INF2220 - 2. NOV. 2017

PARALLELL SORTERING

Arne Maus,

PSE, Ifi

Dagens forelesning

- Hva er et parallelt program med tråder (i Java)
  - Typer av parallelle programmer – her vil vi ha raskere programmer
- Hva er en tråd?
  - Starte og stoppe (=vente på ferdige tråder)
- Felles adresserom – hvordan få det til og hva kan gå feil?
  - Lesing og skriving på felles variabler
- Synkronisering av tråder
  - Hvorfor og hvordan ?
- Hva tar tid i parallell programmer
- Hvordan parallellisere rekursive programmer
- Parallell KvikkSort – to elendige og en effektiv løsning

1) Parallelle algoritmer - hvorfor

- Vi vil ha *raskere* algoritmer, også sortering !
- Vi har nå flere prosessorkjerner, burde bruke alle disse.
- N.B. **Amdahls lov** :
  - Tenk deg at algoritmen din har en del på **p**% (eks. 20%) av kjøretiden som må kjøres ***sekvensielt*** og resten kan gjøres i ***parallell***.

Amdahls lov :

- Med **p% sekvensiell kode** er det raskeste du kan forbedre din algoritmen din er **100/p** ganger (eks. 100/20 =5).
- Uansett hvordan du parallelliserer og uansett hvor mange maskiner du har, fordi den parallelle koden kan aldri gå raskere enn på 0,000.. sekunder
- Den sekvensielle delen vil derfor alltid gi deg en begrensning, og *må gjøres minst* mulig.
- Med k prosessorer er mulig Speedup (= ganger raskere), S:

$$S \leq \frac{100}{p + \frac{100 - p}{k}}, \quad S \leq 3.33 \; med \;\; p = 20, k = 8$$

- Dette er ikke nødvendigvis sannheten. **Gustafson** (med sin lov) hevder at den sekvensielle %-andelen ofte synker når problemet blir større. Amdahl er ikke fasit sier Gustafson.

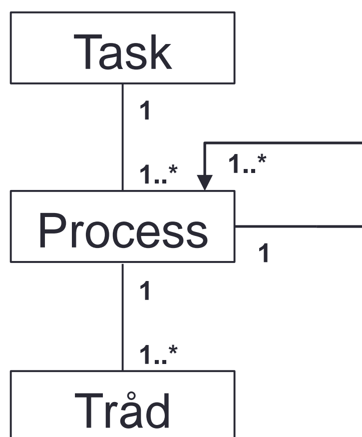
Hva er tråder i Java ?

- En tråd er et sekvensielt program
- I alle programmer kjører minst én tråd – main tråden (starter og kjører public static void main).
- Main-tråden kan starte en eller flere andre, nye tråder.
  - Hver av de nye trådene inneholder metoden **public void run() {}** - som tilsvarer main() for main-tråden
- Enhver tråd som er startet, kan stoppes midlertidig eller permanent av:
  - Av seg selv ved kall på synkroniseringsobjekter hvor den må vente
  - Den er ferdig med sin kode (i metoden run), terminerer da
- Main-tråden og de nye trådene går i parallell ved at:
  - De kjører enten på hver sin kjerne
  - Hvis vi har flere tråder enn kjerner, vil klokka i maskinen sørge for at trådene av og til avbrytes og en annen tråd får kjøretid på kjernen.
- **Alle trådene i et Java-program deler samme adresserom** (= samme plasser i hovedhukommelsen). Alle trådene kan lese og skrive i de variable (objektene) programmet har og ha adgang til samme kode (metodene i klassene).
- Vi bruker tråder til å parallellisere programmene våre

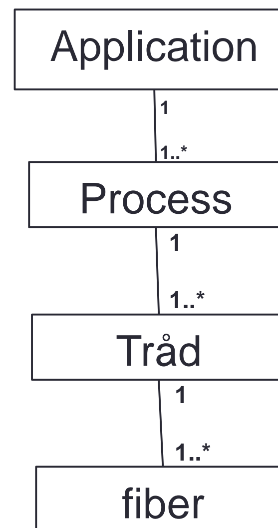
Operativsystemet og tråder

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

MacOS/Linux

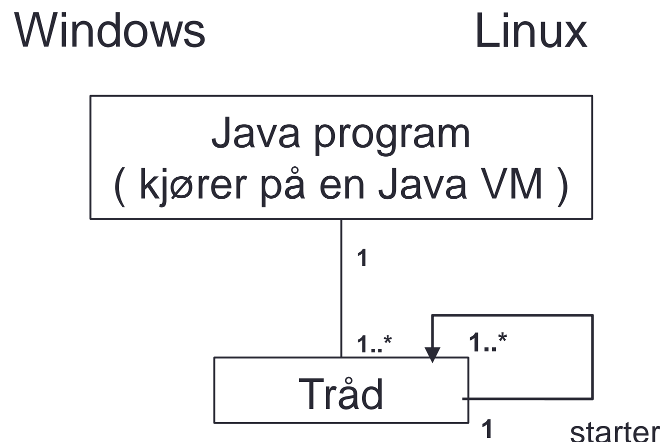


Windows



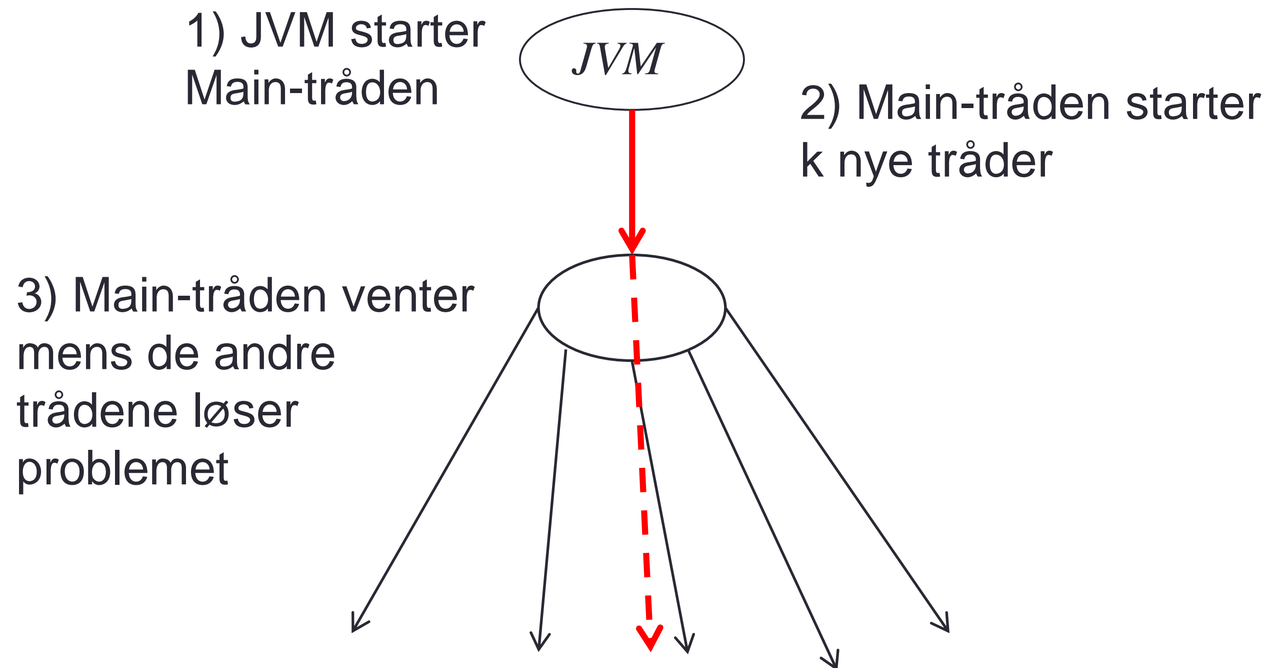
Heldigvis forenkler Java dette

Java forenkler dette ved å velge to nivåer



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

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



Tråder i Java er objekter av klassen Thread.

Hver tråd er et sekvensielt program

2) Av og til må trådene vente på hverandre, synkronisering med CyclicBarrier

- Nesten alltid parallelliserer vi problemer med at data blir delt mellom trådene.
  - Eks. Vi vil finne maksimalverdien i a[] = new int [10 000 000];
  - Vi bruker 10 tråder og lar hver finne sin **max** i hver sine 1mill tall.
- Når trådene har løst sin del av et problemet må de vente på alle trådene før vi kan finne ut hvilken av de 10 stk. **max**-verdiene som er størst.
- Da venter alle trådene på en CyclicBarrier – alle får fortsette når sistemann ankommer.
  - CyclicBarrier bar = new CyclicBarrier (10);
  - < Hver tråd finner max i sin del av a[] ,
    lagrer det i en felles array 10 lang – en plass for hver tråd>
  - Venter slik:

```
try {    bar.await();
 } catch (Exception ex) {return;   }
```

  - <finn så ut hvilken av de 10 max-verdiene som er størst>

Barrier synkronisering

- Hovedfunksjonen til synkroniserings-variable er å :
  - ***Stoppe*** tråder som ikke kan fortsette
  - ***Starte*** igjen stoppede tråder når *en-eller- annen tilstand* har oppstått.
- CyclicBarrier er en klasse i Java API:

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

….

Barrier (= en grind hvor alle tråder venter til alle har ankommet før alle forsetter):

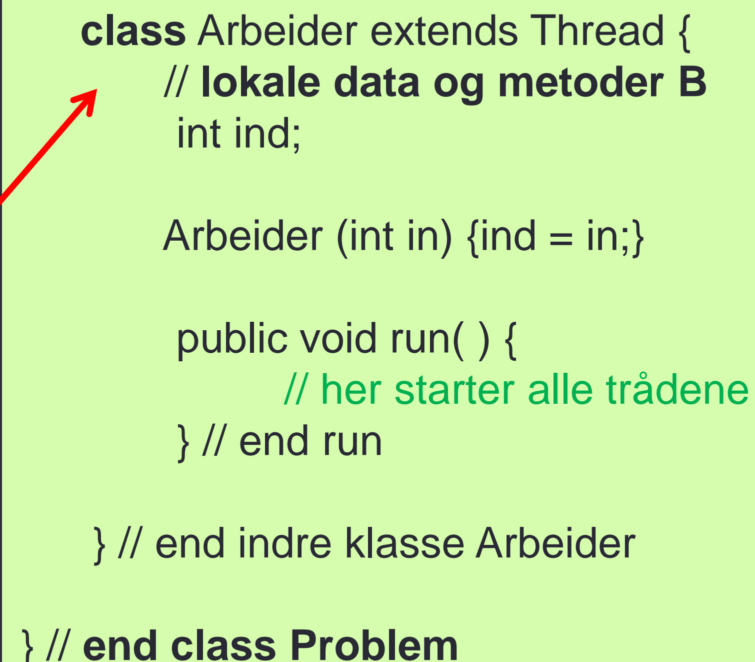
- n tråder kaller på en felles barriere-variabel og de n-1 første må vente. Først når *alle trådene* har kalt variabelen, fortsetter de
  - Kallet: barrier.await();
- Nyttes når alle har har utført en del-beregning. Så må alle stoppes og først startes når alle er ferdige med del-beregningen. (de må "komme i takt")
- Når alle da startes, kan de lese hva alle de andre har beregnet *før* synkroniseringen.
  - `barrier = new CyclicBarrier(numThreads);`
- **`Cyclic`** betyr at etter første vente-situasjonen, kan en slik barriere nyttes omigjen for en **ny** venting med alle trådene.

3) Av og til må trådene sperre alle andre tråder ute fra å lese/skrive på samme variabel som den selv

- Bruker da synkronisering med synchronized metoder
- Eks: **synchronized void økTall(){ tall++;}**
- **økTall** er en metode inne i et objekt. Når en av trådene bruker den, blir det sjekket at:
  - ikke en annen tråd holder på med denne metoden, Hvis det, må den 'nye' tråden vente på at den 'gamle tråden' er ferdig.
  - Når vår tråd får lov til å bruke metoden, sperrer den på samme måte senere tråder fra samtidig å utføre koden i metoden – de må vente
- Ved synkronisering (CyclicBarrier eller synchronized) skrives alle felles variable ned fra cachene og til hovedlageret (slik at alle tråder ser samme verdier)
- En slik synkronisering gjelder bare hvis alle trådene bruker ***samme*** metode i ***samme*** objekt.

4) Hvordan får de felles adresserom – mønster Trådene blir objekter av en **indre klasse** Arbeider

```
class Problem {
   //  felles data og metoder A
   int antTr =  Runtime.getRuntime().availableProcessors();


    public static void main(String [] args) {
        Problem p = new Problem();
        p.utfoer(antTr);
        System.out.println(" Main-tråden  FERDIG");
    } // end main

    void utfoer (int antT) {
          Thread [] t = new Thread [antT];
          for (int i =0; i< antT; i++)
              ( t[i] = new new Arbeider(i)).start();

          try{
             for (int i =0; i< antT; i++) t[i].join();
          } catch(Exception e) {}
    } // end utfoer
```



```
    class Arbeider extends Thread {
          // lokale data og metoder B
           int ind;

          Arbeider (int in) {ind = in;}

           public void run( ) {
                 // her starter alle trådene
           } // end run

    } // end indre klasse Arbeider

} // end class Problem
```

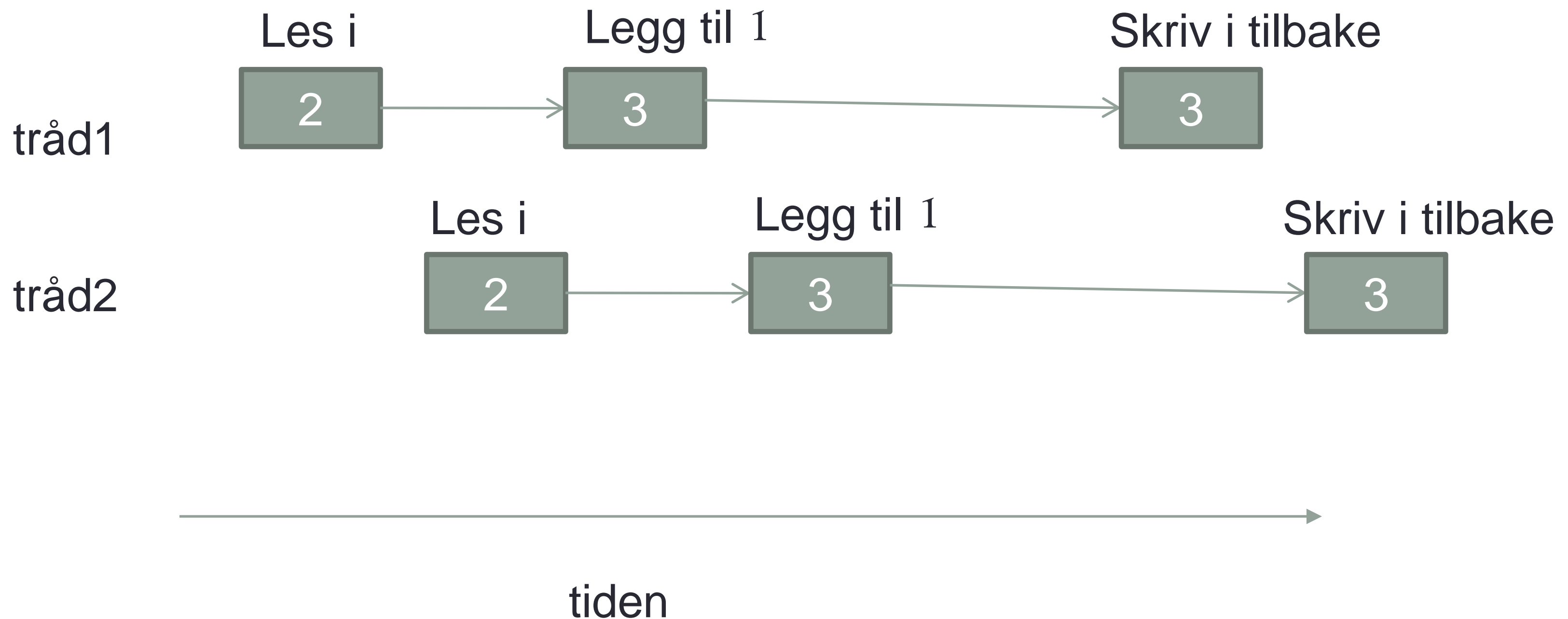
5) Parallell-problem: Samtidig les/skriv på samme variabel uten synkronisering:

- Antar at du har k tråder som alle sier : i++;
- 4 ulike løsninger, bare én er riktig

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

 /** Start >java Parallell <ant tråder> <ant ganger> */

class Parallell{
   long tall=0;        // Trådene summérer i denne
   CyclicBarrier b ; // sikrer at alle er ferdige
   long antTråder, antGanger ;

  public static void main (String [] args) {
       int antKj=
              Runtime.getRuntime().availableProcessors();
       System.out.println("Vi har "+antKj+ " kjerner.");
       Parallell p = new Parallell();
      p.antTråder  = Integer.parseInt(args[0]);
      p.antGanger = Integer.parseInt(args[1]);
      p.utfør();
}

void utskrift( double tid) { System.out.println("Tid"
  +antGanger+" kall:"+ antTråder+" Traader = "+tid+
  " sek,\n sum:"+tall+", tap:"+(antTråder*antGanger-
  tall)+" = "+(antTråder*antGanger-
  tall)*100.0/(antTråder*antGanger))+"%"};
}

synchronized void økTall(){ tall++;}          // 1)
// void økTall() { tall++;}                    // 2)
```

```
void utfør () {
    b = new CyclicBarrier((int)antTråder+1); //også main
    long t = System.nanoTime(); //start klokke
    for (int i = 0; i< antTråder; i++)
         new Para ().start();
    try{ // main tråden venter
        b.await();
    } catch (Exception e) {return;}
       double tid = (System.nanoTime()-t)/1000000000.0;
       utskrift(tid);
   }}

  class Arbeider extends Thread {
   // void økTall() { tall++;}                   // 3
   // synchronized void økTall(){ tall++;}       // 4)
   int ind;

   Arbeider(int in) {ind = in;}
    public void run() {
       for (int i = 0; i< antGanger; i++) {
            økTall();
       }
       try { // wait on all other threads + mai
          b.await();} catch (Exception e) {return;}
        }
    } // end run

  }// end class Para

}// end class Parallell
```

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



Test på i++ parallellisert (løsning: 2, 3 eller 4)

- Setter i gang **n tråder** (på en 2 kjerner CPU) som alle prøver å øke med 1 en felles variabel int i; 100 000 ganger **uten** synkronisering;
-
```
for (int j =0; j< 100000; j++) {
            i++;
}
```
- Vi fikk følgende feil - antall og %, (manglende verdier).Merk: Resultatene *varierer også mye* mellom hver kjøring :

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

6) Optimalisering i Java, omordning av instruksjoner,

- For å få raskere kode, byttes det i JVM ofte om på den kompilerte koden: – eks.:

```
y = 17;
x = 5;
z =  x + 3;
```

- Selv om vi ser at x er 5, **behøver *ikke* y å være lik 17**, fordi vi ikke trenger noen verdi på y enda.
- Hvis programmet går raskere, utsettes tilordningen av 17 til y (til den trenges i en ***beregning*** med y eller i en ***utskrift***).
- Det eneste vi garanteres av kompilatoren er at den kjører et program som gir ***samme resultat* som om vi utførte setningene slik de står i Java-koden – en etter en, ovenfra og nedover** .
- Altså: Bak kullissene utføres koden i en uforutsigbar rekkefølge for at koden for ***én tråd skal gå raskest.***

7) Flere problemer med parallellitet med tråder i Java

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

Hvordan virker cache inn på parallelle beregninger

- En tråd på én CPU kan oppdatere en variabel 'i', men leser/skriver nye verdier av 'i' bare til f.eks level 1 eller 2 cache (det kommer ingen oppdaterte verdier i hukommelsen).
- Andre tråder som vil lese 'i' får da lest gamle verdier – ikke de nye verdiene.
- Selv om en tråd på $core_1$ skriver verdier av beregninger ned i hukommelsen, tar det ca. 250 cycler, og der er god nok tid til at en annen tråd på en annen $core_k$ leser 'gamle' verdier 'lenge etter' at nye verdier har blitt beregnet.

Så mange problemer

- Trådene kan flette beregninger .
- Kompilatoren bytter om på instruksjonene.
- Cachen gjør at ulike tråder kan lese 'gamle' verdier av variable lenge etter at de er oppdatert i en annen tråd.

\+ At det tar 'lang' tid, ca. 3 millisekunder å starte noen få tråder (på den tida kan man sortere ca. 10-40 000 tall)

Er det mulig å lage **riktige og raskere** parallell sortering?

Synkronisering + faste regler kan løse problemene.

1. Vi må sperre ut ***alle tråder unntatt én*** fra å skrive på noen av de felles variable samtidig. Og da må ingen andre tråder samtidig lese disse felles variablene.
2. Vi må sørge for at alle verdier blir **synlige** for de tråder som trenger å lese dem (de må være skrevet ***og*** nådd fram til der andre tråder kan lese dem – hukommelsen, når de skal leses).

Løsning:

A. Har lært: synkroniserte metoder (litt langsom og passer ikke alltid til problemet, lock er litt raskere):

```
synchronized void addi() {
                    i ++;
          }
```

B. Det er mulig *selv* å sørge for at 1) og 2) ovenfor overholdes.

- Bruk barrier-synkronisering - kan løse 2) ovenfor
- Sørg selv ved hvordan du programmerer at 1) overfor gjelder *mellom hver synkronisering*.

Synkroniseringsvariable generelt

- Tråder som skal synkroniseres, gjør et kall på ***samme*** synkroniseringsvariabel (barrier, lock eller semafor) – i samme objekt. Da skjer:
  - Alle instruksjoner 'over' synkroniserings-kallet i koden vil bli ferdig utført *før* synkroniseringskallet – ingen utsatte operasjoner.
  - Alle verdier på *felles variable* som er endret av trådene vil være tilgjengelig og *synlig i felles hukommelse* når kallet blir utført.
  - Dvs. innholdet i alle cachene som har blitt endret, blir skrevet ned i hovedhukommelsen som et resultat av kall på en synkroniseringsvariabel.

8) To sentrale grep for å parallellisere

- Hvordan dele opp data
  - Etter verdiene til data
  - Etter hvor de står i f.eks en array

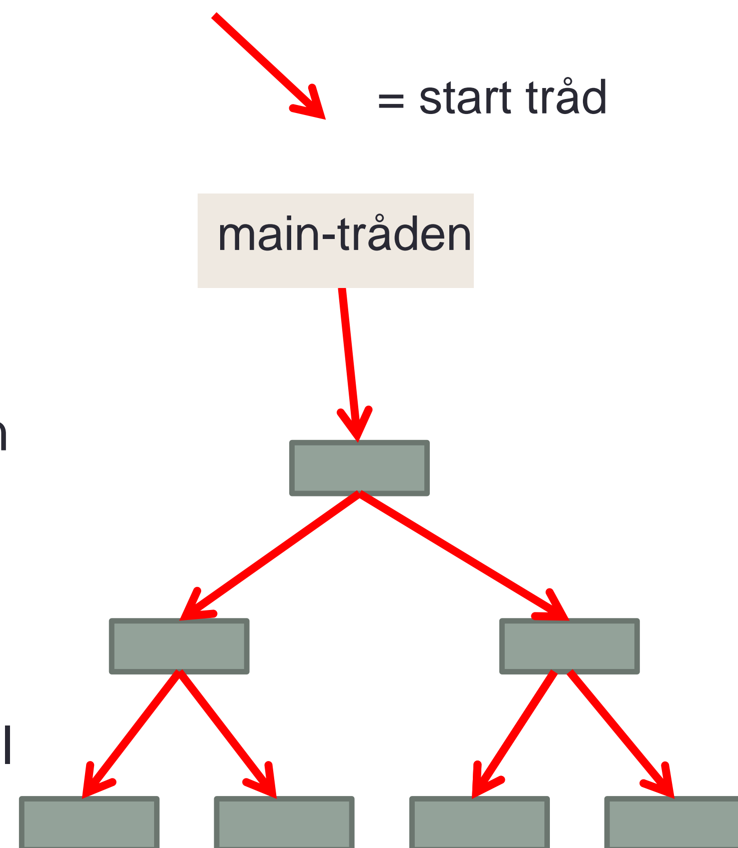
- Hvilke globale data blir kopiert til lokale data i hver tråd

9) Sortering - hvordan **ikke** gå i den rekursive fella!

- Vi skal nå gå gjennom hvordan vi behandler rekursjon og parallellisering av programmer med rekursjon
- Først en 'teoretisk' PRAM-lignende parallell løsning
  - som går ca. 1000x langsommere enn en sekvensiell versjon.
- Så skal vi se på to forbedringer som gjør at den uhyre treige løsningen vår tilslutt har en speedup på ca. 4
- Hele problemet bunner i de tallene vi presenterer i dag :
  - Å starte en ny tråd med vent på terminering tar: **92**μs (snitt mange ganger)
  - Å gjøre et metodekall tar: **0.016** μs (snitt mange ganger)

Parallell Quicksort

1. Bestem først hvor mange tråder vi vil ha (= 2^k-1) 1,3,7,..2^k-1 tråder.
2. main-tråden starter én tråd som gjør vanlig, sekvensiell oppdeling av a[] i to deler som 'vanlig' Quicksort.
3. Main legger seg så og venter på en semafor.
4. Start så en tråd for å sortere for hver av disse to delene.
5. Disse starter så hver sine to tråder, … osv. inntil vi har startet det antall tråder vi har bestemt oss for.
6. Deretter vil hver tråd på vanlig, rekursiv, sekvensiell måte sortere hver sin del og si release() på main's vente-semafor når de er ferdige.



Litt drøfting, parallell Quicksort

- Hver tråd skriver bare i de delene den ’eier’ selv, og ingen andre tråder leser heller disse.
- Siden ’main’ synkroniseres med alle trådene den starter, vet vi at main kan se alle de sorterte elementene i a[] når den vekkes opp.
- Vi har i altså oppfylt 1) og 2) kravet –
  og vel så det.
- Litt *langsom* fordi første oppdelingen ***ikke*** er parallell, neste oppdeling kan kjøre bare 2-parallell, så 4-parallell osv. - jfr. Amdahls lov.
- Vi må ikke erstatte alle rekursive kall med nye tråder – bare så mange nye tråder som antall kjerner

Om rekursiv oppdeling av et problem

- Svært mange problemer kan gis en (sekvensiell og parallell) rekursiv løsning:
  - De fleste søkeproblemer
    - Del søkebunken rekursivt opp i disjunkte deler og søk (i parallell) i hver bunke.
  - Mange sorteings-algoritmer som QuickSort, Flettesortering, og VenstreRadix-sortering er definert rekursivt definert
  - Oblig4 i inf2440 - den konvekse innhyllinga er rekursiv
- Skal nå bruke en ny formulering av QuickSort som eksempel og gi den 3 ulike løsninger:
  A. Ren oversettelse av rekursjonen til tråder
  B. Med to tråder for hvert nivå inntil vi bruker InnstikkSortering
  C. Med en ny tråd for hver nivå og avslutting av tråder når lengden < LIMIT (si 50 000) – deretter vanlig rekursjon

Generelt om rekursiv oppdeling av a[] i to deler


```
void Rek (int [] a, int left, int right) {
        <del opp omradet a[left..right] >
        int deling = partition (a, left,right);

        if (deling - left  > LIMIT ) Rek (a,left,deling-1);
        else <enkel løsning>;
        if (right - deling > LIMIT) Rek (a,deling, right);
        else  <enkel løsning>
}
```

Her noe stilisert, skal egentlig ha new Thread(new(Arbeider (a,left,deling-1))+ run-metode med samme innhold som Rek

```
void Rek(int [] a, int left, int right) {
        <del opp omradet a[left..right]>
        int deling = partition (a, left,right);
        Arbeider  t1 = null, t2= null;

        if (deling - left > LIMIT )  t1 = new Arbeider (a,left,deling-1);
        else <enkel løsning>;
        if (right - deling > LIMIT) t2 = new Arbeider (a,deling, right);
        else  <enkel løsning>
        try{ if (t1!=null)t1.join();
             if (t2!=null)t2.join();} catch(Exception e){};
}
```

A

```
void Rek (int [] a, int left, int right) {
            <del opp omradet a[left..right] >
            int deling = partition (a, left,right);

            if (deling - left  > LIMIT ) Rek (a,left, deling -1);
            else <enkel løsning>;
            if (right - deling > LIMIT) Rek (a, deling , right);
            else  <enkel løsning>
}
```

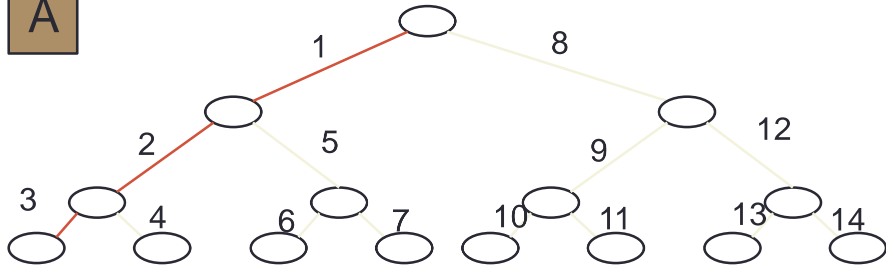
```
void Rek(int [] a, int left, int right) {
            <del opp omradet a[left..right]>
            int deling = partition (a, left,right);
            Thread t1 = null, t2=null;

            if (deling- left > LIMIT )
                (t1 = new Arbeider (a,left, deling -1)).start();
            else <enkel løsning>;
            if (right - deling> LIMIT)
                (t2 = new Arbeider (a, deling , right)).start();
            else  <enkel løsning>
            try{ if (t1!=null) t1.join();
                 if (t2!=null) t2.join();}
          catch(Exception e){return;};
}
```

B

Oppdeling med **to** tråder per nivå i treet:

- Når ventes det i den rekursive løsningen
  - Har det betydning for rekkefølgen av venting ?
- Når ventes det i den parallelle løsningen A?
  - Har rekkefølgen på venting på t1 og t2 betydning?
- Antar at kall på Rek tar T millisek.
- Hvor lang tid tar A og B
- Hvilken er raskest ?

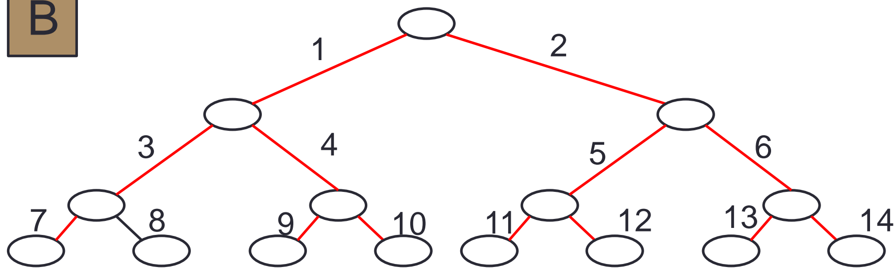
A



Rekursjon

Dybde først – sekvensiell, rekursiv metode

B



Tråd

Bredde først – parallelle tråder

Oppdeling med **en tråd** per nivå i treet:

- Hvorfor virker dette ?
- To alternativ løsning med 1 tråd
  - Har det betydning for rekkefølgen av venting ?
- Når ventes det i C-løsningen?
  - Har rekkefølgen på venting på t1 betydning?
- Når ventes det i D-løsningen?
- Antar at kall på Rek tar T millisek.
- Hvor lang tid tar C
- Hvor lang tid tar D

Hvilken er klart raskest:
C eller D?

D – er raskest fordi både høyre og venstre gren startes før man venter.

C

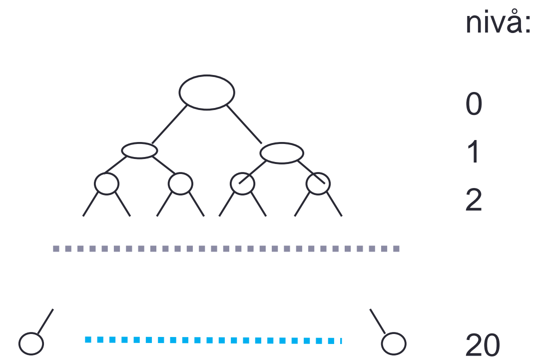
```
void Rek(int [] a, int left, int right) {
        <del opp omradet a[left..right]>
        int deling = partition (a, left,right);
        Thread t1;

        if (deling - left > LIMIT )
                Rek (a,left,deling-1);
        else <enkel løsning>;
        if (right - deling > LIMIT)
                t1 = new Arbeider (a,right,deling-1);
        else  <enkel løsning>
         try{t1.join();} catch(Exception e){};}
```

D

```
void Rek(int [] a, int left, int right) {
        <del opp omradet a[left..right]>
        int deling = partition (a, left,right);
        Thread t1;

        if (deling - left > LIMIT )
                t1 = new Arbeider (a,left,deling-1);
        else <enkel løsning>;
        if (right - deling > LIMIT)
                Rek (a,deling, right);
        else  <enkel løsning>
        try{t1.join();} catch(Exception e){};}
```

Hvor mange kall gjør vi i en rekursiv løsning?

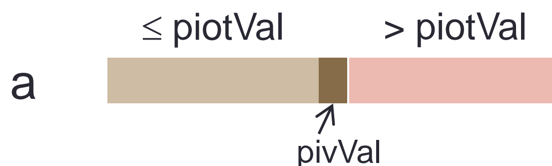
- Anta Quicksort av $n = 2^k$ tall
  ($k = 10 \Rightarrow n = 1000$, $k = 20 \Rightarrow n = 1$ mill)
- Kalltreet vil på første nivå ha 2 lengder av 2^{19}, på neste: $4 = 2^2$ hver med 2^{18} og helt ned til nivå 20, hvor vi vil ha $2^{20+1}-1$ kall hver med $1 = 2^0$ element.
- I hele kalltreet gjør vi altså 2 millioner -1 kall for å sortere **1 mill tall** !
- Bruker vi innstikksortering for $n < 32 = 2^5$ så får vi 'bare' $2^{20-5+1} = 2^{15+1} - 1 =$ **65 535** kall.
- Metodekall tar først: **2 us** men så **0.02** µs og kan også optimaliseres bort (og gis speedup >1)
- Å lage en tråd og starte den opp tar først : ca.**3000** µs, men så ca. **62** µs for de neste trådene (med start() og join())



Vi kan IKKE bare erstatte rekursive kall med nye tråder i en rekursiv løsning !

10) Sekvensiell kvikksort – ny og enklere kode

```
// sekvensiell Kvikksort
void quicksortSek(int[] a, int left, int right) {
        int piv = partition (a, left, right); // del i to
        int piv2 = piv-1, pivotVal = a[piv];
        while (piv2 > left  && a[piv2] == pivotVal) {
             piv2--; // skip like elementer i midten
        }

        if ( piv2-left > LIMIT) quicksortSek(a, left,  piv2);
           else insertSort(a,left, piv2);
        if ( right-piv >LIMIT) quicksortSek(a, piv + 1, right);
          else  insertSort(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++;
        }
    }
    swap(a, index, right); // sett pivot tilbake
    return index;
 } // end partition

 void swap(int [] a, int left, int right) {
      int temp = a[left];
      a[left] = a[right];
      a[right] = temp;
 } // end swap
```

A) En parallell kode (modellert etter A)

```
void Rek(int [] a, int left, int right) {
        <del opp omradet a[left..right]>
        int deling = partition (a, left,right);
        Thread t1 = null, t2=null;

        if (deling- left > LIMIT )
           (t1 = new Thread (a,left, deling -1)).start();
        else insertSort(a,left, deling -1);
        if (right - deling> LIMIT)
          (t2 = new Thread (a, deling , right)).start();
        else  insertSort(a, deling , right);
        try{ if (t1!=null)t1.join();
           if (t2!=null)t2.join();} catch(Exception e){};
}
```

Ren kopi av rekursiv løsning: Katastrofe

```
M:>java QuickSort 100 10 100000 1 uke9.txt
Test av  TEST AV QuickSort
 med 8 kjerner ,   Median av:1 iterasjoner, LIMIT:2
```

n	sekv.tid(ms)	para.tid(ms)	Speedup
100 000	34.813	41310.276	0.0008
10 000	0.772	735.838	0.0010
1 000	0.078	66.007	0.0012
100	0.009	3.491	0.0026

Konklusjon:

- For store n speeddown på > 1000
- Kunne ikke kjøre for n > 100 000 pga. trådene tok for stor plass

B) Hva med en passe innstikksort LIMIT = 32 ?

```
>java QuickSort 100 10 1000000 1 uke9.

Test av  TEST AV QuickSort
med 8 kjerner ,   Median av:1 iterasjoner, LIMIT:32


      n       sekv.tid(ms)    para.tid(ms)     Speedup
  1000000       89.181       41789.076       0.0021
   100000        8.118         823.432       0.0099
    10000        3.021          55.845       0.0541
     1000        0.060           3.463       0.0173
      100        0.006           0.302       0.0185
```

Konklusjon:

- Mye bedre, men fortsatt 100 x langsommere enn sekvensiell(N = 100 000)
- Greide nå n= 1 mill (fordi færre tråder)
- Fortsatt håpløst dårlig pga. for mange tråder
- Trenger ny idé : Bruk sekvensiell løsning når n < 50 000 ? BIG_LIMIT

C) Skisse av ny løsning

```
void Rek(int [] a, int left, int right) {
    if ( right – left  < BIG_LIMIT) quicksort (a, left,right);
    else {
            <del opp omradet a[left..right]>
            int deling = partition (a, left,right);
            Thread t1 = null, t2=null;

            //if (deling- left > LIMIT )
                (t1 = new Thread (a,left, deling -1)).start();
            //else insertSort(a,left, deling -1);
            //if (right - deling> LIMIT)
               (t2 = new Thread (a, deling , right)).start();
            //else  insertSort(a, deling , right);
            try{ if (t1!=null)t1.join();
               if (t2!=null)t2.join();} catch(Exception e){};
    }
}
```

Generere nye tråder bare i toppen av rekusjonstreet

- Kan da også stryke kode om LIMIT (vil ikke bli utført) i parallell kode
- Bruken av insertSort gjøres i (sekv) quicksort(..)

Kjøreeksempel med BIG_LIMIT og LIMIT

```
>java QuickSort 100 10 100000000 1 uke9.txt
Test av  TEST AV QuickSort med BIG_LIMIT
 med 8 kjerner ,   Median av:1 iterasjoner,
                   LIMIT:32, BIG_LIMIT:50000


     n        sekv.tid(ms)   para.tid(ms)     Speedup
 100000000    12042.708       3128.675        3.8491
  10000000     1090.252        277.264        3.9322
   1000000       92.958         32.640        2.8480
    100000        7.682          5.198        1.4777
     10000        0.616          0.737        0.8356
      1000        0.051          0.117        0.4354
       100        0.013          0.015        0.8636
```

BIG_LIMIT = 100 000

```
>java QuickSort 100 10 100000000 1 uke9.txt
Test av  TEST AV QuickSort med BIG_LIMIT
 med 8 kjerner ,   Median av:1 iterasjoner, LIMIT:32,
BIG_LIMIT:100000


      n       sekv.tid(ms)    para.tid(ms)     Speedup
 100000000    12110.344        2967.764        4.0806
  10000000     1084.587         277.454        3.9091
   1000000       93.428          32.078        2.9125
    100000        7.894           7.828        1.0085
     10000        0.815           0.617        1.3224
      1000        0.072           0.101        0.7153
       100        0.013           0.015        0.8410
```

Konklusjon om å parallellisere rekursjon

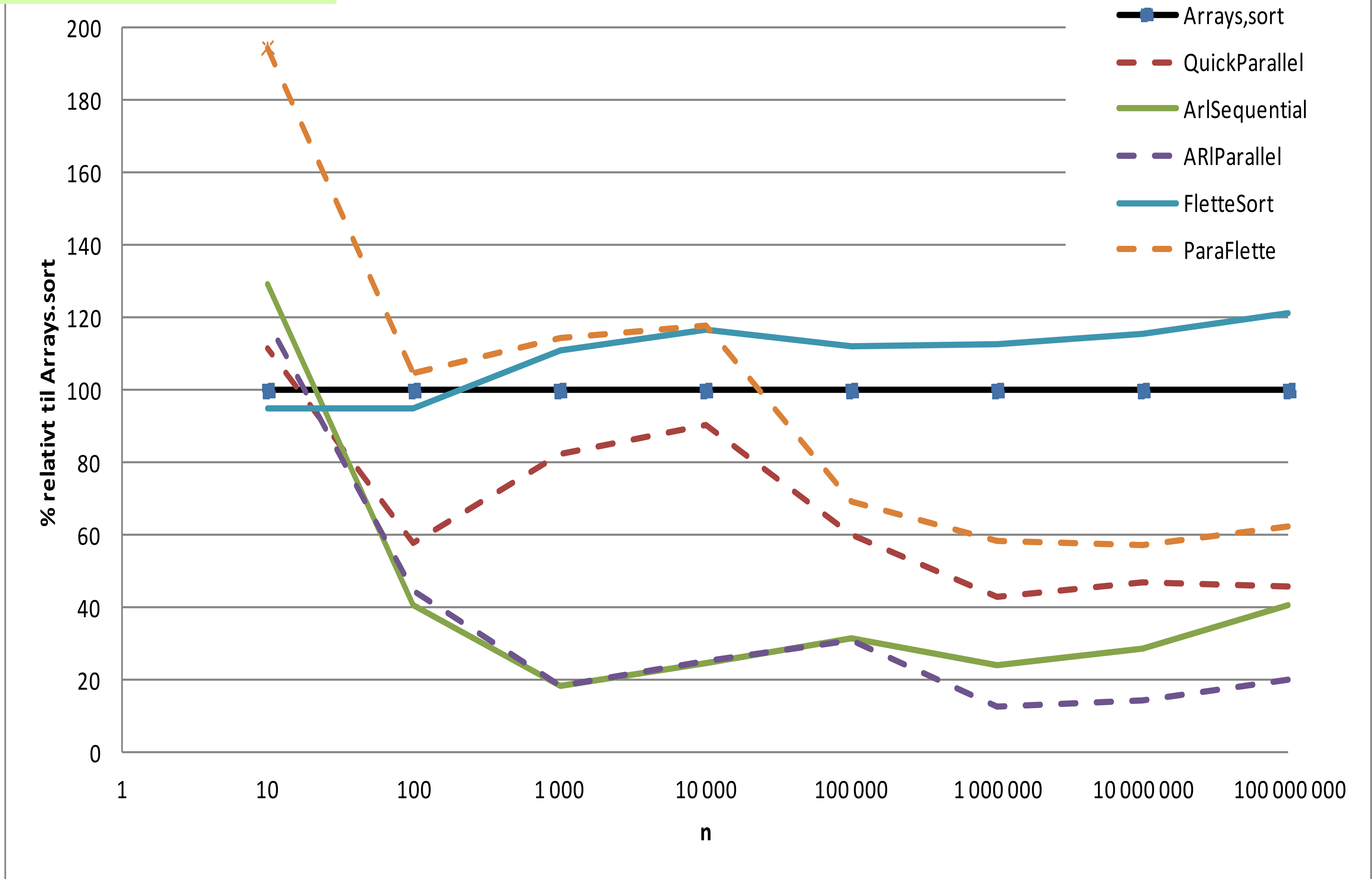
- Antall tråder må begrenses !
- I toppen av treet brukes tråder (til vi ikke har flere og kanskje litt mer)
- I resten av treet bruker vi sekvensiell løsning i hver tråd!
- Viktig også å kutte av nedre del av treet (her med insertSort) som å redusere treets størrelse drastisk (i antall noder)
- Vi har for n = 100 00 gått fra:

n	sekv.tid(ms)	para.tid(ms)	Speedup	
100000	34.813	41310.276	0.0008	Ren trådbasert
100000	8.118	823.432	0.0099	Med insertSort
100000	7.682	5.198	1.4777	+ Avkutting i toppen

- Speedup > 1 og ca. 10 000x fortere enn ren oversettelse.

ARL = VenstreRadix
N.B her er det 1/ speedup. **Lavere verdier er bedre**.

parallelle og 3 sekvensielle sorteringsalgoritmer (Dell 3100 - Intel Core i7- 4 core, 2,67GHz) tider i % relativt til sekvensiell Quicksort



Analyse (I)

- Vi ser at parallell Parallell Venstre Radix (ALR) er:

  ca. **5x** raskere enn Arrays.sort generelt , (n> 1 mill)

  ca. **2-4x** raskere enn parallell Quicksort

  ca. **2x** raskere enn sekv. ALR med 4 core

  ca. **4x** raskere enn sekv. ALR med 4/8 core

- Parallell Quicksort er:
  - 2x raskere enn OJ Dahls Quicksort uansett antall core

- Hvorfor ikke bedre speedup ?

Hvorfor ikke 4x og 8x raskere ?

- Vi gjør en ekstra kopiering av det vi skal sortere i PQuick (til lokale b[] arrayer og tilbake)
- En beregning er ikke raskere enn sin langsomste del
- Sortering gjør *få* operasjoner i CPUen, men leser ofte i hukommelsen.
- Vi vet ikke hvor rask hukommelsen-forbindelsen er på de to CPUene:
  - Kanskje er den en flaskehals?
  - AMD har/hadde raskere hukommelse-kanal enn Intel .
  - Kanskje derfor skalerer AMD bedre enn Intel.

Konklusjoner

- Parallell sortering skalerer bra, og særlig:
  - PLR som ikke har noen sekvensiell del og er relativt sett 2x så rask med 2x antall core.
  - Parallell Quick har ikke samme speedup, og på grunn av sin sekvensielle del, vil neppe bli særlig raskere med 50 eller 100 core.
  - (jeg har publisert en full parallell QuickSort som i stor grad løser dette siste problemet)
- Parallell programmering er vanskelig.
- Programmeringsmønstre som:
  - En skriver og ingen andre leser;
    så synkroniser ;
    så kan andre tråder lese;
  - Er helt avgjørende for å få riktig kode.

Mye mer: **INF2440** våren 2018