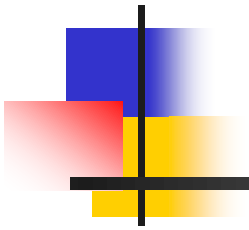


INF2440 Uke 10, v2018 :

Arne Maus
PSE,
Inst. for informatikk

Hva så på i uke 9:

- Et sitat om tidsforbruk ved faktorisering
- Om en feil i Java ved tidtaking (tid == 0 ??)
- Hvordan parallellisere rekursive algoritmer
- Gå IKKE i 'direkte oversettelses-fella'
- eksemplifisert ved Quick-sort, 3 ulike løsninger
- Hvor lang tid tar de ulike mekanismene i Java 6 og 8
- Store forbedringer Java6 til Java8
- Introduktion til Radix Programmering
- Eksempel med spillekort
- Whiteboard Gjennemgang av eksempel

Hva skal vi se på i Uke10

- Del1 : Oblig 4 – parallell MultiRadix
  - Flere flere alternative løsninger til Oblig4 - MultiRadix
- Del 2 :Automatisk parallellisering av rekursjon
  - PRP- Parallel Recursive Procedures
    - Nåværende løsning (Java, multicore CPU, felles hukommelse) – implementasjon: Peter L. Eidsvik
    - Mange tidligere implementasjoner fra 1994
      - (C over nettet, C# på .Net, Java over nettet, med MPI..)
  - Demo av to kjøringer ?
  - Hvordan kan vi bygge en kompilator (preprosessor) for automatisk parallellisering
    - Prinsipper (bredde-først og dybde-først traversering av r-treet)
    - Datastruktur
    - Eksekvering
  - Krav til et program som skal bruke PRP

Generelle grep for parallellisering

- Vi parallelliserer bare løkker som tar 'lang' tid – O(n)
- Mellom løkkene synkronisere vi alle de parallelle objektene på **én og samme: CyclicBarrier cb** når trådene trenger informasjon skapt (skrevet) av en annen tråd.
  - Vi må jo vente på de andre trådene har skrevet sitt (og det er skrevet ned i det felles hovedlageret)
- Synkronisering tar tid slik at hver tråd heller oppdaterer (med sine data) **kopier av felles data**, og først når alle trådene er ferdige, kan slike kopierte data 'legges inn' i de synlige fellesdata.
- Hvis vi trenger det (vel ikke i Oblig 4) kan vi ha flere indre klasser (med ulike run-metoder) for å lage ulike parallelle tråder.
- Husk at en tråd er et sekvensielt program og godt kan kalle og bruke 'alle' metodene i den omsluttende klassen.

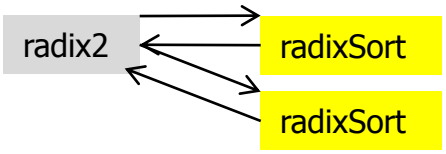
Del 1 : Oblig 4 - Parallell MultiRadix

- Parallelliser Radix-sortering med en-til-flere sifre (her illustrert med to sifre)
- Skriv rapport om speedup for n= 1000, 10 000, 100 000, 1 mill., 10 mill og 100 mill.
- Radix består av to metoder, begge skal parallelliseres.
- Den første har ett steg som skal parallelliseres
  a1) finn max(a[]) – parallelliseres: løst tidligere
  a2) finn høyeste bit i max (skal ikke parallelliseres)
- Den andre har tre steg – løses i parallell effektivt:
  b) tell hvor mange det er av hvert sifferverdi i a[] i count[]
  c) legg sammen verdiene i count[] til 'pekere til b[]
  d) flytt tallene fra a[] til b[]
- Steg b) er løst i ukeoppgave (hvor bla. hver tråd har sin kopi av count[])

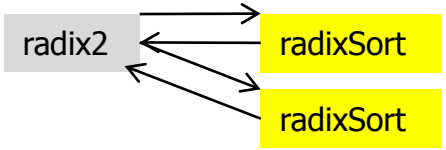
Oblig4 - MultiRadix

- Problemet med data-kappløp: To eller flere tråder skriver 'samtidig' på samme variabel (i++ problemet), f.eks. i samme plass i en array:
  - Løsning: Hver tråd har en kopi av disse felles variable
  - Etter at alle trådene er ferdig (f.eks. etter en barrier-synk) kan resultatene fra hver tråd samstilles (også dette helst i parallell) til et felles svar
  - Muligens må man kopiere data mer enn en gang ?
- To alternativer til oppdeling av data i arrayer man skal behandle med k tråder:
  - Dele opp arrayen i like store deler (det er indeksene man deler opp)
  - Dele opp etter verdiene i elementene (tråd 0 eier de minste verdiene, tråd 1 de nest-minste,.. , tråd k-1 de største)

Den første av to algoritmer som sekvensiell 2-siffer Radix består av.




```
static void radix2(int [] a) {
      // 2 digit radixSort: a[]
      int max = a[0], numBit = 2, n =a.length;

     //a)  finn max verdi i a[]
      for (int i = 1 ; i < n ; i++)
          if (a[i] > max) max = a[i];

       // a2) bestem antall bit i siffer1 og siffer2
       while (max >= (1L<<numBit) )numBit++; // antall siffer i max
       int bit1 = numBit/2,
          bit2 = numBit-bit1;

      int[] b = new int [n];
      radixSort( a,b, bit1, 0);     // første siffer fra a[] til b[]
      radixSort( b,a, bit2, bit1);// andre siffer, tilbake fra b[] til a[]
 }
```

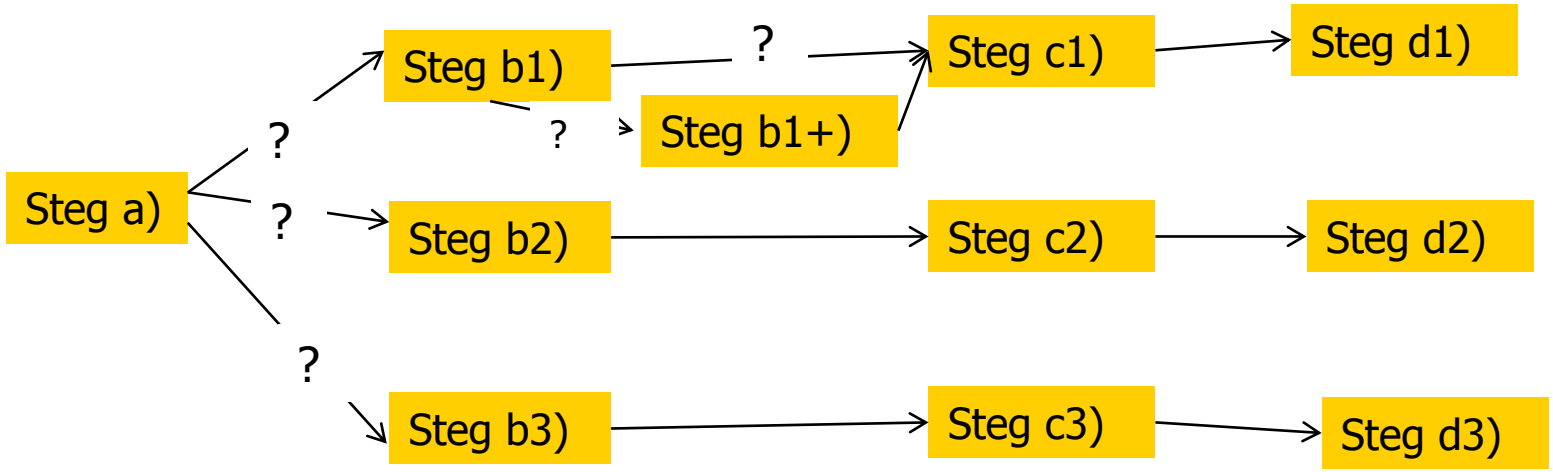



```
/** Sort a[] on one digit ; number of bits = maskLen, shiftet up 'shift' bits */
static void radixSort ( int [] a, int [] b, int maskLen, int shift){
      int  acumVal = 0, j, n = a.length;
      int mask = (1<<maskLen) -1;
      int [] count = new int [mask+1];

     // b) count=the frequency of each radix value in a
      for (int i = 0; i < n; i++)
         count[(a[i]>> shift) & mask]++;

     // c) Add up in 'count' - accumulated values
      for (int i = 0; i <= mask; i++) {
           j = count[i];
            count[i] = acumVal;
            acumVal += j;
       }

     // d) move numbers in sorted order a to b
      for (int i = 0; i < n; i++)
         b[count[(a[i]>>shift) & mask]++] = a[i];

}// end radixSort
```

Datastrukturer og grep i Oblig4 (hvordan parallellisere)

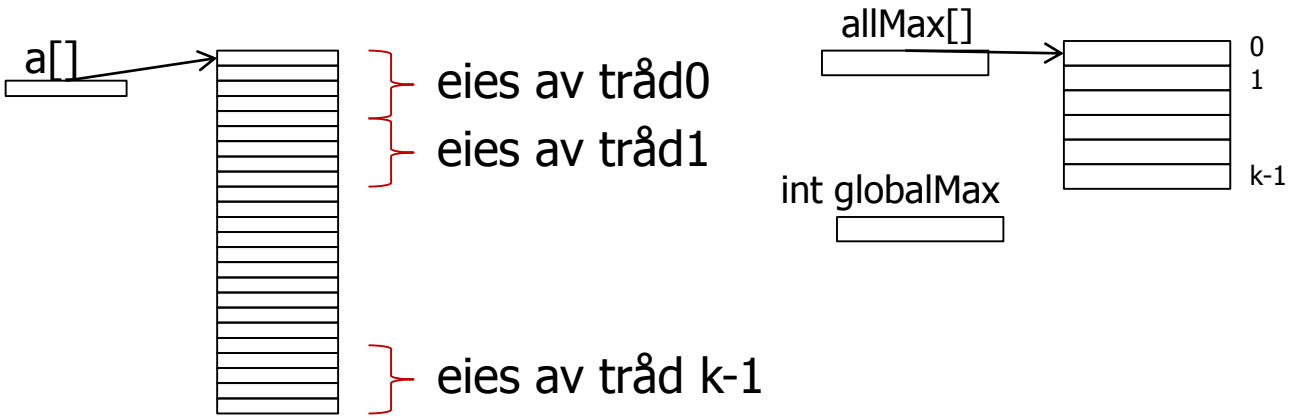
- Radix består av 4 steg (løkker) – flere valg for datastruktur og oppdeling ved parallellisering
  - a1) finn max verdi i a[]
  - b1-3) count= oppttelling av ulike sifferverdier i a[]
  - c1-3) Summér opp i count[] akkumulerte verdier (pekere)
  - d1-3) Flytt elementer fra a[] til b[] etter innholdet i count[]
- Generelt skal vi se følgende teknikker med k tråder:
  - Dele opp data i a [] i *k* like deler
  - Dele opp verdiene i a[] i *k* like deler => dele opp count[] i *k* like deler
  - Kopiere delte data til hver tråd – her lokal count[]-kopi en eller to ganger
  - Innføre ekstra datastrukturer som *ikke* er i den sekvensielle løsningen

Roadmap – oversikt over drøftelsen av Oblig4 – flere alternativer på steg b og senere endringer i c og d



FinnMax Tell sifferverdier lag pekere flytt a[] til b[]

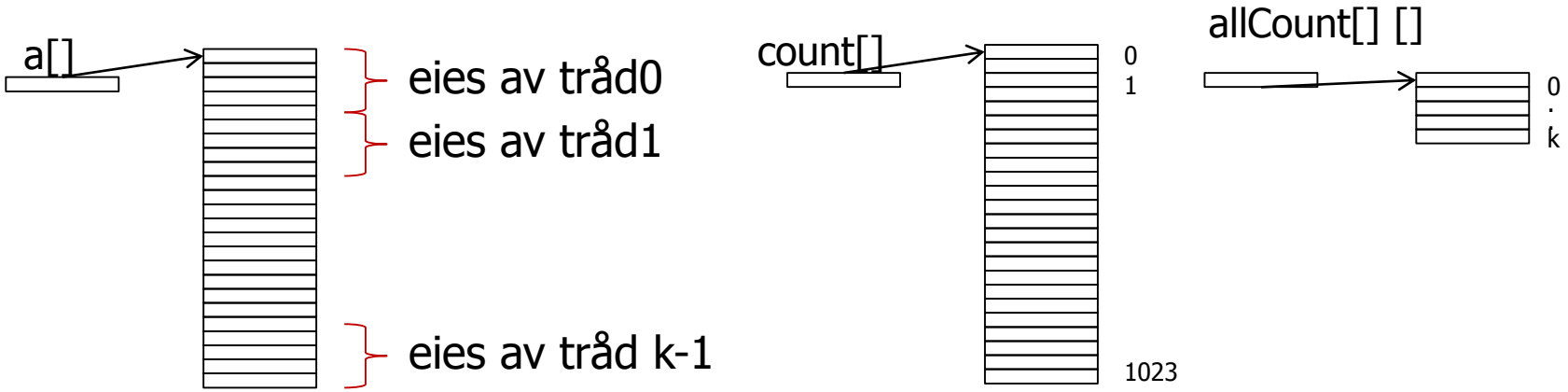
a) finn max verdi i a[]



- Tråd-i finner max i sin del av a[] og legger svaret i allMax[i]
- <sync på en CyclicBarrier cb>
- Nå har alle trådene sin max i allMax[] – valg nå:
  - Skal en av trådene (f.eks. tråd-0) finne svaret og legge det i en felles globalMax (mens de andre trådene venter i så fall nok en <sync på en CyclicBarrier cb>) ?
  - Skal alle trådene hver regne ut en lokal globalMax (de får vel samme svar?) og fortsette direkte til steg b)
  - Alternativer: Ja, men ingen mer effektive.

b) count= oppttelling av ulike sifferverdier i a[]

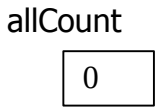
Anta at det er 10 bit i et siffer – dvs. 1024 mulige sifferverdier



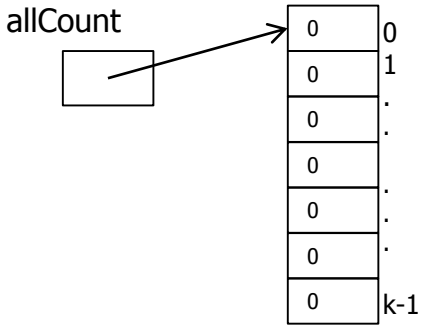
- Skal:
  1. Hver tråd ha en kopi av count[]
  2. Eller skal count være en AtomicIntegerArray
  3. Eller skal de ulike trådene gå gjennom hele a[] og tråd-0 bare ta de små verdiene, tråd-1 de nest minste verdien,..(dvs: dele verdiene mellom trådene)

Om delvis deklarasjoner av arrayer.

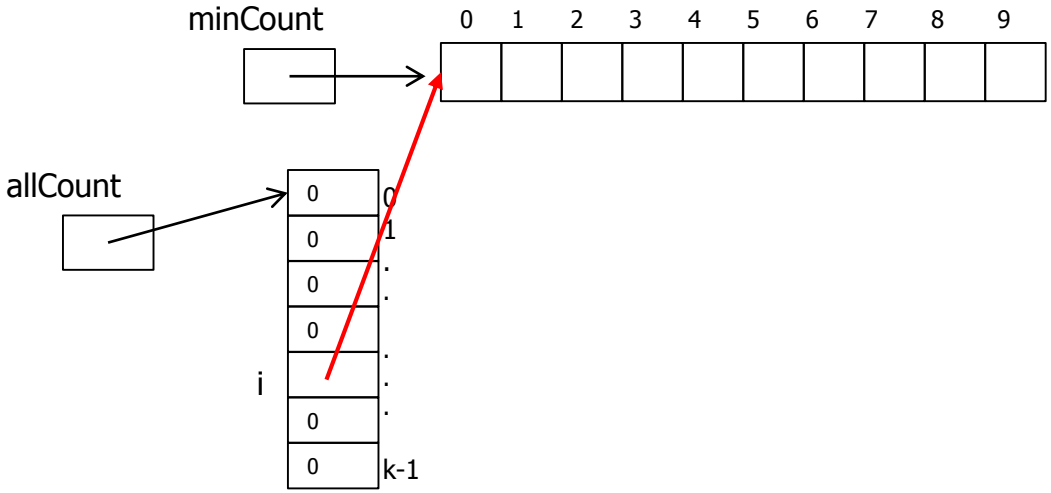
int allCount [] []; - da lages ?



int allCount [] [] = new int [k][];
 - da lages ?

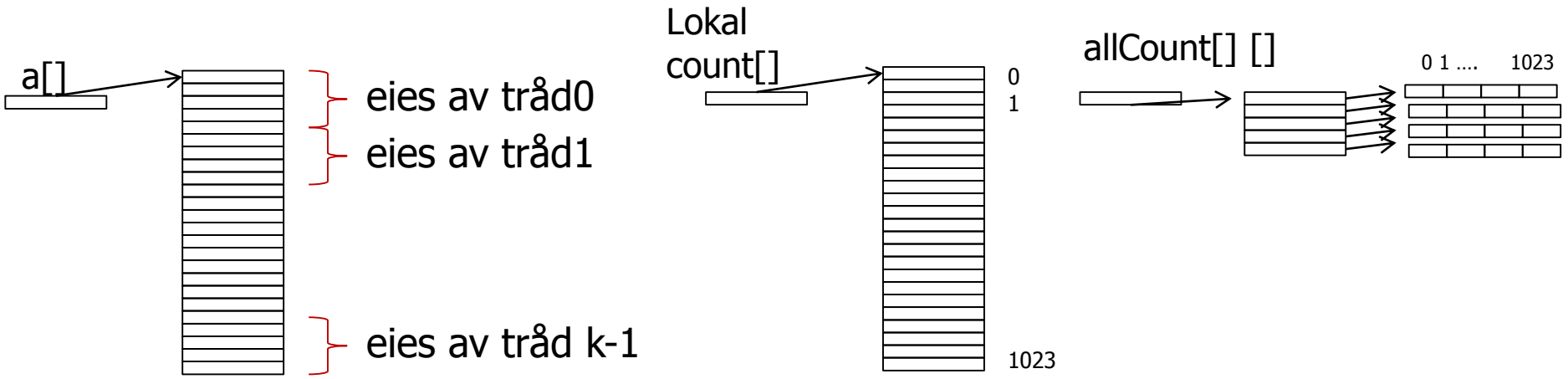


int [] minCount = new int[10] ;



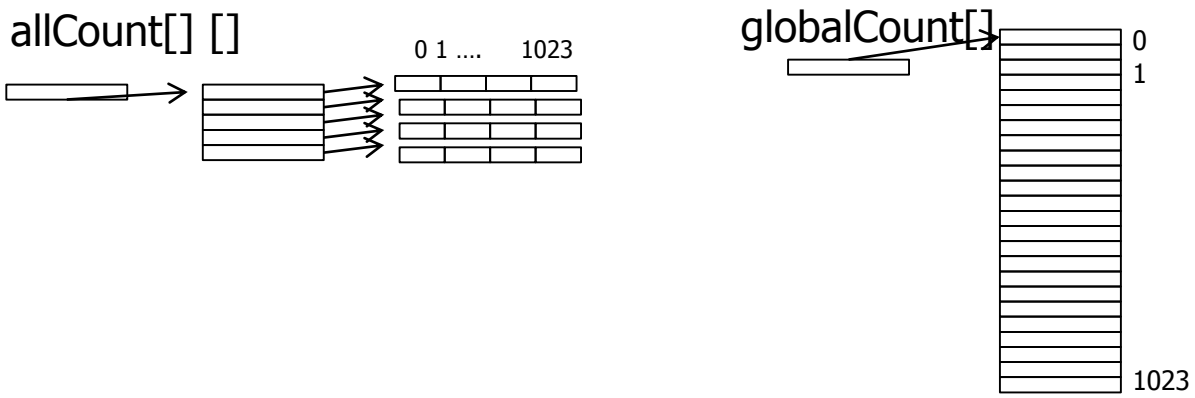
allCount [i] = minCount;

Løsning b1) – lokale count[] i hver tråd som etter opptelling settes inn i allCount[] []



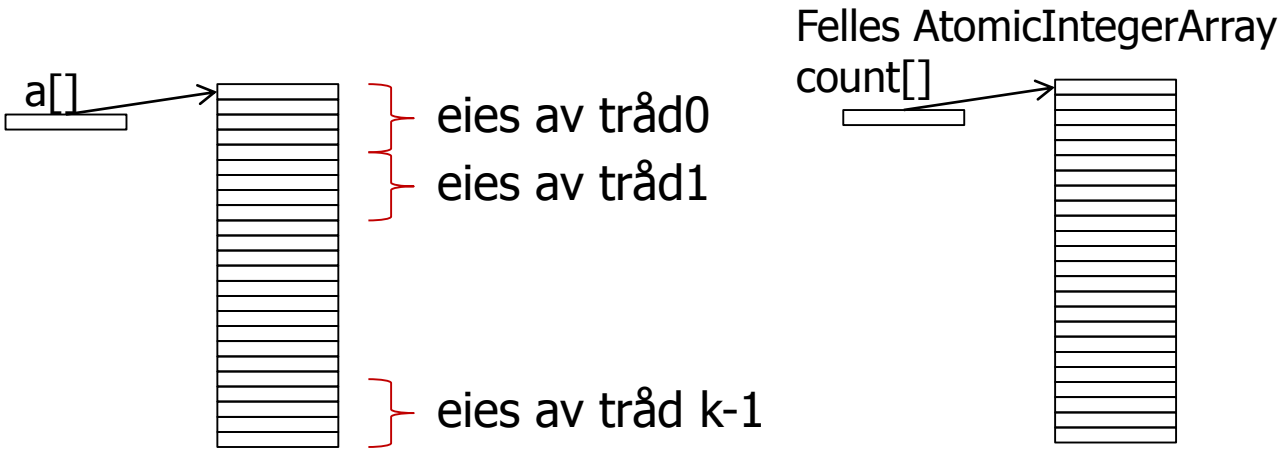
- Hver tråd-i teller opp de ulike sifferverdien i sin del av a[] opp i sin lokale count[] som så hektes inn i allCount[] []
- <sync på en CyclicBarrier cb>
- Nå er hele opptellingen av a[] i de k count[]-ene som henger i allCount[] []
- Denne løsningen trenger (kanskje) et tillegg b1-b:
  - Summering av verdiene i allCount[][] til en felles globalCount[]

b1+) : Summering av verdiene i allCount[][] til en felles: globalCount[]



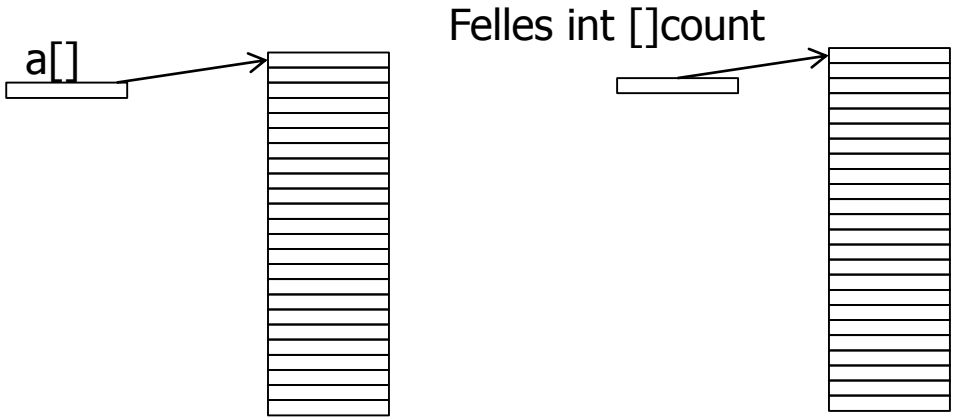
- Deler opp de mulige sifferverdiene (1024) på de k trådene slik at tråd-0 for de 1024/k minste, tråd-1 de 1024/k nest minste,…
- Tråd-i summerer sine verdier (sine kolonner) 'på tvers' i de k count[] – arrayene som henger i allCount[] [], inn i globalCount[]
- Bør allCount[][] transponeres før summering (mer cache-vennlig) ??
- <sync på en CyclicBarrier cb>
- Da er globalCount[] fullt oppdatert
- Spørsmål:
  - Trenger vi globalCount[] , eller holder det med allCount[][]?
  - Svar : Avhenger av neste steg c)

Løsning b2) – Eller skal count[] være en AtomicIntegerArray?



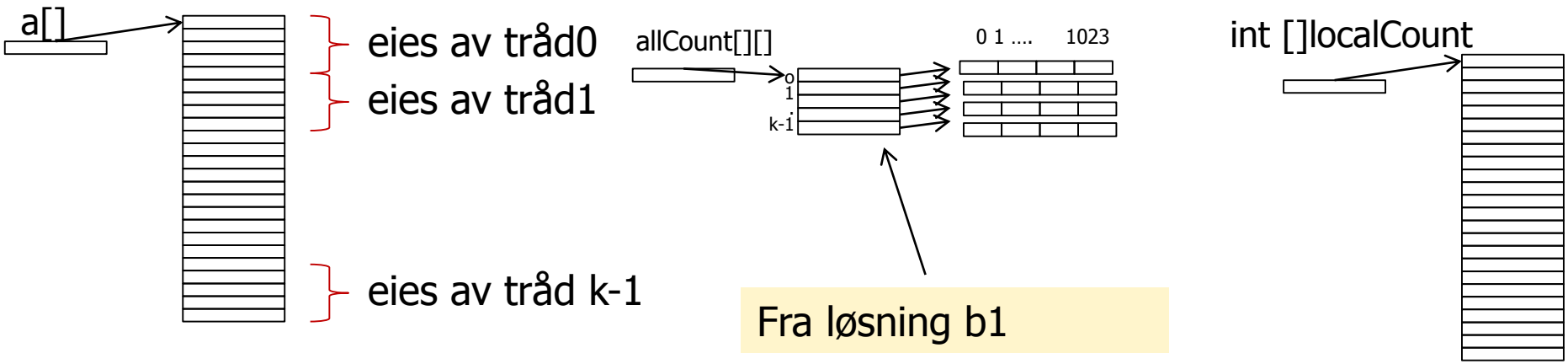
- Alle tråder oppdaterer AtomicIntegerArray count[] samtidig uten fare for synkroniserings-problemer med sifferverdiene fra sin del av a[] pga synkroniseringa av AtomicIntegerArray-elementene
- <sync på en CyclicBarrier cb>
- Spørsmål:
  - Hvor mange synkroniseringer trenger vi med denne løsningen

Løsning b3) – Eller skal de ulike trådene gå gjennom hele a[] og tråd-0 bare ta de n/k minste verdiene, tråd-1 de n/k nest minste verdiene,.. (dvs. dele verdiene mellom trådene)



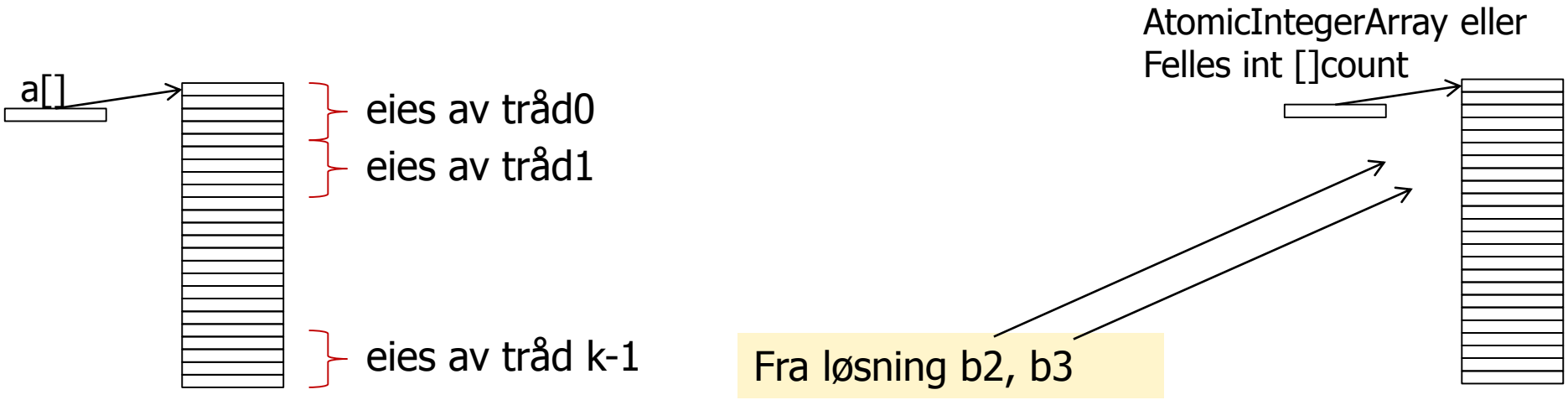
- Tråd-i oppdaterer count[] bare med **de verdiene** som tråd-i eier uten fare for synkroniserings-problemer med sifferverdiene fra hele a[] – ingen andre skriver slike sifferverdier.
- <sync på en CyclicBarrier cb>
- Spørsmål:
  - Hvor mange synkroniseringer trenger vi med denne løsningen
  - Hvor mye av a[] leser hver tråd

c1) **Gitt b1**: Summér opp i count[] akkumulerte verdier (pekere)



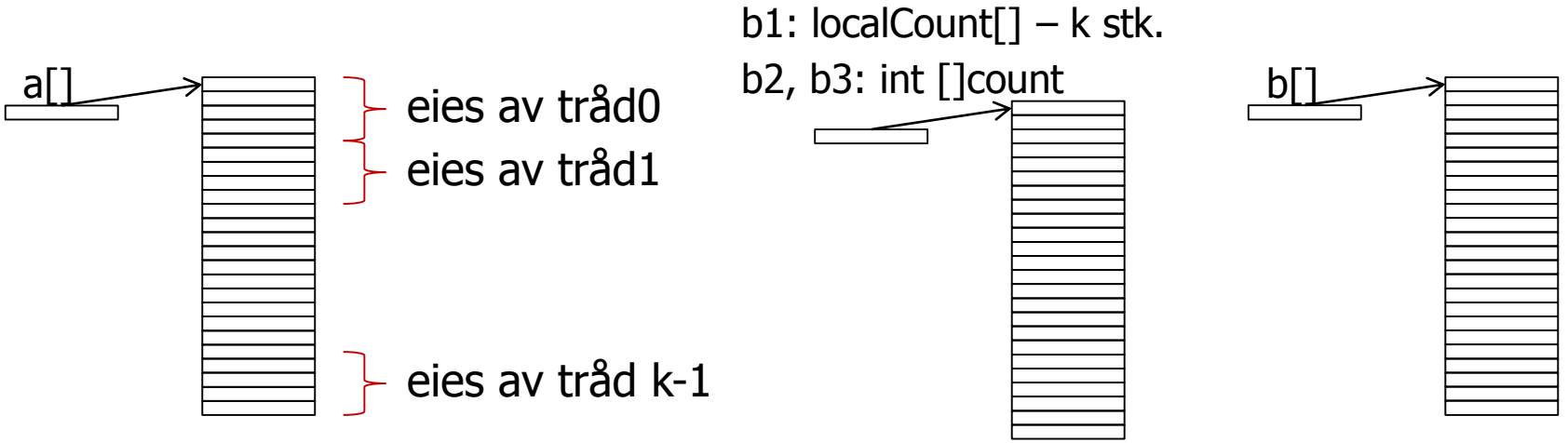
- Vi trenger et prinsipp for at hver tråd finner akkurat hvor den skal plassere sin del av a[].
  - Hver tråd får **nok en kopi** av count[]: localCount[] – initielt tom (=0)
  - Tråd-i sin localCount[t] er lik summen av alle tråder sin opptelling i allCount[r][s] for alle r og s<t + sum av allCount[r][t] r < i når s == t
  - Ideen her er at før tråd-*i* plasserer sine sifferverdier *s*, må det først gjøres plass til alle lavere sifferverdier + sifferverdiene med verdi *s* i tråder med indeks mindre enn *i*.
- <sync på en CyclicBarrier cb>

c2 og c3) Summér opp i count[] akkumulerte verdier (pekere)



- I løsning b2 og b3 har vi samme situasjon som sekvensiell løsning. Kan parallelliseres som følger :Deler verdiene i count[] mellom de k trådene som før. Alle summerer sine verdier og legger de i en separat array delSum[k].
- <sync på en CyclicBarrier cb>
- Deretter justerer du dine plasser i count[] med summen av delsummene i delSum[k] fra tråder med mindre indeks enn deg.
- Spørsmål:
  - Hvor lang tid tar dette kontra at tråd-0 gjør hele jobben og de andre venter.

d1,2 og 3) Flytt elementer fra a[] til b[] etter innholdet i count[]



- Løsning b1: Nå kan alle trådene i full parallell kopiere sine elementer, fordi hver localCount[] peker inn i ulike plasser i b[].
- Løsning b2: Alle trådene kan i full parallell kopiere over fra a[] til b[] fordi hver gang blir synkronisering foretatt av AtomicIntegerArray.
- Løsning b3: Nå kan alle trådene i full parallell flytte elementer fra a[] til b[] fordi hver tråd bare flytter sine sifferverdier. Hver tråd går her gjennom hele a[] .
- Alle b1,b2,b3: <sync på en CyclicBarrier cb>
  - Spørsmål: Hvilken av løsningene tar lengst tid?

Tider på **Oblig 4** – min løsning, 100m tall (Intel i7, 4/8 kjerner, 3.4 GHz)

```
M:\INF2440Para\Oblig4-Radix\ParaRadix>java -Xmx12000m Oblig4 100000000 3 a.txt
 Run no: 0, num cores:4
  Parallel sorting in :            468.201999 millisec.
  Sequential sorting in   :         884.333952 millisec.
  Arrays.sort sequential in   : 8725.968838 millisec.
Run no: 1, num cores:4
  Parallel sorting in :            379.784256 millisec.
  Sequential sorting in   :         844.035614 millisec.
  Arrays.sort sequential in   : 8661.582707 millisec.
Run no: 2, num cores:4
  Parallel sorting in :            357.557321 millisec.
  Sequential sorting in   :         814.044092 millisec.
  Arrays.sort sequential in  :  8714.827131 millisec.

n=  100 000 000
Median Sequential Radix time:    844.036
Median Quick  seq       time:    8714.827
Median Radix parallel   time:     379.784,
 Speedup: SeqRadix/ParaRadix:                    2.22
 Speedup: Arrays.sort (SeqQuicksort)/ SeqRadix: 10.32
 Speedup: Arrays.sort (SeqQuicksort)/ParaRadix: 22.95
```

'Samme' tider med 8 tråder

Fra Oblig4 – teksten – løse b):

- Opprett en to-dimensjonal int[][] allCount = new int[antTraader][] som fellesdata. I tillegg deklareres også int[]sumCount = new int[numSifVal] som fellesdata.
- Du deler så *først* opp a[] slik at tråd$_0$ får de *n/k* første elementene i a[],tråd$_1$ får de neste *n/k* elementene,…, og tråd$_{antTråder-1}$ de siste elementene i a[].
- Hver tråd har en egen int[]count = new int[numSifVal]. Vi teller så i alle trådene opp hvor mange det er av hver mulig sifferverdi i den delen av a[] som vi har, og noterer det i vår lokale count[].
- Når tråd$_i$ er ferdig med tellinga, henger den sin count[] opp i den doble int-arrayen som da vil inneholde alle opptellingene fra alle trådene, slik: allCount[i] = count;

Fra Oblig4 – teksten – del 2:

- Alle trådene synkroniserer på en sykliske barriere '*synk*'.
- Nå skal vi dele opp arrayen allCount[][] etter verdier i a[], slik at tråd$_0$ får de n/k første elementene i sumCount[] og de n/k første kolonnene i allCount[][] ,
  tråd$_1$ får de neste n/k elementene i sumCount[] og kolonnene i allCount[][] ,…, osv.
- Hver tråd$_i$ summerer så tallene i alle sine kolonner 'j' fra allCount[0..antTråder-1][j] til sumCount[j].
- Alle trådene synkroniserer på nytt på den sykliske barrieren '*synk*'.

Del 2 – om automatisk parallellisering

Drømmen om lage automatisk parallellisering

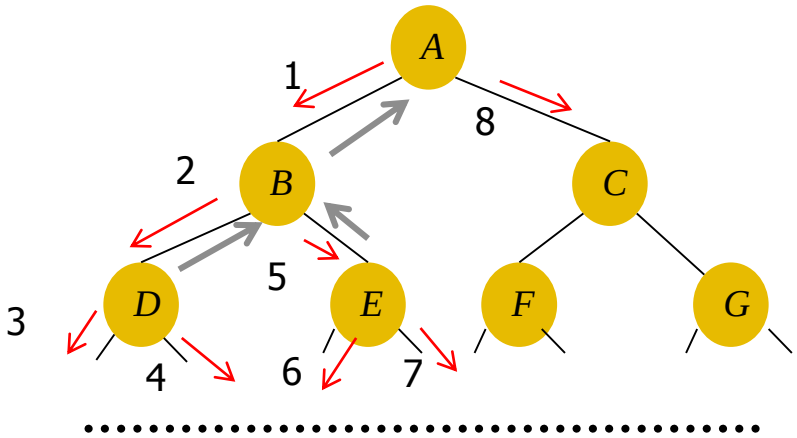
- Parallellisering gir lang og det er 'vanskelig' å debugge kode
- Det finnes særlig to typer av sekvensielle programmer som tar 'lang' tid:
  - A) Med løkker (enkle, doble,..)
  - B) Rekursive
- Drømmen er man bare helt automatisk, eller bare med noen få kommandoer kan oversette et sekvensielt program til et parallelt.
  - Med løkker hadde vi bl.a HPFortran (Fortran90) som parallelliserte løkker (slo ikke helt an)
  - Intel har laget en rekke slike systemer (se neste foil)
  - Rekursive metoder – vi skal se på PRP (et system jeg har fått lager som hovedfagsoppgaver flere ganger siden 1994)

Noen av Intels parallelliserings-biblioteker i C, /C++ og Fortran

- Intel Math Kernel Library
  - Ferdig parallelliserte metoder basert på MPI
- Cilk Plus
  - parallellisering av særlig løkker
  - Implementerer tre kompilator direktiver *cilk_spawn, cilk_sync* og *cilk_for* for
- Open MP
  - en overbygning over MPI med en rekke kompilator-direktiver for parallellisere blokker av koden, særlig for-løkker (har en felles hukommelses-modell, men litt for lett å gjøre data-r).
- MPI
  - Standard bibliotek for meldingsutveksling mellom prosesser (uten felles hukommelse). Brukes bla. I kurset INF3380 for grid-programmering
- Pthreads
  - I Linux og C. Som tråder i Java – full kontroll over parallelliteten og da ikke 'automatisk parallellisering'.

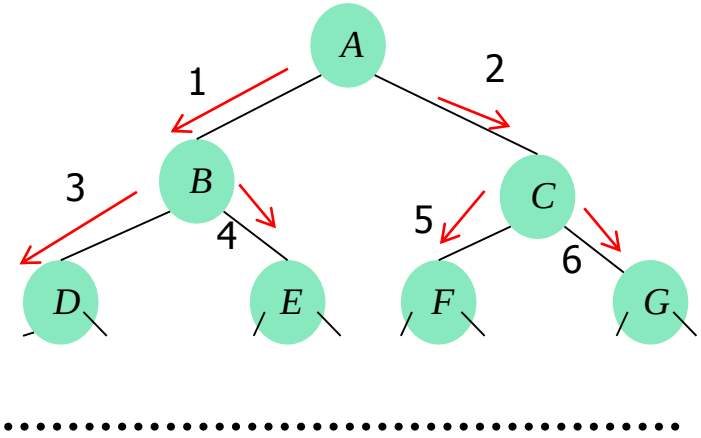
PRP: Den grunnleggende ideen:
– Rekursjonstreet og treet av tråder er 'like', men:

Rekursjon



Dybde først

Tråder

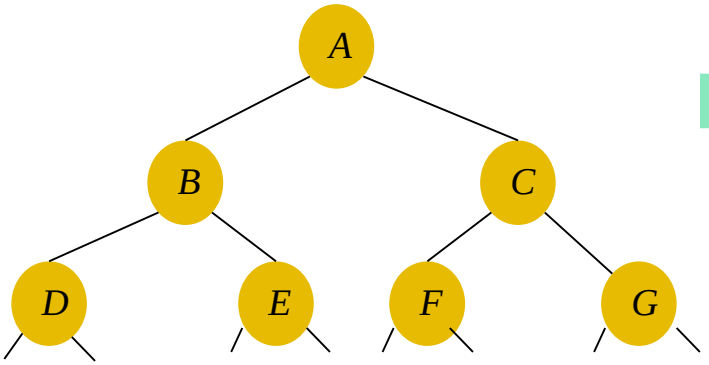


Bredde først

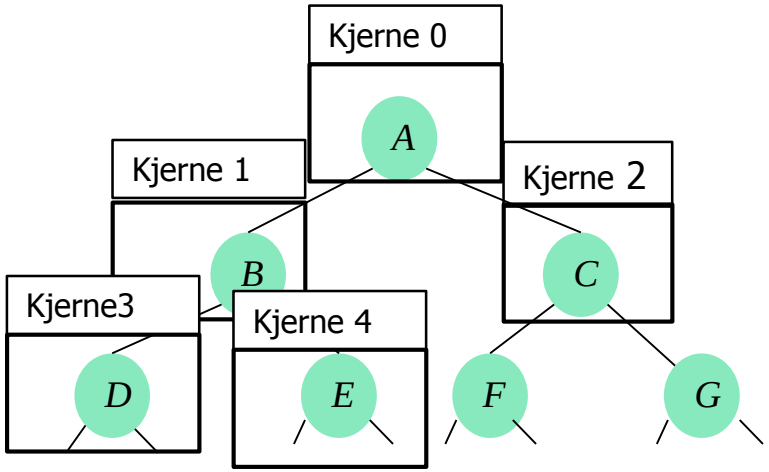
Tråder:
- Kan ikke bruke 'svaret' fra venstre-tråd til å lage parametere for høyre-tråd
- Venter på trådene (og 'svaret') først når begge trådene er sendt ut..

Fra rekursjon til parallelle prosedyrer (metoder)

Rekursjon



Multikjerne CPU



Løst beskrevet, den grunnleggende PRP-ideen:

- Vi omformer alle rekursive metoder slik:
  - Vi 'stjeler' parametrene til de rekursive kallene.
  - Koden frem til de rekursive kallene blir en egen metode:
    - Istedenfor å gjøre det rekursive kallet, kaller vi en metode som 'stjeler' alle parametrene til det rekursive kallet - disse parametrene er da en parallell arbeids-pakke.
    - Vi kjører denne metoden bredde først **til vi har nok** parameter-pakker.
  - En ny metode i 'hver sin tråd' : kalles med en slik parameter-pakke
    - Blir en egen metode i en egen tråd som får en slik parameterpakke , utfører beregningene vanlig sekvensielt og rekursivt , og returnerer svaret. Svarene henges opp i en liste.
    - og når alle disse vanlige metodene har returnert sine svar, utføres resten av metodene i de metodekallene hvor vi 'stjal' parametrene – nedenfra og opp i treet.
  - Vi har altså 3 versjoner av den rekursive metoden:
    - Den hvor vi stjeler parametrene
    - Den som henter et svar istedenfor å utføre kallet
    - Den vanlige rekursive metoden uendret

Hvordan erstatte to rekursive kall med to 'svarpakker'

```
int [] rek (int [] a, int left, int right) {
        int  [] ret;
        if ( right – left < LIM ) {
                ret = <enkel løsning>;
        }  else {
             int deling = partition (a, left,right);
               <mer kode..>
                 int [] v =rek (a,left,deling-1);
                 int [] h =rek (a,deling, right);
                //sett sammen svar
                ret = settSammen (v,h):
         }
         return ret;
}
```

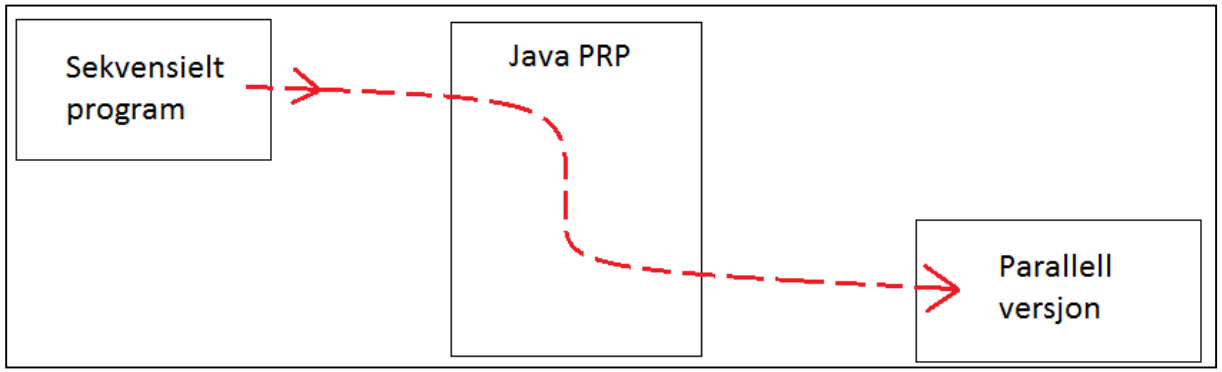
Del A:
Utføres fram til de rekursive kallene i ny metode

Del B:
Erstattes med to kall på en metode som 'stjeler (kopierer) parameterne og henger de opp i en LIFO-liste (stack).

Del C:
Erstattes med en ny metode med to tilordninger: til int [] v og int [] h fra to svarpakker. Deretter utføres resten av koden i en egen ny metode med 'vanlig' retur som henger opp returen som en svarpakke i en FIFO-liste

I Uke 9 og idag så vi på å overføre Rekursjon til tråder - her fra Peter Eidsviks masteroppgave

- Vi skal nå automatisere det
- Vi lager en preprosessor: javaPRP
  - dvs. et Java-program som leser et annet Java-program og omformer det til et annet, gyldig Java-program (som er det parallelliserte programmet med tråder)



- For at JavaPRP skal kunne gjøre dette, må vi legge inn visse kommentarer i koden:
  - Hvor er den rekursive metoden
  - Hvor er de rekursive kallene
- Bare rekursive metoder med to eller flere kall, kan parallelliseres .

Et eksempel før mer 'teori' med en kjørbar sekvensiell Quicksort

```
import java.util.Random;

class QuicksortProg{
    public static void main(String[] args){
        int len = Integer.parseInt(args[0]);
        int [] tid = new int[11];
        for(int i = 0; i < 11; i++){
            int[] arr = new int[len];
            Random r = new Random();
            for(int j = 0; j < arr.length; j++){
                arr[j] = r.nextInt(len-1);
            }
            long start = System.nanoTime();
        int[] k =
         new QuicksortCalc().quicksort (arr,0,arr.length-1);
            long timeTakenNS = System.nanoTime() - start;
            tid[i] = (int) timeTakenNS/1000000.0;
            System.out.println(timeTakenNS/1000000.0);
        }
        tid = QuicksortCalc.insertSort(tid,0,10);
        System.out.println("Median sorteringstid for 11
                    gjennomlop:"+tid[5]+"ms. for n="+len);
    }
}
```

```
class QuicksortCalc{
    int INSERT_LIM = 48;
    int[] quicksort (int[] a, int left, int right){
        if (right-left < INSERT_LIM){
            return insertSort(a,left,right);
        }else{
            int pivotValue = a[(left + right) / 2];
            swap(a, (left + right) / 2, right);
            int index = left;

            for (int i = left; i < right; i++) {
                if (a[i] <= pivotValue) {
                    swap(a, i, index);
                    index++;
                }
            }
            swap(a, index, right);
            int index2 = index;
            while(index2 > left && a[index2] == pivotValue){
                index2--;
            }
           a = quicksort (a, left,  index2);
           a = quicksort (a, index + 1, right);
            return a;
    }   }
```

Nesten helt vanlig QuickSort – vi har riktignok pakket den inn i en klasse
Vi kompilerer og kjører den og tar tiden (11 ganger)

QuickSort av 10 mill tall (ca. 0.72 sek) sekvensielt

```
M:\INF2440Para\Powerpoint\Uke10\PRP>java QuicksortProg
10000000
745.816677 ms
721.98455 ms
717.246485 ms
714.245199 ms
723.833991 ms
Median sorteringstid for 5 gjennomlop:721ms. for n=10000000
```

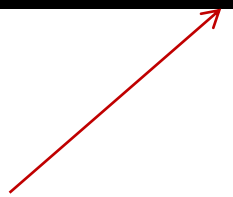
Nå legger vi til tre kommentarer så den kan preprosessereres over i en parallell versjon (/*REC*/ og /*FUNC*/):

```
import java.util.Random;

class QuicksortProg{
    public static void main(String[] args){
        int len = Integer.parseInt(args[0]);
        int [] tid = new int[11];
        for(int i = 0; i < 11; i++){
        for(int i = 0; i < 11; i++){
            int[] arr = new int[len];
            Random r = new Random();
            for(int j = 0; j < arr.length; j++){
                arr[j] = r.nextInt(len-1);
            }
            long start = System.nanoTime();
            int[] k =
             new QuicksortCalc().quicksort(arr,0,arr.length-1);
            long timeTakenNS = System.nanoTime() - start;
             tid[i] = (int) timeTakenNS/1000000;
            System.out.println(timeTakenNS/1000000.0);
        }
        tid = QuicksortCalc.insertSort(tid,0,10);
        System.out.println("Median sorteringstid for 11
                    gjennomlop:"+tid[5]+"ms. for n="+len);
    }
}
```

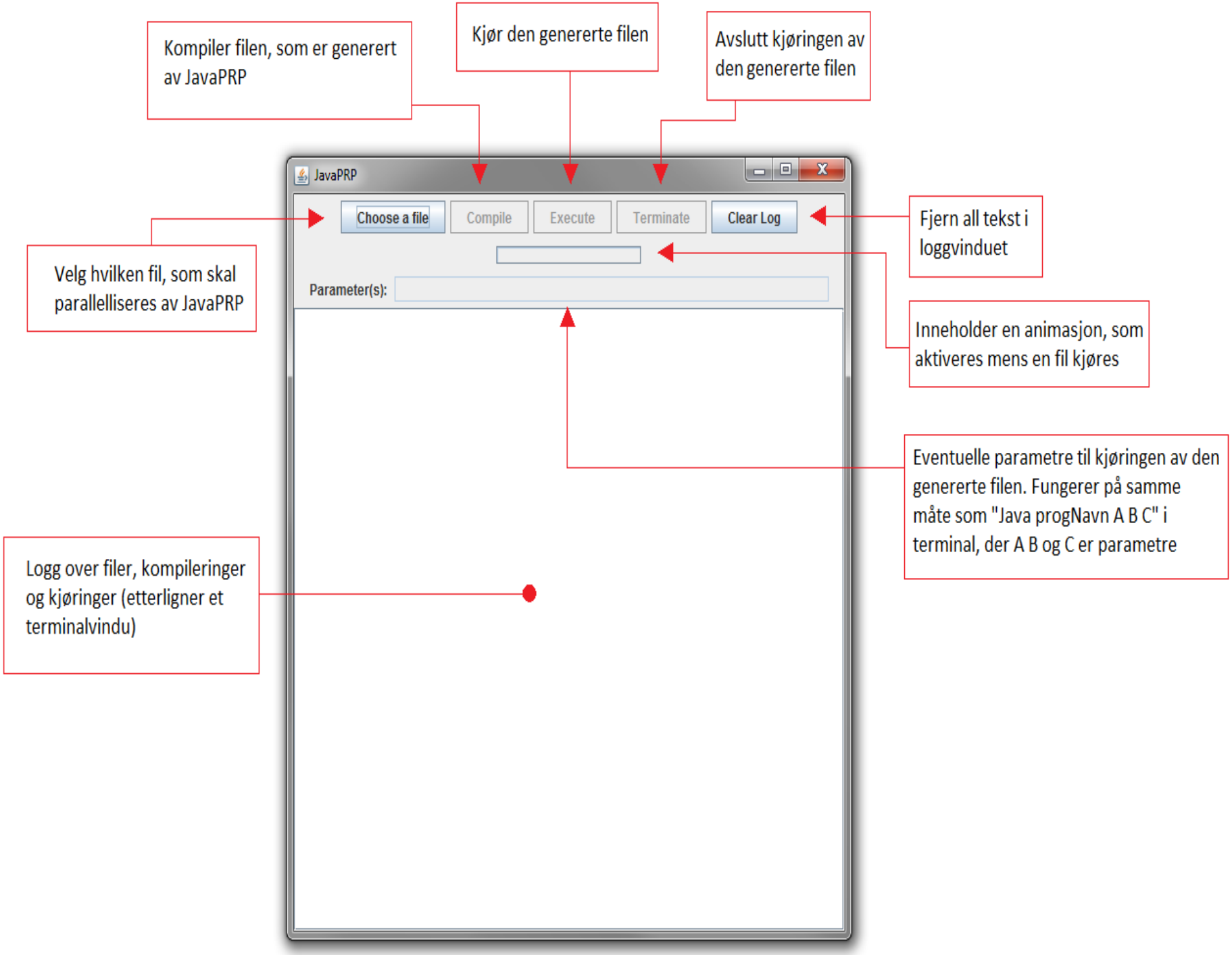
```
class QuicksortCalc{
    int INSERT_LIM = 48;
    /*FUNC*/
    int[] quicksort(int[] a, int left, int right){
        if (right-left < INSERT_LIM){
            return insertSort(a,left,right);
        }else{
            int pivotValue = a[(left + right) / 2];
            swap(a, (left + right) / 2, right);
            int index = left;

            for (int i = left; i < right; i++) {
                if (a[i] <= pivotValue) {
                    swap(a, i, index);
                    index++;
                }
            }
            swap(a, index, right);
            int index2 = index;
            while(index2 > left && a[index2] == pivotValue){
                index2--;
            }

            /*REC*/
            a = quicksort(a, left,  index2);
            /*REC*/
            a = quicksort(a, index + 1, right);
            return a;
        }
    }
```

Kompilér JavaPRP -systemet, så start det

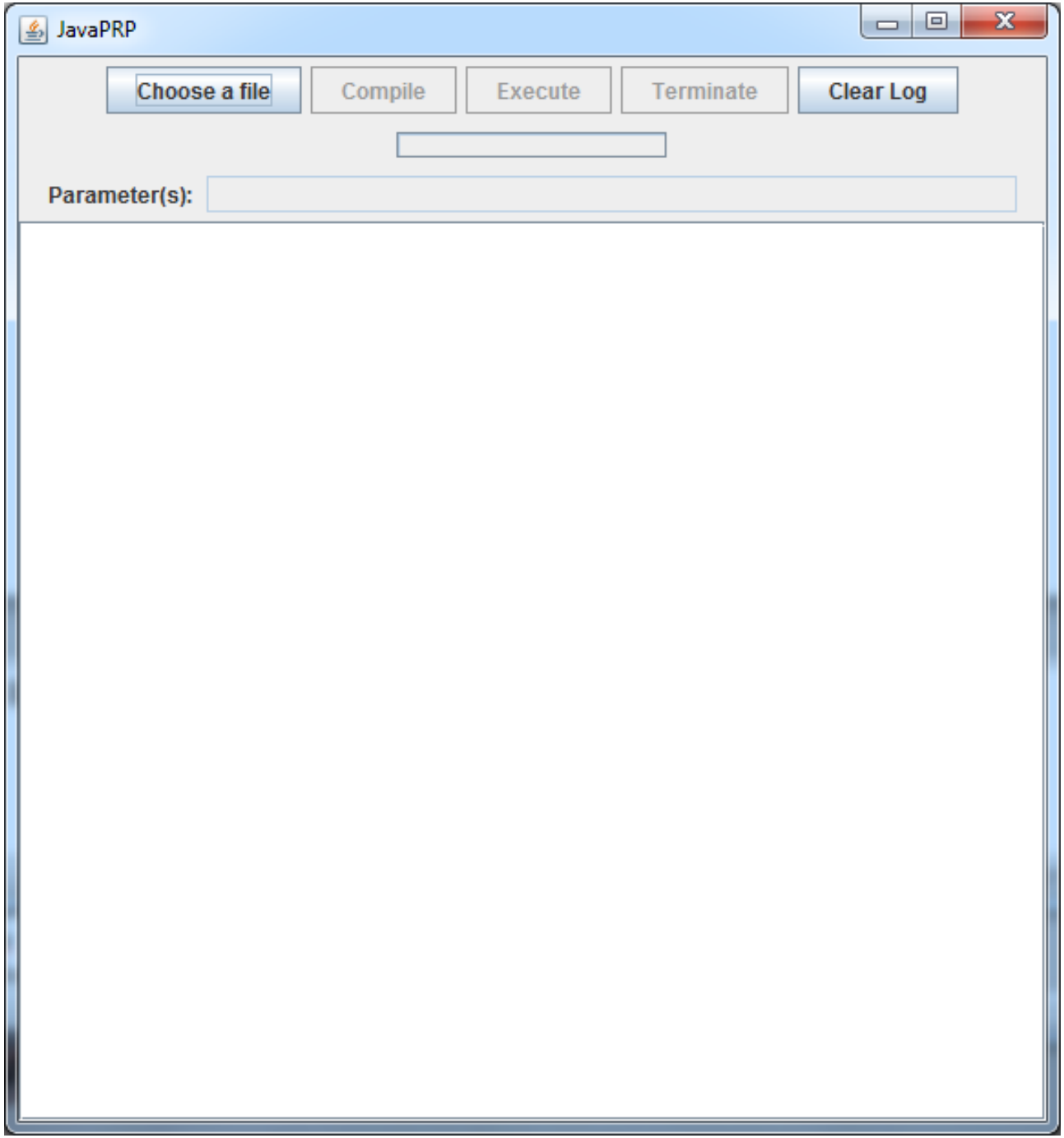
```
M:\INF2440Para\PRP>javac JavaPRP.java

M:\INF2440Para\PRP>java JavaPRP
```

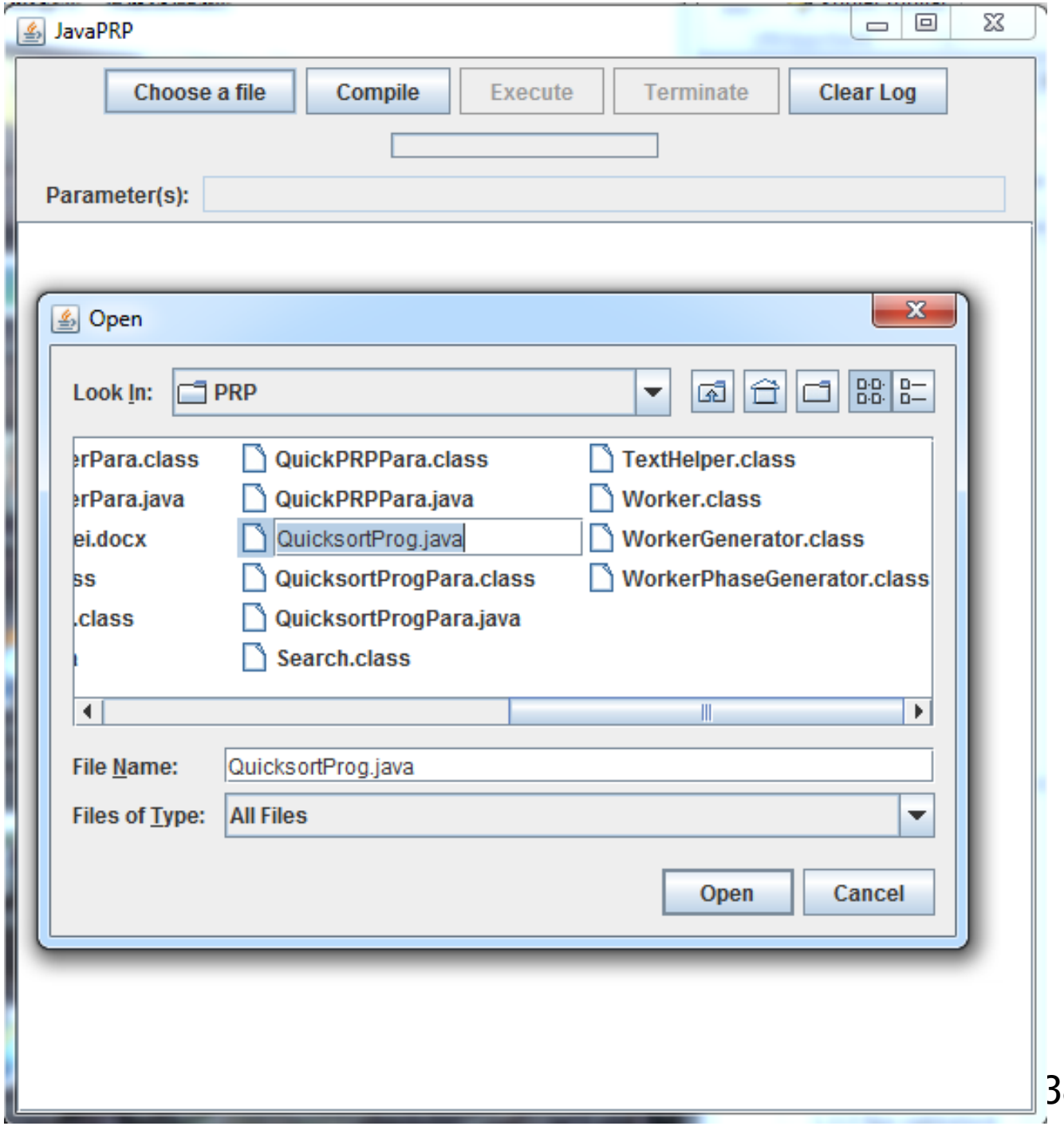


Det starter et GUI-interface


Kompiler filen, som er generert av JavaPRP
Kjør den genererte filen
Avslutt kjøringen av den genererte filen
JavaPRP
Choose a file
Compile
Execute
Terminate
Clear Log
Parameter(s):
Velg hvilken fil, som skal parallelliseres av JavaPRP
Fjern all tekst i loggvinduet
Inneholder en animasjon, som aktiveres mens en fil kjøres
Eventuelle parametre til kjøringen av den genererte filen. Fungerer på samme måte som "Java progNavn A B C" i terminal, der A B og C er parametre
Logg over filer, kompileringer og kjøringer (etterligner et terminalvindu)

Trykker: Choose a file

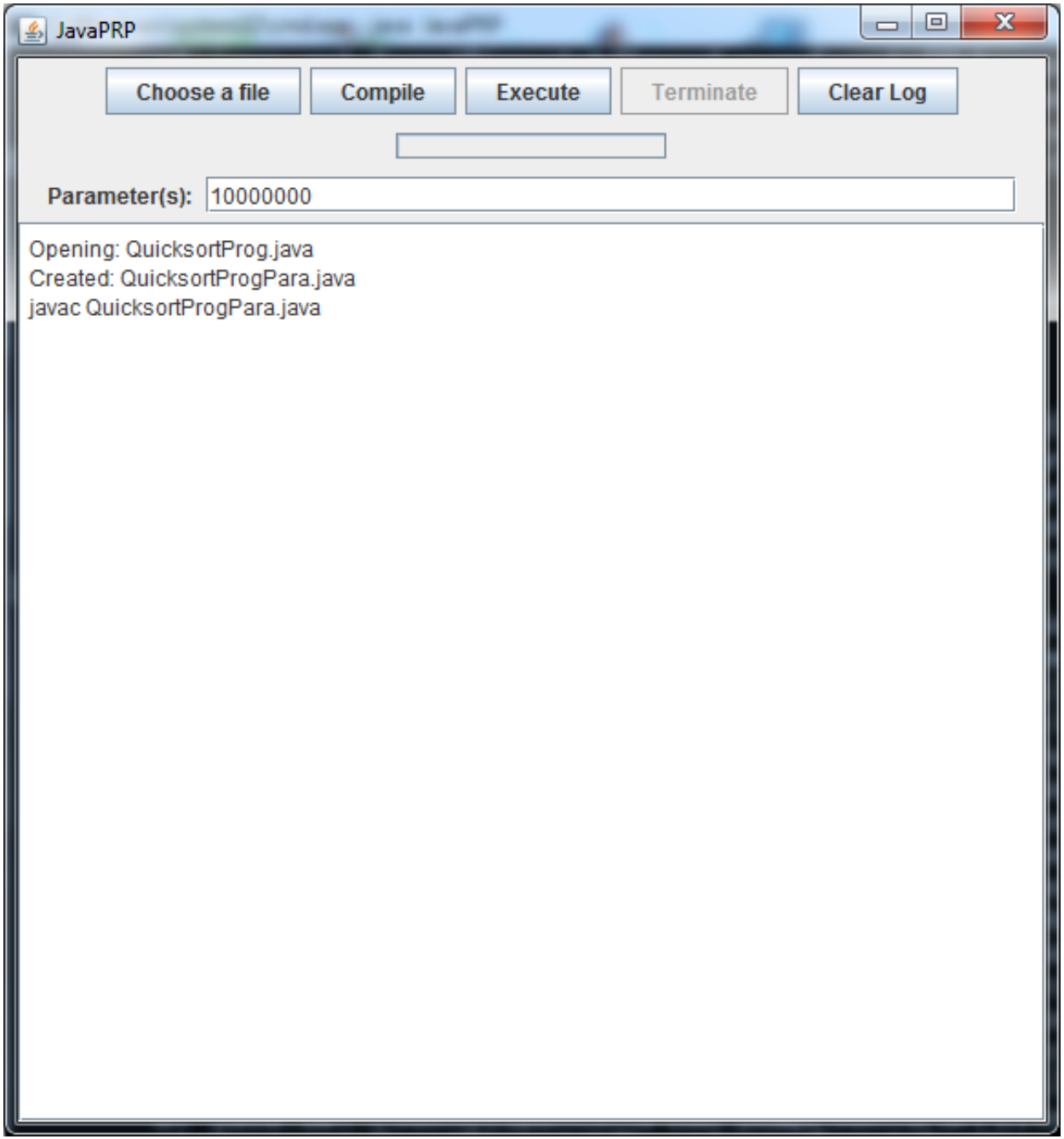



JavaPRP
Choose a file
Compile
Execute
Terminate
Clear Log
Parameter(s):
Open
Look In:
PRP
erPara.class
erPara.java
ei.docx
ss
.class
QuickPRPPara.class
QuickPRPPara.java
QuicksortProg.java
QuicksortProgPara.class
QuicksortProgPara.java
Search.class
TextHelper.class
Worker.class
WorkerGenerator.class
WorkerPhaseGenerator.class
File Name:
QuicksortProg.java
Files of Type:
All Files
Open
Cancel

Trykket så Compile:

- Kompilerte da den parallelliserte filen: QuicksortProg**Para**.java

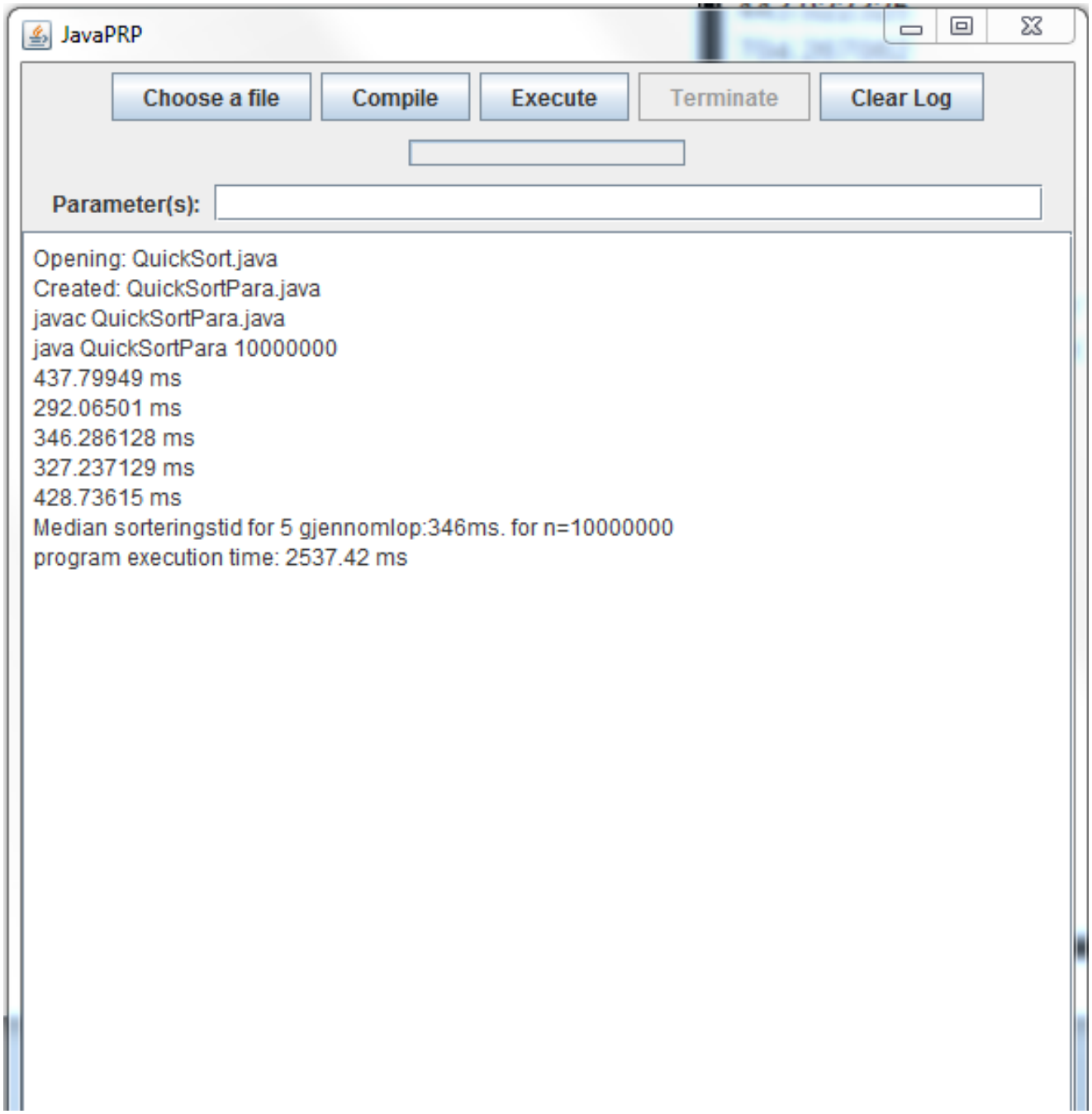
- Legger så inn parameter (10 mill) på kommandolinja og velger Execute.



Resultatet kommer i log-vinduet.
-en OK speedup for den parallelle utførelsen, n= 10m

Speedup= 721/346= 2,08

Sammenligning med Arrays.sort: time: 777ms



Oversettelsen : Quicksort

```
import java.util.Random;

class QuicksortProg{
    public static void main(String[] args){
        int len = Integer.parseInt(args[0]);

        for(int i = 0; i < 11; i++){
            int[] arr = new int[len];
            Random r = new Random();
            for(int j = 0; j < arr.length; j++){
                arr[j] = r.nextInt(len-1);
            }
            long start = System.nanoTime();
            int[] k =
           new QuicksortCalc().quicksort (arr,0,arr.length-
            long timeTakenNS = System.nanoTime() - start
            System.out.println(timeTakenNS/100000.0);

        }
    }
}
```

Starten på brukerens kode (77 linjer)

```
class QuicksortProgPara{
        public static void main(String[] args){
                new Admin(args);
        }
}

class Admin{
    public Admin(String[] args){
        initiateParallel(args);
    }

    void initiateParallel(String[] args){
        int len = Integer.parseInt(args[0]);

        for(int i = 0; i < 11; i++){
        int[] arr = new int[len];
        Random r = new Random();
        for(int j = 0; j < arr.length; j++){
            arr[j] = r.nextInt(len-1);
        }
        long start = System.nanoTime();
        int[] k  = startThreads (arr,0,arr.length-1);
        long timeTakenNS = System.nanoTime() - start
        System.out.println(timeTakenNS/100000.0);
    }
}
```

Starten på oversatt kode (245 linjer)

Dette kan også kjøres delvis linjeorientert

Anta at din sekvensielle rekursive løsning (som er annotert for PRP) heter 'MittProg.java'

1) Hvis du ønsker det, kjør ditt egent program og notere eksekveringstiden.

```
>javac MittProg.java
>java MittProg 1000000
```

2) Kompilér PRP-systemet (hvis du ikke har gjort det tidligere)

```
>javac javaPRP.java
```

3) Oversett ditt program (MittProg.java) til et parallelt program (MittProgPara.java) – dette må gjøres via GUI

```
>java javaPRP – velg da MittProg.java og trykk Compile
```

4) Kjør det genererte parallelle programmet (MittProgPara.java)

```
> java MittProgPara 1000000
```

PRP kan også kan parallelliseres et fasedelt program

- Et fasedelt PRP-program har flere faser som hver består av først en parallell rekursiv del og så en sekvensiell del (kan sløyfes).
- Da må brukeren beskrive det i en egen ADMIN –metode:

```
/*ADMIN*/
public int minAdminMetode(...){
    int svar = rekursivMetode1(...);
    sekvensiellKode1();
    svar = rekursivMetode2(...);
    sekvensiellKode2(...);
    svar = rekursivMetode3(...);
    sekvensiellKode3(...);
    svar = rekursivMetode4(...);
    sekvensiellKode4(...);
    return svar;
}
```

- Innfører da to nye Kommentarkoder: /*ADMIN*/ og **/*FUNC 1*/ , /*FUNC 2*/,... osv**

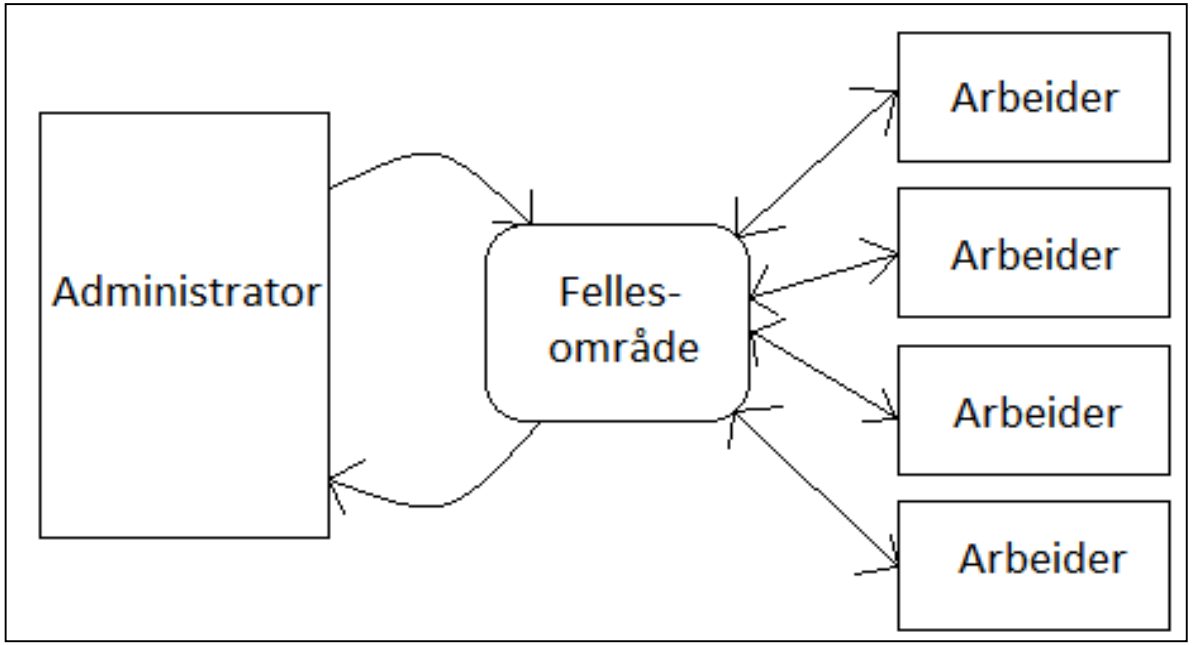
Og hver av disse rekursive metodene er i hver sin klasse.

```
class MittFaseProgram{
   public static void main(String[] args){
    <returverdi> svar =
            new MittFaseProgram().minAdminMetode(...);
   }

   /*ADMIN*/
   <returverdi> minAdminMetode(...){
      <returverdi> svar1 = new Fase1().rekursivMetode(...);
      sekvensiellKode1(...);
      <returverdi> svar2 = new Fase2().rekursivMetode(...);
      sekvensiellKode2(...);
      return ...;
   }

   void sekvensiellKode1(...){
    }

   void sekvensiellKode2(...){
   }
} // end MittFaseProgram
```

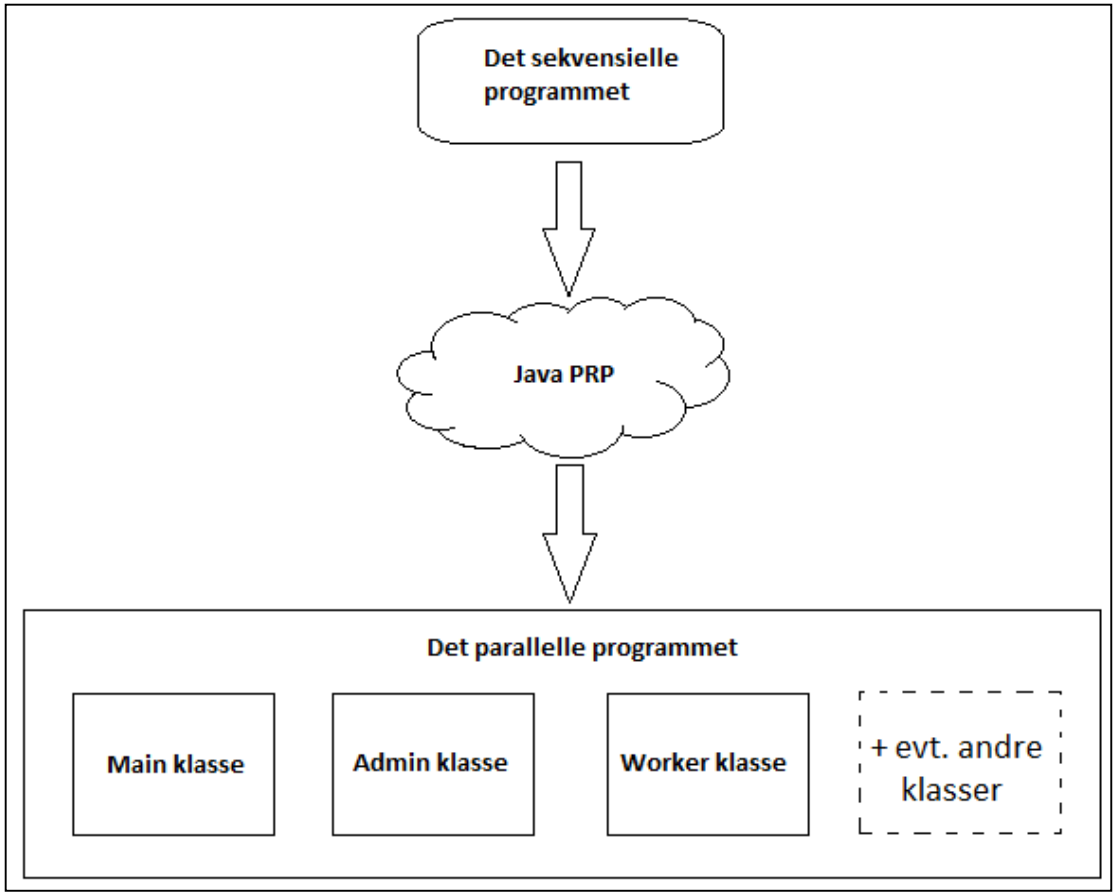
```
class Fase1{
   /*FUNC 1*/
    <returverdi> rekursivMetode(...){
      /*REC*/
      <returverdi> svar1 = rekursivMetode(...);
      /*REC*/
      <returverdi> svar2 = rekursivMetode(...);
      return ...;
   }

}

class Fase2{
   /*FUNC 2*/
 <returverdi> rekursivMetode(...){
      /*REC*/
      <returverdi> svar1 = rekursivMetode(...);
      /*REC*/
      <returverdi> svar2 = rekursivMetode(...);
      /*REC*/
      <returverdi> svar3 = rekursivMetode(...);
      return ...;
   }
}
```

Hvordan gjøres dette? Administrator – arbeider modell

- Oppgaver legges ut i et fellesområde
- Arbeiderne tar oppgaver og legger svar tilbake i fellesområdet



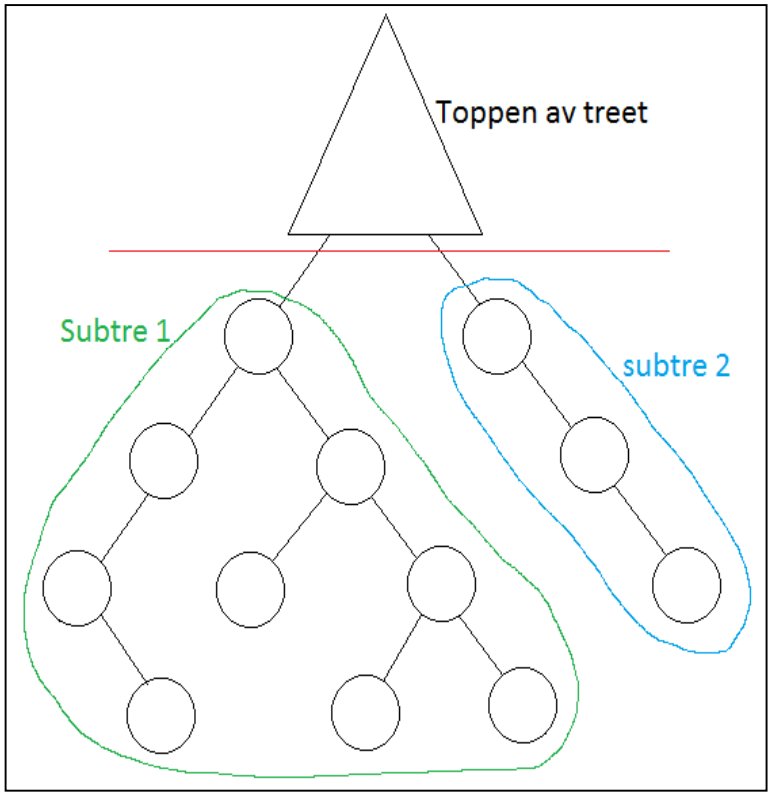
Fra sekvensielt program til parallelt:



Figur 5: *Fra det sekvensielle programmet, gjennom Java PRP og til det parallelle resultatet. Det nye, parallelle programmet vil inneholde en klasse for main, Admin, Worker pluss eventuelt andre klasser fra det sekvensielle programmet, som ikke er en del av parallelliseringen.*

Eksempel: Parallellisering med to tråder

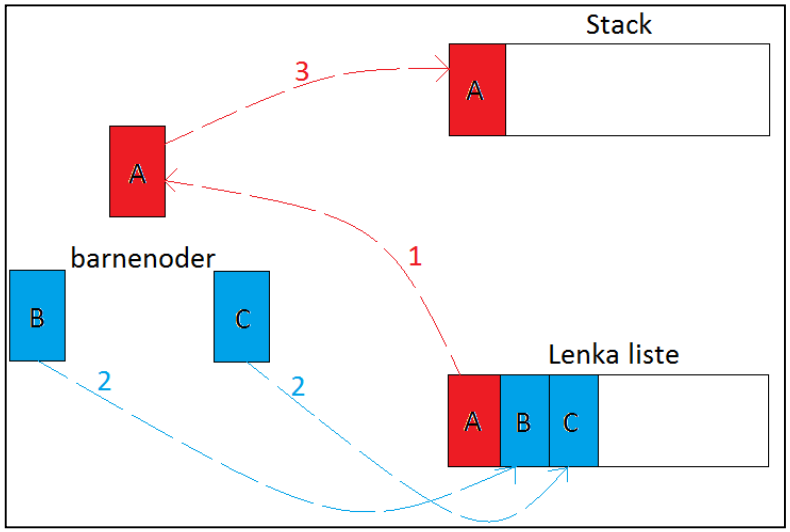
PRP: Toppen kjøres med tråder bredde-først, så går hver tråd over til dybde først og 'vanlige' rekursive kall



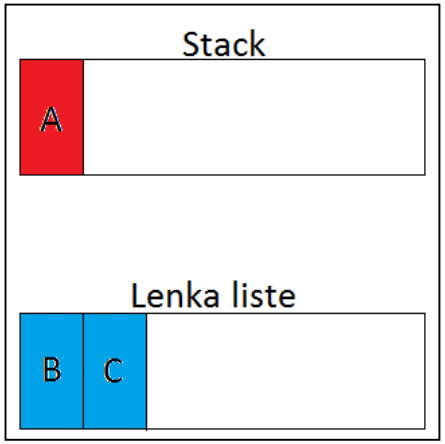
Figur 6: *Visualisering av treet der vi ønsker å parallellisere med to tråder. Toppen av treet tilsvarer bredde først traversering til vi ender på, i dette tilfelle, to subtrær. Disse to subtrærne vil traverseres dybde først*

Fra bredde først til dybde først og så vanlig rekursjon

- Trådene blir når de lages, lagt i en Lenket Liste (FIFO-kø)
- Så tas den første(A) i køen ut, og den lager to nye barne-tråder (B,C) som legges i lista.
- A legges i en stack (LIFO-kø)
- Neste på i Lista (B) tas ut og dens nye barne-tråder (D,E) legges inn i Lista
- B legges på stacken,… osv.
- Bunnen av bredde-først ligger da på toppen av stacken



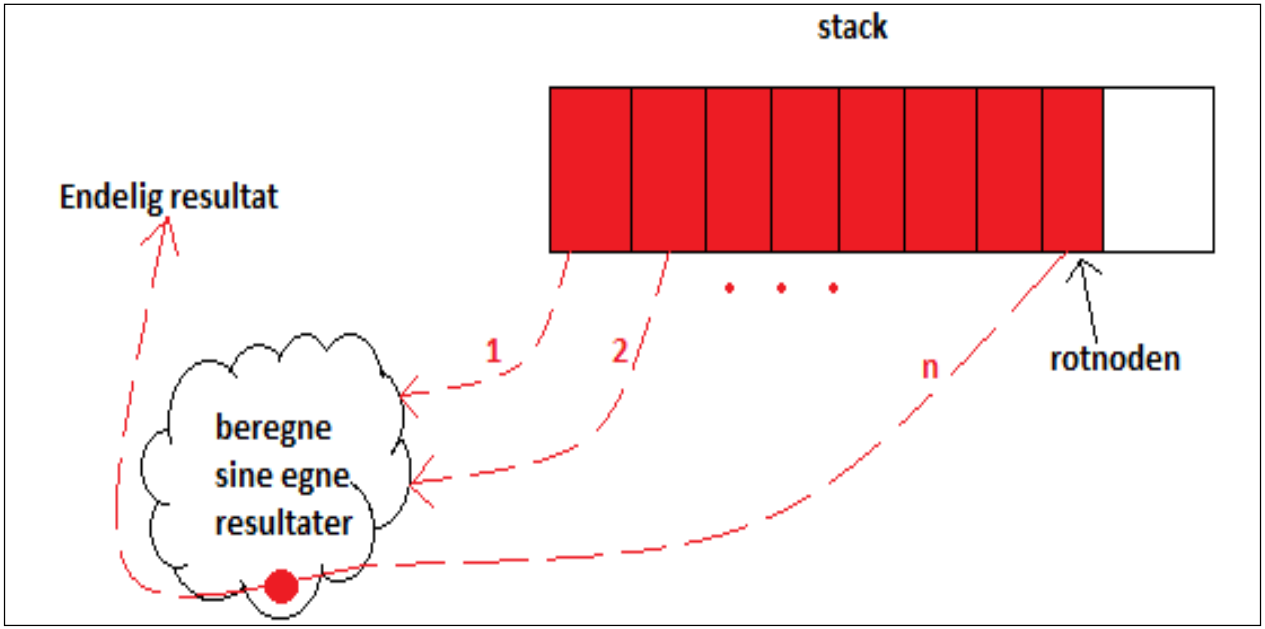
Figur 7: *Administratoren oppretter datastrukturen til arbeiderne.*



Figur 8: *Datastrukturen etter at Figur 7 er ferdig.*

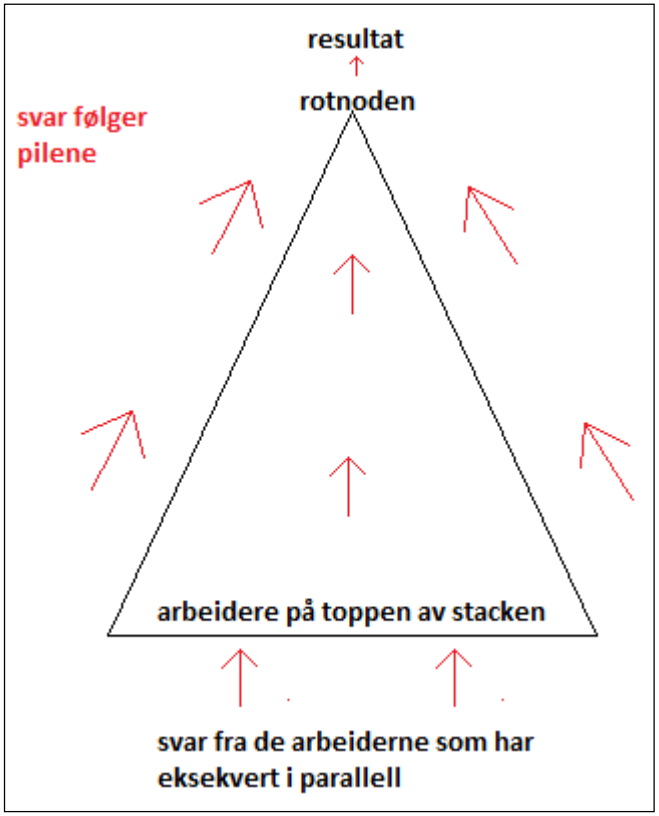
Kjøring og retur av verdier

- Når trådene er brukt opp, pop-es stacken (f.eks øverst er E) og vanlige rekursive kall gjøres fra E , neste pop-es …osv. til stacken er tom
- Svarene fra ethvert element som tas av stacken legges i en tabell og plukkes opp av den som kalte elementet.
- Den som kalte, kan så fortsette sin kode og selv returnere sitt svar,..



Figur 9: *Arbeiderstacken beregner seg nedenfra og opp til roten.*

Svarene genereres nederst stacken og svarene propaganderer oppover til første kall på den rekursive metoden



Hvorfor dette med bredde-og dybde-først ?

- Kunne vi ikke bare startet trådene og latt de alle gå i parallell?
- NEI – fordi:
  - Vi har lovet rekursiv semantikk (virkemåte) i den parallelle.
  - Vi skal derfor oversette det rekursive programmet slik at det gir alltid samme resultat parallelt.
- Eks Quicksort:
  - Anta at vi parallelliserer ned til nivå 3 i treet
  - Hvis nivå 2 og 3 går samtidig, vil dette gå galt fordi de prøver begge lagene å flytte på de samme elementene i a[].

Hva gjøres teknisk – vår program består av (minst) to klasser

- Klassen med 'main' og en klasse som inneholder den rekursive metoden
  - Er det flere rekursive metoder som skal parallelliseres, så skal de være inne i hver sin klasse
- Det genererte programmet består av minst følgende klasser:
  - Admin
  - Worker
- Kort og greit:
  - Det oversatte programmet 'xxxxxPara.java' skal vi egentlig ikke se på og spesielt ikke endre.
  - Det bare virker og parallelliserer etter visse prinsipper.

Eksempel 2: Største tall i en array

```
class LargestNumber{
    public static void main(String[] args){
        int len = Integer.parseInt(args[0]);
        int cores =
              Runtime.getRuntime().availableProcessors();
        int[] arr = new int[len];
        Random r = new Random();
        for(int i = 0; i < arr.length; i++){
            arr[i] = r.nextInt(len-1);
        }
         long t = System.nanoTime();
        /*CALL*/
        int k = (new
        Search()).findLargest(arr,0,arr.length);
        Double t2 =(System.nanoTime()-t)/1000000.0;
        System.out.println("Largest number is " + k+
                                  ", paa:"+t2+"ms.");    }
}
```

```
class Search{
    int k = 5;

    /*FUNC*/
    int findLargest(int[] arr, int start, int end){

        if((end-start) < k){
            return largest_baseCase(arr,start,end);
        }
        int half = (end-start) / 2;
        int mid = start + half;
        /*REC*/
        int leftVal = findLargest (arr,start,mid);
        /*REC*/
        int rightVal = findLargest (arr,mid+1,end);
        if(leftVal > rightVal) return leftVal;
        return rightVal;

    }

    int largest_baseCase(int[] arr, int start, int end){
        int largest = 0;
        for(int i = start; i < end; i++){
            if(arr[i] > largest){
                largest = arr[i];
            }
        }
        return largest;
    }
}
```

NB. for å få kjøretiden riktig for det parallelle programmet

- N.B det som oppgis som «program execution time» er med overhead fra GUI løsningen. Fra GUI-en:

```
Opening: LargestNumber.java
Created: LargestNumberPara.java
javac LargestNumberPara.java
java LargestNumberPara 100000000
Largest number is 99999994, paa:152.171677ms.
program execution time: 1511.89 ms
```

- Kjør det i linjemodus (n= 100 mill.):

```
M:\INF2440Para\PRP>java LargestNumberPara 100000000
Largest number is 99999998, paa:135.922687ms.

M:\INF2440Para\PRP>java LargestNumber 100000000
Largest number is 99999998, paa:417.98199ms.
```

Hva så vi på i Uke10

- **Del 1**: Om Oblig 4 – Parallell Radix og generelt om parallellisering

- **Del 2:** Automatisk parallellisering av rekursjon
- PRP- Parallel Recursive Procedures
  - Nåværende løsning (Java, multicore CPU, felles hukommelse) – implementasjon: Peter L. Eidsvik
- Demo av to eksempler kjøring
- Hvordan virker en kompilator (preprosessor) for automatisk parallellisering
  - Prinsipper (bredde-først og dybde-først traversering av r-treet)
  - Datastruktur og parametre til metodene
  - Eksekvering
- Krav til et program som skal bruke PRP