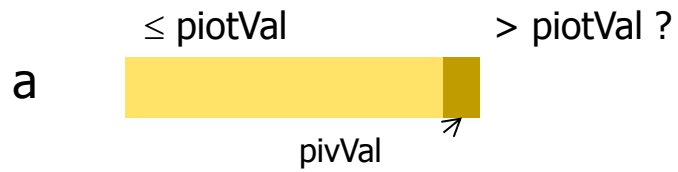
```
n       sekv.tid(ms)    para.tid(ms)     Speedup
Java HotSpot(TM) 64-Bit Server VM warning: Attempt to allocate stack guard pages
 failed.
Exception in thread "Thread-117551" java.lang.OutOfMemoryError: unable to create
 new native thread
        at java.lang.Thread.start0(Native Method)
        at java.lang.Thread.start(Unknown Source)
        at QuickSort.RekPara(QuickSort.java:212)
        at QuickSort$Para.run(QuickSort.java:234)
        at java.lang.Thread.run(Unknown Source)
Exception in thread "Thread-117557" java.lang.OutOfMemoryError: unable to create
 new native thread
        at java.lang.Thread.start0(Native Method)
        at java.lang.Thread.start(Unknown Source)
        at QuickSort.RekPara(QuickSort.java:209)
        at QuickSort$Para.run(QuickSort.java:234)
        at java.lang.Thread.run(Unknown Source)
Exception in thread "Thread-117547" java.lang.OutOfMemoryError: unable to create
 new native thread
        at java.lang.Thread.start0(Native Method)
        at java.lang.Thread.start(Unknown Source)
        at QuickSort.RekPara(QuickSort.java:209)
        at QuickSort$Para.run(QuickSort.java:234)
        at java.lang.Thread.run(Unknown Source)
```

Forsøk4 : n= 200 000; 0,1,2..,9 for både parallell og sekvensiell:
Begge uten while-løkke og piv2 : Stack overflow

```
at QuickSort.quicksortSek(QuickSort.java:153)
        at QuickSort.quicksortSek(QuickSort.java:153)
        at QuickSort.quicksortSek(QuickSort.java:153)
        at QuickSort.quicksortSek(QuickSort.java:153)
        at QuickSort.quicksortSek(QuickSort.java:153)
        at QuickSort.quicksortSek(QuickSort.java:153)
        at QuickSort.quicksortSek(QuickSort.java:153)
        at QuickSort.quicksortSek(QuickSort.java:153)
        at QuickSort.quicksortSek(QuickSort.java:153)
        at QuickSort.quicksortSek(QuickSort.java:153)
        at QuickSort.quicksortSek(QuickSort.java:153)
        at QuickSort.quicksortSek(QuickSort.java:153)
        at QuickSort.quicksortSek(QuickSort.java:153)
        at QuickSort.quicksortSek(QuickSort.java:153)
        at QuickSort.quicksortSek(QuickSort.java:153)
        at QuickSort.quicksortSek(QuickSort.java:153)
        at QuickSort.quicksortSek(QuickSort.java:153)
        at QuickSort.quicksortSek(QuickSort.java:153)
        at QuickSort.quicksortSek(QuickSort.java:153)
        at QuickSort.quicksortSek(QuickSort.java:153)
        at QuickSort.quicksortSek(QuickSort.java:153)
        at QuickSort.quicksortSek(QuickSort.java:153)
```

Hvis det er mange like elementer – f.eks alle like

Etter: partition(a,left,right)




```
// Hvorfor ikke denne – kortere kode ??
void quicksortSek2(int[] a, int left, int right) {
     if (right-left < LIMIT) insertSort (a,left,right);
     else {  int piv = partition (a, left, right); // del i to

     if ( piv-left >1) quicksortSek(a, left,  piv2);
     if ( right-piv >1) quicksortSek(a, piv + 1, right);
   }
} // end quicksort
```

Anta at det er n like elementer i a[], da vil quicksortSek2 hver gang :
- bare skille ut ett element på høyre side
- vestre side ville være n-1 lang

Hvis n = 1 mill, ville venstresiden av treet være 1. mill dypt – vi får Stack overflow og en meget langsom algoritme

Derimot quicksortSek med while-løkke og piv2, vil bare få ett kall og bli ferdig meget raskt :

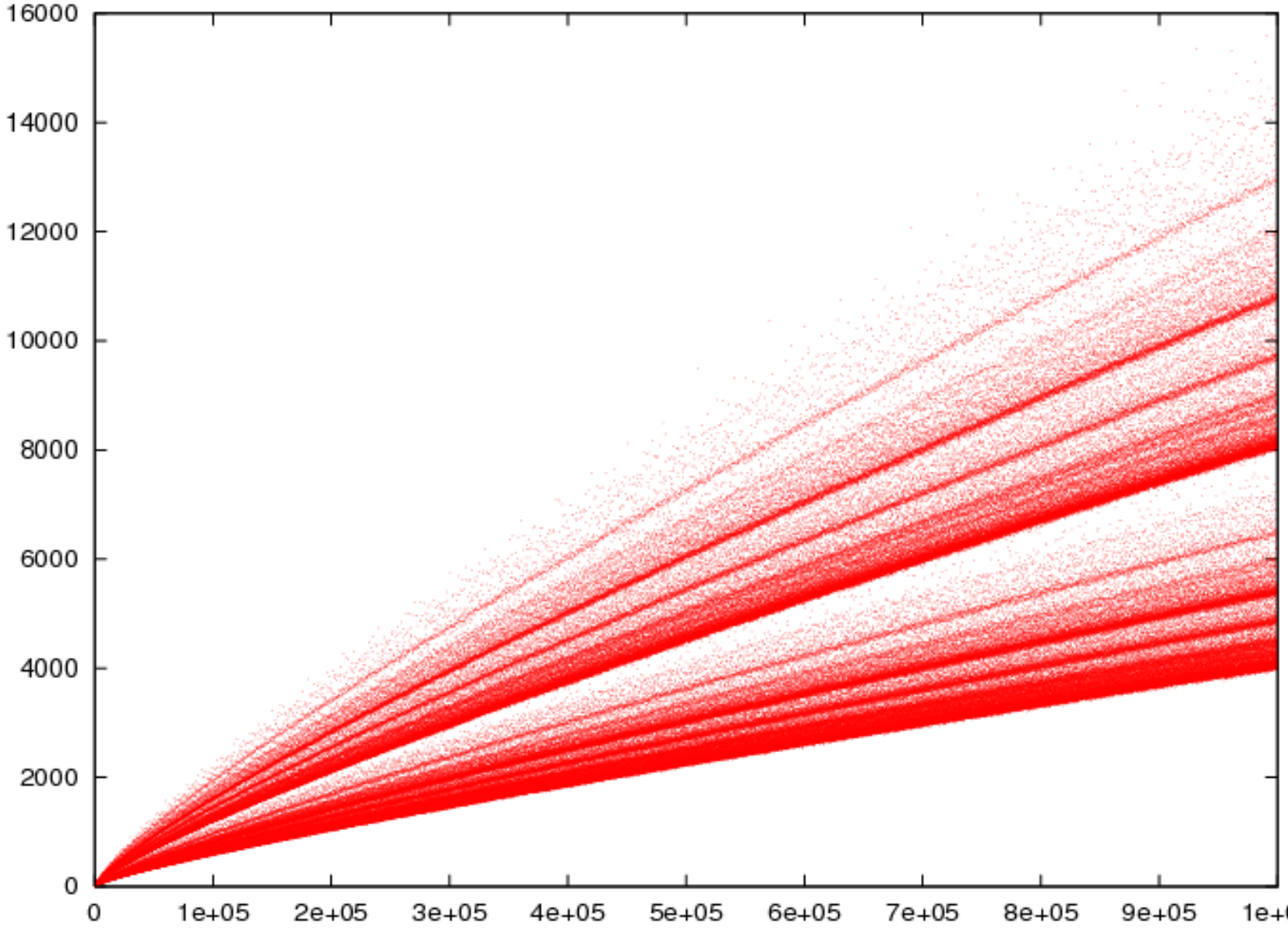
```
int piv2 = piv-1,   pivotVal = a[piv];

 while (piv2 > left && a[piv2] == pivotVal) {
piv2--; // skip like elementer i midten
```

III) Christian Goldbachs påstand i 1742:

- Alle partall m =n+n > 4 kan skrives som en sum av to primtall som er oddetall.
  - m = p1+p2 (p1 $\leq$ p2)
  - Eks: 6 = 3+3, 14 = 7+7 og 14 = 11+3,….
- Antall slike ulike summer av to primtall for en gitt m kaller vi G(m).
- Bevis at G(m) > 0 $\forall$ m.

G(n) for alle tall < 1 mill – varierer mye, men har en skarp nedre grense som vi ikke greier å vise er > 0 !



Hvordan vise Goldbach med et program

- En riking i USA har utlovet $1mill for løsning på 10 matematiske problemer som lenge var uløst.
  - Goldbach er en av disse problemene, og premien er ikke hentet.
  - To måter å vise Goldbach:
    - Komme med et matematisk bevis for at den er sann.
    - Komme med et mot-eksempel: Partallet M som ikke lar seg skrive som summen av to primtall.

Hvordan løser vi dette ? Første sekvensielt

- Hvilket program skal vi lage?
  - **a)** **Prog1**: Som for alle tall m=6,8,….,n finner minst én slik sum m= p1+p2
  - **b)** **Prog2**: Som regner ut G(m) for m=6,8,….,n
  - **c)** **Prog3**: Som tester noen tall > 4×10^{18} om Goldbach gjelder også for disse tallene
    (sette ny rekord – ikke testet tidligere, kanskje finne ett tall som ikke lar seg skrive som summen av to primtall.)

Forskjellen på Prog1/Prog2 og Prog3 er at i Prog1 og Prog2 antar vi at vi vet alle primtall < m, det trenger vi ikke i Prog3 (som da blir en god del langsommere).

Prog1 og Prog2 bruker 'bare' Eratosthenes Sil fra Oblig2.

Prog3 bruker både Eratosthenes Sil og faktoriseringa fra Oblig2.

Skisse av Prog1 : Finn minst én slik sum m=p1+p2, ∀ m <n, m partall, p1 ≤ p2; sekvensielt program

```
<Les inn: n>
<Lag e= Eratosthens Sil(n)>

for (int m = 6 ; m < n; m+=2) {
    // for neste tall m, prøv alle primtall p1 ≤ m/2
     for (int p1 = 3, p1 <= m/2; e.nextPrime(p1){
         if ( e.isPrime(m-p1)) {
             //  System.out.println( m+" = "+ p1 + " + "+ (m-p1) );
             break;
         }
     } // end p1
      if (p1 > m/2) System.out.println(
                    " REGNEFEIL: (Goldbach bevist for n < 4*10**18:"
                     + m + " kan ikke skrives som summen av to primtall ");
} // end m
```

Skisse av Prog2 : Finn G(m) antall slike summmer: m = p1+p2, $\forall$ m <n, m partall, p1 $\leq$ p2; sekvensielt program

```
<Les inn: n>
<Lag e= Eratosthens Sil(n)>
int Gm;

for (int m = 6 ; m < n; m+=2) {
     Gm = 0;
    // for neste tall m, prøv alle primtall p1 ≤ m/2
     for (int p1 = 3, p1 <= m/2; e.nextPrime(p1){
         if ( e.isPrime(m-p1)) { Gm++; }
     } // end p1
     if (Gm == 0)
           println(" REGNEFEIL: (Goldbach bevist for n < 4*10**18):"
                    + m + " kan ikke skrives som summen av to primtall ");
      else println(" Antall Goldbachsummer i "+m+" er:"+Gm);
} // end m
```

Prog1: Hvordan parallellisere å finne én sum ?

- Vi har en dobbelt løkke (m og p1)
- Parallelliser den innerste løkka:
  - Deler opp primtallene og lar ulike tråder summere med ulike p1
  - Sannsynligvis uklokt?
- Parallelliserer den ytterste løkka:
  - Hvis vi lar hver tråd få 1/k-del av tallene å finne p1 for.
  - Med n= 2mrd. får hver tråd 250 mill. tall å sjekke.

```
<Les inn: n>
<Lag e= Eratosthens Sil(n)>

for (int m = 6 ; m < n; m+=2) {
   // for neste tall m, prøv alle primtall p1 ≤ m/2
    for (int p1 = 3, p1 <= m/2; e.nextPrime(p1){
        if ( e.isPrime(m-p1)) {
              break;  // funnet sum
        }
    } // end p1
     if (p1 > m/2) System.out.println(" REGNEFEIL for "+
m);
} // end m
```

  - Litt ulik belastning på trådene, da større tall må lete lenger etter den første p1 hvor m-p1 er primtall.
- Parallelliserer ved å lage en rekursjon på ytterste løkke.
  - Kan bruke PRP
  - Mye kortere kode

Konklusjon: Forsøker først å parallellisere den ytterste løkka ?

Prog 1,

finn én sum:	n	sekv(ms)	para(ms)	Speedup	SekElem(us)	ParaElem(us)
EratosthesSil	2000000000	14872.89	7386.36	2.01		
Goldbach Sum	2000000000	131447.23	31222.86	4.21	0.06572	0.01561
Total tid	2000000000	146350.58	38813.12	3.77		
EratosthesSil	200000000	1156.82	336.60	3.44		
Goldbach Sum	200000000	11224.35	2646.68	4.24	0.05612	0.01323
Total tid	200000000	12395.19	2990.45	4.14		
EratosthesSil	20000000	69.13	29.94	2.31		
Goldbach Sum	20000000	946.12	252.25	3.75	0.04731	0.01261
Total tid	20000000	1015.25	273.34	3.71		
EratosthesSil	2000000	6.26	1.92	3.27		
Goldbach Sum	2000000	81.11	18.89	4.29	0.04056	0.00945
Total tid	2000000	87.38	20.61	4.24		
EratosthesSil	200000	0.56	0.40	1.38		
Goldbach Sum	200000	6.84	2.90	2.36	0.03421	0.01452
Total tid	200000	7.42	3.30	2.25		
EratosthesSil	20000	0.06	0.43	0.13		
Goldbach Sum	20000	0.84	1.73	0.48	0.04187	0.08665
Total tid	20000	0.89	2.04	0.44		
EratosthesSil	2000	0.01	0.36	0.03		
Goldbach Sum	2000	0.35	1.81	0.19	0.17411	0.90483
Total tid	2000	0.36	2.17	0.17		

Kjøretider for alle G(s) < 1 mill + noen eksempler fra 2 maskiner

```
Prog 2,
finn alle summer:

 G(999986) =4385
 G(999988) =4484
 G(999990)=11143
 G(999992) =4858
 G(999994) =4235
 G(999996) =8194
 G(999998) =4206
 G(1000000)=5402

1) Intel i7 M 620 2(4) kjerner @ 2.67 GHz - Java 7
                  n       sekv(ms)   para(ms)     Speedup   SekElem(us)   ParElem(us)

EratosthesSil    1000000     13.33       4.85      2.75
Goldbach Summer  1000000 493772.44 274827.14      1.80    493.77244   274.82714
Total tid        1000000 493785.77 274831.99      1.80


2) Intel i7 870 4(8) kjerner @ 2.93 GHz - Java 8
                  n       sekv(ms)   para(ms)     Speedup   SekElem(us)   ParElem(us)

EratosthesSil    1000000     14.61       2.80      5.21
Goldbach Summmer 1000000 500381.38 152585.06      3.28    500.38138   152.58506
Total tid        1000000 500395.99 152587.87      3.28
```

Hva så vi på i Uke13

I) Om vindus- (GUI) programmering i Java
- Tråder ! Uten at vi vet om det !

II) Hvordan pakke inn en parallell algoritme slik at andre kan bruke den.
- Hvordan gi din Oblig4 til en venn eller en ny jobb som ikke har peiling på parallellitet og si at her er en metode som sorterer 3-10x fortere enn Arrays.sort() ?
- Brukervennlig innpakking (static eller objektmetode) !
- Nødvendige endringer til algoritmene, effektivitet !
- Fordelinger av tall vi skal sortere.
- Fornuftig avslutning av programmet ditt (hvordan avslutte trådene)

III) Litt om løsning på prog1 og prog2 – finn én sum for alle n < 2 mrd. og alle (antall) Goldbach-summer for alle partall < 1 . mill. sekvensielt og parallelt.