{Arne Maus, 23. mai 2011}

Kapittel 19

MER OM PARALLELLE PROGRAMMER I JAVA OG KVIKKSORT

Parallell programmering er vanskelig, og det er derfor utviklet flere synkroniseringsmåter og biblioteker for mer strukturert parallell programmering i Java. Vi skal her gjennomgå bruk av en særlig nyttig synkroniseringsmetodikk; bruk av barrieresynkronisering, men starter med en gjennomgang av hvorfor vi trenger å programmere mer parallelt, og særlig hvorfor det så komplekst. Dels skyldes kompleksiteten maskinens konstruksjon, og dels programmeringsspråk som Java og hvordan de oversettes i maskinen. Etter denne gjennomgangen kan vi lett få inntrykk av at det er umulig å få til parallelle programmer riktig. Det er ikke tilfellet, men for å få det til må man følge noen klare og enkle regler. Bryter man bare én av disse, vil det kunne gå veldig galt.

19.1 HVORFOR LAGE PARALLELLE PROGRAMMER MED TRÅDER?

Det er i hovedsak tre grunner til at vi kan ønske å ha parallellitet i et program:

1. Man skiller ut visse aktiviteter som går *langsommere* i en egen tråd, som tegning av grafikk på skjermen eller søk i en database. Resten av programmet kan da fortsette uten opphold.

2. Logikken i programmet er slik at det naturlig består av en rekke uavhengige aktiviteter som bare sjelden trenger å bruke felles data. Hvis hver slik aktivitet programmeres med hver sin tråd, blir programmet faktisk ofte *lettere* å skrive. Et godt eksempel er at du lager et system for direkte salg av flybilletter (eller innlevering av oppgaver i et kurs i programmering). Siden mange samtidig skal kunne gjøre dette, skriver du programmet slik at en tråd snakker bare med én kunde, og betjener bare henne/ham. Det er relativt lett. Dersom flere «kunder» melder seg,

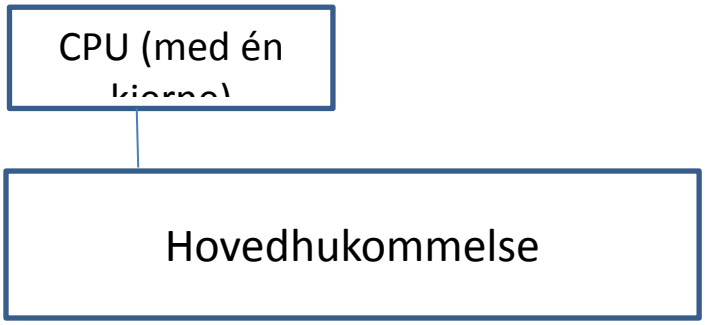
starter vi bare en ny slik tråd for hver ny kunde. Vi ser at av og til må disse ha adgang til felles data, som for eksempel de ledige plassene på en bestemt flyavgang, og da må vi synkronisere trådene og sørge for at bare én får tilgang til å endre felles data av gangen. Feil som kan oppstå da må vi håndtere, men jevnt over kan disse trådene operere i full parallell.

3. Vi ønsker å bruke parallelliteten til å få visse beregninger til å gå *raskere*! Eksempler kan være store ingeniørberegninger, generering av bilder i spillgrafikk eller, som det eksemplet vi til sist skal se på, sortering av større datamengder.

Vi skal i det følgende basere oss på oppgaver av type 3, at vi ønsker et raskere program, men mesteparten av det vi skriver kan også brukes direkte i de to andre tilfellene.

19.2 Maskinens konstruksjon, cache og multikjerne

Før 1980 var datamaskiner relativt enkle, slik det framstilles i figur 19.1. Man hadde en beregningsenhet, også kalt CPU (Central Prosessing Unit). Den utførte én instruksjon av gangen i den rekkefølgen de var spesifisert i programmet, og leste og skrev sine data direkte i hovedhukommelsen.



Figur 19.1 *Skisse av en datamaskin fra ca. 1980 hvor det bare var én beregningsenhet, en CPU, som leste sine instruksjoner og både skrev og leste data (variabler) direkte i hovedhukommelsen.*

Fra 1980-tallet begynte imidlertid CPU-ene å gå mye raskere enn hovedhukommelsen. Ordbruken skiftet også, slik at vi nå snakker om en prosessor istedenfor CPU. Dagens avanserte prosessorer bruker om lag 150–300 ganger så lang tid på å skrive til, eller lese fra, hovedhukommelsen som den tiden det tar å utføre en enkel instruksjon (som å legge sammen to heltall). Her ville det bli mye dødtid.

Svaret var å legge hurtigere, men dyrere, såkalte cache-hukommelser mellom prosessoren og hovedlageret. Først en, senere flere, nå er det vanlig med tre. Når prosessoren «tror» at den lagrer verdien av en variabel i hovedhukommelsen, lagres den først bare i nivå 1-cachen (se figur 19.2), og prosessoren kan fortsette. Så snart som mulig lagres så disse dataene i nivå 2-cachen, så i nivå 3-cachen – og til sist i hovedhukommelsen om lag 200 klokkeenheter senere (1 klokkeenhet= ca. ½ milliardtedels sekund). Tilsvarende gjelder for lesning. Hvis prosessoren ikke finner de opplysningene den vil ha i noen av cachene, må de leses fra hovedhukommelsen og inn i alle cachene før kjernen får adgang til data.

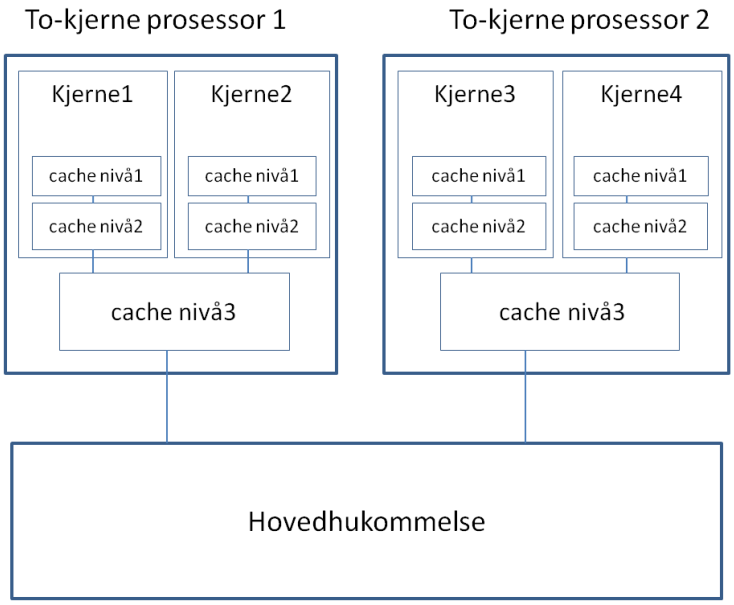
En annen viktig utvikling av prosessorene kom av at man etter ca. år 2000 ikke greide å få en prosessor til å gå fortere. Prøvde man med en raskere klokke, ville den rett og slett først feile og ev. smelte. Imidlertid greide man stadig å lage hver prosessor mindre og mindre lik transistorene den besto av. Hva skulle så databrikkeprodusentene som Intel og AMD gjøre? De la flere prosessorer, heretter kalt prosessorkjerner eller bare kjerner, på hver brikke. Vi fikk da maskiner med to prosessorkjerner (dual-core), så med fire kjerner, osv.

Det er uklart hvor dette ender, men det er i alle fall satt i gang forsøksproduksjon av brikker med ca. 100 slike prosessorkjerner, og det er helt sikkert at utviklingen ikke stopper med det. I tillegg finnes maskiner hvor man har satt 2 eller 4 slike multikjerneprosessorer på samme hovedhukommelse. Dette er ikke uvanlig eller spesielt dyrt. I tillegg kan noen av disse kjernene kjøre 2 tråder hver i parallell; såkalt hyperthreading, fordi en del av elektronikken er duplisert i hver kjerne. Eksemplene i dette kapitlet er testet på 2 slike maskiner, en med 8

kjerner (= 1 prosessor med 4 kjerner som hver har hyperthreading) og en med 64 kjerner (= 4 prosessorer med 8 kjerner som hver har hyperthreading).

Det er også slik at hver kjerne har sin egen nivå 1- og nivå 2-cache, men deler som oftest nivå 3-cachen med alle kjernene på samme brikke, men ikke med de andre prosessorene som ev. er i maskinen. Vi husker fra forrige kapittel at alle trådene i vårt program deler samme område i hovedhukommelsen. En viktig konklusjon på dette er at selv om en tråd som går på én av kjernene har skrevet ned verdien av en variabel som alle trådene har utsyn til, så kan det ta lang tid før de andre trådene greier å se denne nye verdien, f.eks. fordi den holder på å bli skrevet ned via alle cachene, og det kan ta flere hundre klokkesykler før den oppdaterte verdien er regisrert i hovedlageret. Typiske verdier for hvor lang tid det tar å skrive/lese i de ulike cachene er 3 klokkesykler for nivå 1, 11 klokkesykler for nivå 2 og 39 sykler for nivå 3, mens det er ca. 200 klokkesykler for lesing eller skriving i hovedhukommelsen.

I programeksemplet i figur 19.1 vil vi illustrere flere viktige poeng i parallellprogrammering. Først må vi vite at operasjonen **tall++** ikke blir utført som én operasjon, men egentlig er tre operasjoner: Først les verdien av **tall**, så legg 1 til denne verdien, og til sist skrives den nye verdien ned i variabelen **tall**. Som vi vet, betyr dette at både den gamle og nye verdien går via nivå 1-cachen, og veien ned til hovedhukommelsen er lang. Vitsen med disse cachene er at selve beregningsenheten bare forholder seg til sin nivå 1-cache, leser og skriver i den, mens resten av systemet stadig forsøker å holde de andre cachene og hovedhukommelsen oppdatert. Beregningsenheten går altså så fort som nivå 1-cachen greier å lese og skrive data, nesten hundre ganger raskere enn hovedhukommelsen hvis den ikke må vente på data fra en av de langsommere cachene eller hovedhukommelsen.



Figur 19.2 *To dobbeltkjerneprosessorer i én maskin. Når det går parallelle tråder på hver av disse kjernene, ser vi at de som oftest får informasjon om ulik verdi på en felles variabel (eks.* `int i`*) i hovedlageret, hvis noen av trådene leser på en slik variabel samtidig som en annen tråd endrer dens verdi ved skriving. Dette fordi de ulike cachene ikke hele tiden er fullt oppdatert på siste verdi som er skrevet. Samtidig lesing* ***og*** *skriving på en variabel av mer enn én tråd* ***må*** *derfor alltid synkroniseres.*

Man kan jo til slutt spørre hvorfor man har alle disse lagene med hukommelse. Siden man greier å lage en rask nivå 1-cache, hvorfor kunne ikke all hukommelse vært slik? Poenget er at slike cacher er laget på en mye dyrere måte enn hovedhukommelsen, de tar større plass og krever mer strøm. Det ville kort sagt ikke lønne seg å lage all hukommelse slik.

19.3 OM BRUK AV SYNKRONISERINGSPRIMITIVER OG LÅSER

Vi har sett at det oppstår store problemer når ulike tråder vil skrive nye verdier inn i samme variabel – ulike tråder ser ulike verdier. Det er faktisk også vanskelig å avgjøre for én tråd når en annen parallell tråd er ferdig.

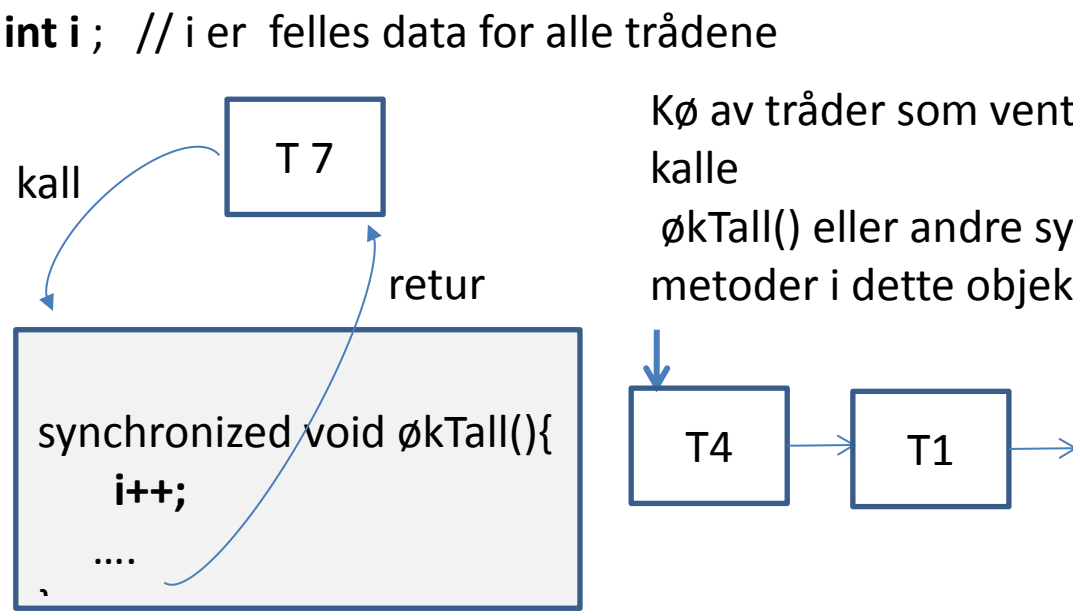
Til å løse disse problemene har man innført flere typer av låser i Java. En lås er en mekanisme som kan stoppe en tråd eller la den vente. Tråden kaller på låsen. Låsen gjennomfører en test og avgjør om kallende tråd må vente eller ikke. Felles for låsene i Java er følgende gode egenskap:

> Når flere tråder gjør et kall på *samme lås,* vil alle disse trådene være sikret at all skriving trådene har gjort på felles variabler *før dette kallet* er synlig for alle de andre etter kallet. De ser da samme verdier på felles variabler.

Vi vil illustrere dette med et program som benytter to typer av låser: **`CyclicBarrier`** som sikrer at vi vet når alle trådene er ferdige, og Javas innebygde **`synchronized`** metoder som skal sikre at bare én tråd av gangen er inne og teller opp en felles variabel, heltallet **`tall`**. Alle andre tråder som kaller metoden **`økTall()`** ser den riktige verdien (slik at den også kan telle den opp og legge til 1). Mer om hver av disse to synkroniseringsmekanismene nedenfor.

19.4 OM SYNKRONISERTE METODER

Synkroniserte metoder bruker låsen til det objektet de tilhører, og sørger for at hvis mer enn én tråd kaller den, vil bare én slippe til av gangen og de andre må vente i en kø (figur 19.3). Kallende tråd får gjort seg helt ferdig, og resultatet blir skrevet ned i hovedhukommelsen før neste tråd får utføre sitt kall.



Figur 19.3 *Synkroniserte metoder lar én tråd slippe til av gangen og køer opp andre tråder som ev. samtidig ønsker å gjøre et kall på synkroniserte metoder i objektet. Her er tråd T7 opptatt med et kall på økTall(), og mange andre tråder som har forsøkt å gjøre kall i mellomtiden, må vente i en kø. Når T7 er ferdig, slipper en av de andre løs fra køen og kan gjøre sitt kall, osv.*

Merk at hvis det er flere synkroniserte metoder i det samme objektet, vil denne låsen i objektet også sperre for samtidige kall fra andre tråder på disse metodene. I ett objekt kan altså høyst én synkronisert metode bli eksekvert av gangen. Merk at hvis man har flere objekter av den klassen hvor de synkroniserte metodene er, vil det være mulig å få utført en synkronisert metode samtidig av to eller flere tråder, en utilsiktet feil hvis kallene kommer til ulike objekter av denne klassen.

Vi kan først prøve å kjøre programmet i figur 19.1. med alle de 4 ulike definisjonene av metoden **`økTall().`** Først spør den systemet om antall kjerner i maskinen og skriver det ut. Deretter lager den et objekt av klassen **`Parallell`**, og så leser den inn fra kommandolinja hvor mange tråder den skal starte, og hvor mange ganger hver av dem skal øke heltallet tall med 1. Vi

bruker her en long, et 64 bits heltall, for variabelen **`tall`** fordi summen av antall opptellinger (**`antGanger*antTråder`**) kan bli større enn største verdi for et 32 bits heltall. Vi kaller så metoden **`utfør()`** i dette objektet p av klassen **`Parallell`**.

Merk at:

Hvert objekt som skapes har en lås. Det er den låsen som nyttes av alle synkroniserte metoder i det *samme* objektet som metoden er i når man kaller dem. En synkronisert metode kan bare låse ute et annet kall på denne metoden hvis de to metodekallene nytter den samme låsen – dvs. det samme objektet.

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

/** Start >java Parallell <ant tråder> <ant ganger> */
class Parallell{
 long tall=0;      // Trådene summerer i denne
 CyclicBarrier b ;    // sikrer at alle er ferdige
 long antTråder, antGanger ; //Etter summering: riktig
               //svar er:antTråder*antGanger

 void utskrift(double tid) {
  System.out.println("Tid "+antGanger+" kall * "+
  antTråder+" Traader ="+Format. align(tid,9,6)+
  " sek,\n sum:"+ tall +", tap:"+
  (antTråder*antGanger -tall)+" = "+
  Format.align( (antTråder*antGanger - tall)*
  100.0/(antTråder*antGanger),5,1)+"%");
 }

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

 public static void main (String [] args) {
     int antKj =
Runtime.getRuntime().availableProcessors();
   System.out.println("Maskinen har "+antKj+ "
kjerner.");
```

```
   Parallell p = new Parallell();
     p.antTråder = Integer.parseInt(args[0]);
     p.antGanger = Integer.parseInt(args[1]);
     p.utfør();
 }

 void utfør () {
  b = new CyclicBarrier((int)antTråder+1); //også main
  long t = System.nanoTime();       //start klokke
  for (int i = 0; i< antTråder; i++)
        new Thread(new Para()).start();

  try{ // main tråden venter
       b.await();
     } catch (Exception e) {return;}

  double tid = (System.nanoTime()-t)/1000000000.0;
  utskrift(tid);
 }

 class Para implements Runnable{
     public void run() {
      for (int i = 0; i< antGanger; i++) {
        økTall();
      }
      try { // wait on all other threads + main
           b.await();
      } catch (Exception e) {return;}
     }

  //  void økTall() { tall++;}          // 3)
  //  synchronized void økTall(){ tall++;}   // 4)
 }
}
```

Program 19.1 *Et program som viser både riktig og gal bruk av låser i Java. Vi ser 4 ulike plasseringer av en metode økTall() – kommentert med 1), 2), 3) og 4). Bare 1 er riktig. Inndata fra kommandolinja er hvor mange tråder vi vil starte, og hvor mange ganger hver av disse trådene vil telle opp en felles variabel (long-tall) i klassen Parallell. Hvis man fjerner kommentarmarkeringen // for én av kallene på økTall(), vil programmet kunne kompileres og kjøre. Bare én av disse plasseringene er riktig. Alle de feilaktige plasseringene av metoden «økTall()» vil som sluttresultat få en altfor liten samlet sum i tall. Kjører vi et feilaktig program*

flere ganger med samme parametrer, vil det også nesten alltid gi ulike svar; typisk for synkroniseringsfeil.

Vi kan lett gjøre den feilen at vi skaper flere objekter, og dermed flere låser. En tråd som kaller en synkronisert metode, vil låse med den låsen som er i det objektet som utfører metoden. Program 19.2 viser et eksempel på en slik feil. Hvis vi «av-kommenterer» plassering 4) og benytter den, ser vi at vi får mange feil når vi kjører programmet (med 1000 eller flere oppdateringer). Grunnen til dette er at vi har laget en lås for hvert av tråd-objektene av klassen Para. Nå vil de synkroniserte variablene som er definert på denne måten, låse bare de kallene som nytter samme lås. Men siden hver tråd har sitt objekt og sin lås, vil ingen av trådene greie å låse ute de andre trådene – fordi de bruker hver sin lås.

	1	**2**	**3**
Svar **uten** synchronized **3)**	7 112 531	5 911 630	6 169 492
Svar **med** synchronized **1)**	10 000 000	10 000 000	10 000 000

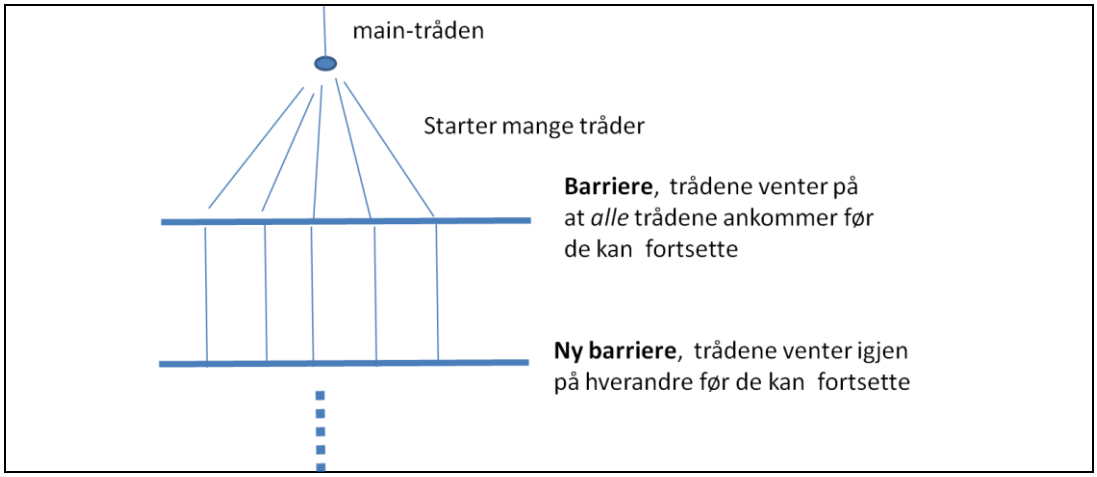
Tabell 19.1 *Tre kjøringer hvor vi starter 100 tråder som hver forsøker å øke* **`i`** *100 000 ganger med 1, dvs.* **`i`** *skulle da bli =* 10 mill*. Vi ser at dersom vi ikke synkroniserer økTall() riktig, får vi mange feil og ulikt svar i hver kjøring.*

Hvis vi derimot bruker plassering 1) for den synkroniserte metoden **`økTall()`**, ser vi at den er inne i et objekt av klassen **`Parallell`**. Det er bare dannet ett objekt av denne klassen, og alle kall på **`økTall()`** bruker da samme lås. Alt går derfor bra uansett hvor mange tråder og kall på **`økTall()`** vi har.

Plasseringene 2) og 3) går også galt fordi det ikke brukes noen låser, tilsvarende som plassering 4), med mange opptellinger som går tapt. Alt fra 0,001 % til over 90 % av summen mangler. Resultatene varierer også fra kjøring til kjøring med samme parametrer. Grunnen til at vi ikke alltid får tapte oppdateringer hvis vi har få kall på `økTall()` per tråd, er at hver tråd vi da starter, greier å gjøre seg ferdig før neste tråd starter. Vi får altså ikke et skikkelig parallelt program, for trådene utføres etter hverandre, sekvensielt.

19.5 BARRIERESYNKRONISERING

Ikke alle beregninger kan løses greit eller effektivt med synkroniserte metoder. Mange beregninger kan parallelliseres ved at man deler dem opp i flere trinn. Hvert trinn gjøres i parallell av et antall tråder, men *alle* trådene må være ferdig med ett trinn før beregningene i neste trinn kan begynne. Da kan alle trådene fortsette med del to av beregningen, og da vet de at de kan lese hva de andre trådene skrev i forrige trinn av beregningen. Kanskje er det mange slike trinn i beregningene. Et viktig spesialtilfelle er at vi ikke trenger å dele opp selve beregningen i flere trinn, men heller ønsker at hovedtråden, dvs. den tråden som programmet starter med i main, skal vente til alle trådene den har startet er ferdige med beregningene. Først da kan hovedtråden presentere resultatet til brukeren.



Figur 19.4 *Programmet starter alltid først bare én tråd – main-tråden. Den lager et objekt b av klassen CyclicBarrier og et større antall tråder. Hvis barrieren er laget for k tråder, så vil alle tråder, også ev. main-tråden, vente hvis de sier b.await() inntil tråd nr. k sier b.await(). Da slippes alle de k trådene løs og kjører videre.*

For å få til en slik ventemekanisme for k stk. tråder lager vi et objekt av klassen CyclicBarrier, og parameteren er antallet tråder som den skal stille i kø. Når den siste melder seg, skal alle trådene igjen slippes løs.

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

CyclicBarrier b = new CyclicBarrier (antTråder);

< I trådene vil vi finne følgende kode når vi skal
vente i «b» på at alle de andre trådene også er
ferdige med beregningene sine:>

try {
  b.await();
} catch (Exception e) {….}
```

Program 19.2 *Et program som skisserer riktig bruk av CyclicBarrier i Java.*

Vi vil i neste programeksempel 19.3 se at vi har en parameter som er 1 større enn antall tråder vi lager, fordi vi benytter den sykliske barrieren b til å få alle trådene *og* main-tråden til å vente på hverandre. Grunnen til at det heter CyclicBarrier, er at når så mange tråder som den er spesifisert for har ventet og blitt sluppet fri, kan den uten videre motta *det samme antallet* tråder til ny runde med venting og frislipping uten ny initialisering (den er straks gjenbrukbar). Hvis trådene har flere trinn hvor de må vente på hverandre, har vi gjerne to CyclicBarrier – én som settes opp med antTråder og som trådene bruker seg

imellom, og én som initieres med antTråder + 1, som trådene venter på når de er helt ferdige **og** som main-tråden også venter på etter at den startet alle de andre trådene. I main-tråden vet vi da at når den slipper løs, har alle trådene blitt ferdige med koden sin.

Husk at her gjelder også det første punktet om synkronisering. Det å kalle på await() i en barriere sørger ikke bare for at alle venter, men også at alle etterpå kan se alt hva de andre skrev på felles variabler før await()-kallet.

19.6 ET PROGRAM SOM BRUKER CYCLICBARRIER OG BEREGNER MAX-VERDIEN I EN ARRAY

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

/** Start >java  FinnMax2 <ant tråder> */
class FinnMax2{
 int[] a,lokalMax; //finn max verdi i a[]
 CyclicBarrier b; // sikrer at alle er ferdige før vi
          // tar tid og sum
 static int antTråder, ant;

 public static void main (String [] args) {
  antTråder = Integer.parseInt(args[0]);
  new FinnMax2().utfør();
 }

 void utfør () {
  a= new int[antTråder*antTråder]; //lag stort problem
  ant= a.length/antTråder;  // ant elementer per tråd
  lokalMax = new int [antTråder];
  Random r = new Random(1337);
   for (int i =0; i< a.length;i++) {
    a[i] = Math.max(r.nextInt(a.length)-i,0);
   }
   b = new CyclicBarrier((int)antTråder+1); //også main
   int totalMax = -1;
   long t = System.nanoTime();    // start klokke
   for (int i = 0; i< antTråder; i++) {
     new Thread(new Para(i)).start();
   }

   try{ // main venter på Barrieren b
    b.await();
   } catch (Exception e) {return;}

   // finn den største max fra alle trådene
   for (int i=0;i < antTråder;i++)
    if(lokalMax[i] > totalMax) totalMax = lokalMax[i];

   System.out.println("Max verdi parallell:"+totalMax +
```

```
    ", paa: "+((double)(System.nanoTime()-t)/1000000.0)+
    " millisek.");

   // sammenlign med sekvensiell utføring av finnMax
   t = System.nanoTime();
   totalMax = 0;
   for (int i=0;i < a.length;i++)
    if(a[i] > totalMax) totalMax = a[i];

   System.out.println("Max sekvensiel:"+totalMax +
   ", paa: "+((double)(System.nanoTime()-t)/1000000.0)+
   " millisek.");
 } // end utfør

 class Para implements Runnable{
  int ind, minMax = -1;
  Para(int i) { ind =i;} // konstruktor

  public void run() { // Det som kjøres i parallell:
   for (int i = 0; i< ant; i++) {
    if (a[ant*ind+i] > minMax) minMax = a[ant*ind+i];
   }
   lokalMax[ind] =minMax; // levér svar

   try { // wait on all other threads + main
    b.await();
   } catch (Exception e) {return;}
  }
 } // end Para

}// end FinnMax2
```

Program 19.3 *Et program som viser riktig bruk av CyclicBarrier i Java. Parameter til programmet er antall tråder, og det lages et array a[] som er antTråder*antTråder lang med tilfeldig positivt innhold. Vi starter så antTråder i parallell som finner maksimalverdien i hver sin del av a[], og tråd nr. i legger sitt svar inn i lokalMax[i]. Hovedprogrammet som venter på den sykliske barrieren kan, når alle trådene er ferdige, selv gå gjennom lokalMax[] og finne totalMax-verdien. Vi skriver ut denne og tidsforbruket. Som en sjekk går vi så sekvensielt gjennom a[] og skriver ut den max-verdien vi da finner, og tidsforbruket for sekvensiell gjennomgang til sammenligning.*

Vi ser at dette er et program som greit parallelliserer beregningen av max-verdien i en array, men vi ser også av tidene som programmet skriver ut, at den parallelle beregningen tar ca. 50–100 ganger så lang tid som bare å lese gjennom arrayen sekvensielt fra start til slutt for beregningen av maksverdien. Dette

eksemplet lærer oss forhåpentlig bruk av en syklisk barriere, men også at parallellisering av svært enkle oppgaver hvor vi bare ser på hvert dataelement én eneste gang i beregningene, har liten hensikt. Å starte og stoppe tråder tar da langt lengre tid enn selve beregningene. Vi skal nå se på et problem, sortering, hvor parallellisering lønner seg, i alle fall hvis vi skal sortere mer enn 100 000 tall.

19.7 GENERELT OM PARALLELLISERING AV ALGORITMER

Her er en skisse av de stegene vi vanligvis foretar når vi lager en parallell algoritme for å løse ett problem.

1. Start med et vel testet sekvensielt program som løser problemet.

2. Del opp problemet i flere mindre deler som kan løses hver for seg. Vanligvis vil man forsøke å dele dataene i like store deler, ofte langt flere enn antall kjerner – f.eks. 20* antKjerner. Grunnen til dette er at en tråd som løser et slikt delproblem kan generere ventesituasjoner, og da er det greit at en annen tråd er i stand til å eksekvere. Imidlertid må det advares mot altfor mange tråder. Det tar tross alt noen få millisekunder å skape og starte en tråd. Vi kan godt bruke 10–500 tråder for å løse et problem, men ikke mange titusener.

3. Start en tråd for hver av disse delene av problemet.
Dette gjør vi hvis ikke oppdeling og starting av tråder er avhengig av hvilke, og hvor mye, data vi har. Da kan vi komme ut for en kombinasjon av oppsplitting av problemet og start av tråder samtidig under eksekvering – pkt. 2 og 3 kombinert.

4. La hver tråd løse en slik del.
Vanligvis vil enten den samme eller en lett modifisert versjon av den sekvensielle algoritmen bli benyttet for hver slik del. Ofte erstattes da rekursjon med tråder. Felles data som flere tråder skriver på samtidig, beskyttes med synkroniserte metoder.

5. Vent til alle trådene er ferdige – f.eks. med en syklisk barriere.

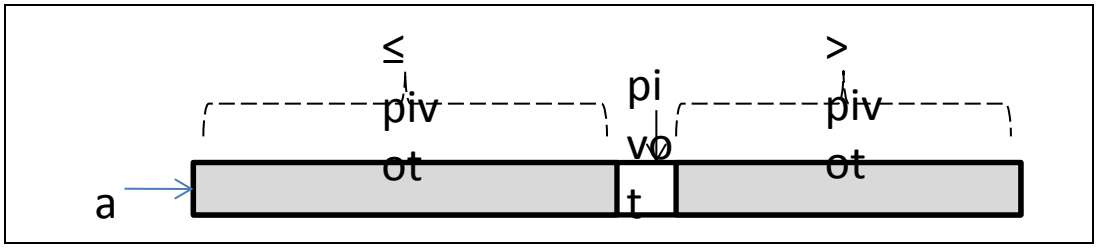
6. Kombiner delsvarene til en løsning på hele problemet. Av og til er det ikke nødvendig, men ofte vil det være noen avsluttende beregninger.

Pkt. 2, 4 og delvis 6 er de vanskeligste og vil kunne variere fra problem til problem, pkt. 3 og 5 går greit.

19.7 KVIKKSORT, EN REKURSIV SORTERINGSALGORITME

Som nevnt ovenfor er en av de gylne reglene i parallellprogrammering at før man skriver et parallelt program, lager man et godt testet sekvensielt program som løser problemet. Parallellisering er endringer til det sekvensielle programmet.

Vi skal først presentere en sekvensiell sorteringsmetode Kvikksort (eng. Quicksort), laget av Tony Hoare i 1962. Den fikk en enkel utforming av Nico Lamuto som vi nytter her. Ideen er enkel: Velg ut et element i arrayen, kalt pivot-elementet. Bytt om elementene i arrayen slik at alle elementer som er ≤ pivot kommer til venstre for pivot, deretter kommer pivot, og så alle elementene som er > enn pivot – jf. figur 19.5. Arrayen a er da ikke ferdig sortert, men hvis vi nå for hver av de to delene foretar samme type oppsplitting med nye valg av pivoter, så kan de igjen oppsplittes gjentatte ganger til alle de ulike delene til slutt har en lengde på 1 eller 0. Da er a[] sortert fordi hvert element står til høyre for et annet element som er mindre eller lik dette.



Figur 19.5 *Ideen bak Kvikksort. Vi velger først et vilkårlig element, pivot, så bytter vi om på elementene i a[] slik at vi får de som er mindre eller lik pivot til venstre for pivot, så pivot selv, så alle de som er større enn pivot til høyre. Dette*

gjentas på hver av de to delene, på hver av disses to deler igjen, osv. til a[] er sortert.

Her er skjelettet til programmet som foretar denne sorteringen med oppsplitting gjentatte ganger, og som måler tiden og skriver ut denne:

```
import easyIO.*;
import java.util.*;

/** Sekvensiell implementasjon av Quicksort */
class SeqQuick {
 int [] a;

 /** bytter om a[i] og a[j] */
 void bytt(int [] a, int i, int j) {..}

 /** Innpakning for for sQuick - enklere kall */
 void sQuick(int [] a) { sQuick(a, 0,a.length-1);}

 /** Rekursiv, sekvensiell QuickSort av a[lav..høy] */
 void sQuick (int [] a, int lav, int høy) {..  }

 /** Konstruktor, fyller a[0..n-1] med tilfeldige tall */
 SeqQuick(int n) {.. }

 /** Tar tider, kaller sQuick og gjør en enkel test */
 void utførOgTest() {.. }

 public static void main (String [] args) {
  new SeqQuick(Integer.parseInt(args[0])).utførOgTest();
 }
} //end SeqQuick
```

Program 19.4 *Skjelettet for programmet som starter i main, og først lager et objekt av klassen SeqQuick med kall på konstruktøren og deretter kaller utførOgTest.*

Koden for konstruktøren og utførOgTest:

```
 SeqQuick(int n) {
```

```
  a = new int [n];
  Random r = new Random(1337157);
  for (int i =0; i< a.length;i++)
    a[i] = r.nextInt(a.length);   // random fill >=0
 }// end konstruktør

 void utførOgTest( ) {
  long t = System.nanoTime();    // start klokke
  sQuick(a);
  t = System.nanoTime()-t;
  System.out.println("Sekvensiell QSort av "+a.length
   +" tall paa:"+double)(t)/1000000.0)+ " millisek.");

  // test
  for (int i = 1; i<a.length; i++){
    if (a[i-1] > a[i] ) {
      System.out.println("FEIL a["+(i-1)+"]:"
            +a[i-1]+"a["+i+"]:"+a[i]);
      return;
  } }
} //end utførOgTest
```

Program 19.5 *Koden for konstruktøren, som oppretter a[] og fyller den med tilfeldige tall < n; og utførOgTest(), som tar tiden med System.nanoTime() som gir tiden i nanosekunder (milliardtedels sekunder). Den kaller så sQuick(), skriver ut tiden i millisekunder og utfører en enkel test på om sorteringen gikk bra.*

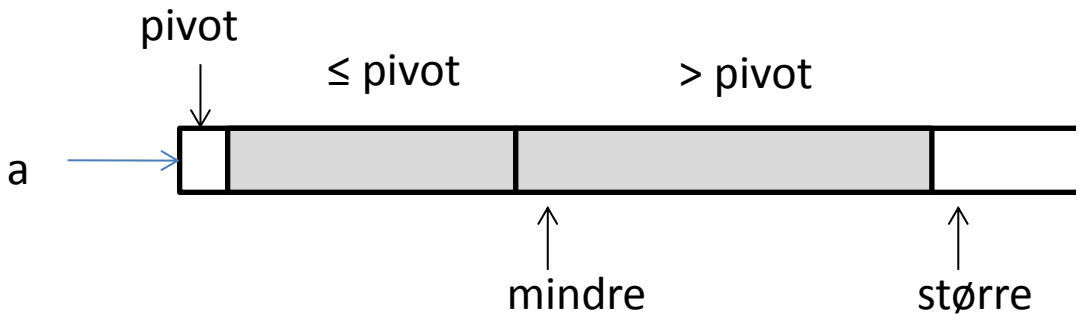
Koden er forklart under kodelistingen. Testen som utføres er strengt tatt ikke god nok. En enkel og komplett test ville være å sortere de samme tallene med Javas innebygde sorteringsalgoritme: java.util. Arrays.sort, og så sammenligne de to sorteringene element for element. Man kunne også ta tiden på Arrays.sort og sammenligne tidene (se oppgave 1).

```
 void bytt(int [] a, int i, int j) {
  int t = a[i];
  a[i] = a[j];
  a[j]=t;
 }// end bytt

 /** Rekursiv, sekvensiell QuickSort av a[lav..høy] */
 void sQuick (int [] a, int lav, int høy) {
  int ind =(lav+høy)/2,
```

```
   pivot = a[ind],
   større = lav+1, // hvor lagre neste '> piv'
   mindre =l av+1; // hvor lagre neste '<= piv'

  bytt (a,ind,lav);// flytt 'piv' til a[lav],

  while (større <= høy) {
   if (a[større] < pivot) {
    // a[større] er 'mindre' – bytt
    bytt(a,større,mindre);
    ++mindre;
   }
   ++større;
  }

  bytt(a,lav,mindre-1);// sett 'piv'mellom store og små

  if ( mindre-lav > 2) sQuick (a, lav, mindre-2); //<=piv
  if ( høy-mindre > 0) sQuick (a, mindre, høy); //> piv

 }// end sQuick
```

Program 19.6 *Koden for selve sorteringen: metodene sQuick og bytt.*

Bytt er rett fram kode. Koden til sQuick går i to faser. Først er det et valg av **`pivot`**-element og oppdeling av den delen av arrayen som pekes ut i kallet med lav og høy. Deretter kommer to rekursive kall – ett på dem som er ≤ **`pivot`** og ett på den delen hvor de som er > **`pivot`** ligger. Selve logikken til oppdelingen illustreres av figur 19.6. Vi har to pekere i en løkke: **`større`**, som peker på den plassen vi vil ha *neste* element som er > **`pivot`**; og mindre som peker på den plassen hvor vi vil ha *neste* element som er **< `pivot`.** Variabelen **`større`** økes alltid med 1 i hver løkkegjennomgang hvor vi ser på elementet **`a[større]`.** Er **`a[større]`**≤**`pivot`**, så bytter vi det med **`a[mindre]`** som jo peker på det elementet som er «lengst til venstre» av de som er > **`pivot`.** **S**å øker vi **`mindre`** med 1.



Figur 19.6 *Kvikksort deler opp a [lav ... høy] fra venstre mot høyre i to deler. Fra lav + 1 og til og med plassen: en til venstre for «større», er arrayen allerede riktig delt i to. Pivot ligger midlertidig i a[lav].*

Merk at vi plasserer **`pivot`** mellom de to delene når vi er ferdige. P**`ivot`** står da på sin endelige plass i sorteringen. Den skal aldri mer flyttes og er ikke med på den videre todelingen (som egentlig da er en tredeling) av hver del som vi skal dele videre opp. Behandlingen av **`pivot`** er litt spesiell – først tar vi og setter den helt til venstre i **`a[lav].`** Så deler vi resten av arrayen i to deler, og så bytter vi **`pivot`**, som står på i **`a[lav]`** med det elementet som står lengst til høyre av dem som er ≤ **`pivot`**: **`a[mindre-1].`** Grunnene til dette er to. Hvis **`pivot`** viste seg å være det største av alle elementene vi nå skal sortere, ville vi få en uendelig rekursjon fordi vi ikke fikk delt opp i to deler, men i en. Den andre grunnen er at vi da plasserer **`pivot`** på sin endelige plass, og at summen av de delene vi sorterer videre er ett element mindre. Dette gjør at programmet vil terminere uansett hvor uheldige vi er i valg av **`pivot`**.

Etter at vi har satt inn **`pivot`** mellom de to delene, gjør metoden kall på seg selv for de to delene til venstre og høyre for **`pivot`** hvis disse har større lengde enn ett element. Det er særlig denne kodedelen som blir annerledes i en parallell versjon av Kvikksort som vi skal se på i neste avsnitt.

Denne koden for Kvikksort er spesielt rask for alle små verdier av n, og like rask for større verdier av n som den innebygde sorteringsmetoden

`Arrays.sort(..)` i biblioteket **`java.util`**, som er en annen og mer komplisert koding av Kvikksort.

19.8 EN BLANDET PARALLELL OG REKURSIV KVIKKSORT

Grunnideen i en parallell versjon av Kvikksort er at vi bytter ut de rekursive kallene med at vi starter en ny tråd for hvert av de to kallene. Men som vi så av de to foregående eksemplene, tar det en viss tid å starte og stoppe tråder, og sortering går meget fort (tabell 19.2). Vi må derfor lage en blandet algoritme som benytter tråder når vi f.eks. deler opp en del av arrayen som er > 50 000 elementer, men som benytter rekursjon for kortere deler.

Skjelettkoden til programmet for parallell sortering en ganske likt det for sekvensiell kvikksortering:

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

class ParaQuick {
 int [] a;
 static int antTråder = 1;
 final static int PARA_LIMIT = 50000;

 synchronized void tellOppAntTråder () { antTråder ++;}

 void bytt(int [] a, int i, int j) {…}

 void pQuick(int [] a) { pQuick(null,a, 0,a.length-1); }

 void pQuick (CyclicBarrier b,int [] a, int lav, int høy) {…}

 ParaQuick(int n) { … } // konstruktor

 void utførOgTest() {… }

 public static void main (String [] args) {
  new ParaQuick(Integer.parseInt(args[0])).utførOgTest();
 }

 class Para implements Runnable{
  int [] a; int lav,høy;CyclicBarrier b;

  Para(CyclicBarrier b, int []a,int lav,int høy) {
   this.a=a;this.b=b;this.lav=lav;this.høy=høy;
   tellOppNumThr();
  }

  public void run() {
```

```
    pQuick(b,a,lav,høy);
  } // end run
 }
} //end ParaQuick
```

Program 19.7 *Skjelettkoden for parallell kvikksortering. Konstruktøren ParaQuick har samme kode som konstruktøren* SeqQuick *i program 19.6. Også metoden bytt er identisk med den sekvensielle bytt.*

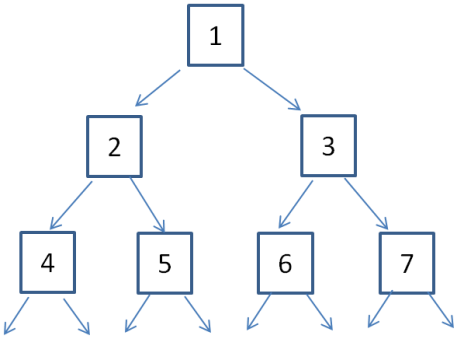
Vi ser at internt i klassen **`ParaQuick`** har vi i tillegg til arrayen **`a`**, en statisk variabel **`antTråder`** som bare er til statistikk, og som teller opp via metoden **`tellOppAntTråder()`** hvor mange tråder vi har i systemet vårt og skriver dette ut til sist. Den vesentligste endringen i forhold til den sekvensielle versjonen er at vi har innført en indre klasse **`Para`** som inneholder **`run()`** - metoden som er den koden vi skal utføre i denne tråden parallelt med de andre trådene. Den parallelle koden er et kall på **`pQuick`** for å få sortert den delen av **`a[]`** som denne tråden skal sortere.

```
void pQuick (CyclicBarrier b,int[]a,int lav,int høy) {
 int ind =(lav+høy)/2,
  piv = a[ind],
  større=lav+1, // hvor lagre neste 'større enn piv'
   mindre=lav+1; / hvor lagre neste 'mindre enn piv'

 bytt (a,ind,lav);    / flytt 'piv' til a[lav]
 while (større <= høy) {
  if (a[større] < piv) {
   bytt(a,større,mindre);
   ++mindre;
  }
  ++større;
 }

 bytt(a,lav,mindre-1);// sett 'piv' mellom store og små

 if ( høy-lav > PARA_LIMIT){
  CyclicBarrier b2 = new CyclicBarrier(3);
  new Thread(new Para(b2,a,lav,mindre-2)).start();//<=
  new Thread(new Para(b2,a,mindre,høy )).start(); //>
  try { // wait on own two calls to complete
   b2.await();
  } catch (Exception e) {return;}
 } else {
```

```
   // korte arraysegmenter raskere med rekursjon
   if (mindre-lav>2) pQuick (null,a, lav,mindre-2);//<=
   if (høy-mindre>0) pQuick (null,a, mindre, høy); //>
  }

  if (b!= null) {
   try { // signaliser til kallende at denne er ferdig
     b.await();
   } catch (Exception e) {return;}
  }
 }// end pQuick
```

Program 19.8 *Sorteringsalgoritmen pQuick. Delingen av a[] er akkurat den samme som ved sekvensiell sortering. Det interessante er hvordan vi parallelliserer de to delene etter oppsplittingen. Hvis hele den delen vi skal sortere er større enn PARA_LIMIT, oppretter vi en syklisk barriere som skal vente på tre tråder – de to nye trådene som startes og den tråden som skaper disse. Ventesetningen like etter de to trådene som skapes, er at den tråden som skapte disse to trådene, venter på at de begge skal bli ferdige. Helt på bunnen av metoden ser vi et nytt await()-kall. Det er signalet til den tråden som kalte denne metoden, om at denne tråden er ferdig – altså et signal oppover.*

Koden forklares i teksten under program 19.8. I oppgave 2 skal dere prøve ut og forklare tidsforbruket uten å kode dette med rekursiv løsning for kortere deler, bare med tråder.



Figur 19.7 *Kall-treet. Parallell Kvikksort bruker enten tråder eller rekursjon for å løse problemet. Dette kan illustreres som et tre (med roten i toppen) av de ulike instansene av metoden pQuick hvor ett høyere nivå gjør to kall på neste nivå. Merk forskjellen på rekursjon og tråder. Starter vi tråder, vil de (grovt sett) starte i den rekkefølgen som de her er nummerert: først 1 som starter 2 og 3, så vil 2 starte 4 og 5, mens 3 vil starte 6 og 7, osv. Dette kalles bredde-først. Hvis det imidlertid dreier seg om rekursive kall, da vil 1 starte 2 som vil starte 4, … og først etter at alle kall som genereres av 4 og dets «barn» har returnert til 2, vil 5 bli kalt, så 3, 6 og 7. Dette kalles dybde-først-traversering av treet.*

19.9 Effektiviteten på sekvensiell og parallell Kvikksort

Vi testet de to versjonene av Kvikksort på to ulike maskiner, én med 8 kjerner og én med 64, og resultatene er referert i tabell 19.2 og 19.3.

	10	100	1 000	10 000	100 000	1 mill.	10 mill.	100 mill.
Sekvensiell	0,01	0,05	0,70	13	18	89	963	11 196
Parallell	0,01	0,06	0,84	16	24	67	356	3579

Tabell 19.2 *Antall millisekunder det tar å sortere en array med tilfeldig innhold med den sekvensielle og parallelle versjonen av Kvikksort (parallelt for deler som er lengre enn 50 000). Kjørt på en Intel i7 870, 3 GHz klokke med 8 kjerner.*

	10	100	1 000	10 000	100 000	1 mill.	10 mill.	100 mill.
Sekvensiell	0,01	0,05	0,81	18	21	197	1413	16 497
Parallell	0,01	0,06	0,99	21	56	106	1332	5025

Tabell 19.3 *Antall millisekunder det tar å sortere en array med tilfeldig innhold med den sekvensielle og parallelle versjonen av Kvikksort (parallelt for deler som er lengre enn 50 000). Kjørt på en maskin med 4 Intel Xeon prosessor L7555, 1,87 GHz klokke, totalt med 64 kjerner.*

Kommentarer til tabellene, først tidene for sekvensiell utførelse. Vi ser at tidene er langt mer enn 20 ganger så lange når vi går fra å sortere 1000 til 10 000 tall (de burde bare vært litt mer enn 10 ganger så lange), og tilsvarende videre til 100 000. Det kan forklares med at mesteparten av arrayen vi sorterer først ikke får plass i nivå 1-cachen, men i nivå 2, og for 100 000 tall i nivå 3-cachen og hovedhukommelsen som jo er mye langsommere.

Det beste vi synes å oppnå, er at den parallelle versjonen går om lag tre ganger så raskt som det sekvensielle programmet, og ikke så mange ganger fortere som vi har kjerner. De parallelle resultatene kan forklares ved flere faktorer, men det interessante spørsmålet er: Hvis vi har k kjerner, hvorfor går det ikke k ganger fortere?

Det er i hovedsak to grunner. Først er ikke parallelliseringen optimal. Den første oppdelingen av `a[]` i to deler skjer sekvensielt og like langsomt som den sekvensielle algoritmen. Først da starter vi «bare» to tråder. Når hver av disse igjen deler opp hver sine deler i to, får vi fire tråder som er aktive (og ikke venter), osv. Vi får altså ikke brukt alle kjernene helt fra starten av programmet.

Den manglende parallelliseringen er imidlertid ikke hovedforklaringen. I figur 19.2 ser vi en strek som forbinder hver prosessor med hovedhukommelsen. Dette kalles en hukommelseskanal, og den kan ikke gå fortere enn hukommelsen klarer å operere. Vi husker også at hovedhukommelsen var mye langsommere enn hver kjerne. Det betyr kort sagt at det er en kø av kjerner på hukommelseskanalen for å lese og skrive i hovedhukommelsen, og det blir mye venting. Vi greier ikke å få trådene til å jobbe med full hastighet.

Lite kø på hukommelseskanalen vil vi oppleve ved problemer som involverer en liten mengde data, og som gjør mer arbeid med hvert dataelement enn de problemene vi har sett på her. Et slikt problem («Selgerens rundreise») illustreres i oppgave 3. Da vil data som det jobbes med i kjernen, i stor grad være i cache, og lesinger og skriveoperasjoner i hovedhukommelsen vil ikke skje så ofte at det skaper kø på hukommelseskanalen. Parallelliseringen av

kvikksorteringen kan imidlertid ikke ses på som mislykket – at store sorteringer går tre ganger så fort, er klart noe man vil ønske i praksis.

19.10 AMDAHLS LOV

Vi så at Kvikksort-eksemplet først hadde en sekvensiell del (dele arrayen i to), og så kunne parallelliseringen begynne. Det er generelt for mange algoritmer at de først har en sekvensiell del. Et eksempel kan være at den sekvensielle delen tar 10 % av tiden, og de delene som kan parallelliseres tar da 90 % av tiden når vi kjører alt sekvensielt. Vi ser at det beste vi da kan oppnå av en parallell algoritme, er at den maksimalt vil gå 10 ganger så fort som den sekvensielle. Uansett hvor mange hundre- eller tusenvis av kjerner vi bruker på den parallelle delen, kan vi ikke få den til å gå raskere enn 0,0 sekunder. Vi står da igjen med 10 % av beregningene i den sekvensielle delen – altså maksimalt 10 ganger raskere enn en sekvensiell beregning.

Generelt, anta at vi har en algoritme som må utføre p % av algoritmen sekvensielt, og at vi greier å få den delen av koden som kan parallelliseres til å gå k ganger raskere. Da er den maksimale hastighetsforbedringen S vi kan få:

$$S = 100/(p + (100-p)/k)$$

Setter vi inn at den sekvensielle delen p = 10 % og antall kjerner k = 1000, ser vi at S blir 9,9 – vi greier altså å få den til å gå 9,9 ganger fortere. Greier vi å få p ned i 1 %, blir S = 91 med de samme forutsetningene. Lærdommen fra Amdahls lov er at før eller siden er det den delen som ikke kan parallellisere som vil dominere kjøretiden. Det er nesten alltid sånn at noe må foregå sekvensielt – f.eks. skal data leses inn, svar skal skrives ut og selve programmet må leses inn i hukommelsen og settes i gang. Vi husker også at det kan ta noen få millisekunder å starte én tråd, så det tar alltid noe tid før vi kan få startet parallell kjøring. Selv om vi kanskje håpet at med 1000 kjerner ville det gå 1000 ganger så fort, så trenger vi ikke å fortvile. At beregninger går fra 9 til ca. 90 ganger fortere er klart et nyttig resultat.

19.11 OM PROSESSOREN, JAVA-KOMPILATOREN OG HUKOMMELSESMODELLEN

I begynnelsen av boka ga vi inntrykk av at instruksjonene blir utført i den rekkefølgen de er presentert i programteksten.

Det er ikke nødvendigvis riktig, av to grunner. Først vil selve prosessoren gjerne bytte om på instruksjoner den holder på å utføre hvis det kan gå fortere, og *dersom det ikke har noen innvirkning på sluttresultatet*. Det samme prøver også Java-kompilatoren. Det er mye tid å spare på å flytte rundt på instruksjoner når de utføres, f.eks. at flere andre operasjoner utføres samtidig som vi holder på med en flyttall (double)-operasjon, som tar lang tid. Slik ombytting kan altså bare foretas hvis *det ikke får innvirkning på sluttresultatet.* Prosessorkjernene har også nå fått egne instruksjoner som kan kjøre flere prosesser av typen flyttall i parallell, f.eks. multiplikasjon, dersom variablene til disse multiplikasjonene som skal utføres, ligger etter hverandre i lageret. Slik omorganisering av rekkefølgen på instruksjoner må imidlertid ikke gå ut over det vi hadde tenkt da vi laget programmet – programlogikken. Trenger vi resultatene av én beregning i høyresiden i en annen beregning, eller i for eksempel en utskrift til skjerm eller fil eller i en test, blir ikke slik ombytting eller parallellkjøring av instruksjoner foretatt.

Det Java og prosessoren garanterer deg, er at du får utført en prosedyre som gir eksakt samme resultat som om programmet ble utført instruksjon etter instruksjon, ovenfra og nedover, en etter en, og hver gang du bruker verdien på en variabel, er den der som om programmet ditt ble utført enkelt og greit ovenfra og nedover.

Siden en programmerer nesten aldri merker effekten av denne ombyttingen av instruksjoner mellom prosessoren og kompilatoren, behøver vi ikke dvele mer ved det, unntatt en feil man kan komme til å gjøre. Se på følgende to linjer i et program:

```
x = 12;
y = 19;
```

Program 19.10 *Tilordning av verdier til to variabler. Hvis vi seinere i programmet tester og finner at y er lik 19, kan vi* ***ikke*** *dermed slutte at x er lik 12 (selv om det ser ut som x = 12 ble utført «før» y = 19). Siden både prosessoren og Java-kompilatoren kan bytte om på tilordningen av verdier til de to variablene, og utsette «x = 12» til verdien av x enten brukes i en annen beregning, i en test eller en utskrift, kan det godt hende at y får sin verdi lenge før x får sin verdi.*

Vi kan konkludere med at den gamle, enkle modellen: Vi har en prosessor og en kjerne som leser og utfører instruksjonene en etter en, ovenfra og nedover, ikke er helt riktig, men at det ikke gjør noe i de aller fleste tilfeller i et program med bare én tråd. Dette er viktig for oss som programmerere. Uten denne enkle modellen vil det nesten ikke være mulig å skrive riktige programmer.

19.10 OPPSUMMERING – OM PARALLELLISERING AV ET PROBLEM

Det er mange måter å parallellisere et problem på i en multikjernemaskin, men de to som er forklart her, bruk av synkroniserte metoder og barrieresynkronisering, skulle være tilstrekkelig. Særlig barrieresynkronisering egner seg for større problemer.

1. For at det skal være noen vits i å parallellisere et problem, må det ha *mer* enn noen få operasjoner for hvert dataelement. Som en huskeregel kan vi si at hvis den største versjonen av problemet vi greier å kjøre sekvensielt ikke tar mer enn ett sekund, er det ingen vits i å parallellisere det.

2. Start med en godt testet sekvensiell løsning av problemet.

3. Se etter om man kan dele *data* opp i et antall like store deler, hvor helst hver del kan løses etter den sekvensielle metoden i hver sin tråd – altså i parallell.

4. Hvis vi under beregningene må ha felles data, må de alltid beskyttes. All skriving og lesing på disse felles dataene skjer med synkroniserte metoder. Hvis *alle* dataene alltid er felles, er det ingen vits i å parallellisere problemet.

5. Når hver tråd har løst sin del, og etter at alle har ventet på *en felles barriere* når de er ferdige, så kan alle trådene etterpå lese resultatene av hverandres beregninger.

6. Kanskje er vi nå enten ferdige, eller resultatene fra første beregninger deles igjen opp i flere tråder med en ny barriere, osv.

7. Tenk spesielt på hvordan main-tråden skal vente og slippe til når alle trådene er ferdige og har løst problemet. Det kan godt være en barriere som venter på antall tråder + 1, som er main-tråden, som main legger seg og venter på når alle trådene er startet.

19.11 OPPGAVER

Oppgave 1

I den sekvensielle versjonen av Kvikksort, modifiser utførOgTest() for sorteringseksemplet slik at du lager en array med samme innhold som har blitt sortert av sQuick, og sorter den med Arrays.sort(). Skriv ut tiden for denne og sammenlign med tiden for Kvikksort.

Oppgave 2

I den parallelle versjonen av Kvikksort, pQuick, fjern de rekursive kallene og løs problemet bare med tråder. Kommenter kjøretidene og antall tråder du får. Er dette lurt?

Oppgave 3

Skriv et program for «Selgerens rundreise»; først sekvensielt (rekursiv) og så en blanding av parallelt og rekursivt. Dette er en oppgave med svært mange

beregninger og svært lite data, og den egner seg svært godt for parallellisering. Problemet er slik: En selger skal reise og besøke n byer – hver by skal besøkes bare én gang. Hun kjenner avstandene fra hver by til alle de andre byene. Disse dataene har hun lagt inn i en todimensjonal array: **`avstand[][]`**, slik at avstand [i][j] er avstanden fra by i til by j. Om avstandene vet du også at **`avstand[i][j]=avstand[j][i]`**, at **`avstand[i][i]=0`**, og at det alltid er raskest å reise direkte til en by – det går aldri noen snarvei via en annen by.

Skriv ut den korteste reiseplan av alle mulige hvor selgeren besøker hver by bare én gang, og til slutt kommer tilbake til utgangsbyen.

```
class SeqRSelger {
 int [][] avstand ;
 int [] x,y;
 int bestHittil =Integer.MAX_VALUE;
 int [] besteRute, reiseRute;
 boolean [] besøkt;
 int n;

 void RSR (int nivå, int by, int lengde) {
  if( nivå == n) { // nå bare reisen tilbake til by:0
    if (lengde+avstand[by][0] < bestHittil){
     <noter ny beste reisevei og lengde>
    }
  } else {
   for (int by = 1; by < n ; by++) {
    // besøk alle ikke-besøkte byer unntatt 0
    if (! besøkt[by]) {
     besøkt[by] = true;
     reiseRute[nivå] = by;
     RSR (nivå+1,by,lengde +avstand[by][by]);
     besøkt[by] = false;
 }}}} // end PSR

 SeqRSelger(int n) {
    <opprett og initier arrayer med tilfeldige tall>
    <beregn avstandsmatrisen med Pytagoras>
 } // end konstruktor

 void utfør() {
  < ta starttidspunkt>
  RSR(1,0,0);
  <skriv ut data, tid brukt på PSR() og beste
```

```
reisevei>;
 }

 public static void main (String [] args) {
  new SeqRSelger(Integer.parseInt(args[0])).utfør();
 }
} //end SeqRSelger
```

Programskisse til oppgave 3 «Selgerens rundreise». *Dette er den sekvensielle varianten av problemet. Få denne til å virke og lag så en parallell versjon. Dette er på ingen måte den beste algoritmen som løser dette problemet, men den korteste. Merk hvor raskt den løser et 10-bys problem, men at det tar altfor lang tid å løse et 15-bys problem. Kjøretiden går som n! (=1*2*..*(n-1)*n). Vi ser at kjøretiden da øker meget raskt med n.*

Oppgave 4

Lag en versjon av parallell Kvikksort hvor du bare bruker én syklisk barriere. Hint: Av kall-treet i figur 19.7 ser du at det er ett kall på **`pQuick`** på nivå 1 (toppen) i treet, samlet 3 kall på nivå 2 og nivå 1 i treet, samlet 7 kall til og med nivå 3 osv. (formelen for antall kall med k nivåer er: 2^k-1). Du må da, før du starter kallene, regne ut hvor mange nivåer du vil ha i kall-treet for å starte nye tråder, og sette opp den sykliske barrieren til å vente på så mange. Resten av problemet løser du som før rekursivt. Er denne versjonen raskere enn den i avsnitt 19.8?