# Effektiv Parallellprogrammering, kompendium i IN3030/4330

av

ARNE MAUS

PT (Programmeringsteknologi),
Institutt for informatikk,
Universitetet i Oslo

7. mai. 2023,
ver. 4

(Ris og ros, feil, forslag til forbedringer/utvidelser/strykninger og andre kommentarer sendes:
arnem@ifi.uio.no)

## Innledning

Dette er en første oppdatering av et kompendium i parallellprogrammering i Java på en flerkjerne maskin med felles lager. Et vesentlig tillegg er et nytt kap. 7 om lambda-uttrykk og strømmer og særlig da parallelle strømmer.

Dette stoffet foreleses på Institutt for informatikk i kursene IN3030/4030 på bachelor- og master-nivå. Det poengteres at dette kompendiet er tillegg til kurset som kan hjelpe til å forstå hvorfor dagens datamaskiner har stadig flere kjerner, hvilke mekanismer disse inneholder som gjør at programmer kan gå fortere alt etter hvordan vi skriver våre parallelliserte algoritmer.

Studentene har som forutsetning hatt det første kurset i Algoritmer og Datastrukturer og har hatt to introduksjonskurs i OO (objektorientert) programmering hvor de også har sett noen få eksempler på parallelle programmer i Java men ingen inngående opplæring i parallellisering. Kurset IN3030 starter derfor på bunnen med hva tråder er, men kommer ganske fort opp på et rimelig avansert nivå i parallellisering av algoritmer. Dette kompendiet er tenkt som støttenotat og utdyping av forelesningene og forelesningsfoilene i kurset.

Vi programmerer i dette kurset typisk for en vanlig PC /mobiltelefon med 2-16 kjerner, men programmene i denne boka vil også virke effektivt på mer uvanlige prosessorer som Intel Xeon Phi med 62 kjerner eller store server-prosessorer med 32-64 kjerner med felles hukommelse (som Graviton3 brikken med 64 kjerner laget med 55 milliarder transistorer på én brikke) . Det som her foreleses her er ideen om en datamaskin med felles hukommelse for flere prosessorkjerner, og at det er flere nivåer, typisk 3, med cache-hukommelse på hver kjerne (delt eller ikke delt cache er ikke viktig). Cache hukommelse forklares senere. Valg av programmeringsspråk, Java, er nok viktig for eksemplene i dette kompendiet, men det er få kjente grunner til at tilsvarende og nær identiske programmer ikke skulle være like effektive i Scala, C++, C#, Object C eller andre objektorienterte språk med tråder og med et felles adresserom (i hovedhukommelsen og cachene).

Det er to krav vi stiller til de parallelle programmene vi skal lage:

- De skal være **riktige** – produsere samme resultater som en riktig og rask sekvensiell løsning
- De skal være **mer effektive, klart raskere** enn en sekvensiell løsning av samme problemet

Viktig i dette kurset er hvordan man parallelliserer et riktig, sekvensielt program + og lage egne parallelle algoritmer. I dette kurset begrenser *vi oss til at **våre parallelle programmer skal gå på én PC med ett felles lager**.* Vi vil altså ikke ta opp bruk to andre aktuelle typer av maskiner, som også nyttes til å løse parallelle problemer som bruk av grafikk-kort og klynger av PC-er i nett.

Grunnen til at vi ikke omhandler bruk av grafikk-kort med mange tusen enkle prosessorer er at programmering av disse krever en helt spesiell kode og at de løser bare visse typer av problemer svært effektivt (mye data, med samme instruksjon utført på alle disse data i parallell)

Dagens virkelig store dataanlegg består klynger av flere millioner PCer koblet sammen i et raskt nett. Alle de metodene som læres i IN3030/4030 er aktuelle der, er det her mange ekstra problemer man får med en slik komplisert maskin – f.eks. strømforbruket (verdens største slik klynge i 2022 bruker ca. 30 000 KW), med tilsvarende store luft-kjølingssystemer. For slike klynger trenger vi også teknikker og programvare til det å spre ut en beregning ut på så mange enkelt-maskiner over et nettverk med den relativt store forsinkelsen i datanettet mellom maskinene (ca. 1000 ganger så stor forsinkelse sammenlignet med tidene til lesing og skriving i én PCs sin hovedhukommelse), og særlig det til sist å få samlet så mange delberegninger til ett, felles svar krever egne programmerings systemer og filsystemer.

Dere skal altså lære de mange problemene vi støter på og hvordan disse kan relativt enkelt kan løses med parallellisering av et sekvensielt program på én multikjerne PC. Kurset er *empirisk* (med tidsmålinger), og ikke basert på en teoretisk modell av parallelle beregninger. Vi oppfatter programmet som en god nok modell av det problemet vi skal løse. Vi trenger ingen modell av modellen. Siden slike teoretiske modeller, særlig ulike varianter av PRAM-modellen (Parallel Random Access Machine) brukes i mange lærebøker i parallelprogrammering, vil vi i dette kurset og i dette kompendiet kommentere hvor feilaktige anslag disse teoretiske modellene er, og gjør at både hvordan vi bør parallellisere våre algoritmer og hvilke kjøretider de faktisk da kan bli svært misvisende.

# 1. OM VESENTLIGE FORHOLD SOM PÅVIRKER EKSEKVERINGSTIDENE

Når vi kjører våre programmer vet vi at vi skriver det i et programmeringsspråk, i vårt tilfelle Java, som så kjører på selve maskinen med sine maskininstruksjoner og oppbygging med ulike typer elektronikk.

I det følgende vil vi gå gjennom de (forbausende mange) mekanismer, både de som har lager elektronikken og Java har lagt inn som vil redusere eksekveringstidene i stor grad, og som vi kan benytte oss av når vi skriver programmer. Jeg presiserer at det er ikke hele datamaskinen eller hele Java som beskrives – bare de mekanismene som vi som programmerere kan bruke til å skrive raskere programmer - både sekvensielle og særlig for oss – parallelle programmer.

Vi begynner først med datamaskinen før vi tar for oss Java, som nå er under stor utvikling med stadig nye begreper inkludert. I den grad det vil endre særlig hastigheten av våre programmer, vil det bli beskrevet. .

## 1.1 DATAMASKINENE

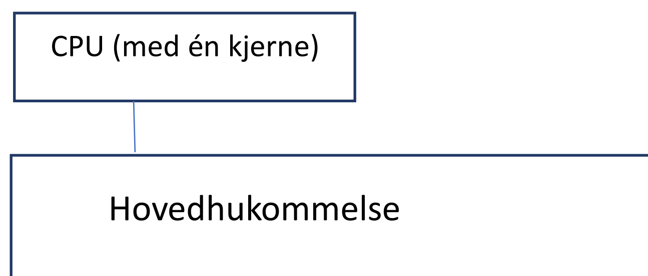
**Hukommelses-systemet**

a) **Lageret er byte-adressert**
   Lageret i alle Intel, AMD og Apple/Arm -maskiner er byte-adressert – med 8 bit i en slik byte. Det betyr at den minste data-enheten vi kan lese og skrive fra CPUen til hukommelsen er én byte lang– f.eks er en **`byte`** variabel i Java er 8 bit lang (eks: **`byte a;`**`),` mens en vanlig **`int`** er 4 byter. Dette er kanskje ikke så viktig å vite i sekvensielle programmer, mens i parallelle programmer er det avgjørende at to parallelle tråder ikke samtidig prøver å skrive på samme byte i lageret, Da er det viktig å vite at hvis to tråder samtidig ønsker å skrive eller endre på ett av bit-ene i en slik byte samtidig, går det galt selv om det er ulike bit vi ønsker å endre på i denne byten.

b) **Cache-systemet.**
   Når vi regner med data i programmet vårt, f.eks. skal utføre setningen: **`a = b + c`**, tenker vi at alle data er i en stor hukommelse og at variablene a, b og c der har hver sin plass i den. Vi leser/kopierer først verdiene av b og c fra hukommelsen, legger disse sammen og skriver resultatet ned i a-plassen . Det som skjer, er imidlertid langt mer komplisert.

## 1.2 LITT OM MASKINENS KONSTRUKSJON, CACHE OG MULTIKJERNE

Før 1980 var datamaskiner relativt enkle slik det framstilles i fig 1.1. Man hadde en beregningsenhet kalt CPU (Central Prosessing Unit). Den utførte én instruksjon av gangen i den rekkefølge de var spesifisert i programmet og leste, og skrev sine data direkte i hovedhukommelsen.




***Figur 1.1*** *Skisse av en datamaskin i ca. 1980 hvor det bare var én beregningsenhet, en CPU, som leste sine instruksjoner og både skrev og leste data (variable) direkte i hovedhukommelsen.*

Fra 1980-tallet begynte imidlertid CPUene å gå mye raskere enn hovedhukommelsen. Ordbruken skiftet også, slik at nå snakker vi om en prosessor istedenfor CPU. Dagens avanserte prosessorer bruker om lag 150-200 ganger så lang tid å skrive til, eller lese fra, hovedhukommelsen som den tid det tar å utføre en enkel instruksjon (som å legge sammen to heltall). Her ville det ha blitt mye dødtid.

Svaret var å legge først én, senere flere mellomhukommelser (cache-hukommelser) mellom prosessoren og hovedlageret Nå er det vanlig med tre lag av cache-hukommelser som er hurtigere, men dyrere enn hovedhukommelsen. Når prosessoren 'tror' at den lagrer verdien av en variabel i hovedhukommelsen, lagres det først bare i nivå1-cachen (kalt L1, se fig 1.2), og prosessoren kan fortsette. Så snart som mulig lagres så disse data deretter i nivå2-cachen (L2), så i nivå3-cachen(L3) – og til sist i hovedhukommelsen omlag 200 klokkeenheter senere (1 klokkeenhet= ca. 1/3 milliardedels (1/3 nano-sekund) . Tilsvarende gjelder for lesning. Hvis prosessoren ikke finner de opplysningene den vil ha i noen av cachene, må det leses fra hovedhukommelsen og inn i alle cachene før prosessoren får adgang til data.

**Flere kjerner.** En annen viktig utvikling av prosessorene var at man etter ca. år 2005 ikke greier å få én prosessor til å gå fortere. Prøver man med en raskere klokke, vil prosessoren rett og slett først feile og så evt. smelte. Imidlertid greier man stadig å lage hver prosessor mindre og mindre ved at de transistorene den består av blir laget mindre. Hva skulle så databrikke-designere som Arm, Intel og AMD gjøre? De la flere prosessorer, heretter kalt prosessorkjerner eller bare kjerner på hver brikke. Vi fikk da maskiner med to prosessor-kjerner (dual-core), så med fire kjerner, osv. Det er uklart hvor dette ender, men det er i alle fall laget forsøksproduksjon av brikker med ca. 100 slike prosessorkjerner, og det er helt sikkert at utviklingen stopper ikke med det. I tillegg finnes maskiner hvor man har satt 2 eller 4 slike multi-kjerne prosessorer 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 kapittelet er eksemplene fra 2017 testet på to slike maskiner, en med 8 tråder (=1 prosessor med 4 kjerner som hver har hyperthreading) og en med 64 kjerner (=4 prosessorer med 8 kjerner som hver har hyperthreading), mens 2022 og 2023 resultatene er testet på en nyere maskin med 8-tråder (4 kjerner). .
Det er også vanlig slik at hver kjerne har sin egen L1 og L2 cache, men deler som oftest L3 cachen med alle kjernene på samme brikke, men ikke med de andre prosessorene som evt. er i maskinen. Alle trådene i vårt program deler samme område i hovedhukommelsen. En viktig konklusjon på dette er at selv om en tråd som går på en av kjernene og 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 fordi den f.eks. holder på å bli skrevet ned via alle cachene og det kan ta flere hundre klokkesykler før den oppdaterte verdien er i hovedlageret. Det er også ca. 10 til 30 registre per kjerne. Disse kan brukes til å holde de mest nyttede variablene i en beregning, slik som indeksen 'i' i en forløkke eller de mest sentrale variablene i en metode, som **`s`** i metoden **`sum`**

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

Prog 1.1 *Programeksempel, summering av verdiene i en array.*

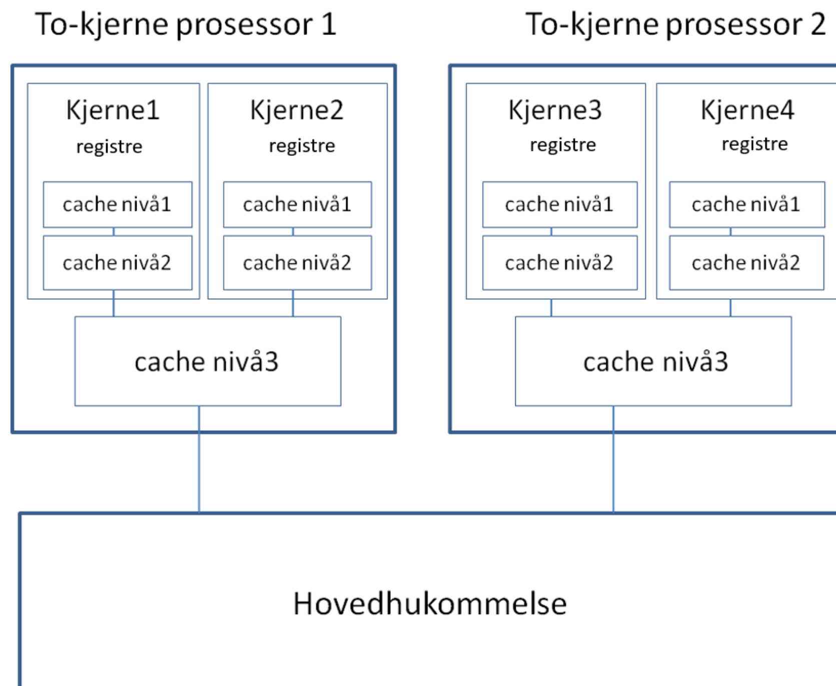
Det er Java-kompilatoren **`javac`** og senere kjøresystemet **`java`** som bestemmer hvor de ulike variablene i et program plasseres. Det er slik at alle variablene har sin plass i utgangspunktet i hovedhukommelsen. Men i koden fig. 1.1 ser vi at metoden sum inneholder to variable, **`s`** og **`i`**, som ikke er synlige utenfor metoden og som ikke kan leses av resten av programmet etter at metoden er utført. Disse to variablene vil derfor i en optimalisering av koden bli lagt i hvert sitt register.

**Parallelle instruksjoner** I de siste årene har det kommet en rekke maskin-instruksjoner, som sammen med en rekke nye registre i kjernen, kan de f.eks både med en og samme instruksjon multiplisere sammen to og to av disse registrene og summere resultatene fra disse 10-20 sett av registre. Dette er også en optimalisering av Java-koden kan nytte seg av.

```
void add1 (int tall, int ant) {
     for (int i = 0; i < ant; i++) {
           tall++;
      }
  }// end add1
```

**Prog 1.2** *Metode som øker variablen **tall** ant ganger.*

I det programeksempelet vi har i 1.2, 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 gamle og nye verdien går via L1 cachen og veien ned til hovedhukommelsen er lang. Vitsen med disse cachene er at selve beregningsenheten bare forholder seg til sin L1 cache, leser og skriver i den, mens resten av systemet stadig forsøker å holde de andre cachene og hovedhukommelsen oppdatert. Beregningsenheten greier altså å gå så fort som L1 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 1.2.*** *To dobbeltkjerne prosessorer i en maskin med ulike hukommelser. Når det går parallelle tråder på hver av disse kjernene, ser vi at de som oftest ser 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 cashene 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 rask L1 cache, hvorfor kunne ikke all hukommelse vært slik? Poenget er at slike cacher er laget på en mye dyrere måte enn hovedhukommelsen, hver byte i cachene tar større plass og krever mer strøm. Det er også slik at måten disse cachene adresseres på en måte som forutsetter at de er relativt 'små'. Det ville kort sagt ikke lønne seg eller mulig å lage all hukommelse slik.

Typiske størrelser og forsinkelser måles i hvor mange tikk (cycle) som klokka i prosessoren må gjøre før en operasjon er ferdig. En prosessor som er på 2GHz, har da en klokke som gjør 2 milliarder slike tikk per sekund. Forsinkelsene i de ulike delene av hukommelses-systemet kan da være:

Registre i kjernen størrelse = 0,1-1 Kb, tidsforsinkelse = 1 cycle

Cache nivå 1 størrelse = 32Kb tidsforsinkelse = 2 cycle

Cache nivå 2 størrelse = 512Kb tidsforsinkelse = 7 cycle

Cache nivå 3 størrelse = 4096Kb tidsforsinkelse = 16 cycle

Hovedhukommelse størrelse = 8 – 16 GB tidsforsinkelse = 190 cycle

**Ekstra L3 cache.** En nyhet i 2021/2022 er at en produsent (AMD) ha laget sin siste prosessor for spill og tekniske beregninger så tynn at det er mulig å legge en ekstra L3 hukommelse rett oppå brikken slik at den faktisk da får ca. 90 Mbyte totalt med L3 cache. Vitsen med å gjøre den tynn, er at det fortsatt er mulig å kjøle brikken selv med den ekstra L3 hukommelsen lagt oppå den.

**Cache-linje størrelsen** Når cache-systemet leser data fra hovedhukommelsen opp til nivå 3, så nivå 2 og til sist nivå1 cache, er det ikke én byte ad gangen, men 64 byte hver gang (fordi det tar så lang tid, er det fordelaktig å laste opp mer enn det det er bedt om). Hvis programmet trenger før eller siden disse bytene, ser vi at programmet vil gå mye raskere enn om man skulle gå helt ned i hovedlageret hver gang programmet trengte neste heltall, flyttall eller neste byte.

**Prefetch mekanismen** Hvis programmet leser eller skriver tilfeldige adresser i en stor array med én indeks (større enn L3 cachen): a[i] , a[k] a[j], a[j],. .. så vil dette gå vesentlig langsommere enn sekvensiell aksess : a[i], a[i+1]. Spesielt oppstår denne effekten når vi bortover ***radene*** i en særlig i en litt større to-dimensjonal array (så stor at det ikke er plass til hele arrayen i L2 eller L3 cachen), og programmet leser f.eks. a[i,j], a[i,j+1], a[i,j+2] vil en mekanisme i prosessorkjernen starte å lese neste cache-linje på 64 byte ***før*** programmet har bedt om det selv. Dette gjør at programmet går vesentlig raskere – mindre venting på data fra hovedhukommelsen. Denne mekanismen virker like bra hvis vi leser radene i en array baklengs fra slutten mot begynnelsen.
Merk at dette gjelder ***radvis*** lesing av en array. Leses derimot arrayen **kolonne**-vis (a{i,j] a[i+1,j] , a[i+2,j], .. vil elektronikken måtte lese en ny cache-linje for hver aksess, og vi vil få ingen hjelp av at vi allerede har lest 64 byter i cache-linjen som inneholder a[i,j] og prefetch-mekanismen gjenkjenner ikke dette som sekvensiell lesing. Lesing av neste kolonne-element vil derfor medføre at hver gang foretas en ny lesing 'helt' fra hovedlageret med en forsinkelse på ca. 200 nano-sekunder mot lesning av L1 cachen på 2 nanosekunder for radvis lesing av hvert element. En slik 'bom' på hva vi skal lese neste gang kalles en cache-miss.

**Pipeline:** Ikke alle maskininstruksjoner tar like lang tid, og de er da delt opp i flere mikroinstruksjoner som samlet sett løser oppgaven som den større instruksjonen skal løse – f.eks divisjon. En kjerne vil da ved neste klokke-tikk prøve å starte første mikroinstruksjon i neste instruksjon, så igjen ved neste tikk neste instruksjon, ...osv. Kjernen kan holde da på med å beregne typisk inntil 5 maskin-

instruksjoner samtidig på forskjellige grader av fullføring. Dette kalles en 'pipeline. Vi vil senere se at f.eks. multiplikasjon tar mye lenger tid (2-7 ganger så lang tid) som addisjon.

**Trådbytte:** er når en tråd som utføres på en av kjernene, avbrytes og en annen tråd overtar kjernen og kjører sitt program på denne kjernen. Den tråden som holdt på med sine beregninger må stoppes og innehold av de registrene som denne avbrutte tråden hadde til sine beregninger må lagres i hukommelsen. Så kan register-innholdet til den nye tråden lastes opp fra der den andre tråden hadde lagt sitt registerinnhold. Først når registerinnholdet fra den nye tråden er i registrene, kan denne startes. Når vi har flere tråder enn vi har kjerner i en prosessor, og det har vi alltid når vi teller med de over 1000 tråder vi har i operativsystemet, så må de ulike trådene bytte om på å kjøre på kjernene. Selv om operativsystemets tråder i sum ikke bruker mer enn ca- 10% av prosessoren klokkesykler fordi de i all hovedsak venter på I/O eller andre begivenheter, så tar de noe tid. Også, hvis ingen tråd ønsker å kjøre på en prosessor, er det et enkelt lite program som kjøres: *idle-loopen*- det enkleste program vi kan tenke oss som bare går rundt i en evig løkke, og som med en gang vil foreta trådbytte med en tråd som har noe å gjøre. Når idle-løkken kjøres vil også prosessoren søke å redusere klokkehastigheten slik at den sparer strøm. Den PC-en som noen av eksemplene i dette kompendiet er kjørt på kan variere klokkehastigheten mellom 1.2 GHz og opp til 3.4GHz. Kort sgt, prosessoren med sine kjerner stopper aldri, og kjernene bytter ofte om på å kjøre de ulike trådene.

**Spekulative beregninger** Når koden som utføres inneholder en test (i f.eks. if- while eller for-løkke) inneholder kjernen ekstra registre og elektronikk slik at den starter med å regne på begge grenene av utkomst av testen – både om testen gir sann eller falsk. Når testen er ferdig beregnet vil kjernen beholde beregningene som fulgte etter med det riktige sanne svaret på testen og fortsette der. Beregningene som fulgte etter at testen feilet vil bli strøket.
En annen enklere løsning på dette er at når en løkke er ferdig, så stoppes alle andre instruksjoner i pipelinen . De andre instruksjonene må da starte om igjen, mens koden for den andre utfallet (at f.eks. løkken var ferdig) kan begynne å eksekvere.

**IKKE så viktig her: Program-cache og cache for virtuell-tabellene mm.** I tillegg til cache-system for data slikt det er beskrevet ovenfor er det også en tilsvarende rekke av cacher for den optimaliserte koden. Også den virtuelle adresseringsmekanismen har cache-områder for sine tabeller, dvs at alle programmer får byttet ut de øvre delen av adressefeltet i instruksjonene med den «virkelige» adressen til hvor data ligger i hovedhukommelsen. Dette er et system som gjør det lettere å skrive programmer ved at *de alle kan skrives som om at data ligger sammenhengende* i *hovedhukommelsen*. Tidligere var det slik at når lageret ble fullt ble større deler av data midlertidig ble skrevet til disk (nå til SSA- 'disken'). I våre dager har man funnet ut at visse dataområder helles kan komprimeres med ZIP-algoritmen i selve hurtighukommelsen, slik at plass blir gitt programmer med høyere prioritet (disse ZIP-komprimerte områdene kan senere dekomprimeres når det blir plass til dem og program etterspør disse dataene).

Alle mekanismer i dette avsnittet bør man vite om, men de er klart mest viktige for de som skal skrive operativsystemer eller kompilatorer o.l., Det er mindre viktige hvis man som programmerer av brukerprogrammer for å påvirke kjøre-hastigheten. Hyggelig er det jo at noe av kompleksiteten i elektronikken er allerede tatt hånd av andre når vi skal skrive våre algoritmer og lage større datasystemer.

**Hvorfor er hukommelsen så 'langsom'**. Når man kjøper en datamaskin, kan man se som om hovedhukommelsen har om lag samme klokkehastighet, f.eks, 2600 MHz som CPU-en, men hvorfor tar det da så lang tid å lese og skrive i den. En lesning av en cache-linje på 64 byte sendes over data-kanalen som en ordre med startadressen til hukommelsen over en linje som enten kan sende 64 bit i parallell, eller mye raskere serielt 1 bit av gangen. Deretter skal data leses i hukommelsen, Den er meget billig,

men også meget kompakt laget. Å lese data der tar ca. 15-17 klokkesykler før de er klare for å sendes ut på datakanalen til CPU-en. Siden vi ber om 8 slike forsendelser, skulle det ta ca. 8*16 cycler – dvs ca 125 cycler bare for lesingen i tillegg til overføringen. Det korte svaret er da at hovedhukommelsen er meget rimelig og kompakt organisert, men at det går ut over hastigheten.

# 2. JAVA .

Java er et språk som er under sterk utvikling. Oracle, som nå eier Java, lager hvert år flere nye versjoner. Det som i hovedsak kommer til er nye begreper og konstruksjoner som records (som er objekter bare med data og ikke metoder). Prinsippet er at hver tredje versjon som lages er stabil (Java 5, Java 8, Java 11, Java 14 og Java 17,…) og strømmer beskrevet i kap. 7. Disse versjonen vil bli vedlikeholdt (feilrettet og tilpasset stadig nye versjoner av operativsystemer) i mange år fremover mens de to versjonene mellom disse er 'forsøksversjoner' hvor nye konstruksjoner kan komme og gå eller bli endret. Powerpointfoilene er i kurset er hovedsak laget med Java 5 og Java 8, mens noen av hastighetsbetraktningene i dette kompendiet er laget med Java14 (komplett liste over java -versjoner på:https://www.java.com/releases/, som sier f.eks at neste stabile versjon 20 vil komme Juli 2023, med en tidlig.versjon i mars 2023). For å få bedre forklaringer på det nye som kommer i Java anbefales å abonnere på 'Java-magazine' (gratis) : [https://blogs.oracle.com/javamagazine](https://blogs.oracle.com/javamagazine) .

## 2.1 JAVA OPTIMALISERING, DEL 1

I det overstående er viktige mekanismer i elektronikken som kan øke hastigheten på programmet vårt, men dette er mekanismer som i hovedsak operer på maskinkodenivå, og vi skriver jo programmene våre i Java. Det er imidlertid mye vi kan oppnå med hvordan vi skriver Java-koden:

1. Data som vi opererer på bør være minst mulig slik at de data programmet beregner på til enhver tid om mulig passer inn i L1 eller L2 cachene.
2. Uansett, prøv å lese og skriv data mest mulig sekvensielt for å utnytte prefetch mekanismen, og spesielt må vi i to-dimensjonale matriser lese/skrive data langs med radene, aldri langs kolonnene (i Java og nesten alle andre programmeringsspråk). Fortran er derimot annerledes, og lagrer data i to-dimensjonale data kolonnevis, og da må vi i Fortran lese/skrive disse matrisene kolonnevis).
3. Lag gjerne mange små metoder som hver løser ett enkelt problem (som f.eks. summen av elementene i en array, eller som finner neste primtall i en primtalls-array). Den mekanismen vi beskriver nedenfor som optimaliserer koden kan ikke like lett optimalisere lange metoder med mange løkker som mange små metoder som kaller hverandre.

Vi har i tidligere kurs lært at java-kompilatoren **`javac`** oversetter vårt Java program til en enkel og kompakt kode, bytekode, som kan sees på som instruksjonene til en byte-maskin som utfører disse byte-instruksjonene; noe tilsvarende som Python utføres idag. I den første varianten av Java, Java1 var det som skjedde – man ga bytekoden til **`java`** som så leste de ulike bytekodene og så utførte dem en-etter-en.
Det som nå fra og med Java 6 skjer er følgende:

1. Første gang et objekt fra en klasse genereres eller en metode kalles i koden, blir den først oversatt til maskinkode på den maskinen man kjører på (just-in time kompilering). Og det er denne maskinkoden som utfører programmet vårt. Det er klart raskere enn å tolke byte-koden som instruksjoner til en byte-maskin. Merk at det bare er de delene som utføres som blir oversatt til maskinkode – resten forblir i byte-kode.
2. Utføres en metode flere ganger (si 10-100 ganger), så optimaliseres den oversatte maskinkoden, Instruksjoner kan bli byttet om og forenklet, men den 'optimaliserte' koden gir samme svar som en ikke-optimalisert kode. Denne koden er nå mye raskere enn den ikke-optimaliserte maskinkoden

3. Utføres en slik optimalisert metode mange ganger – si mer enn 20 000 ganger som blir koden ytterligere optimalisert, og går da enda mye raskere
4. Kalles en metode enda flere ganger. f.eks. over 100 000 ganger, blir koden for denne metoden ytterligere optimalisert (nå for 3dje gang) og kan igjen gå enda raskere.

Denne optimaliseringene som gjør at noen operasjoner i Java kan gå fra 4 - 100 000 ganger fortere, er beskrevet i tabellen nedenfor hvor ulike java-elementer og to sorteringsalgoritmer testes.

| Tider i usek per iterasjon som funksjon av n, antall iterasjoner | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| n | 1 | 2 | 3 | 10 | 100 | 1000 | 10000 | 100000 | 1000000 | **X bedre (from n=1)** | **X bedre (from n=2)** |
| | | | | | | | | | | | |
| for-loop, len = 100 | 0,4 | 0,2 | 0,1 | 0,022 | 0,020 | 0,015 | 0,002 | 0,002 | 0,00000 | **181** | **90** |
| metodekall | 4,5 | 0,9 | 0,3 | 0,098 | 0,087 | 0,063 | 0,005 | 0,005 | 0,00043 | **10465** | **180** |
| int[] new , , len = 100 | 1,1 | 0,3 | 0,2 | 0,117 | 0,108 | 0,275 | 0,160 | 0,160 | 0,09815 | **11** | **2** |
| array kopi for-løkke , len = 100 | 2,4 | 1,7 | 3,2 | 1,980 | 1,880 | 0,889 | 0,018 | 0,237 | 0,01217 | **197** | **7** |
| System.arraycopy , len = 100 | 4,4 | 0,6 | 0,3 | 0,124 | 0,120 | 0,043 | 0,025 | 0,022 | 0,02008 | **219** | **27** |
| new Thread,start&join | 1164 | 362,9 | 224,0 | 197,424 | 174,278 | 172,948 | 175,820 | | | **7** | **2** |
| new Class C m.metodekall | 791,4 | 15,3 | 1,8 | 0,268 | 0,220 | 0,045 | 0,002 | 0,004 | 0,00274 | **288832** | **3438** |
| int [] a skriv , len = 100 | 0,8 | 0,7 | 0,1 | 0,036 | 0,035 | 0,027 | 0,023 | 0,008 | 0,00659 | **121** | **93** |
| int [] les ,, len = 100 | 0,3 | 0,3 | 0,0 | 0,028 | 0,026 | 0,022 | 0,015 | 0,006 | 0,00638 | **47** | **54** |
| double to long | 3,7 | 1,2 | 0,3 | 0,218 | 0,164 | 0,066 | 0,042 | 0,000 | 0,00004 | **92500** | **60000** |
| insertSort double[] , len = 100 | 93,1 | 164,0 | 150,1 | 55,343 | 7,266 | 1,754 | 1,634 | 1,618 | 1,61902 | **58** | **101** |
| Arrays.sort double[] , len = 100 | 404,4 | 73,1 | 60,9 | 56,506 | 6,028 | 1,357 | 1,165 | 1,116 | 1,11126 | **364** | **65** |

***Figur 2.1** Kjøretider per kjøring i **2022** som funksjon av antall ganger eksekvert i µsekunder (million dels) for ulike Java-elementer og for to sorteringsprogrammer: Ett brukerskrevet med kode i testprogrammet (insertsortering av flyt-tall) og et fra Java-biblioteket (Quick-sortering av flyt-tall). De to siste kolonnene viser speedup, hvor mange ganger fortere en eksekvering er etter 1 million kjøringer, regnet ut fra tidene for første kjøring (n=1) eller andre kjøring (n=2). Testene er kjørt på en AMD Ryzen 5 3500U PC med Java versjon 14.01 i Windows 10.*

Kommentarer til tabellen:

1. Vi ser at optimaliseringen er meget sterk for nesten alle konstruksjoner, men særlig metode-kall og det å lage et objekt av en klasse (samt konvertering av flyt-tall (double) til heltall ) effektiviseres vesentlig. Dette er viktig for vår programmering – å dele opp vårt program i klasser med mange mindre metoder, koster lite eller ingen tid når programmet har kjørt 3-10 ganger eller mer.
2. Vi ser at de to kolonnene til høyre regner ut Speedup (SU) henholdsvis fra tiden det tar å kjøre første gang og andre gang. Grunnen til også å regne ut SU fra andre kjøring er at da er all tisdsbruk som kompilering til maskinkode og henting av klassen evt. fra Java-biblioteket unnagjort. Vi ser dette spesielt i eksempelet Arrays.sort som første gamg tar 404 µs mot innstikksort med 93 µs som har koden liggende i testprogrammet. Derimot er optimelisering av maskinkoden der den senere foretas, inkludert i kjøretidene (som f.eks. Inntikksort andre og tredje gang).
3. Det er meget tilfredsstillende at egen skrevet brukerkode som innstikkSort kan optimaliseres opp mot en faktor 60-100.
4. Av det overstående kan det se ut som f.eks. Innstiksort blir effektivisert for all mulig bruk i programmet. Det er galt. Optimalisering av metodene består bl.a. i at hele koden for Innstikksort (og alle andre metoder) blir stappet inn koden der den *kalles fra* - erstatter kallet med selve koden. Det betyr at hvis vi bruker og kaller Innstikksortering fra et annet sted i programkoden må den gå igjennom samme optimalisering. Dette gjelder også optimalisering av å lage et objekt av en klasse (**new**)

## 2.2 PROSESSER OG TRÅDER

Et program (en prosess) i Java består nå av en eller flere tråder. Hver av trådene er ett sekvensielt program som deler i tillegg til at de kan ha hvert sitt private del av hukommelsen, og koden i trådene utføres ovenfra og nedover slik vi tidligere har lært at programmer oppfører seg. Har vi flere tråder i programmet vårt har vi et parallelt program.

Vi vil senere kommentere på nye typer av tråder og enklere begreper som kan bli implementert i Java.

1. Når man starter java-programmet, får vi én tråd: maintråden. Den tråden starter med å utføre metoden main(). Fra main-tråden kan så programmet vårt starte flere andre tråder som vi har skrevet kode for.
2. For å få disse andre trådene må de programmeres som en egen klasse som er en subklasse av klassen Thread – eller de kan være en klasse som implementerer grensesnittet Runable. Alle de objektene vi så lager av denne subklassen vil da inneholde en egen tråd som vil utføres i parallell med de andre trådene i vårt program.
3. Denne tråd-klassen, her kalt Para, legges helst inni den klassen som inneholder main() – metoden, og skal selv inneholde en metode public void run() { .. ] som dere må skrive selv. Denne metoden run() tilsvarer på mange måter i tråd-objektet main() metoden i hovedprogrammet, og er den metoden som kalles av systemet når tråden er laget og startet:

```
Thread[] t = new Thread [antTraader ];
for (int i = 0; i< c; i++) {
    t[i] = new Thread(new Para(i));
    t[i].start();
}
```

4. For senere i programmet å vente på alle disse `antTraader` stk. trådene blir ferdige, kan man utføre flg. setninger:

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

   Kurset inneholder fler andre måter, f.eks. vente på en egen CyclicBarrier i maintråden på at alle de trådene man har startet er ferdige.

5. Legg merke til at det nye trådobjektet er en parameter til klassen Thread.
6. Man kaller ikke si `run()` direkte, men med å kalle metoden `start()` i dette nye trådobjektet, som gjør mye bak kulissene for å lage denne nye tråden og til sist kalles `run()` i trådobjektet.
7. Trådobjektene starter altså med å utføre metoden `run()` som du har skrevet – denne metoden tilsvarer da metoden *main()* for maintråden.
8. En tråd avsluttes når den har utført siste setning i sin main() eller run() – metode.
9. Ingen tråder må drepe/avslutte en annen tråd, men ofte vil en tråd legge seg å vente på at en bestemt operasjon (utført av andre tråder) er ferdig.
10. Ikke før alle trådene, også main-tråden i et program er avsluttet, er programmet ferdig.
11. Alle disse trådene deler samme adresserommet i hovedlageret – de ser de samme variablene (data), classer og metoder som er deklarert der ut fra sitt skop (sitt utsyn til deklarasjoner).

12. Trådene utføres også samtidig og på hver sin kjerne i CPU-en hvis vi har en kjerne for hver tråd. Er det flere tråder enn kjerner, vil operativsystemet prøve å la trådene dele på bruken av kjernene. Dette ordner operativsystemet (Windows, Linux eller MacOS).
13. Operativsystemet har i tillegg en rekke tråder for å ulike oppgaver (administrasjon, I/O, nettet, vinduer på skjermen,…,osv). De denne setningen ble skrevet hadde Win10 operativsystemet 3789 tråder som var aktive, men nesten alle disse trådene lå og ventet. I sum tok de mindre enn 10% av maskinens kapasitet. Disse systemtrådene står i motsetning til de trådene vi skal skrive i IN3030 som hver vil forsøke, grovt sett, å bruke *hele kapasiteten* til en kjerne.

Med flere tråder har vi da et parallelt program-system hvor flere sekvensielle programmer kjører samtidig. De fleste problemene med slike parallelle systemer er når to eller flere tråder vil skrive på samme variabel. Vi må da synkronisere trådene (mer om det senere), eller hver tråd må lokalt ha en kopi av slike data og skrive/lese på disse. Senere samstilles data fra disse lokale kopiene.

## 2.3 NYE KLASSER, SØPPELTØMMING MM

For å lete kodeskrivingen er det i Java 17 innført begrepet 'record' som gjør det lettere å behandle klasser som bare inneholder data (eks: Punkt med en x og en y-verdi), Videre er det rent statiske classer som inneholder data som ikke skal endres etter at de er laget og metodenavn som parametere. Viktigst er kanskje at det er definert tekstblokker som er et antall linjer med tekst. Dette er syntaks som gjør det enklere å skrive kode, men siden det implementeres som 'vanlige' klasser er det ikke begreper eller tillegg til Java som gjør at paralelle programmer går fortere .
Det som gjør at Java 17 programmer nok går fortere enn Java 8 programmer er at søppeltømmingsalgoritmer i Java 17(dvs de objektene som er laget under kjøring og som nå ikke lenger kan brukes fordi ingen av trådene har en peker til dem) nå er byttet helt ut med den 'gamle' algoritmen med en ny som er mer inrementell, Ikke tar alt søppel med en gang, men gradvis fjerner 'søppel'.objekter.

Et annet prosjekt (Valhalla) prøver å lage en mer effektiv inn-utpakking av enkle variable , slik en 'int' blir pakket inn som en Integer med et objekt rundt seg, eksempelvis i ArrayList <Integer>, som klart tar lenger tid og plass. Dette kompendiet har pekt på dette problemet i kap 13, hvor en enkel int[] nå klart er raskere enn en ArrayList<Integer>. Arbeidet med å få basale typer (som byte, int og double) og den motsvarende (Byte, Integer og Double) inn som felles begreper med felles metoder vil bli først komme i Java 22 i 2023 eller 2024 (https://openjdk.org/projects/valhalla/). Poenget med Valhalla er å endre hvordan generiske typer som Integer er definert i Java slik at man kan blande basale typer og generiske typer og at generiske typer kan både få mer effektivitet og ikke mer plasskrevene enn basale typer som 'int'.

# 3 MER OM PARALLELLE PROGRAMMER I JAVA

Vi kan altså få altså feil i programmene våre hvis mer enn én tråd samtidig skriver på en variabel som er felles. Vi ser at alle andre trådene som ønsker å skrive samme stedet da må stoppes og vente mens den første tråden blir ferdig med å skrive. Dette kaller vi å synkronisere trådene. I tillegg sørger for at alle tradene ser samme verdier i alle felles datastrukturer

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 barriere-synkronisering, men starter med en gjennomgang om både hvorfor vi trenger å programmere mer parallelt og særlig hvorfor det så komplekst.. Etter denne gjennomgangen kan vi lett få inntrykk av at det er umulig å få parallelle programmer riktige. Det er ikke tilfellet, men for å få til det må man følge noen klare og enkle regler. Bryter man bare én av disse, vil det kunne gå veldig galt.

## 3.1 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 den. 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 i dette kurset 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 eksempelet vi til sist skal se på, sortering av større datamengder.

Vi skal i det etterfølgende 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.

# 4. OM BRUK AV SYNKRONISERINGSPRIMITIVER OG LÅSER

Vi har sett at det er 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 og la en eller flere tråd(er) vente. Tråden kaller på låsen og låsen gjør 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 variable ***før dette kallet*** er synlig for alle de andre *etter kallet*. De ser da samme verdier på felles variabler**.** Da blir bla. alle felles variabler for disse trådene som er endret skrevet ned i hovedhukommelsen fra cashene før trådene kan fortsette.
I Java er det mange titalls forskjellige låser som kan synkroniserer trådene. Vi skal i hovedsak i våre løsninger bare bruke tre av disse som er kortfattet beskrevet i neste avsnitt. Men for å illustrere visse problemer skal vi også i kurset nytte noen andre slike låser senere. Til alle slike låser/synkroniserings-mekanismer er det en rekke metoder (totalt 19 stk. for ReentrantLock) hvor en bruker kan spørre om hvor mange andre tråder som venter på denne låsen, spørsmål om å neste i køen
osv.
Vi vil illustrere dette senere med programmer som benytter tre typer av låser: CyclicBarrier som sikrer at vi vet når aller trådene er ferdige og som vil stoppe tråder og la dem vente inntil alle trådene er ferdige med en bestemt del av koden. En nyttig og den raskeste låsen er ReentrantLock for å beskytte en bestemt metode fra at flere tråder samtidig kan utføre den metoden. Bare en tråd slipper inn ad gangen og andre tråder som litt senere kaller denne metoden må vente til den første tråden er ferdig (en tråd av gangen). Til sist skal vi bruke Javas innebygde *synchronized* mekanisme som skal sikre at bare én tråd av gangen er inne i en av alle de metoder som er 'beskyttet av' synchronized-ordet i dette objektet. Hvis metodene er deklarert som static, vil denne synkroniseringen gjelde i alle objekter av denne klassen hvor denne beskyttelsen er brukt, men hvis metodene ikke er static, vil metode-beskyttelsen bare gjelde metodene i ett objekt ad gangen. Synchronized (som er et reservert ord i Java) kan beskytte en hel metode eller bare en blokk med setninger inne i en metode.

## 4.1 HVORDAN VIRKER SYNKRONISERING AV TRÅDER

Tråder er altså hver selvstendige sekvensielle programmer som kjører samtidig, vi sier i parallell, på én multikjerne PC.For å få adgang til CyclicBarrier, ReentrantLock som begge er klasser som man finner på Java.biblioteket ved å ha følgende import-setninger i toppen av program-koden :

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

Vi lager først ett objekt av en slik synkroniserings-klasse, og det er bare de trådene som kaller metodene i dette objektet som synkroniseres med hverandre (ved å kalle f.eks metoden await()).

I tillegg skal vi bruke synchronized som er bygget inn I programmeringsspråket Java og som kan sørge for at bare én tråd av gangen kan utføre en slik metode av alle de metodene i dette objektet med synchronized ordet foran seg i deklarasjonen. Vi kan bare synkronisere tråder i forhold til hverandre som bruker samme synkroniserings-objekt og alle objekter kan synkroniseres.
Et enkelt eksempel er at vi ønsker å debugge et parallelt program med tråder, og at vi ønsker å skrive ut verdiene av en variabel i en slik

De er altså alle tre mekanisme som greier å stoppe andre tråder midlertidig når en tråd skal skrive på data, en fil eller skjerm som er felles for alle trådene.

## 4.2 OM BRUK AV REENTRANTLOCK

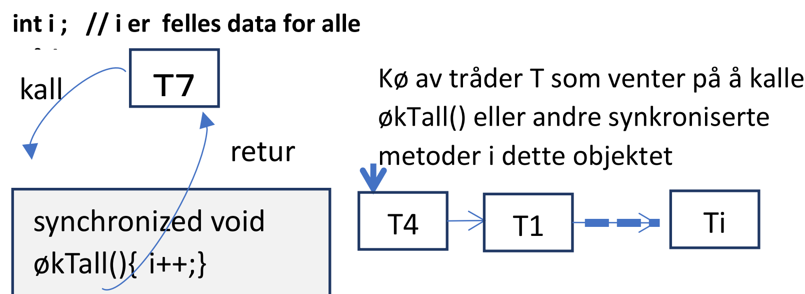
Den enkleste og raskeste låsen er  ReentrantLock:

```
class X {
   private final ReentrantLock lock = new ReentrantLock();
   // ...

   public void m() {
     lock.lock();  // block until condition holds
     try {
       // ... method body
     } finally {
       lock.unlock()
     }
   }
 }
```

***Program  3.1*** *Kode for bruk av ReentrantLock som beskytter en metode m() fra å bli brukt av høyst en tråd.*

### 4.2  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 (fig   1.3). Kallende tråd får gjort seg helt ferdig og resultatet blir skrevet ned i hovedhukommelsen før neste tråd får utføre sitt kall.




***Figur  3.1*** *Synkroniserte metoder lar én tråd slippe til av gangen og køer opp andre tråder som evt. samtidig ønsker å gjøre et kall på synkroniserte metoder i alle objekter av denne klassen. Her holder tråd T7 på med et kall på økTall(), og mange andre tråder som har forsøkt å gjøre kall mens T7 holder på å få utført sitt kall, 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 samme objektet, vil denne låsen i objektet sperre også for samtidige kall fra andre tråder på disse andre metodene. I ett objekt kan altså høyst en synkroniserte metode bli eksekvert av gangen. Merk at hvis man har flere objekter av den klassen hvor de synkroniserte metodene er, er det mulig å få utført en synkronisert metode samtidig av to eller flere tråder. Det kan være en utilsiktet feil, hvis kallene kommer til ulike objekter av denne klassen.
Vi kan først prøve å kjøre programmet i 3.1. 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 om hvor mange ganger hver av dem skal øke heltallet tall med 1.Vi bruker her en long, 64 bit heltall for variabelen tall fordi summen av antall opptellinger (antGanger*antTråder) kan bli større enn største verdi for et 32 bit heltall. Vi kaller så metoden utfør() i dette objektet p av klassen Parallell.
Merk at:

> Hvert objekt som skapes med new har en lås. Det er den låsen som nyttes ved kall på alle de synkroniserte metoder i det samme objekt som metoden er i. Kall på en synkronisert metode (den samme metoden eller en av de andre) fra en annen tråd i det samme objektet vil da bli låst fordi to slike metodekall nytter den samme låsen – dvs. det samme objektet. Når det første kallet er ferdig, slipper en av de andre trådene til med sine kall,… osv.

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

/** Start >java Parallell <ant tråder> <ant ganger i løkke> */
class Parallell{
      long  tall=0;   // Sum av at 'antTråder' tråder teller opp denne
      CyclicBarrier b ;  //sikrer at alle er ferdige før vi tar tid og
sum
      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)+»%»);
     } // end utskrift

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

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

     void  utfør () {
        b = new CyclicBarrier((int)antTråder+1);  //+1, også main venter
        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);
       } // utfør


 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;}
              } // end run

              //    void økTall() { tall++;}              // 3)
              //    synchronized void økTall(){ tall++;}  // 4)
       } // end class Para
}// END class Parallell
```

***Program 3.2.*** *Et program som viser både riktig og gal bruk av låser i Java. Vi ser 4 ulike plassering 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 alt for liten samlet sum i tall. Kjører vi et feilaktig program flere ganger med samme parametre 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 1.2 viser et eksempel på en slik feil. Hvis vi 'av-kommenterer' plassering 4) og nytter 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 definert på denne måten, låses 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.

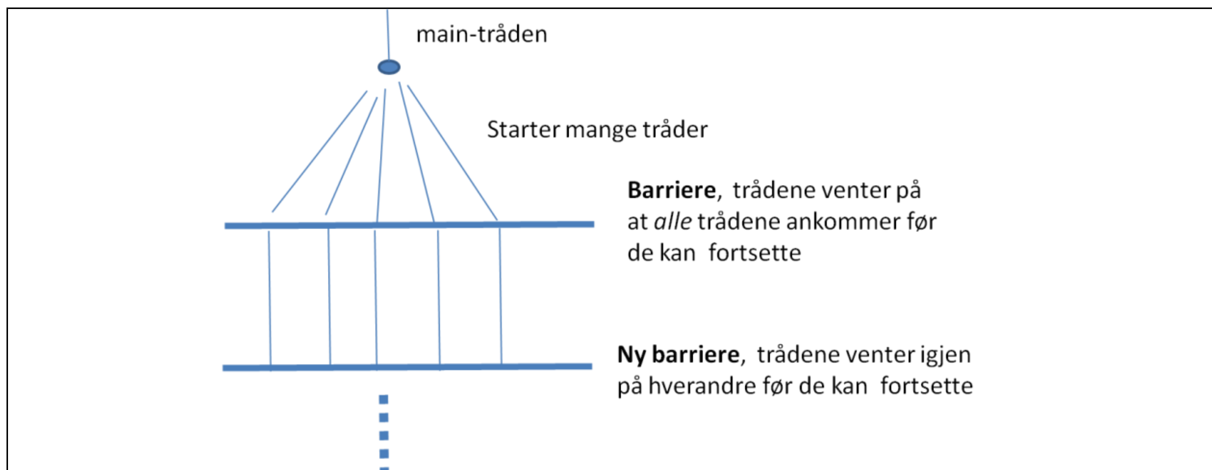
| | **1** | **2** | **3** |
|---|---|---|---|
| Svar **uten** synchronized **3)** | 7 112 531 | 5 911 630 | 6 169 492 |
| Svar **med** synchronized **1)** | 10 000 000 | 10 000 000 | 10 000 000 |

**Tabell 3.1** *Tre kjøringer hvor vi starter 100 tråder som hver forsøker å øke variablen i 100 000 ganger med 1, dvs. i skulle da bli = 10 mill. Vi ser at uten at økTall() er riktig synkronisert, får vi mange feil og ulikt svar i hver kjøring.*

Hvis vi derimot nytter plassering 1) for den synkroniserte metoden økTall(), ser vi at den er inne i ett objekt av klassen Parallell. Det er bare dannet ett objekt av denne klassen, og alle kall på økTall()nytter da samme lås, og alt går da 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 parametere. 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 da ikke et skikkelig parallelt program, men at trådene utføres etter hverandre, sekvensielt.

# 5 BARRIERE SYNKRONISERING

Ikke alle beregninger kan greit eller effektivt løses 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 at vi vil 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 5.1** *Programmet starter alltid først bare en tråd – maintråden. Den lager ett objekt b av klassen CyclicBarrier og et større antall tråder. Hvis barrieren er laget for **k** tråder, så vil alle tråder, også evt. maintråden, vente hvis de sier b.await() inntil **k** tråder har sagt b.await(). Da slipper 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 køe opp; og 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 5.1.*** *Et program som skisserer riktig bruk av CyclicBarrier i Java.*

Vi vil i neste programeksempel se at vi har en parameter som er 1 større enn antall tråder vi lager, fordi vi bruker den sykliske barrieren b til at alle trådene og main-tråden venter 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 helt ferdige og som main-tråden også har lagt seg til å vente 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() på en barriere sørger ikke bare for at alle venter, men også at alle etterpå kan se alt hva de andre skrev på felles variable før await()-kallet.

## 5.1 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();
       } // end main

      void  utfør () {
                a= new int[antTråder*antTråder]; // større problem
                ant= a.length/antTråder;   // antall 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); //+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 i a:"+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.");
        } // 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 run
         } // end class Para
}// END class Parallell
```

***Program 5.1.*** *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 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 forhåpentligvis 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, er det ingen vits i å parallellisere. Den ekstra tiden det tar å starte og stoppe tråder tar da langt lenger 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.

# 6 GENERELT OM PARALLELLISERING AV ALGORITMER, DEL 2

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 søke å dele dataene i like store deler, ofte langt flere enn du har kjerner – f.eks 20* antKjerner. Grunnen til dette er at en tråd som løser et slikt delproblem kan få ventesituasjoner, og da er det greit at en annen tråd er i stand til å eksekvere. Imidlertid må det advares mot alt for mange tråder. Det tar tross alt noen få milliekunder å skape og starte en tråd. Vi kan godt bruke 10-500 tråder for å løse et problem, men ikke mange 10-tusner.
3. Start en tråd for hver av disse delene av problemet.
   Dette gjør vi hvis ikke oppdelingen og starting av tråder er avhengig av hvilke, og hvor mye data vi har. Da skjer 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 den sekvensielle algoritmen nyttes 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 del-svarene til en løsning på hele problemet. Av og til er det ikke nødvendig, men ofte er det slik at det er noen avsluttende beregninger.
7. Når du skal debugge et slikt parallelt program og f.eks. ønsker å skrive ut verdier av en eller flere variable i de ulike trådene, virker det dårlig å nytte System.out.println(String s) fordi den ikke er synkronisert og du vil oppdage at utskriften fra de ulike trådene blander seg på skjermen i en ikke lesbar mølje. Dette løser du enkelt ved å lage flg. variant av println:

   **synchronized void println(String s){System.out.println(s);}**

   Da vil Javas synkronisering sørge for at utskriften fra de ulike trådene *ikke* blander seg.

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

# 7 OM PARALLELLISERING MED STREAMS

I Java 8 og senere versjoner, nå er det innført begreper som stammer mer fra funksjonell programmering – her spesielt lambda-uttrykk og strømmer (streams). Grunnen til å nevne det her er at her synes det å være er programmerings-vennlig måte å parallellisere et program . Først en kort innføring i selve begrepene: lambda-utrykk og sekvensielle strømmer og deretter et eksempel på en vellykket parallellisering med parallelle strømmer, og til sist kommentarer til når denne mekanismen er egnet til parallelisering.
(Kan du vanlige sekvensielle strømmer og lambda-uttrykk kan du hoppe over avsnittene 7.1-7.5.)s

## 7.1 Lambda-uttrykk og strømmer (streams)

I Java 8 ble det innført et nytt nytt begrep som kalles *lambda-uttrykk*. Enkelt sagt er dette en kompakt måte å skrive (enkle) metoder som vi skal utføre. Kompilatoren *javac* finner da ut hvilke typer parametre vi har i *lambda-metoden* slik at vi kan slippe å skrive det, og mye annet vi kanskje ellers må skrive for å lage en metode. Koden vår blir kortere og mer lettlest.

Vi skal her gå gjennom hvordan *lambda-uttrykk* skrives, deklarasjoner og bruk. Slike uttrykke egner seg vel mest for små korte metoder som kan skrives på noen få linjer. Alt vi kan gjøre med *lambda-uttrykk* kan vi også få til med vanlige metode-deklarasjoner og bruk, men riktig brukt er *lambda-uttryk* mer elegant, lettere forståelig og gir brukere langt mindre å skrive særlig når *lambda-uttrykk* kombineres med *streams*.

## 7.2 Hvordan skrive lambda-uttrykk

Det er to måter å skrive lambda-uttrykk:

(parametre) -> uttrykk **eller** (parametre) -> { setninger;}

Begge tar parametre spesifisert på venstre side og bruker den på høyre side til å evaluere et uttrykk eller utføre setningene i klammeparentesen. Et lambda-uttrykk kan returnere en verdi eller bare gjøre noe.

Her er eksempler på lambda-uttrykk:

```
1. () -> 7              // har ingen parameter og returer 7
2. x -> 2 * x           // en parameter (x) og returnerer den doble verdien
3. (x, y) -> x – y      //tar to tall inn, returnerer differansen
4. (int x, int y) -> x + y          // tar to int inn, returnerer summen
5. (String s) -> System.out.print(s) // Tar en String s og printer den
```

Følgende er valgfritt når vi skriver *lambda-uttryk* :

- Type-deklarasjoner på parametre (kompilatoren finner det ut)
- Parentes rundt parametre hvis vi har bare en parameter.
- Krøll-parenteser {} rundt høyresida hvis det bare har en setning.
- Bruke av *return*-ordet i høyresida. Kompilatoren returnerer bare siste verdi beregnet, men utelater vi *return* må vi ha krøll-parenteser.

## 7.3 Hvordan lages lambda-uttrykk

Når vi lager et *lambda-uttrykk,* må vi ha et grensenitt med en metode som har samme antall og typer på parametrene som det vi trenger i vårt *lambda-uttrykk.* Hvis du enda ikke har lest kapittelet om grensesnitt, så er *grensesnitt* enkelt forklart en java – konstruksjon som ser ut som en meget forenklet klasse-deklarasjon. Et grensnitt begynner med ordet *interface* (i steden for class) og inneholder bare spesifikasjon av metoder (deres navn, parametre med typer og metodene returverdi, men *ikke* kode inne i metodene). Her er tre eksempler du selv kan skrive i ditt program:

```
interface  Calc{
      double beregn (int x);
}
interface  Hei{
      void si (String s);
}

interface  Matte{
      int oper (int x, int y);
}
```

Merk at grensesnitt for å lage *lambda-uttrykk,* kan bare har én metode. (Java-bibliotekene inneholder også mange slike grensesnitt som kan brukes til å lage *lambda-uttrykk,* men du kan like godt deklarere selv de grensesnitt du trenger):

I programmet ditt kan du så lage flere ulike *lambda-uttrykk* fra samme metode i samme grensesnitt når de bare har samme antall og type av parametre og samme returverdi.

### 7.4 Hvordan bruke Lambda

Før vi kan bruke et *lambda-uttrykk* må vi altså koble det uttrykket vi vil bruke med et grensesnitt med samme antall og type parametre. Vi navngir da et *lambda-uttrykk* (egentlig navngir vi et objekt av en klasse som lager (implementerer) som den metoden som er i grensesnittet) slik :

```
Hei joa  = (s) -> System.out.println("Jeg sier:"+ s);
```

Og vi kan kjøre denne metoden slik (i en av våre vanlige metoder som main):

```
joa.si(« Lambda er bra»);  // ut: Jeg sier: Lambda er bra
```

Her er er flere eksempler:

```
Calc areal = x -> x*x*3.14159/4;
Calc radius = x -> x*1.0/2;
Matte mult = (x,y) -> { return x*y;};
```

Og det kan brukes slik:

```
Scanner keyboard = new Scanner (System.in);
System.out.print ("Gi et heltall: ");
num = keyboard.nextInt();

System.out.println(" Sirkel med diam:"+num+" m har areal:"
 + areal.beregn(num)+" m2, og radius:"
 + radius.beregn(num)+"m");
System.out.println("3*7=" + mult.oper(3,7) );
joa.si(" - dette er bra");
```

Vi ser at grensenittet Calc har to ulike implementasjoner (areal og radius). Kjører vi programmet over får vi:

```
Gi et heltall: 137
Sirkel med diam:137 m har areal: 14741.1256775 m2, radius:68.5m
3*7=21
Jeg sier: - dette er bra
```

For å oppsummere: *lambda* gir ikke noe nytt, men du skriver mindre kode for å få det utført. I neste delkapittel ser vi hvordan *lambda-uttrykk* med brukes sammen med strømmer.

## 7.5 Strømmer (streams) fra mengder

Strømmer i Java er kort fortalt å kunne stille spørsmål som minner om SQL spørsmål fra database-verden. Vi har en startmengde i Java (List, ArrayList, array, HashMap,..) som vi først omgjør til en strøm av enkeltobjekter og fra denne lager vi en ny mengde (svar-mengden), som er de fra startmengden som tilfredsstiller de kravene vi kommer med i den strømmen vi lager. Ta et eksempel fra programmet nedenfor om personer, deres navn, kjønn, inntekt osv. Vi kunne være interessert i å vite hvilke

personer som tjener over 1 mill kr. Da søker vi ut en ny mengde. Vil vi bare ha navnet til én person med så stor inntekt, søker vi etter en string. Vi kan også være interessert i bare å vite hva er snittinntekten til de som tjener over 1 mill. kr. Vi søker da et tall. Vi kan også være interessert i finne kvinners inntekt.
Vi kan altså både søke ut en ny mengde, men også tall, Stringer, ol.

```
import java.util.*;

class Person{
    String navn; String kjonn ; int alder; int lonn;
    Person (String na, String kj, int ald, int ln) {
          navn = na; kjonn = kj; alder =ald; lonn = ln;
    }
 }

 public class StreamEks{
     ArrayList <Person> alle = new ArrayList <Person> ();

     public static void main (String [] args) {
          new StreamEks().doIt();
    }
    void doIt() {
       alle.add(new Person("Ola",  "M", 55,1200000));
       alle.add(new Person("Kari" ,"F", 44, 600000));
       alle.add(new Person("Jonas","M", 65, 110000));
       alle.add(new Person("Tora", "F", 12,  10000));
       alle.add(new Person("Arne", "M", 69, 2000000));

      // 1) skriv alle som tjener mer enn 1. mill
        alle.stream()
         .filter(s-> s.lonn > 1000000)
         .forEach(p->System.out.println(p.navn+
                     " tjener kr. "+ p.lonn+" per år"));

      // 2) Finn person med størst inntekt
        int ml =
          alle.stream()
           .mapToInt(s -> s.lonn)
           .max()
           .getAsInt();

     // 3) finn gjennomsnittsinntekt for alle
          alle.stream()
           .mapToInt(p -> p.lonn)
           .average()
            .ifPresent(t -> System.out.println("snitt lønn= "+ t));

      // 4) Beregn snitt inntekt for de med lønn > 1.mill
           alle.stream()
             .mapToInt(p -> p.lonn)
             .filter (r -> r > 1000000)
             .average()
             .ifPresent(t -> System.out.println(
               "Snitt de over 1.mill = " +t+ " kr."));

     // 5) Beregn snitt kvinners lønn
          double kvinneLønn =
             alle.stream()
             .filter(p -> p.kjonn == "F")
             .mapToInt(p -> p.lonn)
             .average()
             .getAsDouble();

            System.out.println("Gjsnitt kvinnelønn er kr."+
               kvinneLønn+", max lønn er:"+ ml);
```

```
        // 6) Finn første kvinne med lønn < kr. 100 000
               alle.stream()
                  .filter(p -> p.kjonn == "F")
                  .filter (r -> r.lonn < 100000)
                  .findFirst()
                  .ifPresent(t -> System.out.println(
                            "Lavlønnet kvinne er:"+t.navn));

     } // end doIt
} // end StreamEks
```

Resultatet fra kjøring er:

```
Ola tjener kr. 1200000 per år
Arne tjener kr. 2000000 per år
snitt lønn= 784000.0
Snitt de over 1.mill = 1600000.0 kr.
Gjsnitt kvinnelønn er kr.305000.0, max lønn er:2000000
Lavlønnet kvinne er:Tora
```

Forklaring til alle eksemplene er at vi omgjør mengden vår først til en strøm (stream()) av enkelt-elementer, her Person-objekter. Videre i strømmen er det enten funksjoner som begrenser hvilke objekter som kommer videre (eks. filter()) , funksjoner som omgjør et objekt til en annen bestemt type (eks MapToInt()) og funksjoner som ber om alle (som max(), average(), sum(), forEach()) eller som bare ber om ett eksemplar (findFirst()). En slik strøm drives fremover ved det stadig bes om nye elementer, og hvis den funksjonen som er sist eller nest sist i har blitt tilfredstillet, som i eks 6 at vi finner første kvinne med lav lønn (under 100 000). Se dokumentasjonen til grensesnittene java.util.stream og java.util.collection i Java-dokumentasjonen.

Grunnen til at at vi ikke behøver å skrive egne grensesnitt her er at de bibliotekene vi her bruker i java.util har deklarert det vi trenger med riktige typer på parametre og returverdier. Vi ser også at vi må prøve oss frem når vi tak et gjennomsnitt og må i Eks.3 konvertere det som 'flyter nedover strømmen' før vi kan ta average(), eller i Eks. 5 må vi også konvertere resultatet til en ekte double-verdi før vi kan gjøre tilordning til en enkel double-variabel.

## 7.5 Parallelle strømmer

Her er en meget enkel metode som avgjør om parameteren er primtall eller ikke. Dette er ikke en effektiv metode (Eratosthenes sil er langt raskere) og er bar tatt med her som eksempel på en metode som bruker en del tid – den dividerer parameteren med 2 pluss alle oddetall < enn kvadratroten av parameterverdien. (Dette eksempelet har jeg fått av prof. Peter Sestoft, IT-Universitet i København og omarbeidet noe.

```
/* returnerer true hvis p er primtall */
private static boolean isPrime(int p) {
       if (p % 2 == 0) return p == 2;
       int k=3, k2=9;

       while ((p%k != 0) && k2 <= p ){
              k+=2;
              k2=k*k;
       }
       return  k2 > p;
}// end isPrime
```

Vi kan nå lage en sekvensiell strøm og parallell strøm som bruker isPrime og senere et sekvensielt og parallelt program slik vi hittil har lært i IN3030 for å sammenligne effektiviteten av slike strømmer.

```
//    Sekvensiell strøm:
          int antall=
             IntStream.range(2,n)
             .filter (i-> isPrime(i))
             .count();

//        Parallell strøm:
          int  antall=
               IntStream.range(2,n)
               .parallel()
               .filter (i-> isPrime(i))
               .count();
```

Legg merke til at at det eneste som settes inn i programmet for den sekvensielle strømmen er en kall på funksjonen som parallellisere strømmen av tall som kommer fra IntStream.range(2,n) som genererer tallene fra og med 2 til (men ikke med) n. Siden dette kjøres på 4 kjernes CPU som på grunn av at hver av disse er hyperthreaded, så vil operativsystemet si at den har 8 kjerner. Legg merke til hvor enkelt det er å parallellisere en strøm, og siden utviklerkostnadene (lønn) er langt viktigere enn kostnadene til en server, vil dette være en kost-effektiv måte å paralleliser problemer som er enkelt parallelliserbare.

For å lage et sekvensiell vanlig program  som gjør det samme som strømmeløsningene må vi skrive tilsvarende funksjoner.  IntStream.range(2,n) programmeneres som en vanlig for-løkke. Så må vi lage en metode som teller opp hvor mange primtall vi finner < n ( tilsvarer funksjonen filter () og count i strømmeløsningene:

```
/* returnerer anttall primtall p < n */
int tellPrim (int n) {
    int count = 0;
    if ( n>2 ) count = 1;// because of 2

    for (int i =3; i < n; i+=2)
         if ( isPrime(i)) count++;
    return count;
} // end tellPrim
```

For å lage et parallelt program som løser dette problemet må vi i tillegg lage en metode som teller hvor vi finner antall  primtall mellom a ≤ p <b. Vi deler da problemets data mellom de ulike trådene:

```
/* parallell løsning, returnerer antall primtall p for a<= p <b */
 int tellPrim (int a, int b) {
      int count = 0, size = b-a;
      if (size <2) {
             if (a == 1)          return 0;
             else if (isPrime(a)) return 1;
      } else {
           if (a%2 == 0) a++;   // test only odd numbers

           for (int i =a; i < b; i+=2)
                 if( isPrime(i)) count++;
      }

      return count;
} // end tellPrim parallel
```

Så kommer koden til Arbeiderne som finner antall primtall i hver sin del av talllinjen. I tillegg må ha kode som starter trådene og som så venter på at alle Arbeider-objektene terminer og som så summererer verdiene i 'tell'-arrayen.

```
class Arbeider extends Thread { //implements Runnable {
              int ind;
              int left,right;
              // lokale data og metoder

              Arbeider (int in, int left, int right) {
                        ind = in;
                        this.left = left;
                        this.right= right;
               }

               public void run( ) {
                       tell[ind] = tellPrim(left,right);
               } // end run

 } // end indre class Arbeider
```

Legg merke til at vi skriver vesentlig mer kode vi får når vi lager parallelle programmer i IN3030 enn man gjør med parallelle strømmer. Den eneste funksjonen vi skriver med parallelle strømmer er: 'isPrime(int p). Siden lønningene til systemutviklere er den største utgiften i ethvert prosjekt favorisere det klart strømmer.

Tidsforbruket for de 4 algoritmene vi testet (sekvensielt 'vanlig' program, sekvensiell strøm, vanlig' parallelt program og parallelt strøm) for ulike verdier av n =1,,1 000 000 er (merk logaritmiske verdier på begge akser fordi verdiene varierer så mye):

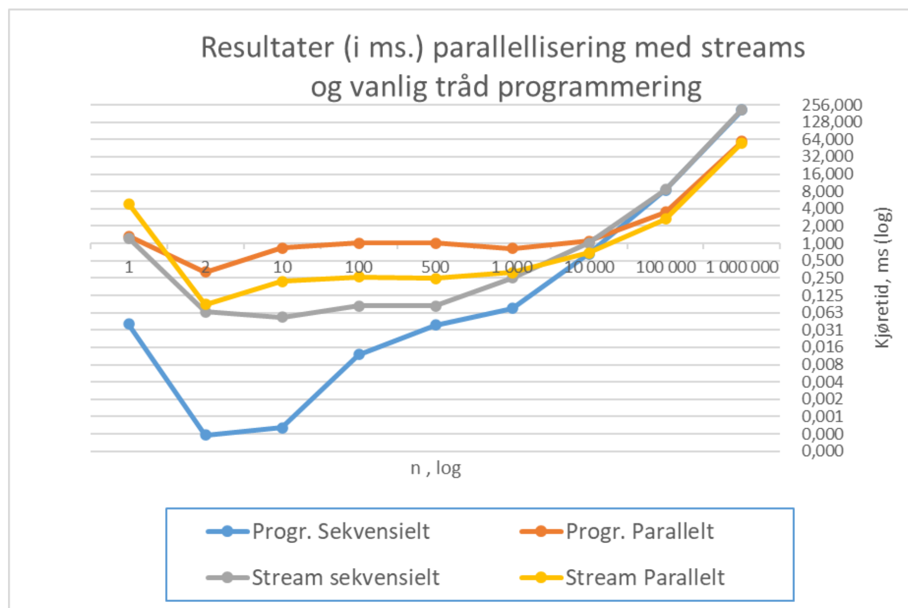



**Fig 7.1** *Eksekveringstiden for de 4 ulike metodene ('vanlig' sekvensiell og parallell løsning) og strømme løsning(sekvensiell og parallell).*

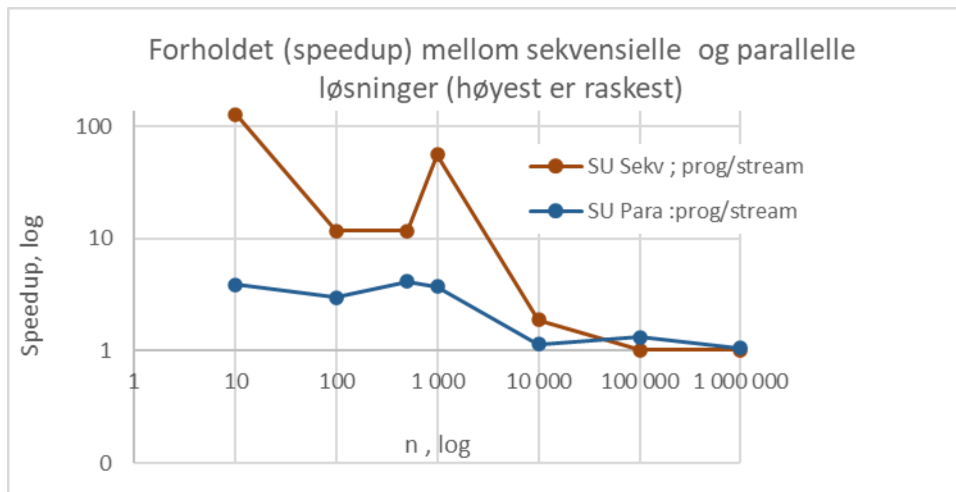
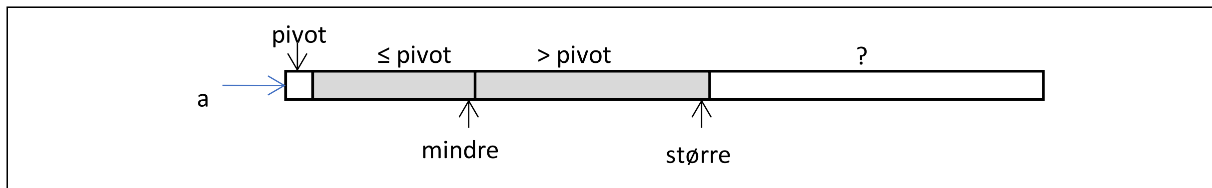



**Fig 7.12** *Forholdet mellom 'vanlig' program og strømme-løsning (speedup sekvensiell og parallell).*

Vi ser av fig. 7.1 og 7.2 at det sekvensielle vanlige programmet er over 100 x raskere enn en sekvensiell strøm for små verdier av n, men når kjøretidene er >1 ms for n > ca 20 000, forsvinner denne klart raskere faktoren, og at de parallelle versjonene av både strømmer og vanlig program er om lag rike raske og begge de to parallelle løsningene er nesten 4 ganger raskere enn de sekvensielle løsningen – fordi vi har 4 kjerner. Konklusjonen må bli at parallelle strømmer må foretrekkes der de kan brukes på store strømmer, og selv om det er 'mye' langsommere for mindre vedier av n, antall objekter i strømmen er kjøretidene så små at det vel ikke er viktig. Fordelen med strømmer er at mye triviell kode er fjernet og er programmeringstiden er vesentlig kortere og enklere.
En klar begrensning med strømmer er at man kan ikke endre mengden strømmen starter med under kjøring og at man parallelliserer hvert element i strømmen, noe som vel ikke er mulig med Eratosthnes algortme. Vi ser også at parallelliseringen i strømmer er per element i strømmen, mens vi parallelliser i IN3030 måten å programmere på gir hver tråd hver sin separate del av talllinja.

# 8 KVIKKSORT, PARALLELLISERING AV EN REKURSIV ALGORITME

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. Parallelliseringen er endringer til det sekvensielle programmet.
Vi skal da presentere først en sekvensiell sorteringsmetode KvikkSort (eng. QuickSort), laget av Tony Hoare i 1962. Den fikk en enkel utforming av Nico Lamuto som vi nytter her. Senere skal vi parallellisere denne- Idéen 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 alle de elementene som er > pivot. For å forenkle koden plasserer vi selve pivot elementet helt til venstre i den delen vi nå sorterer. Når vi er helt ferdige med sorteringen, plasseres 'pivot' elementet mellom de to delene, Arrayen a er da ikke ferdig sortert, men hvis vi nå for hver av de to delene gjør samme type oppsplitting *med nye valg av pivoter*, så kan de igjen oppsplittes gjentatte ganger til alle de mange delene til slutt har en lengde på 1 eller 0. Da er a[] sortert fordi ethvert element da står til høyre for et annet element som er mindre eller lik dette.




***Figur 8.1*** *Idéen 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, så alle de som er større enn pivot til høyre og pivot midlertidig helt til venstre. Til slutt plasseres pivot mellom de to delene 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 gjør denne sorteringen med oppsplitting gjentatte ganger, og som måler tiden og skriver ut denne:

```
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 8.1*** *Skjelettet for programmet som starter i main, som 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 8.2** *Koden for konstruktøren, som oppretter a[] og fyller den med tilfeldige tall <n; og* utførOgTest(), som tar tida med System.nanoTime() som gir tida i nanosekunder (milliardedels sekunder). *Den kaller så sQuick(), skriver ut tida i millisekunder og gjør en enkel test på om sorteringa 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 innbygde sorteringsalgoritme: java.util. Arrays.sort, og så sammenligne de to sorteringene element for element. Man kunne også ta tida 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];
      int    større = lav+1,  // hvor lagre neste '> piv'
             mindre =lav+1;  // hvor lagre neste '<=  piv'

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

      while (større <= høy) {
            if (a[større] < pivot) {
                  // a[større] er 'mindre' - bytt
                  bytt(a,større,mindre);
                  ++mindre;
            }
            ++større;
      } // end gå gjennom a[lav+1..høy]

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

      if ( mindre-lav > 2)  sQuick (a, lav,mindre-2); // sorter <= pivot
      if ( høy-mindre > 0)  sQuick (a , mindre, høy); // sorter > pivot
  }// end sQuick
```

PROGRAM 8.2 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 de som er ≤ **pivot** og ett på den delen hvor de som er > **pivot** ligger. Selve logikken til oppdelingen illustreres av fig 1.6. 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.

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 todeling (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 de 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 hvis vi ikke gjør dette 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 delen 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 byttet 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 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 sortingsmetoden **Arrays.sort(..)** i biblioteket **java.util**, som er en annen og mer komplisert koding av Kvikksort.

# 10 EN BLANDET PARALLELL OG REKURSIV KVIKKSORT

Grunnidéen i en parallell versjon av kvikksort er at vi bytter ut de rekursive kallene med at vi i stedenfor 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 (fig. 2.1). Vi må derfor lage en blandet algoritme som nytter tråder når vi f. eks deler opp en del av arrayen som er lenger enn 50 000 elementer, men som nytter rekursjon for kortere deler.
Skjelett-koden til programmet for parallell sortering en ganske likt det for sekvensiell kvikksortering:

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

class ParaQuick {
      int [] a;
      int antTråder = Runtime.getRuntime().availableProcessors();
      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
```

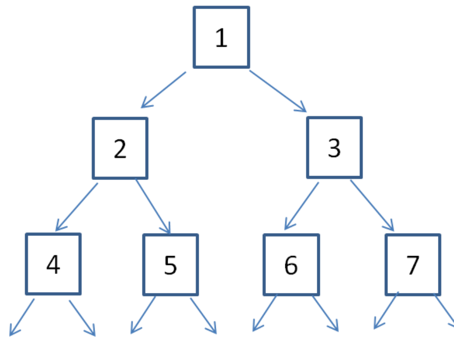
***Program 10.1*** *Skjelettkoden for parallell kvikksortering. Konstruktøren paraQuick har samme kode som konstruktøren SeqQuick i prog. 1.6. Også metoden bytt er identisk med den sekvensielle bytt.*

Vi ser at internt i klassen **ParaQuick** har vi i tillegg til arrayen **a**, en variabel **antTråder** som spør operativsystemet hvor mange kjerner vi har og da hvor mange tråder ***k*** vi kan starte i parallell på denne maskinen. 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];
      int    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] , sortér resten

      while (større <= høy) {
         if (a[større] < piv) {
            bytt(a,større,mindre);
            ++mindre;
         }
         ++større;
      }

      bytt(a,lav,mindre-1);        // Plassert '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(); //  <= piv
         new Thread(new Para(b2,a,mindre,høy  )).start(); //   > piv
         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);//  <= piv
          if ( høy-mindre > 0) pQuick (null,a, mindre, høy); //  > piv
      }

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

***Program 10.2*** *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 3 tråder – de to nye trådene som startes og den tråden som skaper disse. Ventesetningen like etter at de to trådene som skapes er at den tråden som skapte disse to trådene, venter på at de begge er ferdige. Helt på bunnen av metoden ser vi en ny await() kall. Det er signalet til den tråden som kalte denne metoden, at denne tråden er ferdig – altså et signal oppover.*

Koden forklares i teksten under program 9.1. I oppgave 2 skal dere prøve ut og forklare tidsforbruket uten å kode dette med rekursiv løsning for kortere deler, men bare med tråder.




***Figur 10.2*** *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.*

## 10.1 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 1.2 og 1.3.

| | 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 | 11196 |
| Parallel | 0,01 | 0,06 | 0,84 | 16 | 24 | 67 | 356 | 3579 |
| SU (Speedup) | 1,00 | 0,83 | 0,84 | 0,81 | 0,75 | 1,32 | 2,71 | 3,12 |

***Tabell 10.2*** *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) og Speedup (sekvensiell/parallell). Kjørt på en Intel i7 870, 3Ghz 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 | 16497 |
| Parallel | 0,01 | 0,06 | 0,99 | 21 | 56 | 106 | 1332 | 5025 |
| SU (Speedup) | 1,00 | 0,83 | 0,82 | 0,85 | 0,38 | 1,85 | 1,06 | 3,28 |

***Tabell 10.3*** *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) og Speedup (sekvensiell/parallell). Kjørt på en maskin med 4 Intel Xeon prosessor L7555, 1.87Ghz 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) 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 nivå2, og for 100 000 tall i nivå3 cachen og hovedhukommelsen som jo er mye langsommere.

Det beste vi synes å oppnå er at den parallelle versjonen går om lag 3 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 alltid hovedforklaringen. I figur 19.2 ser vi en strek som forbinder hver prosessor med hovedhukommelsen. Det kalles en hukommelses- eller data-kanal, og kan ikke gå fortere enn hukommelsen kan 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 køing på hukommelseskanalen har bare problemer som har lite data og som gjør mer arbeid med hvert dataelement enn de problemene vi har sett på her. Et slikt problem ("Selgerens rundreise") ser vi på i oppgave 3. Da vil data som det jobbes med i kjernen i stor grad være in cache-hukommelsene og antall lesinger og skriveoperasjoner i hovedhukommelsen vil ikke skje så ofte at det skaper kø på hukommelseskanalen. Parallelliseringen av kvikksorteringen kan imidlertid ikke sees på som mislykket – at store sorteringer går tre ganger så fort er klart noe man vil ønske i praksis.

# 11 AMDAHLS LOV

Vi ser av Kvikksort-eksempelet først hadde en sekvensiell del (dele arrayen i to) , og så kunne parallelliseringen begynne. Dette er generelt for mange algoritmer, at de har ofte 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 da 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 på 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 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 gå 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 problemet vårt gå 1000 ganger så fort, så trenger vi ikke fortvile. At beregninger går fra 9 til ca. 90 ganger fortere er klart nyttige resultater.

# 12 OM PROSESSOREN, JAVA-KOMPILATOREN OM HUKOMMELSES-MODELLEN

I begynnelsen av kom ga vi inntrykk av at instruksjonene blir utført i den rekkefølgen de er 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 bare foretas hvis *det ikke får innvirkning på sluttresultatet.* Prosessorkjernene har også nå fått egne instruksjoner som kan kjøre flere instruksjoner av type 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 slik vi har tenkt når vi laget programmet - programlogikken. Trenger vi resultatene av én beregning i høyresiden i en annen beregning, eller i for eksempel i en utskrift til skjerm eller fil eller i en test, blir *ikke* slik ombytting eller parallellkjøringer av instruksjoner foretatt.
Det Java og prosessoren garanterer deg er at du får utført et program 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 slik ombytting av instruksjoner av prosessoren og kompilatoren, behøver vi ikke dvele mer med det unntatt å advare mot en feil man som programmerer kan gjøre. Se på flg. to linjer i et program:

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

***Program 12.1**. Tilordning av verdier til to variabler. Hvis vi senere 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 i en utskrift, kan det godt hende at y får sin verdi lenge før x får sin verdi.*

Vi kan konkludere at den gamle enkle modellen om at vi har en prosessor og en kjerne og at den 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.

## 12.1 HVA SKJER NÅR VI SYNKRONISERER FLERE TRÅDER PÅ SAMME OBJEKT

Når to eller flere tråder synkroniserer på **samme** objekt (sier f.eks await på samme CyclicBarrier , eller kaller en metode som er beskyttet ReentrantLock) vil alle disse oppleve følgende:

- Alle kode som er ovenfor synkroniserings-setningen i trådene og som hittil ikke er utført (f.eks, er utsatt av optimaliseringsgrunner), blir utført.
- Alle data som er skrevet på av de synkroniserende trådene blir skrevet ned i hovedhukommelsen (eller i alle fall en *felles* nivå3 cache, noe som er vanlig på en CPU med mange kjerner).

Dette betyr at alle tråder som har synkronisert på felles variabel ser samme verdier på falles variable slik de er hittil skrevet på av de deltagende trådene. Tråder som derimot ikke har synkronisert på dette felles synkroniseringsobjektet, er ikke garantert å se de siste verdiene på slike felles data.

Det er også slik at alle felles data (som har blitt endret av en av trådene) blir skrevet ned i hovedhukommelsen når trådene er ferdige, dvs. er ferdige med siste setning i run-metoden.

# 13 MER OM OPTIMALISERING

Noen programmer er man avhengig av at går raskt, og da oppdager man en egenskap med Java, at første gang utfører en viss type kode kan koden ta en viss tid, mens neste gang kan samme koden gå mye raskere. Kjøre man samme koden veldig mange ganger kan den typisk gå fra 20 til flere 1000 ganger raskere enn første gang. Hvorfor kan dette skje?

Det er riktig å påpeke at de fleste programmeringsspråk kan optimaliseres på denne måten slik som C, Fortan og C#. To ting bør man merke seg:

- Denne optimaliseringen kan gjøres totalt av kompilatoren når vi kompilerer vårt program og da kan man f.eks i C spesifisere om man vil ha o1, o2 eller o3 optimalisering, og da blir hele programmet optimalisert og blir da en del større enn den holdningen Java har, at bare den koden som virkelig blir brukt kompileres til maskinkode, og det er antall ganger den koden-biten blir eksekvert som avgjør hvor sterkt den etter hvert optimaliseres. I Java er det også slik at hvis det brukes en klasse eller metode fra en biblioteks-klasse (som Arrays.sort i Fig 17.3 nedenfor) må bytekoden fra biblioteket først lastes ned før den kan kompileres og så utføres, og blir da langsommere første gang den utføres enn kode som ligger i selve programmet.
- I hvor stor grad det er mulig å optimalisere kode avgjøres i stor grad om språket som skal optimaliseres er statisk typet eller ikke (dvs. at hver variabel får fastlagt sin type som int eller String, deklarert i selve koden eller ikke). De fleste vanlige programmeringsspråk er statisk typet. Men i Pyton, som ikke er statisk typet, kan en variabel 'x' for eksempel av og til inneholde et flyttall og av og til en tekst i samme programutførelse. Det er først når programmet kjører og f.eks. setningen: y = x+ 1 skal utføres er at typen til x må bestemmes, og det skjer da under selve kjøringen og forsinker utførelsen og begrenser selvsagt da også hvor mye koden kan optimaliseres.
- Uansett hvordan og hvor mye et program optimaliseres, så vil det utad, dvs. det som skrives ut til fil, på skjerm e.l. gi det samme resultat som et uoptimalisert program. Dvs. at programmet ble utført slik vi har lært i begynnerundervisningen, ovenfra og nedover, linje for linje til vi er ferdige.

Vi vet fra innføringskapitlene at først oversetter *javac* (java-kompilatoren) koden – f.eks klassen *MittProgram* din til noe som heter byte-kode (det ligger på filen *MittProgram.class).*Dette er kode for en tenkt maskin med enkle byteinstruksjoner. Så kaller du opp selve kjøresystemet *java* (som også kalles JVM – Java Virual Machine) for å kjøre programmet. Det første den gjør er å oversette all byte-kode den eksekverer til maskin-instruksjoner på den maskinen du bruker. Den oversetter ikke hele programmet ditt, bare de delene (metodene og klassene) du virkelig utfører. Første gang du kjører får du altså tider som både er oversetting til maskinkode + selve kjøretida for din kode.

Hvis den så kjøre en viss del av koden din flere ganger (f.eks 10-5000 ganger), vil den begynne å optimalisere på den maskinkoden den har laget. Enda mer kjøring kan gi ytterligere optimalisering. Optimalisering kan grovt sett beskrives som at den lager enda lurere og raskere maskin-kode av de delene av programmet du kjører ofte. Slik optimalisering kan gå i 2-3 omganger og bli stadig raskere. Det du derimot skal være sikret, uansett hvor mye maskinkoden forbedres, er at vi har sekvensiell semantikk. Dvs *du kan store på at du kan tenke og feilsøke programmet ditt ut fra selve Java-koden og du kan tenke på den som om koden blir utført ovenfra og nedover, linje for linje, løkke for løkke som om det aldri ble utført noen oversettelse til maskinkode eller senere optimalisering av denne.*

La oss se på et eksempel:

```
class C { int i;
   C(int i){ this.i =i;}
   int les () { return i+10;}
} // end C

s =”new Class C + metodekall “;

t = System.nanoTime();
for (int i = 0; i<n; i++) {
        k = new C(k).les();
}
d = (double) ((System.nanoTime() -t)/(n*1000.0));
println(s+d + «us.snitt « + n+» ganger»);
```

Vi ser at dette er en kode som først gjør en *new C(k)* og så kaller les-metoden i objektet vi har laget av klassen C og skriver ut tiden for det. Så gjør den det samme, bare n ganger. Første gang tok det 2697 µs (milliondels sekund), mens det med n=2 tok i snitt 0.45µs. En forbedring på ca 5000 ganger raskere! Selvsagt kan vi gjøre samme for ulike Java-konstruksjoner (se tabellen nedenfor). Grunnen til at det går mye raskere er ikke bare at vi oversetter til maskinkode første gang, men også at det i JVM bygges opp datastrukturer for dine klasser og metoder slik at neste gang har JVM en mye lettere og raskere jobb med å kjøre ditt program.

| n, ant. ganger | 1 | 2 | 3 | 100 | 10000 | 100000 | X bedre, (speedup) |
|---|---|---|---|---|---|---|---|
| for-løkke | 0,3 | 0,15 | 0,03 | 0,018 | 0,009 | 0,007 | 42 |
| metode-kall | 2697 | 0,45 | 0,06 | 0,054 | 0,026 | 0,026 | 103730 |
| new int[100] | 1,2 | 0,6 | 0,24 | 0,195 | 0,151 | 0,136 | 33 |
| array copy med for-loop, n=100 | 1,8 | 1,5 | 2,64 | 2,500 | 1,177 | 0,188 | 9 |
| System.arraycopy, n=100 | 5,7 | 0,3 | 0,15 | 0,126 | 0,072 | 0,064 | 89 |
| new Thread med start & join() | 3015 | 336 | 66,6 | 61,68 | 61,87 | 61,86 | 48 |
| new C(int) og metodekall | 2697 | 0,45 | 0,15 | 0,21 | 0,035 | 0,035 | 77 057 |
| Int [] array read | 0,3 | 0,3 | 0,06 | 0,036 | 0,012 | 0,012 | 25 |
| Innstikk-sortering (n=100) | 46,6 | 42,8 | 42,42 | 21,27 | 19,60 | 1,45 | 32 |

**Tabell 12.1** Kjøretider fra sekvensielt testprogram i **2017** i µs med Java 1.8.0 for ulike Java-konstruksjoner

og et enkelt sorteringsprogram som funksjon av antall utførte ganger + speedup for n = 100 000, Intel i7-7600 @3,4 Ghz.

| n | 1 | 2 | 3 | 100 | 10000 | 1000000 | x bedre (SU) |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| for-loop | 0,7 | 0,2 | 0,023 | 0,023 | 0,002 | 0 | **350,0** |
| metodekall | 9,7 | 0,9 | 0,1 | 0,102 | 0,005 | 0 | **1940,0** |
| int[] new | 0,6 | 0,3 | 0,152 | 0,24 | 0,137 | 0,087 | **6,9** |
| copy array for-loop | 2 | 2,3 | 2,793 | 1,336 | 0,635 | 0,011 | **181,8** |
| arraycopy | 4 | 0,7 | 0,208 | 0,036 | 0,033 | 0,021 | **190,5** |
| new Thread,start&join | 2576,9 | 301,8 | 210,659 | 175,962 | 0 | | **14,6** |
| new Class C m.metodekall | 2301,6 | 8,4 | 0,272 | 0,011 | 0 | 0 | **209236,4** |
| int [] a skriv | 0,6 | 0,5 | 0,028 | 0,006 | 0,01 | 0,007 | **60,0** |
| int [] les | 0,3 | 0,2 | 0,023 | 0,005 | 0,013 | 0,007 | **23,1** |
| double to long | 4,2 | 1,2 | 0,164 | 0,007 | 0,044 | 0 | **95,5** |
| insertSort int[] | 118,3 | 154 | 47,332 | 3,201 | 1,65 | 1,771 | **71,7** |
| Arrays.sort | 471 | 70,9 | 45,023 | 4,68 | 1,221 | 1,187 | **385,7** |

**Tabell 12.2**. En Litt forenklet versjon av fig .2.1. Kjøretider fra **2022** i µs + speedup fra n=2 til n= 1000 000, Java 14.0 for ulike Java-konstruksjoner og to sorteringsprogram som funksjon av antall utførte ganger av et sekvensielt testprogram på en AMD Ryzen 5 3500U, 2.0 3.4GHz. Kommentar: New Thread eksemplet er bare kjørt for n= 1,2,3 og 100, pga tidsforbruket.

Tallene ovenfor varierer noe for hver gang de kjøres, men viser klare trekk som at særlig metodekall og det å lage objekter effektiviseres ekstremt, men at arrayer ikke effektiviseres i samme grad. Oppmuntrende er det at brukerkode som en innstikks-sorterings algoritme som vi har skrevet (og som hver gang sorterer en ny usortert double array av lengde 100) kan effektiviseres med en faktor ca. 70 .

Delvis kommer det at din kode blir forbedret, men også at de data og variable har om ditt program blir kraftig forbedret. Uansett behøver en Java-programmerer bry seg om hvordan det skjer, men bare vite at ethvert Java-program med tiden vil gå stadig fortere, men alltid produsere samme svaret.

Dette at optimaliseringen gir oss samme svaret som om vi ikke hadde optimalisert koden, kalles ***sekveniell konsistens*** og er meget viktig for oss når vi skal feilrette programmet. Vi kan tenke på vårt program som om det blir utført linje for linje, ovenfra og nedover. Selve optimalisingen kan man **slå av** med å starte programmet vårt slik etter kompileringen til bytekode:

```
>java -Xint MittProgram   … de vanlige parametrene …
```

(brukes bare ved debugging hvis man tror at optimaliseringen har 'ødelagt' programmet, noe den nesten aldri har gjort.)

Det er viktig å vite at kallet på en metode eller det å lage et objekt av en klasse er knyttet til ***kallstedet***. Det betyr at hvis du f.eks kaller samme metode fra et annet sted i din kode, vil den også bli optimalisert der på ny. Det som grovt sett skjer er at det ikke blir foretatt et hopp til en optimalisert metode, men at denne optimaliserte koden blir lagt inn direkte der kallet på metoden er. Vi kan si at kallet som blir optimaliser ved å bli fjernet på kallstedet og erstattet med metodens kode.

Likevel kan av dette lære oss en del om hvordan vi bør skrive våre programmer:

1. Siden metodekall og det å bruke klasser og lage objekter av disse effektiviseres sterkt, bør vi ikke være redde for å bruke disse i rikt monn i våre programmer. Mange algoritmer kan være ganske kompliserte med mange steg og løkker. Hvis hvert slik steg kan utformes som

en egen, mindre metode, og at det hele så bygges sammen med en overordnet metode som i rekkefølge kaller disse mindre metodene, får vi et program som både er lettere å forstå og debugge, og mulig også raskere (optimalisatoren liker godt små metoder).

2. Når vi skal lage parallelle programmer med tråder, ser vi at det bare er det første trådobjektet som tar lang tid å lage og starte/avslutte (koster ca. 2,5 millisek). De neste, nr 2,3,.. tar hver bare ca 1/10-del av denne tida. Antall tråder vi deklarerer i et parallelt program behøver derfor ikke ha mye å si for kjøretiden. Ofte vil f.eks. det å bruke flere tråder som vi har kjerner være et vellykket valg for løsning av et parallelt problem.
3. Ofte er det også slik at noen problemer, som sortering av tall, går så raskt at det ca. 4 millisek. å sortere si 25 000 tall med en sekvensiell metode. Det gjør at en sekvensiell sorteringsalgoritme vi være ferdig før den parallelle versjonen av algoritmen har greid å ha fått starte sine tråder og begynt å løse problemet, De fleste parallelle programmer vil derfor inneholde en første metode av typen:

```
void løsProblemet ( ..param..) {
    if(størrelsen_av_problemet   <   minimum_størrlse)
        løsProblemetSekvensielt(..param..);
    else løsProblemetParallelt)..param…);
}// end løsProblement
```

4. Med få unntak, ikke prøv å optimaliser koden selv, optimalisatoren er langt flinkere enn deg.
5. Unntakene er at arrayer med to dimensjoner alltid bør leses, beregnes og skrives ***radvis*** og sekvensielt, og at de data vi leser/skriver i en løkke helst bør passe inn i størrelsen på cache, nivå1 eller cache-nivå 2.
6. Du skal selvsagt, når du lager en parallellisert løsning på et problem, bruke den raskeste, sekvensielle metoden som et utgangspunkt for den parallelle løsningen.
7. Husk at når vi lager en parallell løsning med *k* tråder har vi laget *k* sekvensielle programmer som deler felles kode, men som har ulike deler av problemets data. Hvis de data som deles har blitt delt opp i *k* deler, er det å forvente at hver del passer langt bedre inn i cache nivå 1 og 2. Hvis det er tilfellet, vil et slikt parallelt program kunne gå raskere enn *k* ganger fortere enn det opprinnelige sekvensielle løsningen fordi mer av data ligger høyere opp i cache-hierarkiet.
8. Derimot er det et annet forhold som peker mot at programmet ikke går så fort, og det er forbindelsen mellom kjernene og hovedhukommelsen, kalt datakanalene. Når *k* kjerner samtidig vil skrive og lese via datakanalene i hovedhukommelsen, blir det ofte kø og ventetider med langsommere eksekvering som resultat.
9. For å oppsummere, det er et rimelig håp at et parallellisert program på en multikjerne PC med *k* kjerner, vil ha en speedup på ca. *k*.

## 13. Feilaktige antagelser i PRAM-modellen

Som tidligere nevnt har det blitt laget en teori for å analysere og lage parallelle programmer som kalles PRAM (Parallell Random Access Machine). Den gjør følgende forenklinger for lettere å analysere parallelle maskiner og programmer:

1. Tiden det tar å lese eller skrive i hukommelsen er konstant og den samme, og settes =1.
2. Vi kan lage programmer som kan ha så mange kjerner vi vil - f.eks. *n* kjerner hvis vi skal sortere *n* tall.
3. Alle parallelle programmer kan startes samtidig på samme klokke-sykel
4. De parallelle kjernene går synkront, dvs. hvis de utfører samme program, vil de i én klokke-sykel utføre eksakt samme instruksjon på samme tidspunkt.

Alle disse påstandene er ganske langt fra virkeligheten og gale, og kan gi feilaktige analyser og dysfunksjonelle programmer fordi:

1. Påstanden om konstant tid for lesing og skriving ignorerer prefetch og cache-systemet. Leser/skriver vi data som befinner seg i et register eller i cache nivå1 vet vi at det går 100-200 ganger fortere enn om vi får en cache-feil og vi må da gå helt ned i hovedhukommelsen for data til hver ny instruksjon. I praksis opplever vi tidsforskjeller på minst faktor 20-40 hvis en litt stor (si minst 100x100) todimensjonal array aksesseres kolonnevis istedenfor radvis.

2. Feilen med at vi kan ikke fritt skrive programmer med veldig mange tråder (si mer enn ca. 1000-100 000) og i tillegg tro at dette går greit fordi man i parallell har like mange kjerner. Dette vil det raskt fylle hele data-maskinen, da hvert objekt av klassen Thread tar en viss plass. Og siden ingen multikjerne maskin i praksis har mer enn 32-64 kjerner, vil det bli en evig køing av tråder på å få eksekvere på en virkelig kjerne med mye utskifting av systemdata i hovedhukommelsen og kø på data-kanalene. Antagelsen av vi har valgfritt antall kjerner tilgjengelig vil lett få oss til å skrive dysfunksjonelle programmer.
   Riktignok kan vi med å programmere GPU-en (grafikkprosessoren) kunne få flere tusen prosessorer i parallell som utfører samme instruksjon på ulike data. Men her er vi begrenset til noen tusen kjerner, og er det bare noen typer problemer som løses effektivt med grafikkprosessoren. Dette kurset et kurs i å programmere parallelle algoritmer på en multikjerne CPU. Tilleggskurs i GPU- og programmering av beregningsklynger samt snart også på kvantemaskiner finnes på Ifi. Generelt kan det sies at ikke alle typer av problemer lar seg løse mer effektivt på slike spesielle maskiner enn på en multikjerne PC.

3. og 4 : Antagelse om full synkronisering er gal. I praksis startes trådene så raskt man greier sekvensielt av et sekvensielt program. Hvis trådene f.eks. skal gjøre veldig lite (si øke en felles variabel 100 ganger), så kan vi lett oppleve at vi ikke får noen feil fordi den første tråden som starter er ferdig før neste tråd får begynt på å utføre sin kode. Vi kan da lett få det inntrykket at vi ikke trenger å synkronisere adgangen til denne felles variabelen med f.eks en *ReentrantLock* eller *synchronised.* Elektronikken i kjernene er heller ikke alltid synkronisert slik at vi fysisk sett kan få litt ulik hastighet på de ulike kjernene.

Riktignok har PRAM-modellen blitt modifisert i ulike retninger. og vel særlig antagensen om all aksess til data tar samme tid, men kurset IN3030 finner det uproduktivt og galt å basere sin programmering på en sterkt forenklet teori og modell for parallellprogrammering. Vi trenger ingen annen modell annet enn programmet for problemene vi skal løse. Å se bort fra en rekke mekanismer i elektronikken og i optimaliseringen i kjøresystemet java (JVM Java Virtuell Machine), finner vi uproduktivt og tidvis villedende for å finne den beste løsningen og algoritmen for et problem.

Kurset IN3030/4040 baserer seg på tidtaking som målestokk på en effektiv algoritme i tillegg til at vi nytter O(n) - analyse av både den sekvensielle og parallelle algoritmen for å fine forventet kjøretid som funksjon av antall dataelementer n.

# 14 MENGDENE I JAVA KAN VÆRE LANGSOMME

Når man ser på hastighet i Java-programmer, så bør man også se på at i alle mengdene i Java-biblioteket som var mengder av basale typer som int, double, long ... byttet ut med tilsvarende klasse- representasjonen av tilsvarenede basale typer som Integer for int, Double for double, …osv. Vi deklarerer en ArrayList for heltall slik, og her legger vi samtidig inn tallene 1 til n i lista:

```
ArrayList<Integer> liste1 = new ArrayList <Integer> ();
for (int i = 1 ; i <= n; i++) liste1.add(i);
```

Det som skjer bak kulissene er at hver int-verdi blir pakket inn i et Integer objekt før det blir lagret med en peker fra lista til dette objektet. (på engelsk: boxing) . Skal vi så lese verdien i Integer-objektet , må heltallsverdiene pakkes ut byte for byte (på engelsk: unboxing) fra et Integer-objekt på 16 byte + 8 byte til en peker til objektet i motsetning til en 'vanlig' int i en array på 4 byte . Det vil si at en Integer i en array tar omlag 6x så stor plass som en int-variabel, noe som gjør at de raskere fyller opp de ulike cachene, og det tar lenger tid for å gjøre les og skriv på heltallselementene man tross alt behandler.

For å vurdere hvor rask ArrayList er, kan man lage en alternativ implementasjon *IntList* med en heltallsarray med den funkjonalitetet som ArrayList har:

```
class IntList{
    // representerer k heltall, adressert: 0..k-1
    int [] data;
    int len =0;
    IntList(int len){data = new int [Math.max(1,len)];}
    IntList() {data = new int [16]];}

    void add(int elem) {
    if (len == data.length) {
        int [] b = new int [data.length*2];
        System.arraycopy(data,0, b,0,data.length);
        data =b;
          }
          data[len++] = elem;
      } // end add

    void clear(){len =0;}

    int get (int pos){
        // error: antar at svaret brukes til array-indeks
        if (pos > len-1 ) return -1; else return data [pos];
    } // end get

    int size() {return len;}
} // end class IntList
```

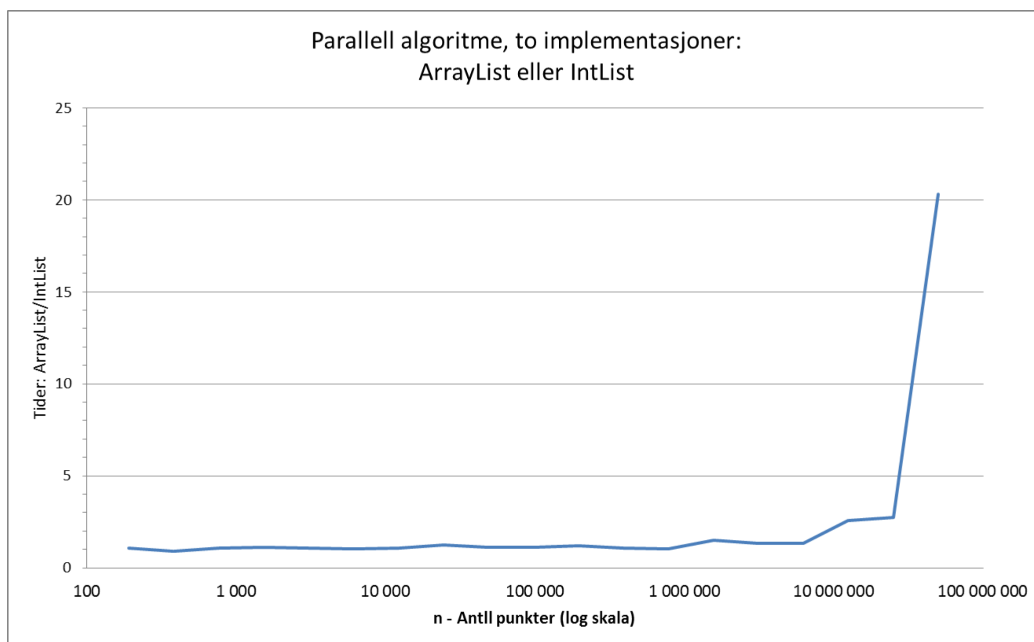
Kjører vi en enkel test av innlegging (add) og les(get) n ganger ser vi at ArrayList er klart langsommere enn IntList .

***Tabell 13.2** Forholdet i kjøretider for ArrayList/IntList for ulike lengder av listene, og viser hvor mange ganger langsommere ArrayList er enn IntList.*

| n | IntList(ms) | ArrayList(ms) | forhold |
|---:|---:|---:|---:|
| **500 000 000** | 6 194.84 | 211 277.47 | 34.11 |
| **50 000 000** | 146.46 | 2 054.59 | 14.03 |
| **5 000 000** | 15.06 | 41.22 | 3.05 |
| **500 000** | 1.13 | 3.44 | 3.43 |
| **50 000** | 0.10 | 0.36 | 1.57 |
| **5 000** | 0.02 | 0.04 | 2.50 |
| **500** | 0.00 | 0.00 | 2.00 |

Grunnen til at det ikke blir verre forsinkelse er at ArrayList, som mange av de andre mengdene, er parallellisert (for 'store' verdier av n). Hvis du bruker ArrayList i en parallell tråd, kan ny parallellisering være en dårlig ide, fordi istedenfor *k* parallelle tråder har du plutselig *k*k* parallelle tråder som slåss om de *k* fysiske kjernene. Dette gir vanligvis hastighetsreduksjon. Parallelle tråder bør ikke selv starte sine *k* tråder osv.

Kjører vi et parallelt program som bruker IntList eller ArrayList mye (testet på et som løste den konvekse innhyllinga til n punkter), får om lag like store forsinkelser ved å bruke ArrayList. Derfor bruker vi IntList.




***Figur 17.1** Vi ser at jevnt over er den parallelle algoritmen som bruker IntList ca 10-15% raskere når n er liten, men blir når n > 10 mill blir IntList mye raskere enn ArrayList.*

# 15 OPPSUMMERING – OM PARALLELLISERING AV ET PROBLEM

Det er mange måter å parallellisere et problem i en multikjerne-maskin, men de to som er forklart her, bruk av synkroniserte metoder, barriere synkronisering samt Reentrantlock, skulle være tilstrekkelig for de aller fleste problemer. Særlig barrieresynkronisering egner seg for større problemer.

1. For at det skal være vits 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 0.01 sekund, er det ingen vits i å parallellisere det.
2. Start med en godt testet og effektiv **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/når vi under beregningene må ha felles data, må de alltid beskyttes. All skriving og lesing på disse felles data skjer med synkroniserte metoder.
5. Hvis ***alle data alltid*** er felles, er det ingen vits i å parallellisere problemet hvis ikke trådene hver kan ha kopier av relevante felles data og at disse til sist greit kan samstilles etter beregningene.
6. Vi kan godt uten beskyttelse la ulike tråder i parallell skrive på **ulike elementer** i en array ( men ***ikke*** i ulike bit i samme byte)
7. 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.
8. Kanskje er vi nå enten ferdig, eller resultatene fra første beregninger kan igjen deles opp i flere tråder med en ny barriere., osv.
9. Tenk spesielt på hvordan main-tråden skal vente og slippe løs 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.
10. For noen klasser av problemer lønner det seg å sette i gang flere tråder enn man har *k* kjerner (inkludert hyperthreaded kjerner) i maskinen – f.eks 2k,3k eller 4k tråder. Bare test om dette evt. går fortere. Grunnen til en hastighetsøkning her er at de deler av data vi behandler i en tråd passer bedre i cache-systemet. Ulempene ved å få oftere byte av kjerner og mer synkronisering oppveies da av hastigheten av cache-systemet.

To typiske parallelliseringer med *k* kjerner og *k* tråder:

A. De sentrale data i problemet lar seg enkelt dele i *k* like deler hvor vi kan la den sekvensielle algoritmen løse hver sin 1/*k* del av problemet med hver sin tråd i parallell Hvis problemet er så enkelt at man skal finne det størst elementet i en array, vil vi til sist etter at hver tråd har køet på en CyclicBarrier, så kan en av trådene (f.eks. tråd nr. 0) sammenligne de *k* svarene og rapportere den største av disse på skjermen eller fil.

B. Hvis den sekvensielle løsningen er rekursiv, er det en enkel men ikke helt optimal løsning å erstatte de få øverste rekursive kallene med at vi lager en tråd og for hver av disse. Det er viktig at der er toppen av rekursjonstreet (de øverste 2 til 4 lagene hvor de rekursive kallene er erstattet av en parallell tråd). Problemet med denne løsningen er at den første oppdelingen i toppen med to rekursive kall blir sekvensiell. Det er først på neste lag at vi kan få til 2-parallell, så 4-parallell på neste nivå osv. Det er for noen rekursive algoritmer som Kvikksort mulig å endre disse sli at vi kan starte med full parallell med alle trådene.[ https://ojs.bibsys.no/index.php/NIK/article/view/247 ] . For andre rekursive algoritmer som Flettesortering mulig å heve parallelliteten til 2-parallell , så 4-parallell,…, på det øverste laget i rekursiviteten ved at de kortere sorterte delene flettes både fra starten for å finne de minste elementene og samtidig fra endene sorteres for finne de største elementene i parallell [ https://ojs.bibsys.no/index.php/NIK/article/view/491 ]

# OPPGAVER

1. Lag den sekvensielle versjonen av Kvikksort, modifiser utførOgTest() for sorteringseksempelet 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 kvikksorttiden.
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?
3. Skriv et program for 'En Selgers rundtur'; først sekvensielt (rekursiv) og så en blanding av parallelt og rekursivt. Dette er en oppgave med svært mange beregninger og svært lite data, og egner seg svært godt for parallellisering.
   **Problemet er slik:** En selger skal reise og besøke n byer – hver by skal besøkes bare en gang. Hun vet avstandene fra enhver by til alle de andre byene. Disse dataene har hun 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 ved å reise via en annen by.
   Skriv ut den korteste reiseplan av alle mulige hvor selgeren besøker hver by bare en gang og som kommer til slutt 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) { // gjenstår bare reisen tilbake til by:0
           if (lengde+avstand[by][0] < bestHittil){
                  <notér ny beste reisevei og lengde>
           }
      } else {
           for (int nesteBy = 1; nesteBy < n ; nesteBy++) {
             // prøv å besøke alle ikke-besøkte byer unntatt 0
             if (! besøkt [nesteBy]) {
                besøkt[nesteBy] = true;
                reiseRute[nivå] = nesteBy;
                RSR (nivå+1,nesteBy,lengde +avstand[by][nesteBy]);
                besøkt[nesteBy] = false;
    }}}} // end PSR

    SeqRSelger(int n) {
          <opprett og initier arrrayer med tildeldige 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 nok den korteste. Merk hvor raskt den løser et 10-bys problem, men at det tar alt for lang tid å løse et 15-byes problem. Kjøretiden går som* ***n!*** *(=1*2*..*(n-1)*n). Vi ser at kjøretiden da øker meget raskt, eksponensielt med n.*

4. Lag en versjon av parallell Kvikksort hvor du bare bruker én sykliske barriere. Hint: Ser du på kall-treet i fig. 1.7 ser du at det er ett kall på **pQuick** på nivå 1 (toppen) i treet, samlet 3 kall på nivå 2 og nivå1 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 raskere enn den som står i avsnitt 8.1?

## REFERANSER PÅ NYERE PROSESSORER


*1 -Amazone Graviton :* *https://www.nextplatform.com/2022/01/04/inside-amazons-graviton3-arm-server-processor/*

*https://www.anandtech.com/show/15578/cloud-clash-amazon-graviton2-arm-against-intel-and-amd*

*2: Intel :*
https://images.anandtech.com/doci/17596/VIP%20Acceleration%20Experience%20Primer%20Meeting%20Slides%208.png

*3: AMD EPYC:* HTTPS://EN.WIKIPEDIA.ORG/WIKI/TEMPLATE:AMD_EPYC_7003_SERIES

*4: Apple : https://www.apple.com/newsroom/2022/03/apple-unveils-m1-ultra-the-worlds-most-powerful-chip-for-a-personal-computer/*