IN3030/IN4330 Uke 14, v2021

Arne Maus
PT
Inst. for informatikk

Hva så vi på i Uke 12

- Om å lage en egen 'ArrayList' for heltall
  - Hvorfor og hvordan (kan lage arrayer av IntList)
  - For Oblig 5
- (En første) gjennomgang av Oblig 5 Konveks innhyllning

Hva skal vi se på i denne uka

I) Om vindus- (GUI) programmering i Java

- Tråder ! Uten at vi vet om det !

II) Parallellisering av Oblig 5 – den konvekse innhyllinga til n punkter ved først å se på full parallell Quicksort

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

- Kan du gi din Oblig 4 til en venn eller en ny jobb som ikke har peiling på parallellitet og si at her er en metode som sorterer opp til 8x 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

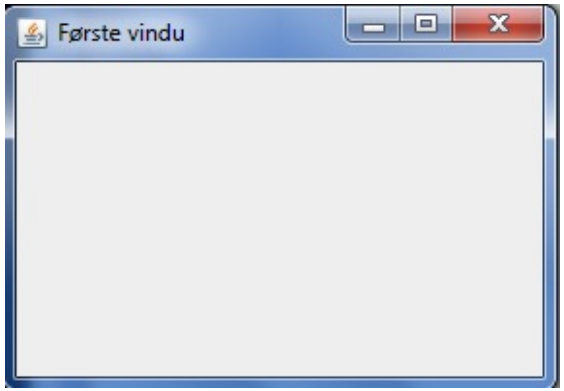
Tråder og GUI

- (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) – nå også FX
- 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 run
                       } );// end Runnable

                      System.out.println("4) a="+ a );
                  } // end main
   }; // end Swing
```

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

- Vinduet som kommer opp (2017 og 2021- modell)



- 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 parallellisere Oblig5 – den konvekse innhyllinga

- Gjøre det som vanlig Quicksort

- Gjøre det som full parallell Quicksort
  – artikkel på NIK 2015
  - En algoritme som ikke har en meningsfull sekvensiell versjon.
  - Trøst: Full Parallell CoHull er enklere enn Full Parallell Quicksort

A FULL PARALLEL QUICKSORT ALGORITHM FOR MULTICORE PROCESSORS

Arne Maus,
Dept. of Informatics,
Univ. of Oslo

Outline of presentation

- Presenting good old sequential Quicksort: seqQuick
- Traditional parallel Quicksort: TradQuick
  - What is not optimal with TradQuick and other attempts to do parallel Quicksort?
- ParaQuick, an alternative new **full parallel** Quicksort explained
- Performance of: Arrays.sort, seqQuick, TradQuick and ParaQuick
- Three questions on why ParaQuick is (much) faster than TradQuick:
- Conclusion

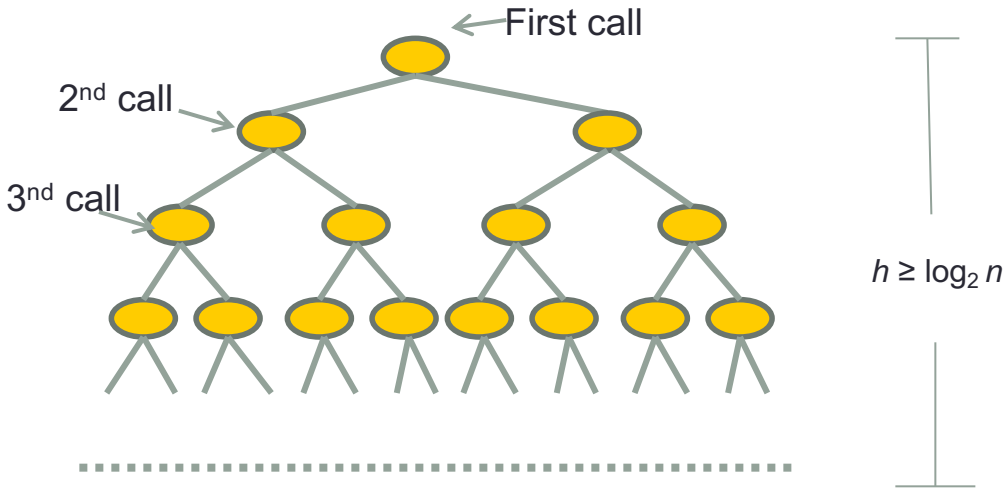
Note: Performance figures updated from paper

Everybody knows *sequential* Quicksort of an array: *a*[0..n-1]?

1. Start by selecting a partitioning element: *pivot*
2. Partition *a*[] in two:
   a. Those elements in a[] that are < *pivot* are placed in the left part of *a*[], while those who are ≥ *pivot* are placed in the right part of a[].
   b. This is achieved by swapping elements < *pivot* with elements ≥ *pivot.*
3. Pts. 1 and 2 are repeated recursively on each of the two parts until all such segments are of length ≤ 1.

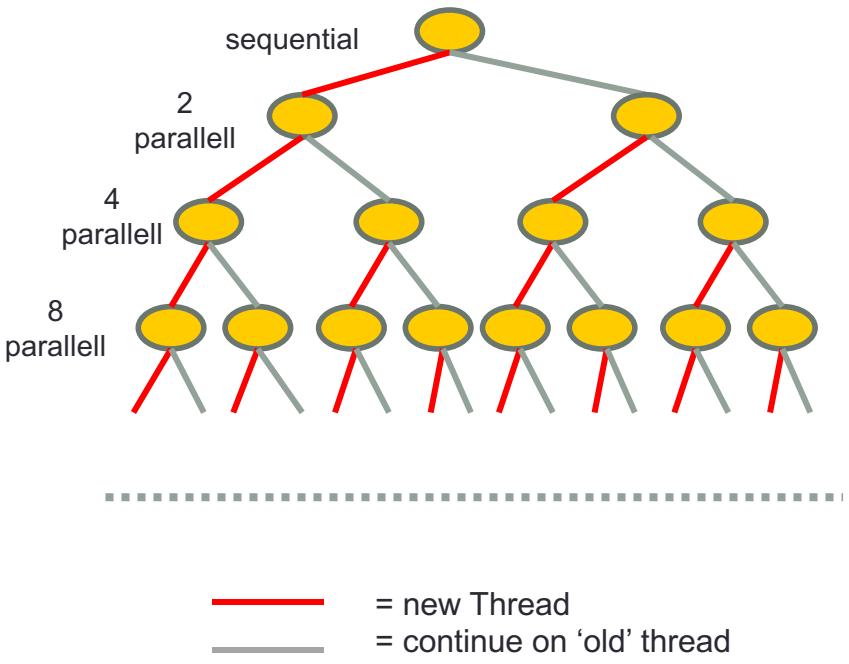
   (Or use 'insertionSort' if len < 47)

Recursion tree of method calls to Quicksort



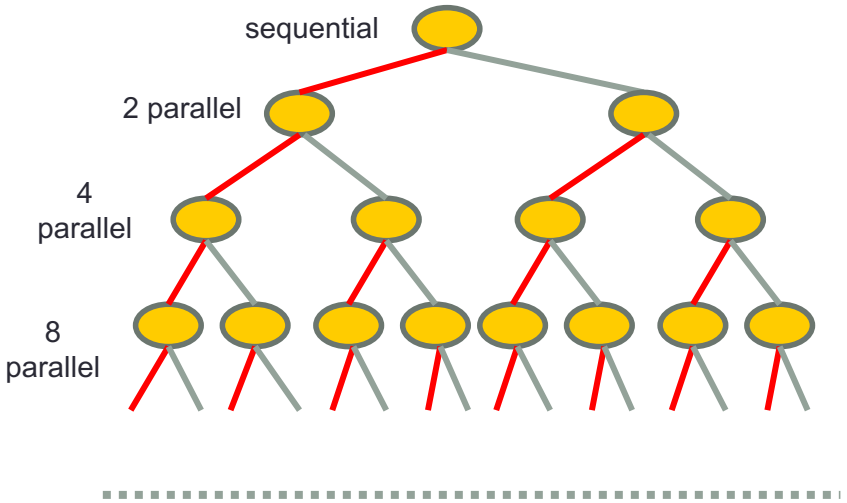
How to parallelize Quicksort (traditionally & 'smart')

1. Select an element: *pivot* in a[ ].
2. In each node:
   a. Partition in small elements to the left and large elements to the right.
   b. Start **one new thread** on the left branch (the 'small' numbers < pivot)
   c. Then call recursively on the 'old' thread' that entered this node, for the right part, 'big' numbers ≥ *pivot*
   d. Wait for the **new left thread to return**.
3. In any branch where the length of the part to be sorted < 50 000, use ordinary sequential, recursive Quicksort.



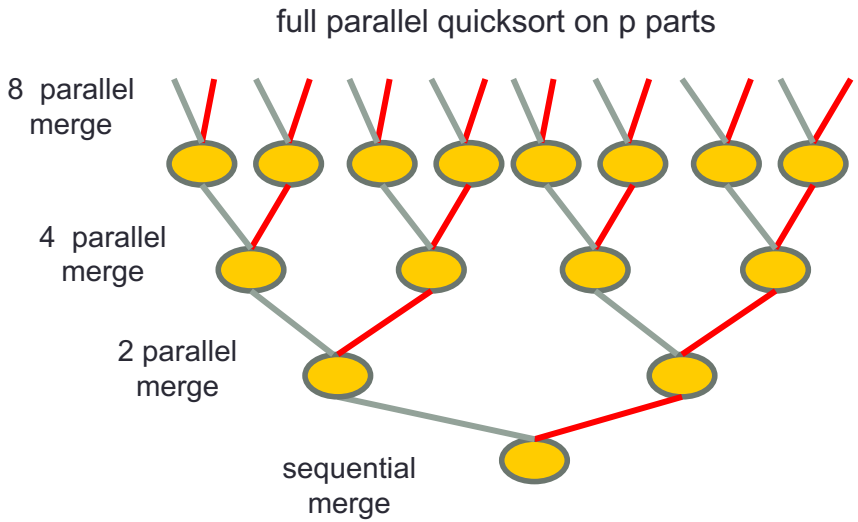
What is wrong with traditional parallelization ?

- **SLOW START** :
  - First sequential
  - Then next level: 2 parallel
  - Then : 4 parallel
  - …
- Can this be done in full parallel - i.e. with p cores at least with p parallel from the start?



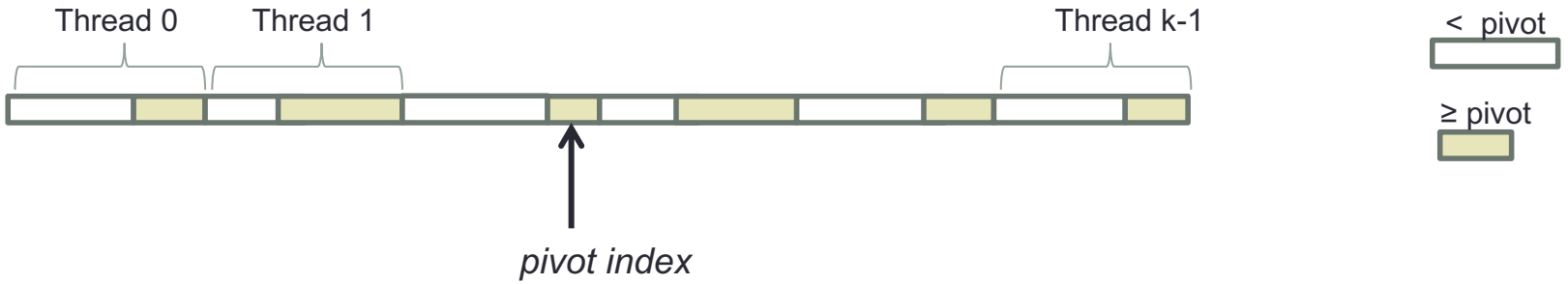
- **Yes**, some have tried (bitonic sort,.) with starting with quicksort in *p* parallel parts, and then merge sort the resulting sorted parts together.
- Resulting in a quick start, and a **slow** ending.

End result : Same, slow result

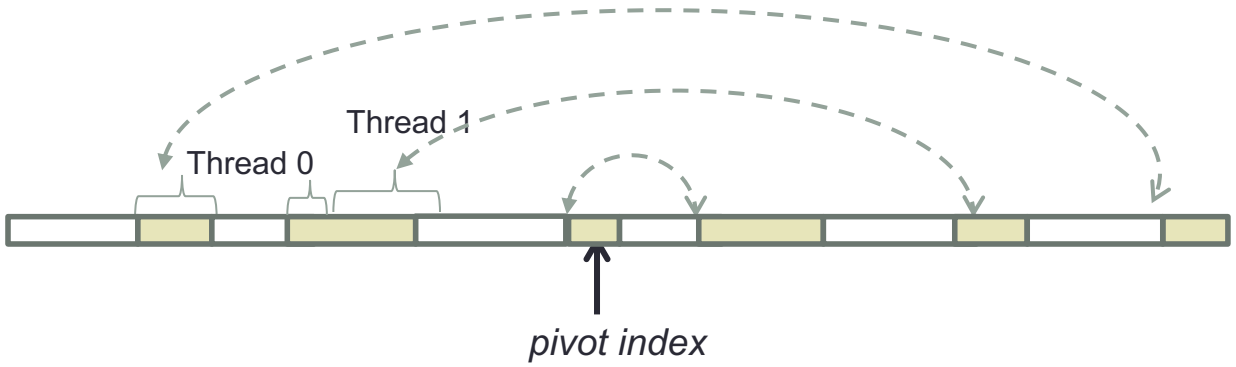


A new idea for **full** parallelization

- With p cores, start *k* = *m*p* threads (*m* =4,8,…64?)
- First select a common *pivot* value for all threads and then **partition** (not sort) **in parallel** the k parts of a[] with that *pivot*.
- Let *pivot indeks* = $\sum_0^{k-1} length\ of\ all\ small\ elements$

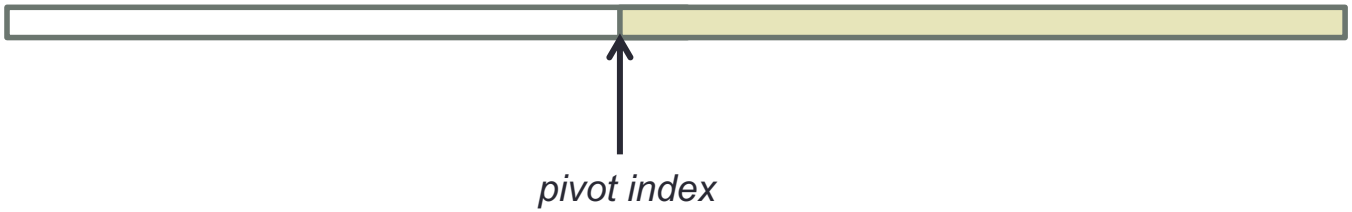


- Then swap all displaced 'big' elements to the left of *pivot index* with displaced 'small' element to the right.
- This can be done in parallel without any data race (no synchronization needed)



Full parallel Quicksort idea (cont.)

- After parallel swap, we have the ordinary quicksort partition



- We can then continue recursively (or iteratively) on the left part and right part with half of the threads for each part.
- Continue with this in parallel until only one thread left in a partition, then call sequential quicksort for that part.

- **NOTE:** The price we pay for starting full parallel is this extra parallel swap + more synchronization.

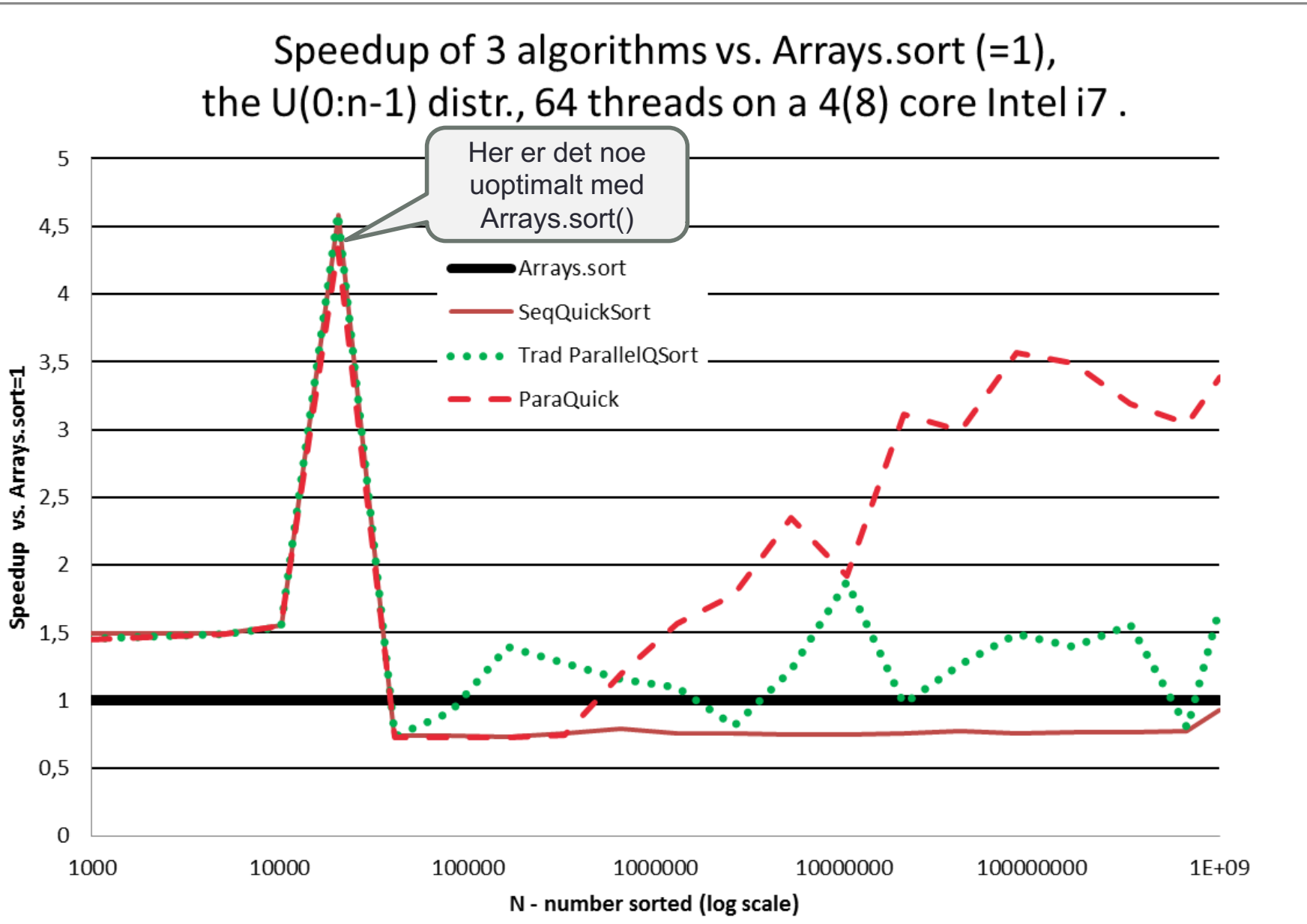
Performance of a full parallel Quicksort

- In Java there is an library sorting method: Arrays.sort – an advanced sequential Quicksort (1200 Lines of Code) with *dual pivot*.
- We want to compare runtime efficiency of alternative algorithms with speedup compared with Arrays.sort:

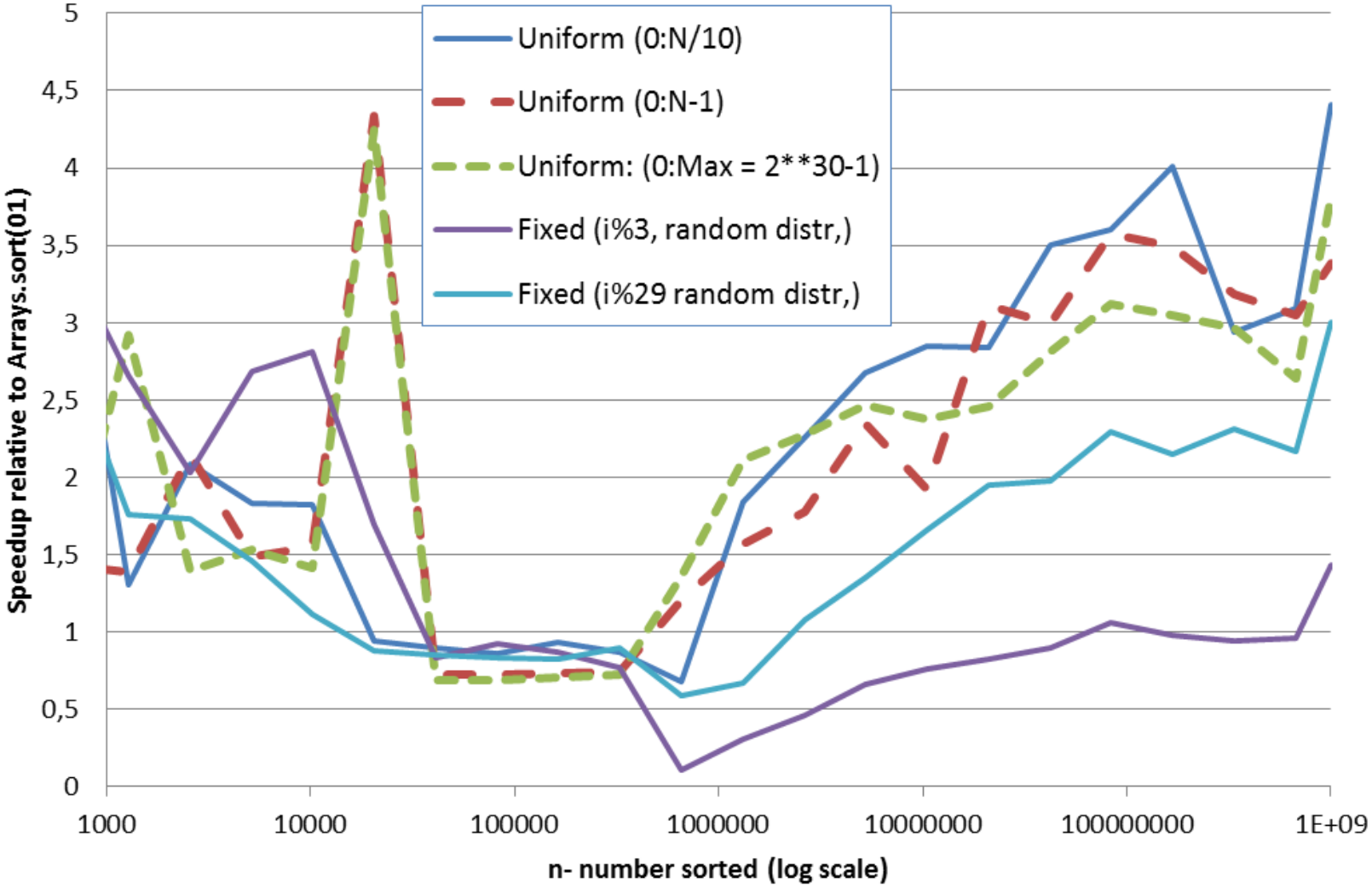
$$S = \frac{Execution\ time\ Arrays.sort}{Execution\ time\ alternative\ algorithm}$$, $S > 1$ means faster alternative

- Alternative algorithms tested:
  1. My own sequential Quicksort (which is some 20% slower than Arrays.sort())
  2. Traditional parallel Quicksort
  3. ParaQuick – the new full parallel quicksort

- All algorithms tested with:
  1. $n = 100, \ldots, 10^9$
  2. Distribution of numbers to be sorted:
     a) Uniform (0:*n*/10)
     b) Uniform (0:*n*-1)
     c) Uniform (0: max= 2^{30} -1)
     d) Uniform(i%3) – 3 possible values
     e) Uniform (i%29) – 29 posible values


Speedup of 3 algorithms vs. Arrays.sort (=1),
the U(0:n-1) distr., 64 threads on a 4(8) core Intel i7 .
Her er det noe uoptimalt med Arrays.sort()
Arrays.sort
SeqQuickSort
Trad ParallelQSort
ParaQuick
Speedup vs. Arrays.sort=1
5
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
1000
10000
100000
1000000
10000000
100000000
1E+09
N - number sorted (log scale)

Speedup of ParaQuick vs. Arrays.sort on a 4(8) core* Intel i7 machine, 64 threads, 5 distributions



3 questions:

1. Why is ParaQuick much faster than TradQuick?

2. Why, by using 8x*k* threads (*k* = # cores), makes ParaQuick (much) faster than only using *k* threads?

3. Why is ParaQuick only fast for $n > ½$ mill?

Then : Data

1. Why is ParaQuick so much faster than TradQuick?

- As explained earlier, fast start – 'removes' top 6 levels of recursion tree.
- We start with say 64 'small' problems + extra parallel swaping
- Smaller problems fits better into the caching system, and hence faster ; the 'single' big problem gets more cache misses to main memory.
- Sequential programs have only a single level1 and level2 cache compared with four level1 and four level2 caches for parallel.
- $\Rightarrow$ the 6 top levels of the recursion tree are really slow.

3 data cache levels detected by latency.exe; 2017 Workstation

Level 1	size = 32Kb	latency = 4 cycles
Level 2	size = 256Kb	latency = 12 cycles
Level 3	size = 8192Kb	latency = 36 cycles
Main mem.	size= 8Gb	latency = 171 cycles

2. When using 8•k threads, what makes ParaQuick go faster

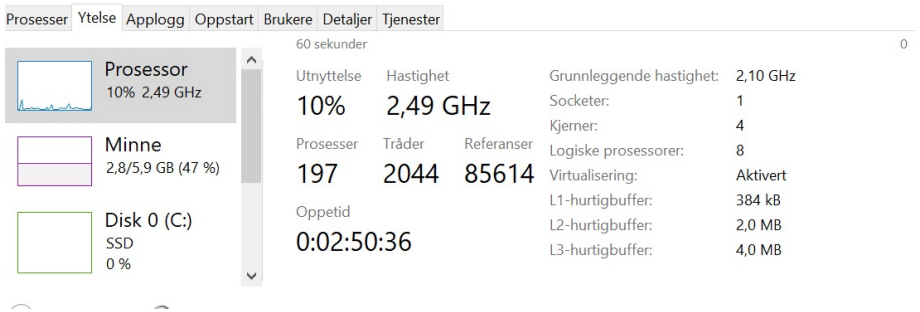
- We have *k* cores, but use *p* =*m***k* threads (*m* =4, 8,16,..,64)
- Why is that faster ?

If we have an O(n *log* n) process and divide it into 64 smaller sub problems, the running time *t* of that is sequentially:

$$t = \sum_1^{64} \frac{n}{64} * \log\left(\frac{n}{64}\right) = n * \log\left(\frac{n}{64}\right) = nlogn - nlog64 = nlogn - n * 6$$

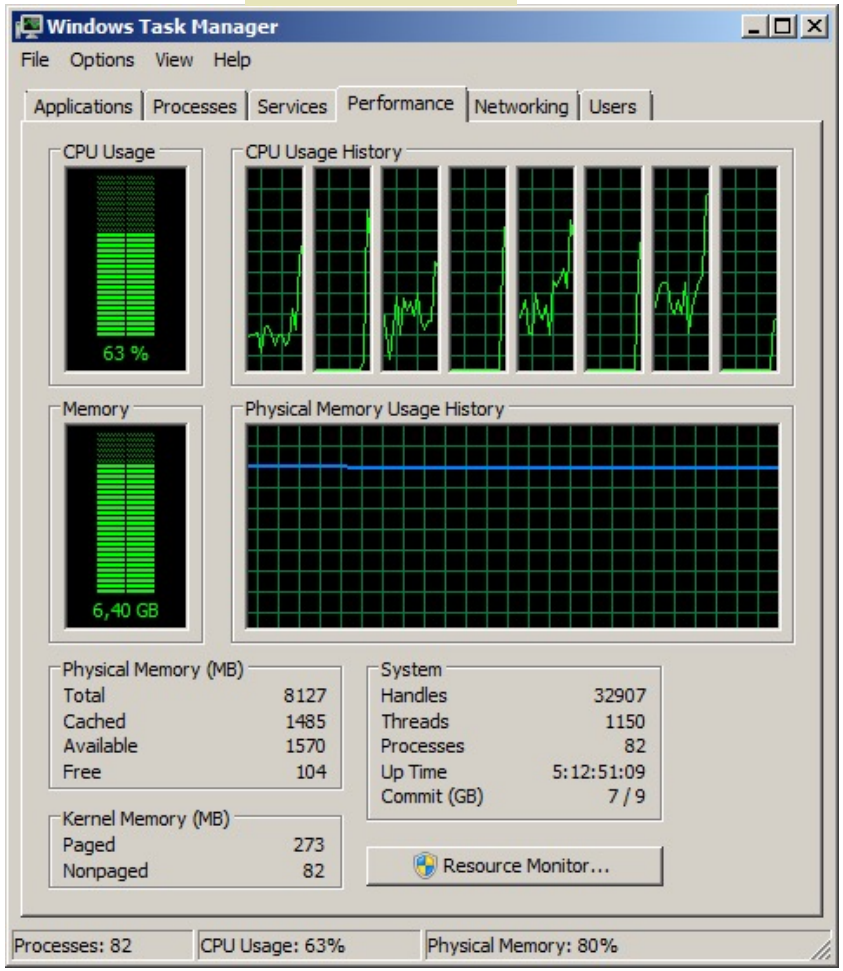
- The time for the extra swaps is less than 6*n,* hence a speedup with 8 cores
  - Swapping at each of the 6 levels affects *n*/2 elements in parallel each time – timewise that is 6* n/(2*8)= 6n/16 – seems to be 16x faster.
- When we have 8 cores, then with 64 threads we have a queue of 8 threads on each core – Better utilization of the cores on cache miss – see next foil.

Det kjører mange tråder på maskinen allerede.
Windows er basert på prosesser, tråder og handles (referanser)

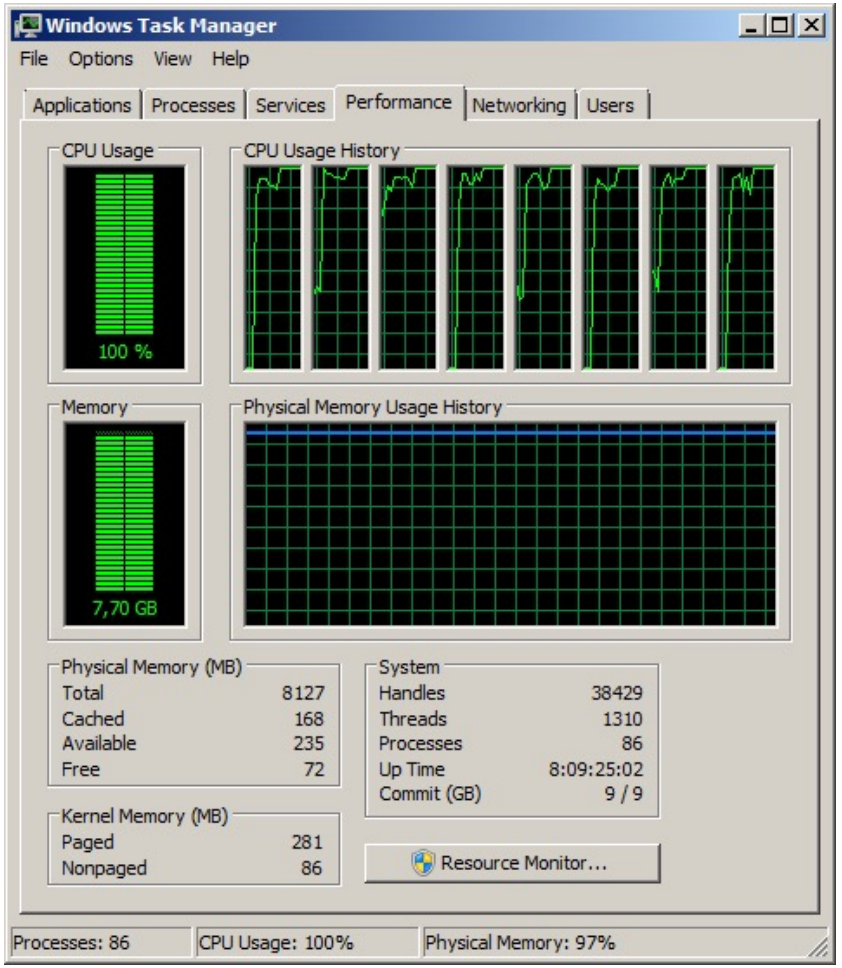


Different **max** CPU utilization with 8 and 64 threads for QuickPara, U(n-1) distribution, $n=10^9$

8 threads



64 threads



3. Why is ParaQuick first really fast for n > ½ mill?

- All 64 threads started (as a threadpool)
  - Outside timing – no effect
- Synchronizing 64 threads 3 times on each of 6 levels (CyclicBarrier)
  - CyclicBarrier: await() : 0,4 msec first time, 0,34 msec. subsequent times
  - On each of the 8 cores we do: (64/8)*3*6 sync, each 0.35msec.
    - All sync in ParaQuick = **50** msec. (timewise in parallel)
  - Arrays.sort() sorts 500 000 integers in **30** msec, and 1 mill. in **74** msec

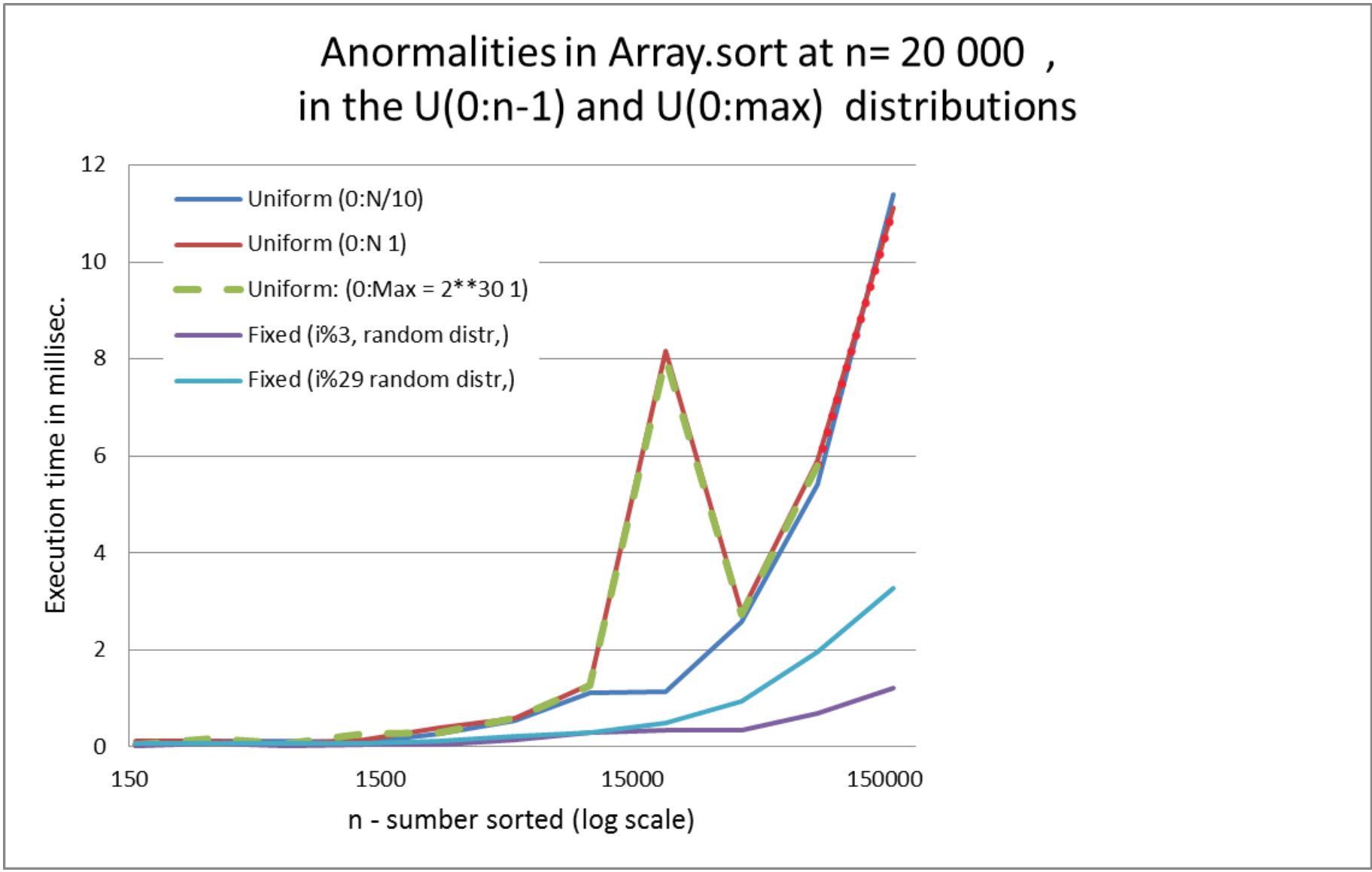
- Note other effects of 'Just In Time compilation':
  - Method calls:
    - First call: 2,8 μsec ; subsequent calls: 0,15- 0,05 μsec,
      3000x faster than CyclicBarrier.await()
  - Create object (new class) + method call
    - First object. 3,5 msec., subsequent objects + method call: 0,13 μsec
      after first object : **300x** faster than CyclicBarrier.await()
  - new Thread start and stop (join())
    - First thread 12,1 msec., subsequent Threads start/stop: 0,21 msec. each
    - Much slower (**40x**) first time – after that almost same time as a CyclicBarreir await()

Conclusion

- I have presented a new way of parallelising Quicksort that is significantly faster than other known attempts to parallelize Quicksort when $n > ½$ million.
- For large n > 10 mill, it is 3-4 times faster than Arrays.sort and often twice as fast (or better) than other parallel Quicksort algorithms.

- Some improvements already made:
  - Using (far) more threads than cores – thanks to a referee !
  - Only stopping in QuickPara when there is only one thread left in a segment

- Work in progress, further improvements/experiments will be made:
  - Better sequential quicksort for QuickPara investigated (dual pivot)
  - Scaling number of threads also to n, not only to number of cores.
  - Different values for *starting* parallel (both TradPara and QuickPara) and *stopping* parallel(TradPara)
  - ……

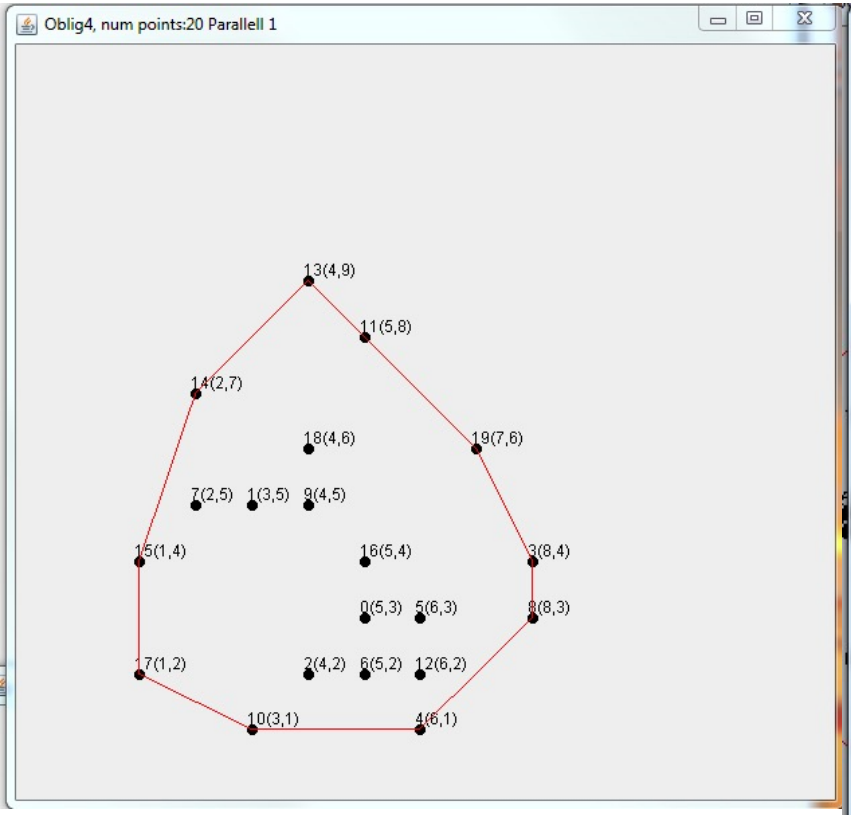

Anormalities in Array.sort at n= 20 000 ,
in the U(0:n-1) and U(0:max) distributions
Uniform (0:N/10)
Uniform (0:N 1)
Uniform: (0:Max = 2**30 1)
Fixed (i%3, random distr,)
Fixed (i%29 random distr,)
Execution time in millisec.
0
2
4
6
8
10
12
150
1500
15000
150000
n - sumber sorted (log scale)

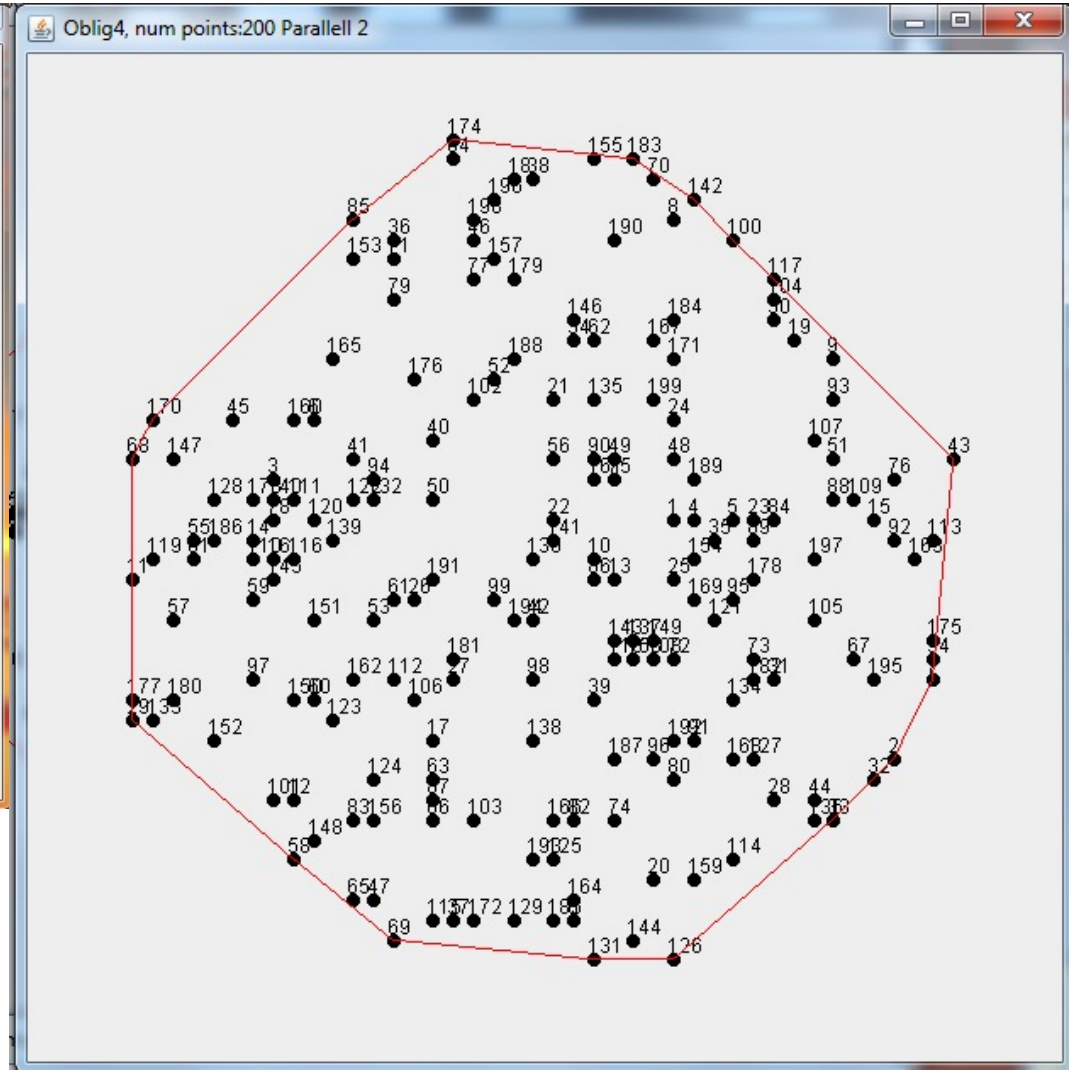
II) Hvordan parallellisere Oblig 5 – den konvekse innhyllinga

- Gjøre det som vanlig Quicksort – parallellisér rekursiv sekvensiell løsning.

- Gjøre det som full parallell Quicksort
  – artikkel på NIK 2015
  - En algoritme som ikke har en meningsfull sekvensiell versjon.
  - Trøst: Full Parallell CoHull er enklere enn Full Parallell Quicksort

To sett av løsninger, n = 20 og n=200





Oblig5 – parallelliseringen av konveks innhylling

- Antar at du har to riktige metoder:
  - a. IntList Sekv(int low, int high, IntList alle) – sekvensiell løsning
  - b. IntList RekPara(...) - rekursiv parallellisering av Sekv(..)
- 5 mulige parallelliseringer av oblig 5:

1) Del punktene i mange deler (#tråder k = f.eks 5* antall kjerner)
   - Kjør Sekv på hver av disse delene –
     du har da k 'små' konvekse innhyllinger over samme område.
     - Du kan lett vise at alle punktene på den 'store' innhyllinga også er et punkt på en av de 'små' innhyllingene.
2) Problem – hvordan slå disse k 'små'-innhyllingene sammen til den ene store vi ønsker (de små har i sum ca. 10-15% flere punkter enn den store):

Parallellisering av Oblig5-forts

- Husk at sammenstillingen av n slike 'små'-innhyllinger til den ene, store innhyllinga har kjøretid $nlogn$, nesten alle punktene i de små er på den store innhylinga (når n er stor)
- Dvs. Du har ca. 16 000 pkt, men logn(16 000) ≈14, dvs. Problemet har tar tilsvarende problem med 16 000*14 uniformt fordelte punkter = 224 000 punkter – eller ca. 10 ms.
  1. Slå sammen kohyll til de små og kjør Sekv på denne mengden - *
  2. Slå sammen de små og kjør RekPara på denne mengden *
  3. Flett sammen to små til en litt større hvor du også har fjernet overflødige punkter (vanskelig) – gjentatt til alle er slått sammen. *
  4. Slå alle puktene sammen og sortér mengden (med Radix) på y og så på x-verdier for hvert av de 4 kvadrantene. Fjern så ekstra indre punkter.
  5. Flette-sorter sammen to og to slike innhyllinger til alle er flettet sammen. Fjern så ekstra punkter. (Du kan flette-sortere fordi hver av de små er jo 'perfekt' sortert)
  - .....?
- Fjern indre punkter = se på tre fortløpende punkter (sortert) , og hvis p2 ligger til venstre (på innsida) for linje p1-p3 , fjern p2.

* - er implementert og testet

Speedup Oblig 5 konv. innhyll – forelesers løsning fra 2015 – som parallellisering av vanlig Quicksort

Test av Parallell og sekvensiell konveks innhylling av n punkter med 8 kjerner , og 8 traader,
 Median av:3 iterasjoner

n	sekv.tid(ms)	para.tid(ms)	Speedup
200 000 000	10213.669	6074.485	1.6814
20 000 000	792.634	452.406	1.7520
2 000 000	71.935	111.246	0.6466
200 000	7.648	37.369	0.2047
20 000	*0.999*	*11.629*	*0.0859*
2 000	*0.115*	*3.201*	*0.0360*
200	*0.016*	*1.223*	*0.0130*
20	0.002	0.266	0.0090

Kommentar: Ikke veldig speedup – 1.75 på 8 kjerner
Kan klart gjøres bedre.

Speedup Oblig 5 konv. innhyll – forelesers løsning fra 2017 – full parallellisering av Konveks hull.

```
Test av  parallell og sekvensiell: Den konvekse innhyllinga av n
punkter med 8 kjerner , og 16 traader,  Median av:3 iterasjoner
```

n	sekv.tid(ms)	para.tid(ms)	Speedup	KoHyll.size()
200000000	8941.138	3099.202	2.8850	18896
20000000	772.244	348.533	2.2157	6011
2000000	69.031	36.129	1.9107	1905
200000	6.782	6.726	1.0084	592
20000	0.638	3.688	0.1730	165
2000	0.098	2.192	0.0445	68
200	0.012	1.838	0.0064	20
20	0.005	1.384	0.0035	7

Kommentar: En god del raskere, nær dobbelt så rask parallelt, men speedup kunne vært noe bedre (2.88). MERK 16 tråder.

Fast antall nivåer med tråder i rekursjonstreet (etter antall kjerner):

```
Test av  to parallelle og en sekvensiell: Den konvekse innhyllinga av n
punktermed 8 kjerner , og 32 traader,  Median av:3 iterasjoner


    n        sekv(ms)   p1(ms)    Spdup1   # koHyll
200000000   8091.500   8606.116   0.9402    16626
 20000000    764.615    829.141   0.9222     5196
  2000000     70.678     76.739   0.9210     1672
   200000      6.556     10.110   0.6485      528
    20000      0.629      1.966   0.3200      149
     2000      0.080      1.649   0.0486       59
      200      0.010      1.487   0.0065       20
       20      0.002      3.805   0.0005       10
```

Nye tråder etter størrelsen på søkemengden (> 100 000)

```
Test av  to parallelle og en sekvensiell: Den konvekse innhyllinga av n
punkter med 8 kjerner , og 32 traader,  Median av:3 iterasjoner

    n        sekv(ms)   p1(ms)    Spdup1    # koHyll
200000000   8191.384   6197.223   1.3218     16626
 20000000    798.683    642.539   1.2430      5196
  2000000     71.981     66.232   1.0868      1672
   200000      6.755      8.870   0.7616       528
    20000      0.638      3.014   0.2118       149
     2000      0.082      1.394   0.0591        59
      200      0.011      1.304   0.0085        20
       20      0.002      2.067   0.0012        10
```

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 <len
        } 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()’ , men B synes noe bedre.
- 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 etter sorteringa avsluttes.

(Vi skal nå 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
100 000 000    12042.708         3128.675         3.8491
 10 000 000     1090.252          277.264         3.9322
  1 000 000       92.958           32.640         2.8480
    100 000        7.682            5.198         1.4777
     10 000        0.616            0.737         0.8356
      1 000        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 paraQuick - 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)
- 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 meget enkle(re) å sortere:
  - Anta at a[] er sortert stigende ($a[i] \leq a[i+1] \ \forall \ i < n-2$)
  - Anta at a[] er synkende sortert ($a[i] \geq a[i+1] \ \forall \ i < n-2$)
  - 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); // 16-50
        else if( n < PARA_LIMIT) sekvensiellQuicksort(a);  // 50 000
        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 ?

- NEI: Fail-fast – dvs hvis arrayene ikke er forlengs eller baklengs sortert feiler disse testene etter 2-4 tester.

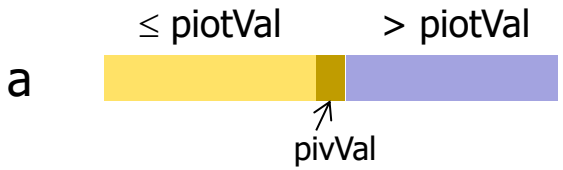
Hvordan sikre oss mot mange like elementer med dual pivot (to pivot-indekser)?

```
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
```

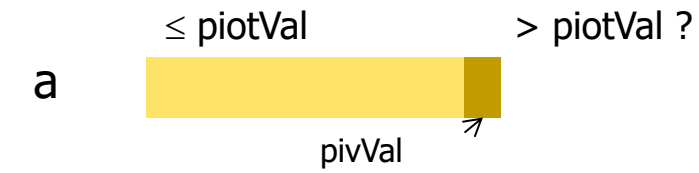
```
// Hvorfor ikke denne – kortere kode ??
void quicksortSek2(int[] a, int left, int right) {
    if (right-left < LIMIT) insertSort (a,left,right);
    else {  int piv = partition (a, left, right); // del i to

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

Etter: partition(a,left,right)



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
 }
```

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

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

n	sekv.tid(ms)	para.tid(ms)	Speedup
20 000 000	216.149	203.474	1.0623
2 000 000	17.205	22.083	0.7791
200 000	1.658	6.363	0.2605
20 000	0.175	0.177	0.9881
2 000	0.020	0.019	1.0182
200	0.007	0.009	0.7600

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

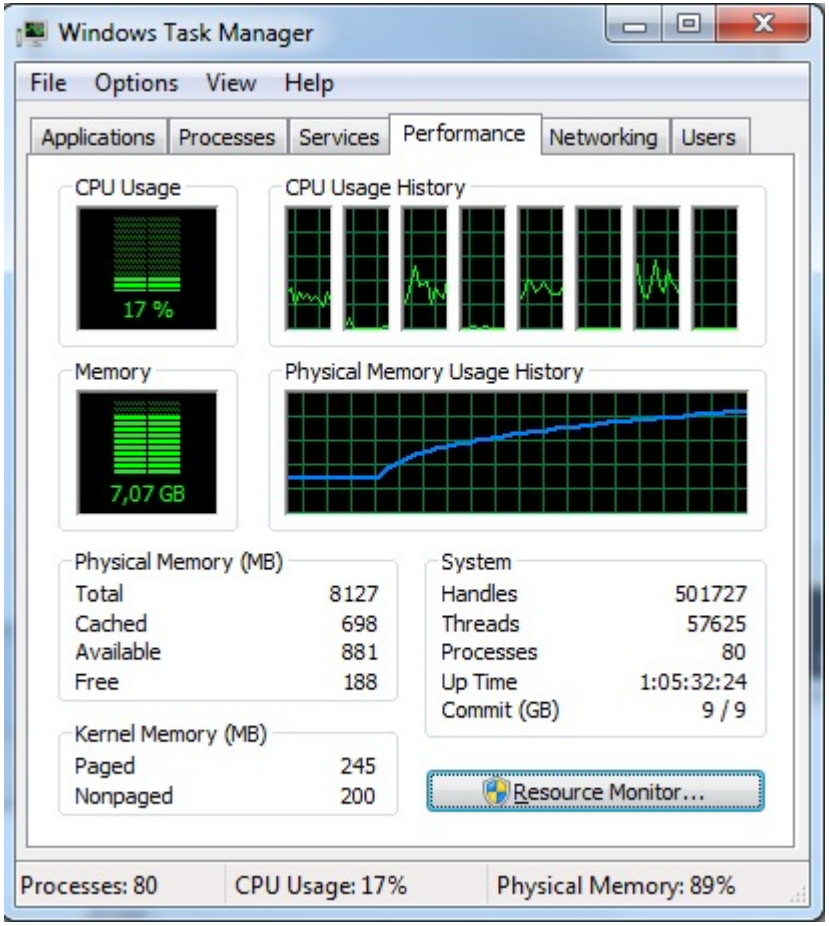
```
// fyll a[] med 0,1,2,…,9, 0,1  både sekvensiell og para
for (int i =0; i<n; i++) a[i] = i%10;
// Største n =20 000 det virket for:
```

n	sekv.tid(ms)	para.tid(ms)	Speedup
20 000	61.692	60.320	1.0227
2 000	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)
```

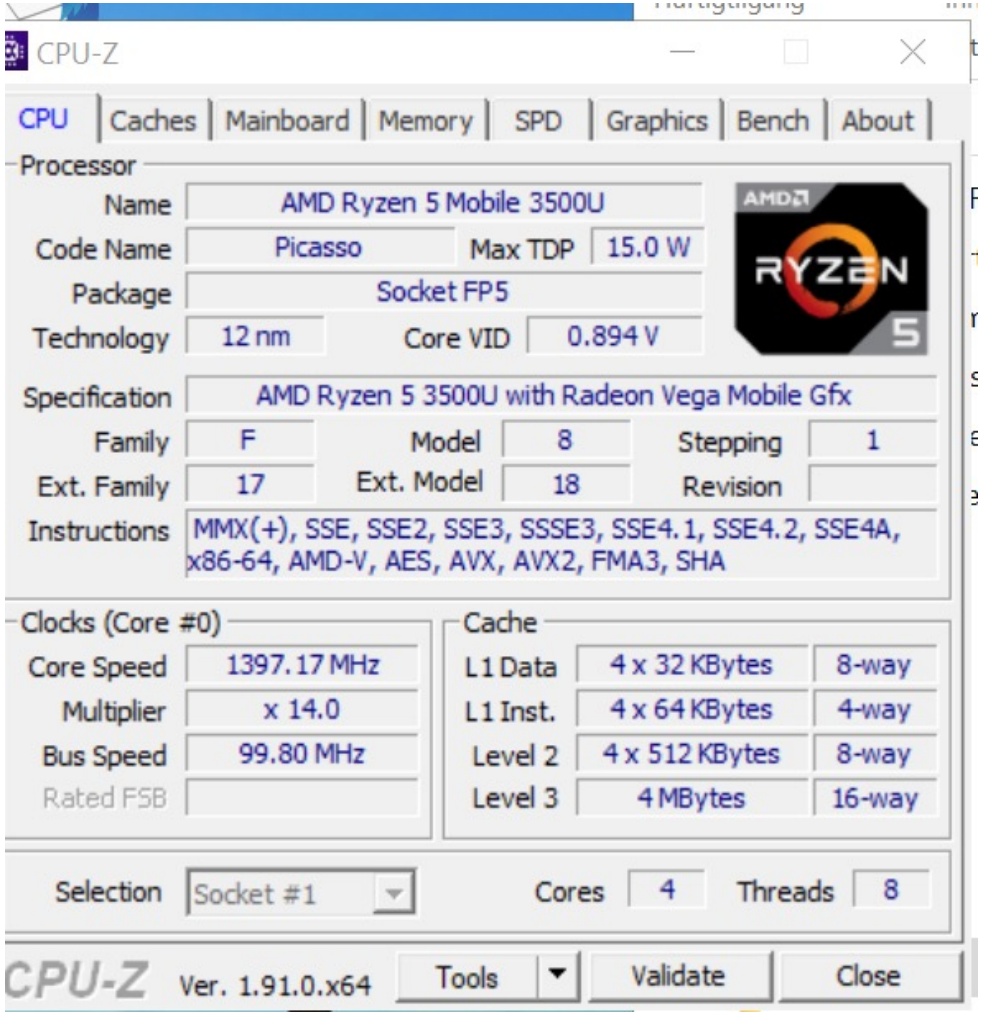
Forsøk4 : n= 200 000; 0,1,2..,9 for både parallell og sekvensiell:
Begge uten while-løkke og piv2 : Stack overflow

```
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)
```

Teste maskiner med program-vare fra CPUz

Går CPU-er raskere i 2021 enn i 2005.
Svar: Grovt sett nei , men mer parallelle instruksjoner (SSE 1-4,..).

Min nåværende PC (Azus ZenBook) AMD Ryzen 5 3500U har 4 kjerner/8 tråder og har en hastighet (variabelt≈ 2,63 -3,xx GHz med 8 Gb DDR4 RAM)



Fra 'latency.exe':

3 cache levels detected -2021 + raskere cach

Level 1	size = 4x32Kb	latency =	2 cycles
Level 2	size = 4x512Kb	latency =	7 cycles
Level 3	size = 2x2048Kb	latency =	15 cycles
Memory:	size = 8 Gb	Latency	= 200 cycles

Hva så vi på i Uke14

I) Om vindus- (GUI) programmering i Java

- Tråder ! Uten at vi vet om det !

II) Parallellisering av Oblig 5 – den konvekse innhyllinga til n punkter ved først å se på full parallell Quicksort. Mer neste forelesning

III) 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 ?