IN3030 – Effektiv parallellprogrammering

Studiepoeng: 10 **Nivå:** Bachelor

Undervisning: Vår **Eksamen:** Vår **Undervisningsspråk:** Engelsk

Beskrivelse av emnet

Timeplan, pensum og eksamensdato

Vis tidligere semestre +

Kort om emnet

Emnet vil gi kunnskap in ulik bruk av parallellitet på en flerkjernet datamaskin og særlig gi innsikt i hvordan og når man i Java kan utvikle parallelle programmer som kan bli klar raskere eller enklere enn et sekvensielt program som løser det samme problemet.

Hva lærer du?

Etter å ha tatt emnet vil du:

- ha god kunnskap om basal trådprogrammering i Java og bruk av sentrale komponenter i java.util.concurrent
- vite hvilke nye vansker parallell programmering med tråder gir og hvordan disse kan løses
- beherske teknikker for å omforme en sekvensiell algoritme til en effektiv parallell algoritme
- kunne lage flere ulike parallelle løsninger på et problem og vurdere effektiviteten til disse
- kunne ta eksekveringstider på programmer og bestemme eventuelle hastighetsforbedringer
- kjenne grunnleggende begreper for multikjernedatamaskiner

Opptak til emnet

Studenter ved UiO søker plass på undervisning og melder seg til eksamen i Studentweb.

Spesielle opptakskrav

I tillegg til generell studiekompetanse eller realkompetanse må du dekke spesielle opptakskrav:

- Matematikk R1 eller Matematikk (S1+S2)

De spesielle opptakskravene kan også dekkes med fag fra videregående opplæring før Kunnskapsløftet, eller på andre måter. Les mer om spesielle opptakskrav.

Obligatoriske forkunnskaper

IN1000 – Introduksjon til objektorientert programmering/INF1000 – Grunnkurs i objektorientert programmering (videreført) (eller IN1900 – Introduksjon i programmering for naturvitenskapelige anvendelser/INF1100 – Grunnkurs i programmering for naturvitenskapelige anvendelser (videreført)) og IN1010 – Objektorientert programmering/INF1010 – Objektorientert programmering (videreført)

Anbefalte forkunnskaper

IN2010 – Algoritmer og datastrukturer/INF2220

Overlappende emner

- 10 studiepoeng overlapp med INF2440 – Effektiv parallellprogrammering (videreført).
- 10 studiepoeng overlapp med IN4330 – Effektiv parallellprogrammering.

Undervisning

2 timer forelesning og 2 timer øvelser hver uke.

Obligatoriske øvelser må være godkjent for å kunne gå opp til eksamen. Obligatoriske oppgaver er gyldige i 1.5 år

Obligatorisk oppmøte på første forelesning.

Eksamen

4 timer skriftlig digital eksamen.

Obligatoriske øvelser må være godkjent for å kunne gå opp til eksamen. Obligatoriske oppgaver er gyldige i 1 1/2 år

Som eksamensforsøk i dette emnet teller også forsøk i følgende tilsvarende emner: IN4330 – Effektiv parallellprogrammering, INF2440 - Effektiv parallellprogrammering (videreført)

Hjelpemidler til eksamen

Alle skriftlige hjelpemidler er tillatt.

Eksamensspråk

Eksamensoppgaven blir gitt på norsk. Hvis emnet undervises på engelsk vil oppgaven kun gis på engelsk. Du kan svare på norsk, svensk, dansk eller engelsk.

Karakterskala

Emnet bruker karakterskala fra A til F, der A er beste karakter og F er stryk. Les mer om karakterskalaen.

Adgang til ny eller utsatt eksamen

Studenter som dokumenterer gyldig fravær fra ordinær eksamen, kan ta utsatt eksamen i starten av neste semester.

Det tilbys ikke ny eksamen til studenter som har trukket seg under ordinær eksamen, eller som ikke har bestått.

Mer om eksamen ved UiO

- Kildebruk og referanser
- Tilrettelegging på eksamen
- Trekk fra eksamen
- Syk på eksamen / utsatt eksamen
- Begrunnelse og klage
- Ta eksamen på nytt
- Fusk/forsøk på fusk

Andre veiledninger og ressurser finner du på fellessiden om eksamen ved UiO.

Sist hentet fra Felles Studentsystem (FS) 22. mai 2024 11:57:02

Kontakt

Institutt for informatikk

Lectures in IN3030 Spring 2024

L01v24: Intro
L02v24: Threads and Concurrency problem
L03v24: More on threads and Caching intro
L04v24: Caching and Caching Oblig
L05v24: Michael: concerning Quantum Computing
L06v24: Primes and *Eratosthenes Sieve*; paralli
L07v24: Amdahl and Gustavsons laws; timing problems; Java Measurement Harness
L08v24: Moore's «Law» and the sloooow speed of light
L09v24: Sync sketch with Oliver Ruste Jahren; Cooks and Waiters; Hoare Monitors
L10v24: No lecture
L11v24: Convex Hull
L12v24: Parallelization of Convex Hull and recursive algorithms in general
L13v24: Synchronization examples;
L14v24: No Lecture
L15v24: No Lecture
L16v24: No Lecture (Kristi Himmelfartsdag)
L17v24: Repetition
L18v24: Exam prep and hints

Effektiv Parallellprogrammering, kompendium i IN3030/4330

av

ARNE MAUS

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

15. mars 2024

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

Innledning

Dette er en tredje oppdatering av et kompendium i parallellprogrammering i Java på en vanlig flerkjerne maskin med felles lager. Dette stoffet foreleses på Institutt for informatikk i kursene IN3030/4030 på bachelor- og master-nivå. Det poengteres at dette kompendiet er et 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. Å parallellisere kjente algoritmer på en datamaskin med flere prosessorkjerner synes de siste 10-20 årene å være den mest generelle løsning på svært mange typer av problemer innen ingeniør-beregninger, administrasjon og KI. Når vi ser på antall ulike typer av datamaskiner solgt og installert i serverparker og i generelt salg som mobiltelefoner/ PC-er og arbeidsstasjoner, finner vi at multikjerne maskiner med felles lager nå er klart dominerende i antall; mens maskiner som skal brukes f.eks. til KI-anvendelser (både opplæring av KI -modellene og senere kjøring av disse KI-modellene) er opp til en faktor 1000 ganger raskere ved å bruke spesiell KI-maskinvare på visse problemer. Det er imidlertid viktig, uansett hvordan markedet for ulike typer av maskiner går, bør man lære grunnleggende parallellisering i dette kurset før man gå løs på mer spesialiserte brikker som KI-brikker.

Studentene har som en 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 4-16 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 her). 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 , Kotlin 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 den 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 flere andre aktuelle typer av maskiner, som også nyttes til å løse parallelle problemer:

Grunnen til at vi ikke omhandler bruk av grafikk-kort GPU med mange tusen enkle prosessorer er at programmering av de alle 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). Andre spesielle brikker er Field programmerbar array, GPU og ASIC -kretser (Se referansen nest sist i dette kompendiet til en lang drøfting av ulike brikker og trender innen elektronikkindustrien med 225 referanser!).

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, i tillegg 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 når vi bruker på én multikjerne PC og hvordan disse kan relativt enkelt kan løses med parallellisering av et sekvensielt program. 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 parallellprogrammering, vil vi i dette kurset flere ganger kommentere hvor feilaktige anslag disse teoretiske modellene er. De kjøretider PRAM anbefaler kan bli svært misvisende. PRAM modellen er mer utfyllende behandlet i kap. 11.

N.B. En liten irritasjon: Java har gått over fra ASCII til UTF-8 koding av Java-programmer og aksepterer i kompilatoren **ikke lenger bokstavene**: *æ, ø , å*. Gir stygge feilmeldinger. Tar du klipp og lim av programmer fra denne teksten må du f.eks. bytte ut: *æ* med ae, *ø* med oe , *å* med aa. Også i kommentarer.

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 en CPU-basert maskin med sine maskininstruksjoner og oppbygging med ulike typer elektronikk.

I det følgende vil vi gå gjennom de (forbausende mange) mekanismer, både i elektronikken og i Java som vil redusere eksekveringstidene i meget stor grad, og som vi kan benytte oss av når vi skriver programmer. Det presiseres at det er ikke hele datamaskinen eller hele Java som beskrives – bare de vesentligste mekanismene som vi som programmerere kan bruke til å få 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 som er innført i språket. I den grad slike nye begreper i også vil kunne endre hastigheten av våre programmer negativt, er et vesentlig eksempel beskrevet i kap. 12.

1.1 DATAMASKINENE

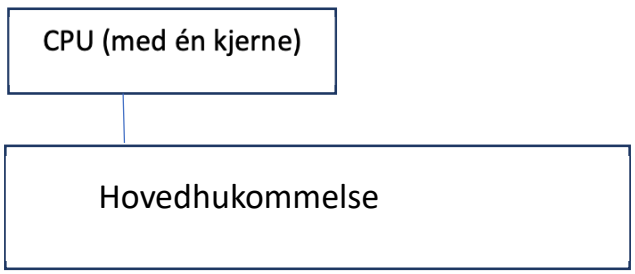
Hukommelses-systemet

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 i hovedhukommelsen. 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 en 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 cache-hukommelser som er laget av hurtigere, men dyrere elektronikk 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. ½ milliarddels sekund = ½ 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 vesentlig 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 resultatene er testet på en nyere maskin også 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 heltallet **`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
```

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

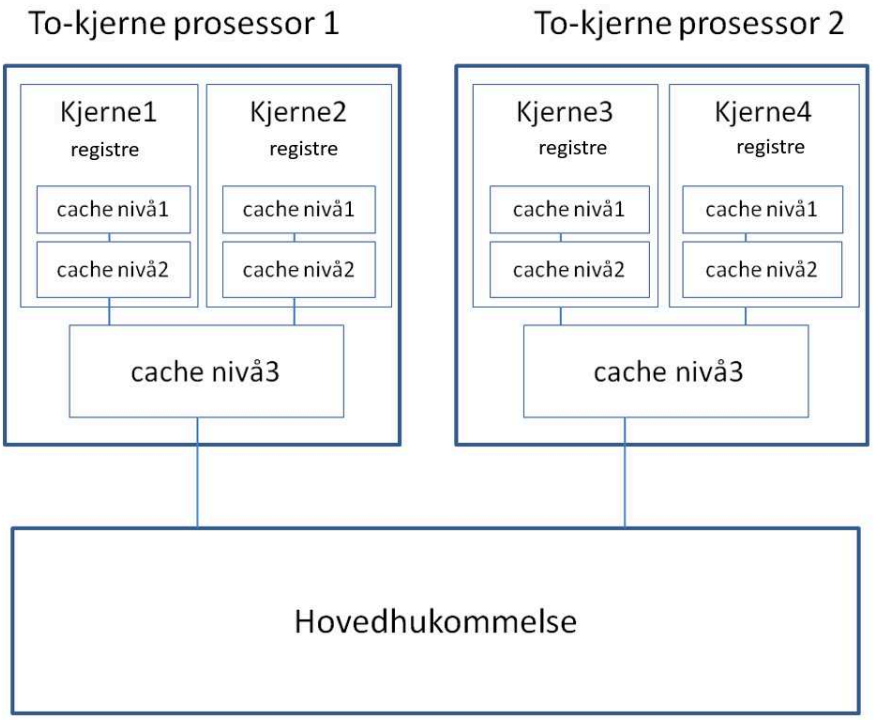
Det er Java-kompilatoren **javac** og senere kjøresystemet **java** som bestemmer hvor de ulike variablene i et program plasseres. Det er slik at alle variablene har sin plass i utgangspunktet i hovedhukommelsen. Men i koden Program 1.1 ser vi at metoden sum inneholder to variable, **s** og **i ,** som 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 og ikke i hovedlageret da heller ikke i cachene.

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.

```
int tall =…
void oekTall(int ant){
      for (int i = 0; i < ant; i++) {
            tall++;
       }
  }// end oektall
```

Program 1.2 *Metode oekTall som øker variabelen tall med 1 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 langt 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 cycler
Cache nivå 2	størrelse = 512Kb	tidsforsinkelse = 7 cycler
Cache nivå 3	størrelse = 4096Kb	tidsforsinkelse = 16 cycler
Hovedhukommelse	størrelse = 8 – 16 GB	tidsforsinkelse = 190 cycler

Ekstra L3 cache. En nyhet i 2022 var 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 av 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 før eller siden trenger 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[m],. .. så vil dette gå vesentlig langsommere enn sekvensiell aksess: a[i], a[i+1], a[i+2],... Spesielt oppstår denne effekten når vi leser bortover ***radene*** i en 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. 100 nano-sekunder mot lesning av L1 cachen på 2 nanosekunder for radvis lesing av hvert element. En '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'.

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, dvs. hvor den var i programmet og hva den foreløpig har regnet ut, kan denne nye tråden startes. Når vi har langt flere tråder enn vi har kjerner i en prosessor, og det har vi alltid når vi teller med de langt 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-løkken*- 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 4.2GHz. *Kort sagt*: 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 utfallet av testen – både om testen kommer til å gi sann eller falsk. Når testen er ferdig beregnet, vil

kjernen beholde beregningene som fulgte etter den beregningen med det riktige sanne svaret på testen og fortsette der. Beregningene som fulgte etter at testen fikk det gale svaret vil bli strøket.

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 – et roterende platelager (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 mer plass blir gitt programmer med høyere prioritet (disse ZIP-komprimerte områdene kan senere de-komprimeres når det blir plass til dem og hvis et 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 hovedhukommelsen. 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 – en cache-linje, 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.

1.3. 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 *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.

2. JAVA.

Java er et språk som er under sterk utvikling. Oracle, som nå eier Java, lager hvert år nye versjoner. Det som i hovedsak kommer til er nye begreper og konstruksjoner, men det hender også at 'gamle' (*depricated)* metoder og begreper fjernes. Ett eksempel på et nytt begrep er records, som er objekter bare med data og ikke metoder. Prinsippet er at noen versjoner (hver tredje til femte) som lages er stabile (Java 5, Java 8, Java 11, Java 17 og Java 21, …). Disse versjonen vil bli vedlikeholdt (feilrettet og tilpasset stadig nye versjoner av operativsystemer) i mange år fremover mens versjonene mellom disse er 'forsøksversjoner' hvor nye konstruksjoner kan komme og gå og bli endret eller fjernet til neste versjon. Powerpointfoilene i kurset er hovedsak laget med Java 5 og Java 8, mens noen av hastighetsbetraktningene i dette kompendiet er laget med Java14 og Java 21(komplett liste over java - versjoner på: https://www.java.com/releases/ . For å få bedre forklaringer på det nye som kommer i Java anbefales å abonnere på 'Java-magazine' (gratis) : https://blogs.oracle.com/javamagazine .

2.1 PARALLELLE PROGRAMMER I JAVA

Vi kan altså få 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 det for at alle tradene ser samme verdier i alle felles datastrukturer.

Parallell programmering kan være vanskelig, og det er derfor utviklet flere synkroniseringsmåter og biblioteker for mer strukturert parallell programmering i Java. Vi skal senere i dette kapittelet gjennomgå bruk av tre nyttige synkroniseringsmetodikker; Bruk av barriere-synkronisering (for synkronisering mellom trådene våre), synchronized methods (synkronisering mellom metodene i *ett* objekt) og ReentrantLock (for synkronisering av trådenes adgang til *én* bestemt metode). Vi starter med en gjennomgang om både hvordan vi kan 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 basert på bruk av de tre nevnte synkroniseringsmetodikkene nevnt ovenfor. Bryter man bare én av disse, vil det kunne gå veldig galt.

2.2 JAVA OPTIMALISERING, DEL 1

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

1. Data som vi til enhver tid bruker bør være minst mulige slik at de 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 helst 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 enn et program med som mange små metoder som kaller hverandre

4. 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. Denne utfører så disse byte-instruksjonene i samme rekkefølge sekvensielt; noe tilsvarende som Python utføres. I den første varianten av Java, Java1 var det 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 3 skjer, er følgende:

   Første gang et objekt fra en klasse opprettes og 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 tenkt byte-maskin. Merk at det bare er de delene som utføres som blir oversatt til maskinkode – resten forblir i byte-kode. 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

   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. Disse 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. Utføres en slik optimalisert metode mange ganger – si mer enn 200 000 ganger som blir koden ytterligere optimalisert, og går da enda mye raskere

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 for ulike* ***sekvensielle programmer*** *i* ***2022*** *som funksjon av antall ganger eksekvert i µsekunder (million dels sekunder) for ulike Java-elementer og for to sorteringsprogrammer: Ett brukerskrevet med kode i testprogrammet (innstikk-sortering av flyt-tall) og et er fra Java-biblioteket (Kvikk-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 for metode-kall og det å lage et objekt av en klasse (samt konvertering av flyt-tall (double) til heltall). Disse 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

tidsbruk som kompilering til maskinkode og henting av klassen evt. fra Java-biblioteket unnagjort. Vi ser dette spesielt i eksempelet Arrays.sort som første gang tar 404 µs mot innstikksort med 93 µs som har koden liggende i testprogrammet. Derimot er optimalisering av maskinkoden der den senere foretas, inkludert i kjøretidene (som f.eks. Inntikksort andre og tredje gang).

3. Det er meget tilfredsstillende at en egen skrevet brukerkode som Innstikksort kan optimaliseres opp med en faktor 60-100.
4. Av det overstående kan det se ut som f.eks. Innstikksort 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*** – og det 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).**

2.3 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 prosessens dataområde. I tillegg til at de kan ha hver sin 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.

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 den skal selv inneholde en metode public void run() { .. } som dere må skrive selv. Denne metoden run() tilsvarer på mange måter main()-metoden i hovedprogrammet, og er den metoden som kalles av systemet når den nye 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 flere 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 `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. En tråd avsluttes når den har utført siste setning i sin main() eller run() – metode.
8. 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.
9. Ikke før alle trådene, også main-tråden i et program er avsluttet, er programmet ferdig.
10. 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), men kan se litt gamle eller nyere verdier på ulike data de har utsyn til (løsning på dette problemet kommer senere i dette kapittelet).
11. 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).
12. 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 de aller fleste av disse trådene lå og ventet. I sum tok de omlag 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 da lokalt ha en kopi av slike data og skrive/lese på disse. Senere samstilles data fra disse lokale kopiene.

2.4 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 parallelle programmer går fortere .
Det som gjør at Java 17 programmer nok går fortere enn Java 8 programmer er at søppeltømmingsalgoritmer i Java 17(dvs av 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 inkrementell, Den tar ikke alt søppel med en gang, men gradvis fjerner alle 'søppel'- objekter.

Et annet prosjekt (Valhalla) prøver å lage en mer felles inn-utpakking av enkle variable , slik en 'int' blir pakket inn som en Integer med et objekt rundt seg, eksempelvis i ArrayList <Integer>, noe som klart tar lenger tid og plass. Dette kompendiet peker på dette problemet i kap. 12, hvor en enkel int[] nå er 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 plasskrevende enn basale typer som 'int'.

3. 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 kan på samme tidspunkt se ulike verdier av samme variabel. 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 den 'enkleste': 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 å 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 siste 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 *ett* objekt. Hvis metodene er deklarert som static, vil denne synkroniseringen gjelde i alle objekter av denne klassen hvor denne beskyttelsen er brukt. Hvis metodene ikke er static, vil metode-beskyttelsen bare gjelde metodene i ett objekt av gangen. Synchronized (som er et reservert ord i Java) kan da beskytte en hel metode eller bare en blokk med setninger inne i en metode.

3.1 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 programkoden :

```
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 tråd.
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.

3.2 OM BRUK AV REENTRANTLOCK

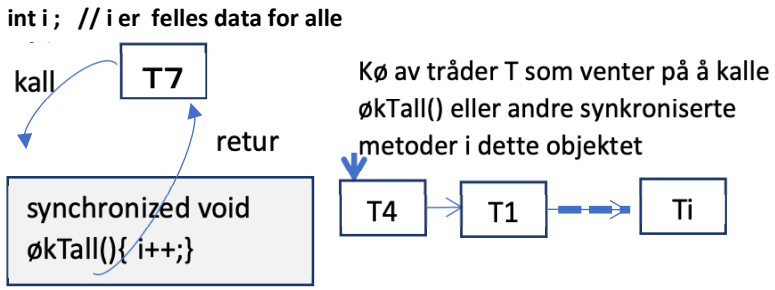
Den enkleste og raskeste låsen er ReentrantLock.

```
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.*

3.3 OM SYNKRONISERTE METODER

Synkroniserte metoder bruker låsen til det objektet de tilhører (alle objekter har én slik lås hver) 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 3.1). 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 et objekt 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 da 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 én 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 3.2. 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 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.

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

/** Start >java Parallell <ant tråder> <ant ganger i løkke> */
class Parallell{
      long  tall=0;      // Sum som 'antTråder' tråder teller opp
      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:»+ (antTraader*antGanger -tall)+» = «+
          Format.align( (antTraader*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 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.

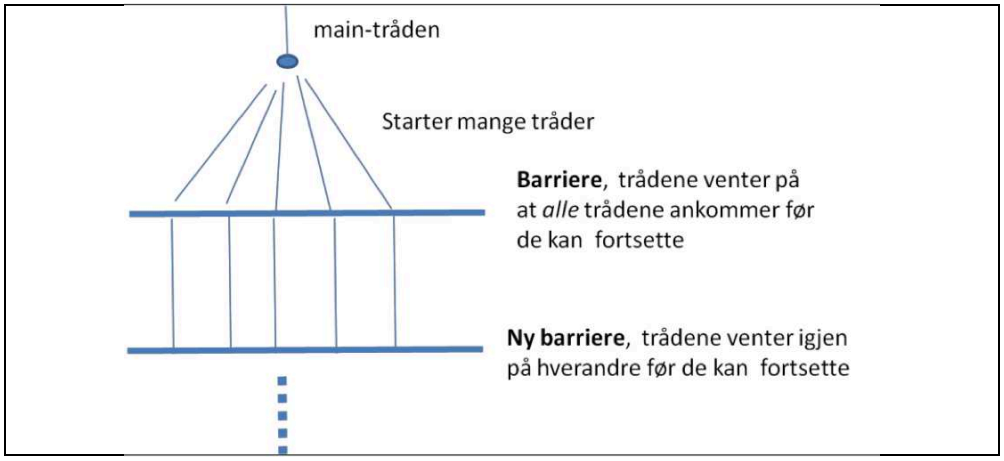
Tre kjøringer	**1**	**2**	**3**
Svar **uten** synchronized **3)**	7 112 531	5 911 630	6 169 492
Svar **med** synchronized **1)**	10 000 000	10 000 000	10 000 000

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.

4 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 4.1 *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 4.1. *Program som skisserer riktig bruk av CyclicBarrier i Java.*

Vi vil i neste programeksempel 4.2 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 5.2. *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 vil lønner se; 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. 16* 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 millisekunder å skape og starte den første tråden; raskere for de neste trådene. Vi kan godt bruke 10-500 tråder for å løse et problem, men ikke mange 10-tusner.
   Husk også at hvis lengden av problemet ikke er delbart med tallet antKjerner , så får vi en liten rest som den siste tråden, den med høyest indeks, også må ta med i sitt område.
3. Start en tråd for hver av disse delene av problemet.
   Dette gjør vi hvis ikke oppdelingen og start 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 til sist 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);}

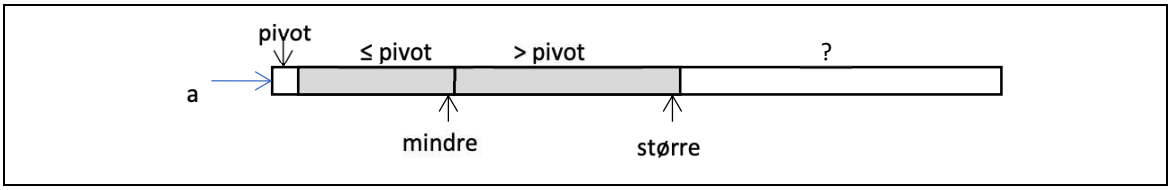
   Da vil Javas synkronisering sørge for at utskriften fra de ulike trådene *ikke* blander seg.

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

7 KVIKKSORT, EKSEMPEL PÅ 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 andre 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 7.1 *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 Kvikksort */
class SeqKvikk {
      int [] a;

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

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

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

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

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

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

Program 7.1 *Skjelettet for den sekvensielle programmet for Kvikksort som starter i main, som først lager et objekt av klassen SeqKvikk med kall på konstruktøren og deretter kaller utførOgTest.*

Koden for konstruktøren og utførOgTest:

```
     SeqKvikk(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
           sKvikk(a);
           t = System.nanoTime()-t;
           System.out.println("Sekvensiell Sort 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 7.2 *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 sKvikk(), 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 KvikkSort av a[lav..høy] */
   void sKvikk (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)  sKvikk (a, lav,mindre-2); // sorter <= pivot
      if ( høy-mindre > 0)  sKvikk (a , mindre, høy); // sorter > pivot
  }// end sKvikk
```

PROGRAM 7.3 *Koden for selve sorteringen: metodene sKvikk og bytt.*

Bytt er rett fram kode. Koden til sKvikk 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.** Så øker vi **mindre** med 1.

Merk at vi plasserer **pivot** mellom de to delene når vi er ferdige. **Pivot** 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].** Det er to grunner til dette. 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 kode-delen 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.

7. EN BLANDET PARALLELL OG REKURSIV KVIKKSORT

Grunnidéen i en parallell versjon av kvikksort er at vi bytter ut de rekursive kallene med at vi istedenfor 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. 6.1). 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 ParaKvikk {
      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 pKvikk(int [] a) {  pKvikk(null,a, 0,a.length-1); }

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

      ParaKvikk(int n) { … } // konstruktor

      void utførOgTest() {… }

      public static void main (String [] args) {
             new ParaKvikk(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() {
                     pKvikk(b,a,lav,høy);
               } // end run    }
} //end ParaKvikk
```

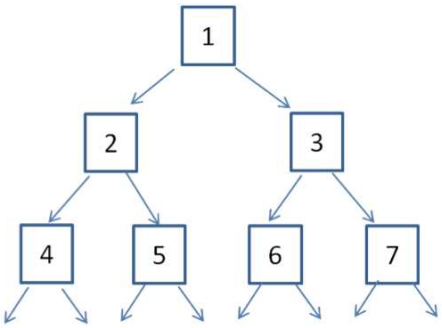
Program 7.1 *Skjelettkoden for parallell kvikksortering. Konstruktøren paraKvikk har samme kode som konstruktøren SeqKvikk i prog. 6.3. Også metoden bytt er identisk med den sekvensielle bytt.*

Vi ser at internt i klassen **ParaKvikk** 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 **pKvikk** for å få sortert den delen av **a[]** som denne tråden skal sortere.

```
void pKvikk (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) pKvikk (null,a, lav,mindre-2);//  <= piv
          if ( høy-mindre > 0) pKvikk (null,a, mindre, høy); //  > piv
      }

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

Program 7.2 *Sorteringsalgoritmen pKvikk. 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 et nytt 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 7.2. 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 7.2** 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 pKvikk 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.*

7.1 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
Parallell	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 7.2** 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
Parallell	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 7.3** 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 nivå 2, og for 100 000 tall i nivå 3 cachen og hovedhukommelsen som jo er mye langsommere.

Det beste vi synes å oppnå er at den parallelle versjonen går om lag 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 1.2 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.

8 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.

9. OM PROSESSOREN, JAVA-KOMPILATOREN OG HUKOMMELSES-MODELLEN

Det er skrevet et større notat kalt «Java memory model» som beskriver hvordan tråder skal kunne bruke hukommelsen under eksekvering. Dette er et meget komplekst dokument og ikke nødvendig å lære seg når man skriver parallelle programmer beskyttet av låser som de som beskrives i dette kompendium og i kurset. I hovedsak er dette et notat for de som skriver kompilatorer med optimaliseringer for Java, men det kan likevel være greit å se på ett tilfelle hvor en vanlig programmerer lett kan få seg en overraskelse.

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 en programmerer kan gjøre. Se på flg. to linjer i et program:

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

***Program 9.1**. 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 om 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 flere tråder når vi synkroniserer med låser. Dette er viktig for oss som programmerere. Uten denne enkle modellen vil det nesten ikke være mulig å skrive riktige programmer.

9.1 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 felles 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.

10. OPPSUMMERENDE KOMMENTARER 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 har vi sett at den i Java kan den typisk gå fra 20 til flere 1000 ganger raskere enn første gang.

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 utføre 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, som deklarasjoner 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. være det samme resultat som det et uoptimalisert program vil gi. Dvs. at vi kan alltid tenke oss at programmet blir 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 byte-kode (det ligger på filen *MittProgram.class).* Dette er kode for en tenkt maskin med enkle byteinstruksjoner. Så kaller du opp kjøresystemet *java* (som også kalles JVM – Java Virtual Machine) for å kjøre programmet. Det første den gjør er å oversette all byte- kode den utfører til maskin-instruksjoner på den maskinen du bruker. Java 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), vil den som vi tidligere har sett 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 ett eksempel fra beregning av en av radene i tabellene 10.1 og 10.2 nedenfor:

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

   String s = "new Class C + metodekall";

   long t = System.nanoTime();
      for (int i = 0; i< n; i++) {
           k= new C(k).les();
      }
   double d = (double) ((System.nanoTime() -t)/(n*1000.0));
```

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. Dette gjør vi ant ganger , ant =1,2,3,100,10000, 100000 ganger. Første gang tok det 2697 µs (milliondels sekund), mens det med ant=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 10.1 Kjøretider fra sekvensielt testprogram i **2017** i µs med Java 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 10.2. 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 .

Man kan også lure på hvilke resultater som kommer fra en noe raskere maskin og hva som kommer fra en bedre optimalisering i Java i disse to tabellene. Uten å ha kjørt store tester på dette vil jeg antyde at en bedre speedup kommer fra Java, men lavere tider for samme 'n' vel kommer fra en raskere maskin.

Delvis kommer det at din kode blir forbedret, men også at de data og variable du har i 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 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. 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. 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 at de data vi leser/skriver i en løkke helst bør passe inn i størrelsen på cache, nivå1 eller cache-nivå 2.
6. Råd nr. 5 synes umulig å følge i f.eks matrisemultiplikasjon av to n*n matriser B og C for å lage en ny matrise : A= BxC, hvor den matematiske definisjonen krever at element A[i,j] er lik summen av hvert element i raden B[i] ganger hvert element i *kolonnen* C[j]. Det synes som om vi *må* lese matrisen C kolonnevis. Her kan vi gjøre det trikset har sett på forelesningene, hvor vi først, før multiplikasjonen bytter om elementene C [i,j] med C[j,i] – og så multipliserer hver *rad* i den ombyttede matrisen C med hver *rad* i B. Da blir A slik det er matematisk definert. Det kalles å transponere matrisen C, og det er en meget rask algoritme – hvert element blir bare lest og skrevet én gang, mens i selve multiplikasjonen blir hvert element i B og C lest og summert n ganger. Den sekvensielle multiplikasjonen vi da gå om lag 40 ganger fortere fordi vi ikke får cache-miss og får brukt prefetch-mekanismen. Når man er ferdig med matrise-multiplikasjonen A = BxC, bør man transponere tilbake C slik at vi får den samme C som vi begynte med - fortsatt en operasjon som er meget rask og som igjen omtrent ikke berører kjøretiden. Merk at transponering av C er et råd som ikke er vesentlig hvis matrisene er meget små (f.eks 4x4 matriser) hvor hele matrisen får plass i level 1 eller level 2 cache.
7. Du skal selvsagt, når du lager en parallellisert løsning på et problem, bruke den raskeste, sekvensielle metoden som et utgangspunkt for den parallelle løsningen.
8. 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.
9. 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.
10. 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*.

11. 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 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 det er 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 på KI-maskiner 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.

1 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 som *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 det problemet 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*.

12 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 ... nå er byttet ut med tilsvarende klasse- representasjonen av 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 ArrayList tar omlag 6x så stor plass som en int-variabel, noe som gjør at Integer-objektene 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 funksjonalitet som ArrayList har:

```
class IntList{
// representerer k heltall, adressert: 0..k-1
int [] data;
    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;
    } // end clear

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

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

Program. 12.1 *Klassen IntList, bruk av en array for oppbevaring av heltallsverdier som alternativ til ArrayList*

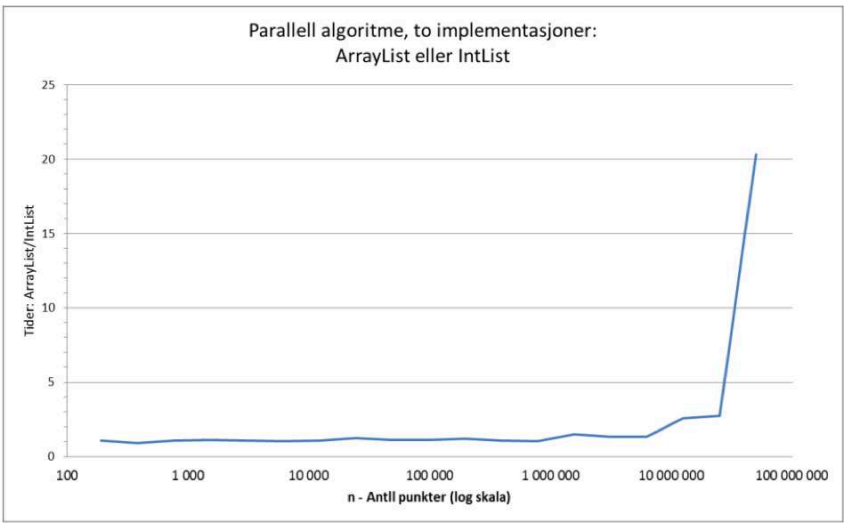
Kjører vi en enkel test av innlegging (add) og les(get) n ganger, ser vi at ArrayList er klart langsommere enn IntList .

Tabell 12.2 *Forholdet i kjøretider for ArrayList/IntList for ulike lengder av listene, og viser hvor mange ganger langsommere ArrayList er enn IntList i et sekvensielt program.*

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

ArrayList er per definisjon sekvensiell. Hvis du imidlertid bruker ArrayList inne i en parallell strøm i et parallelt program, kan en slik ny parallellisering være en dårlig idé. Fordi istedenfor *k* parallelle tråder har du plutselig nå *k*k* parallelle tråder som slåss om de *k* fysiske kjernene. Dette gir vanligvis en 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 12.*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 > 50 mill blir IntList mye raskere enn ArrayList.*

13 OM PARALLELLISERING MED STREAMS

I Java 8 og i senere versjoner, er det innført begreper som stammer fra funksjonell programmering – med lambda-uttrykk og strømmer (streams). Grunnen til å nevne det her er at her synes det å være en 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 velegnet til generell parallelisering.
(Kan du vanlige sekvensielle strømmer og lambda-uttrykk, kan du hoppe over avsnittene 13.1-13.5.)

13.1 Lambda-uttrykk og strømmer (streams)

Enkelt sagt er *lambda uttrykk* 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. Andre variable må vi deklareres som vanlig (trenger vi et heltall i, deklarerer vi det som: int i;

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-uttrykk* er mer elegant, lettere forståelig og gir brukere langt mindre å skrive særlig når *lambda-uttrykk* kombineres med *streams*.

13.2 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 én 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.

13.3 Hvordan lages lambda-uttrykk

Når vi lager et *lambda-uttrykk,* må vi ha et grensesnitt med en metode som har samme antall og typer på parameterne 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.

13.4 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.

13.5 Strømmer (streams) fra mengder

Strømmer i Java er kort fortalt å kunne stille spørsmål om mengder av objekter som minner om SQL spørsmål fra database-verden. Et noe alternativt syn på strømmer fra Eyvind W. Axelsen. Den ligger sist i kap. 13.7.

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 få vite hva gjennomsnittet er 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 inntekt til 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) stopper strømmen. Se dokumentasjonen til grensesnittene java.util.stream og java.util.collection i Java-dokumentasjonen.

Grunnen til 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 parametere 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.

Husk at vanlige arrayer ikke kan brukes direkte i en strøm , den må først konverteres til en strøm som her:

```
class GFG {
   public static void main(String[] args)
    {  // Converting int array to stream
        int arr[] = { 2, 1, 3, 4, 5 };
         IntStream stm = Arrays.stream(arr);
         stm.forEach(a -> System.out.print(a + " "));
   }
}
```

Og utskriften blir: **2 1 3 4 5.** Vi skal senere se hvordan vi kan sortere en slik strøm.

13.8 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 bare 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, Teknisk, Universitet i København og omarbeidet litt):

```
private static boolean isPrime(int i) {
   if (i %2 == 0 ) return i == 2;
   // vi starter med å teste '3'
    int k=3;  long k2=9;
    while (k2 <= i && (i % k) != 0){
            k+=2;
            k2=k*k;
     }
     return  k2 > i;
} // end isPrime
```

Denne metoden kan vi bruke til å først lage en sekvensiell strøm som teller hvor mange primtall vi finner under en bestemt grense 'n':

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

så innfører vi bare en parallelliserings-kommando i strømmen:

```
int antall =
        IntStream.range(2, range)
        .parallel()
        .filter(i -> isPrime(i))
        .count();
```

Som Peter Sestoft sier i sin forelesning:" Pure functional programming ... ... and thus parallelizable and thread-safe".

13.9 Hvor rask er strømmer sammenlignet med 'vanlig' trådprogrammering

For enkle problemer hvor vi skal enten telle eller summere verdier i en mengde ferdige objekter som tilfredsstiller visse kriterier (som en strømbasert løsning på primtallene ovenfor), er strømmer en god og effektiv måte å løse ett problem. Også parallelt. Det må imidlertid presiseres at ikke alle problemer egner seg for en slik løsning.

Det ble derfor laget et program som sammenlignet fire ulike måter (sekvensiell strøm, parallell strøm, sekvensielt program og trådbasert parallelt program) for å regne ut antall primtall < n med isPrime(i). Vi kaller da isPrime(i) *for alle heltall* : 10<= i <=n. Øvre grense n var: 10, 100, 1000,…,10 mill.

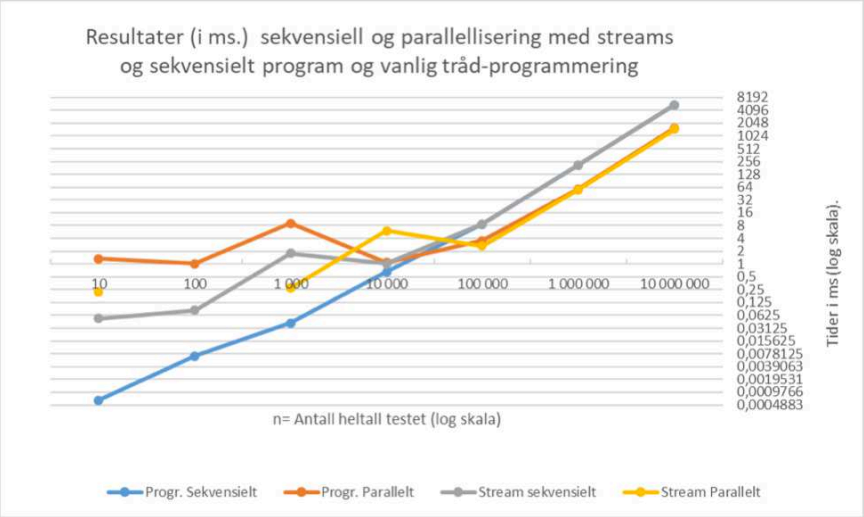


Fig 13.1 *Eksekveringstider for fire ulike måter å finn antall primtall < n (10 <= n <= 20 mill) med isPrime().Begge aksene er logaritmiske, dvs at hvert tall på x-aksen er n 10 ganger så stort som det forrige, mens verdiene på y-aksen er hver linje 2 ganger så stor som linjen under. Vi ser at sekvensielt program (blå)er raskest for alle n < 10 000, mens den trådbaserte parallelle programløsningen (rød) er langsomst i det samme området. I området for n > 10 000 blir kjøretiden for de to sekvensielle (program og strøm) like, og tilsvarende for de to parallelle løsningene. Merk at de to parallelle er begge der nesten 4 ganger så raske som de sekvensielle.*

Vi kan fra dette denne testing av *isPrime(),* at for korte kjøretider, dvs. mindre enn ca. 1 sekund er en vanlig sekvensiell løsning raskest, men for tider større enn ca. 4 sekunder er begge de to parallelle løsningene raskest. Det må også bemerkes art den parallelle strømmen var langt kortere og enklere å programmere enn den trådbaserte parallelle løsningen.

Det er imidlertid slik at strømmer ikke er like velegnet for å løse alle problemer – ikke alle algoritmer kan enkelt formuleres som filtrering av en strøm av objekter. Det er f.eks. vanskelig å tenke seg at en rekursiv algoritme som Kvikksort kan løses effektivt med strømmer. Man måtte vel da først gjøre alle tallene man skulle sortere til en liste av objekter med ett tall i hvert objekt, og så lese denne listen først én gang for å finne pivot-elementet; så lese den originale listen to ganger til, andre gang for å lage en liste av alle elementene med verdi <= pivot-objektet, så lese original-listen for tredje gang for å skjøte på den nye lista med de elementer som er større. Dette er tre ganger lesning av original-listen. I Kap 12 har vi sett hvor langsomt det kan være å operere på lister av slike Integer-objekter. For å fortsette rekursivt bør vi nok da oppbevare dette som to lister, den med elementer mindre enn 'pivot' og den med større enn pivot. Vi kan så fortsette med den 'lille listen på samme måten helt til den stadig mindre lille, venstre listen til sist har lengde =1. Dette høres å være vanskelig og lite effektivt. Viktigere er at primtalls-problemet løses med bare *ett* lite, vanlig sekvensielt program (isPrime(int p)). Det er ikke greit å tenke seg at hvordan man skulle skrive den rekursive logikken som

sekvensielle småprogrammer til de strømmene vi lager. Mulig er det kanskje, men det synes klart at dette vil bli langt mer komplisert og svært mye langsommere enn vanlig (sekvensiell eller parallell) Kvikksort. I java-biblioteket Arrays.sort er det egne sorteringsmetoder (for de ulike typene av basaltyper som byte, int, double,… osv) som også er tilpasset til også å kunne nyttes i en strøm.

```
List<Type> result = list
                    .stream()
                    .sorted(Comparator.comparing(Type::getValue))
                    .toList();
```

Program 13.7.1 *Enkleste måte å bruke en av de to innebygde sorteringsmetodene i Arrays.sort, for arrayer. Man lager en strøm av elementene i en array 'list', som så sorteres med Flette-sortering for lange arrayer og innstikk-sortering for kortere arrayer (Tim-sort), og som til sist omgjøres til en liste 'result' av de nå sorterte elementene.*

For å konkludere – strømmer gir oss en effektiv løsning på visse sekvensielle problemer. Men mer tyngre algoritmer hvor vi f.eks. rekursivt må finne en løsning som ved sortering eller 'den konvekse innhyllingen av en punktmengde', som er en obligatorisk oppgave i kurset, synes strømmer lite effektivt og bare mer komplisert.

13.8 En noe alternativ start på et kapittel om Strømmer i Java (fra Eyvind W.Axelsen)

Strømmer er rett og slett en «strøm» av objekter. En strøm har den viktige egenskapen at den er såkalt «lazy», noe som gjør at hvert objekt «evalueres» når det er bruk for det. Det vil si at en strøm godt kan være uendelig (eller uten kjent lengde). Det er nyttig når man f.eks. strømmer ting over nettet.

Et enkelt eksempel:

```
IntStream is = IntStream.iterate(0, i -> i + 1).filter(i -> i %2 == 0);
```

Her har vi en «uendelig» strøm (selv om vi i praksis er begrenset av størrelsen på en int), som så filtreres på alle som er delelige med 2. Om du legger inn dette i programmet ditt, så stopper det likevel ikke opp, fordi evalueringen er «lazy». Det skjer ingen ting før vi faktisk spør om verdier. Og når vi spør om verdier, så kan vi be om en og en, og ikke havne i en uendelig løkke selv om strømmen i seg selv er uendelig.

Dette poenget er viktig å få med i et kapittel som omhandler strømmer bør vel egentlig starte med et litt mer overordnet blikk på hva en strøm er, og hvorfor det er nyttig.

14 OPPSUMMERING – OM PARALLELLISERING AV ET PROBLEM

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

1. For at det skal være vits å parallellisere et problem, må det utføre *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 tar mindre tid enn 1 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 om lag 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 bytte 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 løsning på neste nivå osv. Det er for noen rekursive algoritmer som Kvikksort mulig å endre disse slik at vi kan starte med full parallellitet med alle trådene [1] [A full parallel Quicksort algorithm for multicore processors] .
For andre rekursive algoritmer som Flettesortering er det også 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 den andre enden endene flettes for å finne de største elementene i parallell
[2] [A faster, all parallel Merge sort algorithm for multicore processors].

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 sKvikk og sorter den så med Arrays.sort() fra Java-biblioteketS Skriv ut tiden for denne og sammenlign med tiden for kvikksorttiden.
2. I den parallelle versjonen av Kvikksort, pKvikk, fjern de rekursive kallene og løs problemet bare med tråder. Kommenter kjøretidene og antall tråder du får. Er dette lurt?
3. Til tross for advarselen i kap. 13.7, prøv å lage en strømbasert løsning på Kvikksort og sammenlign kjøretiden til denne med Arrays.sort() fra Java-biblioteket.
4. Skriv et program for 'En Selgers rundtur'; først sekvensielt (rekursiv) og så en blanding av parallellt 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 *én* 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 4. *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 ekstremt raskt med n.*

5. Lag en versjon av parallell Kvikksort hvor du bare bruker én sykliske barriere. Hint: Ser du på kall-treet ser du at det er ett kall på **pKvikk** 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 7.1?

REFERANSER:

Siden UiO nå har fjernet alle *direkte* referanser til de ansattes arbeider i et åpent journalsystem vi nyttet før 24. februar 2024, er det enklest å finne disse to artiklene (den første på NIK2015, den andre på NIK2018) via https-adressen:

https://www.researchgate.net/profile/Arne-Maus/research

Trykk først på artikkel-navnet, og så på [full-text available], så får du artiklene nedlastet i pdf.

[1] - A full parallel Quicksort algorithm for multicore processors

[2] - A faster, all parallel Merge sort algorithm for multicore processors

https://www.ntnu.no/ojs/index.php/nikt/article/view/5385/4861

https://www.ntnu.no/ojs/index.php/nikt/article/view/5268/4744

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://www.intel.com/content/www/us/en/newsroom/news/intel-core-14th-gen-desktop-processors.html

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/

OM KI (AI) BEREGNINGER OG SPESIELLE AI-CHIPS SOM ALLE BRUKER PARALLELLE BEREGNINGER

https://cset.georgetown.edu/wp-content/uploads/AI-Chips%E2%80%94What-They-Are-and-Why-They-Matter.pdf

UTDYPENDE FORKLARINGER PÅ MANGE JAVA-RELATERTE BEGREPER

https://jenkov.com/tutorials/java-concurrency/java-memory-model.html

Gordon E. Moore, Co-founder
Intel Corporation

Progress In Digital Integrated Electronics

Complexity of integrated circuits has approximately doubled every year since their introduction. Cost per function has decreased several thousand-fold, while system performance and reliability have been improved dramatically. Many aspects of processing and design technology have contributed to make the manufacture of such functions as complex single chip microprocessors or memory circuits economically feasible. It is possible to analyze the increase in complexity plotted in Figure 1 into different factors that can, in turn, be examined to see what contributions have been important in this development and how they might be expected to continue to evolve. The expected trends can be recombined to see how long exponential growth in complexity can be expected to continue.

A first factor is the area of the integrated structures. Chip areas for some of the largest of the circuits used in constructing Figure 1 are plotted in Figure 2. Here again, the trend follows an exponential quite well, but with significantly lower slope than the complexity curve. Chip area tor maximum complexity has increased by a factor of approximately 20 from the first planar transistor in 1959 to the 16,384-bit charge-coupled device memory chip that corresponds to the point plotted for 1975, while complexity, according to the annual doubling law, should have increased about 65,000-fold. Clearly much of the increased complexity had to result from higher density of components on the chip, rather than from the increased area available through the use of larger chips.

Figure 1 Approximate component count for complex integrated circuits vs. year of Introduction.

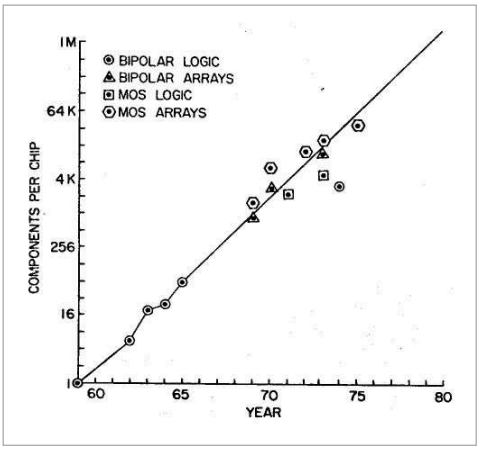
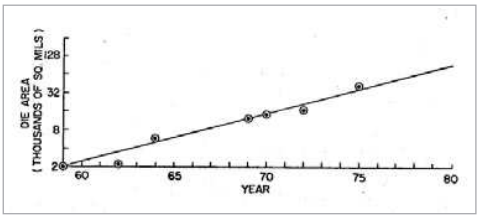


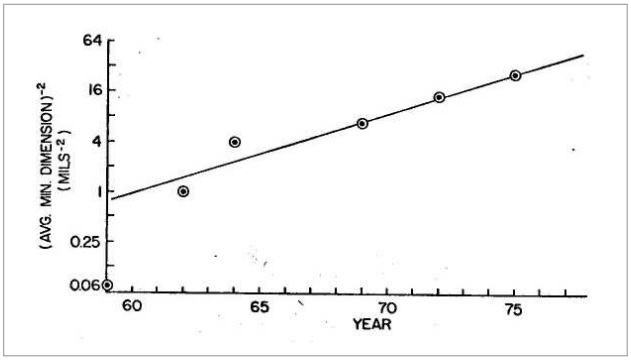
Figure 2 Increase in die area for most complex integrated devices commercially available.



Density was increased partially by using finer scale microstructures. The first integrated circuits of 1961 used line widths of 1 mil (~25 micrometers) while the 1975 device uses 5 micrometer lines. Both line width and spacing between lines are equally important in improving density. Since they have not always been equal,

G. E. Moore, "Progress in Digital Integrated Electronics." © 1975 IEEE. Reprinted, with permission, from Technical Digest 1975. International Electron Devices Meeting, IEEE, 1975, pp. 11-13.

Figure 3 Device density contribution from the decrease in line widths and spacings.

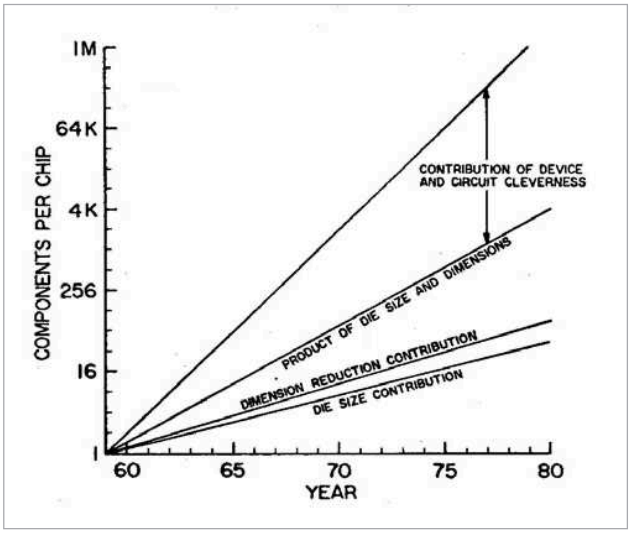


the average of the two is a good parameter to relate to the area that a structure might occupy. Density can be expected to be proportional to the reciprocal of area, so the contribution to improve density vs. time from the use of smaller dimensions is plotted in Figure 3.

Neglecting the first planar transistor, where very conservative line width and spacing was employed, there is again a reasonable fit to an exponential growth. From the exponential approximation represented by the straight line in Figure 3, the increase in density from this source over the 1959-1975 period is a factor of approximately 32.

Combining the contribution of larger chip area and higher density resulting from geometry accounts for a 640-fold increase in complexity, leaving a factor of about 100 to account for through 1975, as is shown graphically in Figure 4. This factor is the contribution of circuit and device advances to higher density. It is noteworthy that this contribution to complexity has been more important than either increased chip area or finer lines. Increasingly the surface areas of the integrated devices have been committed to components rather than to such inactive structures as device isolation and interconnections, and the components themselves have trended toward minimum size, consistent with the dimensional tolerances employed.

Figure 4 Decomposition of the complexity curve into various components.



Can these trends continue?

Extrapolating the curve for die size to 1980 suggests that chip area might be about 90,000 sq. mils, or the equivalent of 0.3 inches square. Such a die size is clearly consistent with the 3 inch wafer presently widely used by the industry. In fact, the size of the wafers themselves have grown about as fast as has die size during the time period under consideration and can be expected to continue to grow. Extension to larger die size depends principally upon the continued reduction in the density of defects. Since the existence of the type of defects that harm integrated circuits is not fundamental, their density can be reduced as long as such reduction has sufficient economic merit to justify the effort. I see sufficient continued merit to expect progress to continue for the next several years. Accordingly, there is no present reason to expect a change in the trend shown in Figure 2.

With respect to dimensions, in these complex devices we are still far from the minimum device sizes limited by such fundamental considerations as the charge on the electron or the atomic structure of matter. Discrete devices with sub-micrometer dimensions show that no basic problems should be expected at least until the average line width and

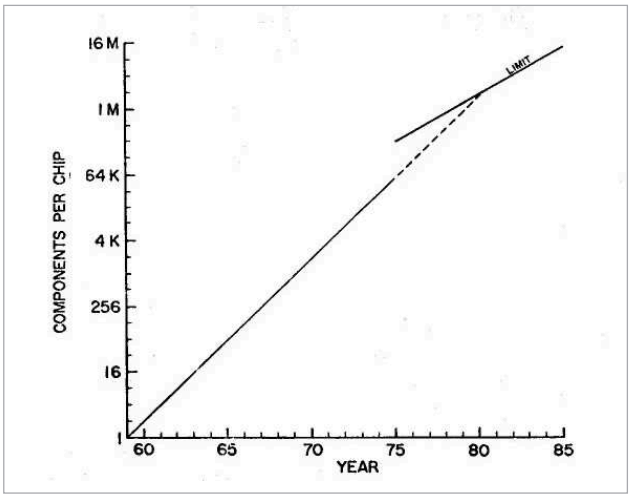
spaces are a micrometer or less. This allows for an additional factor of improvement at least equal to the contribution from the finer geometries of the last fifteen years. Work in non-optical masking techniques, both electron beam and X-ray, suggests that the required resolution capabilities will be available. Much work is required to be sure that defect densities continue to improve as devices are scaled to take advantage of the improved resolution. However, I see no reason to expect the rate of progress in the use of smaller minimum dimensions in complex circuits to decrease in the near future. This contribution should continue along the curve of Figure 3.

With respect to the factor contributed by device and circuit cleverness, however, the situation is different. Here we are approaching a limit that must slow the rate of progress. The CCD structure can approach closely the maximum density practical. This structure requires no contacts to the components within the array, but uses gate electrodes that can be at minimum spacing to transfer charge and information from one location to the next. Some improvement in overall packing efficiency is possible beyond the structure plotted as the 1975 point in Figure 1, but it is unlikely that the packing efficiency alone can contribute as much as a factor of four, and this only in serial data paths. Accordingly, I am Inclined to suggest a limit to the contribution of circuit and device cleverness of another factor of four in component density.

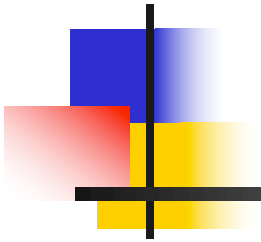
With this factor disappearing as an important contributor, the rate of increase of complexity can be expected to change slope in the next few years as shown in Figure 5. The new slope might approximate a doubling every two years, rather than every year, by the end of the decade.

Even at this reduced slope, integrated structures containing several million components can be expected within ten years. These new devices will continue to reduce the cost of electronic functions and extend the utility of digital electronics more broadly throughout society.

Figure 5 Projection of the complexity curve reflecting the limit on increased density through invention.

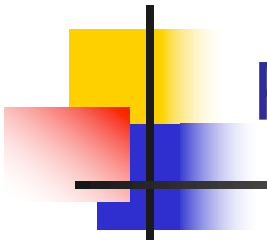


0305/TSM/LAI/XX/PDF 306956-001US



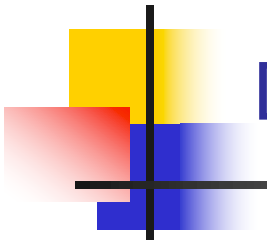
IN3030/IN4330
Effektiv parallellprogrammering
L1, (Uke 3) våren 2024

Eric Jul
Professor
Gruppeleder Programmeringsteknologi
Institutt for Informatikk
Universitetet i Oslo
Norge



Recording of Lectures

- **Recording lectures: I will attempt BUT I make no promises – a few times last spring my recording did not work ☹**
- **Lecture slides are mostly in Norwegian (perhaps a little Danish and English intermixed in a few places…)**



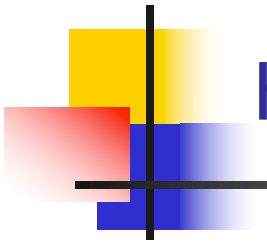
Do not be shy...

- **Ask any question at ANY TIME during the lecture – I shall try to respond fairly quickly ☺**
- **Also, REMEMBER, in education:**

THERE ARE NO STUDPID QUESTIONS, only STUPID ANSWERS!

English

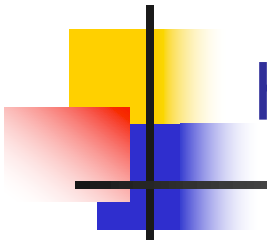
- **I am Danish: so my Danwegian sounds like a terrible dialect of Norwegian – spoken only about 200 km southeast of Kristianssand.**
- **Many Norwegians understand what I say, but, alas, many that have Norwegian as a second language have trouble doing so**
- **And there are a few international students in the course.**
- **So I will lecture in English.**



Hvem

- **Meg**
- **Arne Maus –** original course designer, Emeritus
- **Michael Kirkedal Thomsen –** new førsteamanuensis in PT ☺

- **2 Teaching Assistants**
  - **Oliver, Friday**
  - **Michal, Monday and Wednesday**
- **About 110 students signed up (usually somewhat fewer show up)**



Hvorfor dette kurs?

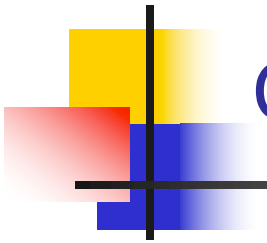
- **Love of *Learning by Doing***
- **Oppleve progammer, der gjør en forskjell!!**
- **Uttnytte *multicore – multikjerne***
- ***Teori er bra, MEN praksis essentiel!***
- ***Bli bedre programmør!!***
- ***… And have fun doing so!!!***

Litt om Eric

- **Ph.D. University of Washington, 1989**
- **Dansk-amerikaner**
- **Bor i Danmark – pendler til Oslo ca 2-3 gange/måned**
- **1989-2000: Førsteamauensis Københavns Universitet**
- **2000-2009: Professor Københavns Universitet**
- **2009-2015: Bell Labs, Dublin & Professor II, IfI**
- **2016- Professor IfI**
- **2019- Gruppeleder *Programming Technology***

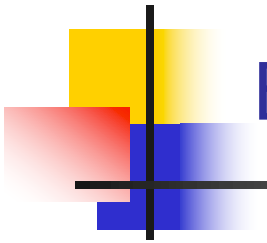
- ***Favorite hobbies: Skiing and Gliding (Seilfly) and a little bicycling***

Litt mere om Eric



Communication

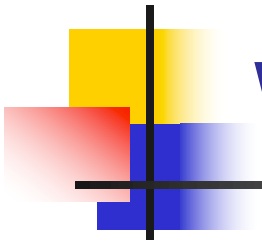
- **Please keep up with the messages on the web site: The course website is THE source of information**
- **Use the Hjeplelærer ☺**
- **Ask questions when in doubt – we will be setting up a Q&A Forum**
- **IN4330 students – use the IN3030 web site**



Bakgrunn IN3030

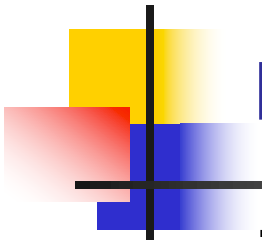
- **Kurs er fra 2014 – tidligere INF2440**
- **Laget av Arne Maus – siden 2018 Emeritus**
- **Overtaket av Eric 2018**
- **I 2019 endring: INF2440 -> IN3030/IN4330**

- **IN4330: versjonen for Masters student: bruk BARE IN3030 web!**



Why are YOU here??

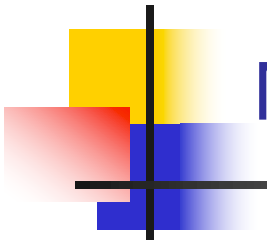
- **What do you expect from the course?**



Let's get started on parallelism

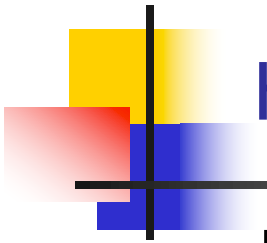
Two examples:

- **Plant a tree**
  - **Embarrassingly paralizable!**
- **Make a baby**
  - **Inherently sequential!**



Motivation for Parallelle Programmer

- **Maskiner kan bestå av flere CPU-er**
  - Hver CPU kan utføre programmer uavhengig av de andre CPUer
- **Hver CPU kan have flere kjerner**
  - Hver kjerne kan utføre programmer
- Vi vil gjerne utnytte disse muligheter for parallellisme

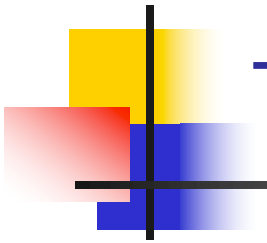


Hva vi skal lære om i dette kurset:

Lage parallelle programmer (algoritmer) som er:

- **Riktige**
  - Parallelle programmer er klart vanskeligere å lage enn sekvensielle løsninger på et problem.
- **Effektive**
  - dvs. raskere enn en sekvensiell løsning på samme problem

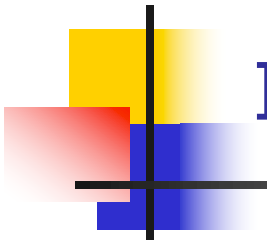
- Lære hvordan man parallelliserer et riktig, sekvensielt program + lage egne parallelle algoritmer som ikke bare er en slik parallellisering;
- Lære de mange problemene vi støter på og hvordan disse kan takles.
- Kurset er **empirisk** (med tidsmålinger), ikke basert på en teoretisk *modell* av parallelle beregninger,
- Vi oppfatter programmet som en god nok modell av det problemet vi skal løse. Vi trenger ingen modell av modellen.
- Presenterer en klassifikasjon av parallelle algoritmer (nytt).



Tre grunner til å lage parallelle programmer

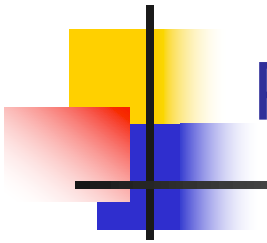
1) Skille ut aktiviteter som går **langsommere** i en egen tråd.
   - Eks: Tegne grafikk på skjermen, lese i databasen, sende melding på nettet. Asynkron kommunikasjon.
2) Av og til er det **lettere** å programmere løsningen som flere parallelle tråder. Naturlig oppdeling.
   - Eks: Kundesystem over nettet hvor hver bruker får en tråd.
   - Hele operativsystemet har mye parallellitet – 1572 aktive tråder i Windows 7 akkurat da denne foilen blev skrevet i 2017 – og 1921 aktive tråder i Mac OS X Sierra.
3) Vi ønsker **raskere** programmer, raskere algoritmer.
   - Eks: Tekniske beregninger, søking og sortering.

Dette kurset legger nesten all vekt på raskere algoritmer



IN3030/IN4330

- Et relativt UNIKT kurs – set internationalt.
- Planlagt fem obliger - ca. 2-3 uker per oblig.
  - De individuelle innleveringer : Man kan samarbeide om algoritmer, men **ikke** ha helt lik eller delvis felles kode med andre. Rapport skal skrives selv.
- En oblig er ikke bare innlevering av ett eller flere parallelle programmer, men **også en liten rapport** om de testene man har gjort: hastighetsmålinger på disse for ulike størrelse av data med konklusjoner – f.eks speedup + O( ) og en forklaring på resultatene .
- Gruppetimer:
  - Jobbing med ukeoppgaver og obligene



Pensum

- Ingen dekkende lærebok er funnet, men:
  - **Kompendium av Arne Maus:** legges opp på web
  - **Det som foreleses (Powerpoint slides) + oppgavene (obliger + ukeoppgaver) er pensum.**

  - **Bra bok:** Brian Goetz, T.Perlis, J. Bloch, J. Bobeer, D. Holms og Doug Lea::"Java Concurrency in practice", Addison Wesley 2006
  - Kap. **18 og 19** i A. Brunland, K. Hegna, O.C. Lingjærde, A. Maus:"Rett på Java" 3.utg. Universitetsforlaget, 2011.
  - I tillegg leses *fra en maskin på Ifi* kap 1 til 1.4, hele 2 og 3.1 til 3.7 (hopp over programeksemplene) i :

http://www.sciencedirect.com/science/book/9780124159938

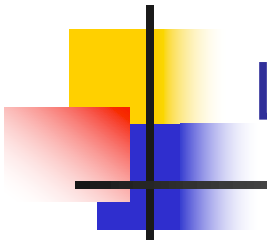
(Hvis leses utenfor Ifi, så koster det $!)

I dag – teori og praksis

- Ulike maskiner og kurs – hvor plasserer INF2440 seg?
- Begrunnelse for multikjerne CPU og parallelle løsninger/algoritmer.
- Parallelle løsninger på et problem er lengere (ofte minst dobbelt så lang kode) og (en god del) vanskeligere å lage enn en sekvensielt algoritme som løser samme problem.
- Den eneste grunnen til å lage parallelle algoritmer er at de går fortere enn samme sekvensielle algoritme – i alle fall for tilstrekkelig stor n (= antall data).
- Vi måler hvor-mange-ganger-fortere-det-går – speedup S:

$$S = \frac{tid\ (sekvensiell\ algoritme)}{tid\ (parallell\ algoritme)}$$

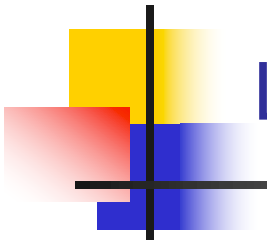
- som da skal være > 1 , men vi skal også lage og teste programmer som har S < 1 og forklare hvorfor.



Lineær speedup ?

- Selvsagt ønsker vi lineær speedup – dvs. bruker vi *k* kjerner skulle det helst gå *k* ganger fortere enn med 1 kjerne.
- Meget sjelden at det kan oppnås (mer om det siden)
- Kan superlineær speedup oppnås?
  - *PLEASE THINK ABOUT THIS!*
- Speedup: a central metric.

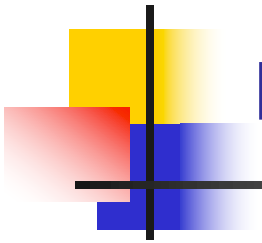
Lineær speedup analogier



- Selvsagt ønsker vi lineær speedup, MEN:
- Noen problem er **nemme** at parallellisere: Eksempel: 10 personer kan plante 10 treer ca 10 gange fortere end 1 person kan plante 10 treer.
- Andre problemer er *vanskeligere* at parallellisere: Eksempel: hvis 1 person kan samle et Lego-set på 10 timer, så vil det være vanskelig for 10 personer at samle det på 1 time – der er avhengigheter mellom delene.
- Andre problemer er nærmest ***umulig*** at parallellisere: Eksempel: hvis 1 kvinne kan lage et barn på 9 måneder, kan 9 kvinner så lage et barn på 1 måned?

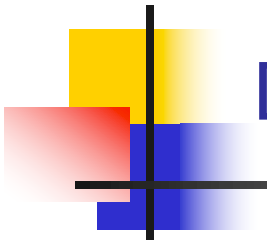
Flynns klassifikasjon av datamaskiner:

	Single Instruction	Multiple Instruction
Single Data	SISD : Enkeltkjerne CPU	MISD : Pipeline utførelse av instruksjoner i en CPU. Flere maskiner som av sikkerhetsgrunner utfører samme instruksjoner.
Multiple Data	SIMD : **GPU** – samme operasjon på mange elementer (en vektor) Det finnes også slike SSE – instruksjoner på Intel og AMDs CPU-er	**MIMD :** klynge av datamaskiner, og **Multikjerne CPU**



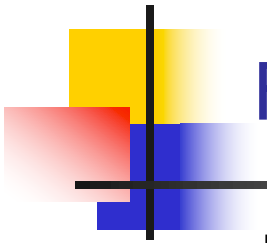
Dette kurset handler om multikjernemaskiner

- Nå har vi multikjerne maskiner!
  - Din 2500 NOK bærbar: 2-4 kjerner
  - Min 2023 MacBook Pro: 19 kjerner
  - Stor server: 64-256 kjerner
- Hvordan uttnytter vi dem??
- Svar: vi bruker tråde!



Dette kurset handler om tråder og effektivitet

- Hva er tråder
  - Se litt på maskinen
  - Se på operativsystemet
  - Hvordan skal vi oppfatte en tråd i et Java-program
  - Senere se på kompileringen og kjøring av Java-kode
- Hvordan måle effektivitet
  - Hvordan ta tiden på ulike deler av et program; både:
    - Den sekvensielle algoritmen
    - En eller flere parallelle løsninger
  - I dag: Enkel tidtaking
  - Neste gang: Bedre tidtaking
- Praktisk i dag
  - Standard måte å starte programmet med tråder

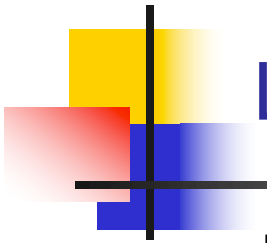


Flere mulige synsvinkler

Mange nivåer i parallellprogrammering:

1. Maskinvare
2. Programmeringsspråk
3. Programmeringsabstraksjon.
4. Hvilke typer problem egner seg for parallelle løsninger?
5. Empiriske eller formelle metoder for parallelle beregninger

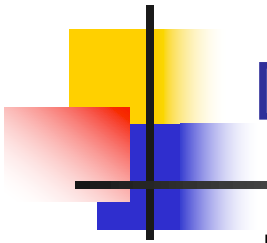
IN3030/IN4330: Parallellprogrammering av ulike algoritmer med tråder på multikjerne CPU i Java – empirisk vurdert ved tidsmålinger.



Learning-by-doing

Dette kurs er et Learning-by-doing kurs!

Utbyttet ditt er avhengig av **DIN** innsats!



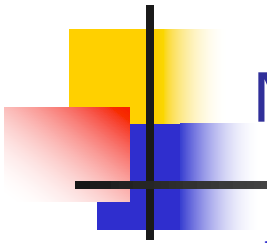
DIN innsats

Dette kurs er et Learning-by-doing kurs!

Go program lots of parallel programs

AND

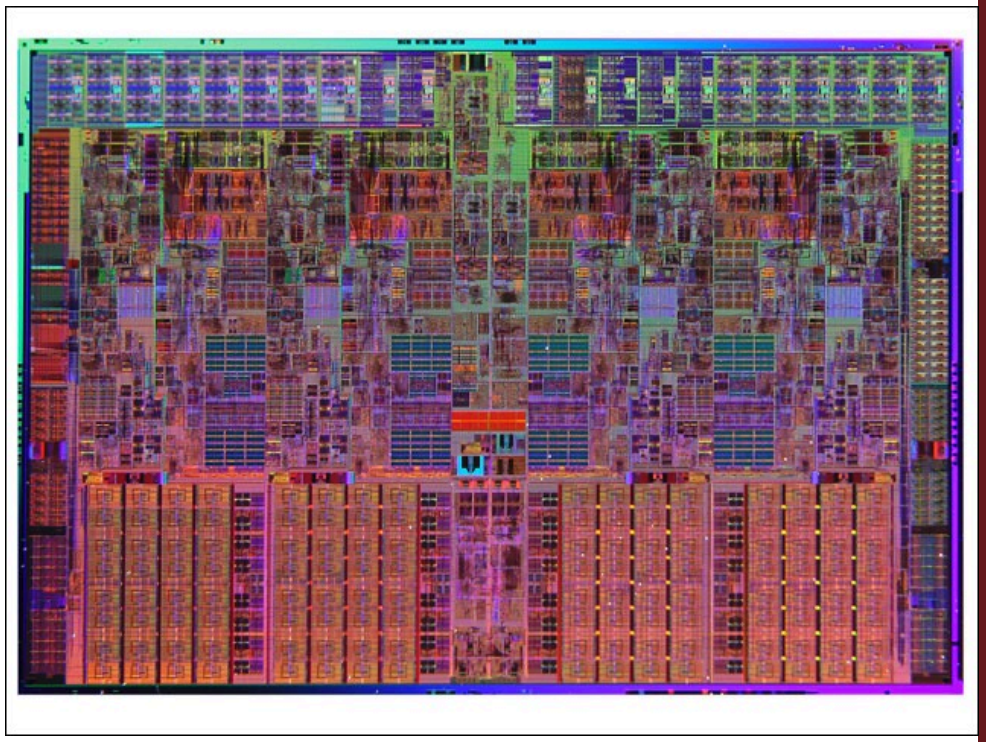
make them ***FAST***

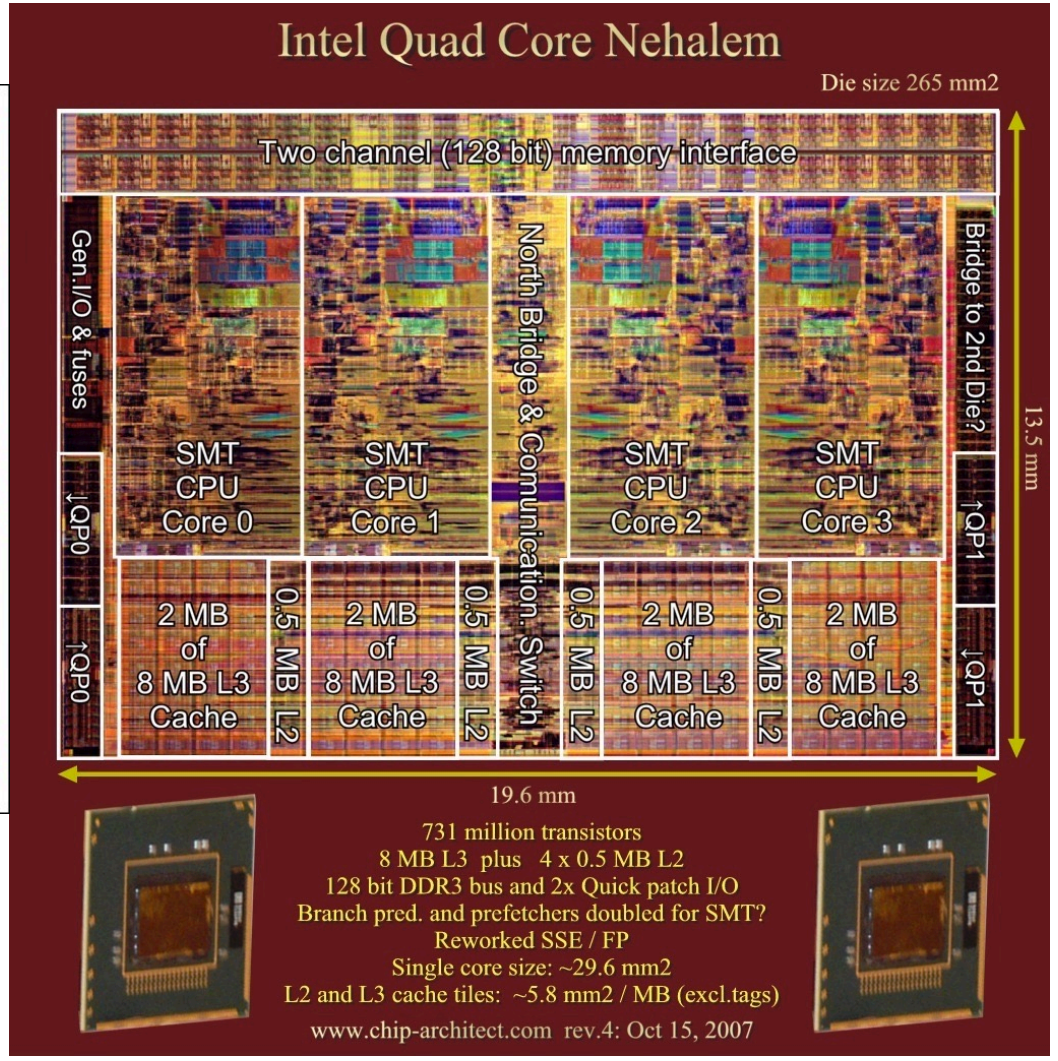


Maskinvare og språk for parallelle beregninger og Ifi-kurs

1. Grafikkkort GPU med 2000+ små-kjerner, **IN5050**
   - Eks Nvidia med flere tusen småkjerner (SIMD – maskin)
2. Mange maskiner løst koblet over internett. **IN5570**
   - Planetlab – world-wide testbed
   - Emerald – språk til distribuert programmering
3. Mange maskiner løst koblet i utkanten av nettet. **IN5600**
   - Fog Computing
4. Teoretiske modeller for beregningene **IN5170**
   - PRAM modellen og formelle modeller (f.eks FSM)

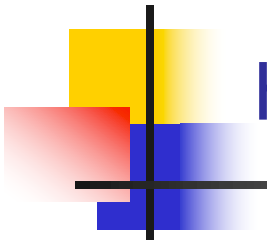
Multikjerne - Intel Multicore Nehalam CPU





Mange ulike deler i en Multicore CPU – bla. en pipeline av maskininstruksjoner; kjernene holder på med 10 til 20 instruksjoner ***samtidig*** dvs. instruksjonsparallellitet

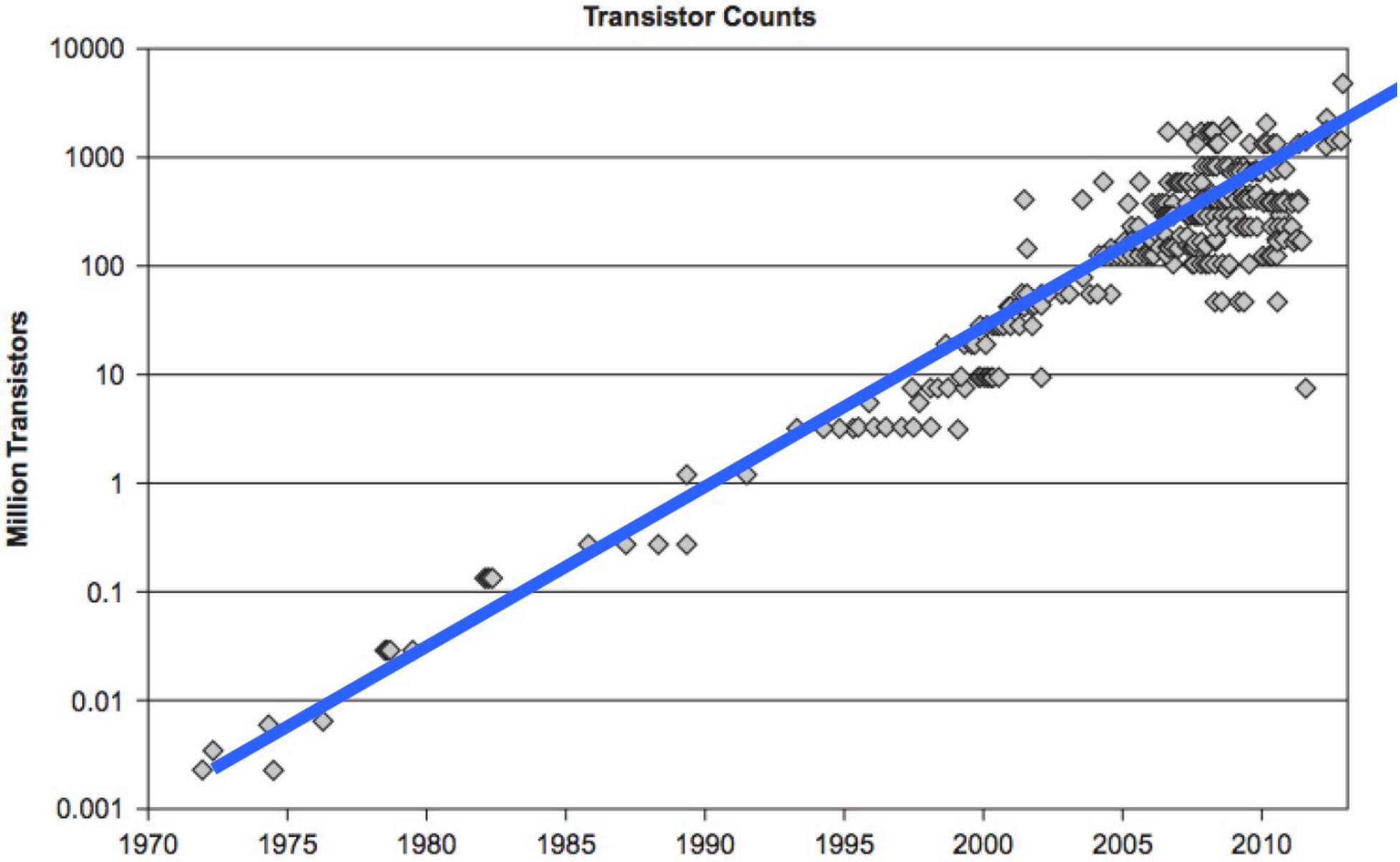
Hvorfor får vi multikjerne CPUer ?

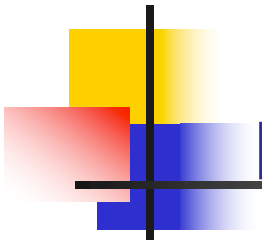


- Hver 18-24 måned dobler antall transistorer vi kan få på en brikke: Moores lov
- Vi kan ikke lage raskere kretser fordi da vil vi ikke greie å luftkjøle dem (ca. 120 Watt på ca. 2x2 cm – varmere enn en rødglødende kokeplate). Og der er kvantemekaniske begrensninger også.
- Med f.eks dobbelt så mange transistorer ønsker vi oss egentlig en dobbelt så rask maskin, men det vi får er dessverre 'bare' dobbelt så mange CPU-kjerner.
- Med flere regnekverner (kjerner) må vi få opp hastigheten ved å lage parallelle programmer !

Transistors per Processor over Time

Continues to grow exponentially (Moore's Law)



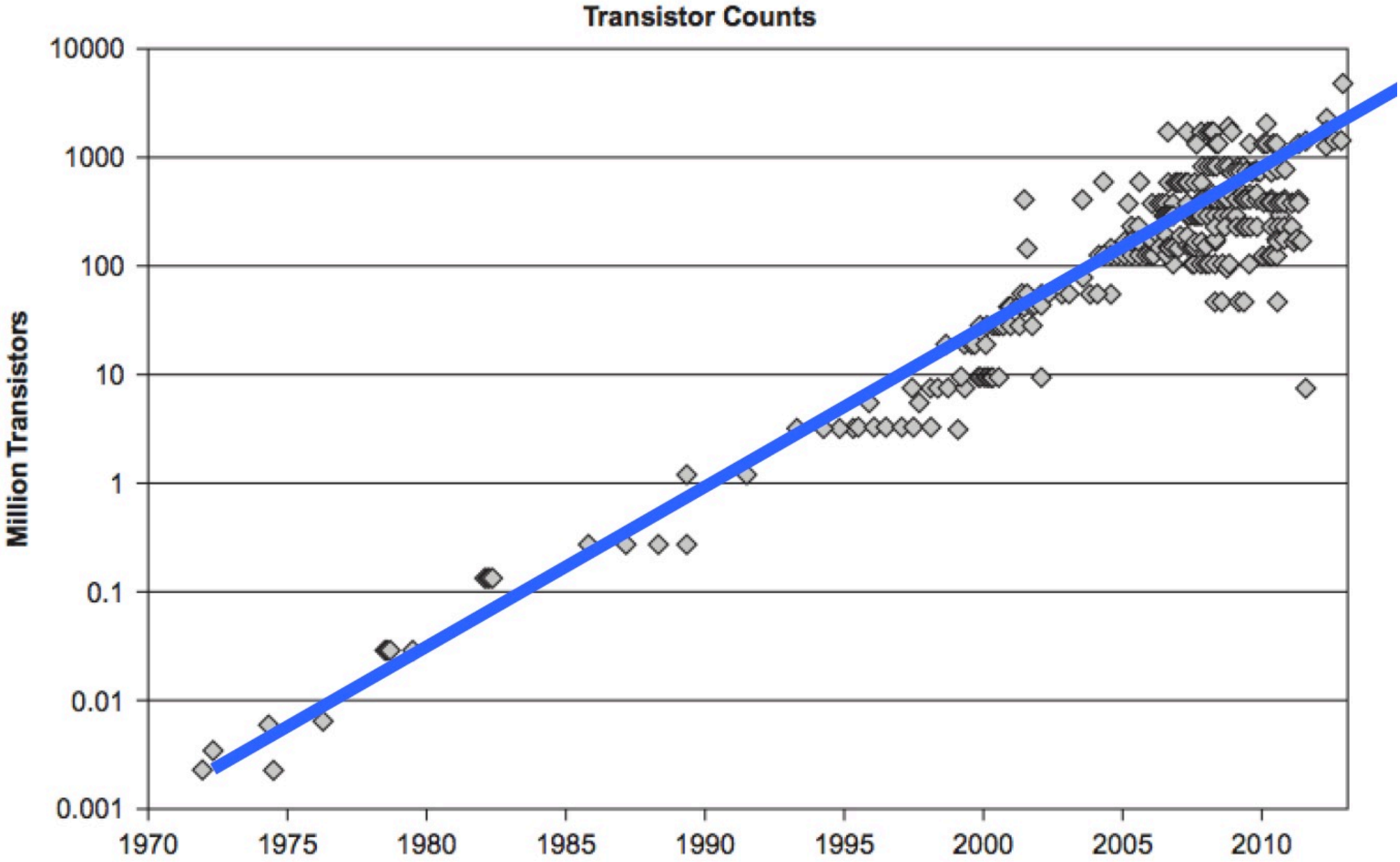


Moore's «Law»

- How many more transistors on a chip?
- Effect on speed of processor

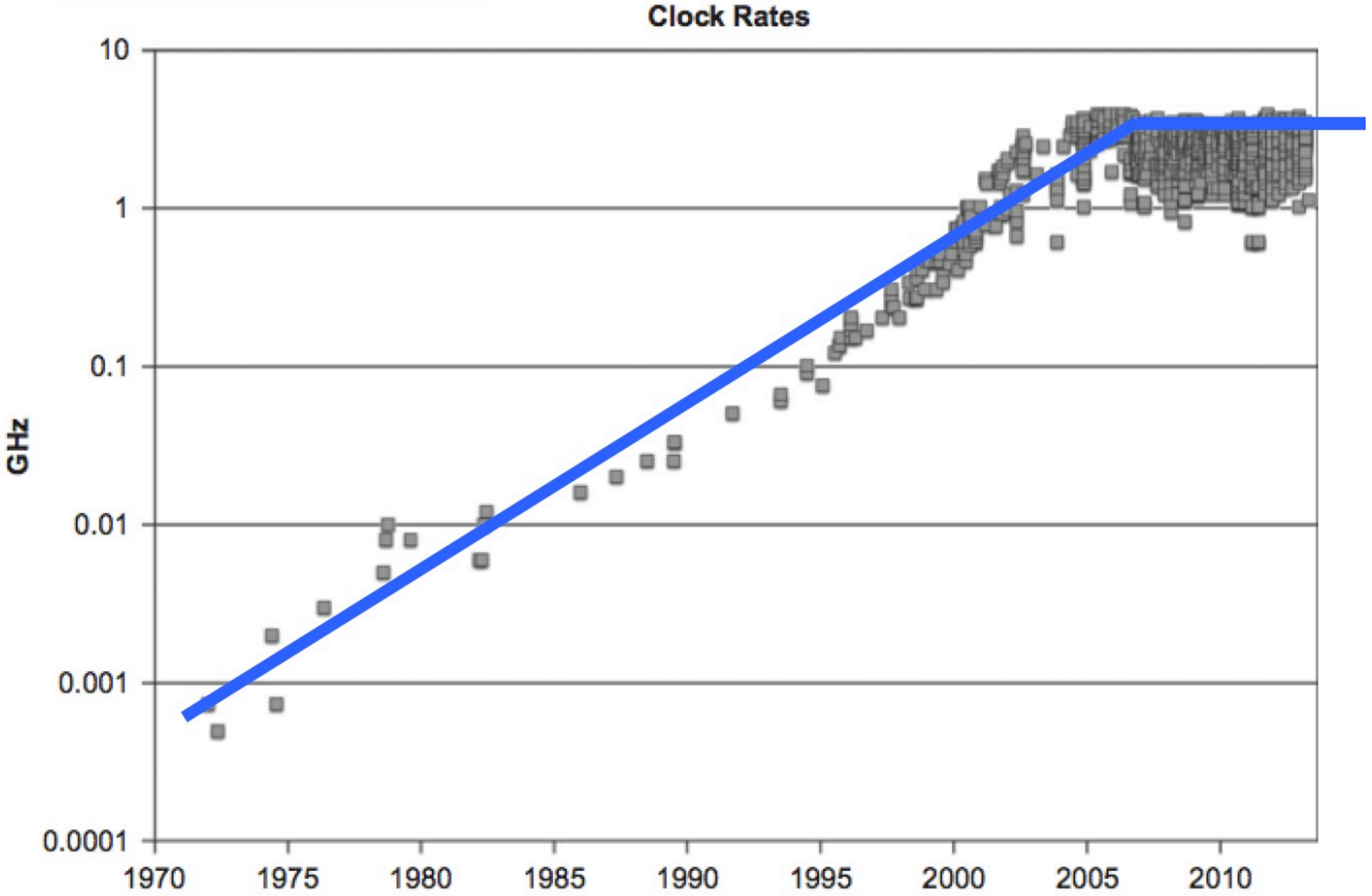
Transistors per Processor over Time

Continues to grow exponentially (Moore's Law)



Processor Clock Rate over Time

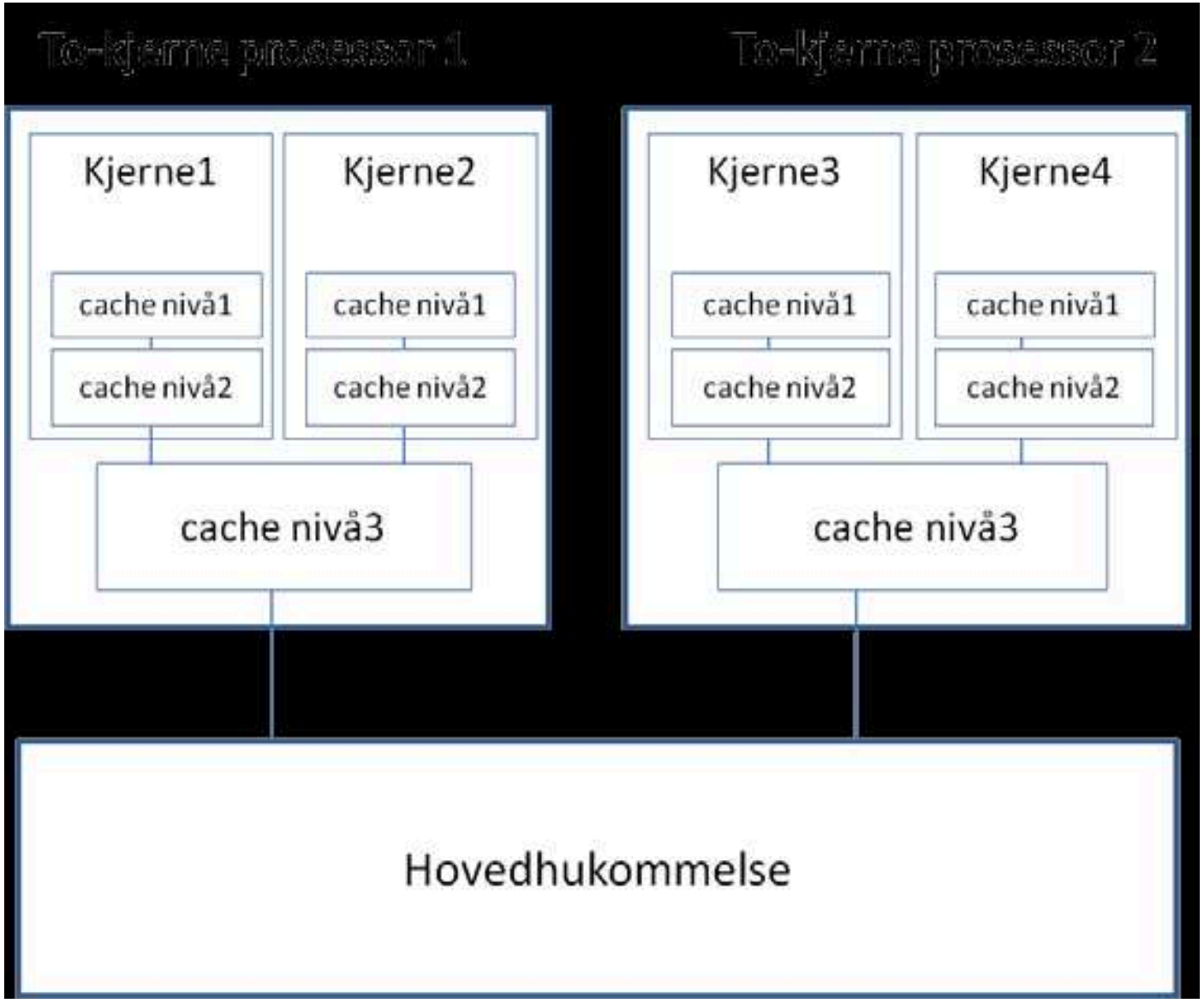
Growth halted around 2005



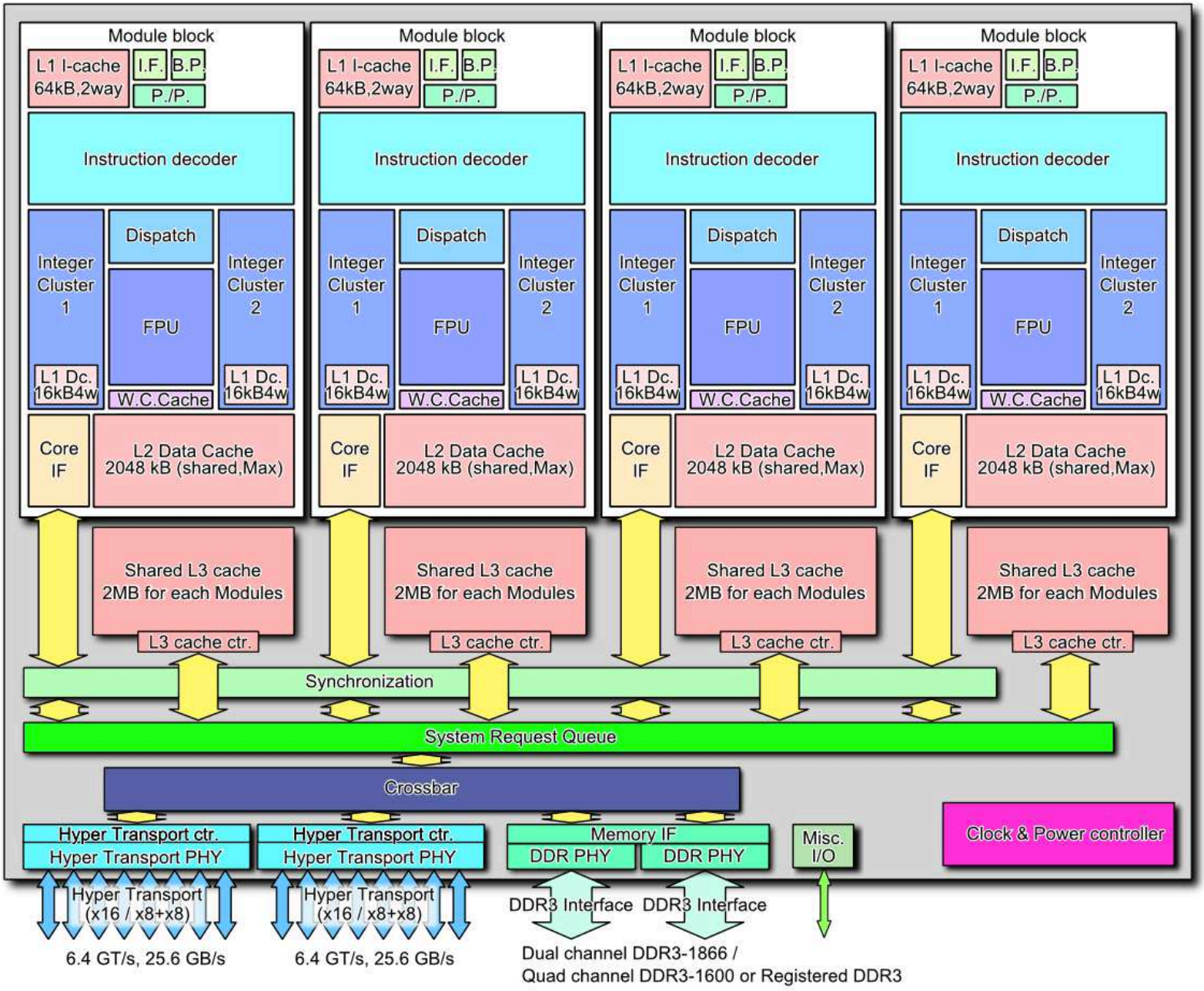
Maskin 1980 (uten cache)

Figur 19.1 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. Intel 8080: 1 MHz CPU 64 kbyte minne

Maskin ca. 2010 med to dobbeltkjerne CPU-er



Hukommelses-systemet i en 4 kjerne CPU – mange lag og flere ulike beregningsmoduler i hver kjerne.:

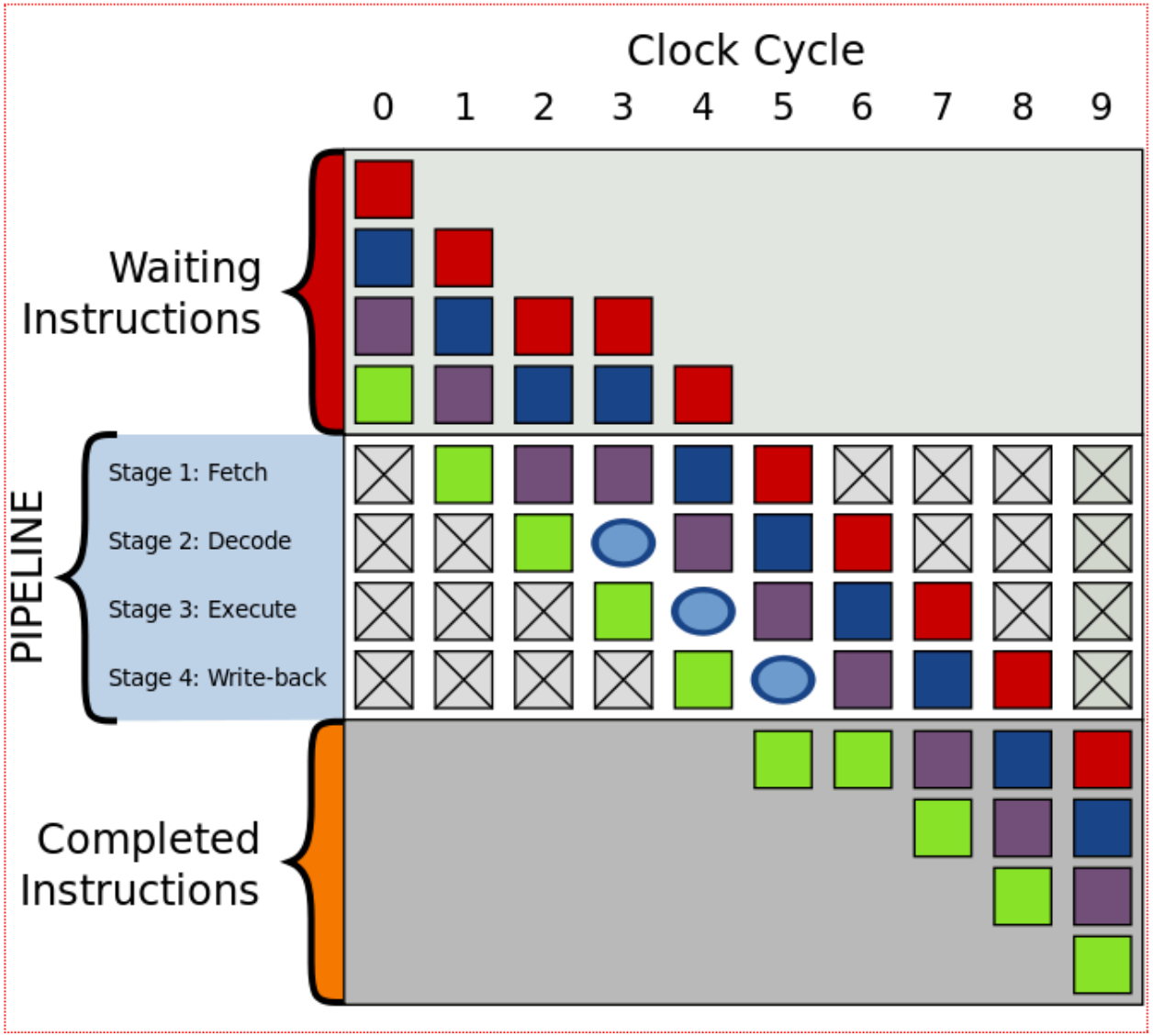


Hvordan tar vi hensyn til cache-systemet for å få raskere programmer?

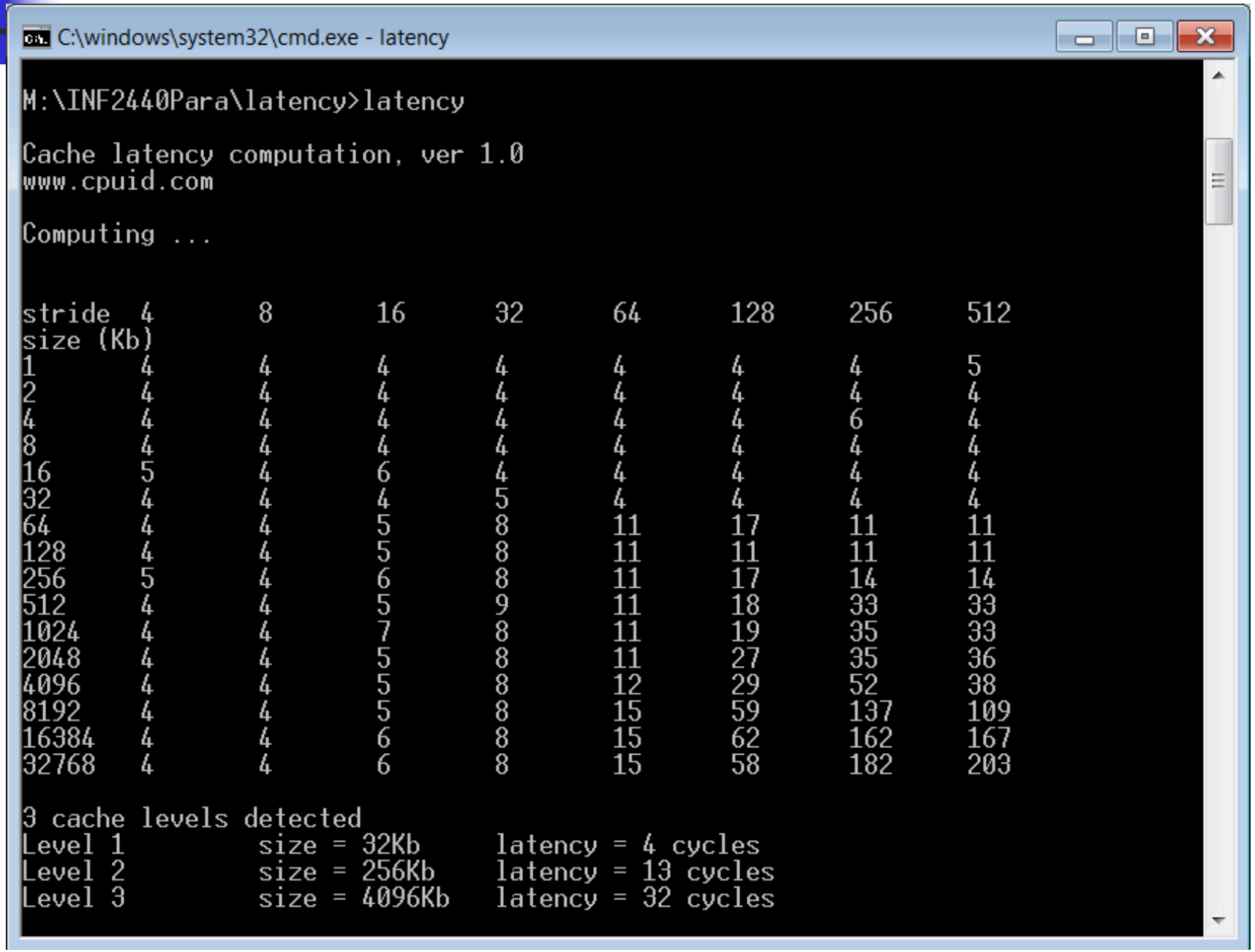
- Vi ser bare på data-cachene (lite å hente på instruksjonene)
- Viktig å vite er at hver gang vi skal hente data i hovedlageret , får vi en cach-linje = 64 byte = f.eks 8 heltall (int)
- Det er svært begrenset plass i cachene, og en cach-linje som ikke har vært brukt på 'lenge' vil bli 'kastet ut'(overskrevet av en annen, nyere) cache-linje
- Slik er raskest:
  - Jobber på få data (korte deler av en array) 'lenge' av gangen – ikke hoppe rundt.
  - Helst gå forlengs eller baklengs gjennom data (arrayene) (i, i+1,.. eller: i, i-1,..)

Vi må lage slike cache-vennlige programmer !

Instruksjonsparallellitet i en CPU-kjerne. Pipeline – flere instruksjoner (her 4) utføres *samtidig* i raskest mulig rekkefølge.



Test av forsinkelse i data-cachene og hovedhukommelsen - latency.exe (fra CPUZ)



Oppsummering – ideen om at vi har *uniform* aksesstid i hukommelsen er helt galt

- Hukommelses-systemet i en multicore CPU ,Intel Core i5-459 3.3 GHz, – mange lag (typisk aksesstid i instruksjonssykler):

1. Registre i kjernen (1) – 8/32 registre
2. L1 cache (3-4) – 32 Kb
3. L2 cache (13) – 256 kb
4. L3 cache (32) – 8Mb
5. Hovedhukommelsen (virtuell hukommelse) (ca. 200) – 8-64 GB
6. Disken (15 000 000 roterende) = 5 ms – 1000 GB – 1-5 TB
   FlashDisk (ca 2 000 000 les, ca. 10 000 000 skriv) = ca. 1 ms

Helt avgjørede for oss – cache-hukommelse

- Hva er cache
  - Raskere (men også dyrere) hukommelse mellom hovedlageret og kjernene.
  - Vi må ha cache fordi det er så store hastighetsforskjeller mellom en CPU-kjerne og hovedlageret ('main memory')
  - Ofte nå 3-4 lag med cache hukommelser + et antall registere i kjernen (enda raskere enn cache-hukommelsene) som holder data eller intruksjoner
  - Når en kjerne trenger data eller en ny instruksjon (og den ikke har det i et register) leter den nedover i cache-hukommelsene. Først cache level 1 (L1), så L2 cachen, .. , før den går til hovedhukommelsen for data eller instruksjoner.
  - Det finns flere teknikker for å gjøre dette raskt (som pre-fetch , dvs at systemet henter neste data/instruksjon uten at kjernen eksplisitt har bedt om det)

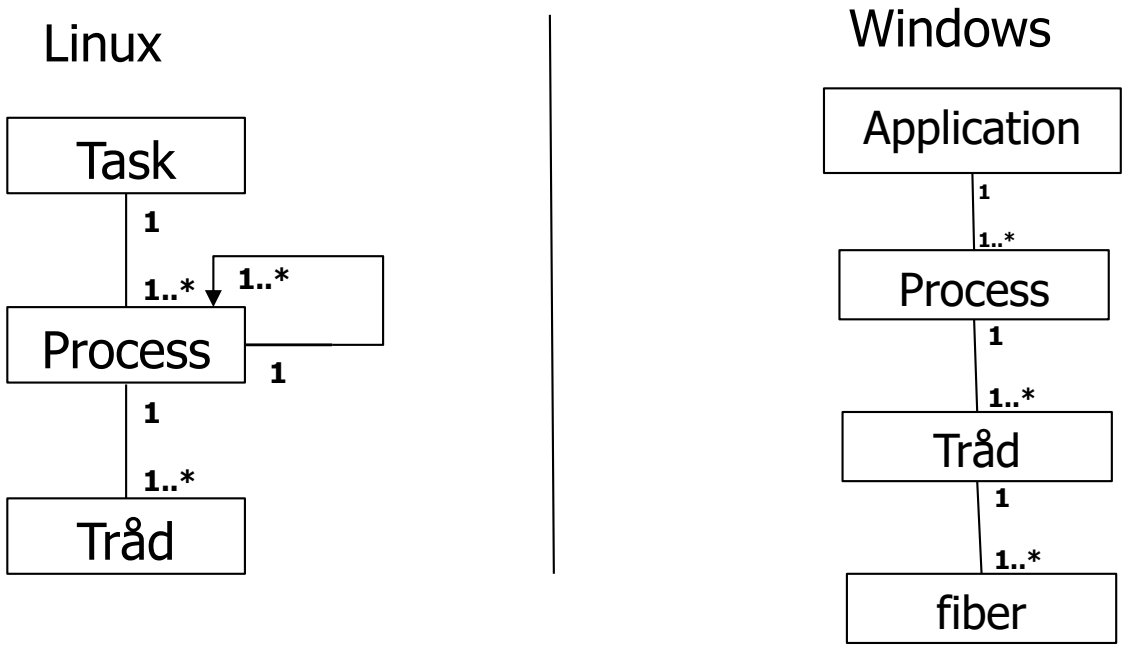
Vi kan ikke hente mer fra automatisk forbedring av hastigheten på våre programmer:

- **Ikke raskere maskiner** – luftkjølingsproblemet
- **Hovedhukommelsen - både** *mye* langsommere enn CPU-ene (derfor cache), og det å sette stadig flere kjerner oppå en langsom hukommelse gir køer.
- **Instruksjons-parallelliteten** i hver kjerne (pipelinen) er fult utnyttet – ikke mer å hente
- **Kompilatoren** – Java (etter ver 1.3) kompilerer videre til maskinkode og (etter ver 1.6) optimaliserer mye. JIT-kompilering. Ikke mulig å gjøre særlig mer effektiv

⇒ **Konklusjon** Skal vi ha raskere programmer, må vi som programmerere *selv* skrive parallelle løsninger på våre problemer.

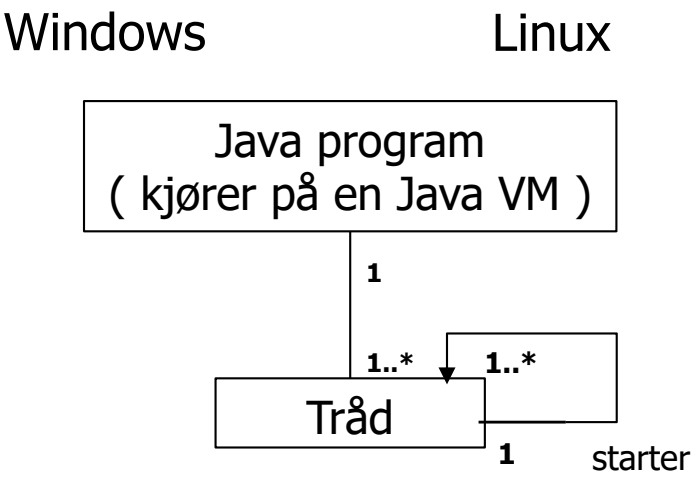
Operativsystemet og tråder

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

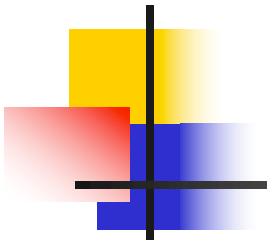


Heldigvis forenkler Java dette

Java forenkler dette ved å velge to nivåer



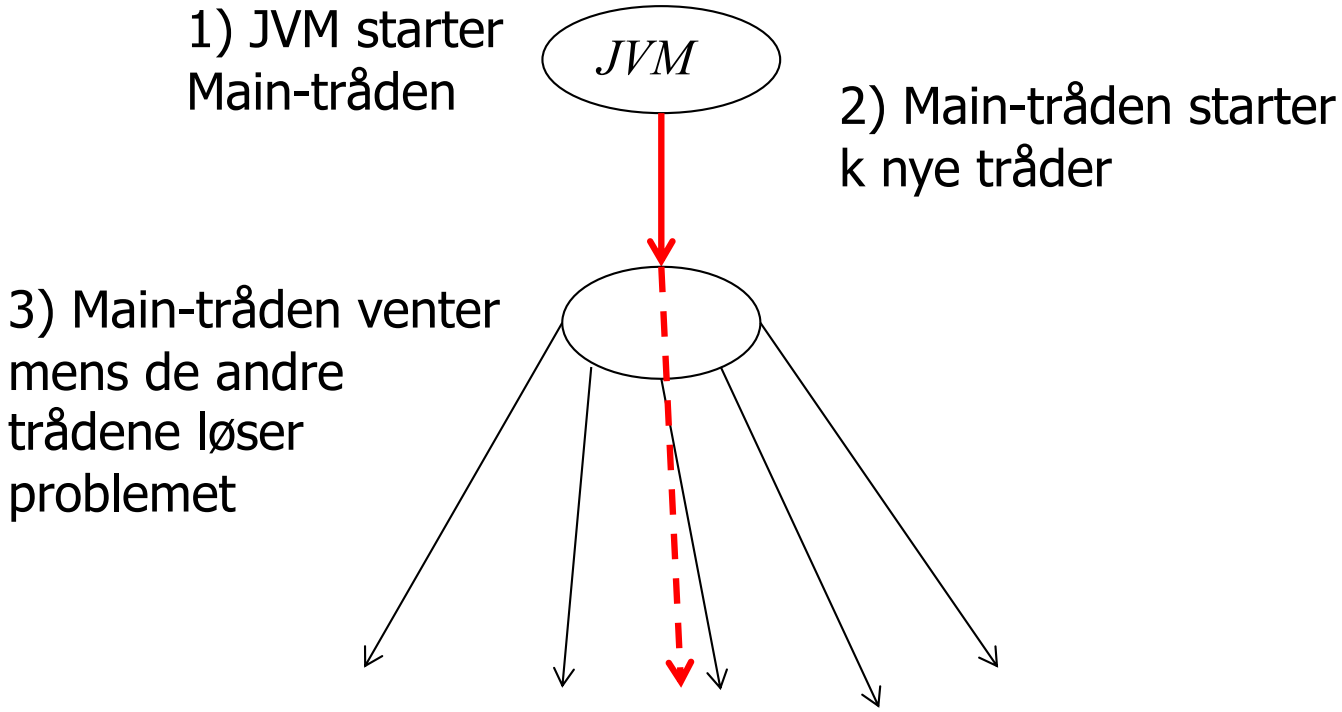
- **Alle trådene i et Java-program deler samme adresserom** (= samme plasser i hovedhukommelsen). Alle trådene kan lese og skrive i de variable (objektene) programmet har og ha adgang til samme kode (metodene i klassene).



Hva er tråder i Java ?

- I alle programmer kjører minst en tråd – main tråden (starter og kjører i public static void main).
- Main-tråden kan starte en eller flere andre, nye tråder.
- Enhver tråd som er startet, kan stoppes midlertidig eller permanent av:
  - Av seg selv ved kall på synkroniseringsobjekter hvor den må vente
  - Den er ferdig med sin kode (i metoden run), terminerer da
- Main-tråden og de nye trådene går i parallell ved at:
  - De kjører enten på hver sin kjerne
  - Hvis vi har flere tråder enn kjerner, vil klokka i maskinen sørge for at trådene av og til avbrytes og en annen tråd får kjøretid på kjernen.
- Vi bruker tråder til å parallellisere programmene våre

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



Tråder i Java er objekter av klassen Thread.

Konstruktør til Thread-klassen

Thread

`public Thread(Runnable target)`

Allocates a new `Thread` object. This constructor has the same effect as `Thread (null, target, gname)`, where `gname` is a newly generated name. Automatically generated names are of the form `"Thread-"+`*n*, where *n* is an integer.

Parameters:

`target` - the object whose `run` method is invoked when this thread is started. If `null`, this classes `run` method does nothing.

- **`Runnable target`** er :
  - En klasse som implementerer grensesnittet 'Runnable'
- Det er en annen måte å starte er tråd hvor vi lager en subklasse av Thread (ikke fullt så fleksibel).

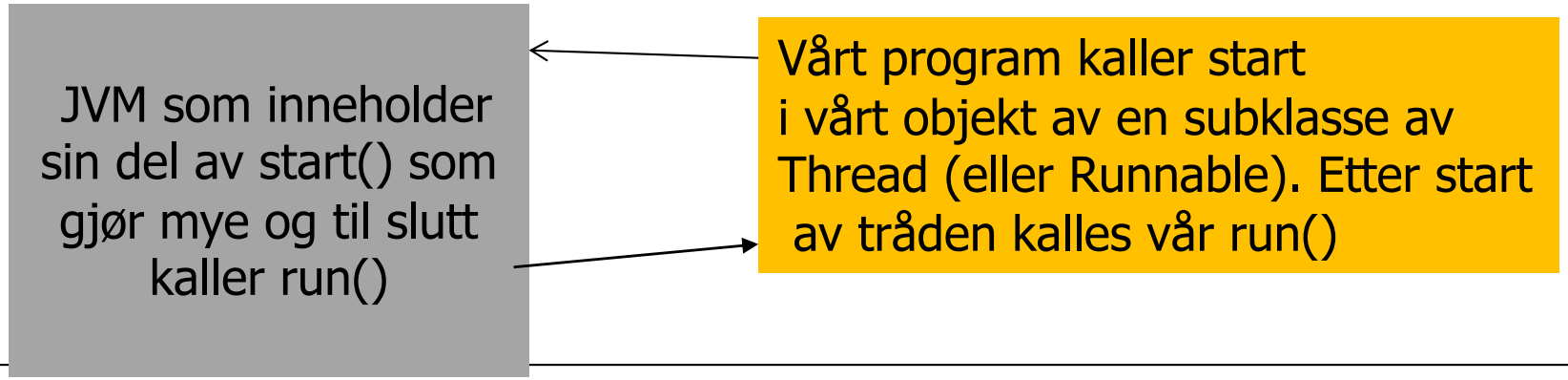
Tråder i Java

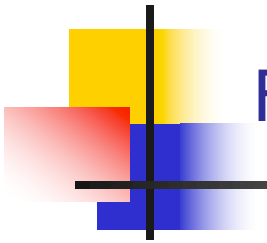
- Er én programflyt, dvs. en serie med instruksjoner som oppfører seg som ett vanlig, sekvensielt program – og kjører på én kjerne
- Det kan godt være (langt) flere tråder enn det er kjerner.
- En tråd er ofte implementert i form av en indre klasse i den klassen som løser problemet vårt (da får trådene greit aksess til **felles data**):

```
import java.util.concurrent.*;
class Problem { int [] fellesData ; // dette er felles, delte data for alle trådene
        public static void main(String [] args) {
                Problem p = new Problem();
                        p.utfoer();
                }
                void utfoer () { Thread  t = new Thread(new Arbeider());
                    t.start();
                 }

          class Arbeider implements Runnable {
                int i,lokalData; // dette er lokale data for hver tråd
                   public void run() {
                          // denne kalles når tråden er startet
                   }
           } // end indre klasse Arbeider
} // end class Problem
```

Tråder i Java

- En tråd er enten subklasse av Thread eller får til sin konstruktør et objekt av en klasse som implementerer Runnable.
- Poenget er at begge måtene inneholder en metode:
  - ’ public void run()’
- Vi kaller metoden start() i klassen Thread . Det sørger for at JVM starter tråden og at ’run()’ i vår klasse deretter kalles.





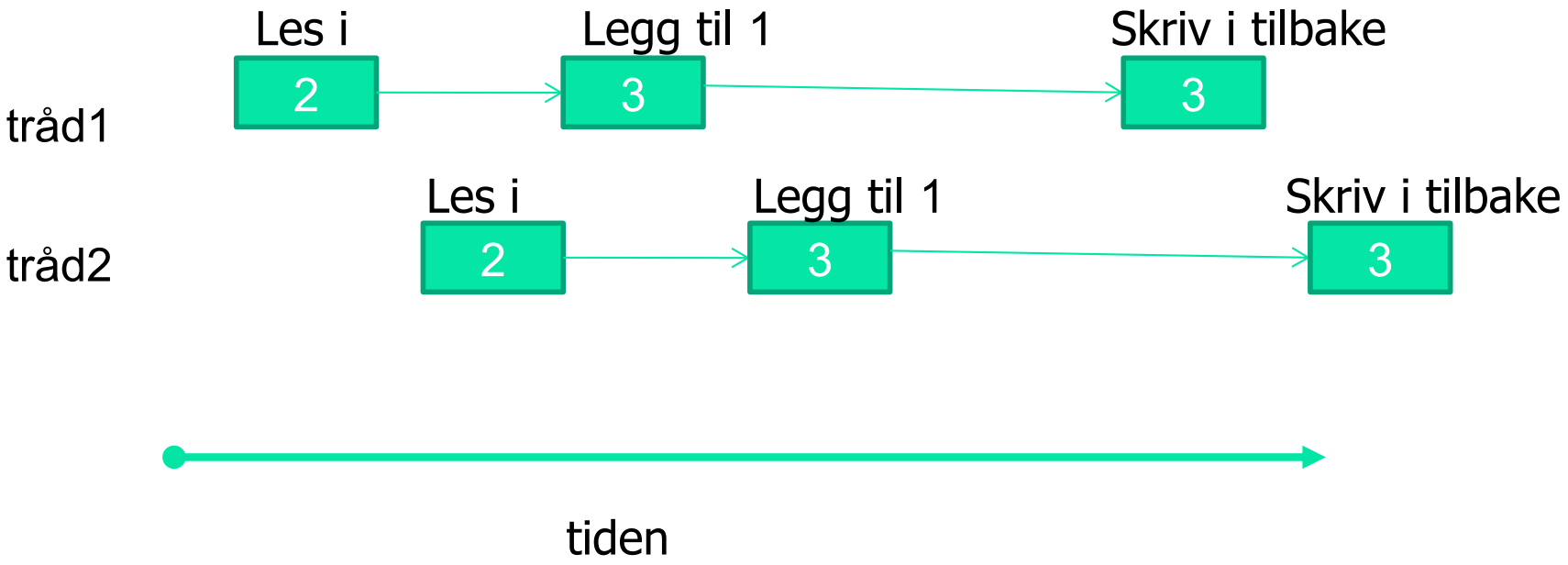
Flere problemer med parallellitet og tråder i Java

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

- Vi må finne på 'skuddsikre' måter å programmere parallelle programmer
  - De er kanskje ikke helt tidsoptimale
  - Men de er lettere å bruke !!
  - Det er vanskelig nok likevel.
- Bare oversiktelige, 'enkle' måter å programmere parallelt er mulig i praksis

1) Ett problem i dag: operasjoner blandes ved samtidige oppdateringer

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



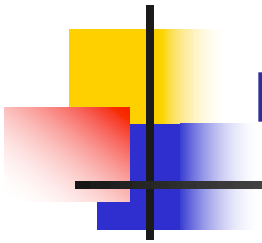
Test på i++; parallell

- Setter i gang **n tråder** (på en 2-kjerner CPU) som alle prøver å øke med 1 en felles variabel int i; 100 000 ganger uten synkronisering;

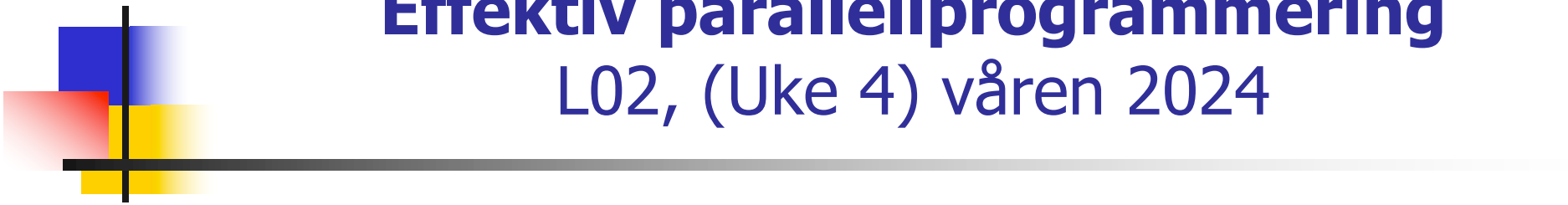
```
for (int j =0; j< 100000; j++) {
        i++;
}
```

- Vi fikk følgende feil - antall og %, (manglende verdier).
  Merk: Resultatene *varierer også mye* mellom hver kjøring :

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



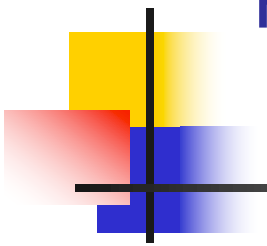
End of L1 IN3030 Lecture 2023-01-18



IN3030/IN4330
Effektiv parallellprogrammering
L02, (Uke 4) våren 2024

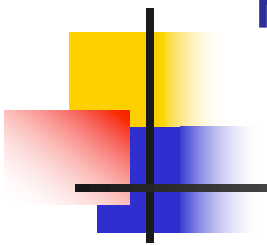
Eric Jul
Professor
Gruppeleder Programmeringsteknologi
Institutt for Informatikk
Universitetet i Oslo
Norge

Resume of first lecture v2024



- Motivation
  - Utilization of Multi-core – by changing sequential programs into parallel programs
  - Multi-core hardware driving this trend
- Purpose
  - Convert sequential program into parallel versions
  - Achieve speedup
- Requirements
  - Correctness – Effective!
  - Efficient – MUST be FASTER – measured by speedup
- Approach
  - Empirical – *the proof of the pudding is in the eating!*
  - *Timings have the ULTIMATIVE SAY!*
  - *(Theory is fine; but practise is essential!)*

Resume of first lecture v2021 (second page)

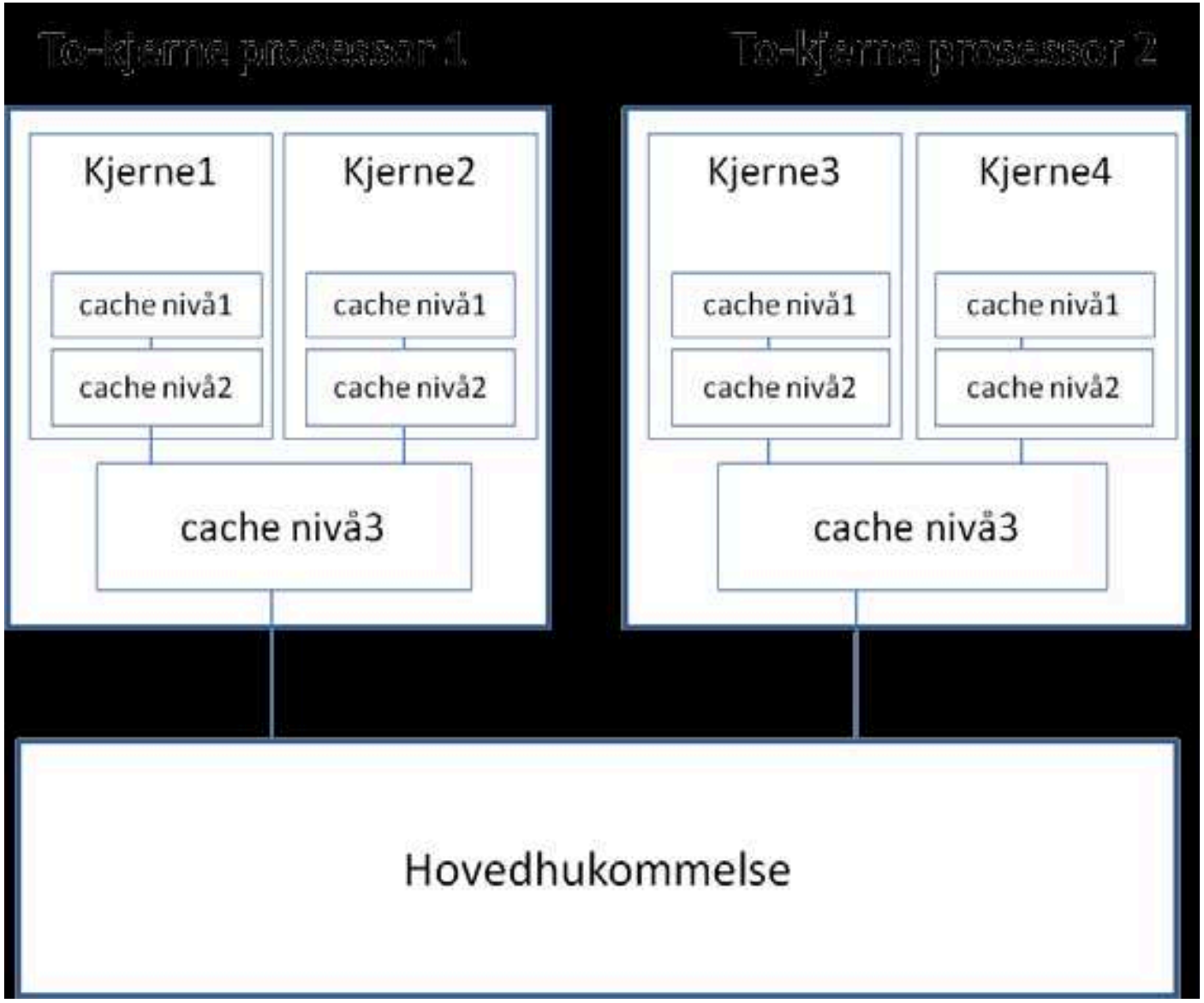


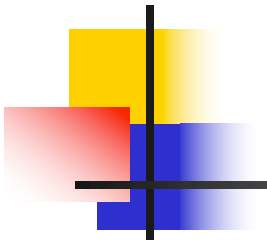
- Central metric: SPEEDUP
  - Speedup: sequential time / parallel time
  - Want speedup > 1
  - Really want speedup = *number of cores*
- How?
  - Parallel threads in Java
  - Must synchronize
- Evaluation?
  - Real-time clock times!!
- Multi-core architecture
- Non-uniform memory access
  - Multi-level caching
- Threads in Java

Maskin 1980 (uten cache)

Figur 19.1 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. Intel 8080: 1 MHz CPU 64 kbyte minne

Maskin ca. 2010 med to dobbeltkjerne CPU-er

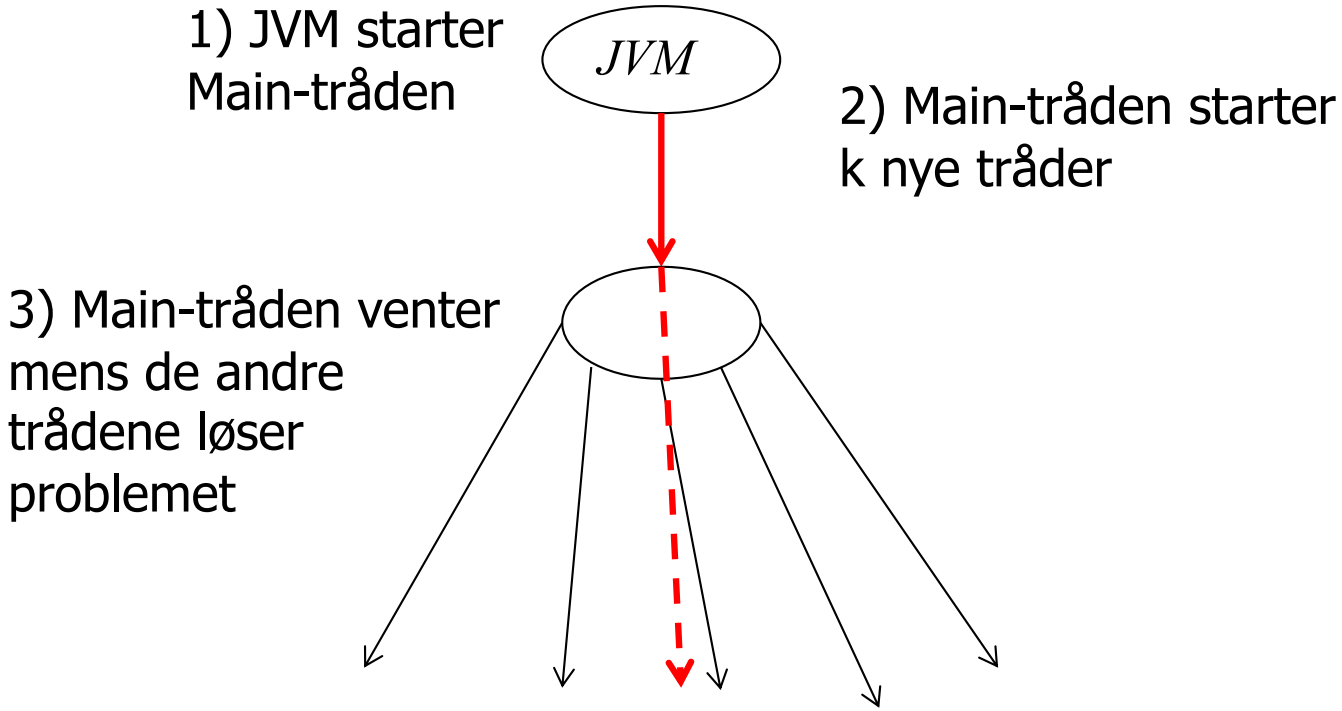




Hva er tråder i Java ?

- I alle programmer kjører minst en tråd – main tråden (starter og kjører i public static void main).
- Main-tråden kan starte en eller flere andre, nye tråder.
- Enhver tråd som er startet, kan stoppes midlertidig eller permanent av:
  - Av seg selv ved kall på synkroniseringsobjekter hvor den må vente
  - Den er ferdig med sin kode (i metoden run), terminerer da
- Main-tråden og de nye trådene går i parallell ved at:
  - De kjører enten på hver sin kjerne
  - Hvis vi har flere tråder enn kjerner, vil klokka i maskinen sørge for at trådene av og til avbrytes og en annen tråd får kjøretid på kjernen. «**Time Slicing**»
- Vi bruker tråder til å parallellisere programmene våre

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



Tråder i Java er objekter av klassen Thread.

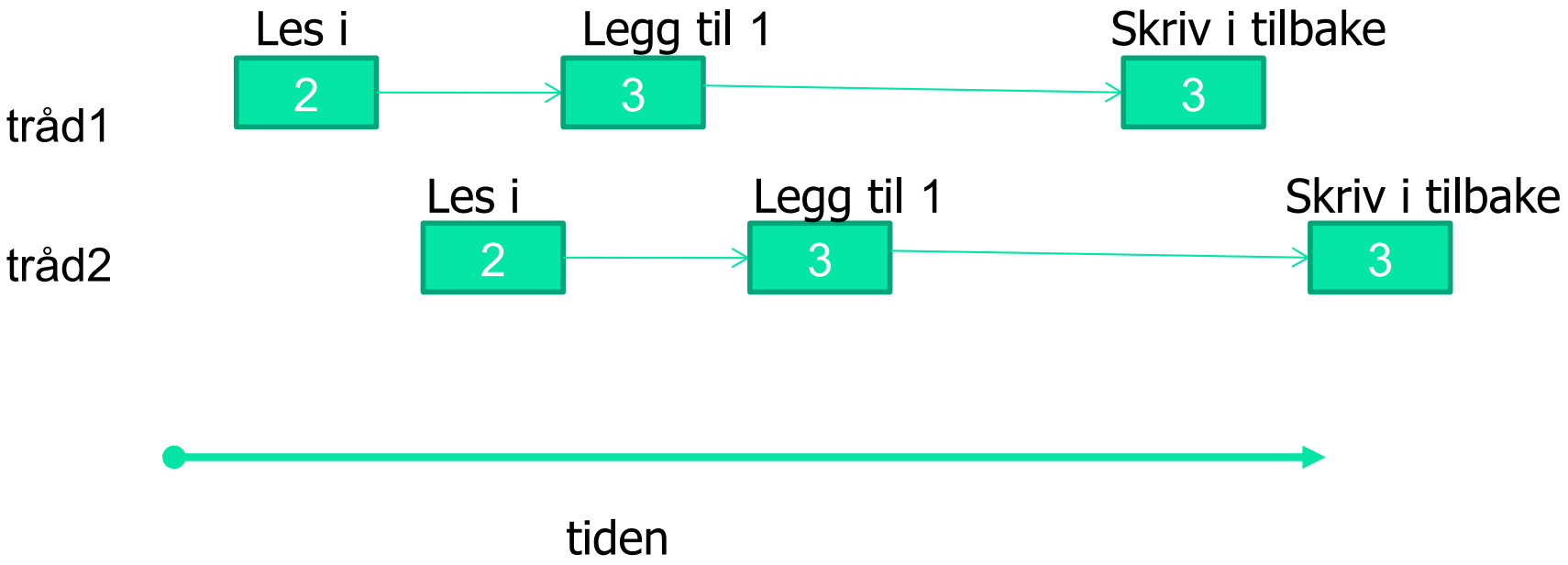
Tråder i Java

- Er én programflyt, dvs. en serie med instruksjoner som oppfører seg som ett vanlig, sekvensielt program – og kjører på én kjerne
- Det kan godt være (langt) flere tråder enn det er kjerner.
- En tråd er ofte implementert i form av en indre klasse i den klassen som løser problemet vårt (da får trådene greit aksess til **felles data**):

```
import java.util.concurrent.*;
class Problem { int [] fellesData ; // dette er felles, delte data for alle trådene
        public static void main(String [] args) {
                Problem p = new Problem();
                        p.utfoer();
                }
                void utfoer () { Thread  t = new Thread(new Arbeider());
                    t.start();
                 }

          class Arbeider implements Runnable {
                int i,lokalData; // dette er lokale data for hver tråd
                   public void run() {
                          // denne kalles når tråden er startet
                   }
           } // end indre klasse Arbeider
} // end class Problem
```

1) Ett problem i dag: operasjoner blandes ved samtidige oppdateringer

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



Test på i++; parallell

- Setter i gang **n tråder** (på en 2-kjerner CPU) som alle prøver å øke med 1 en felles variabel int i; 100 000 ganger uten synkronisering;

```
for (int j =0; j< 100000; j++) {
        i++;
}
```

- Vi fikk følgende feil - antall og %, (manglende verdier).
  Merk: Resultatene *varierer også mye* mellom hver kjøring :

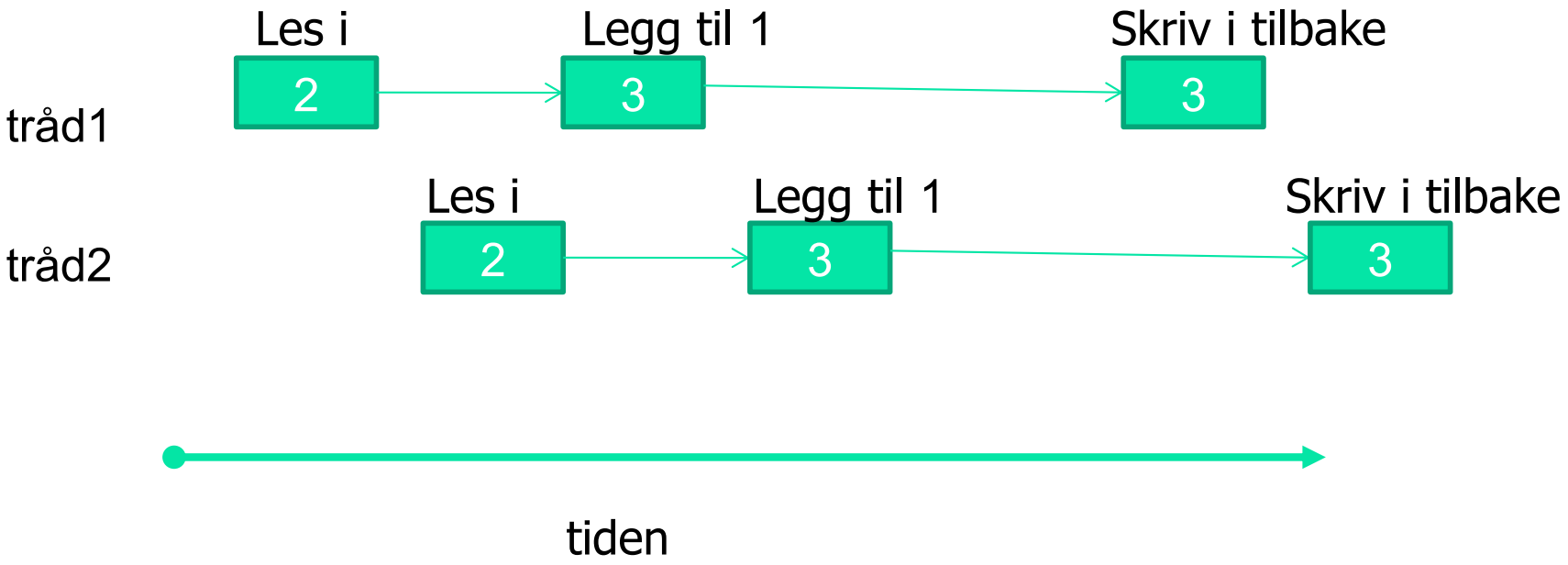
Antall tråder n	**1**	**2**	**20**	**200**	**2000**
Svar 1.gang	100 000	200000	1290279	16940111	170127199
2.gang	100 000	159234	1706068	16459210	164954894
Tap 1.gang	0 %	0%	35,5%	15,3%	14,9%
2. gang	0%	20,4%	14,6%	17,7%	17,5%

Example: Concurrent update of a variable Let's try! (Spoiler: we will fail!)

```
import java.util.concurrent.*;
class Problem { int i ; // dette er felles, delte data for alle trådene
        public static void main(String [] args) {
                Problem p = new Problem();
                        p.utfoer();
                }
                void utfoer () {
                    int j, k;
                    j=10;
                    for (k=0; k< j; k++)  {
                        new Thread(new Arbeider()).start();
                 }

            class Arbeider implements Runnable {
                int i,lokalData; // dette er lokale data for hver tråd
                    public void run() { // denne kalles når tråden er startet
                        i++;
                    }
             } // end indre klasse Arbeider
} // end class Problem
```

1) Ett problem i dag: operasjoner blandes ved samtidige oppdateringer

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



Test på i++; parallell

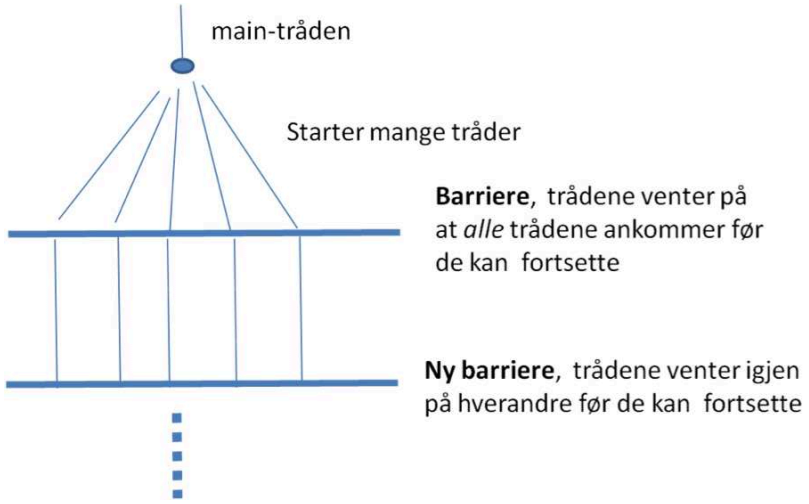
- Setter i gang **n tråder** (på en 2-kjerner CPU) som alle prøver å øke med 1 en felles variabel int i; 100 000 ganger uten synkronisering;

```
for (int j =0; j< 100000; j++) {
        i++;
}
```

- Vi fikk følgende feil - antall og %, (manglende verdier).
  Merk: Resultatene *varierer også mye* mellom hver kjøring :

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

Kommende program bruker CyclicBarrier. Hva gjør den?



- Man lager først ett, felles objekt `b` av klassen CyclicBarrier med et tall: `ant` til konstruktøren = det antall tråder den skal køe opp før alle trådene slippes fri 'samtidig'.
- Tråder (også main-tråden) som vil køe opp på en CyclicBarrier sier await() på den.
- De `ant-1` første trådene som sier await(), blir lagt i en kø.
- Når tråd nummer `ant` sier await() på `b`, blir alle trådene sluppet ut av køen 'samtidig' og fortsetter i sin kode.
- Det sykliske barriere objektet `b` er da med en gang klar til å være kø for nye, `ant` stk. tråder.

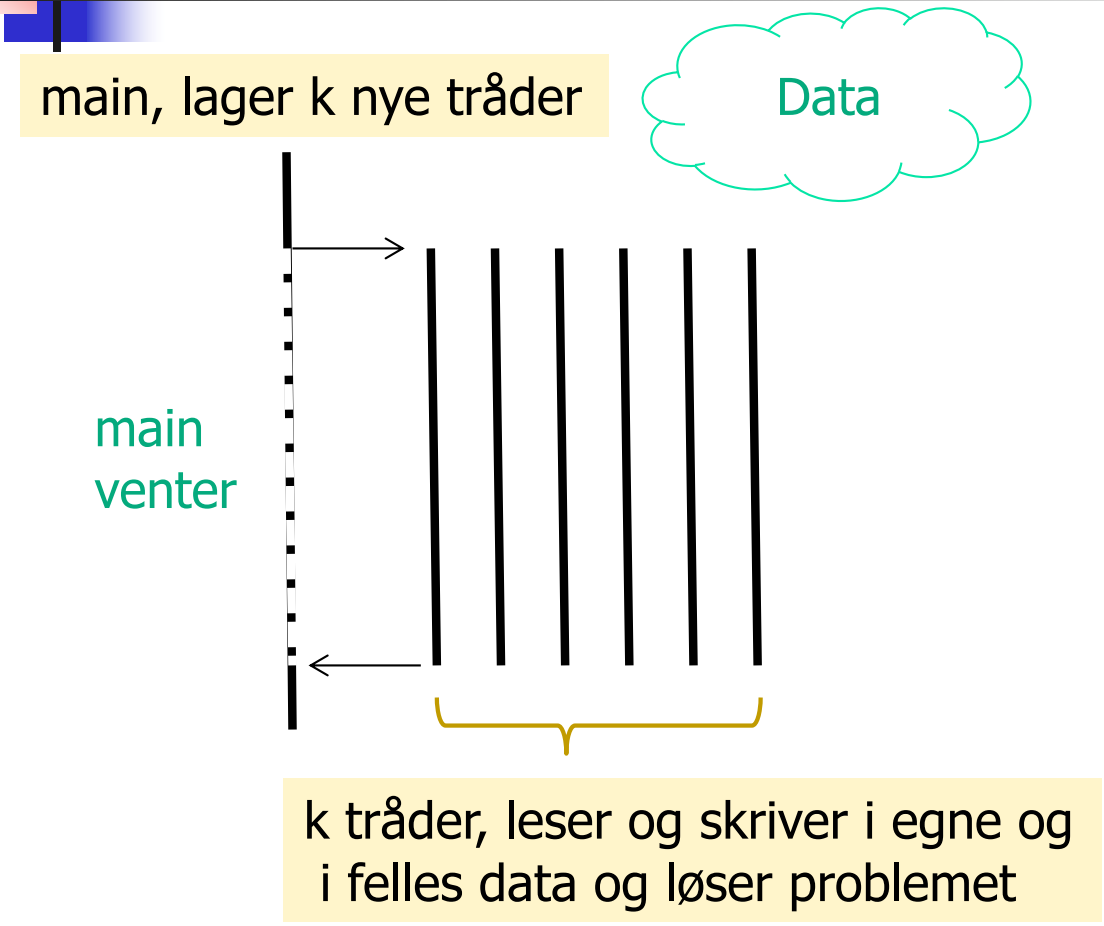
Praktisk: skal nå se på programmet som laget tabellen

```
import java.util.*;
import easyIO.*;
import java.util.concurrent.*;
/** Viser at manglende synkronisering på ett felles objekt gir feil – bare loesning 1) er riktig'*/

public class Parallell {
        int tall;                               // Sum av at 'antTraader' traader teller opp denne
        CyclicBarrier b ;                       // sikrer at alle er ferdige naar vi tar tid og sum
        int antTraader, antGanger ,svar; // Etter summering: riktig svar er:antTraader*antGanger

        // det kommer I alt 4 forsøk på å øke i, bare en av dem er riktig
        //synchronized void inkrTall(){ tall++;}      // 1) –OK fordi synkroniserer på ett objekt (p)
        void inkrTall() { tall++;}                    // 2) - feil

        public static void main (String [] args) {
                if (args.length < 2) {
                        System.out.println("bruk >java Parallell <antTraader> <n= antGanger>");
                }else{  int antKjerner = Runtime.getRuntime().availableProcessors();
                        System.out.println("Maskinen har "+ antKjerner + " prosessorkjerner.");
                        Parallell p =  new Parallell();
                        p.antTraader = Integer.parseInt(args[0]);
                        p.antGanger = Integer.parseInt(args[1]);
                        p.utfor();
        } } // end main
```

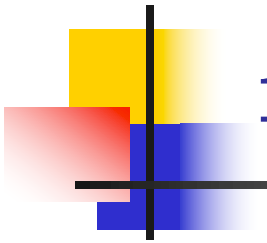
```
void utskrift (double tid) {
            svar = antGanger*antTraader;
            System.out.println("Tid "+antGanger+" kall * "+ antTraader+" Traader ="+
                                                    Format.align(tid,9,1)+ " millisek,");
             System.out.println(" sum:"+ tall +", tap:"+ (svar -tall)+" = "+
              Format.align( ((svar - tall)*100.0 /svar),12,6)+"%");

} // end utskrift

 void  utfor () {   b = new CyclicBarrier(antTraader+1);       //+1, også main
                    long t = System.nanoTime();                 // start klokke

                    for (int j = 0; j< antTraader; j++) {
                          new Thread(new Para(j)).start();
                     }

                   try{   // main thread venter
                          b.await();
                    } catch (Exception e) {return;}
                    double tid = (System.nanoTime()-t)/1000000.0;
                    utskrift(tid);
  } // utfor
```

```
class Para implements Runnable{
        int ind;
        Para(int ind) { this.ind =ind;}

        public void run() {
            for (int j = 0; j< antGanger; j++) {
                inkrTall();
              }
              try {  // wait on all other threads + main
                  b.await();
              } catch (Exception e) {return;}
          } // end run

        //    void inkrTall() { tall++;}                    // 3) Feil - usynkronisert
        //    synchronized void inkrTall(){ tall++;}  // 4) Feil – kallene synkroniserer på
                                                                //      hvert sitt objekt
    } // end class Para
} // END class Parallell
```

Husk: Vanligste oppsett av main-tråden + k tråder

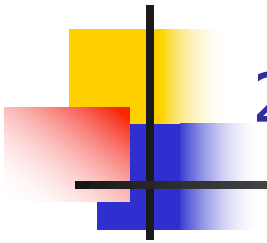


Hver av trådene (main + k nye) er sekvensielle programmer.
Problemet er at de samtidig *ikke kan skrive* på felles data



1) Avslutning med en CyclicBarrier

- En CyclicBarrier (cb= new CyclicBarrier (n+1))
  - Er tenkt som et ventested, en bom/grind for et antall (i dette tilfellet for n+1) tråder - de n 'nye' trådene + main. Alle må vente når de sier sier cb.await() til sistemann ankommer køen, og **da** kan alle fortsette.
  - Trådene kan da være ferdige med en beregning kan selv avslutte med å bli ferdige med sin run() -kode. Main-tråden forsetter, og vet at de andre trådene er ferdige. Main-tråden kan da bruke resultatene fra trådene.
  - Den sykliske barrieren cb er da strakt klar til å køe nye n tråder som sier cb.await() , .. osv
  - cb.await() sies inne i en try-catch blokk



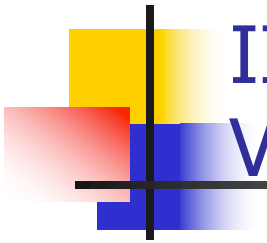
2) Avslutning med en Semaphore

- En Semaphore (sf = new Semaphore(-n+1))
  - Administrerer (i dette tilfellet) –n+1 stk. **tillatelser**.
  - To sentrale primitiver:
    - sf.acquire() – ber om **en** tillatelse. Antall tillatelser i sf blir da 1 mindre hvis antallet er >0. Hvis det ikke er noen ledig tillatelse, må tråden vente i en kø (inne i en try-catch blokk)
    - sf.release() – gir **én** tillatelse tilbake til semaforen sf. Ikke try-catch blokk (Den tillatelsen som gis tilbake behøver ikke vært 'fått' ved hjelp av aquire() ; den er bare et tall).
  - Avlutning med Semaphore sf:
    - Maintråden sier sf.acquire() – og må vente på at det er minst en tillatelse i sf.
    - Alle de n nye trådene sier sf.release() når de terminerer, og når den siste sier sf.release() blir det 1 tillatelse ledig og main fortsetter.
    - Ikke syklisk.

3) Avslutning med join() - enklest

- Logikken er her at i den rutinen hvor alle trådene lages, legges de også inn i en array. Main-tråden legger seg til å vente på den tråden som den har peker til skal terminere selv.
  Venter på alle trådene etter tur at de terminerer:

```
 // main –tråden i konstruktøren
Thread [] t = new Thread[n];
for (int i = 0; i < n; i++) {
   t[i] = new Thread (new Arbeider(..));
   t[i].start();
 }
  ……….
  // main vil vente her til trådene er ferdige
for(int i = 0; i < n; i++) {
       try{ t[i].join();
        }catch (Exception e){return;};
}    ……..
```



II) Mange ulike synkroniserings primitiver Vi skal bare lære noen få !

- java.util.concurrent

Classes

AbstractExecutorService
ArrayBlockingQueue
ConcurrentHashMap
ConcurrentLinkedDeque
ConcurrentLinkedQueue
ConcurrentSkipListMap
ConcurrentSkipListSet
CopyOnWriteArrayList
CopyOnWriteArraySet
CountDownLatch
CyclicBarrier
DelayQueue
Exchanger
ExecutorCompletionService
ecutor
eadPoolExecutor
ThreadPoolExecutor.AbortPolicy
ThreadPoolExecutor.CallerRunsPolicy
ThreadPoolExecutor.DiscardOldestPolicy
ThreadPoolExecutor.DiscardPolicy
Semaphore
SynchronousQueue
ThreadLocalRandom
ThrExecutors
ForkJoinPool
ForkJoinTask
ForkJoinWorkerThread
FutureTask
LinkedBlockingDeque
LinkedBlockingQueue
LinkedTransferQueue
Phaser
PriorityBlockingQueue
RecursiveAction
RecursiveTask
ScheduledThreadPoolEx

Interfaces

BlockingDeque
BlockingQueue
Callable
CompletionService
ConcurrentMap
ConcurrentNavigableMap
Delayed
Executor
ExecutorService
ForkJoinPool.ForkJoinWorkerThreadFacto
ry
ForkJoinPool.ManagedBlocker
Future
RejectedExecutionHandler
RunnableFuture
RunnableScheduledFuture
ScheduledExecutorService
ScheduledFuture
ThreadFactory
TransferQueue

java.util.concurrent.atomic

De har samme virkning (semantikk) som volatile variable (forklares senere), men kan gjøre mer sammensatte operasjoner. Mye raskere enn sychronized methods.

Classes

AtomicBoolean
AtomicInteger
AtomicIntegerArray
AtomicIntegerFieldUpdater
AtomicLong
AtomicLongArray
AtomicLongFieldUpdater
AtomicMarkableReference
AtomicReference
AtomicReferenceArray
AtomicReferenceFieldUpdater
AtomicStampedReference

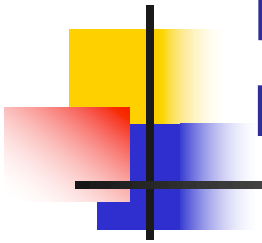
Eksempel på operasjoner i **AtomicIntegerArray:**

int	**get**(int i) Gets the current value at position i.
int	**getAndAdd**(int i, int delta) Atomically adds the given value to the element at index i.
int	**getAndDecrement**(int i) Atomically decrements by one the element at index
void	**set**(int i, int newValue) Sets the element at position i to the given value.

Vi skal bare lære ett fåtall av dette

- Her er de vi skal konsentrere oss om:
  - new Thread – join()
  - synchronized method
  - Semaphore – aquire() og release()
  - CyclicBarrier – await()
  - ExecutorService pool = Executors.newFixedThreadPool(k);
    med Futures - forklares senere
  - AtomicIntegerArray – get(), set(), getAndAdd(),..
  - ReentrantLock (i pakken: **java.util.concurrent.locks**)
  - volatile variable - forklares senere

- Alle de synkroniseringer vi trenger, kan gjøres med disse!
- De fleste andre har sine måter å gjøre det på, men man har neppe tid til å lære seg alle.
- Bedre å bli flink i et lite og tilstrekkelig sett av synkroniseringsprimitiver, enn halvgod i de fleste.



Kan det gå galt når to tråder samtidig skriver i ulike plasser i en array?

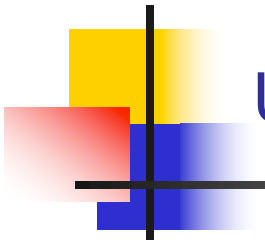
- Et problemet kunne være at når en av tråden lester opp et element i a[i] (int = 4 byte), så er cache-linja 64 byte, så den får med seg flere elementer før og etter a[i].
- Disse 'andre' elementene er det andre tråder som skriver på.
- Vi skriver et testprogram (ParaArray) hvor 10 tråder med indeks : 0,1,2,..,9 som øker hvert sitt element i en array tall[index] 100 000 ganger.

Skriving på nærliggende elementer i en array.

```
class ParaArray{
                int  []tall;
                CyclicBarrier b ;
                int antTraader, antGanger ;
....
class Para implements Runnable{
   int indeks;
  Para(int i) { indeks =i;}
     public void run() {
                for (int j = 0; j< antGanger; j++) {
                      oekTall(indeks);
                 }
                try {  // wait on all other threads + main
                                b.await();
                } catch (Exception e) {return;}
     } // end run
      void oekTall(int i) { tall[i]++; }
  }
} // end ParaArray
```

- Cache-linja er nå 64 byte (og en int er 4 byte)
- Går det greit med at flere tråder (indeks=0,1,…,k-1) skriver på a[tråd.indeks] mange ganger i parallell?
- Tester: Vi lageret program som gjør det :

```
>java ParaArray 10 100000000
Maskinen har 8 prosessorkjerner.
Tid 100000000 kall * 10 Traader =
0.032600 sek,
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
```



Ukeoppgave L02

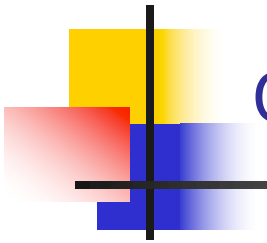
- Find the largest number in a large array

Oblig plan 2024

IN3030/IN4330 Oblig plan 2024:

O1	L2	2 weeks	25/1 – 7/2
O2	L4	2 weeks	8/2 – 21/2
O3	L7	4 weeks	22/2 – 20/3
[Easter Break]			
O4	L11	3 weeks	4/4 – 24/4
O5	L14	1 week	25/4 – 2/5

Written exam 27/5 – 2024



Oblig plan 2024

Lectures L1 – L10 are given in calendar weeks 3 thru 12.

(Add 2 to the lecture number to get the week number.)

No lecture 28/3 (Easter break).

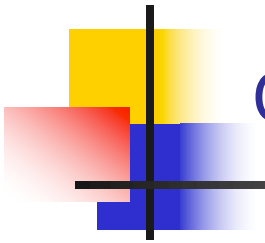
Lectures after Easter and before Kristi himmelfartsdag start with L11 in week 14.

(After Easter, add 4 to the lecture number to get the week number.)

Lectures after Kristi himmelfartsdag start with L17 in week 20.

Obligs are delivered in Devilry no later than 23:59:00 on the deadline date.

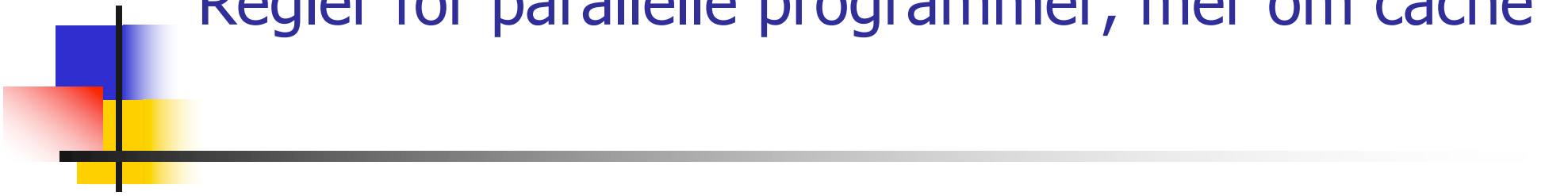
NOTE: the deadline is ONE MINUTE before midnight!



Oblig 1

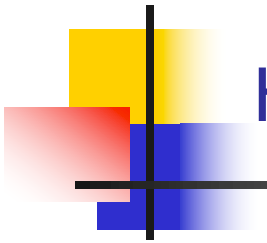
- Oblig 1 presenteres

A
B
O(N²)



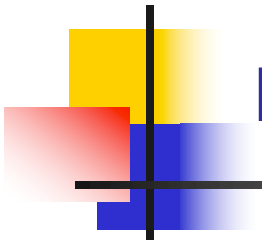
IN3030, L03, våren 2024
Regler for parallelle programmer, mer om cache

Eric Jul
Programming Technology Group
Programming Section
Department of Informatics
University of Oslo



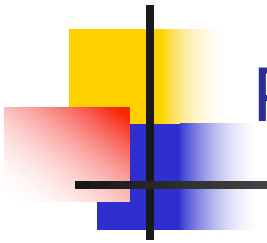
Hva har vi sett på i L02

- Tråde i Java
- Én stygg feil vi kan gjøre: Samtidig oppdatering (skriving) på delte data (eks: to tråde skriver på samme variabel: `i++`)
- Synkronisering er vanskelig!
- Samtidig skriving på en variabel ***må synkroniseres***:
  - Alle objekter kan nyttes som en synkroniseringsvariabel, og da kan vi bruke enten en `synchronized` metode (treigt) for å gjøre det,
  - eller objekter av spesielle klasser som:
    - **CyclicBarrier**
    - **Semaphore**
    - **AtomicInteger**
    - En av MANGE andre mulige



Hva har vi sett på i L02 (fortsatt)

- I) Tre måter å avslutte tråder vi har startet.
  - join(), Semaphore og CyclicBarrier.
- II) Ulike synkroniseringsprimitiver
  - Vi skal bare lærte oss noen få - ett tilstrekkelig sett

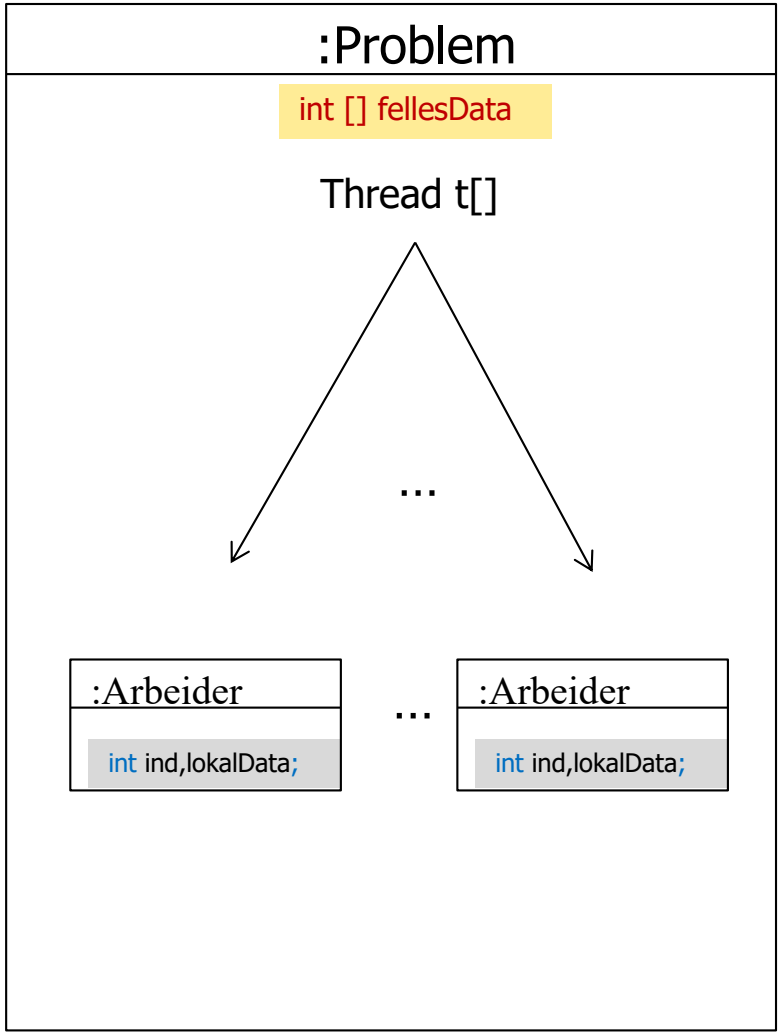


Plan for L03

- Modell(er) for hvordan vi programmerer
- Viktige regler om lesing og skriving på felles data.
- Synlighetsproblemet – «memory is not memory»
  - Hvilke verdier ser ulike tråder som leser variable som en annen tråd skriver på?
- Hva er Cache – intro
- Effekten på eksekveringstid av cache
- Matrixmultiplisering.

Modell for parallelle programmer

```
import java.util.concurrent.*;
class Problem { int [] fellesData , //  felles data
      public static void main(String [] args) {
            Problem p = new Problem();
            p.utfoer(12);
      }
      void utfoer (int antT) {
            … // utfør sekvensiell kode med tidtaking
          Thread [] t = new Thread [antT];

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

          for (int i =0; i< antT; i++) t[i].join();
      }
      //.. metoder for sekvensielt  & parallell

      class Arbeider implements  Runnable {
        int ind, lokaleData; // lokale data
        //.. metoder for parallelt problem
         Arbeider (int in) {ind = in;)
             public void run(int ind) {
                    // kalles når tråden er startet
             } // end run
      } // end indre klasse Arbeider
} // end class Problem
```



Dette gjør at programmet blir mer effektivt

- I et virkelig brukerprogram vil vi ha testen:

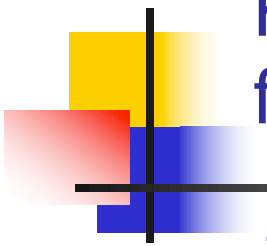
```
if (n < LIMIT ){ løsXSekvensielt(param)
 } else {
```

  - I IN3030 skal vi ikke ha denne testen fordi vi er mer interessert i å se når en parallell løsning er langsommere **og** når den er raskere.
  - Vi kan si vi bestemmer LIMIT for ulike problemer:
    - For FinnMax er LIMIT ca = 1 mill.
    - For andre problemer er LIMIT langt lavere, f.eks senere vil vi se: 40 000 for sortering og 150 for matrise-multiplikasjon.
  - I sekvensielle programmer, som sortering gjøres også en slik test og man bruker 'innstikkSortering', hvis n < 32.
    - Arrays.sort() – som er Quicksort, bruker LIMIT = 47

Sekvensielt for små problemer: Slik skal virkelige programmer se ut (**ikke** i kurset)

```
class ProblemX{
    <felles data>

     <type> løsX(...) {
     if (n < LIMIT ){ løsXSekvensielt(param)
     } else {
        <start tråder. De løser hver sin del av
           problemet og tilsammen hele problemet>;
        <vent på at trådene er ferdige>;
        <hent svaret i felles data og returner>
    } // end løsX

    class ArbeiderTråd extends Thread{
       <Lokale data for en tråd>;
       ArbeiderTråd (param) {
          <lokale data = param>;
       public static void run() {
          <her løses denne trådens del av problemet
            i ett eller flere steg med synkronisering
             mellom hvert steg når vi bruker parallell kode>;
        } // end run
    } // end ArbeiderTråd
} // end class ProblemX
```

Konvensjoner som gjør at programmet ikke blir forkert - I



1. Alle arbeider-tråder har en lokal variabel: int indeks (=0,1,2,..,antTråder-1)
2. Vi antar at brukere som kaller løsX-metoden, kjører på main-tråden.
   - Dårlig idé er å la en tråd i et 'annet' parallelt problem kalle på en parallell løsning som løsX. Blir fort for mange tråder og treigt. (dvs. ikke parallelliser inne i en allerede parallellisert kode)
3. Vi lar trådene løse **hele** problemet.
4. Main-tråden bare initierer felles data og starter hver tråd - før den legger seg og venter på at trådene blir ferdige.
   Da er hele problemet løst og ligger i felles data.
5. Problemet som arbeider-trådene skal løse, kan bestå av ett eller flere steg. Vi synkroniserer da alle arbeider-trådene med en CyclicBarrier mellom hvert av stegene.

(fortsettes neste foil)

Konvensjoner som gjør at programmet ikke blir forkert - II

6. Må ett av stegene (f.eks det siste) være sekvensielt, lar vi bare tråd med indeks == 0 gjøre det:

```
if (indeks == 0) {
   < Gjør det sekvensielle steget før neste synkronisering>;
}
```

   De andre arbeider-trådene går her bare rett til neste barrier-synkronisering (eller avslutning).
7. Hvis behovet for å ha en enkel seksvensiell kode oppstår midt under beregningene, kan alle trådene regne ut samme svar uten synkronisering seg imellom (skjer f.eks i parallell Quicksort)
8. Arbeider-trådene initierer bare lokale variable i sin konstruktør.
   8. Husk at objektet ikke er ferdig når konstruktøren kjører. Mye galt kan skje (se boka JCiP kap 3.2) hvis andre tråder får en peker til objektet før det er ferdig.
   9. Ingen kall til andre metoder i konstruktøren.
   10. Kan forebygges: Lad ALLE tråde passer en cyclisk barrier etter initialisering.
9. All handling i arbeider-trådene skjer i run() og i metoder kalt fra run().

VIGTIKT Konvensjon: Tre avgjørende prinsipper for lesing og skriving på felles data.

- Før (og etter) synkronisering på felles synkroniserings-objekt gjelder :
  A. Hvis ingen tråder skriver på en felles variabel, kan alle tråder lese denne.
  B. To tråder må aldri skrive samtidig på en felles variabel (eks. `i++` går galt)
  C. Hvis bare én tråd skriver på en variabel må også bare denne tråden lese denne variabelen før synkronisering – ingen andre tråder må lese den før synkronisering.

Muligens ikke helt tidsoptimalt, men enkel å følge – gjør det mulig å skrive parallelle programmer.

Har vist pkt. A og B, skal nå vise pkt. C

Synlighetsproblemet (hvilke verdier ser ulike tråder som leser variable som en annen tråd skriver på)

- Lage et testprogram som har:
  - To **felles** variable. int a,b;
  - To klasser, arbeider-tråder SkrivA og SkrivB,
  - en som øker a & en øker b (100 000 ganger) og skriver ned verdiene av a og b i hver sin *lokale* arrayer: mA[] og mB[] (antså to sett av disse).

```
for (int j = 0; j<antGanger; j++) {
        a++;
        mA[j] =a;
        mB[j] =b;
 }
```

  - og en annen tråd som tilsvarende øker b

```
for (int j = 0; j<antGanger; j++) {
        b++;
        mA[j] =a;
        mB[j] =b;
 }
```

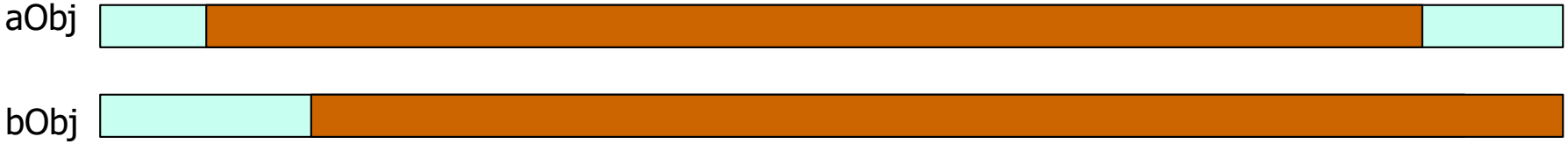
Ytre klasse SamLes med to indre klasser SkrivA og SkrivB

```
public class SamLes{
      int  a=0, b=0;                 // Felles variable a , b
      CyclicBarrier sync, vent ; // begge starter 'samtidig'
      int  antGanger ;
      SkrivA aObj;
      SkrivB bObj;


void utskrift() {  ...   };

void  utfor () {

              vent = new CyclicBarrier((int)antTraader+1);
              sync = new CyclicBarrier((int)antTraader);

              (aObj =  new SkrivA()).start();
              (bObj =  new SkrivB()).start();

               try{
                  // main venter på aObj og bObj ferdige
                     vent.await();
              } catch (Exception e) {return;}
        utskrift();
 } // utfor
```

```
class SkrivA extends Thread{
      int [] mB = new int[antGanger],
             mA = new int[antGanger];
      public void run() {
             try {  // wait on the other thread
                          sync.await();
             } catch (Exception e) {return;}

             for (int j = 0; j<antGanger; j++) {
                   a++;
                    mA[j] =a;
                    mB[j] =b;
              }
              try {  // wait on the other thread + main
                          vent.await();
              } catch (Exception e) {return;}
    } // end run A
} // end class Para

class SkrivB extends Thread{
      int [] mB = new int[antGanger],
             mA = new int[antGanger];
      public void run() {
             try {  // wait on the other thread
                          sync.await();
             } catch (Exception e) {return;}

             for (int j = 0; j<antGanger; j++) {
                   b++;
                    mA[j] =a;
                    mB[j] =b;
              }
              try {  // wait on the other thread + main
                          vent.await();
              } catch (Exception e) {return;}
    } // end run B
} // end class SamLes
```

Hva tester vi her ?

- Ser på om de to trådene (aObj og bObj) alltid ser oppdaterte verdier av den andre variabelen (ser f.eks objA at b er helt oppdatert) ?
- Utskrift vanskelig: Selv om starter nesten likt, må de synkroniseres på utskrift (og ikke skrive ut alt!):



Leter utover i de to arrayene: mA[] i de to objektene aObj og bObj og starter utskrift ut når a-verdiene i er like og > 0, og skriver da ut de 10 neste verdiene av a og b i aObj og bObj

Resultater: Er det feil her (gamle verdier, e.l)

= like verdier i a og b

SkrivA			SkrivB		
a.mA[722]= 723	a.mB[722]= 1458	|	b.mA[1457]= 723	b.mB[1457]= 1458
a.mA[723]= 724	a.mB[723]= 1458	|	b.mA[1458]= 724	b.mB[1458]= 1459
a.mA[724]= 725	a.mB[724]= 1460	|	b.mA[1459]= 725	b.mB[1459]= 1460
a.mA[725]= 726	a.mB[725]= 1460	|	b.mA[1460]= 726	b.mB[1460]= 1461
a.mA[726]= 727	a.mB[726]= 1461	|	b.mA[1461]= 727	b.mB[1461]= 1462
a.mA[727]= 728	a.mB[727]= 1463	|	b.mA[1462]= 728	b.mB[1462]= 1463

- NB. SkrivA (=aObj) har a-ene riktige (oppdatert) og SkrivB har b-ene oppdatert
- For eksempel. første og andre linje tvilsomme sammen:
  - A har akkurat økt a fra 722 til 723, og ser b som 1458, MEN
  - B har akkurat økt b fra 1458 til 1459, og ser a som 723
  - I neste linje ser A fortsatt b som 1458, men a i aObj er lik 724
- Dette kan bare forklares ved at A og B operasjonene blandes
- Vi vet ikke når b for aObj har en verdi (eks 1458 eller1460) hvilken a-verdi som hører til disse.
- Og noen verdier for b (1459, 1462) sees aldri av aObj, men bObj ser dem.
- **Konklusjon:** Ulike tråder kan se ulike verdier for felles variable og man vet ikke når en tråd har oppdatert (skrevet) på 'sine' variable og dette er synlig i annen tråd.

WTF: What Terrible Fiasco!

= like verdier i a og b

SkrivA			SkrivB		
a.mA[722]= 723	a.mB[722]= 1458	|	b.mA[1457]= 723	b.mB[1457]= 1458
a.mA[723]= 724	a.mB[723]= 1458	|	b.mA[1458]= 724	b.mB[1458]= 1459
a.mA[724]= 725	a.mB[724]= 1460	|	b.mA[1459]= 725	b.mB[1459]= 1460
a.mA[725]= 726	a.mB[725]= 1460	|	b.mA[1460]= 726	b.mB[1460]= 1461
a.mA[726]= 727	a.mB[726]= 1461	|	b.mA[1461]= 727	b.mB[1461]= 1462
a.mA[727]= 728	a.mB[727]= 1463	|	b.mA[1462]= 728	b.mB[1462]= 1463

DISASTER: Think CAREFULLY about what happened here!

MAYDAY, MAYDAY, MAYDAY:

*You MAY think that a core can »write to memory» BUT in reality there is no guarentee that others can see the new value right away!! It CAN take time – and in the meanwhile two cores can **READ DIFFERENT VALUES** from the **SAME** variabel!!!*

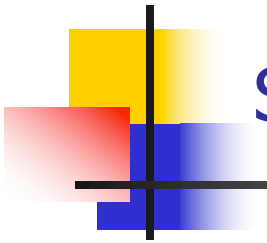
FIX: the two thread must synchronize first! ☺ Problem solved!

4) Skriving på **nærliggende** elementer i en array, kode.

```
class ParaArray{
                int  []tall;
                CyclicBarrier b ;
                int antTraader, antGanger ;
....
class Para implements Runnable{
   int indeks;
  Para(int i) { indeks =i;}
     public void run() {
                for (int j = 0; j< antGanger; j++) {
                      oekTall(indeks);
                 }
                try {  // wait on all other threads + main
                     b.await();
                } catch (Exception e) {return;}
     } // end run
      void oekTall(int i) { tall[i]++; }
  }
} // end ParaArray
```

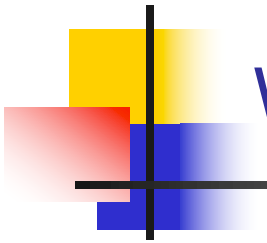
- Cache-linja er nå 64 byte (og en int er 4 byte)
- Går det greit med at flere tråder (indeks=0,1,…,k-1) skriver på a[tråd.indeks] mange ganger i parallell?
- Tester: Vi lageret program som gjør det :

```
>java ParaArray 10 100000000
Maskinen har 8 prosessorkjerner.
Tid 100000000 kall * 10 Traader =
0.032600 sek,
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
```



SAVED BY THE CAVALRY

SAVED!: We can write to separate elements in an array WITHOUT synchronization!!



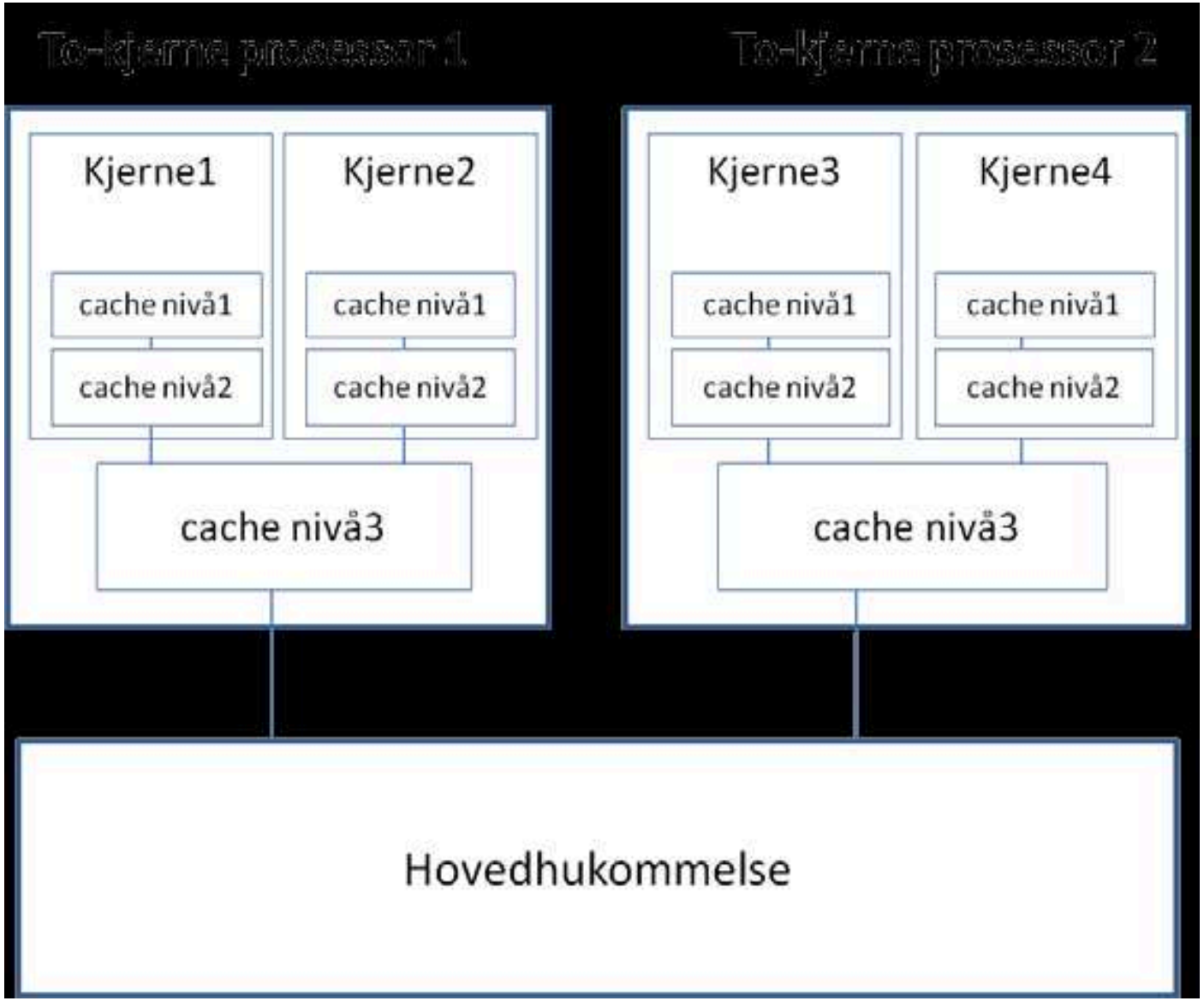
WHAT is a cache?

- Principle of a cache
- Speed of memory vs size
- Second reason: Speed of electricity:
  - Electricity travels about 6-7 cm per machine cycle on a 3 GHz CPU

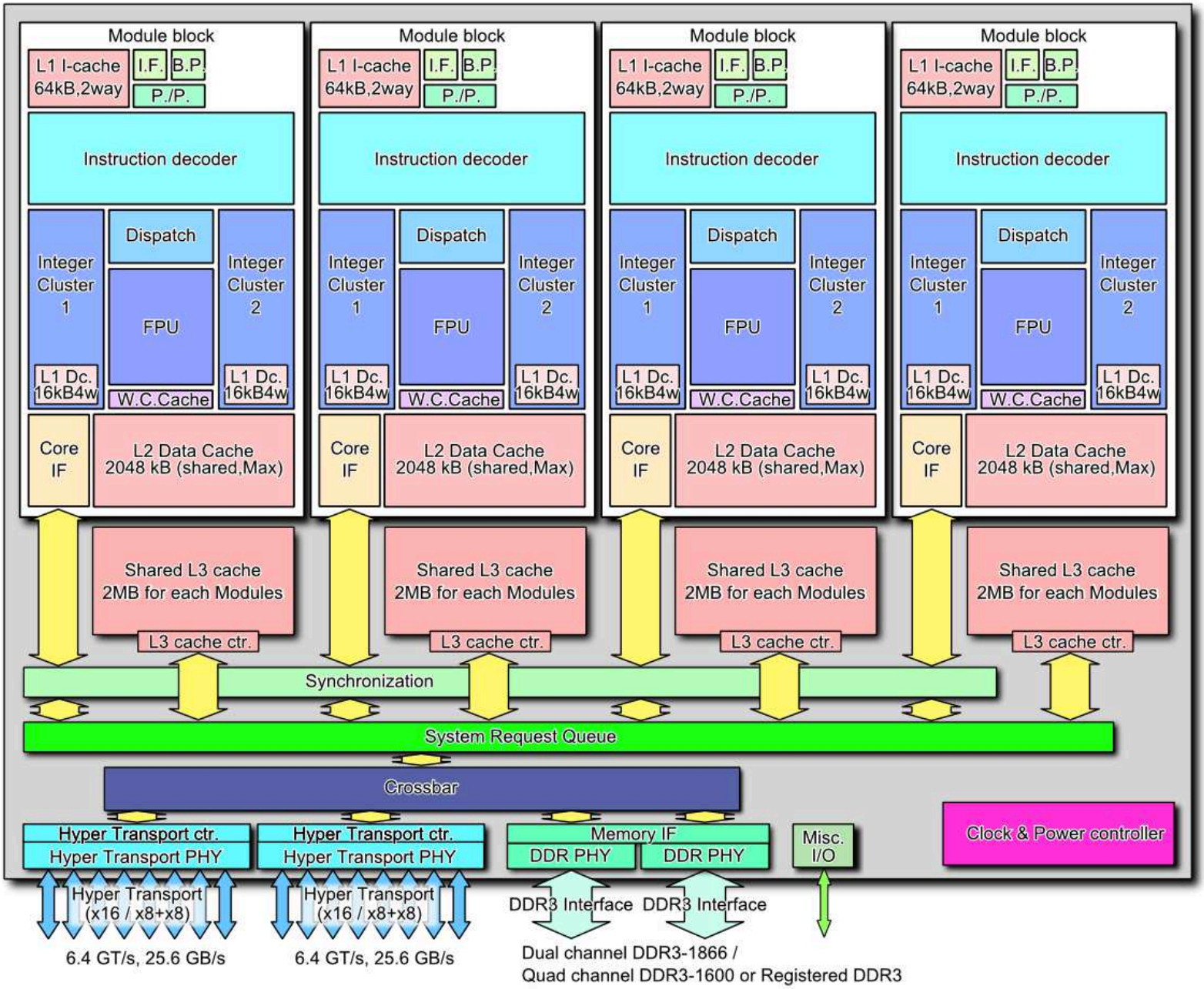
Maskin 1980 (uten cache)

Figur 19.1 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. Intel 8080: 1 MHz CPU

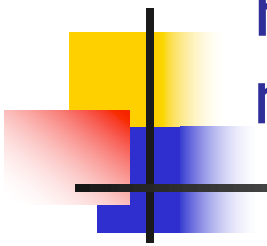
Maskin ca. 2010 med to dobbeltkjerne CPU-er



Hukommelses-systemet i en 4 kjerne CPU – mange lag og flere ulike beregningsmoduler i hver kjerne.:



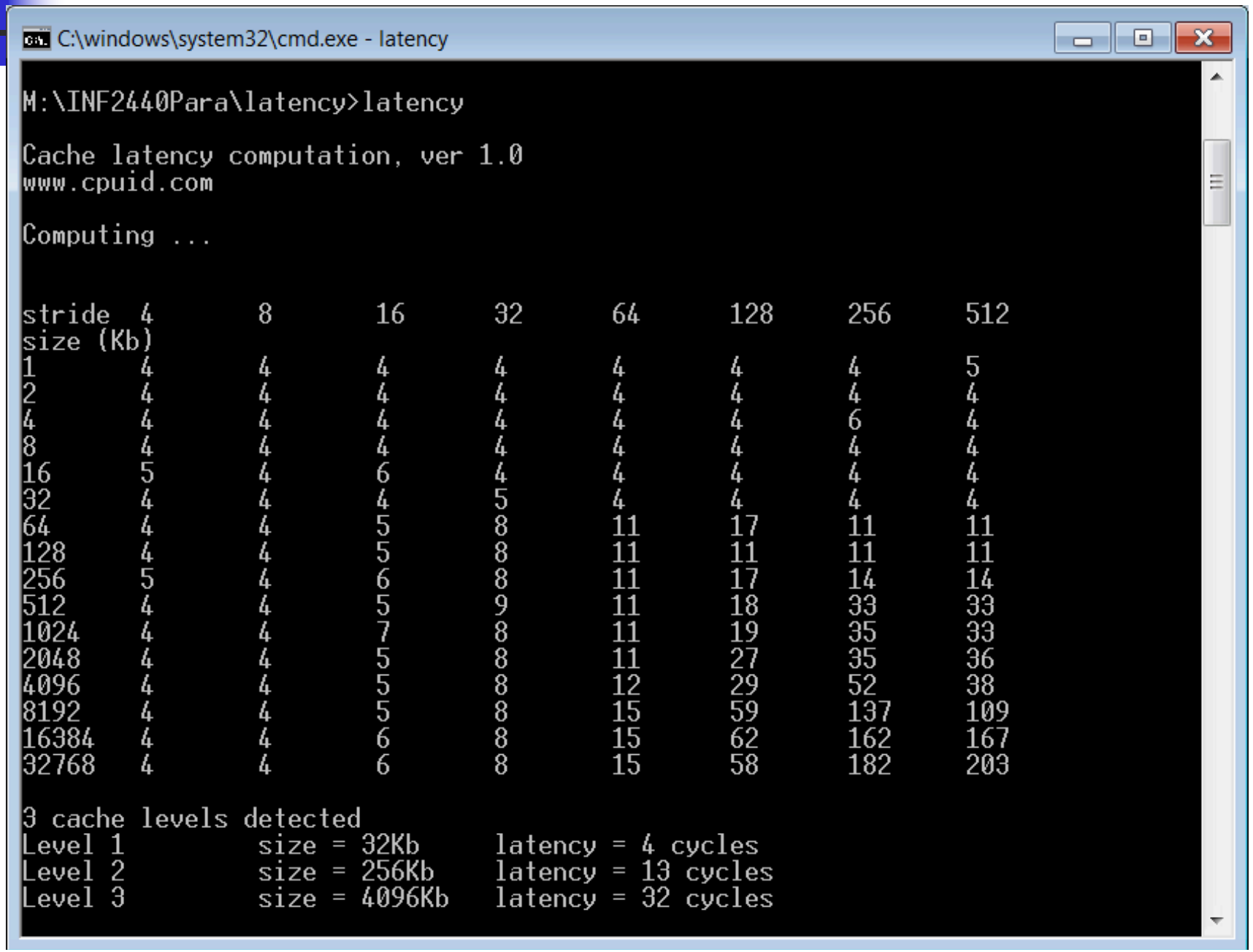
Hvordan tar vi hensyn til cache-systemet for å få raskere programmer?



- Vi ser bare på data-cachene (lite å hente på instruksjonene)
- Viktig å vite er at hver gang vi skal hente data i hovedlageret , får vi en cach-linje = 64 byte = f.eks 8 heltall (int)
- Det er svært begrenset plass i cachene, og en cach-linje som ikke har vært brukt på 'lenge' vil bli 'kastet ut'(overskrevet av en annen, nyere) cache-linje
- Slik er raskest:
  - Jobber på få data (korte deler av en array) 'lenge' av gangen – ikke hoppe rundt.
  - Helst gå forlengs eller baklengs gjennom data (arrayene) (i, i+1,.. eller: i, i-1,..)

Vi må lage slike cache-vennlige programmer !

Test av forsinkelse i data-cachene og hovedhukommelsen - latency.exe (fra CPUZ)



Oppsummering – ideen om at vi har *uniform* aksesstid i hukommelsen er helt galt

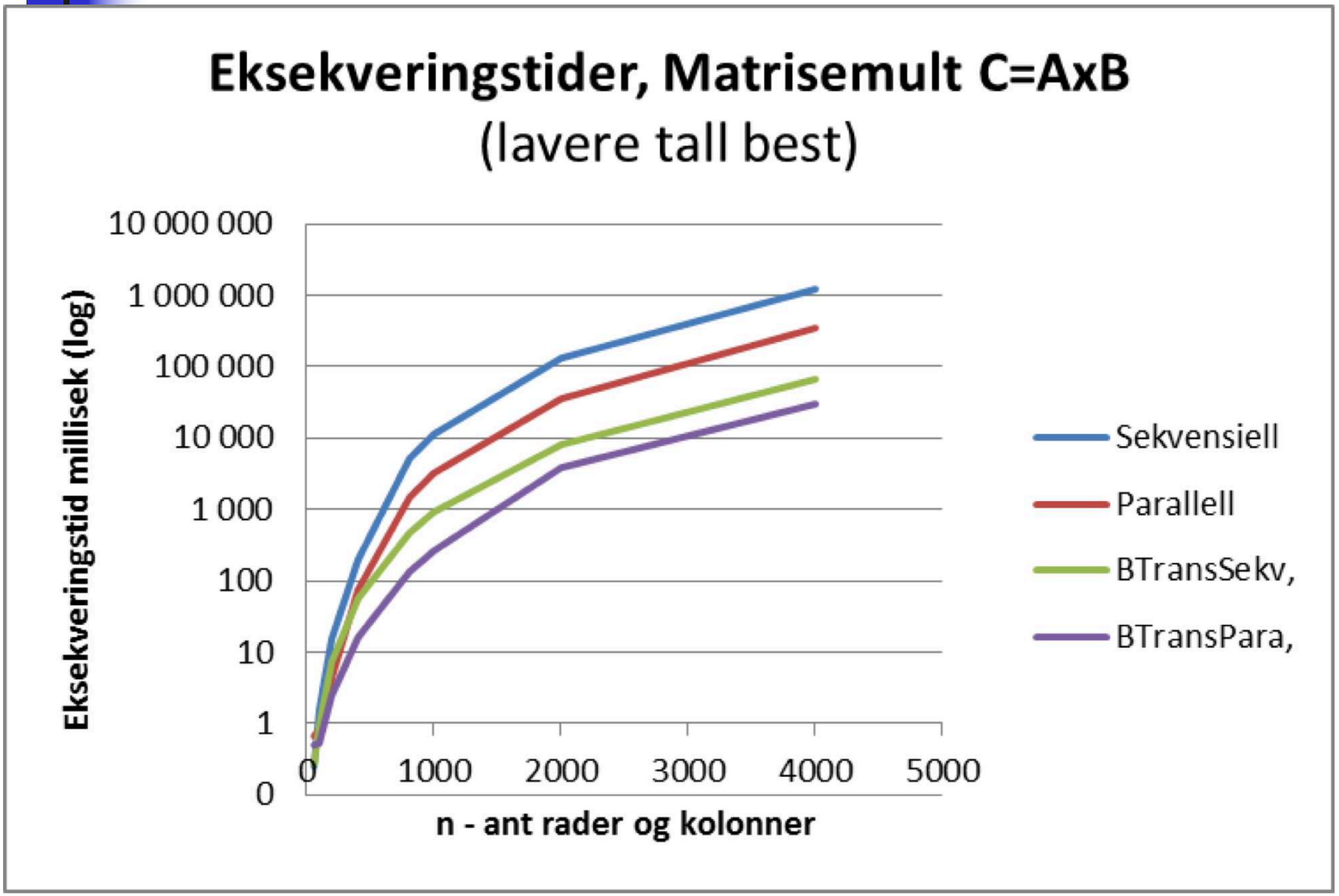
- Hukommelses-systemet i en multicore CPU ,Intel Core i5-459 3.3 GHz, – mange lag (typisk aksesstid i instruksjonssykler):

1. Registre i kjernen (1) – 8/32 registre
2. L1 cache (3-4) – 32 Kb
3. L2 cache (13) – 256 kb
4. L3 cache (32) – 8Mb
5. Hovedhukommelsen (virtuell hukommelse) (ca. 200) – 8-64 GB
6. Disken (15 000 000 roterende) = 5 ms – 1000 GB – 1-5 TB
   FlashDisk (ca 2 000 000 les, ca. 10 000 000 skriv) = ca. 1 ms

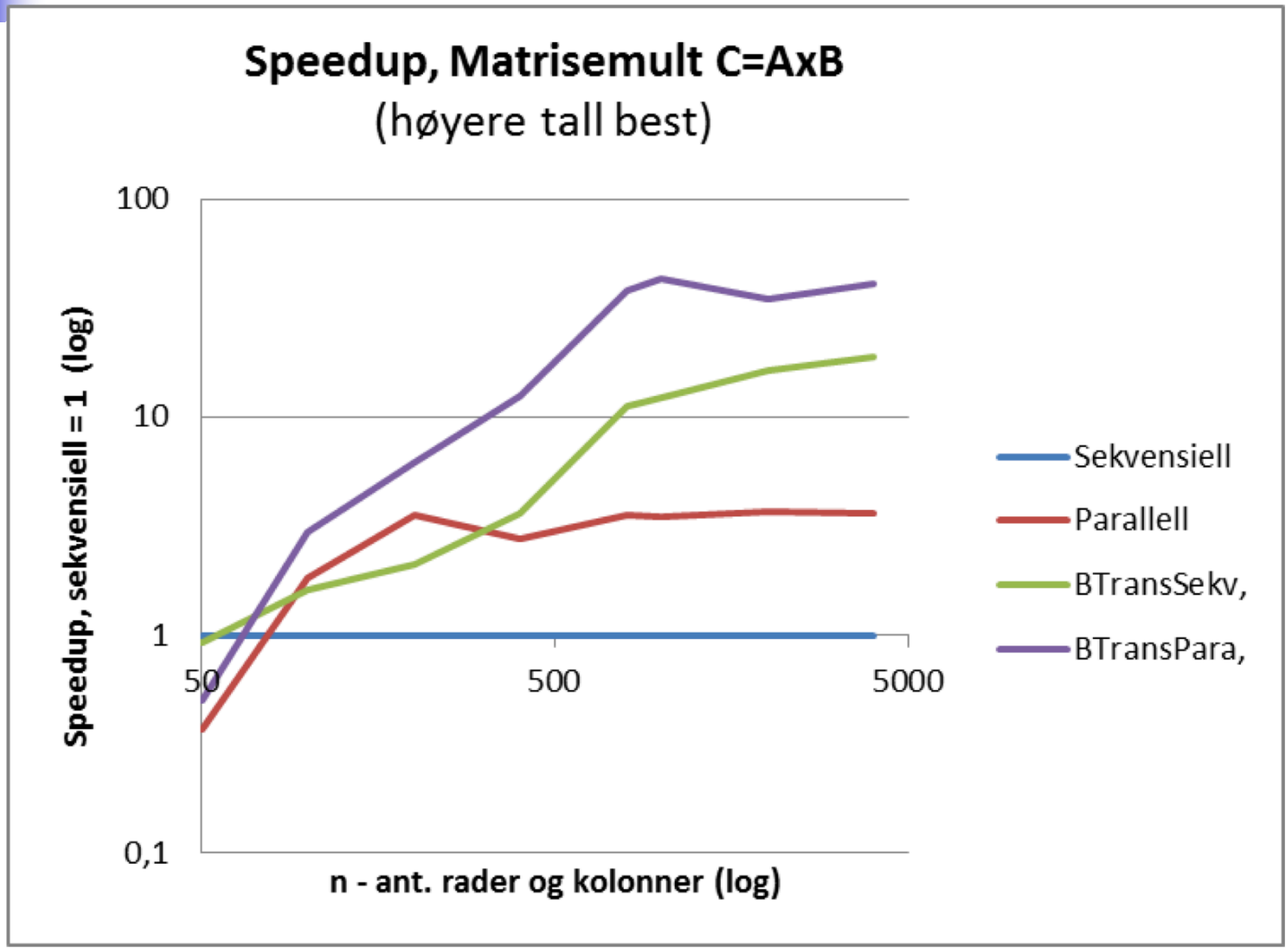
Helt avgjørede for oss – cache-hukommelse

- Hva er cache
  - Raskere (men også dyrere) hukommelse mellom hovedlageret og kjernene.
  - Vi må ha cache fordi det er så store hastighetsforskjeller mellom en CPU-kjerne og hovedlageret ('main memory')
  - Ofte nå 3-4 lag med cache hukommelser + et antall registere i kjernen (enda raskere enn cache-hukommelsene) som holder data eller intruksjoner
  - Når en kjerne trenger data eller en ny instruksjon (og den ikke har det i et register) leter den nedover i cache-hukommelsene. Først cache level 1 (L1), så L2 cachen, .. , før den går til hovedhukommelsen for data eller instruksjoner.
  - Det finns flere teknikker for å gjøre dette raskt (som pre-fetch , dvs at systemet henter neste data/instruksjon uten at kjernen eksplisitt har bedt om det)

Kjøretider – i millisek. (y-aksen logaritmisk)



Kjøretidsresultater – Speedup , y-aksen logaritmisk

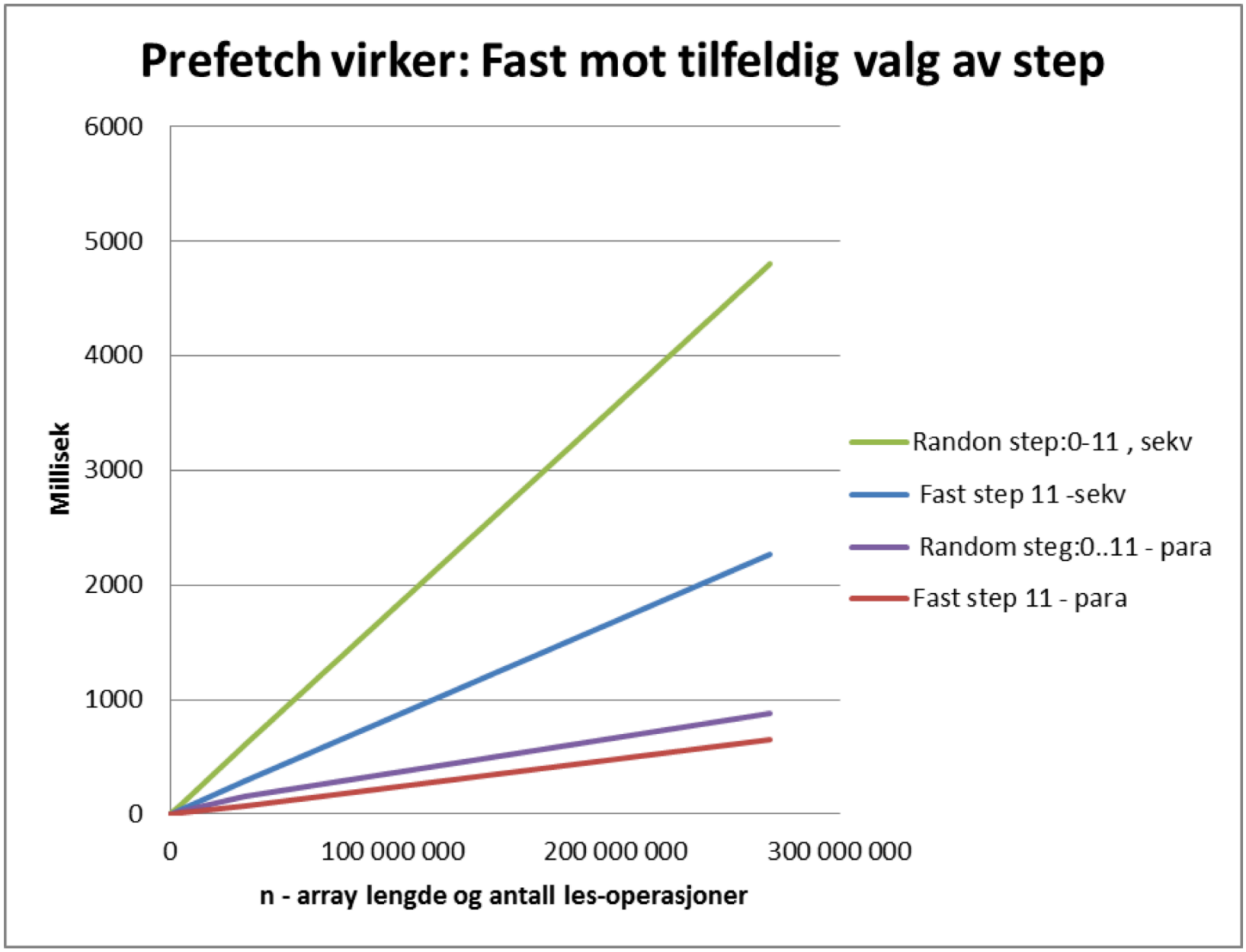


Prefetch-mekanismen på brikken

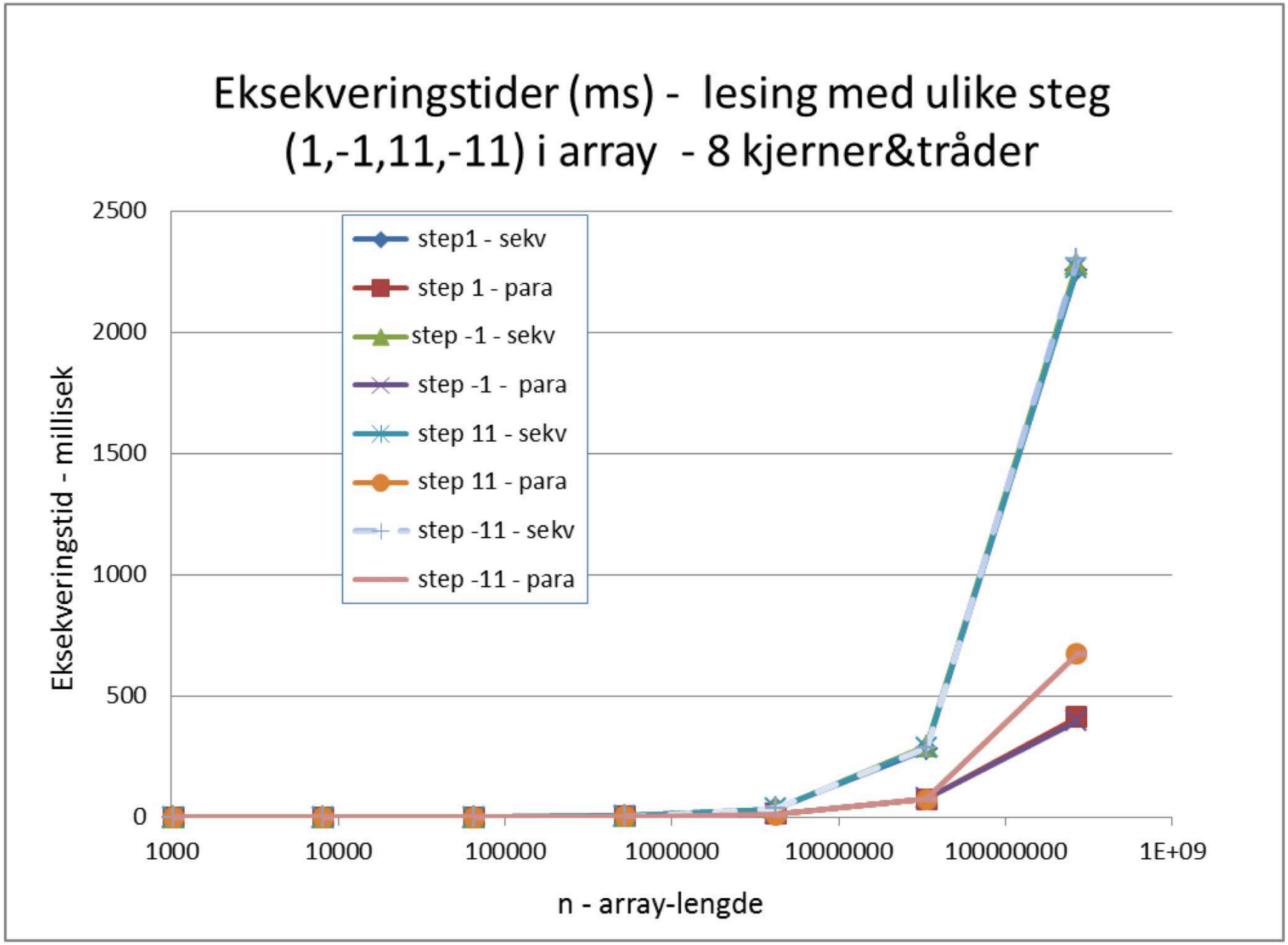
- Består i at hvis vi aksesserer (leser eller skriver) element i k i en array a[] og så element k+m, så prøver elektronikken å hente element a[k+2m], a[k+3m],…, ***før*** vi har bedt om det.
- Tillegget m kan være både positivt og negativt .
- Hvis elementene a[k+2m] ligger i samme cachelinje, går dette spesielt fort.
- Skrev testprogram for dette og testet m = 1, -1,11 og -11

```
for (int i=0;i < a.length; i++){
    // index = Math.abs((index+r.nextInt(step+1))%a.length);
    index = Math.abs((i+step)%a.length );
    sum += a[index];
}
```

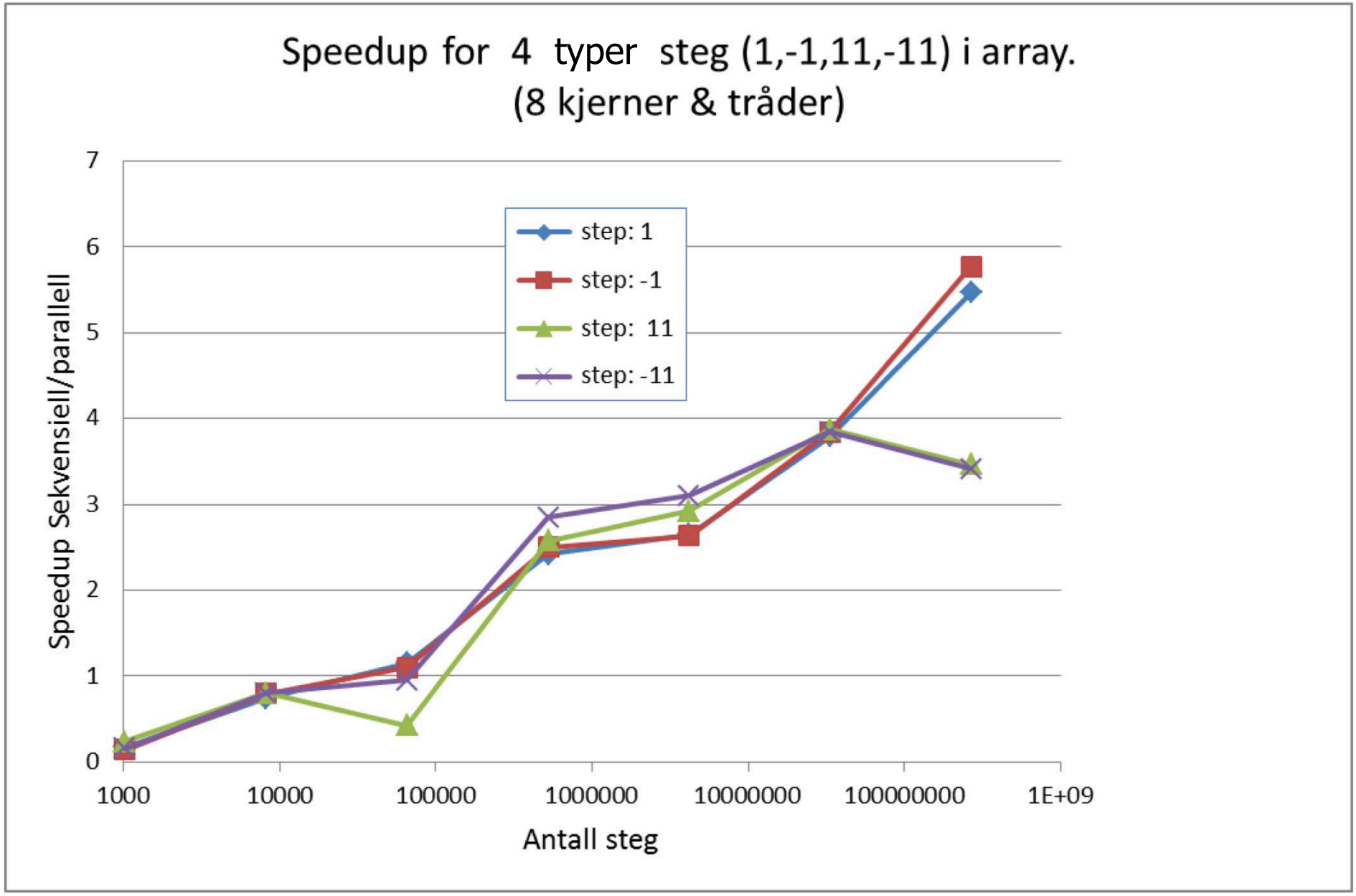
- Hva fant vi og hva kan vi slutte av det.
- Først grafer



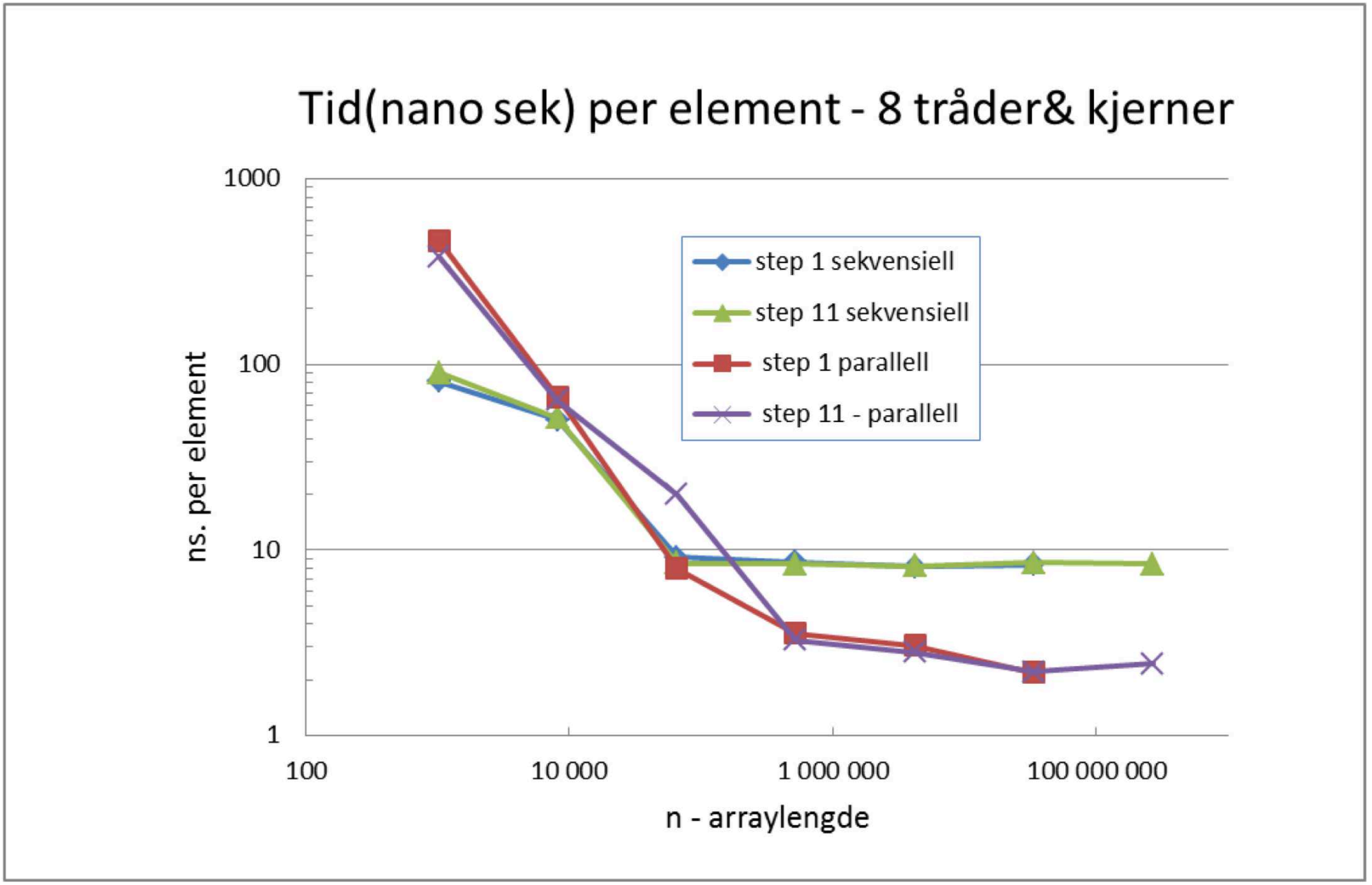
Konklusjon1: Fast steglengde er ca. dobbelt så raskt som tilfeldig (pga prefetch)



Konklusjon2: negative og positive tillegg er like raske.

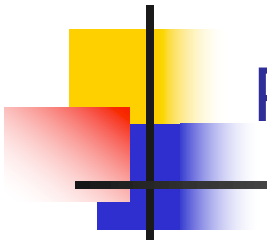


Konklusjon3: Spesielt steg 1,-1 har bra speedup pga. samme cachelinje



Konklusjon 4: Prosesseringstiden per element synker med økende n (JIT?)

Konklusjon 5: Per element tar det litt over 2*2,8 = ca. 6 instruksjoner å summere et element til en sum i parallell.



Prefetch-mekanismen hjelper en del

- Ikke så viktig som cache-systemet
- Ikke så viktig som JIT-kompilering
- men hjelper til og går på ingen måte i veien for de to viktigste mekanismene – ca. 2x raskere
- Programmet som laget data til disse grafene er laget av programmet Prefetch.java som er lagt ut på hjemmesida
- Grafene er laget i Excel (velg graftype:scatter diagram):
  - sett inn et slikt i regnearket og trykk så Select Data

Effekten på eksekveringstider av cache

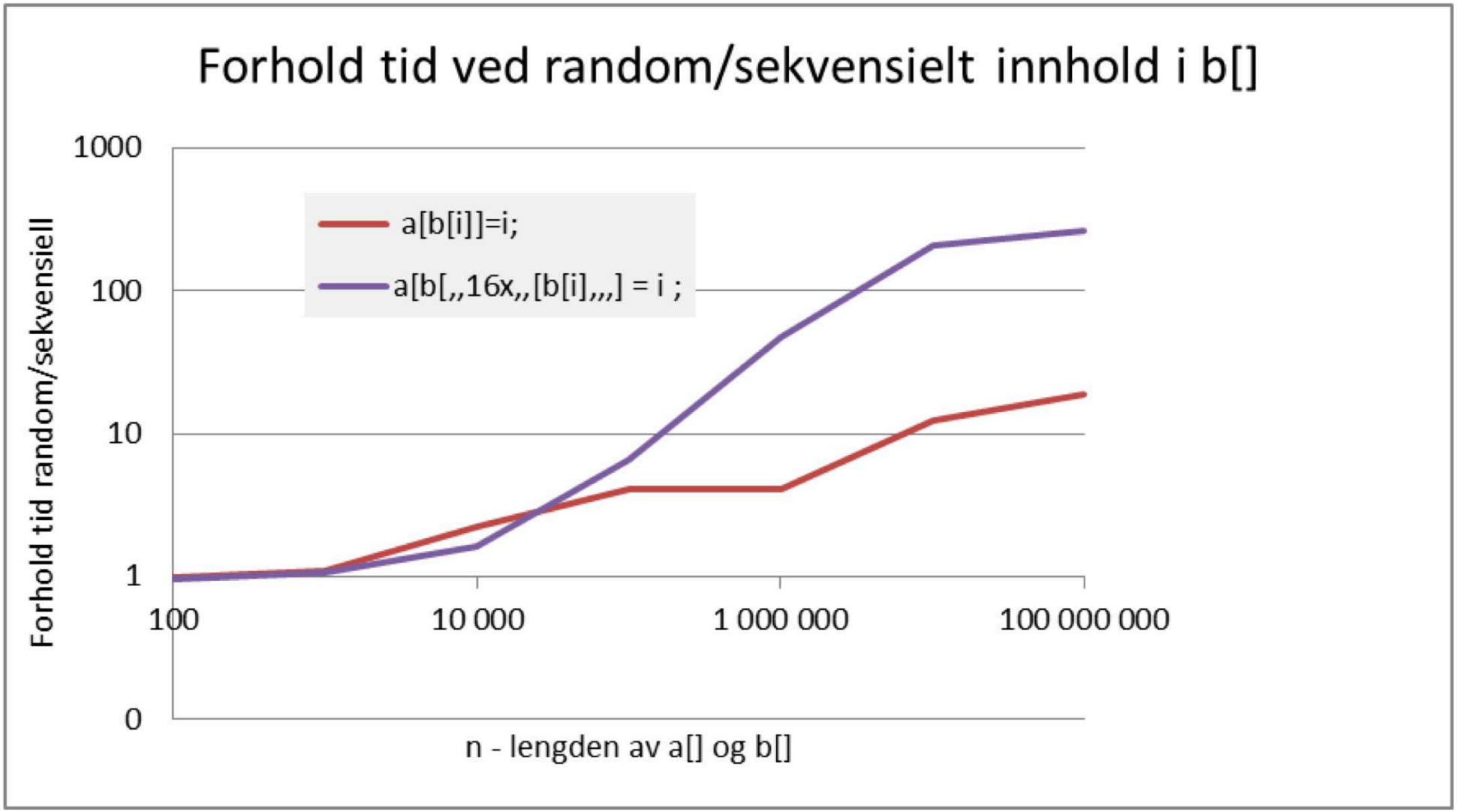
1) Hvor lang tid tar det å utføre n ganger (n=100, 1000, 10 000,….,100 mill):

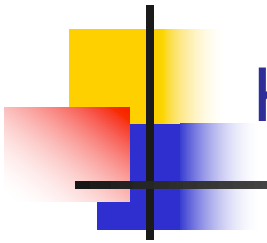
   a[b[i]]=i;

2) Avhenger av hva b[] inneholder:
   1) Hvis b[i] = i (sekvensiell) , så er a[b[i]] = a[i] og vi har 'alt' i cachen
   2) Hvis innholdet i b[] er tilfeldig trukket mellom 0:n-1, så er hver les/skriv i lageret en hopping frem og tilbake i a[] – ingen nytte av cachen

3) Neste graf viser hvor mange ganger lenger tid det tar å utføre ganger de to måtene å fylle b[]
   – enten b[i] = i, eller b[i] = random(0..n-1)

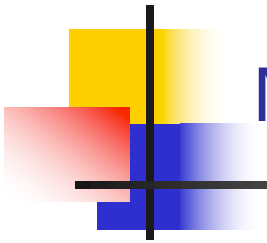
Hvor mange ganger tregere går random innhold i b[] enn b[] = 0,1,2,3,.. ?





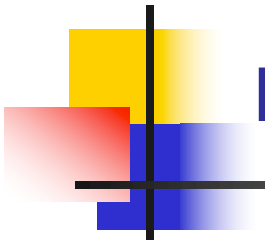
Konklusjon – nestet aksess a[b[i]]

- For 'små' verdier av n <1000, gir cachen god aksess til både hele a[] (viktigst), og til b[].
- For store verdier av n > 100 0000 blir det meget langsommere, og vi kan få mellom 12 – 240 ganger langsommere kode (pga. cache-miss) når innholdet av b[] er 'tilfeldig'.
- Slike uttrykk a[b[i]] og a[b[c[i]]] finner vi i Radix-sortering som vi skal granske i en senere forelesning.



Matrix Multiplication

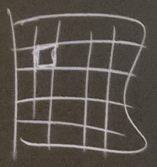
- (Blackboard)

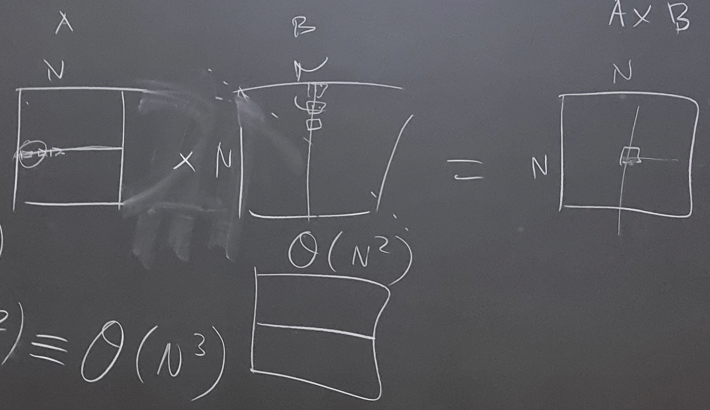


End L03v24

- End of lecture L03v24

Tiling





$O(N^3) + O(N^2)$

$O(N^3) + O(N^2) \equiv O(N^3)$

Single element

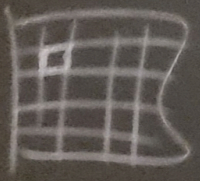
$O(2N)$

$O(N)$

For entire array

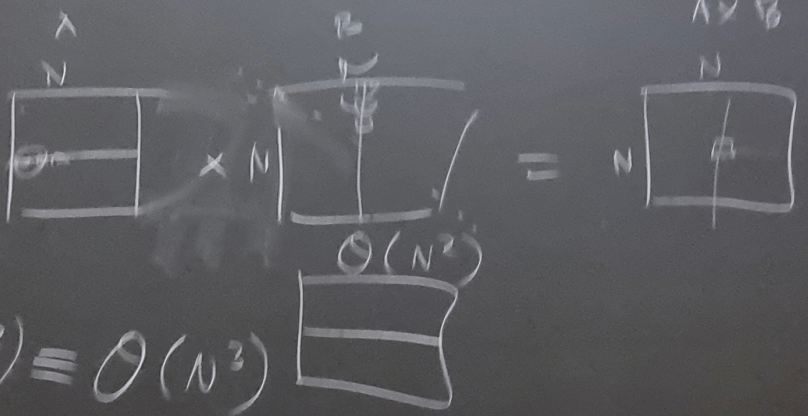
$O(N^3)$

Tiling



$\Theta(N^3) + \Theta(N^2)$

$\Theta(N^3) + \Theta(N^2) \equiv \Theta(N^3)$



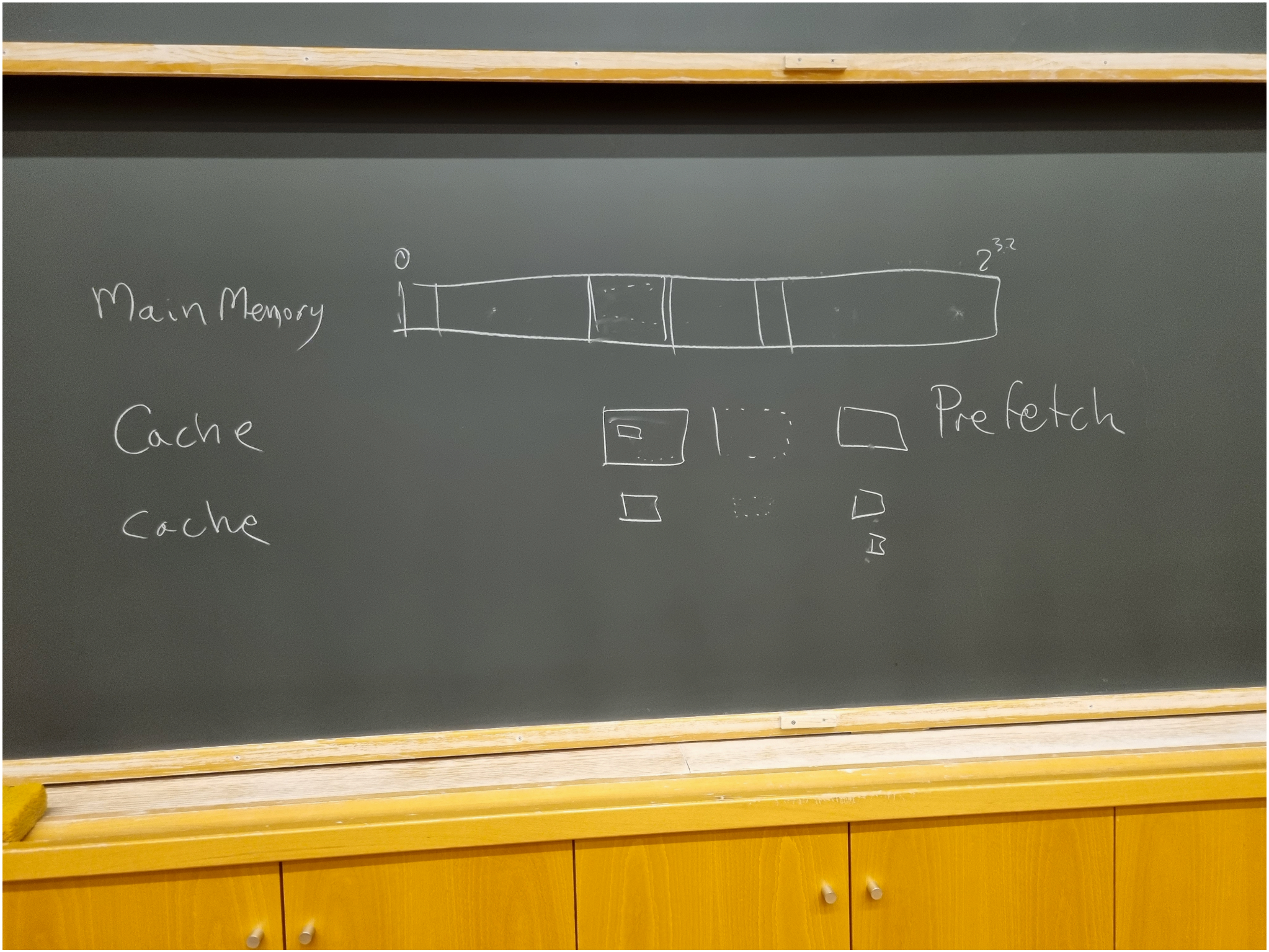
Single element

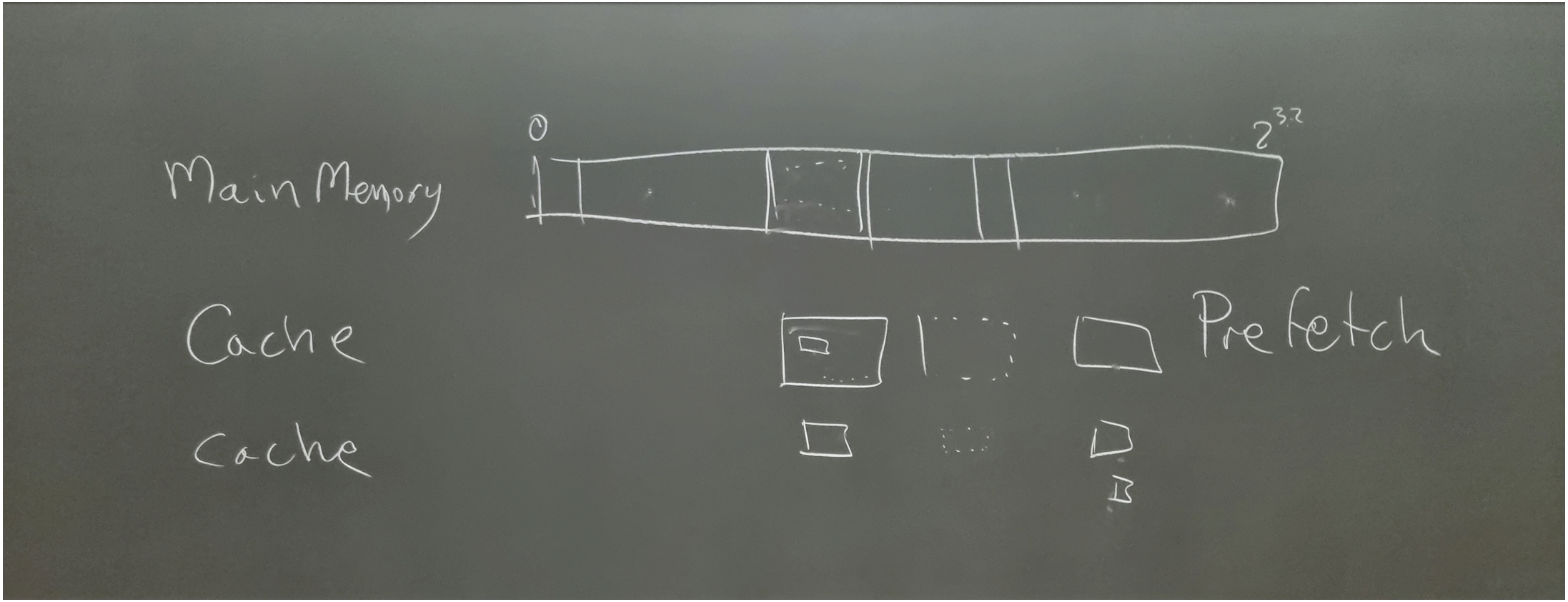
$\Theta(2N)$

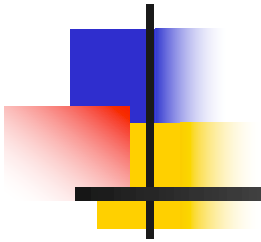
$\Theta(N)$

For entire array

$\Theta(N^3)$

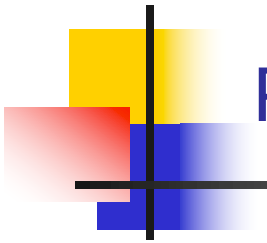

Main Memory
0
2^{32}
Cache
Cache
Prefetch


Main Memory
0
2^{32}
Cache
Cache
Prefetch



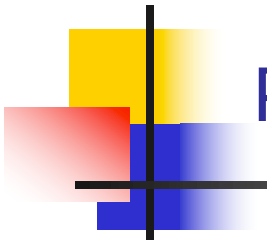
IN3030 L06v23 – Prime Numbers

Eric Jul
Programming Technology Group
Department of Informatics
University of Oslo



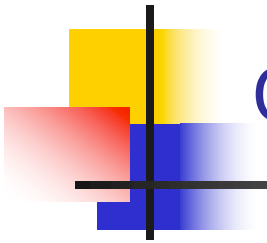
Review F05

I. Introduction to Quantum Computing



Plan for F06v24

I. Prime Numbers
II. Oblig 3



Om primtall

- Primtall og faktorisering av ikke-primtall.
- Et primtall er:
  Et heltall som bare lar seg dividere med 1 og seg selv.
  - 1 er ikke et primtall (det mente mange på 1700-tallet, og noen mener det fortsatt)

Om primtall og faktorisering av heltall

- Ethvert heltall N > 1 lar seg faktorisere som et produkt av primtall:
  - $N = p_1 * p_2 * p_3 * \ldots\ldots\ldots * p_k$
  - Denne faktoringen er entydig (pånær rækkefølge)
  - gjøres entydig hvis tall i faktoriseringen sorteres
  - Hvis det bare er ett tall i denne faktoriseringen, er N selv et primtall
- Eksempler:
  - 2 = 2
  - 3 = 3
  - 4 = 2*2
  - 5 = 5
  - 6 = 2 * 3 = 3 * 2
  - 7 = 7
  - 8 = 2 * 2 * 2

2 måter å lage primtall

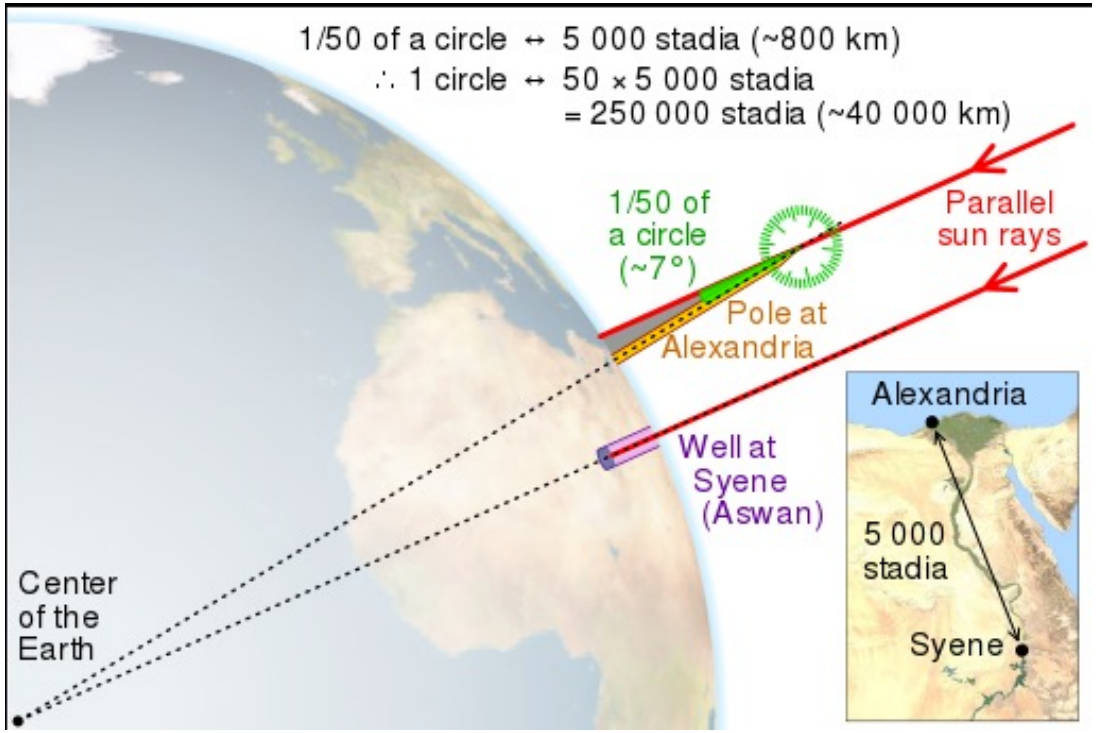
Ønsker at finne alle primtal $p_i < N$

- Dividere alle tall < N med alle tall < N
  - Divisjonsmetoden
  - Bare oddetall (2 spesiel)
  - Bare opp til $\sqrt{N}$ -- hvorfor?
  - Bare primtall opp til $\sqrt{N}$ -- hvorfor?

- Lage en tabell over alle de primtallene vi trenger
  - Eratosthene sil

Litt mer om Eratosthenes

Eratosthenes, matematikker, laget også et estimat på jordas radius som var < 1,5% feil, grunnla geografi som fag, fant opp skuddårsdagen + at han var sjef for Biblioteket i Alexandria (den tids største forskningsinstitusjon).



Hvad er raskest?

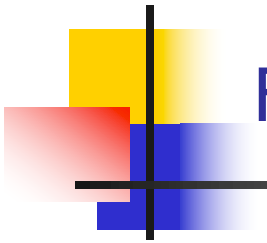
- A) Med Eratosthenes sil:

```
Z:\INF2440Para\Primtall>java PrimtallESil 2000000000
 max primtall m:2000000000
Genererte alle primtall <= 2000000000  paa     18 949 millisek
med Eratosthenes sil  og det største primtallet er:1999999973
```

- Med gjentatte divisjoner

```
Z:\INF2440Para\Primtall>java PrimtallDiv 2000000000
Genererte alle primtall <=2000000000  paa   1 577 302  millisek med
 divisjon , og det største primtallet er:1999999973
```

- Å lage primtallene p og finne dem ved divisjon (del på alle oddetall < SQRT(p), p = 3,5,7,..) er ca. 100 ganger langsommere enn Eratosthenes avkryssings-tabell (kalt Eratosthenes sil).



Finne primtall -- Eratosthenes sil

- Hvordan?
- (Blackboard)

Om primtall og faktorisering av heltall

- Ethvert heltall N > 1 lar seg faktorisere som et produkt av primtall:
  - $N = p_1 * p_2 * p_3 * \ldots\ldots\ldots * p_k$
  - Denne faktoringen er entydig (pånær rækkefølge)
  - gjøres entydig hvis tall i faktoriseringen sorteres
  - Hvis det bare er ett tall i denne faktoriseringen, er N selv et primtall
- Eksempel: faktorisering av 532
  - 532 = 2 * 266
  - 532 = 266*2 = 2 * 2 * 133
  - 532 = 266*2 = 2 * 2 * 7 * 19

Å lage og lagre primtall (Erotosthenes sil)

- Som en bit-tabell (1- betyr primtall, 0-betyr ikke-primtall)
  - Påfunnet i jernalderen av Eratosthenes (ca. 200 f.kr)
  - Man skal finne alle primtall < M
  - Man finner da de første primtallene og krysser av alle multipla av disse (N.B. dette forbedres/endres senere):
    - Eks: 3 er et primtall, da krysses 6, 9,12,15,.. Av fordi de alle er ett-eller-annet-tall (1,2,3,4,5,..) ganger 3 og følgelig selv ikke er et primtall. 6=2*3, 9 = 3*3,
      12 =2*2*3, 15 = 5*3, ..osv
    - De tallene som *ikke blir* krysset av, når vi har krysset av for alle primtallene vi har, er primtallene
- Vi finner 5 som et primtall fordi, etter at vi har krysset av for 3, finner første ikke-avkryssete tall: 5, som da er et primtall (og som vi så krysser av for, …finner så 7 osv)

Litt mer om Eratothenes sil

- Vi representerer ikke partallene på den tallinja som det krysses av på fordi vi vet at 2 er et primtall (det første) og at alle andre partall er ikke-primtall.
- Har vi funnet et nytt primtall p, for eksempel. 5, starter vi avkryssingen for dette primtallet først for tallet p*p (i eksempelet: 25), men etter det krysses det av for p*p+2p, p*p+4p,.. (i eksempelet 35,45,55,…osv.). Grunnen til at vi kan starte på p*p er at alle andre tall t < p*p slik det krysses av i for eksempel Wikipedia-artikkelen har allerede blitt krysset av andre primtall < p.
- Det betyr at for å krysse av og finne alle primtall < N , behøver vi bare å krysse av på denne måten for alle primtall p ≤ sqrt(N). Dette sparer svært mye tid.

Vise at vi trenger bare primtallene <10 for å finne alle primtall < 100, avkryssing for 3 (3*3, 9+2*3,9+4*3, ….)

1	**3**	**5**	**7**	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75	77	79
81	83	85	87	89
91	93	95	97	99

1	**3**	**5**	**7**	**9**
11	13	**15**	17	19
21	23	25	**27**	29
31	**33**	35	37	**39**
41	43	**45**	47	49
51	53	55	**57**	59
61	**63**	65	67	**69**
71	73	**75**	77	79
81	83	85	**87**	89
91	**93**	95	97	**99**

Avkryssing for 5 (starter med 25, så 25+2*5, 25+4,5,..):

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75	77	79
81	83	85	87	89
91	93	95	97	99

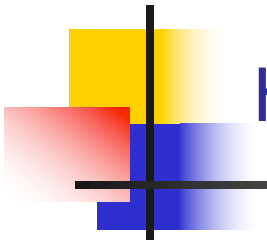
1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

Avkryssing for 7 (starter med 49, så 49+2*7,49+4*7,.):

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63 63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

Er nå ferdig fordi neste primtall vi finner: 11, så er 11*11=121 utenfor tabellen

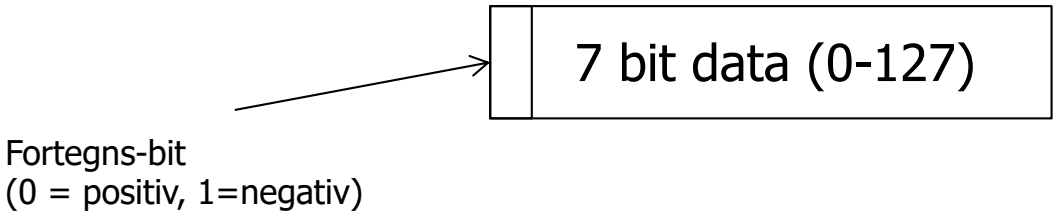


Hvordan representeres tallene?

- Kun oddetall – 2 kjenner vi!
- Array of Boolean?
  - Problem: 32 bit per primtall
- Kompakter bitarray
    - Kun 1 bit per oddetall

Hvordan bruke 8 eller 7 bit i en **byte-array** for å representere primtallene

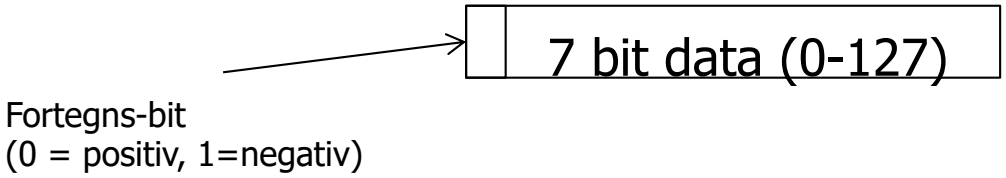
En byte = 8 bit heltall:



- Vi representer alle oddetallene (1,3,5,,,) som ett bit (0= ikke-primtall, 1 = primtall)
- Bruke alle 8 bit :
  - Fordel: mer kompakt lagring og litt raskere(?) adressering
  - Ulempe: Kan da ikke bruke verdien i byten direkte (f.eks som en indeks til en array), heller ikke +,-,* eller /-operasjonene på verdien
- Bruke 7 bit:
  - Fordel: ingen av ulempene med 8 bit
  - Ulempe: Tar litt større plass og litt langsommere(?) adressering

Hvordan representere 8 (eller 7) bit i en byte-array

byte = et 8 bit heltall



- Bruker alle 8 bitene til oddetallene:
  - Anta at vi vil sjekke om tallet ***k*** er et primtall, sjekk først om ***k*** er 2, da ja, hvis det er et partall (men ikke 2) da nei – ellers sjekk så tallets bit i byte-arrayen
    - Byte nummeret til ***k*** i arrayen er da:
      - Enten: ***k***/16, eller: ***k*** >>>4 (shift 4 høyreover uten kopi av fortegns-bitet er det samme som å dele med 16)
    - Bit-nummeret er i denne byten er da enten (***k*** %16)/2 eller (***k*** &15)>>1
  - Hvorfor dele på 16 når det er 8 bit
    - fordi vi fjernet alle partallene – egentlig 16 tall representert i første byten, for byte 0: tallene 0-15
  - Om så å finne bitverdien – se neste lysark.

Bruke 7 bit i hver byte i arrayen

- Anta at vi vil sjekke om tallet ***k*** er et primtall sjekk først om ***k*** er 2, da ja, ellers hvis det er et partall (men ikke 2) da nei – ellers:
- Sjekk da tallets bit i byte-arrayen
  - Byte nummeret til ***k*** i arrayen er da: ***k***/14
  - Bit-nummeret er i denne byten er da: (***k*** %14)/2
- Nå har vi byte-nummeret og bit-nummeret i den byten. Vi kan da ta AND (&) med det riktige elementet i en av de to arrayene som er oppgitt i skjelett-koden og teste om svaret er 0 eller ikke.
- Hvordan sette alle 7 eller 8 bit == 1 i alle byter)
  - 7 bit: hver byte settes = 127 (men bitet for 1 settes =0)
  - 8 bit: hver byte settes = -1 (men bit for 1 settes = 0)

- Konklusjon: bruk 8 eller 7 bit i hver byte (valgfritt) i Oblig3

Faktorisering av et tall M i sine primtallsfaktorer

- Vi har laget og lagret ved hjelp av Erotosthanes sil alle (unntatt 2) primtall < N i en bit-array over alle odde-tallene.
  - 1 = primtall, 0=ikke-primtall
  - Vi har krysset ut de som ikke er primtall
- Hvordan skal vi så bruke dette til å faktorisere et tall M < N*N ?
- **Svar:** Divider M med alle primtall $p_i < \sqrt{M}$ (p_i =2,3,5,..), og hver gang en slik divisjon M%p_i ==0, så er p_i en av faktorene til M. Vi forsetter så med å faktorisere ett mindre tall M'=M/p_i.
- Faktoriseringen av M = p_i *..* p_k er da produktet av alle de primtall som dividerer M uten rest.
- HUSK at en p_i kan forekommer flere ganger i svaret. eks: 20= 2*2*5, 81 = 3*3*3*3, osv
- Finner vi ingen faktorisering av M, dvs. ingen $p_i \leq \sqrt{M}$ som dividerer M med rest == 0, så er M selv et primtall.

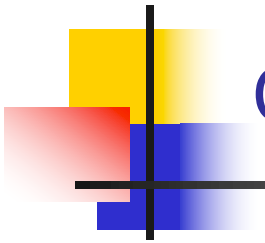
Hvordan parallellisere faktorisering ?

1. Gjennomgås neste uke - denne uka viktig å få på plass en effektiv sekvensiell løsning med om lag disse kjøretidene for N = 2 mill:

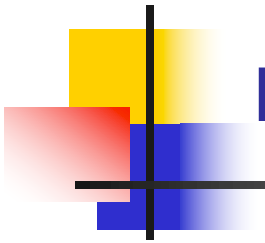
```
M:\INF2440Para\Primtall>java PrimtallESil 2000000
 max primtall m:2 000 000
Genererte primtall <= 2000000  paa        15.56 millisek
med Eratosthenes sil (  0.00004182 millisek/primtall)
……
3999998764380 = 2*2*3*5*103*647248991
3999998764381 = 37*108108074713
3999998764382 = 2*271*457*1931*8363
3999998764383 = 3*19*47*1493093977
3999998764384 = 2*2*2*2*2*7*313*1033*55229
3999998764385 = 5*13*59951*1026479
3999998764386 = 2*3*3*31*71*100964177
3999998764387 = 1163*1879*1830431
3999998764388 = 2*2*11*11*17*23*293*72139
 100 faktoriseringer beregnet paa:     422.0307ms -
 dvs:      4.2203ms. per faktorisering
```

Faktorisering av store tall med 18-19 desimale sifre

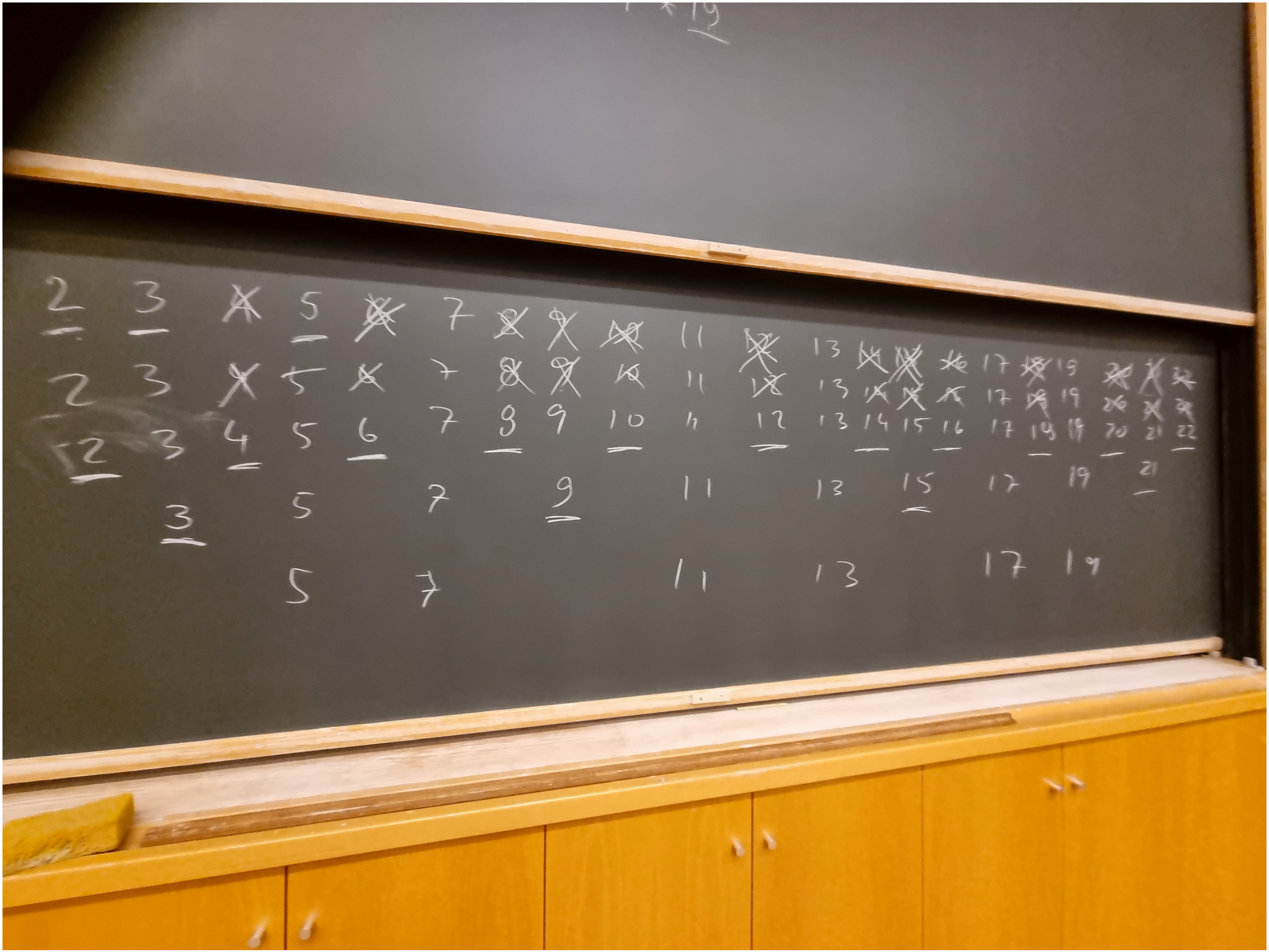
```
Uke5>java PrimtallESil 2140000000
max primtall m:2 140 000 000
bitArr.length:133 750 001
Genererte primtall <= 2 140 000 000  paa    11030.36 millisek
med Eratosthenes sil ( 0.00010530 millisec/primtall)
 antall primtall < 2 140 000 000 er: 104 748 779, dvs:  4.89% ,
 og det største primtallet er: 2 139 999 977

4 579 599 999 999 999 900 = 2*2*3*5*5*967*3673*19421*221303
4 579 599 999 999 999 901 = 457959999999999901
4 579 599 999 999 999 902 = 2*228979999999999951
4 579 599 999 999 999 903 = 3*31*13188589*3733758839
4 579 599 999 999 999 904 = 2*2*2*2*2*19*71*106087842846553
4 579 599 999 999 999 905 = 5*7*1308457142857142 83
.............
4 579 599 999 999 999 997 = 11*41632727272727272 7
4 579 599 999 999 999 998 = 2*121081*18911307306679
4 579 599 999 999 999 999 = 3*17*19*6625387*713333333
100 faktoriseringer beregnet paa:  333481.4427ms
 dvs:   3334.8144ms. per faktorisering

 largestLongFactorizedSafe: 4 579 599 841 640 001 173= 2139999949*2139999977
```

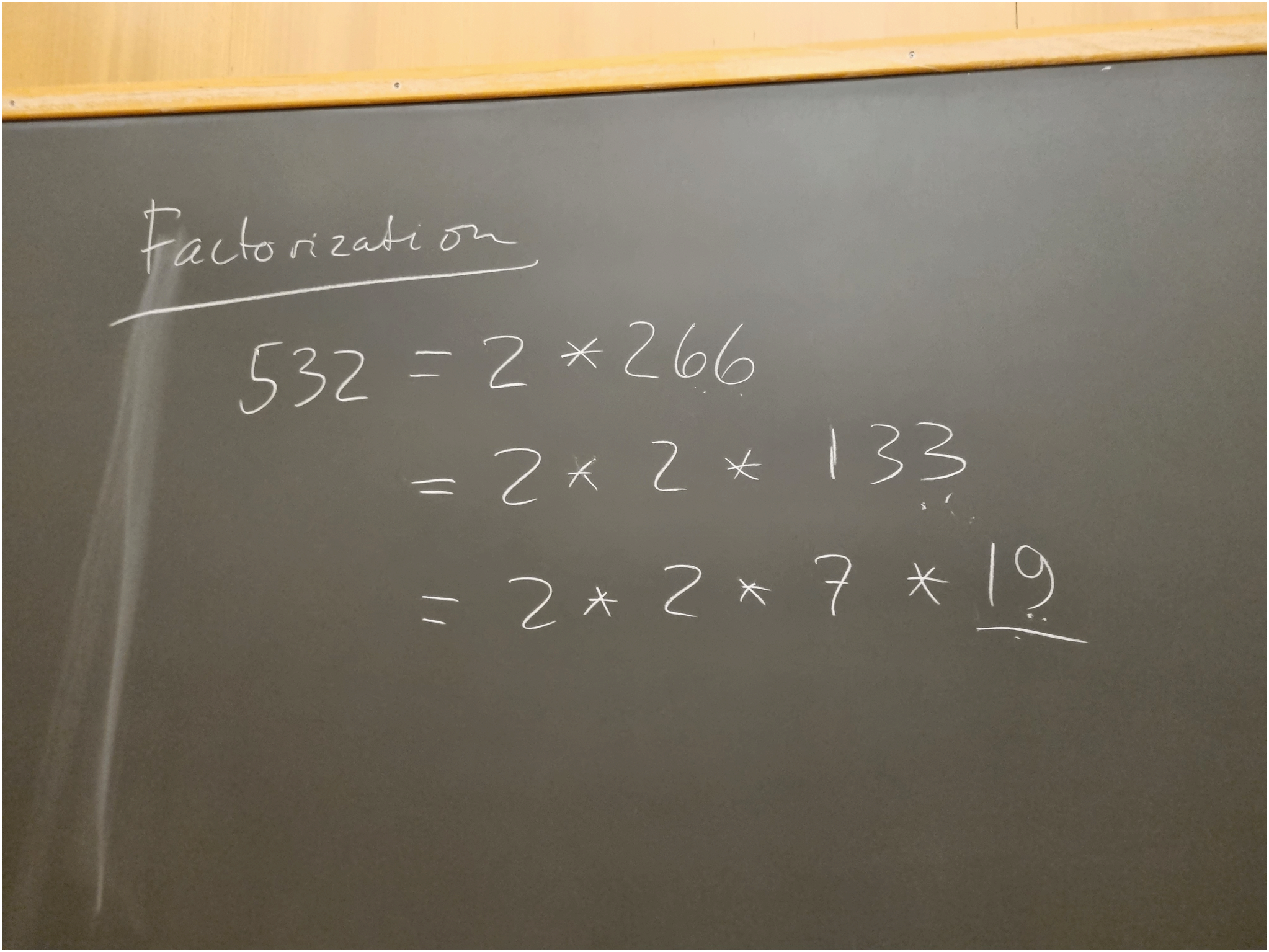


Oblig 3:



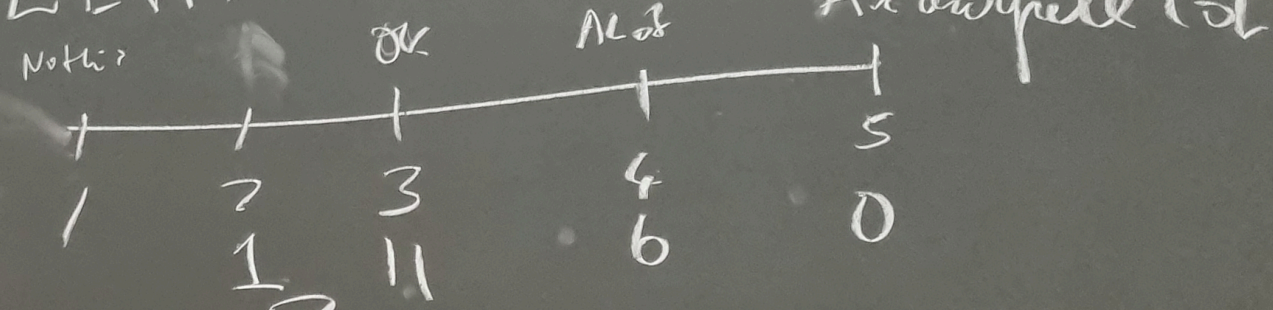
End of lecture L06v23 and L06v24



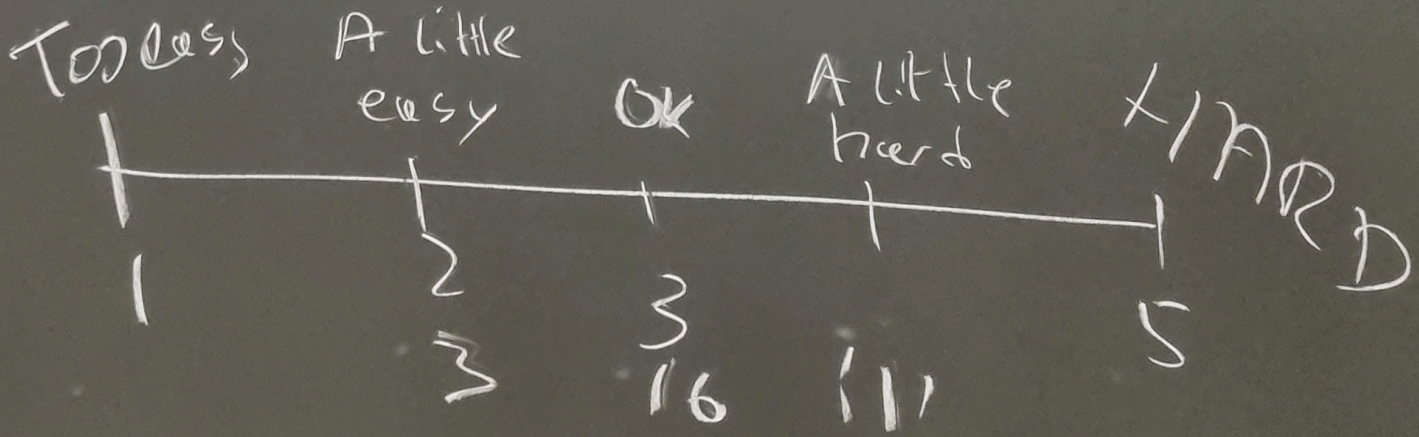

Factorization
532 = 2 * 266
= 2 * 2 * 133
= 2 * 2 * 7 * 19

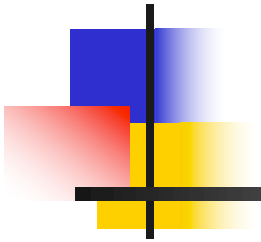
For FIRST 7 lectures

DID YOU LEARN ANYTHING?



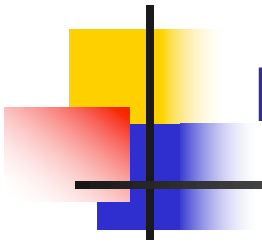
WAS IT HARD?





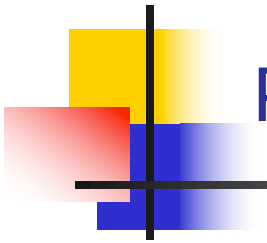
IN3030 L07v24 – Prime Numbers, Timing

Eric Jul
Programming Technology Group
Department of Informatics
University of Oslo
2024-02-29



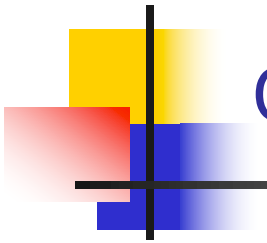
Review L06v23

I. Prime Numbers, Erasthophenes sieve
II. Oblig Prime Numbers



Plan for L07v24

I. Prime Numbers, review
II. Oblig Prime Numbers -- Review
III. Tidtagning
    I. JIT compilation
    II. Operativsystem?
    III. Søppel/Garbage Collection
IV. JMH
V. Amdahl
VI. Gustavson



Om primtall

- Primtall og faktorisering av ikke-primtall.
- Et primtall er:
  Et heltall som bare lar seg dividere med 1 og seg selv.
  - 1 er ikke et primtall (det mente mange på 1700-tallet, og noen mener det fortsatt)

Om primtall og faktorisering av heltall

- Ethvert heltall N > 1 lar seg faktorisere som et produkt av primtall:
  - $N = p_1 * p_2 * p_3 * \ldots\ldots\ldots * p_k$
  - Denne faktoringen er entydig (pånær rækkefølge)
  - gjøres entydig hvis tall i faktoriseringen sorteres
  - Hvis det bare er ett tall i denne faktoriseringen, er N selv et primtall

2 måter å lage primtall

Ønsker at finne alle primtal $p_i < N$

- Dividere alle tall < N med alle tall < N
  - Divisjonsmetoden
  - Bare oddetall (2 spesiel)
  - Bare opp til $\sqrt{N}$ -- hvorfor?
  - Bare primtall opp til $\sqrt{N}$ -- hvorfor?

- Lage en tabell over alle de primtallene vi trenger
  - Eratosthene sil

Hvad er raskest?

- A) Med Eratosthenes sil:

```
Z:\INF2440Para\Primtall>java PrimtallESil 2000000000
 max primtall m:2000000000
Genererte alle primtall <= 2000000000  paa     18 949 millisek
med Eratosthenes sil  og det største primtallet er:1999999973
```

- Med gjentatte divisjoner

```
Z:\INF2440Para\Primtall>java PrimtallDiv 2000000000
Genererte alle primtall <=2000000000  paa   1 577 302  millisek med
 divisjon , og det største primtallet er:1999999973
```

- Å lage primtallene p og finne dem ved divisjon (del på alle oddetall < SQRT(p), p = 3,5,7,..) er ca. 100 ganger langsommere enn Eratosthenes avkryssings-tabell (kalt Eratosthenes sil).

Om primtall og faktorisering av heltall

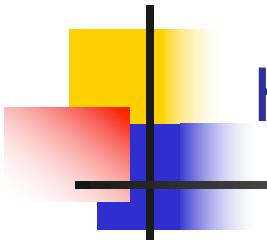
- Ethvert heltall N > 1 lar seg faktorisere som et produkt av primtall:
  - $N = p_1 * p_2 * p_3 * \ldots\ldots\ldots * p_k$
  - Denne faktoringen er entydig (pånær rækkefølge)
  - gjøres entydig hvis tall i faktoriseringen sorteres
  - Hvis det bare er ett tall i denne faktoriseringen, er N selv et primtall
- Eksempel: faktorisering av 532

Å lage og lagre primtall (Erotosthenes sil)

- Som en bit-tabell (1- betyr primtall, 0-betyr ikke-primtall)
  - Påfunnet i jernalderen av Eratosthenes (ca. 200 f.kr)
  - Man skal finne alle primtall < M
  - Man finner da de første primtallene og krysser av alle multipla av disse (N.B. dette forbedres/endres senere):
    - Eks: 3 er et primtall, da krysses 6, 9,12,15,.. Av fordi de alle er ett-eller-annet-tall (1,2,3,4,5,..) ganger 3 og følgelig selv ikke er et primtall. 6=2*3, 9 = 3*3,
      12 =2*2*3, 15 = 5*3, ..osv
    - De tallene som *ikke blir* krysset av, når vi har krysset av for alle primtallene vi har, er primtallene
- Vi finner 5 som et primtall fordi, etter at vi har krysset av for 3, finner første ikke-avkryssete tall: 5, som da er et primtall (og som vi så krysser av for, …finner så 7 osv)

Litt mer om Eratothenes sil

- Vi representerer ikke partallene på den tallinja som det krysses av på fordi vi vet at 2 er et primtall (det første) og at alle andre partall er ikke-primtall.
- Har vi funnet et nytt primtall p, for eksempel. 5, starter vi avkryssingen for dette primtallet først for tallet p*p (i eksempelet: 25), men etter det krysses det av for p*p+2p, p*p+4p,.. (i eksempelet 35,45,55,…osv.). Grunnen til at vi kan starte på p*p er at alle andre tall t < p*p slik det krysses av i for eksempel Wikipedia-artikkelen har allerede blitt krysset av andre primtall < p.
- Det betyr at for å krysse av og finne alle primtall < N , behøver vi bare å krysse av på denne måten for alle primtall p ≤ sqrt(N). Dette sparer svært mye tid.



Hvordan representeres tallene?

- Kun oddetall – 2 kjenner vi!
- Array of Boolean?
  - Problem: 32 bit per primtall
- Kompakter bitarray
    - Kun 1 bit per oddetall

Faktorisering av et tall M i sine primtallsfaktorer

- Vi har laget og lagret ved hjelp av Erotosthanes sil alle (unntatt 2) primtall < N i en bit-array over alle odde-tallene.
  - 1 = primtall, 0=ikke-primtall
  - Vi har krysset ut de som ikke er primtall
- Hvordan skal vi så bruke dette til å faktorisere et tall M < N*N ?
- **Svar:** Divider M med alle primtall $p_i < \sqrt{M}$ (p_i =2,3,5,..), og hver gang en slik divisjon M%p_i ==0, så er p_i en av faktorene til M. Vi forsetter så med å faktorisere ett mindre tall M'=M/p_i.
- Faktoriseringen av M = p_i *..* p_k er da produktet av alle de primtall som dividerer M uten rest.
- HUSK at en p_i kan forekommer flere ganger i svaret. eks: 20= 2*2*5, 81 = 3*3*3*3, osv
- Finner vi ingen faktorisering av M, dvs. ingen $p_i \leq \sqrt{M}$ som dividerer M med rest == 0, så er M selv et primtall.

Hvordan parallellisere faktorisering ?

1. Denne uka viktig å få på plass en effektiv sekvensiell løsning med om lag disse kjøretidene for N = 2 mill:

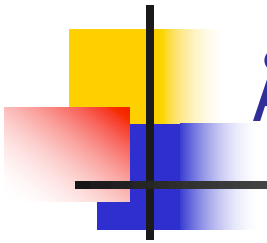
```
M:\INF2440Para\Primtall>java PrimtallESil 2000000
 max primtall m:2 000 000
Genererte primtall <= 2000000  paa        15.56 millisek
med Eratosthenes sil (  0.00004182 millisek/primtall)
……
3999998764380 = 2*2*3*5*103*647248991
3999998764381 = 37*108108074713
3999998764382 = 2*271*457*1931*8363
3999998764383 = 3*19*47*1493093977
3999998764384 = 2*2*2*2*2*7*313*1033*55229
3999998764385 = 5*13*59951*1026479
3999998764386 = 2*3*3*31*71*100964177
3999998764387 = 1163*1879*1830431
3999998764388 = 2*2*11*11*17*23*293*72139
 100 faktoriseringer beregnet paa:     422.0307ms -
 dvs:      4.2203ms. per faktorisering
```

Faktorisering av store tall med 18-19 desimale sifre

```
Uke5>java PrimtallESil 2140000000
max primtall m:2 140 000 000
bitArr.length:133 750 001
Genererte primtall <= 2 140 000 000  paa    11030.36 millisek
med Eratosthenes sil ( 0.00010530 millisec/primtall)
 antall primtall < 2 140 000 000 er: 104 748 779, dvs:  4.89% ,
 og det største primtallet er: 2 139 999 977

4 579 599 999 999 999 900 = 2*2*3*5*5*967*3673*19421*221303
4 579 599 999 999 999 901 = 457959999999999901
4 579 599 999 999 999 902 = 2*228979999999999951
4 579 599 999 999 999 903 = 3*31*13188589*3733758839
4 579 599 999 999 999 904 = 2*2*2*2*2*19*71*106087842846553
4 579 599 999 999 999 905 = 5*7*1308457142857142 83
.............
4 579 599 999 999 999 997 = 11*41632727272727272 7
4 579 599 999 999 999 998 = 2*121081*18911307306679
4 579 599 999 999 999 999 = 3*17*19*6625387*713333333
100 faktoriseringer beregnet paa:  333481.4427ms
 dvs:   3334.8144ms. per faktorisering

 largestLongFactorizedSafe: 4 579 599 841 640 001 173= 2139999949*2139999977
```

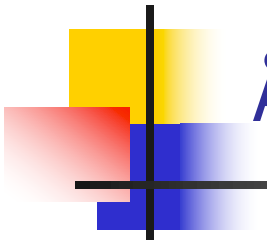
Om å parallelliser et problem

- Utgangspunkt: Vi har en sekvensiell effektiv og riktig sekvensiell algoritme som løser problemet.
- Vi kan dele opp både koden og data (hver for seg?)
- Vanligst å dele opp data
  - Som oftest deler vi opp data, og lar 'hele' koden virke på hver av disse data-delene (en del til hver tråd).
  - Eks: Matriser
    - radvis eller kolonnevis oppdeling av C til hver tråd
    - Omforme data slik at de passer bedre i cachene (transponere B)
  - Rekursiv oppdeling av data ('lett')
    - Eks: Quicksort
- Også mulig å dele opp koden:
  - Alternativ Oblig3 i INF1000: Beregning av Pi (3,1415..) med 17 000 sifre med tre ArcTan-rekker
  - Primtalls-faktorisering av store tall N for kodebrekking:
    - $N = p_1 * p_2$



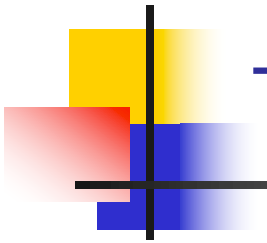
Å dele opp algoritmen

- Koden består en eller flere steg; som oftest i form av en eller flere samlinger av løkker (som er enkle, doble, triple..)
- Vi vil parallellisere med k tråder, og hver slikt steg vil få hver sin parallellisering med en CyclickBarrier-synkronisering mellom hver av disse delene + en synkronisert avslutning (join(), ..).
- Eks:
  - finnMax – hadde ett slikt steg: for (int i = 0 …n-1) -løkke
  - MatriseMult hadde ett slikt steg med trippel-løkke
  - Flere steg mulig: Eksempel Radix sort (nevnes senere)



Å dele opp data – del 2

- For å planlegge parallellisering av ett slikt steg må vi finne:
  - Hvilke data i problemet er lokale i hver tråd?
  - Hvilke data i problemet er felles/delt mellom trådene?
- Viktig for effektiv parallell kode.
  - Hvordan deler vi opp felles data (om mulig)
  - Kan hver tråd beregne hver sin egen, disjunkte del av data
  - Færrest mulig synkroniseringer (de tar 'mye' tid)

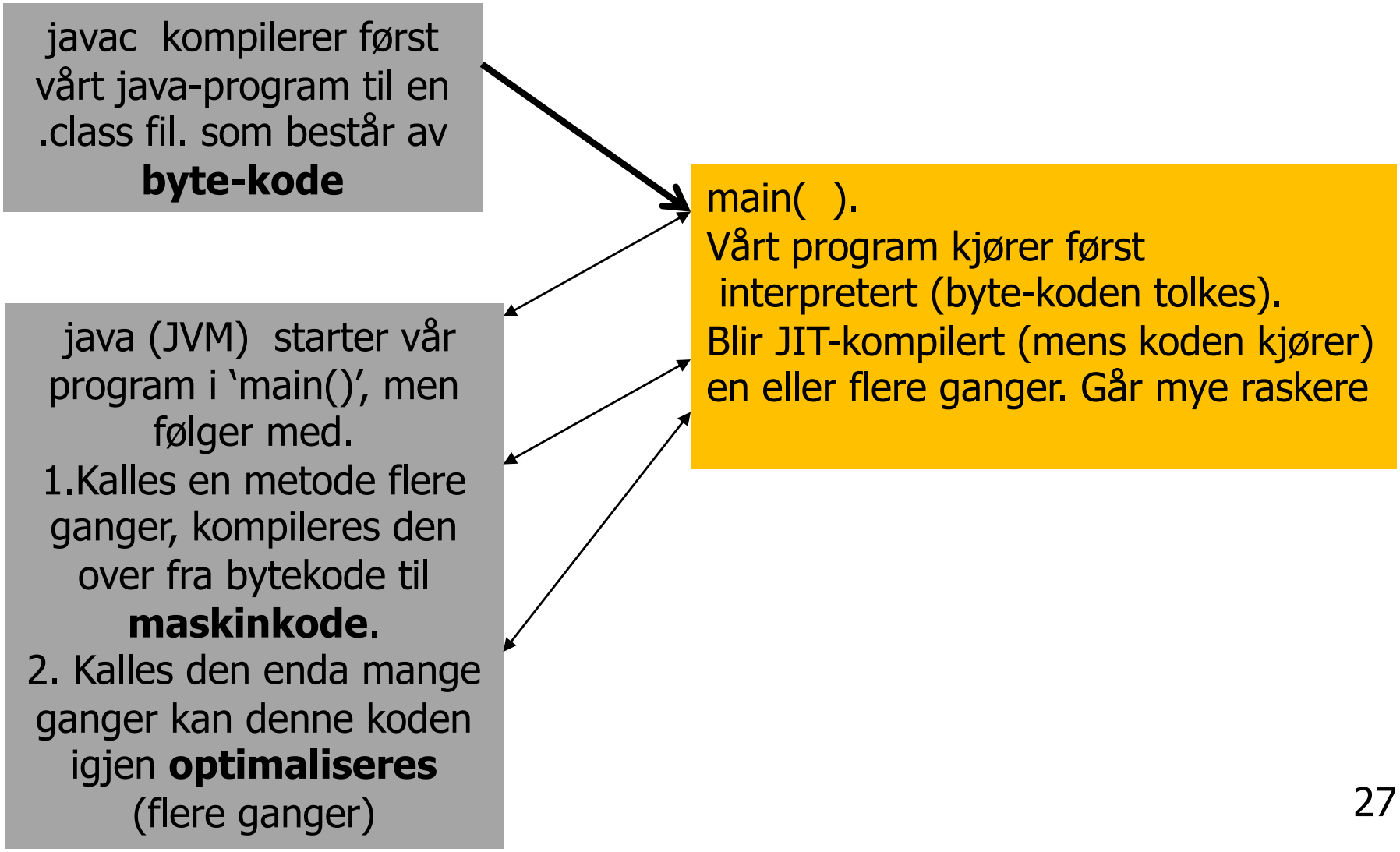


Tidtagning

- JIT –kompilering
  - Hvor mye betyr det egentlig
- Operativsystemet (Windows eller Linux)
  - Er de like raske?
- Søppeltømming i Java
  - Skjer under kjøring (med i tidene)

Tidsmålinger og JIT (Just In Time) -kompilering

- Tilbake til kompileringen av et Java-program:



Optimaliserng – ett eksempel

Original kode

```
class A {
  B b;
  public void newMethod() {
    y = b.get();
    ...do stuff...
    z = b.get();
    sum = y + z;
  }
}
class B {
   int value;
   final int get() {
      return value;
   }
}
```

1) Inline get

```
public void
newMethod() {
    y = b.value;
    ...do stuff...
    z = b.value;
    sum = y + z;
}
```

2) Fjern overflødige les

```
public  void
newMethod() {
    y = b.value;
    ...do stuff...
    z = y;
    sum = y + z;
}
```

3) Fjern overflødige variable

```
public  void
newMethod() {
    y = b.value;
    ...do stuff...
    y = y;
    sum = y + y;
}
```

4) Fjern død kode

```
public  void
newMethod() {
    y = b.value;
    ...do stuff...
    sum = y + y;
}
```

Mediantider for finnMax fra ukeoppgavene:

n= 10 000

Vi ser at kjøretidene (sekv og para) synker dramatisk fra 1.ste til neste kjøring.
Pga JIT-optimalisering

```
M:\INF2440Para\FinnMax>java FinnMaxMulti 10000 7

Kjøring:0, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     6.30 msek. , nanosek/n:  630.46
Max sekv = a:9853, paa:     0.28 msek. , nanosek/n:   28.38

Kjøring:1, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.57 msek. , nanosek/n:   56.87
Max sekv = a:9853, paa:     0.27 msek. , nanosek/n:   26.95

Kjøring:2, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.35 msek. , nanosek/n:   35.07
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.36

Kjøring:3, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.57 msek. , nanosek/n:   56.87
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    0.66

Kjøring:4, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.43 msek. , nanosek/n:   43.47
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.33

Kjøring:5, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.49 msek. , nanosek/n:   49.20
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.36

Kjøring:6, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.48 msek. , nanosek/n:   47.84
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.43

Median seq time:   0.014, median para time:   0.569,
 Speedup:    0.03, n = 10000
```

```
M:\INF2440Para\FinnMax>java FinnMaxMulti 10000000 5

Kjøring:0, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    14.08 msek. , nanosek/n:    1.41
Max sekv = a:9999216, paa:     6.98 msek. , nanosek/n:    0.70

Kjøring:1, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     3.17 msek. , nanosek/n:    0.32
Max sekv = a:9999216, paa:     4.75 msek. , nanosek/n:    0.47

Kjøring:2, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     2.79 msek. , nanosek/n:    0.28
Max sekv = a:9999216, paa:     5.04 msek. , nanosek/n:    0.50

Kjøring:3, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     2.87 msek. , nanosek/n:    0.29
Max sekv = a:9999216, paa:     5.05 msek. , nanosek/n:    0.51

Kjøring:4, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     2.92 msek. , nanosek/n:    0.29
Max sekv = a:9999216, paa:     5.03 msek. , nanosek/n:    0.50

Median seq time:   5.052, median para time:   3.173,
 Speedup:    1.59, n = 10 000 000
```

n= 10 mill

```
M:\INF2440Para\FinnMax>java -Xint FinnMaxMulti 10000000 5

Kjøring:0, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    67.24 msek. , nanosek/n:    6.72
Max sekv = a:9999216, paa:   179.40 msek. , nanosek/n:   17.94

Kjøring:1, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    64.00 msek. , nanosek/n:    6.40
Max sekv = a:9999216, paa:   175.12 msek. , nanosek/n:   17.51

Kjøring:2, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    51.42 msek. , nanosek/n:    5.14
Max sekv = a:9999216, paa:   176.23 msek. , nanosek/n:   17.62

Kjøring:3, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    64.95 msek. , nanosek/n:    6.49
Max sekv = a:9999216, paa:   173.17 msek. , nanosek/n:   17.32

Kjøring:4, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    60.11 msek. , nanosek/n:    6.01
Max sekv = a:9999216, paa:   185.84 msek. , nanosek/n:   18.58

Median seq time: 179.403, median para time:  64.950,
 Speedup:    2.76, n = 10 000 000
```

JIT-kompilering avslått :
> java –Xint
…..
n= 10 mill

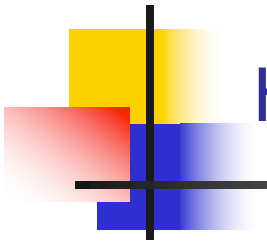
```
M:\INF2440Para\FinnMax>java FinnM 100000000 5
Kjoering:0, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 41.913504 ms.
Max verdi sekvensiell i a:99989305, paa: 238.799921 ms.

Kjoering:1, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 26.78024 ms.
Max verdi sekvensiell i a:99989305, paa: 235.431219 ms.

Kjoering:2, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 27.791271 ms.
Max verdi sekvensiell i a:99989305, paa: 248.066478 ms.

Kjoering:3, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 26.86283 ms.
Max verdi sekvensiell i a:99989305, paa: 236.013201 ms.

Kjoering:4, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 27.755575 ms.
Max verdi sekvensiell i a:99989305, paa: 223.535073 ms.


Median sequential time:236.013201, median parallel time:27.755575,
 n=  100000000, Speedup:  8.59
```

n= 100 mill

JIT-kompilering +optimalisering

Søppel-tømming

Hva betyr dette for tidsmålingene



- Første gangen vi gjører er tiden vi måler en sum av:
  - Først litt interpretering av bytekode
  - Så oversetting(kompilering) av hyppig brukte metoder til maskinkode
  - kjøring av resten av programmet dels i maskinkode.
- Andre gang vi kjører, kan følgende skje:
  - JVM finner at noen av maskinkompilerte metodene våre må optimaliseres ytterligere
  - Kjøretiden synker ytterligere
- Tredje gang er som oftest optimaliseringa ferdig, men ytterligere optimalisering kan bli gjort
- Tidtakningen vår må endres !
- Vi kjører det sekvensielle og parallelle programmet f.eks 9 ganger i en løkke , noterer alle kjøretider i to arrayer som så sorteres og vi velger medianverdien = a[(a.length-1)/2]
- Du får aldri samme svaret to ganger – mye variasjon !!

FinnMax 3 ulike kjøringer (samme parametre , varierer antall tråder: 8, 16, 4)

Uke2>java FinnM 1000000 9
Kjøring:0**, ant kjerner:8, antTråder:8**
Max verdi parallell i a:999216, paa: 23.860968 ms.
Max verdi sekvensiell i a:999216, paa: 3.468803 ms.

Kjøring:1, ant kjerner:8, antTråder:8
Max verdi parallell i a:999216, paa: 0.311465 ms.
Max verdi sekvensiell i a:999216, paa: 0.549437 ms.
.....
Kjøring:8, ant kjerner:8, antTråder:8
Max verdi parallell i a:999216, paa: 0.422752 ms.
Max verdi sekvensiell i a:999216, paa: 0.532639 ms.

Median sequential time:0.52004,
median parallel time:0.429051,
Speedup: **1.26,** n = 1000000

Uke2>java FinnM 1000000 9
Kjøring:0, **ant kjerner:8, antTråder:16**
Max verdi parallell i a:999216, paa: 18.808946 ms.
Max verdi sekvensiell i a:999216, paa: 3.558043 ms.

Kjøring:1, ant kjerner:8, antTråder:16
Max verdi parallell i a:999216, paa: 1.847439 ms.
Max verdi sekvensiell i a:999216, paa: 0.453898 ms.
.....
Kjøring:8, ant kjerner:8, antTråder:16
Max verdi parallell i a:999216, paa: 0.502542 ms.
Max verdi sekvensiell i a:999216, paa: 0.471396 ms.

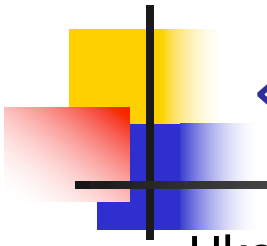
Median sequential time:0.509891,
median parallel time:0.646726,
Speedup: **0.90,** n = 1000000

Uke2>java FinnM 1000000 9
Kjøring:0, **ant kjerner:8, antTråder:4**
Max verdi parallell i a:999216, paa: 16.154151 ms.
Max verdi sekvensiell i a:999216, paa: 3.75507 ms.

Kjøring:1, ant kjerner:8, antTråder:4
Max verdi parallell i a:999216, paa: 1.280854 ms.
Max verdi sekvensiell i a:999216, paa: 0.520741 ms.

Kjøring:2, ant kjerner:8, antTråder:4
Max verdi parallell i a:999216, paa: 0.557136 ms.
Max verdi sekvensiell i a:999216, paa: 0.509191 ms.
.....
Kjøring:8, ant kjerner:8, antTråder:4
Max verdi parallell i a:999216, paa: 0.628527 ms.
Max verdi sekvensiell i a:999216, paa: 0.52354 ms.

Median sequential time:0.520741, median parallel time:0.628527,
Speedup: **0.88**, n = 1000000



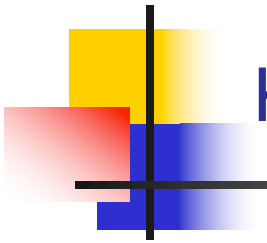
«Aldri» samme resultatet to ganger

Uke2>java FinnM 1000000 9
ant kjerner:8, antTråder:8, n = 1mill

Med antall kjøringer for median = 9
1) Speedup: **0.68**, n = 1000000
2) Speedup: 0.96, n = 1000000
3) Speedup: 0.84, n = 1000000
4) Speedup: 0.71, n = 1000000
5) Speedup: 1.06, n = 1000000
6) Speedup: 1.26**,** n = 1000000

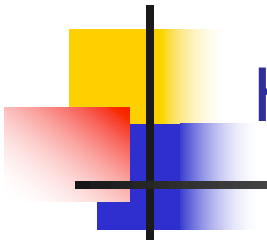
Med antall kjøringer for median = 21
7) Speedup: 1.00, n = 1000000
8) Speedup: 0.84, n = 1000000
9) Speedup: 0.88, n = 1000000
10) Speedup: **1.75**, n = 1000000
11) Speedup: 0.87, n = 1000000
12) Speedup: 1.11, n = 1000000
13) Speedup: 1.03, n = 1000000

Konklusjon på JIT-kompilering



- JIT-kompilering kan skrues av med >java –Xint MittProg ..
  - Brukes bare for debugging
- JIT kompilering kan gi 10 til 30 ganger så rask eksekvering for liten n (en god del mer for stor n)
- Første, andre (og tredje) kjøring er tidsmessig sterkt misvisende
- Vi må:
  - Kjøre programmet i en løkke f.eks 9 (eller 7 eller 11) ganger
  - Legge tidene i hver sin array (sekvensielt og parallell tid)
  - Sortere arrayene
  - Ta ut medianen (element (length-1)/2), som blir vår tidsmåling

```
import java.util.concurrent.*;
import java.util.*;
class Problem2 { int [] fellesData  ; // dette er felles, delte data for alle trådene
      double [] tidene ;
      int ant, svar;
     public static void main(String [] args) {
            ( new Problem()).utfoer(args);
      }
      void utfoer (String [] args) {
               ant = new Integer(args[0]);
               fellesData = new int [ant];
               tidene = new double[9];
               for (int m = 0; m <9; m++) {
                              long tid = System.nanoTime();
                              Thread t = new Thread(new Arbeider());
                               t.start();
                               try{t.join();}catch (Exception e) {return;}
                              tidene[m] = (System.nanoTime() -tid)/1000000.0;
                              System.out.println("Tid for "+m + ", tråd:"+tidene[m]+"ms");
               }
               Arrays.sort(tidene);
               System.out.println("Median med svar:"+svar+", for trådene:"+tidene[(tidene.length-1)/2]+" ms");
      } // end utfoer

      class Arbeider implements Runnable {
         int i,lokalData;    // dette er lokale data for hver tråd
                  public void run() {     int sum =0;
                                          for (int i = 0; i < ant; i++) sum +=fellesData[i];
                                          svar =sum;
                  }
    } // end indre klasse Arbeider
} // end class Problem
```



Hva med operativsystemet:

- Linux og Windows har om lag like rask implementasjon av Java og trådprogrammering,
- Dag Langmyhr testet to helt like maskiner med hhv. Linux og Windows, og resultatene tidsmessig (medianer) var nesten helt like, men
  - Ulike maskiner som Ifis store servere (diamant, safir,..) har en annen Linux og en noe langsommere ytelse for korte, trådbaserte programmer.

Hva med søppeltømming – garbage collection:

- Søppeltømming (=opprydding i lageret og fjerning av objekter vi ikke lenger kan bruke) kan slå til når som helst under kjøring:

```
Kjøring:2, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.35 msek. , nanosek/n:   35.07
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.36

Kjøring:3, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.57 msek. , nanosek/n:   56.87
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    0.66

Kjøring:4, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.43 msek. , nanosek/n:   43.47
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.33

Kjøring:5, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.49 msek. , nanosek/n:   49.20
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.36
```

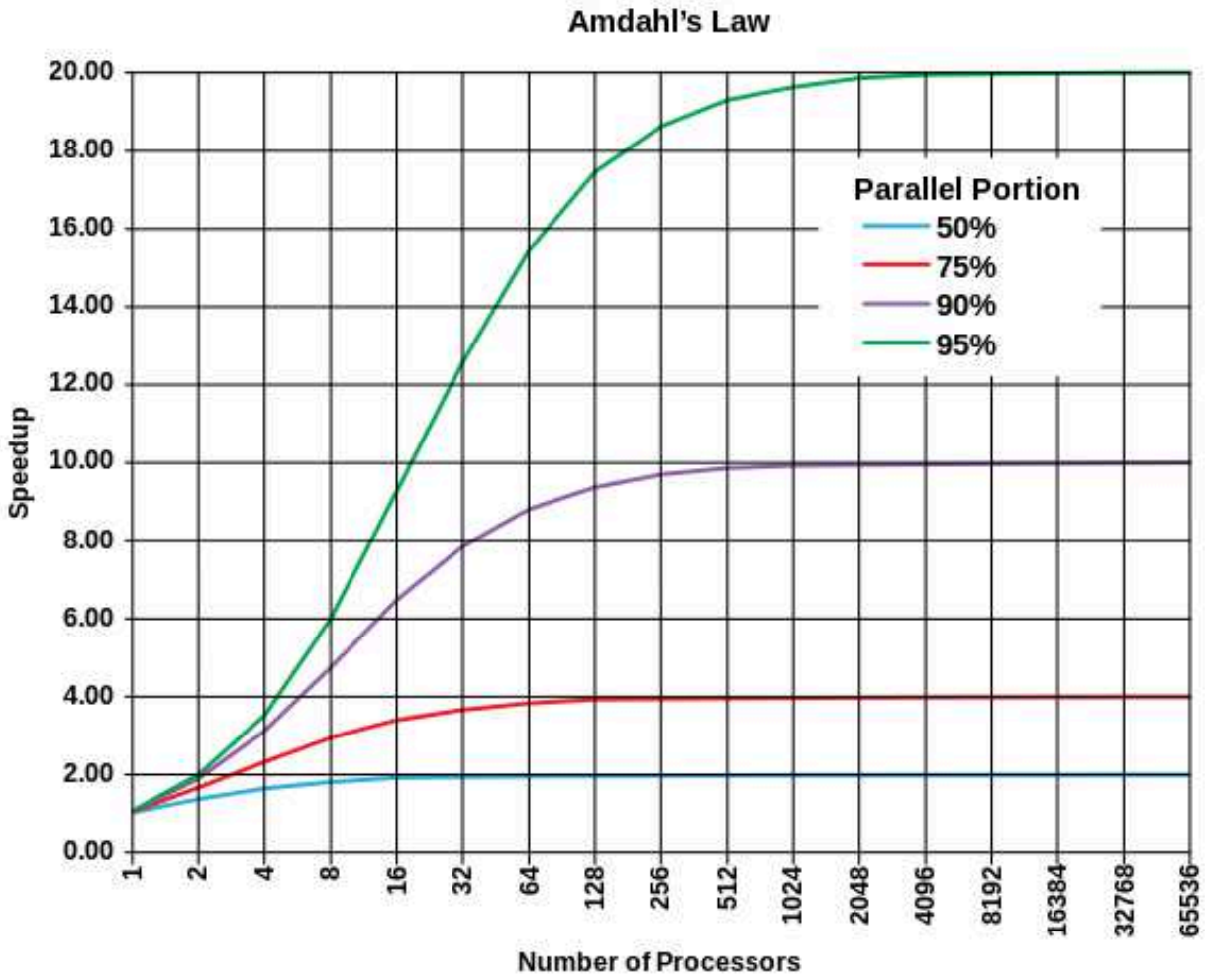
Amdahl lov for parallelle beregninger

- Amdahl lov: Har du **seq** andel sekvensiell kode og da **p** andel parallelliserbar kode i et parallelt program, **seq+p=1**, er den største speedup S du kan få med k kjerner:

$$S = \frac{tid(sekvensiell)}{tid\ (parallell)} = \frac{1}{seq+p/k} = \frac{1}{1-p+p/k}$$

- Når k $\rightarrow \infty$, vil S $\rightarrow \frac{1}{1-p}$.
- Er p=0.9, så er S ≤ 10 uansett hvor mange kjerner du har, og har du 'bare' 50, er S = $\frac{1}{1-0.9+0,9/50}$ = 8,5.
- Amdahls lov er pessimistisk- antar fast størrelse på problemet
- «Hvis du først har brukt 10% av tida på en sekvensiell del, så kan resten av programmet ikke gå fortere enn 0.00 sekunder uansett hvor mange prosessorer du bruker på det. Dvs. at speedup ≤ 10»

Amdahl for ulike verdier av p



Gustafsons lov for parallelle beregninger

- La S være speedup, P antall kjerner og α andel sekvensiell kode, så er:

$$S(P) = P - \alpha\,(P-1)$$

Fordi:

Parallell løsning: $a + b\ (a = sekvensiell\ tid, b = parallel\ tid)$

Sekvensiell løsning : $a + P * b$

Speedup er da:

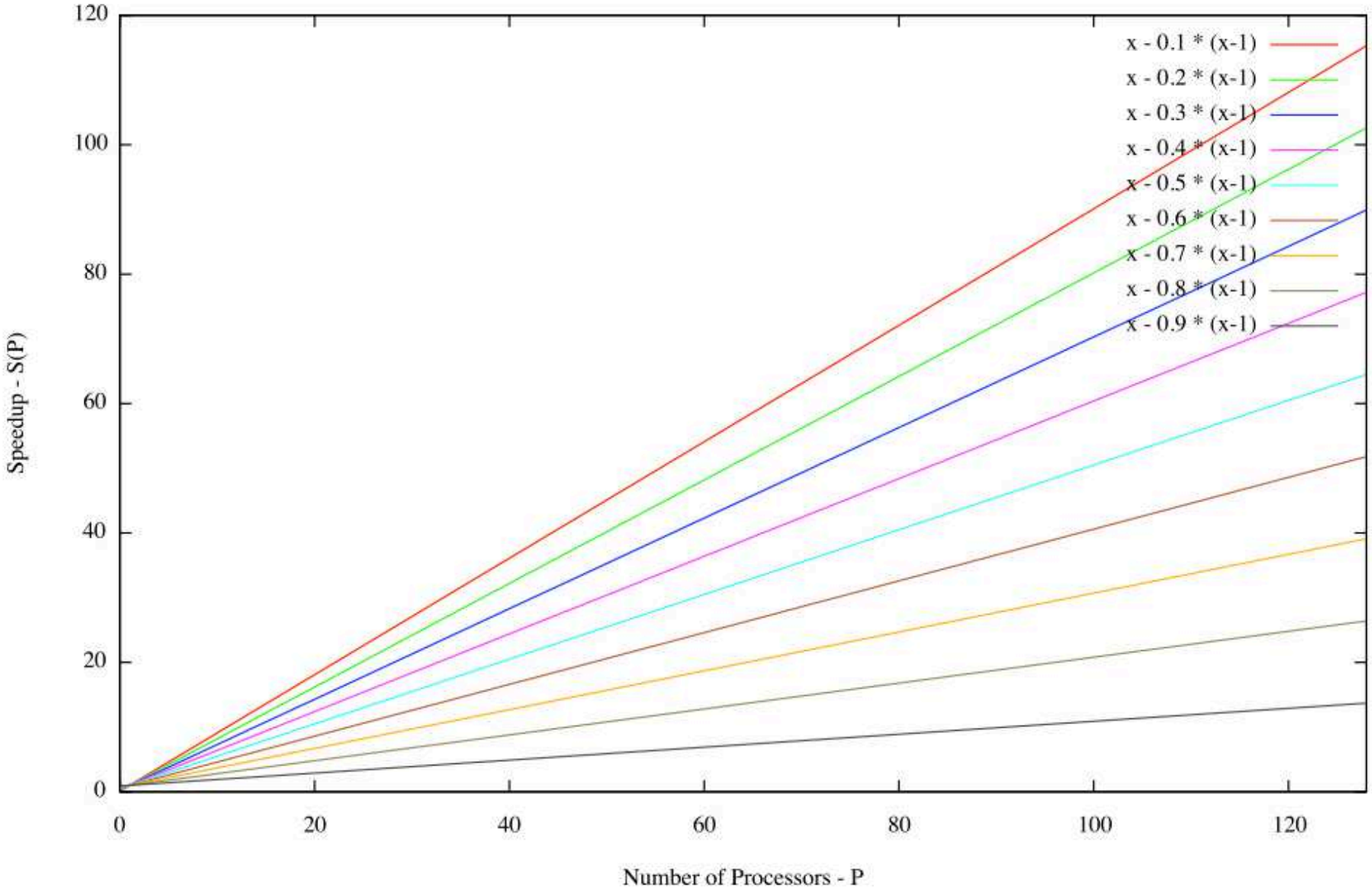
$(a + P * b)/(a + b\ )$, og definerer $\alpha = \frac{a}{a+b}$, så er:

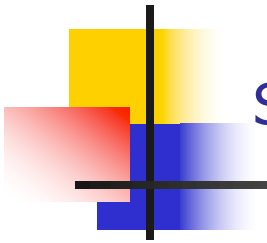
$$S(P) = \alpha + P * (1-\alpha) = P - \alpha(P-1)$$

- Gustavson er mer optimistisk enn Amdahl, gir høvere speedup fordi han antar at med flere maskiner vil vi øke størrelsen på problemet.

- «Hvis du tidligere brukte 1 time på å løse et problem sekvensielt, vil du nå også bruke 1 time på å løse et større, mer nøyaktig problem parallelt da med større speedup– for eksempel i meteorologi»

Gustafson's Law: $S(x) = x - \alpha(x - 1),$



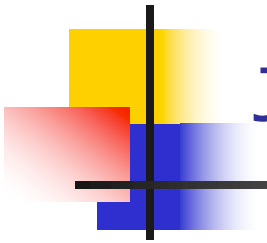


Sammenligning av Amdahl og Gustafson + egne betraktninger

- Amdahl antar at oppgaven er fast av en gitt lengde(n)
- Gustafson antar at du med parallelle maskiner løser større problemer (større n) og da blir den sekvensielle delen mindre.
- Min betraktning:
  1. En algoritme består av noen sekvensielle deler og noen parallelliserbare deler.
  2. Hvis de sekvensielle delene har lavere orden – f.eks O(log n), men de parallelle har en større orden – eks O(n) så vil de parallelle delene bli en stadig større del av kjøretida hvis n øker (Gustafson)
  3. Hvis de parallelle og sekvensielle delene har samme orden, vil et større problem ha samme sekvensielle andel som et mindre problem (Amdahl).
  4. I tillegg kommer alltid et fast overhead på å starte k tråder (1-4 ms.)

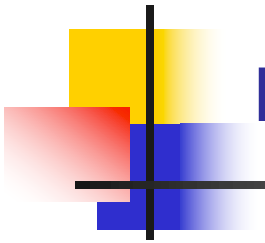
  Algoritmer vi skal jobbe med er mer av type 2 (Gustafson) enn type 3(Amdahl) men vi har alltid overhead, så små problemer løses best sekvensielt.

Konklusjon: For store problemer bør vi ha håp om å skalere nær lineært med antall kjerner hvis ikke vi får kø og forsinkelser når alle kjernene skal lese/skrive i lageret.



Java Measurement Harness

- Seperate slides jmh.pttx



Eval

- I would like to have a quick evaluation ☺

Java Microbench Harness

Slides from: Magnus S. Espeland (magnuesp@ifi.uio.no)

What is JMH and Linux Perf?

Java Microbench Harness is a Java harness for building, running, and analysing nano/micro/milli/macro benchmarks written in Java and other languages targeting the JVM.

http://openjdk.java.net/projects/code-tools/jmh/

perf: it can instrument CPU performance counters, tracepoints, kprobes, and uprobes (dynamic tracing). It is capable of lightweight profiling. It is also included in the Linux kernel, under tools/perf, and is frequently updated and enhanced.

https://perf.wiki.kernel.org/index.php/Main_Page

So, what are they good for?

JMH can be used to benchmark your Java application, and even different algorithms in it. It takes care of warmup (getting the JVM with JIT into “steady state”), and executing each of your algorithms many times.

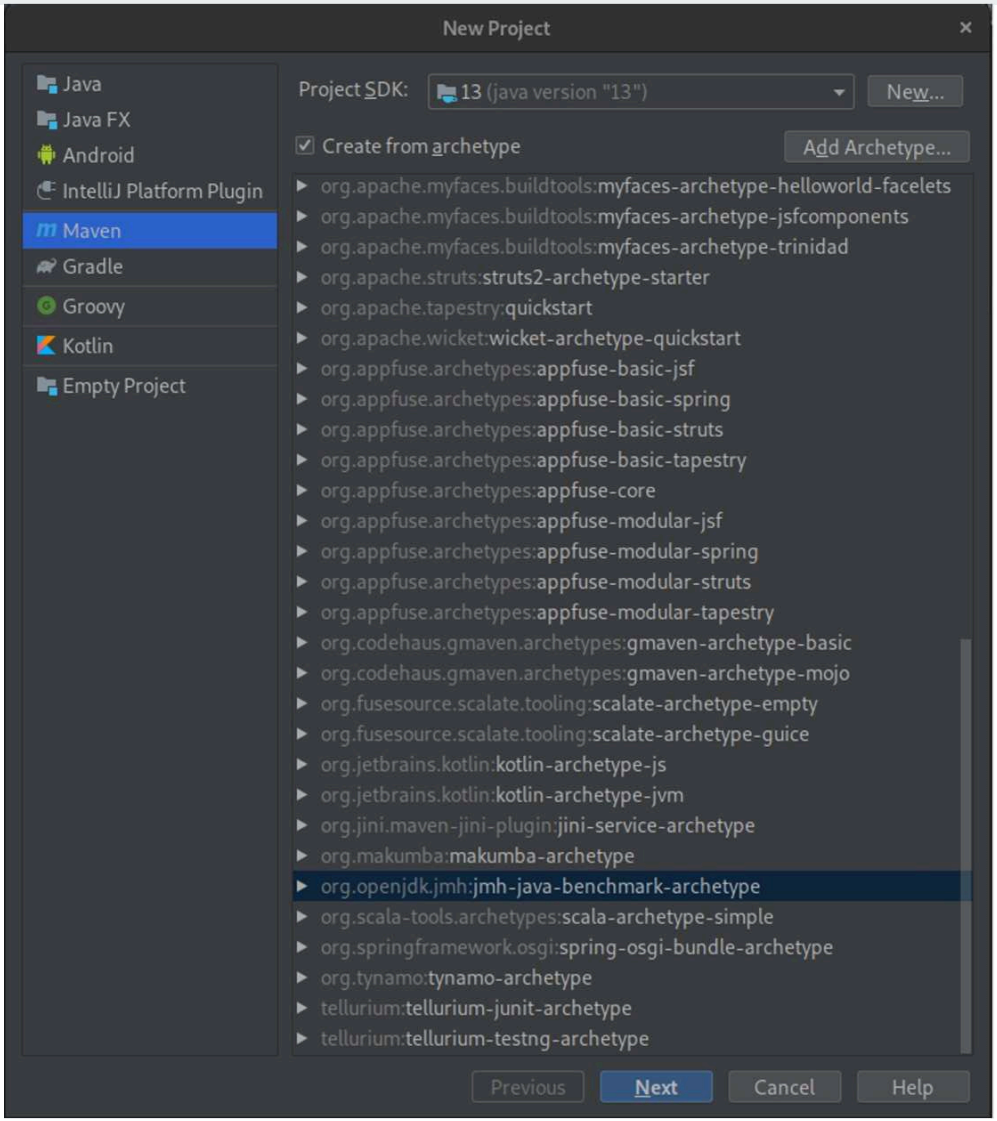
This scientific approach gives empiric data as to which algorithm performs better.

Together with *perf* it can show important details as to what happens in the CPU during execution.

(but, you can get interesting results without *perf* as well!)

Creating the project …

```
mvn archetype:generate \
        -DinteractiveMode=false \
        -DarchetypeGroupId=org.openjdk.jmh \
        -DarchetypeArtifactId=jmh-java-benchmark-archetype \
        -DgroupId=no.uio.ifi.yourusername \
        -DartifactId=demo-benchmark \
        -Dversion=1.0
```



Now show me the code!

```java
import org.openjdk.jmh.annotations.Benchmark;
import org.openjdk.jmh.infra.Blackhole;

public class MyBenchmark {
    @Benchmark
    public void naiveSingleSieve(Blackhole blackhole) {
        SieveSuperClass sieve = new NaiveSingleSieve();
        int[] primes = sieve.generatePrimes( max: 2_000_000_000);

        blackhole.consume(primes);
    }
}
```

```
$ mvn clean install && java -jar target/benchmarks.jar -prof perf
```

And the output? (matrix multipl., oblig 2)

```
Secondary result "org.openjdk.jmh.samples.MyBenchmark.colFirst:·perf":
Perf stats:
--------------------------------------------------

        13 753,76 msec task-clock:u              #    0,697 CPUs utilized
                0      context-switches:u        #    0,000 K/sec
                0      cpu-migrations:u          #    0,000 K/sec
              249      page-faults:u             #    0,018 K/sec
   57 470 280 173      cycles:u                  #    4,179 GHz                      (33,32%)
       25 345 146      stalled-cycles-frontend:u #    0,04% frontend cycles idle     (33,37%)
   47 228 301 463      stalled-cycles-backend:u  #   82,18% backend cycles idle      (33,43%)
   50 502 840 834      instructions:u            #    0,88  insn per cycle
                                                 #    0,94  stalled cycles per insn  (33,46%)
    8 659 401 617      branches:u                #  629,603 M/sec                    (33,48%)
        1 566 317      branch-misses:u           #    0,02% of all branches          (33,43%)
   29 846 953 344      L1-dcache-loads:u         # 2170,095 M/sec                    (33,38%)
    3 017 742 965      L1-dcache-load-misses:u   #   10,11% of all L1-dcache hits    (33,32%)
  <not supported>      LLC-loads:u
  <not supported>      LLC-load-misses:u
      212 672 259      L1-icache-loads:u         #   15,463 M/sec                    (33,28%)
        1 688 170      L1-icache-load-misses:u   #    0,79% of all L1-icache hits    (33,26%)
    2 055 024 465      dTLB-loads:u              #  149,416 M/sec                    (33,25%)
    2 000 591 054      dTLB-load-misses:u        #   97,35% of all dTLB cache hits   (33,25%)
            5 800      iTLB-loads:u              #    0,422 K/sec                    (33,25%)
            1 656      iTLB-load-misses:u        #   28,55% of all iTLB cache hits   (33,26%)
      110 928 694      L1-dcache-prefetches:u    #    8,065 M/sec                    (33,26%)
  <not supported>      L1-dcache-prefetch-misses:u

      19,738682523 seconds time elapsed

      19,925289000 seconds user
       0,585981000 seconds sys
```

Column first:

```
L1-dcache-load-misses:u   #   10,11% of
all L1-dcache hits

0,88  insn per cycle

0,94  stalled cycles per insn
```

Row first:

```
L1-dcache-load-misses:u   #    0,34% of
all L1-dcache hits

2,04  insn per cycl

0,37  stalled cycles per insn
```

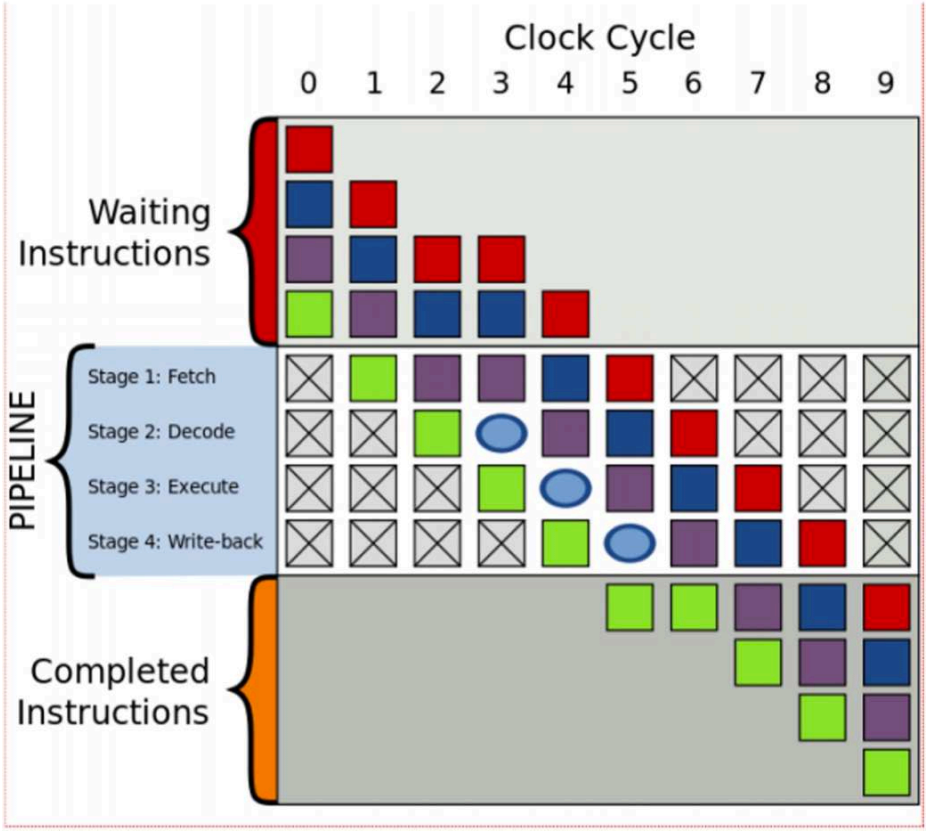
What is stalling?

- Front-end is stage **1 and 2** of of the pipeline
- Back-end is stage **3 and 4** of the pipeline

In simple terms:

Stalling in the back-end means the CPU cannot execute the instruction. Why? Mostly because it has to wait for the data to arrive from memory.

(The slow speed of light….)

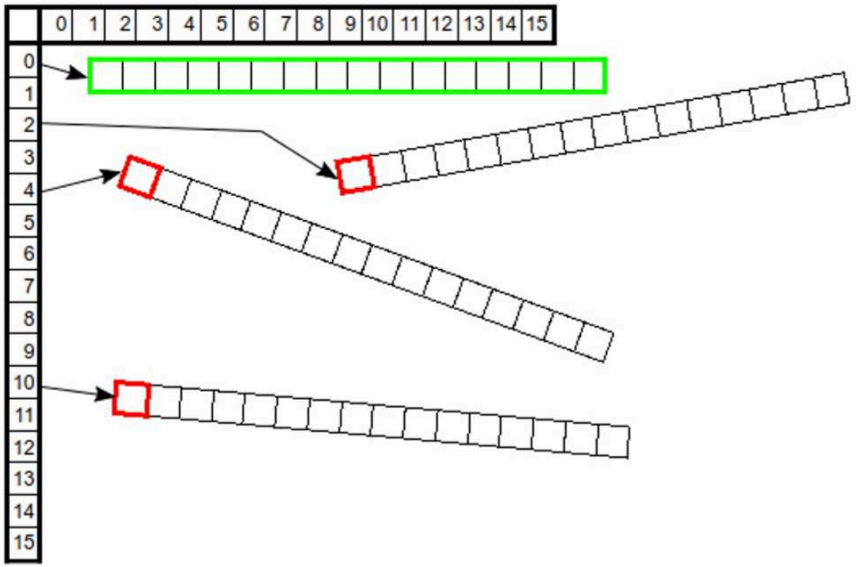


I don’t get it...

When doing matrix multiplication by the column first, the CPU can only buy one beer at a time (red squares).

When doing it row first, it can get a six-pack (actually a sixteen-pack, a **cache line**) each time it goes to the store (or RAM) via to the **pre-fetch** / readahead mechanism (green rectangle).

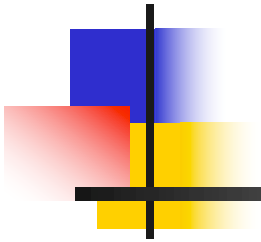
(Java’s multi-dim arrays are stored with references to the rows, not as one long memory segment.)



Summary

JMH (and perf) helps you understand the fine details of how your code behaves in the CPU. Very useful for tuning critical parts of your applications.

Experiment with it and use it in your reports, if you want. If you're already struggling to get the obligs done, don't worry about it.

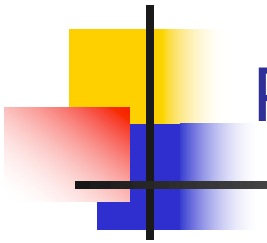


IN3030 L07v24 – Prime Numbers, Timing

Eric Jul
Programming Technology Group
Department of Informatics
University of Oslo
2024-02-29

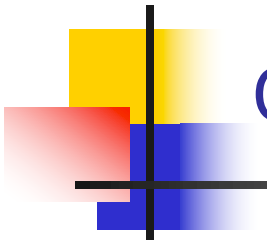
Review L06v23

I. Prime Numbers, Erasthophenes sieve
II. Oblig Prime Numbers



Plan for L07v24

I. Prime Numbers, review
II. Oblig Prime Numbers -- Review
III. Tidtagning
    I. JIT compilation
    II. Operativsystem?
    III. Søppel/Garbage Collection
IV. JMH
V. Amdahl
VI. Gustavson



Om primtall

- Primtall og faktorisering av ikke-primtall.
- Et primtall er:
  Et heltall som bare lar seg dividere med 1 og seg selv.
  - 1 er ikke et primtall (det mente mange på 1700-tallet, og noen mener det fortsatt)

Om primtall og faktorisering av heltall

- Ethvert heltall N > 1 lar seg faktorisere som et produkt av primtall:
  - $N = p_1 * p_2 * p_3 * \ldots\ldots\ldots * p_k$
  - Denne faktoringen er entydig (pånær rækkefølge)
  - gjøres entydig hvis tall i faktoriseringen sorteres
  - Hvis det bare er ett tall i denne faktoriseringen, er N selv et primtall

2 måter å lage primtall

Ønsker at finne alle primtal $p_6 < N$

- Dividere alle tall < N med alle tall < N
  - Divisjonsmetoden
  - Bare oddetall (2 spesiel)
  - Bare opp til $\sqrt{N}$ -- hvorfor?
  - Bare primtall opp til $\sqrt{N}$ -- hvorfor?

- Lage en tabell over alle de primtallene vi trenger
  - Eratosthene sil

Hvad er raskest?

- A) Med Eratosthenes sil:

```
Z:\INF2440Para\Primtall>java PrimtallESil 2000000000
 max primtall m:2000000000
Genererte alle primtall <= 2000000000  paa     18 949 millisek
med Eratosthenes sil  og det største primtallet er:1999999973
```

- Med gjentatte divisjoner

```
Z:\INF2440Para\Primtall>java PrimtallDiv 2000000000
Genererte alle primtall <=2000000000  paa   1 577 302  millisek med
 divisjon , og det største primtallet er:1999999973
```

- Å lage primtallene p og finne dem ved divisjon (del på alle oddetall < SQRT(p), p = 3,5,7,..) er ca. 100 ganger langsommere enn Eratosthenes avkryssings-tabell (kalt Eratosthenes sil).

Om primtall og faktorisering av heltall

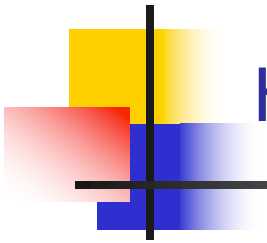
- Ethvert heltall N > 1 lar seg faktorisere som et produkt av primtall:
  - $N = p_1 * p_2 * p_3 * \ldots\ldots\ldots * p_k$
  - Denne faktoringen er entydig (pånær rækkefølge)
  - gjøres entydig hvis tall i faktoriseringen sorteres
  - Hvis det bare er ett tall i denne faktoriseringen, er N selv et primtall
- Eksempel: faktorisering av 532

Å lage og lagre primtall (Erotosthenes sil)

- Som en bit-tabell (1- betyr primtall, 0-betyr ikke-primtall)
  - Påfunnet i jernalderen av Eratosthenes (ca. 200 f.kr)
  - Man skal finne alle primtall < M
  - Man finner da de første primtallene og krysser av alle multipla av disse (N.B. dette forbedres/endres senere):
    - Eks: 3 er et primtall, da krysses 6, 9,12,15,.. Av fordi de alle er ett-eller-annet-tall (1,2,3,4,5,..) ganger 3 og følgelig selv ikke er et primtall. 6=2*3, 9 = 3*3,
      12 =2*2*3, 15 = 5*3, ..osv
    - De tallene som *ikke blir* krysset av, når vi har krysset av for alle primtallene vi har, er primtallene
- Vi finner 5 som et primtall fordi, etter at vi har krysset av for 3, finner første ikke-avkryssete tall: 5, som da er et primtall (og som vi så krysser av for, …finner så 7 osv)

Litt mer om Eratothenes sil

- Vi representerer ikke partallene på den tallinja som det krysses av på fordi vi vet at 2 er et primtall (det første) og at alle andre partall er ikke-primtall.
- Har vi funnet et nytt primtall p, for eksempel. 5, starter vi avkryssingen for dette primtallet først for tallet p*p (i eksempelet: 25), men etter det krysses det av for p*p+2p, p*p+4p,.. (i eksempelet 35,45,55,…osv.). Grunnen til at vi kan starte på p*p er at alle andre tall t < p*p slik det krysses av i for eksempel Wikipedia-artikkelen har allerede blitt krysset av andre primtall < p.
- Det betyr at for å krysse av og finne alle primtall < N , behøver vi bare å krysse av på denne måten for alle primtall p ≤ sqrt(N). Dette sparer svært mye tid.



Hvordan representeres tallene?

- Kun oddetall – 2 kjenner vi!
- Array of Boolean?
  - Problem: 32 bit per primtall
- Kompakter bitarray
    - Kun 1 bit per oddetall

Faktorisering av et tall M i sine primtallsfaktorer

- Vi har laget og lagret ved hjelp av Erotosthanes sil alle (unntatt 2) primtall < N i en bit-array over alle odde-tallene.
  - 1 = primtall, 0=ikke-primtall
  - Vi har krysset ut de som ikke er primtall
- Hvordan skal vi så bruke dette til å faktorisere et tall M < N*N ?
- **Svar:** Divider M med alle primtall $p_i < \sqrt{M}$ (p_i =2,3,5,..), og hver gang en slik divisjon M%p_i ==0, så er p_i en av faktorene til M. Vi forsetter så med å faktorisere ett mindre tall M'=M/p_i.
- Faktoriseringen av M = p_i *..* p_k er da produktet av alle de primtall som dividerer M uten rest.
- HUSK at en p_i kan forekommer flere ganger i svaret. eks: 20= 2*2*5, 81 = 3*3*3*3, osv
- Finner vi ingen faktorisering av M, dvs. ingen $p_i \leq \sqrt{M}$ som dividerer M med rest == 0, så er M selv et primtall.

Hvordan parallellisere faktorisering ?

1. Denne uka viktig å få på plass en effektiv sekvensiell løsning med om lag disse kjøretidene for N = 2 mill:

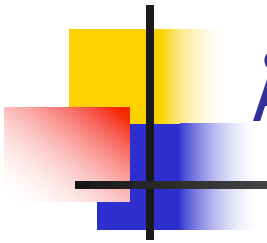
```
M:\INF2440Para\Primtall>java PrimtallESil 2000000
 max primtall m:2 000 000
Genererte primtall <= 2000000  paa        15.56 millisek
med Eratosthenes sil (  0.00004182 millisek/primtall)
……
3999998764380 = 2*2*3*5*103*647248991
3999998764381 = 37*108108074713
3999998764382 = 2*271*457*1931*8363
3999998764383 = 3*19*47*1493093977
3999998764384 = 2*2*2*2*2*7*313*1033*55229
3999998764385 = 5*13*59951*1026479
3999998764386 = 2*3*3*31*71*100964177
3999998764387 = 1163*1879*1830431
3999998764388 = 2*2*11*11*17*23*293*72139
 100 faktoriseringer beregnet paa:     422.0307ms -
 dvs:      4.2203ms. per faktorisering
```

Faktorisering av store tall med 18-19 desimale sifre

```
Uke5>java PrimtallESil 2140000000
max primtall m:2 140 000 000
bitArr.length:133 750 001
Genererte primtall <= 2 140 000 000  paa    11030.36 millisek
med Eratosthenes sil ( 0.00010530 millisec/primtall)
 antall primtall < 2 140 000 000 er: 104 748 779, dvs:  4.89% ,
 og det største primtallet er: 2 139 999 977

4 579 599 999 999 999 900 = 2*2*3*5*5*967*3673*19421*221303
4 579 599 999 999 999 901 = 457959999999999901
4 579 599 999 999 999 902 = 2*228979999999999951
4 579 599 999 999 999 903 = 3*31*13188589*3733758839
4 579 599 999 999 999 904 = 2*2*2*2*2*19*71*106087842846553
4 579 599 999 999 999 905 = 5*7*1308457142857142 83
.............
4 579 599 999 999 999 997 = 11*41632727272727272 7
4 579 599 999 999 999 998 = 2*121081*18911307306679
4 579 599 999 999 999 999 = 3*17*19*6625387*713333333
100 faktoriseringer beregnet paa:  333481.4427ms
 dvs:   3334.8144ms. per faktorisering

 largestLongFactorizedSafe: 4 579 599 841 640 001 173= 2139999949*2139999977
```

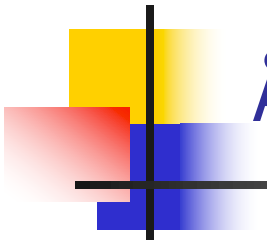
Om å parallelliser et problem

- Utgangspunkt: Vi har en sekvensiell effektiv og riktig sekvensiell algoritme som løser problemet.
- Vi kan dele opp både koden og data (hver for seg?)
- Vanligst å dele opp data
  - Som oftest deler vi opp data, og lar 'hele' koden virke på hver av disse data-delene (en del til hver tråd).
  - Eks: Matriser
    - radvis eller kolonnevis oppdeling av C til hver tråd
    - Omforme data slik at de passer bedre i cachene (transponere B)
  - Rekursiv oppdeling av data ('lett')
    - Eks: Quicksort
- Også mulig å dele opp koden:
  - Alternativ Oblig3 i INF1000: Beregning av Pi (3,1415..) med 17 000 sifre med tre ArcTan-rekker
  - Primtalls-faktorisering av store tall N for kodebrekking:
    - $N = p_1 * p_2$



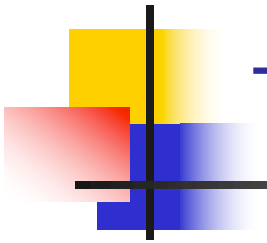
Å dele opp algoritmen

- Koden består en eller flere steg; som oftest i form av en eller flere samlinger av løkker (som er enkle, doble, triple..)
- Vi vil parallellisere med k tråder, og hver slikt steg vil få hver sin parallellisering med en CyclickBarrier-synkronisering mellom hver av disse delene + en synkronisert avslutning (join(), ..).
- Eks:
  - finnMax – hadde ett slikt steg: for (int i = 0 …n-1) -løkke
  - MatriseMult hadde ett slikt steg med trippel-løkke
  - Flere steg mulig: Eksempel Radix sort (nevnes senere)



Å dele opp data – del 2

- For å planlegge parallellisering av ett slikt steg må vi finne:
  - Hvilke data i problemet er lokale i hver tråd?
  - Hvilke data i problemet er felles/delt mellom trådene?
- Viktig for effektiv parallell kode.
  - Hvordan deler vi opp felles data (om mulig)
  - Kan hver tråd beregne hver sin egen, disjunkte del av data
  - Færrest mulig synkroniseringer (de tar 'mye' tid)

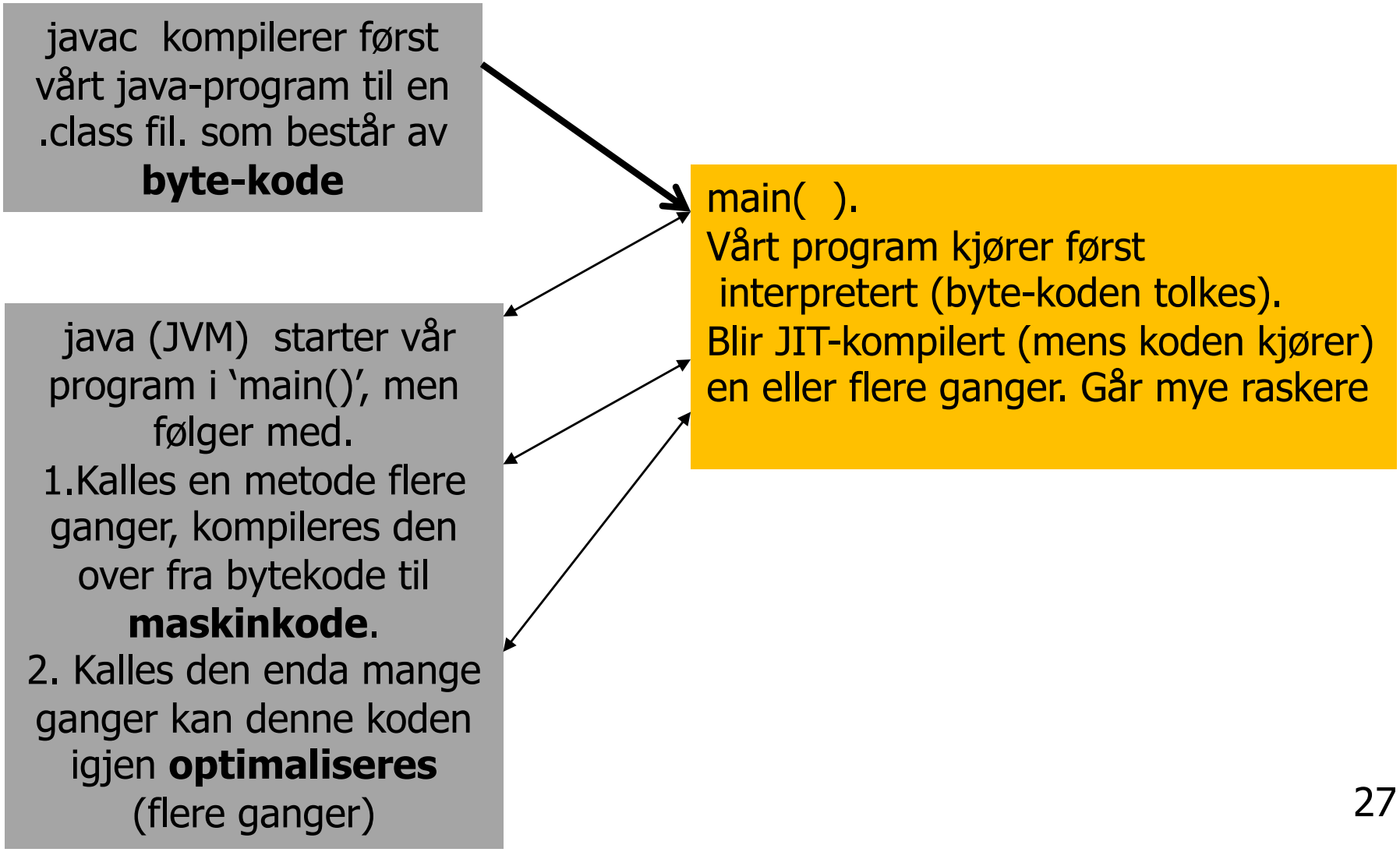


Tidtagning

- JIT –kompilering
  - Hvor mye betyr det egentlig
- Operativsystemet (Windows eller Linux)
  - Er de like raske?
- Søppeltømming i Java
  - Skjer under kjøring (med i tidene)

Tidsmålinger og JIT (Just In Time) -kompilering

- Tilbake til kompileringen av et Java-program:



Optimaliserng – ett eksempel

Original kode

```
class A {
  B b;
  public void newMethod() {
    y = b.get();
    ...do stuff...
    z = b.get();
    sum = y + z;
  }
}
class B {
   int value;
   final int get() {
      return value;
   }
}
```

1) Inline get

```
public void
newMethod() {
    y = b.value;
    ...do stuff...
    z = b.value;
    sum = y + z;
}
```

2) Fjern overflødige les

```
public  void
newMethod() {
    y = b.value;
    ...do stuff...
    z = y;
    sum = y + z;
}
```

3) Fjern overflødige variable

```
public  void
newMethod() {
    y = b.value;
    ...do stuff...
    y = y;
    sum = y + y;
}
```

4) Fjern død kode

```
public  void
newMethod() {
    y = b.value;
    ...do stuff...
    sum = y + y;
}
```

Mediantider for finnMax fra ukeoppgavene:

n= 10 000

Vi ser at kjøretidene (sekv og para) synker dramatisk fra 1.ste til neste kjøring.
Pga JIT-optimalisering

```
M:\INF2440Para\FinnMax>java FinnMaxMulti 10000 7

Kjøring:0, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     6.30 msek. , nanosek/n:  630.46
Max sekv = a:9853, paa:     0.28 msek. , nanosek/n:   28.38

Kjøring:1, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.57 msek. , nanosek/n:   56.87
Max sekv = a:9853, paa:     0.27 msek. , nanosek/n:   26.95

Kjøring:2, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.35 msek. , nanosek/n:   35.07
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.36

Kjøring:3, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.57 msek. , nanosek/n:   56.87
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    0.66

Kjøring:4, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.43 msek. , nanosek/n:   43.47
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.33

Kjøring:5, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.49 msek. , nanosek/n:   49.20
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.36

Kjøring:6, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.48 msek. , nanosek/n:   47.84
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.43

Median seq time:   0.014, median para time:   0.569,
 Speedup:    0.03, n = 10000
```

```
M:\INF2440Para\FinnMax>java FinnMaxMulti 10000000 5

Kjøring:0, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    14.08 msek. , nanosek/n:    1.41
Max sekv = a:9999216, paa:     6.98 msek. , nanosek/n:    0.70

Kjøring:1, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     3.17 msek. , nanosek/n:    0.32
Max sekv = a:9999216, paa:     4.75 msek. , nanosek/n:    0.47

Kjøring:2, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     2.79 msek. , nanosek/n:    0.28
Max sekv = a:9999216, paa:     5.04 msek. , nanosek/n:    0.50

Kjøring:3, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     2.87 msek. , nanosek/n:    0.29
Max sekv = a:9999216, paa:     5.05 msek. , nanosek/n:    0.51

Kjøring:4, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     2.92 msek. , nanosek/n:    0.29
Max sekv = a:9999216, paa:     5.03 msek. , nanosek/n:    0.50

Median seq time:   5.052, median para time:   3.173,
 Speedup:    1.59, n = 10 000 000
```

n= 10 mill

```
M:\INF2440Para\FinnMax>java -Xint FinnMaxMulti 10000000 5

Kjøring:0, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    67.24 msek. , nanosek/n:    6.72
Max sekv = a:9999216, paa:   179.40 msek. , nanosek/n:   17.94

Kjøring:1, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    64.00 msek. , nanosek/n:    6.40
Max sekv = a:9999216, paa:   175.12 msek. , nanosek/n:   17.51

Kjøring:2, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    51.42 msek. , nanosek/n:    5.14
Max sekv = a:9999216, paa:   176.23 msek. , nanosek/n:   17.62

Kjøring:3, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    64.95 msek. , nanosek/n:    6.49
Max sekv = a:9999216, paa:   173.17 msek. , nanosek/n:   17.32

Kjøring:4, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    60.11 msek. , nanosek/n:    6.01
Max sekv = a:9999216, paa:   185.84 msek. , nanosek/n:   18.58

Median seq time: 179.403, median para time:  64.950,
 Speedup:    2.76, n = 10 000 000
```

JIT-kompilering avslått :
> java –Xint
…..
n= 10 mill

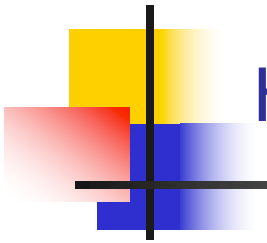
```
M:\INF2440Para\FinnMax>java FinnM 100000000 5
Kjoering:0, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 41.913504 ms.
Max verdi sekvensiell i a:99989305, paa: 238.799921 ms.

Kjoering:1, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 26.78024 ms.
Max verdi sekvensiell i a:99989305, paa: 235.431219 ms.

Kjoering:2, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 27.791271 ms.
Max verdi sekvensiell i a:99989305, paa: 248.066478 ms.

Kjoering:3, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 26.86283 ms.
Max verdi sekvensiell i a:99989305, paa: 236.013201 ms.

Kjoering:4, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 27.755575 ms.
Max verdi sekvensiell i a:99989305, paa: 223.535073 ms.


Median sequential time:236.013201, median parallel time:27.755575,
 n=  100000000, Speedup:  8.59
```

n= 100 mill

JIT-kompilering +optimalisering

Søppel-tømming

Hva betyr dette for tidsmålingene



- Første gangen vi gjører er tiden vi måler en sum av:
  - Først litt interpretering av bytekode
  - Så oversetting(kompilering) av hyppig brukte metoder til maskinkode
  - kjøring av resten av programmet dels i maskinkode.
- Andre gang vi kjører, kan følgende skje:
  - JVM finner at noen av maskinkompilerte metodene våre må optimaliseres ytterligere
  - Kjøretiden synker ytterligere
- Tredje gang er som oftest optimaliseringa ferdig, men ytterligere optimalisering kan bli gjort
- Tidtakningen vår må endres !
- Vi kjører det sekvensielle og parallelle programmet f.eks 9 ganger i en løkke , noterer alle kjøretider i to arrayer som så sorteres og vi velger medianverdien = a[(a.length-1)/2]
- Du får aldri samme svaret to ganger – mye variasjon !!

FinnMax 3 ulike kjøringer (samme parametre , varierer antall tråder: 8, 16, 4)

Uke2>java FinnM 1000000 9
Kjøring:0**, ant kjerner:8, antTråder:8**
Max verdi parallell i a:999216, paa: 23.860968 ms.
Max verdi sekvensiell i a:999216, paa: 3.468803 ms.

Kjøring:1, ant kjerner:8, antTråder:8
Max verdi parallell i a:999216, paa: 0.311465 ms.
Max verdi sekvensiell i a:999216, paa: 0.549437 ms.
.....
Kjøring:8, ant kjerner:8, antTråder:8
Max verdi parallell i a:999216, paa: 0.422752 ms.
Max verdi sekvensiell i a:999216, paa: 0.532639 ms.

Median sequential time:0.52004,
median parallel time:0.429051,
Speedup: **1.26,** n = 1000000

Uke2>java FinnM 1000000 9
Kjøring:0, **ant kjerner:8, antTråder:16**
Max verdi parallell i a:999216, paa: 18.808946 ms.
Max verdi sekvensiell i a:999216, paa: 3.558043 ms.

Kjøring:1, ant kjerner:8, antTråder:16
Max verdi parallell i a:999216, paa: 1.847439 ms.
Max verdi sekvensiell i a:999216, paa: 0.453898 ms.
.....
Kjøring:8, ant kjerner:8, antTråder:16
Max verdi parallell i a:999216, paa: 0.502542 ms.
Max verdi sekvensiell i a:999216, paa: 0.471396 ms.

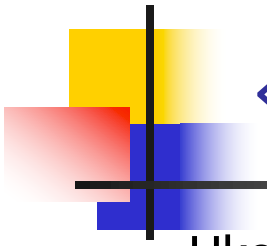
Median sequential time:0.509891,
median parallel time:0.646726,
Speedup: **0.90,** n = 1000000

Uke2>java FinnM 1000000 9
Kjøring:0, **ant kjerner:8, antTråder:4**
Max verdi parallell i a:999216, paa: 16.154151 ms.
Max verdi sekvensiell i a:999216, paa: 3.75507 ms.

Kjøring:1, ant kjerner:8, antTråder:4
Max verdi parallell i a:999216, paa: 1.280854 ms.
Max verdi sekvensiell i a:999216, paa: 0.520741 ms.

Kjøring:2, ant kjerner:8, antTråder:4
Max verdi parallell i a:999216, paa: 0.557136 ms.
Max verdi sekvensiell i a:999216, paa: 0.509191 ms.
.....
Kjøring:8, ant kjerner:8, antTråder:4
Max verdi parallell i a:999216, paa: 0.628527 ms.
Max verdi sekvensiell i a:999216, paa: 0.52354 ms.

Median sequential time:0.520741, median parallel time:0.628527,
Speedup: **0.88**, n = 1000000



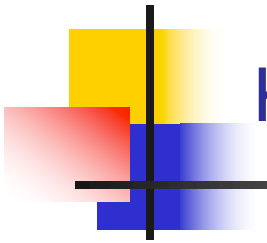
«Aldri» samme resultatet to ganger

Uke2>java FinnM 1000000 9
ant kjerner:8, antTråder:8, n = 1mill

Med antall kjøringer for median = 9
1) Speedup: **0.68**, n = 1000000
2) Speedup: 0.96, n = 1000000
3) Speedup: 0.84, n = 1000000
4) Speedup: 0.71, n = 1000000
5) Speedup: 1.06, n = 1000000
6) Speedup: 1.26**,** n = 1000000

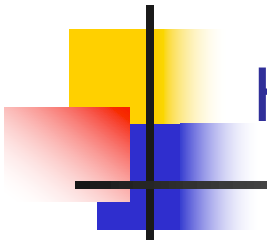
Med antall kjøringer for median = 21
7) Speedup: 1.00, n = 1000000
8) Speedup: 0.84, n = 1000000
9) Speedup: 0.88, n = 1000000
10) Speedup: **1.75**, n = 1000000
11) Speedup: 0.87, n = 1000000
12) Speedup: 1.11, n = 1000000
13) Speedup: 1.03, n = 1000000

Konklusjon på JIT-kompilering



- JIT-kompilering kan skrues av med >java –Xint MittProg ..
  - Brukes bare for debugging
- JIT kompilering kan gi 10 til 30 ganger så rask eksekvering for liten n (en god del mer for stor n)
- Første, andre (og tredje) kjøring er tidsmessig sterkt misvisende
- Vi må:
  - Kjøre programmet i en løkke f.eks 9 (eller 7 eller 11) ganger
  - Legge tidene i hver sin array (sekvensielt og parallell tid)
  - Sortere arrayene
  - Ta ut medianen (element (length-1)/2), som blir vår tidsmåling

```
import java.util.concurrent.*;
import java.util.*;
class Problem2 { int [] fellesData  ; // dette er felles, delte data for alle trådene
      double [] tidene ;
      int ant, svar;
     public static void main(String [] args) {
            ( new Problem()).utfoer(args);
      }
      void utfoer (String [] args) {
               ant = new Integer(args[0]);
               fellesData = new int [ant];
               tidene = new double[9];
               for (int m = 0; m <9; m++) {
                              long tid = System.nanoTime();
                              Thread t = new Thread(new Arbeider());
                               t.start();
                               try{t.join();}catch (Exception e) {return;}
                              tidene[m] = (System.nanoTime() -tid)/1000000.0;
                              System.out.println("Tid for "+m + ", tråd:"+tidene[m]+"ms");
               }
               Arrays.sort(tidene);
               System.out.println("Median med svar:"+svar+", for trådene:"+tidene[(tidene.length-1)/2]+" ms");
      } // end utfoer

      class Arbeider implements Runnable {
         int i,lokalData;    // dette er lokale data for hver tråd
                  public void run() {     int sum =0;
                                          for (int i = 0; i < ant; i++) sum +=fellesData[i];
                                          svar =sum;
                  }
    } // end indre klasse Arbeider
} // end class Problem
```



Hva med operativsystemet:

- Linux og Windows har om lag like rask implementasjon av Java og trådprogrammering,
- Dag Langmyhr testet to helt like maskiner med hhv. Linux og Windows, og resultatene tidsmessig (medianer) var nesten helt like, men
  - Ulike maskiner som Ifis store servere (diamant, safir,..) har en annen Linux og en noe langsommere ytelse for korte, trådbaserte programmer.

Hva med søppeltømming – garbage collection:

- Søppeltømming (=opprydding i lageret og fjerning av objekter vi ikke lenger kan bruke) kan slå til når som helst under kjøring:

```
Kjøring:2, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.35 msek. , nanosek/n:   35.07
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.36

Kjøring:3, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.57 msek. , nanosek/n:   56.87
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    0.66

Kjøring:4, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.43 msek. , nanosek/n:   43.47
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.33

Kjøring:5, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.49 msek. , nanosek/n:   49.20
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.36
```

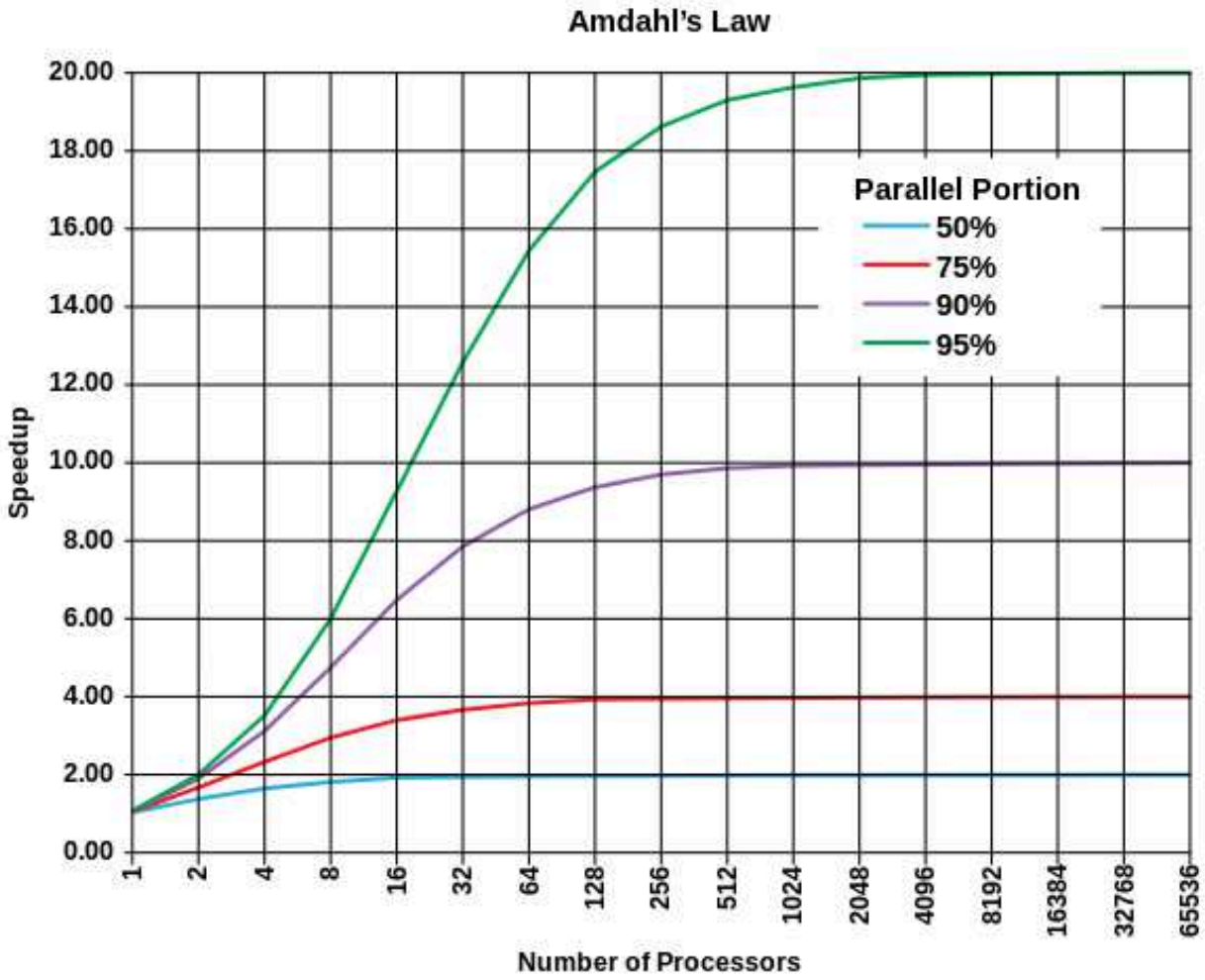
Amdahl lov for parallelle beregninger

- Amdahl lov: Har du **seq** andel sekvensiell kode og da **p** andel parallelliserbar kode i et parallelt program, **seq+p=1**, er den største speedup S du kan få med k kjerner:

$$S = \frac{tid(sekvensiell)}{tid\ (parallell)} = \frac{1}{seq+p/k} = \frac{1}{1-p+p/k}$$

- Når k $\rightarrow \infty$, vil S $\rightarrow \frac{1}{1-p}$.
- Er p=0.9, så er S ≤ 10 uansett hvor mange kjerner du har, og har du 'bare' 50, er S = $\frac{1}{1-0.9+0,9/50}$ = 8,5.
- Amdahls lov er pessimistisk- antar fast størrelse på problemet
- «Hvis du først har brukt 10% av tida på en sekvensiell del, så kan resten av programmet ikke gå fortere enn 0.00 sekunder uansett hvor mange prosessorer du bruker på det. Dvs. at speedup ≤ 10»

Amdahl for ulike verdier av p



Gustafsons lov for parallelle beregninger

- La S være speedup, P antall kjerner og α andel sekvensiell kode, så er:

$$S(P) = P - \alpha\,(P-1)$$

Fordi:

Parallell løsning: $a + b\ (a = sekvensiell\ tid, b = parallel\ tid)$

Sekvensiell løsning : $a + P * b$

Speedup er da:

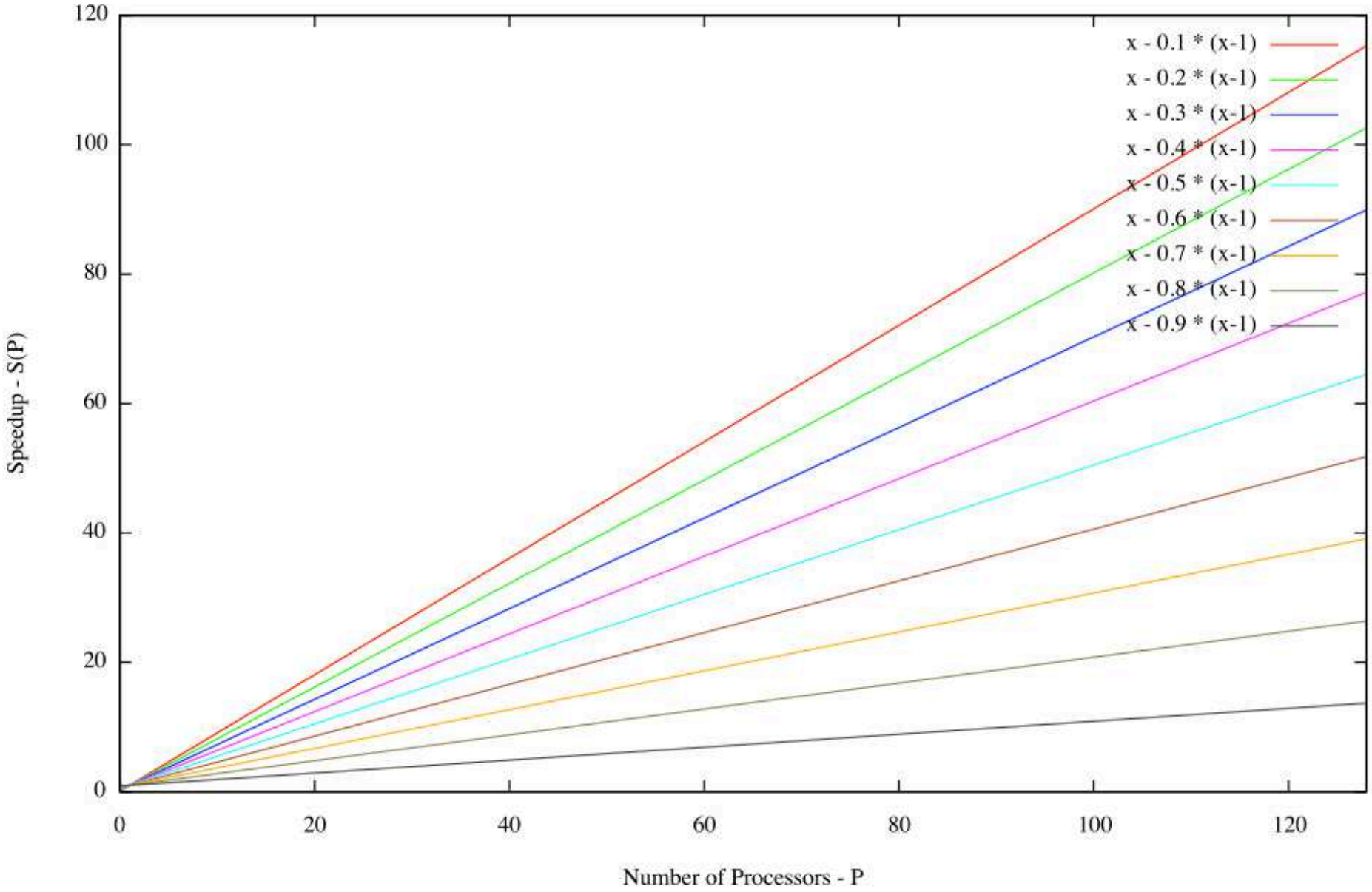
$(a + P * b)/(a + b\ )$, og definerer $\alpha = \frac{a}{a+b}$, så er:

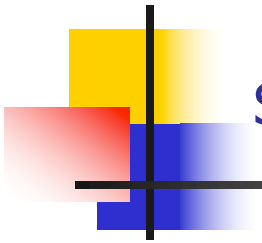
$$S(P) = \alpha + P * (1-\alpha) = P - \alpha(P-1)$$

- Gustavson er mer optimistisk enn Amdahl, gir høvere speedup fordi han antar at med flere maskiner vil vi øke størrelsen på problemet.

- «Hvis du tidligere brukte 1 time på å løse et problem sekvensielt, vil du nå også bruke 1 time på å løse et større, mer nøyaktig problem parallelt da med større speedup– for eksempel i meteorologi»

Gustafson's Law: $S(x) = x - \alpha(x - 1),$



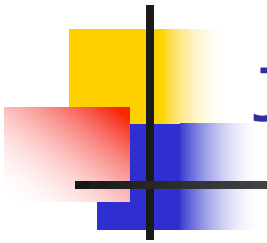


Sammenligning av Amdahl og Gustafson + egne betraktninger

- Amdahl antar at oppgaven er fast av en gitt lengde(n)
- Gustafson antar at du med parallelle maskiner løser større problemer (større n) og da blir den sekvensielle delen mindre.
- Min betraktning:
  1. En algoritme består av noen sekvensielle deler og noen parallelliserbare deler.
  2. Hvis de sekvensielle delene har lavere orden – f.eks O(log n), men de parallelle har en større orden – eks O(n) så vil de parallelle delene bli en stadig større del av kjøretida hvis n øker (Gustafson)
  3. Hvis de parallelle og sekvensielle delene har samme orden, vil et større problem ha samme sekvensielle andel som et mindre problem (Amdahl).
  4. I tillegg kommer alltid et fast overhead på å starte k tråder (1-4 ms.)

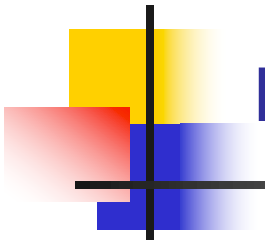
  Algoritmer vi skal jobbe med er mer av type 2 (Gustafson) enn type 3(Amdahl) men vi har alltid overhead, så små problemer løses best sekvensielt.

Konklusjon: For store problemer bør vi ha håp om å skalere nær lineært med antall kjerner hvis ikke vi får kø og forsinkelser når alle kjernene skal lese/skrive i lageret.



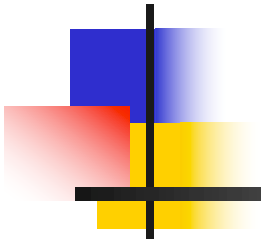
Java Measurement Harness

- Seperate slides jmh.pttx



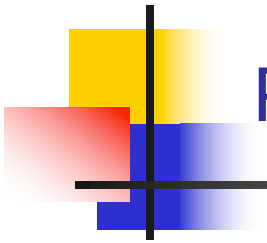
Eval

- I would like to have a quick evaluation ☺



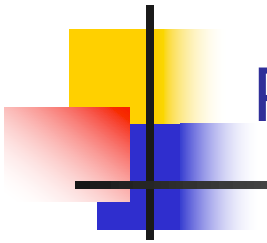
IN3030 L08v24 – About Moore's Law, the speed of light, latency

Eric Jul
Programming Technology Group
Department of Informatics
University of Oslo



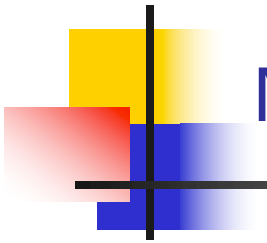
Review F7

I. Prime Numbers, review
II. Oblig Prime Numbers -- Review
III. Tidtagning
    I. JIT compilation
    II. Operativsystem?
    III. Søppel/Garbage Collection
IV. JMH
V. Amdahl
VI. Gustavson



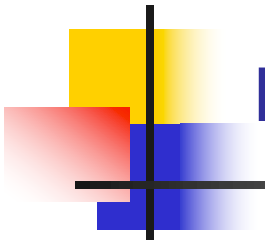
Plan for F8

I. Moore's Law
II. Performance improvements in processor power
III. Speed of light
IV. Why distribution?
v. How to present your timing results
VI. Hva er PRAM modellen - og hvorfor er den ubrukelig for oss



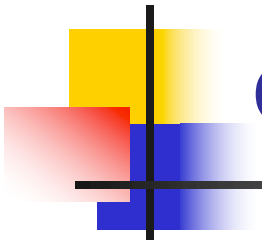
Moore's Law

- Used to be known as: «CPU speeds double every 2 years»



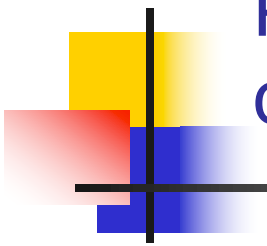
Moore's Law

- Who knows this law?



Original Moore's Law

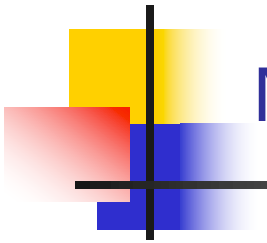
- Transistors per square cm doubles:
- 1965 Article: Every year
- 1975 Article: Every two years



Perspektiv: Utvikling i CPU, minne, nettverk og mere om Moore's law

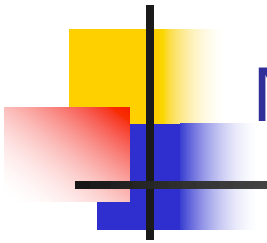
- 1980 CPU: Intel 8080 8-bit processor
  - «int»: one byte
  - «long»: two bytes
  - CPU clock frequency: 1 MHz
  - Memory: 64 kilobytes max, *i.e.*, 16 bit addresses (2 bytes)
  - 128 kB floppy disk
  - Data transmission 1.2 kbit/s == 150 bytes/s

- A look at Moore's law
  - Moore's original prediction 1965
  - Moore's revisted prediction 1975



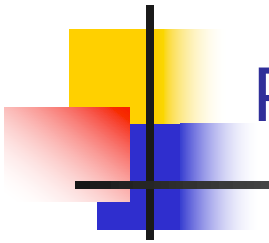
Moore's Law Summary of Effects

- 1958-1975
  - Doubling of transistors every year
  - Derived effect: doubling of CPU speeds
- 1975-2005
  - Doubling of transistors every two years
  - Derived effect: doubling of CPU speeds – screaching halt at about 3 GHz
- 2005-2020
  - Doubling of transistors every two years
  - Derived effect: doubling of number of CORES *or* doubling of the amount of on-chip cache



Newer Machines

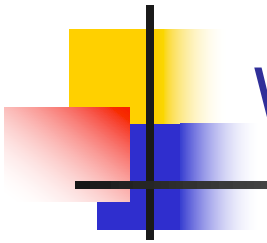
- 2023: My Macbook M2 Pro w M2 procesor
  - 2.4 up-to 3.2 GHz – 19x cores, 12 GPU cores
  - 16 Gbyte Memory
  - 10 Gbit/s network
  - 1 GB Solid State Disk (SSD)
  - Liquied Retina XDR Display 14" 3024x1964 pixels



Performance 1980 vs 2023

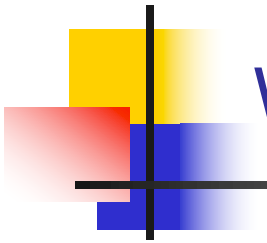
- 2023: Intel 8080 vs M2
  - 1 MHz CPU vs 8 x 3.2 GHz ~ factor 60,000
  - 16 Gbyte Memory vs. 64 kbytes ~ factor 250,000
  - 10 Gbit/s network vs 1 kbit/s ~ factor 10,000,000
  - 128 kB floppy disk vs 1 TB SSD ~ factor 8,000,000
  - 1200 baud serial line vs 10 Gbit/s Ether ~ factor 8,000,000

- ☺



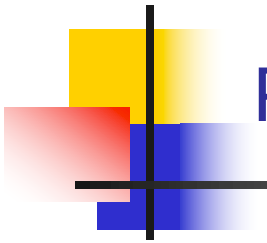
What is ping?

- `Ping`: low-level internet messaging: sends an empty message from one computer to another. The other computer returns it
- sending an empty message from Copenhagen/Oslo to Seattle & back
- In 1988, when the internet was first »opened» in Scandinavia: a ping took a little under 200 ms
- How long does this take 35 years later – in 2023?



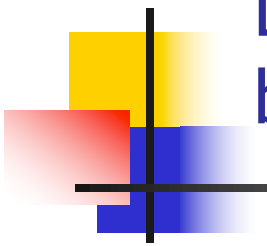
What about ping time?

- Ping: sending an empty message from Oslo to Seattle & back
- How long does this take 1988 vs 2022?



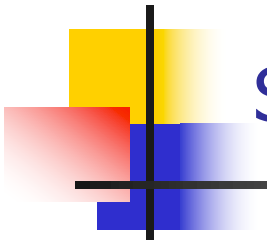
Round-trip ping Time

- 1988: Approximately 180-200 ms
- 2022: How much faster?
  - 1,000,000x faster?
  - 100,000x faster?
  - 10,000x faster?
  - 1,000x faster?
  - 100x faster?
  - 10x faster?
  - Same time?



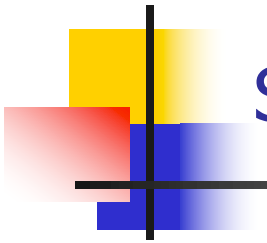
Latency has ***not*** changed: dominated and limited by the *incredibly* slow speed of light!

- How fast is the speed of light approximately?
- Great circle route over and back 16,000 km
- Speed of light in fiber 11/15, copper 3/5
- Great Circle round trip time: minimum of about 100 ms
- So ping time will ***NEVER*** go below 100 ms!



Speed of Light

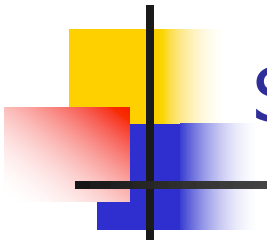
- How fast is the speed of light approximately?
  - 300,000 km/s
- Speed of light
  - in fiber 11/15: about 220,000 km/s
  - in copper 3/5: about 180,000 km/s



Speed of light revisited

- How fast is the speed of light approximately?
  - 300,000 km/s

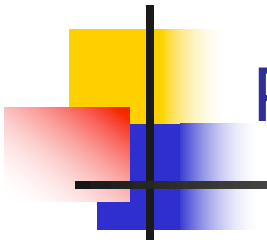
- How far does light travel in 1 ns?
- How far in copper in 1 ns?
- When a 3 GHz core executes one cycle, how far does light travel?



Speed of light: answers to questions

- How fast is the speed of light?
  - About 300,000 km/s
  - 299,792,458 m/s EXACTLY – by definition
- How far does light travel in 1 ns?
  - About 30 cm – call it a *lightfoot*
- How far in copper in 1 ns?
  - About 18 cm
- When a 3 GHz core executes one cycle, how far does light travel in vacuum and in copper?
  - About 10 cm (vacuum) and 6 cm (copper)

PRAM modellen for parallelle beregninger



- PRAM (Parallel Random Access Memory) antar to ting:
  - Du har uendelig mange kjerner til beregningene
  - Enhver aksess i lageret tar like lag tid,
    - ignorerer f.eks fordelen med cache-hukommelsen
- Da blir mange algoritmer lette å beregne og programmere
- Problemet er at begge forutsetningene er helt gale.
- Det PRAM gjør er å telle antall instruksjoner utført

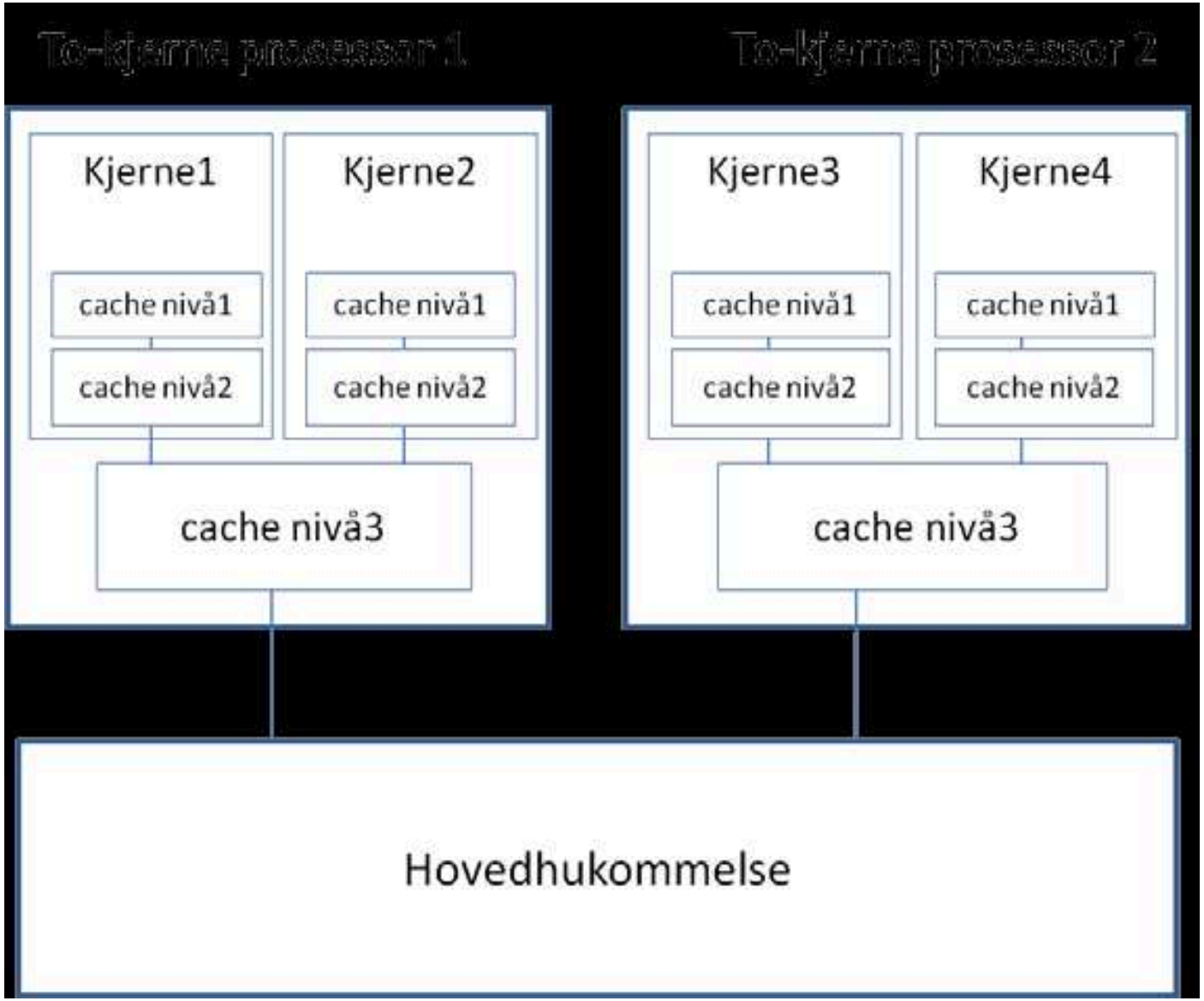
  Det har vi sett er helt feilaktig (Radix og Matrise-mult)
- Svært mange kurs og lærebøker er basert på PRAM

- PRAM vil si at de to sekvensielle algoritmene (med og uten transponering) gikk den utransponerte fortest!
- Dette kurset bruker **ikke** PRAM-modellen!

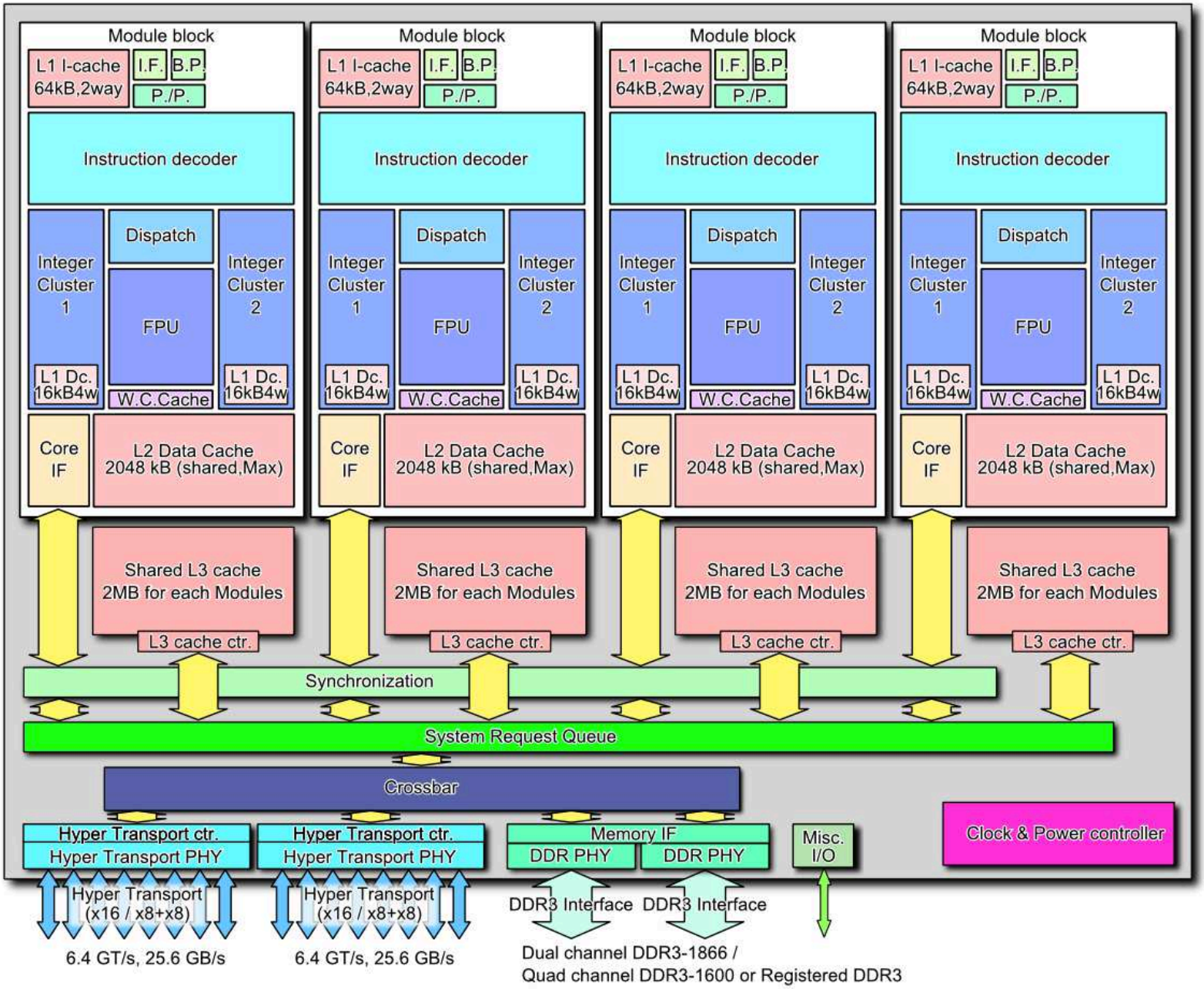
Maskin 1980 (uten cache)

Figur 19.1 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. Intel 8080: 1 MHz CPU

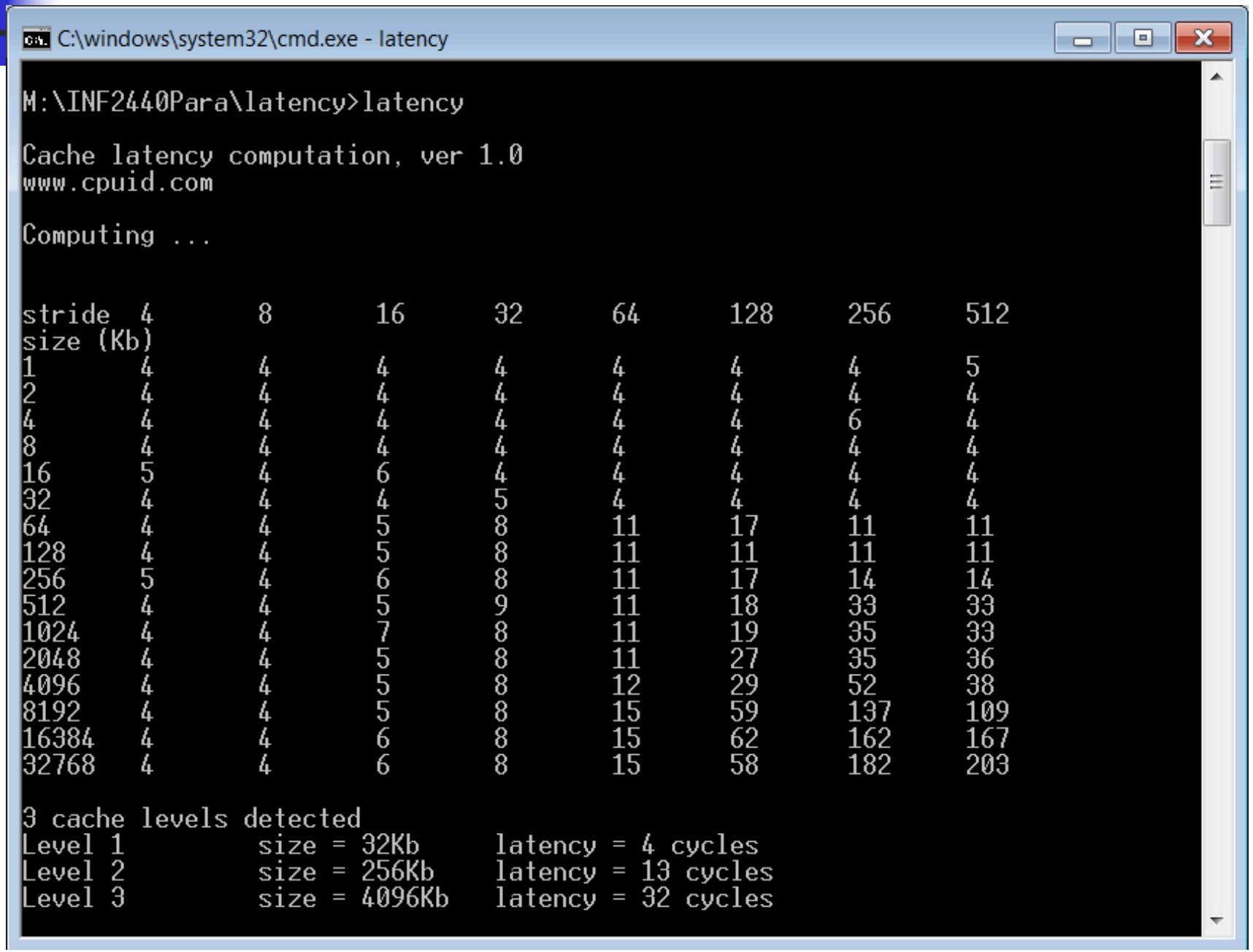
Maskin ca. 2010 med to dobbeltkjerne CPU-er



Hukommelses-systemet i en 4 kjerne CPU – mange lag og flere ulike beregningsmoduler i hver kjerne.:



Test av forsinkelse i data-cachene og hovedhukommelsen - latency.exe (fra CPUZ)

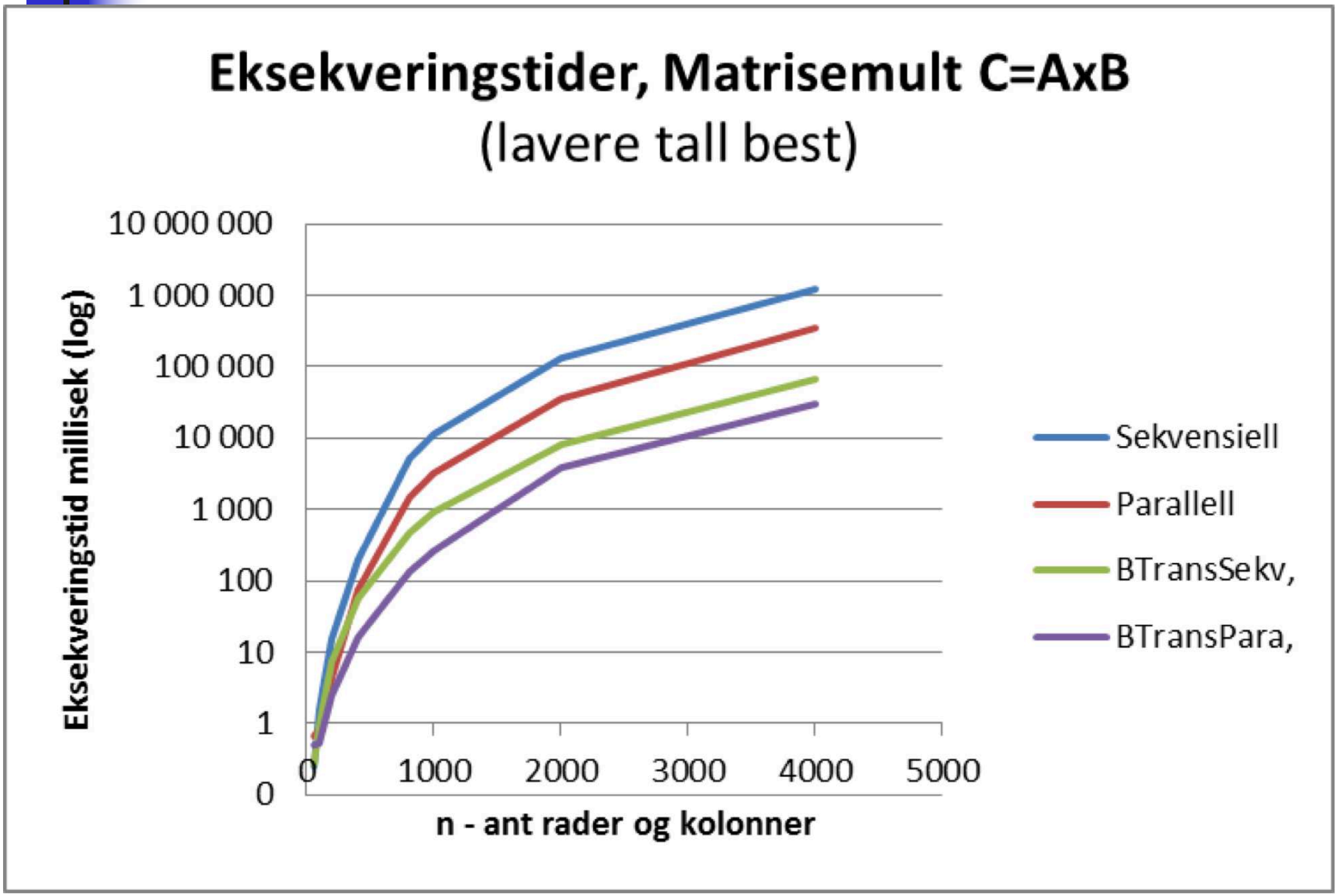


Oppsummering – ideen om at vi har *uniform* aksesstid i hukommelsen er helt galt

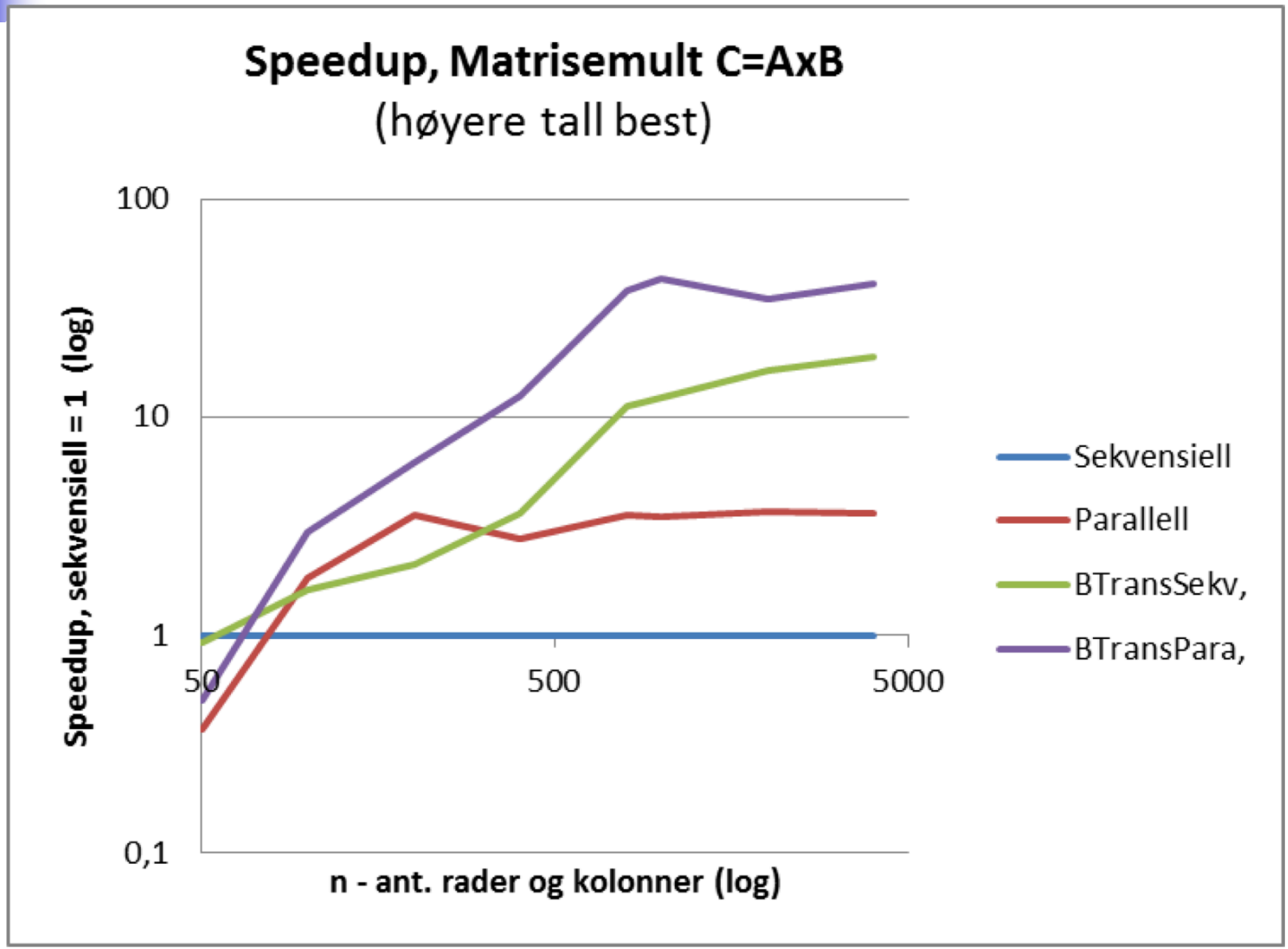
- Hukommelses-systemet i en multicore CPU ,Intel Core i5-459 3.3 GHz, – mange lag (typisk aksesstid i instruksjonssykler):

  1. Registre i kjernen (1) – 8/32 registre
  2. L1 cache (3-4) – 32 Kb
  3. L2 cache (13) – 256 kb
  4. L3 cache (32) – 8Mb
  5. Hovedhukommelsen (virtuell hukommelse) (ca. 200) – 8-64 GB
  6. Disken (15 000 000 roterende) = 5 ms – 1000 GB – 1-5 TB
     FlashDisk (ca 2 000 000 les, ca. 10 000 000 skriv) = ca. 1 ms

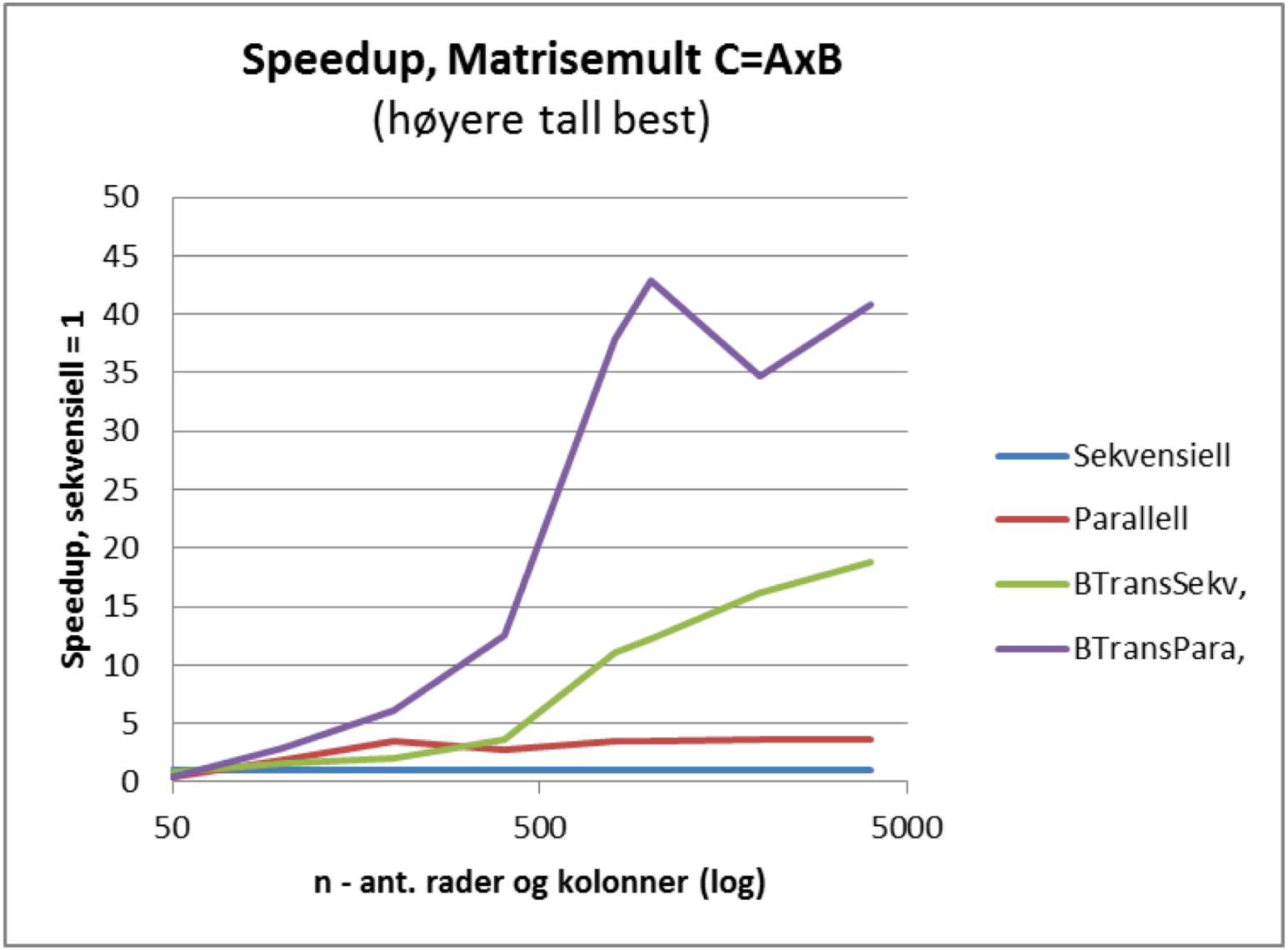
Kjøretider – i millisek. (y-aksen logaritmisk)



Kjøretidsresultater – Speedup , y-aksen logaritmisk

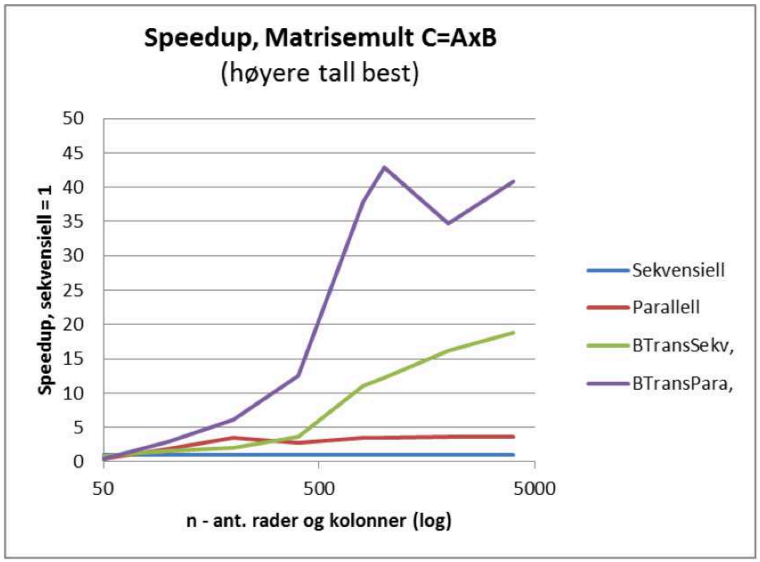


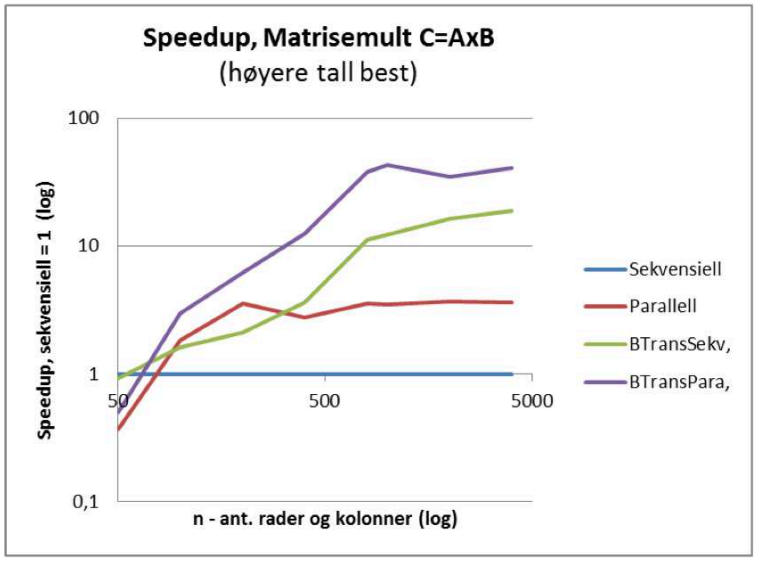
Speedup – med lineær y-akse

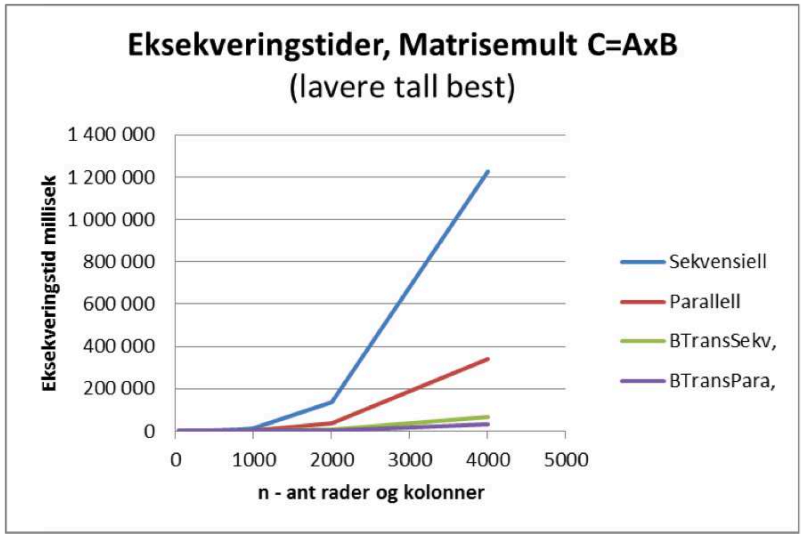


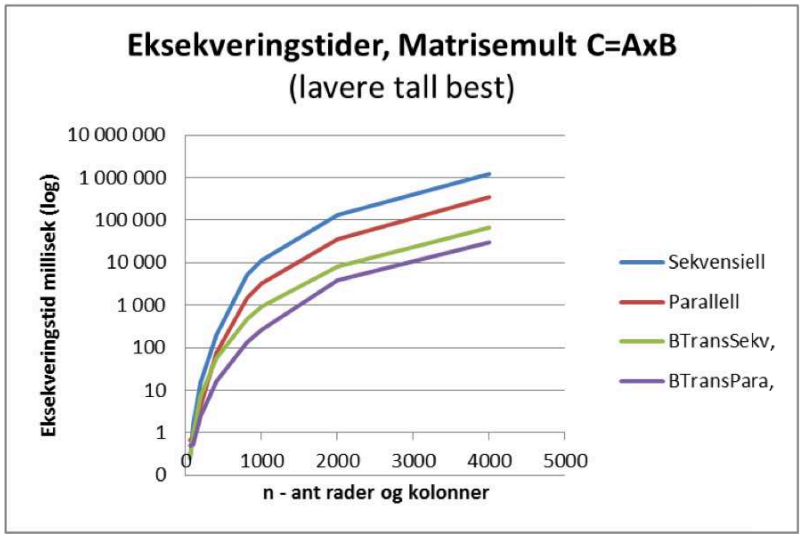
Hvordan framstille ytelse I

- Disse fire kurvene fremviser samme tall! Hvordan ?

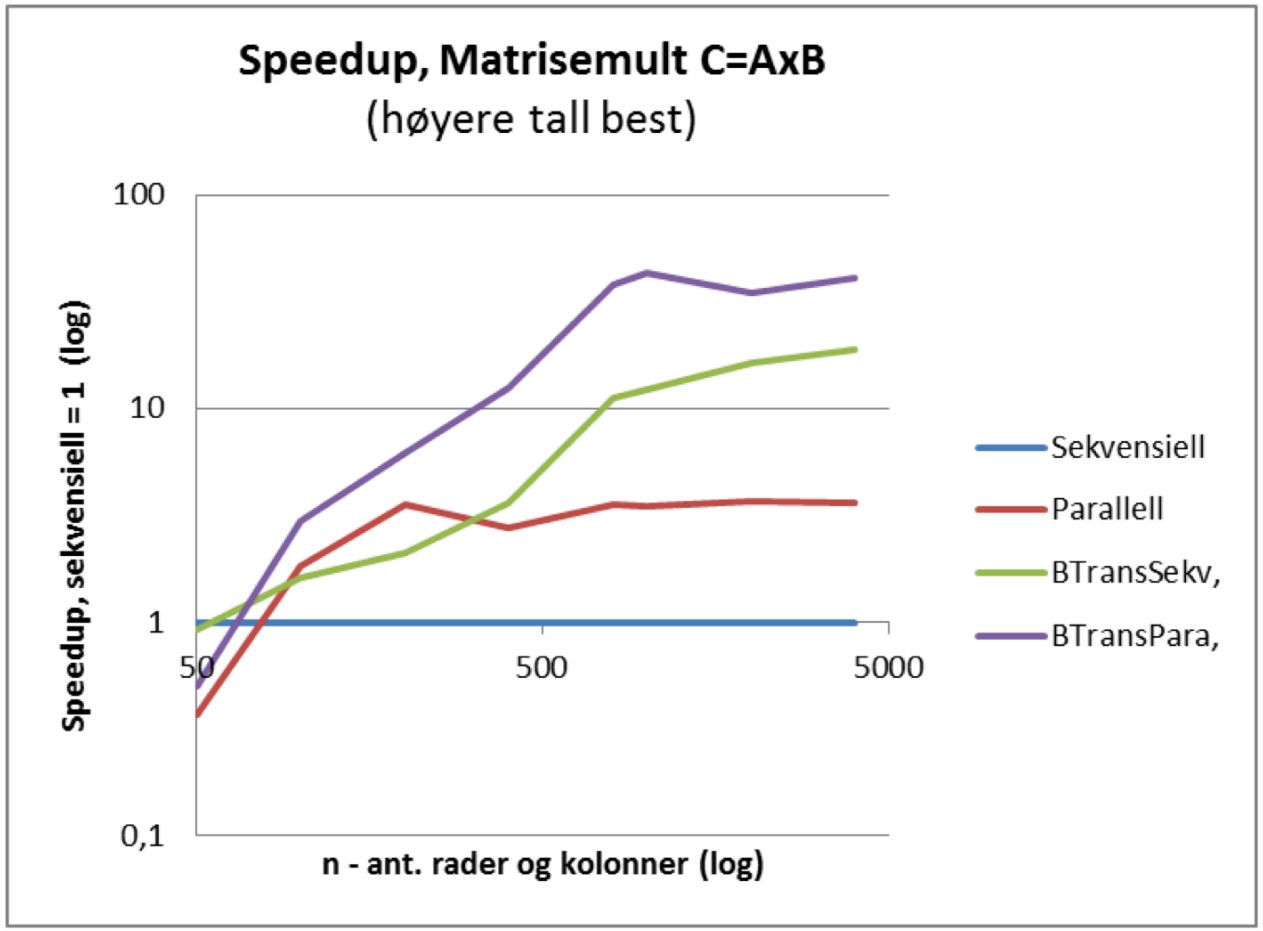








Både logaritmisk x- og y-akse



Fordel med log-akse er at den viser fram nøyaktigere små verdier, men vanskelig å lese nøyaktig mellom to merker på aksene.

IN3030 F09, våren 2024

Eric Jul
PT
Inst. for informatikk
UiO

Review F08v24

I. Moore's Law
II. Performance improvements in processor power
III. Speed of light
IV. Why distribution?
v. How to present your timing results
VI. Hva er PRAM modellen - og hvorfor er den ubrukelig for oss

Plan for F09

1. Synchronization revisited
2. Cooks and waiters
3. 3 solutions
4. Hoare Monitors

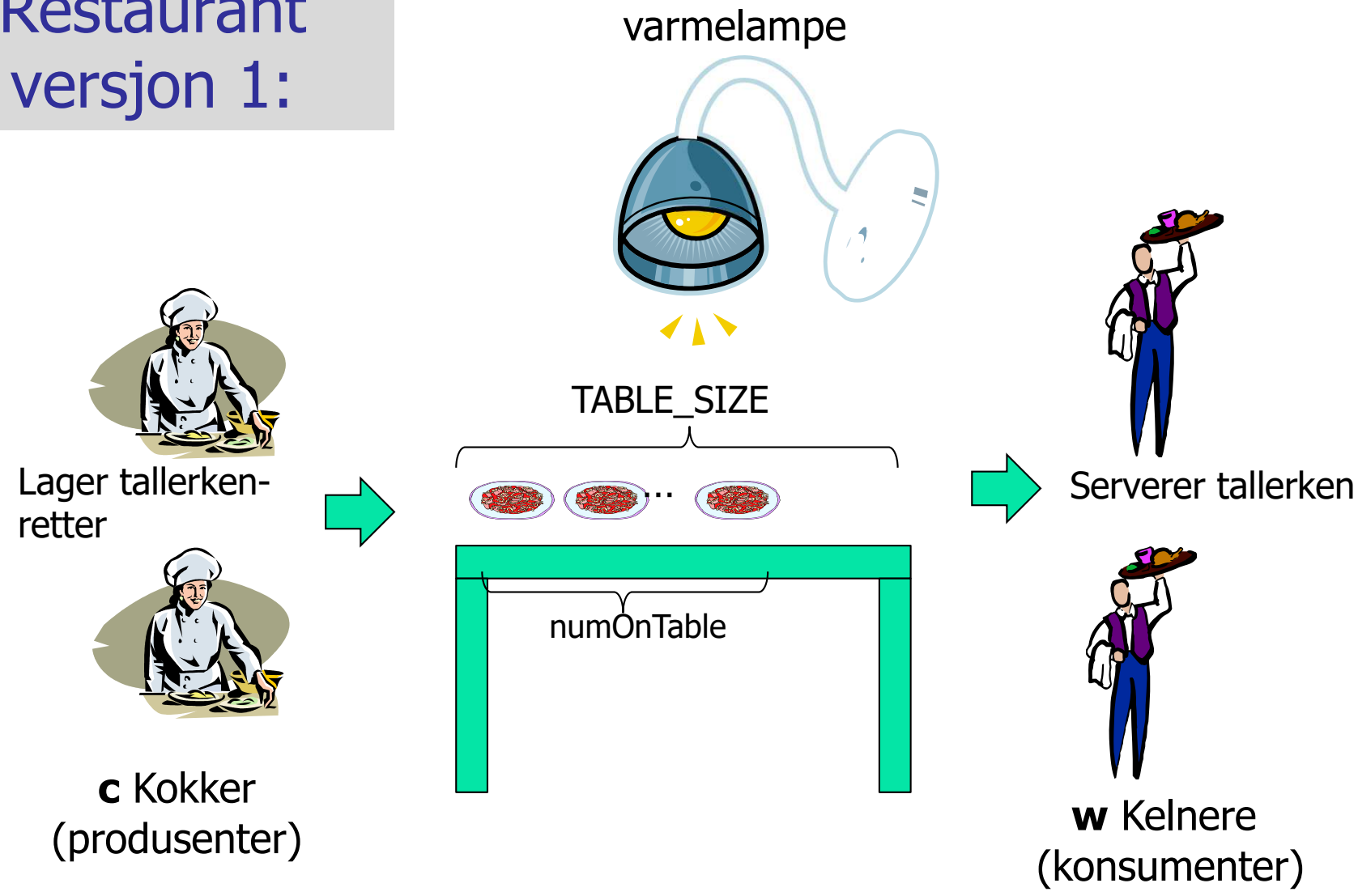
Oblig 3 Prime Numbers

Questions?

Problemet vi nå skal løse: En restaurant med kokker og kelnere og med et varmebord hvor maten står

- Vi har **c** Kokker som lager mat og
  w Kelnere som server maten (tallerkenretter)
- Mat som kokkene lager blir satt fra seg på et **varmebord** (med TABLE_SIZE antall plasser til tallerkener)
- Kokkene kan ikke lage flere tallerkener hvis varmebordet er fullt
- Kelnerne kan ikke servere flere tallerkener hvis varmebordet er tomt
- Det skal lages og serveres NUM_TO_BE_MADE antall tallerkener

Restaurant versjon 1:



3) Om monitorer og køer (tre eksempler på concurrent programming). Vi løser synkronisering mellom to ulike klasser.

- **Først** en aktivt pollende (masende) løsning med synkroniserte metoder (Restaurant1.java).
  - Aktiv venting i en løkke i hver Kokk og Kelner
    + at de er i køen på å slippe inn i en synkronisert metode
- **Så** en løsning med bruk av monitor slik det var tenkt fra starten av i Java (Restaurant2.java).
  - Kokker og Kelnere venter i samme wait()-køen
    + i køen på å slippe inn i en synkronisert metode.
- **Til siste** en løsning med monitor med Lock og Condition-køer (flere køer – en per ventetilstand (Restaurant9.java)
  - Kelnere og Kokker venter i hver sin kø
    + i en køen på å slippe inn i de to metoder beskyttet av en Lock

.

Felles for de tre løsningene

```
import java.util.concurrent.locks.*;
class Restaurant {

  Restaurant(String[] args) {
     <Leser inn antall Kokker, Kelnere og  antall retter>
      <Oppretter Kokkene og Kelnerne og starter dem>
   }

    public static void main(String[] args) {
          new Restaurant(args);
       }
    } // end main
} // end class Restaurant

class HeatingTable{  // MONITOR
     int numOnTable  = 0,
         numProduced = 0,
         numServed=0;
     final int MAX_ON_TABLE  =3;
     final int NUM_TO_BE_MADE;
     // Invarianter:
     //   0 <= numOnTable  <=  MAX_ON_TABLE
     //          numProduced  <= NUM_TO_BE_MADE
     //          numServed      <= NUM_TO_BE_MADE

    < + ulike data i de tre eksemplene>
```

```
public xxx boolean putPlate(Kokk c)
   <Leggen tallerken til på bordet
   (true) ellers (false) Kokk må vente)
} // end put

public xxx boolean getPlate(Kelner w)  {
   <Hvis bordet tomt (false) Kelner venter
      ellers (true) - Kelner tar da en
      tallerken og serverer den>
  } // end get
} // end class HeatingTable

class Kokk extends Thread {
    HeatingTable tab;
    int ind;
    public void run() {
         while/if (tab.putPlate(..))
        < Ulik logikk i eksemplene>
    }
} // end class Kokk

class Kelner extends Thread {
    HeatingTable tab;
    int ind;
    public void run() {
          while/if (tab.getPlate()){
           <ulik logikk i eksemplene>
       }
} // end class Kelner
```

Invariantene på felles variable

- Invariantene må **alltid holde** (være sanne) utenfor monitor-metodene.
- Hva er de felles variable her:
  - MAX_ON_THE_TABLE
  - NUM_TO_BE_MADE
  - numOnTable
  - numProduced
  - numServed = numProduced – numOnTable
- Invarianter:
  1. **0 ≤ numOnTable ≤ TABLE_SIZE**
  2. **numProduced ≤ NUM_TO_BE_MADE**
  3. **numServed ≤ numProduced**

Invariantene viser 4 tilstander vi må ta skrive kode for

Invarianter:

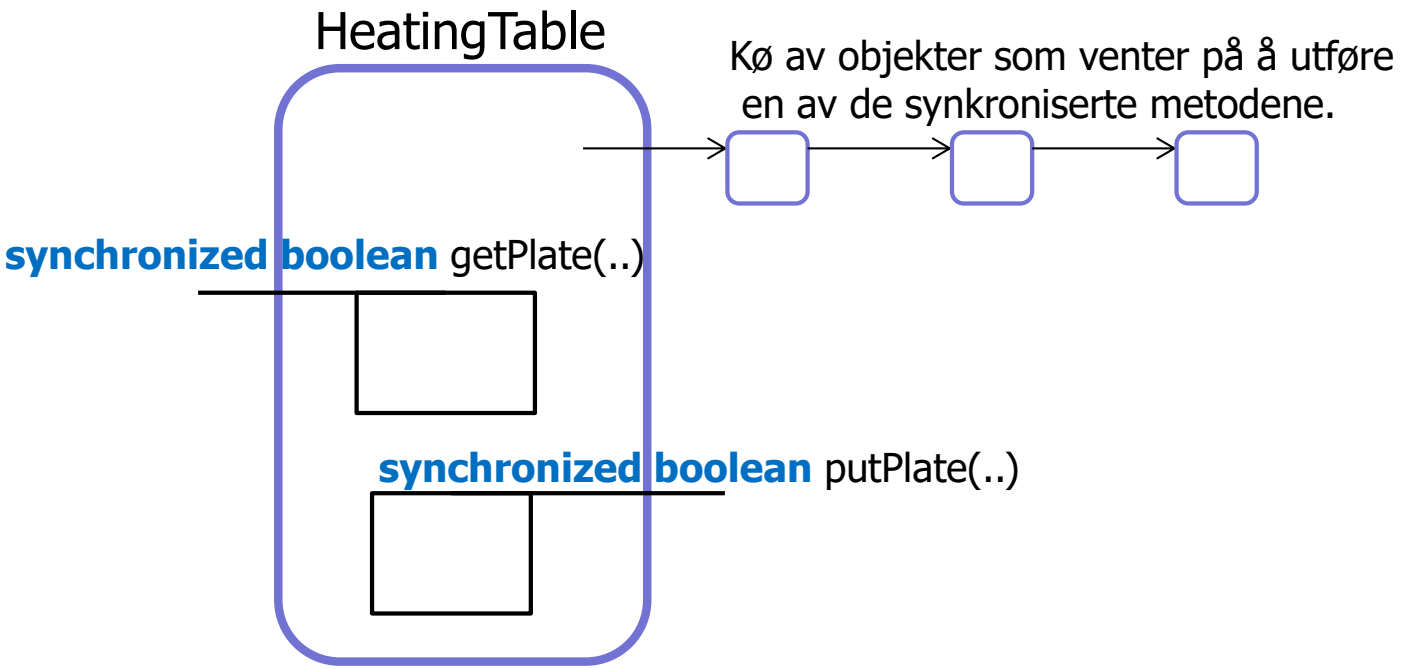
0 <= numOnTable <= MAX_ON_TABLE

numServed <= numProduced <= NUM_TO_BE_MADE

1. **numOnTable == MAX_ON_TABLE**
   → **Kokker venter**
2. **0 == numOnTable**
   → **Kelnere venter**
3. **numProduced == NUM_TO_BE_MADE**
   → **Kokkene ferdige**
4. **numServed == NUM_TO_BE_MADE**
   → **Kelnerene ferdige**

Først en aktivt pollende (masende) løsning med synkroniserte metoder (Restaurant1.java).

- Dette er en løsning med **en kø**, den som alle tråder kommer i hvis en annen tråd er inne i en synkronisert metode i samme objekt.
- Terminering ordnes i hver kokk og kelner (i deres run-metode)
- Den køen som nyttes er en felles kø av alle aktive objekter som evt. samtidig prøver å kalle en av de to synkroniserte metodene **get** og **put.** Alle objekter har en slik kø.



Restaurant løsning 1

```
class Kokk extends Thread {
....
    public void run() {
       try {
           while (tab.numProduced < tab.NUM_TO_BE_MADE) {
             if (tab.putPlate(this) ) {
                // lag neste tallerken
             }
             sleep((long) (1000 * Math.random()));
           }
        } catch (InterruptedException e) {}
        System.out.println("Kokk "+ind+" ferdig: " );
   }
} // end Kokk

class Kelner extends Thread {
.............
     public void run() {
           try {
             while ( tab.numServed< tab.NUM_TO_BE_MADE) {
                  if ( tab.getPlate(this)) {
                     // server tallerken
                  }
                  sleep((long) (1000 * Math.random()));
               }
           } catch (InterruptedException e) {}
             System.out.println("Kelner " + ind+" ferdig");
     }
} // end Kelner
```

```
synchronized boolean  putPlate(Kokk c) {
      if (numOnTable == TABLE_SIZE) {
                return false;
       }
       numProduced++;
      // 0 <= numOnTable  <  TABLE_SIZE
      numOnTable++;
      // 0 < numOnTable  <=  TABLE_SIZE
      System.out.println("Kokk no:"+c.ind+",
        laget  tallerken no:"+numProduced);
     return true;
  } // end putPlate

synchronized boolean  getPlate(Kelner w) {
     if  (numOnTable == 0)     return false;
     // 0 < numOnTable  <=  TABLE_SIZE
     numServed++;
     numOnTable--;
     // 0 <= numOnTable <  TABLE_SIZE
     System.out.println("Kelner no:"+w.ind+
     ",  serverte  tallerken no:"+numServed);
    return true;
  }
```

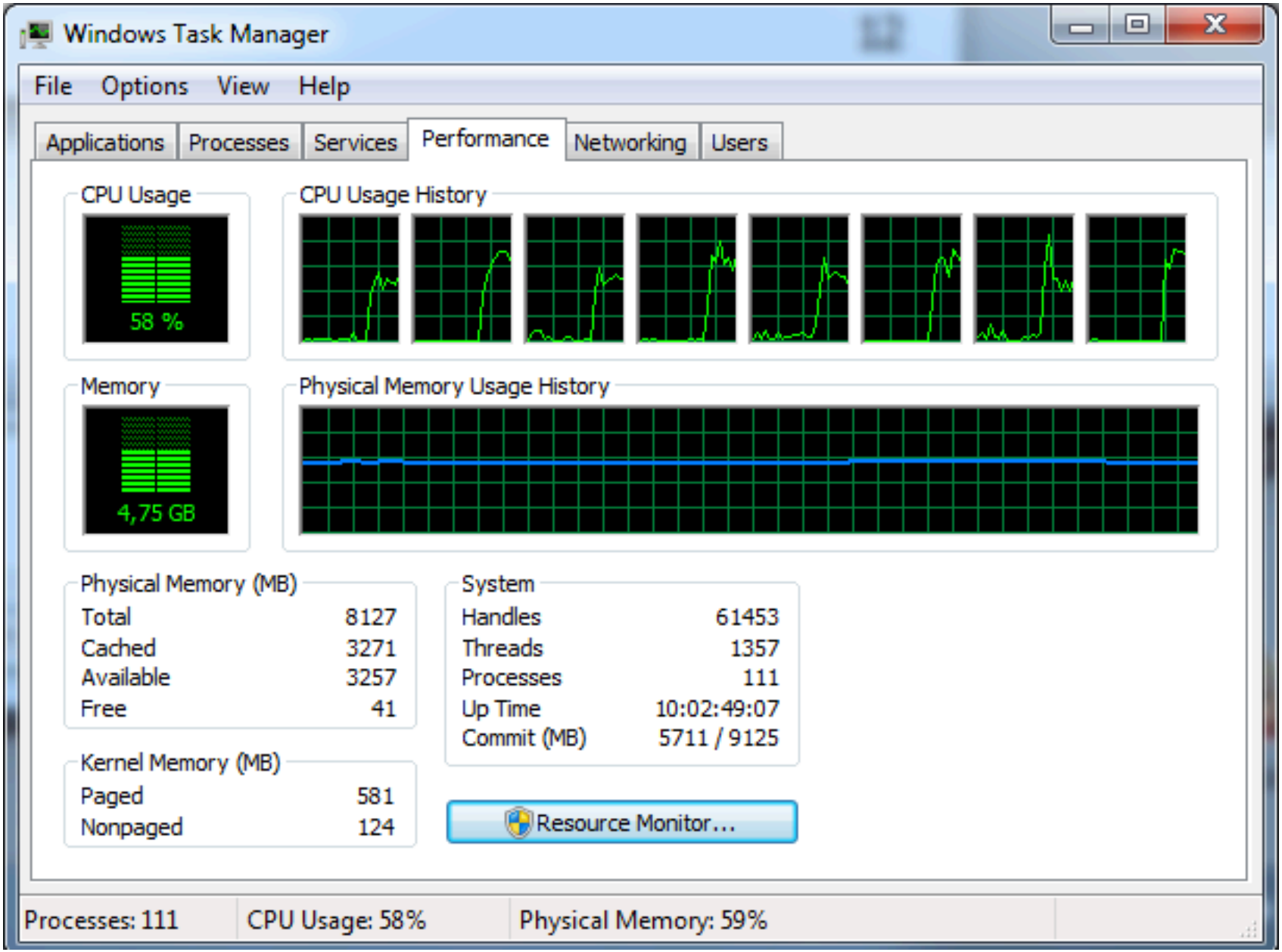
```
M:Restaurant1>java Restaurant1 11 8 8
Kokk no:8, laget  tallerken no:1
Kokk no:4, laget  tallerken no:2
Kokk no:6, laget  tallerken no:3
Kelner no:5, serverte  tallerken no:1
Kokk no:3, laget  tallerken no:4
Kelner no:2, serverte  tallerken no:2
Kokk no:1, laget  tallerken no:5
Kelner no:5, serverte  tallerken no:3
Kelner no:4, serverte  tallerken no:4
Kokk no:7, laget  tallerken no:6
Kokk no:2, laget  tallerken no:7
Kelner no:7, serverte  tallerken no:5
Kokk no:4, laget  tallerken no:8
Kelner no:3, serverte  tallerken no:6
Kelner no:3, serverte  tallerken no:7
Kokk no:1, laget  tallerken no:9
Kelner no:2, serverte  tallerken no:8
Kokk no:6, laget  tallerken no:10
Kokk no:3, laget  tallerken no:11
Kokk 8 ferdig:
```

```
Kelner no:8, serverte  tallerken no:9
Kelner no:7, serverte  tallerken no:10
Kelner no:6, serverte  tallerken no:11
Kokk 3 ferdig:
Kokk 5 ferdig:
Kelner 1 ferdig
Kokk 1 ferdig:
Kokk 4 ferdig:
Kelner 5 ferdig
Kokk 7 ferdig:
Kelner 2 ferdig
Kokk 2 ferdig:
Kelner 4 ferdig
Kelner 3 ferdig
Kelner 7 ferdig
Kelner 6 ferdig
Kokk 6 ferdig:
Kelner 8 ferdig
```

Problemer med denne løsningen er aktiv polling

- Alle Kokke- og Kelner-trådene går aktivt rundt å spør:
  - Er der mer arbeid til meg? Hviler litt, ca.1 sec. og spør igjen.
  - Kaster bort mye tid/maskininstruksjoner.
- Spesielt belastende hvis en av trådtypene (Produsent eller Konsument) er klart raskere enn den andre,
  - Eks . setter opp 18 raske Kokker som sover bare 1 millisek mot 2 langsomme Kelnere som sover 1000 ms.
  - I det tilfellet tok denne aktive ventingen/masingen 58% av CPU-kapasiteten til 8 kjerner
- Selv etter at vi har testet i run-metoden at vi kan greie en tallerken til, må vi likevel teste på om det går OK
  - En annen tråd kan ha vært inne og endret variable
- Utskriften må være i get- og put-metodene. Hvorfor?

Løsning1 med 18 raske Kokker (venter 1 ms) og 2 langsomme Kelnere (venter 1000 ms). Kokkene stresser maskinen med stadige mislykte spørsmål hvert ms. om det nå er plass til en tallerken på varmebordet . CPU-bruk = 58%.



Løsning 2: Javas originale opplegg med monitorer og **to køer**.

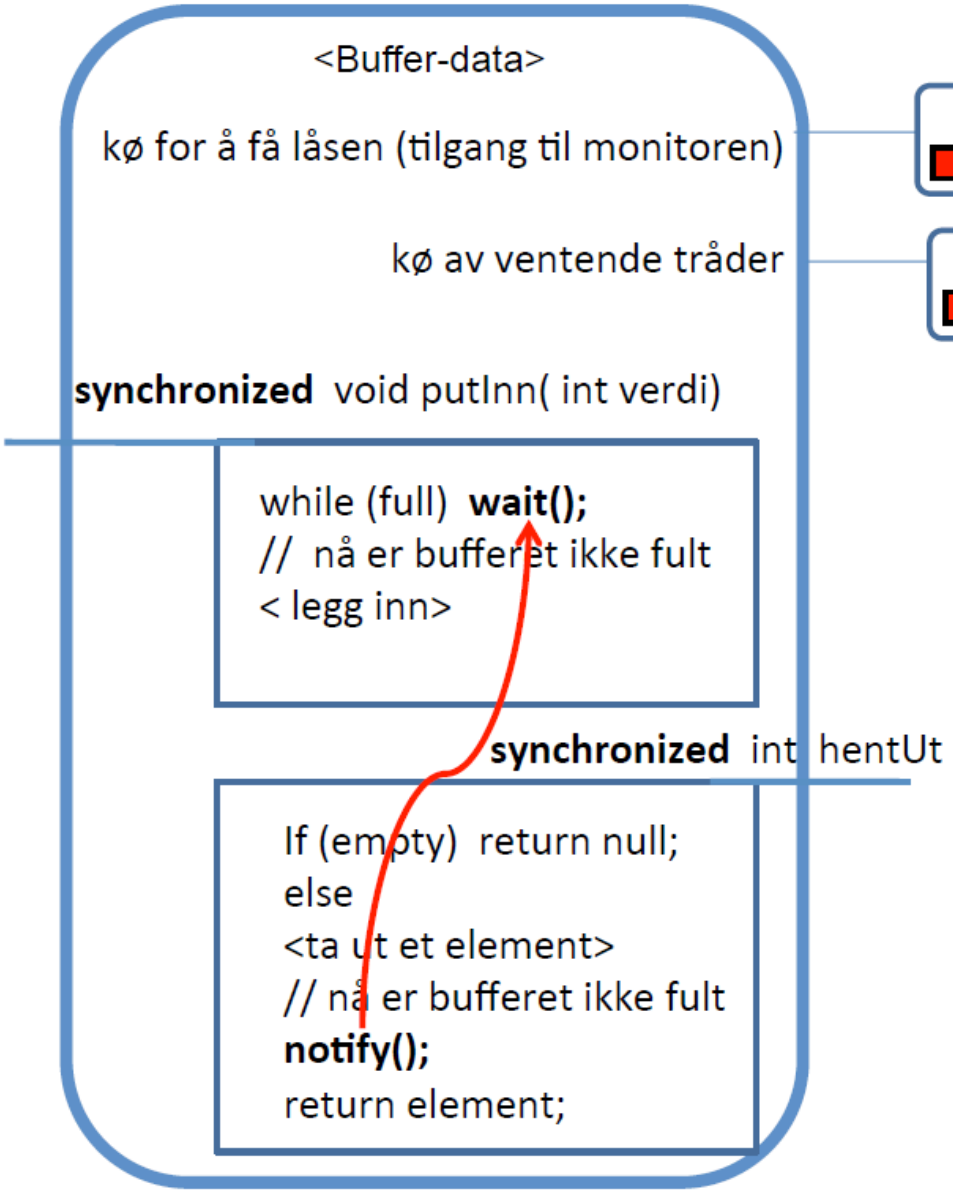
- Den originale Java løsningen med synkroniserte metoder og en rekke andre metoder og følgende innebygde metoder:
- **sleep(t):** Den nå kjørende tråden sover i 't' millisek.
- **notify():** (arvet fra klassen Object som alle er subklasse av). Den vekker opp **en** av trådene som venter på låsen i inneværende objekt. Denne prøver da en gang til å få det den ventet på.
- **notifyAll():** (arvet fra klassen Object). Den vekker opp **alle de** trådene som venter på låsen i inneværende objekt. De prøver da alle en gang til å få det de ventet på.
- **wait():** (arvet fra klassen Object). Får nåværende tråd til å vente til den enten blir vekket med notify() eller notifyAll() for dette objektet.

Å lage parallelle løsninger med en Java 'monitor'

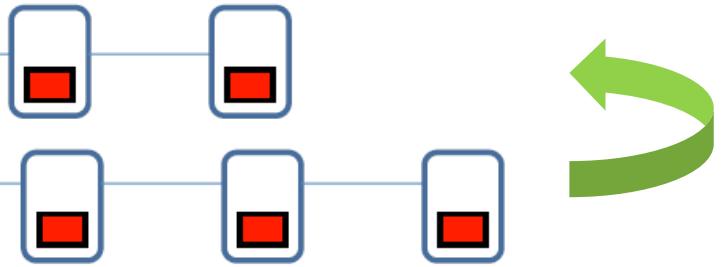
- En Java-monitor er et objekt av en vilkårlig klasse med synchronized metoder
- Det er to køer på et slikt objekt:
  - En kø for de som venter på å komme inn/fortsette i en synkronisert metode
  - En kø for de som her sagt **wait()** (og som venter på at noen annen tråd vekker dem opp med å si notify() eller notifyAll() på dem)
    - wait() sier en tråd inne i en synchronized metode.
    - notify() eller notifyAll() sies også inne i en synchronized metode.

Monitor-ideen er sterkt inspirert av Tony Hoare (mannen bak Quicksort)

To køer i en basal Java monitor:

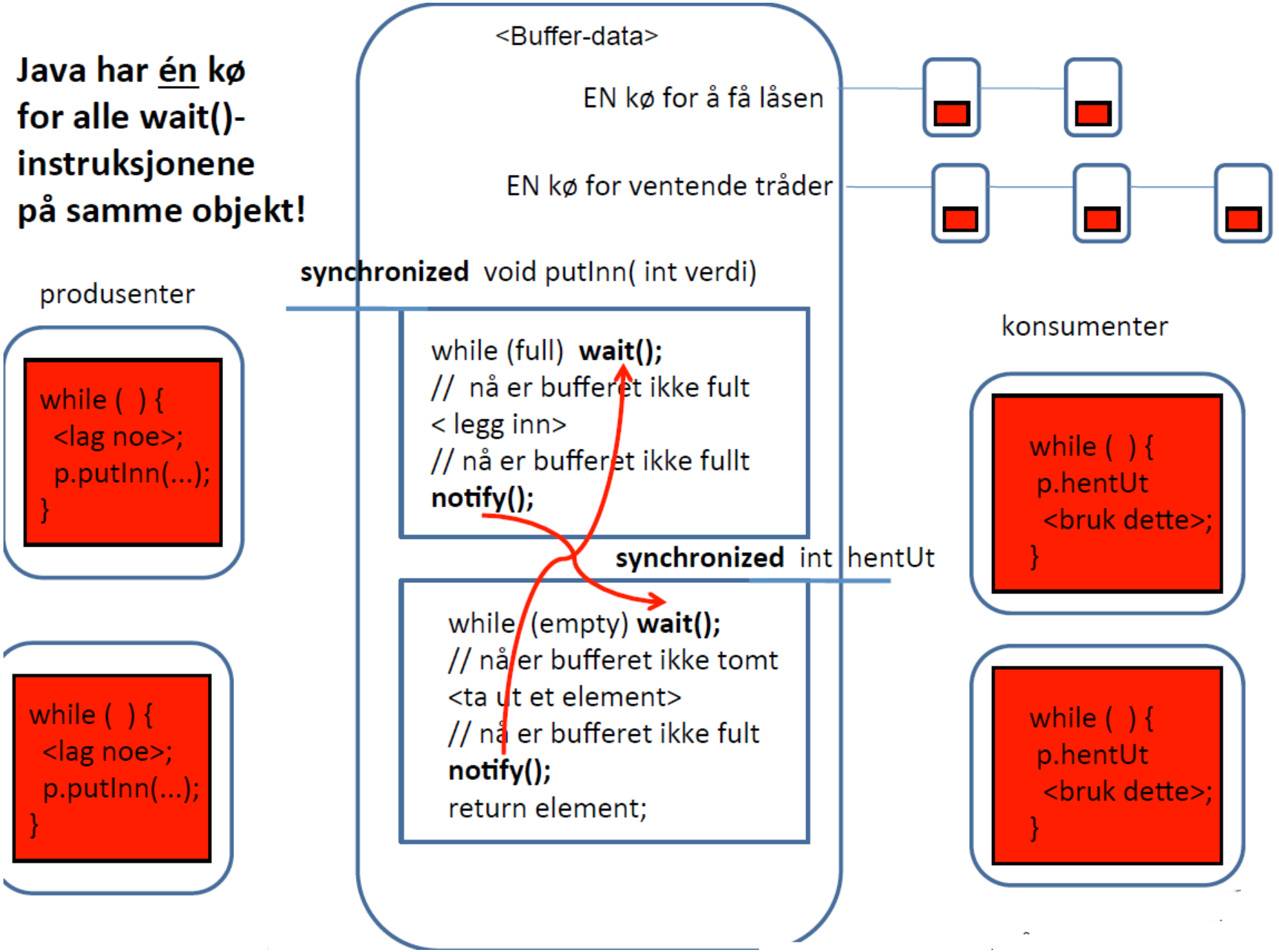


En kø av ventende tråder
på hele monitoren

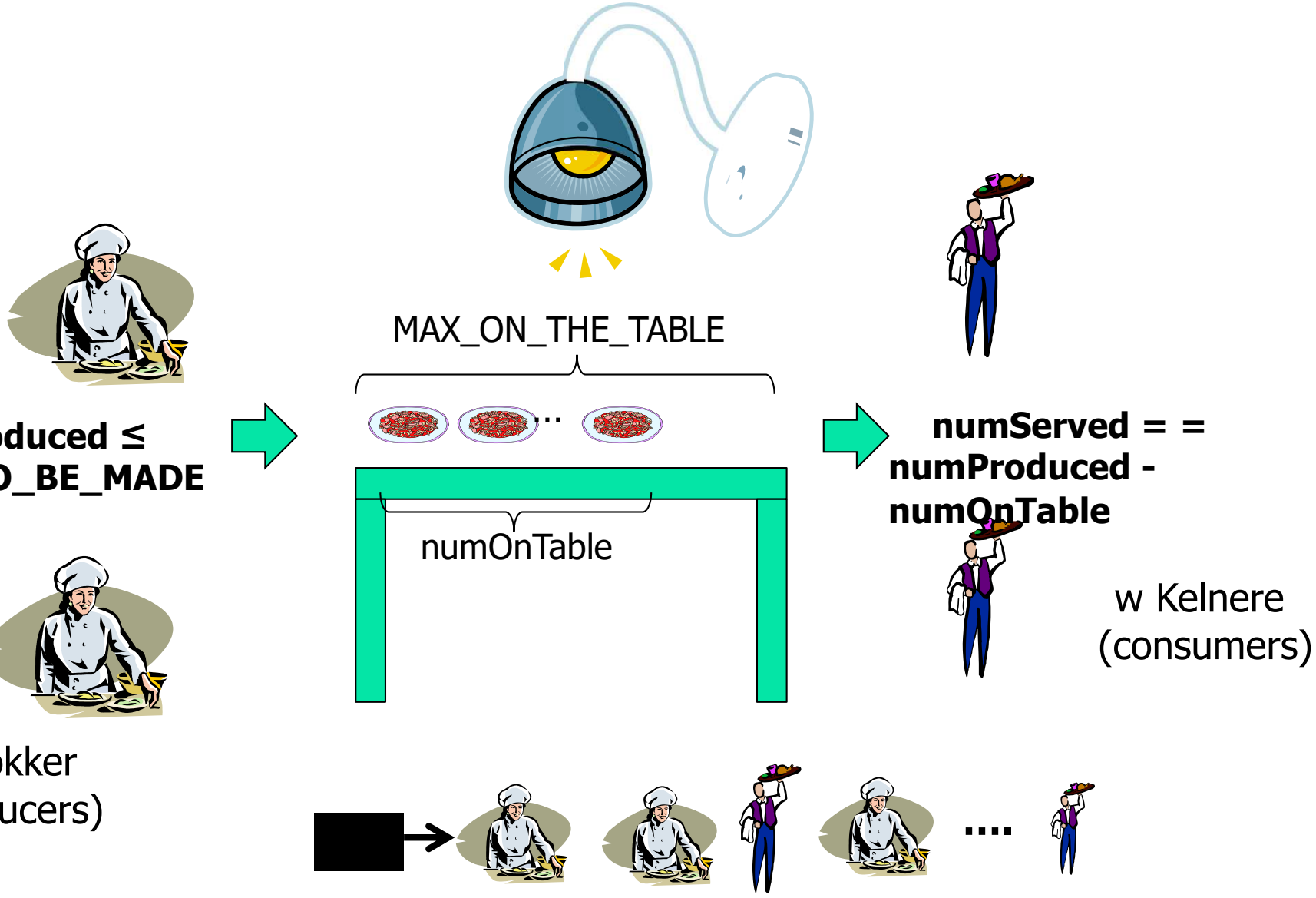


En kø av ventende tråder på
"wait"-instruksjoner
(wait-set).
Startes av notify () og/eller
notifyAll()
Legges da i den andre køen
(først ? (Nei, ingen garanti))
Derfor er det nødvendig med
"while ..."

Java har <u>én</u> kø for alle wait()-instruksjonene på samme objekt!



Restauranten (2):



En kø: Kokker og Kelnere venter samme wait-køen

Løsning 2, All venting er inne i synkroniserte metoder i en to køer.

- All venting inne i to synkroniserte metoder
- Kokker and Kelnere venter på neste tallerken i wait-køen
- Vi må vekke opp alle i wait-køen for å sikre oss at vi finner en av den typen vi trenger (Kokk eller Kelner) som kan drive programmet videre
- Ingen testing på invariantene i run-metodene

Begge løsninger 2) og 3):
run-metodene prøver en gang til hvis siste operasjon lykkes:

Kokker:

```
public void run() {
        try {
            while (tab.putPlate(this)) {
                sleep((long) (1000 * Math.random()));
            }
        } catch (InterruptedException e) {}
        // Denne Kokken er ferdig
    }
```

Kelnere:

```
public void run() {
        try {
            while (tab.getPlate(this) ){
                sleep((long) (1000 * Math.random()));
            }
        } catch (InterruptedException e) {}
          // Denne Kelneren er ferdig
    }
```

Løsning2

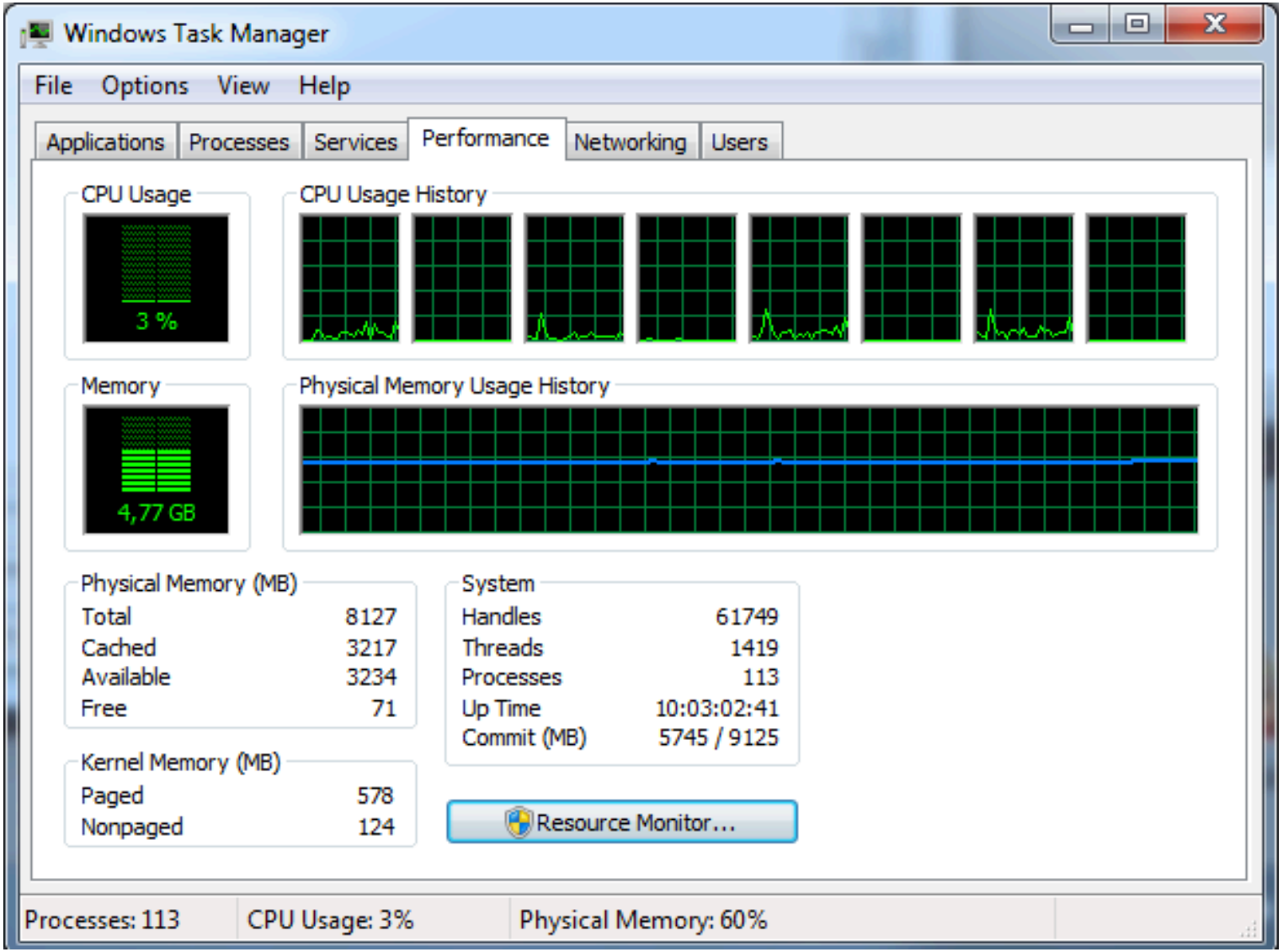
```
public synchronized boolean putPlate (Kokk c) {
    while (numOnTable == TABLE_SIZE &&
            numProduced <   NUM_TO_BE_MADE) {
        try { // The while test holds here meaning  that a Kokk should
            // but can not make a dish, because the table is full
            wait();
        } catch (InterruptedException e) { // Insert code to handle interrupt }
    }
    // one or both of the loop conditions are now false
    if (numProduced < NUM_TO_BE_MADE) {
        // numOnTable < TABLE_SIZE
        // Hence OK to increase numOnTable
        numOnTable++;
        // numProduced < NUM_TO_BE MADE
        // Hence OK to increase numProduced:
        numProduced++;
        // numOnTable > 0 , Wake up a waiting
        // waiter, or all if
        //numProduced ==NUM_TO_BE_MADE
        notifyAll();  // Wake up all waiting
```

```
        if (numProduced ==
            NUM_TO_BE_MADE) {
                return false;
        } else{ return true; }
    } else {
        // numProduced ==
        //NUM_TO_BE_MADE
        return false;}
} // end putPlate
```

Løsning2

```
public synchronized boolean getPlate (Kelner w) {
      while (numOnTable == 0 && numProduced < NUM_TO_BE_MADE ) {
            try { // The while test holds here the meaning that the table
                  // is empty and there is more to serve
                  wait();
            } catch (InterruptedException e) { // Insert code to handle interrupt }
       }
      //one or both of the loop conditions are now false
      if (numOnTable > 0) {
            // 0 < numOnTable <= TABLE_SIZE
            // Hence OK to decrease numOnTable:
            numOnTable--;
            // numOnTable < TABLE_SIZE
            // Must wake up a sleeping Kokker:
            notifyAll();   // wake up all queued Kelnere and Kokker
            if (numProduced == NUM_TO_BE_MADE && numOnTable == 0) {
                  return false;
            }else{ return true;}
      } else { // numOnTable == 0 && numProduced == NUM_TO_BE_MADE
            return false;}
} // end getPlate
```

Løsning2 med 18 raske Kokker (venter 1 ms) og 2 langsomme Kelnere (venter 1000 ms). Kokkene stresser ikke maskinen med stadige mislykte spørsmål , men venter i kø til det er plass til en tallerken til på varmebordet . CPU-bruk = 3%.

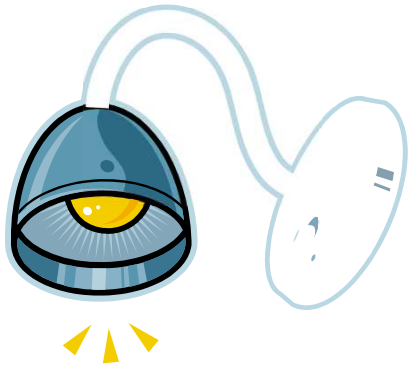


3) En parallell løsning med Conditions.

- Bruker to køer:
  - En for Kokker som venter på en tallerkenplass på bordet
  - En for Kelnere som venter på en tallerken
- Da trenger vi ikke vekke opp alle trådene, **Bare en** i den riktige køen.
  - Kanskje mer effektivt
  - Klart lettere å programmere

Restauranten (3):

numProduced ≤ NUM_TO_BE_MADE



MAX_ON_THE_TABLE

numOnTable

numServed = = numProduced – numOnTable

w Kelnere (konsumenter)

c Kokker (produsenter)

To køer:
a) Kokker venter på en plass på bordet

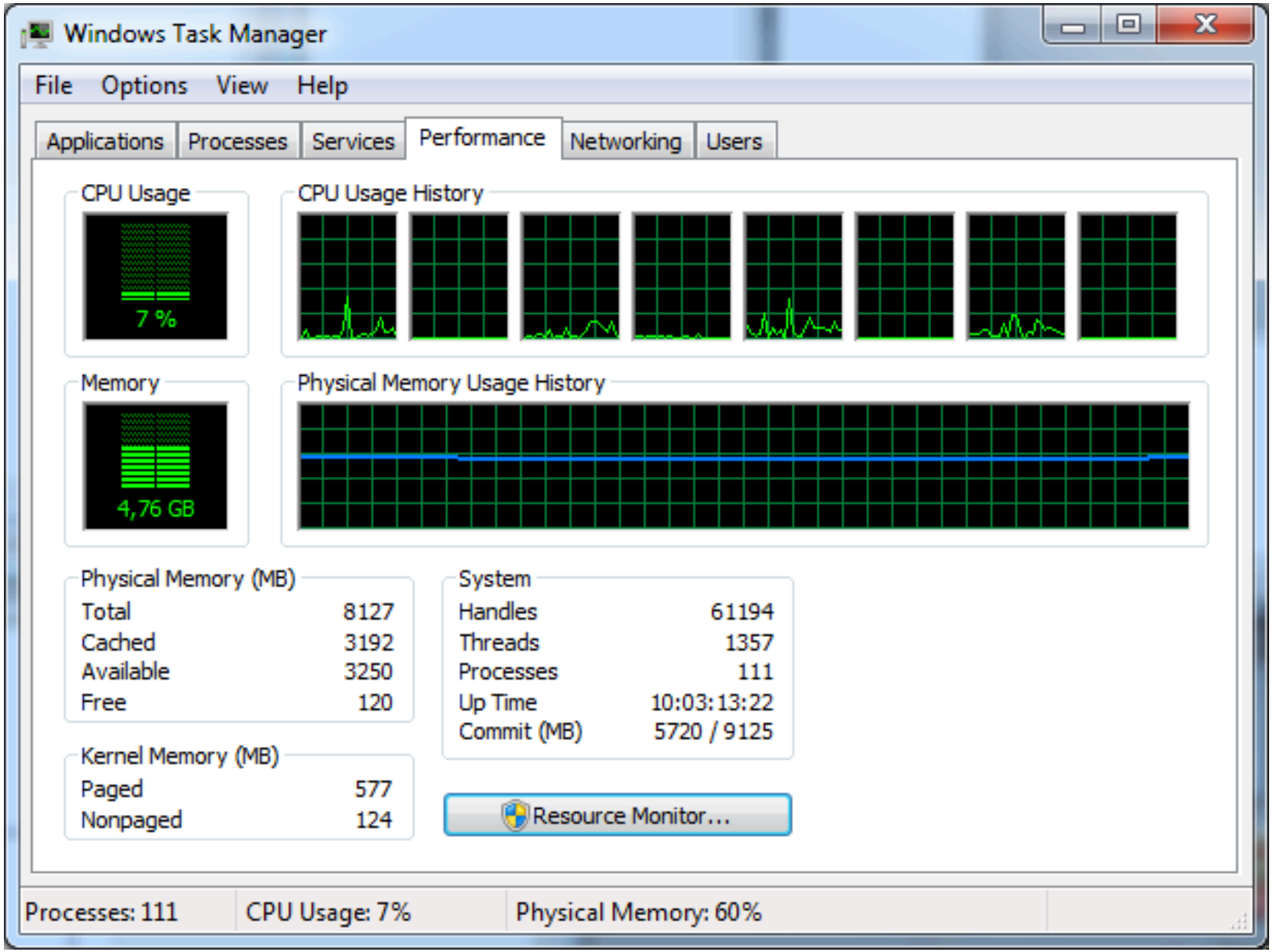
b) Kelnere venter på flere tallerkener

```
final Lock lock = new ReentrantLock();
final Condition notFull  = lock.newCondition();    // kø for Kokker
final Condition notEmpty = lock.newCondition(); // kø for Kelnere

public boolean putPlate (Kokker c) throws InterruptedException {
     lock.lock();
      try {while (numOnTable == MAX_ON_TABLE && numProduced < NUM_TO_BE_MADE){
                notFull.await(); // waiting for a place on the table
             }
            if (numProduced < NUM_TO_BE_MADE) {
                  numProduced++;
                  numOnTable++;;
                  notEmpty.signal(); // Wake up a waiting Kelner to serve
                  if (numProduced == NUM_TO_BE_MADE) {
                       //  I have  produced the last plate,
                      notEmpty.signalAll();   // tell Kelnere to stop waiting, terminate
                      notFull.signalAll();       // tell Kokker to stop waiting and terminate
                      return false;
                   }
                 return true;
            } else {  return false;}
      } finally {
         lock.unlock();
  }  } // end putPlate
```

```
public boolean getPlate (Kelnerw) throws InterruptedException {
    lock.lock();
    try {
        while (numOnTable == 0 && numProduced < NUM_TO_BE_MADE ) {
            notEmpty.await();      // This Kelner is waiting for a plate
        }
        if (numOnTable > 0) {
            numOnTable--;
            notFull.signal();  // Signal to one Kokk in the Kokker's waiting queue
            return true;
        } else {
            return false;}
    } finally {
        lock.unlock();
    }
} // end getPlate
```

En Kelner eller en Kokk blir signalisert av to grunner:
- for å behandle (lage eller servere) en tallerken til
- ikke mer å gjøre, gå hjem (tøm begge køene)

Løsning3 med 18 raske Kokker (venter 1 ms) og 2 langsomme Kelnere (venter 1000 ms). Kokkene stresser ikke maskinen med stadige mislykte spørsmål , men venter i kø til det er plass til en tallerken til på varmebordet . CPU-bruk = 7%.



Vurdering av de tre løsningene

- **Løsning 1**: Enkel, men kan ta for mye av CPU-tiden. Særlig når systemet holder av andre grunner å gå i metning vil typisk en av trådene da bli veldig treige, og da tar denne løsningen plutselig ½-parten av CPU-tiden.
- **Løsning 2:** God, men vanskelig å skrive
- **Løsning 3:** God, nesten like effektiv som løsning 2 og lettere å skrive.

Avsluttende bemerkninger til Produsent-Konsument problemet

- Invarianter brukes av alle programmerere (ofte ubevisst)
  - program, loop or metode (sekvensiell eller parallell)
  - Å si dem eksplisitt hjelper på programmeringen
- HUSK: synchronized/lock virker bare når alle trådene synkroniserer på samme objektet.
  - Når det skjer er det **sekvensiell tankegang** mellom wait/signal
- Når vi sier notify() eller wait() på en kø, vet vi ikke:
  - Hvilken tråd som starter
  - Får den tråden det er signalisert på kjernen direkte etter at den som sa notify(), eller ikke ??. Ikke definert
- Debugging ved å spore utførelsen (trace) – System.out.println(”..”)
  - Skriving utenfor en Locket/synkronisert metode/del av metode, så lag en:
    - synchronized void println(String s) {System.out.println(s);}
  - Ellers kan utskrift bli blandet eller komme i gal rekkefølge.

Hoare Monitors

- The concept of Monitors was originally presented by Per Brinch Hansen who put the concept into his language Concurrent Pascal
- Brinch Hansen's Monitors were refined and presented in a nice, clean version in his seminal paper: *Monitors: An Operating System Structuring Concept* published in 1974.