



INF2440 – Effektiv parallellprogrammering

Uke 1, våren2014

Arne Maus
OMS,
Inst. for informatikk



Hva vi skal lære om i dette kurset:

Lage parallelle programmer (algoritmer) som er:

- **Riktige**

- Parallele 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; 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.



Tre grunner til å lage parallelle programmer

- Skille ut aktiviteter som går **langsommere** i en egen tråd.
 - Eks: Tegne grafikk på skjermen, lese i databasen, sende melding på nettet. Asynkron kommunikasjon
- 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.
- Vi ønsker **raskere** programmer, raskere algoritmer.
 - Eks: Tekniske beregninger, søking og sortering.

Dette kurset legger nesten all vekt på raskere algoritmer



INF2440 - et **nytt** kurs

- Ikke alt er ferdig enda – mye vil bli skrevet ut over våren.
- Planlagt tre obliger - den første legges ut 24. jan og vil få innleveringsfrist 14 feb. Så ca 1 måned per oblig.
- En oblig er ikke bare innlevering av ett eller flere parallelle programmer, men også en liten rapport om hastighetsmålinger på disse for ulike størrelse av data.
- Opplegget av forelesningene:
 - En time «teori» , forklaring av problemer
 - En time mer praksis – framvisning av løsninger og drøfting av disse
- Gruppetimer:
 - Jobbing med ukeoppgaver og oblig
- Nytt kurs betyr at også dere selv vil være med på forme kurset, og at ikke alt vil være perfekt.

- Ingen egentlig 100% dekkende lærebok er funnet, men:
 - Det som foreleses **er pensum** – viktig.
 - **Lærebok:** 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 programeksempelene) i :
 - <http://www.sciencedirect.com/science/book/9780124159938>



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 :

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

- 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 forttere enn med 1 kjerne.
- Meget sjelden at det kan oppnås (mer om det siden)



Flynns klassifikasjon av datamaskiner:

	Single Instruction	Multiple Instruction
Single Data	SISD : EnkeltCore 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)	MIMD : klynge av datamaskiner, Multikjerne CPU



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 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



Mange 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

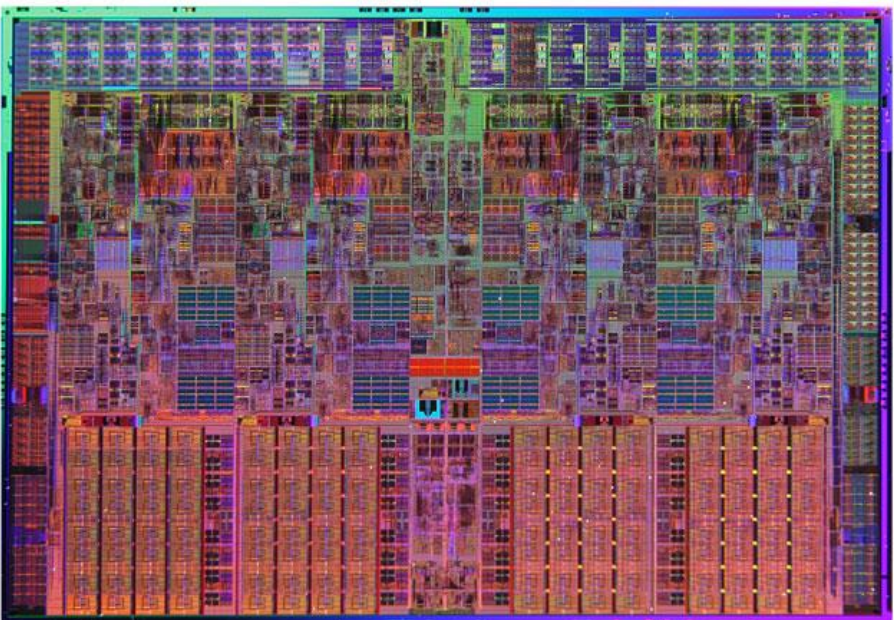
INF2240: Parallellprogrammering av ulike algoritmer med tråder på multikjerne CPU i Java – empirisk vurdert.



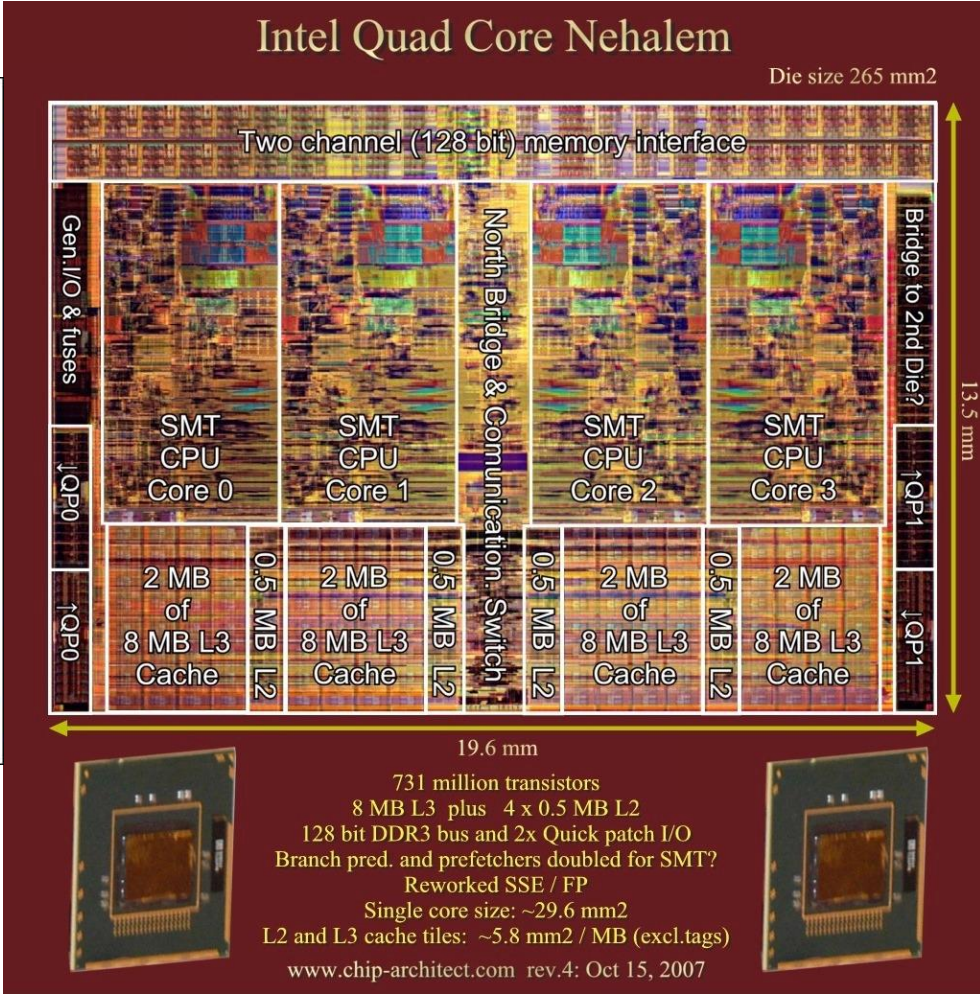
Maskinvare og språk for parallelle beregninger og Ifi-kurs

1. Klynger av datamaskiner - **INF3380**
 - jfr. Abelklyngen på USIT med ca 10 000 kjerner. 640 maskiner (noder), hver med 16 kjerner
 - C, C++, Fortran, MPI –de ulike programbitene i kjernene sender meldinger til hverandre
2. Grafikkort GPU med 2000+ små-kjerner, **INF5063**
 - Eks Nvidia med flere tusen småkjerner (SIMD – maskin)
 - Cell prosessoren
3. Multikjerne CPU (2-100 kjerner per CPU) **INF2440**
 - AMD, Intel, Mobiltelefoner,...
 - De fleste programmeringsspråk: Java, C, C++,...
4. Mange maskiner løst koblet over internett. **INF5510**
 - Planet Lab
 - Emerald
5. Teoretiske modeller for beregningene **INF2140, INF4140**
 - PRAM modellen og formelle modeller (f.eks FSM)

Multikjerne - Intel Multicore Nehalem CPU



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



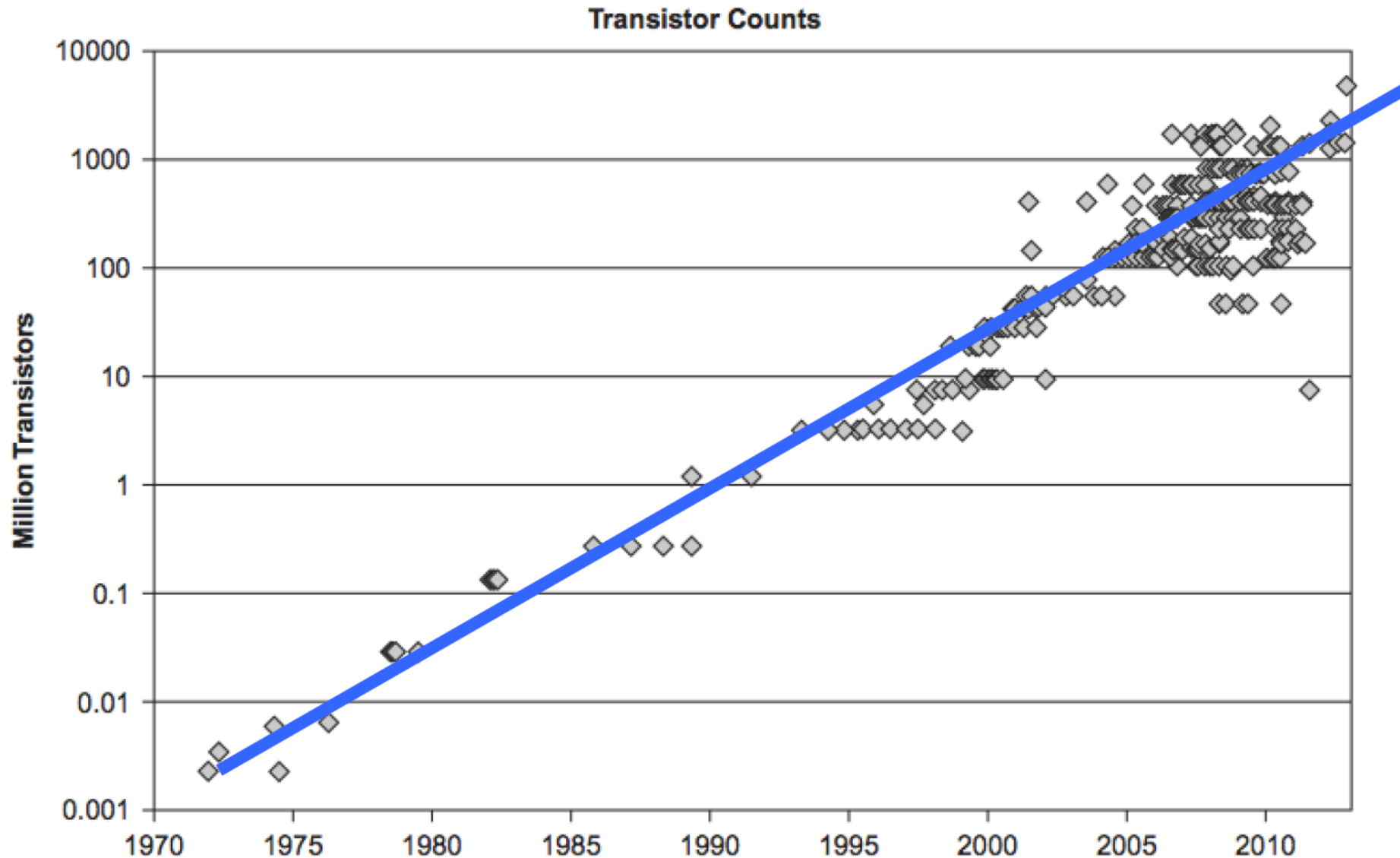


Hvorfor får vi multikjerne CPUer ?

- Hver 18-24 måned doubler 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 kokeplater).
- Med f.eks dobbelt så mange transistorer ønsker vi oss egentlig en dobbelt så rask maskin, men det vi får er dessverre dobbelt så mange CPU-kjerner.
- Med flere regnekverner (kjerner) må vi få opp hastigheten ved å lage parallelle programmer !

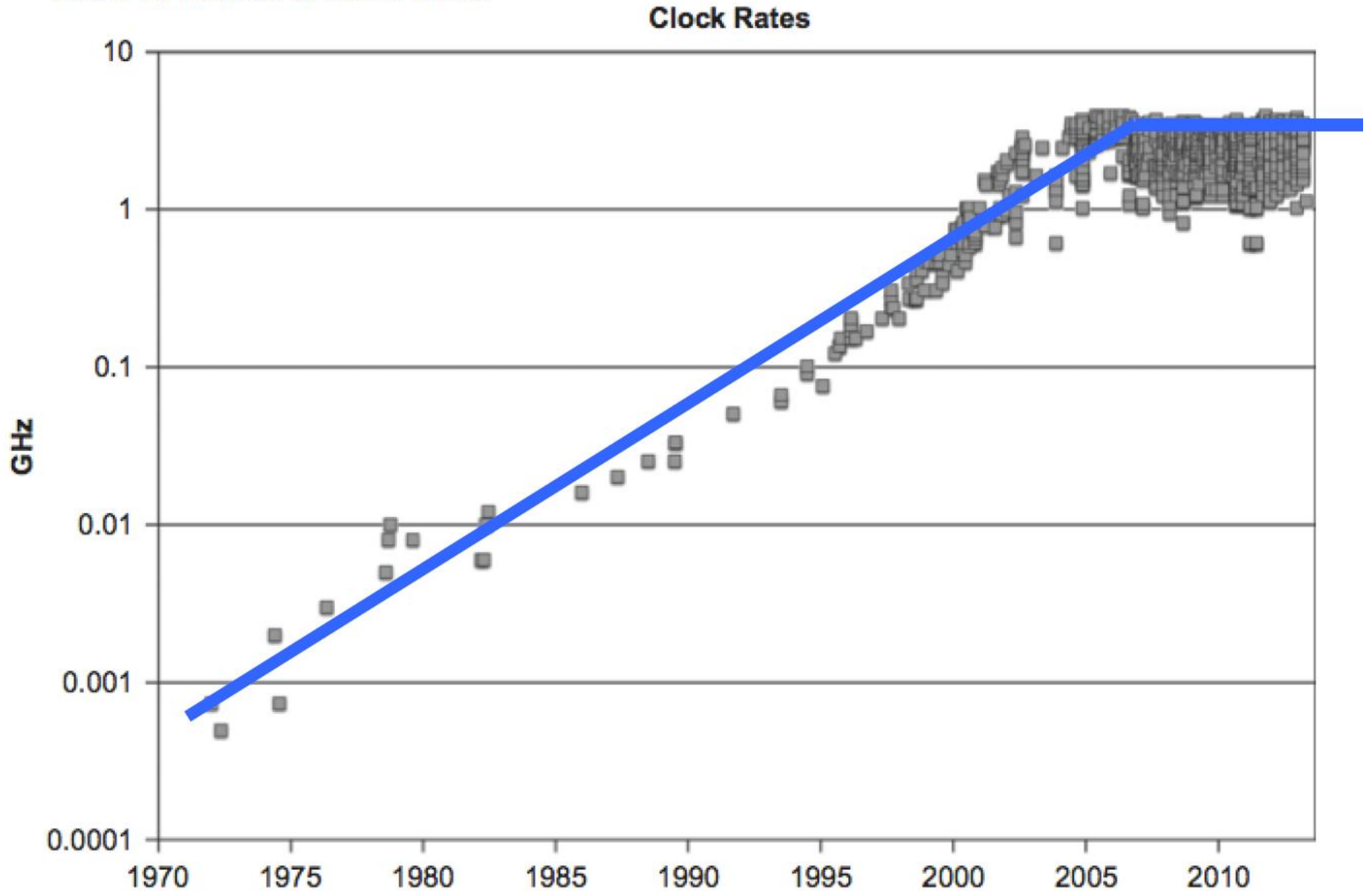
Transistors per Processor over Time

Continues to grow exponentially (Moore's Law)

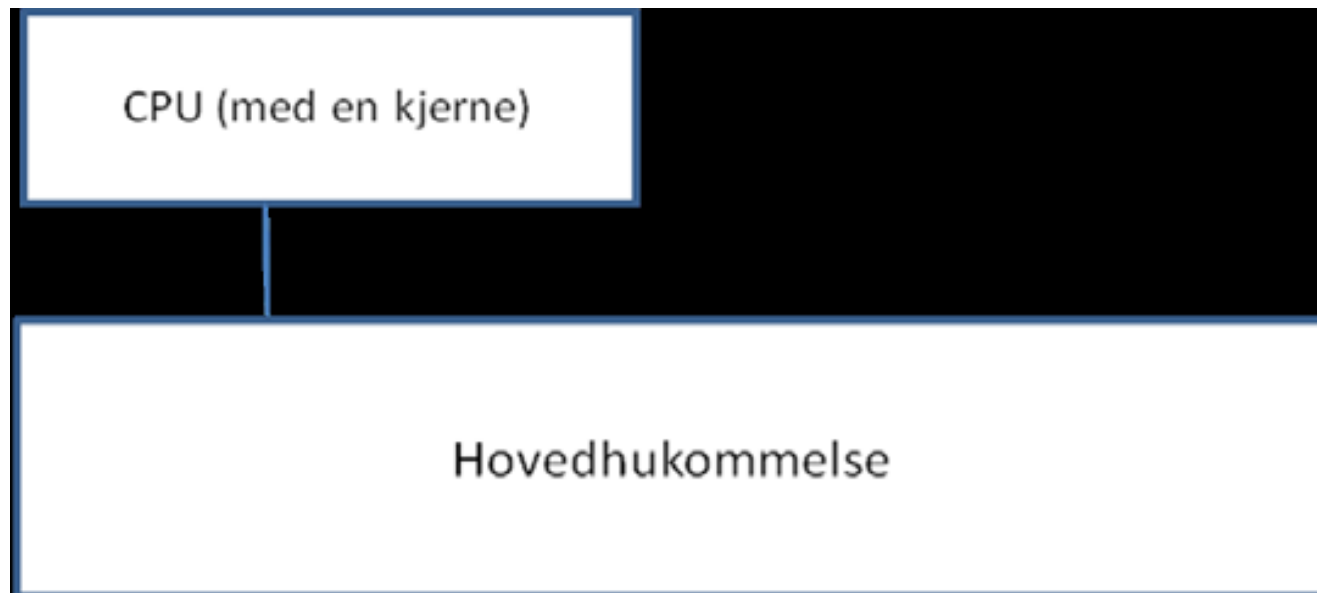


Processor Clock Rate over Time

Growth halted around 2005

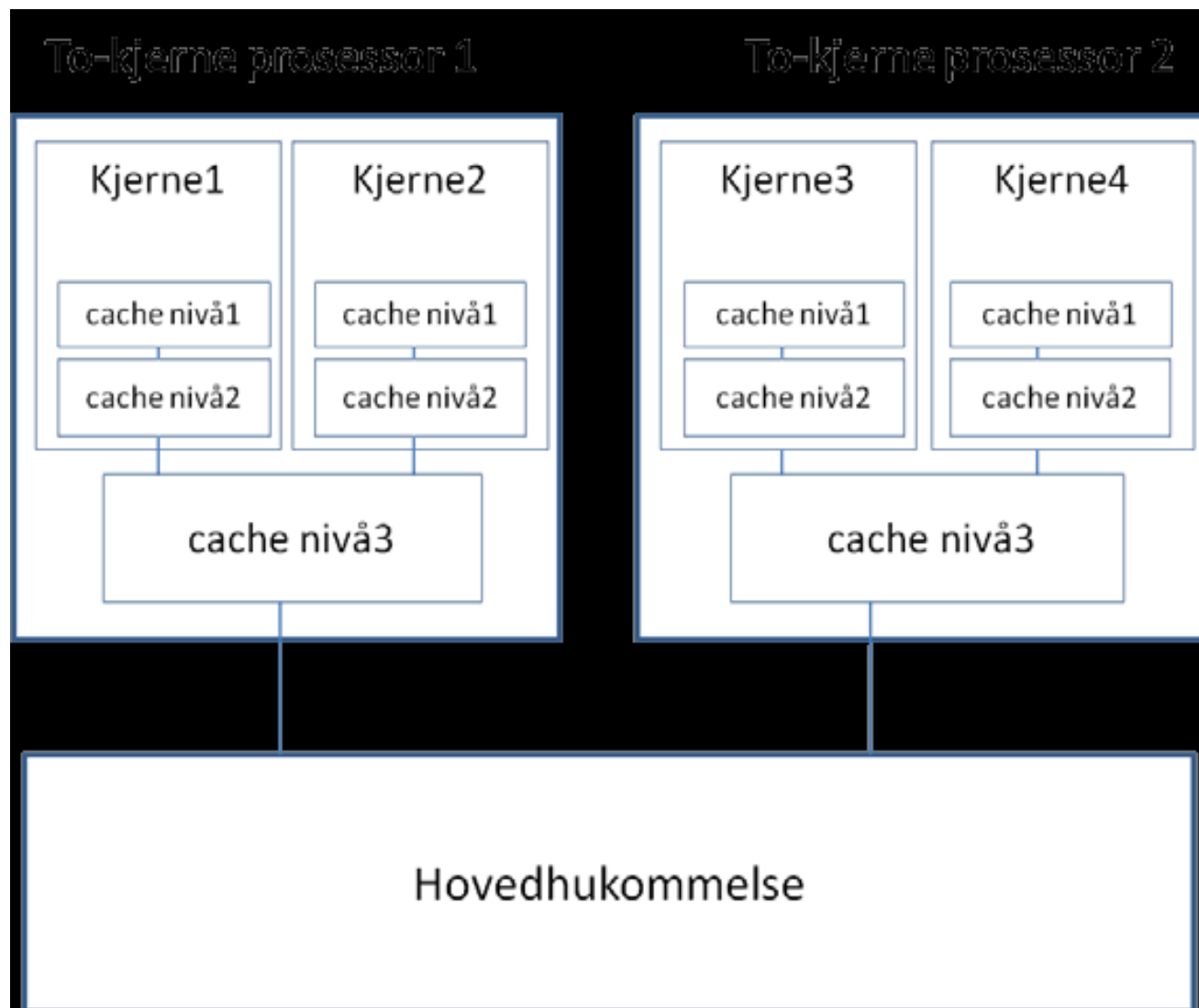


Maskin 1980 (uten cache)

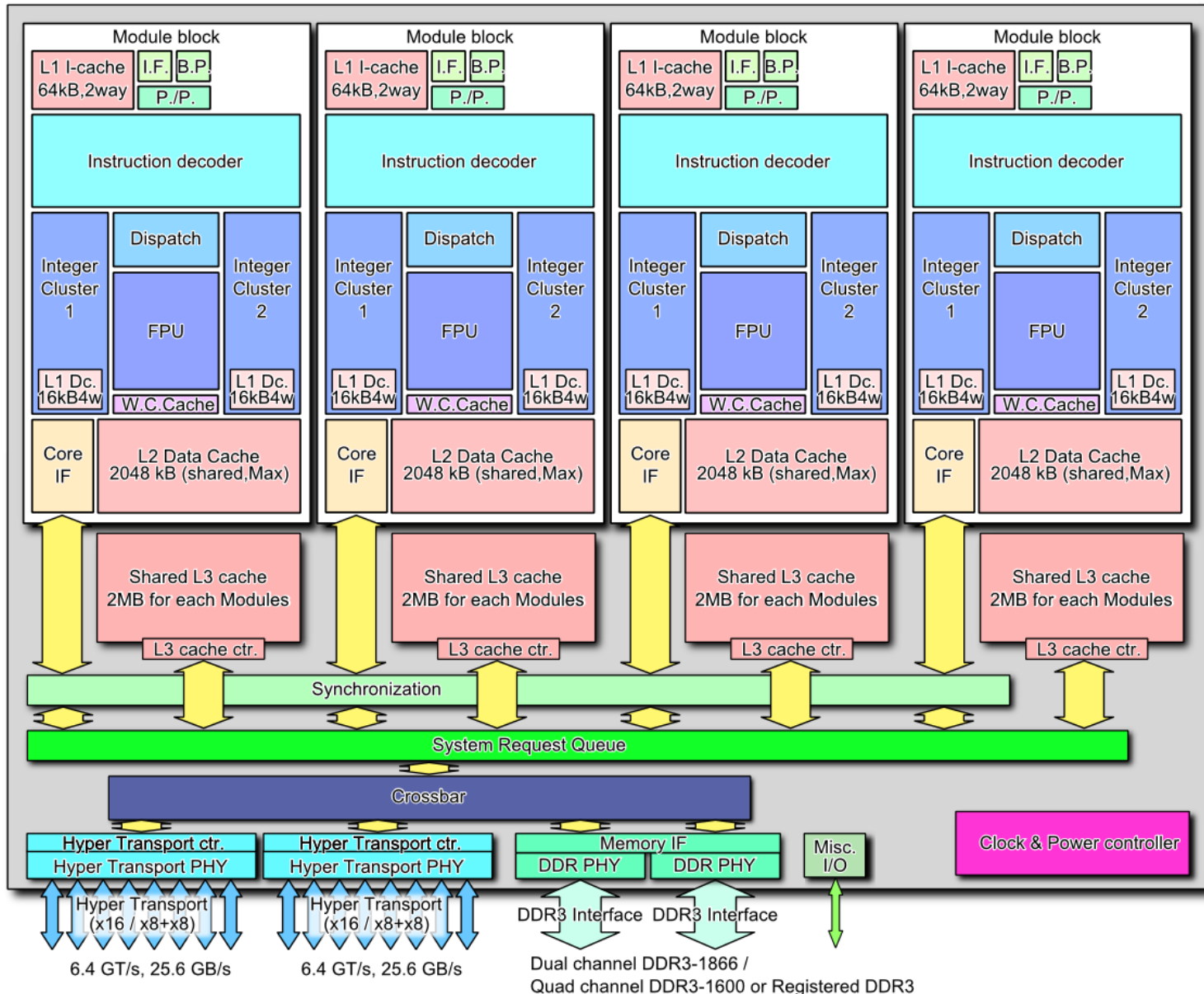


Figur 19.1 Skisse av en datamaskin i ca. 1980 hvor det bare var én beregningsenhet, en CPU, som leste sine instruksjoner og både skrev og leste data (variable) direkte i hovedhukommelsen.

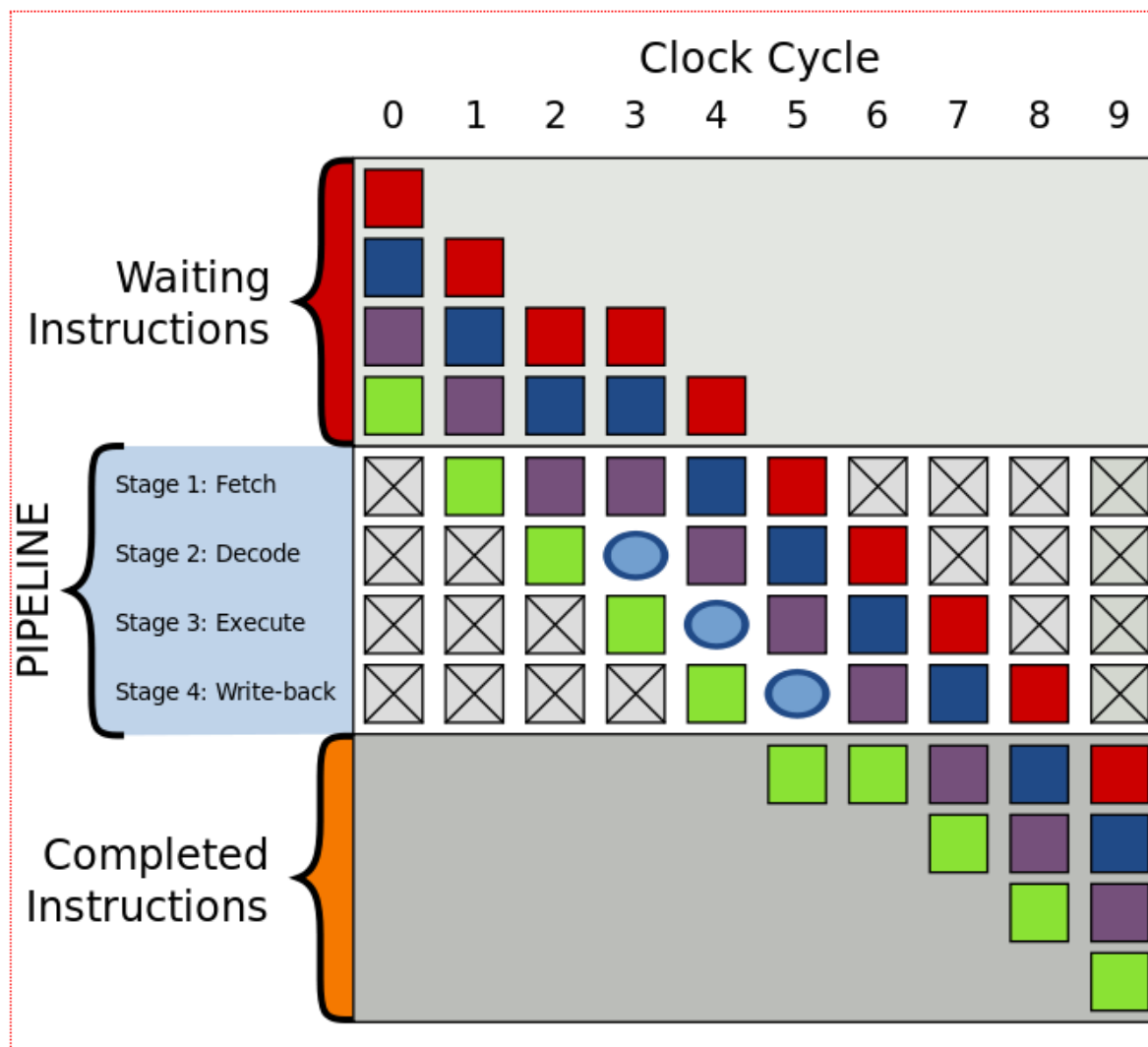
Maskin ca. 2010 med to dobbeltkjerne CPU-er



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



Instruksjonsparallelitet i en CPU-kjerne. Pipeline – flere instruksjoner utføres samtidig i raskest mulig rekkefølge.



Testing av forsinkelse i data-cachene

- Programmet : latency.exe (fra CPUZ)

```
M:\INF2440Para\CPU-old\CPU\CPU\latency.exe
```

stride	4	8	16	32	64	128	256	512
size (Kb)								
1	7	7	7	7	7	3	3	3
2	3	6	4	4	3	3	3	3
4	7	3	3	3	7	3	3	3
8	4	3	3	4	3	3	3	3
16	3	3	6	3	3	7	4	3
32	3	3	3	3	7	4	4	6
64	4	3	6	6	8	8	11	13
128	3	4	5	6	9	8	11	12
256	4	3	4	7	9	10	16	16
512	4	7	4	6	10	27	45	46
1024	3	4	8	6	14	27	42	42
2048	3	4	5	6	14	27	45	45
4096	3	4	5	10	10	27	44	45
8192	4	4	4	6	23	35	89	63
16384	3	4	4	10	12	88	158	167
32768	3	8	5	6	10	88	165	169

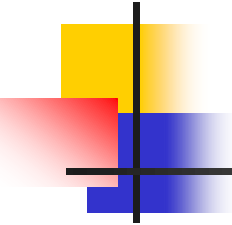
3 cache levels detected

Level 1	size = 32Kb	latency = 3 cycles
Level 2	size = 256Kb	latency = 11 cycles
Level 3	size = 8192Kb	latency = 43 cycles



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

- Hukommelses-systemet i en multicore CPU (Intel Core i/ 870 3.07 GHz) – mange lag (typisk aksesstid i instruksjonssykler):
 1. Registre i kjernen (1) – 8/16 registre
 2. L1 cache (3) – 32 Kb
 3. L2 cache (13) – 256 kb
 4. L3 cache (43) – 8Mb
 5. Hovedhukommelsen (virtuell hukommelse) (170) – 8-16 GB
 6. Disken (5 000 000) = 5 ms – 1000 GB

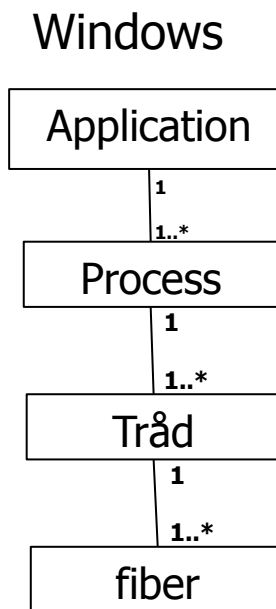
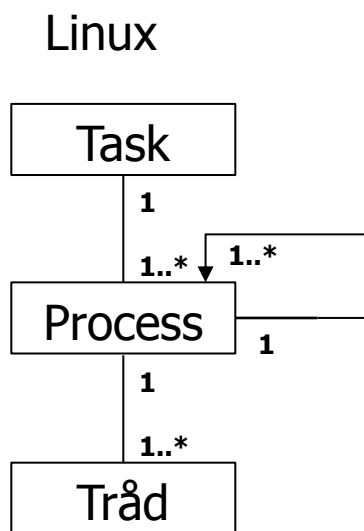


Vi kan ikke hente mer fra automatisk forbedring av hastigheten på våre programmer:

- **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 fullt utnyttet – ikke mer å hente
 - **Kompilatoren** – Java (etter ver 1.3) kompilerer videre til maskinkode og optimaliserer mye. JIT-kompilering. Ikke mulig å gjøre særlig mer effektiv
- ⇒ **Konklusjon** Skal vi ha raskere programmer, må vi som programmerere *se/v* skrive parallelle løsninger på våre problemer.

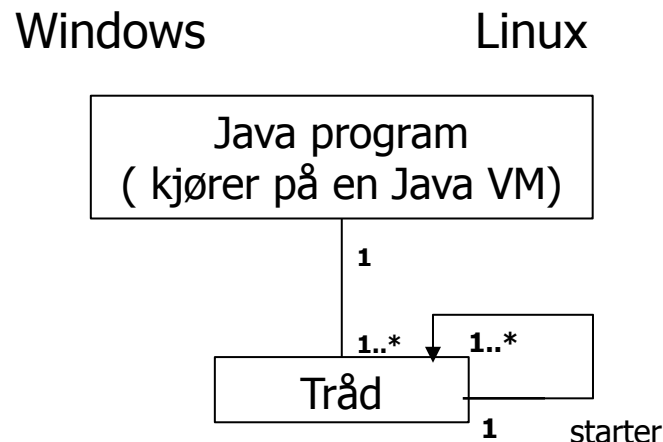
Operativsystemet og tråder

- 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



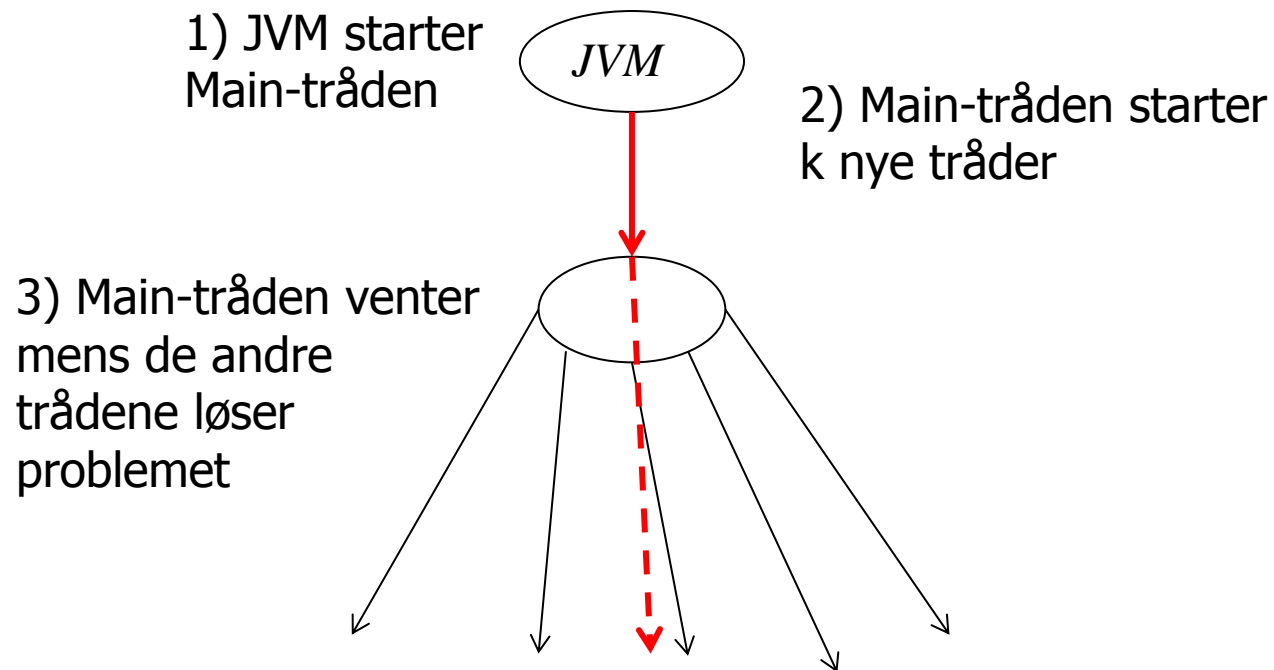
- **Alle trådene i et Java-program deler samme adresserom** (= samme plasser i hovedhukommelsen), og alle trådene kan lese og skrive i de variable (objektene) programmet har og ha adgang til samme kode (metodene i klassene).



Hva er tråder i Java ?

- I alle programmer kjører minst en tråd – main tråden (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`)
- 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 class's `run` method does nothing.

- **Runnable target** er :
 - En klasse som implementerer grensesnittet 'Runnable'
- Det er en annen måte å starte en 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 program – og kjører på en 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 de greit **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 et som implementerer Runnable.
- Det ser ut som om man får to objekter av klassen Thread, men man får bare ett `[Thread t = new Thread(new Arbeider());]`
- 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 til sist kalles.

JVM som inneholder sin del av `start()` som gjør mye og til slutt kaller `run()`

Vårt program kaller `start` i vårt objekt av en subklasse av Thread (eller Runnable). Etter start av tråden kalles vår `run()`

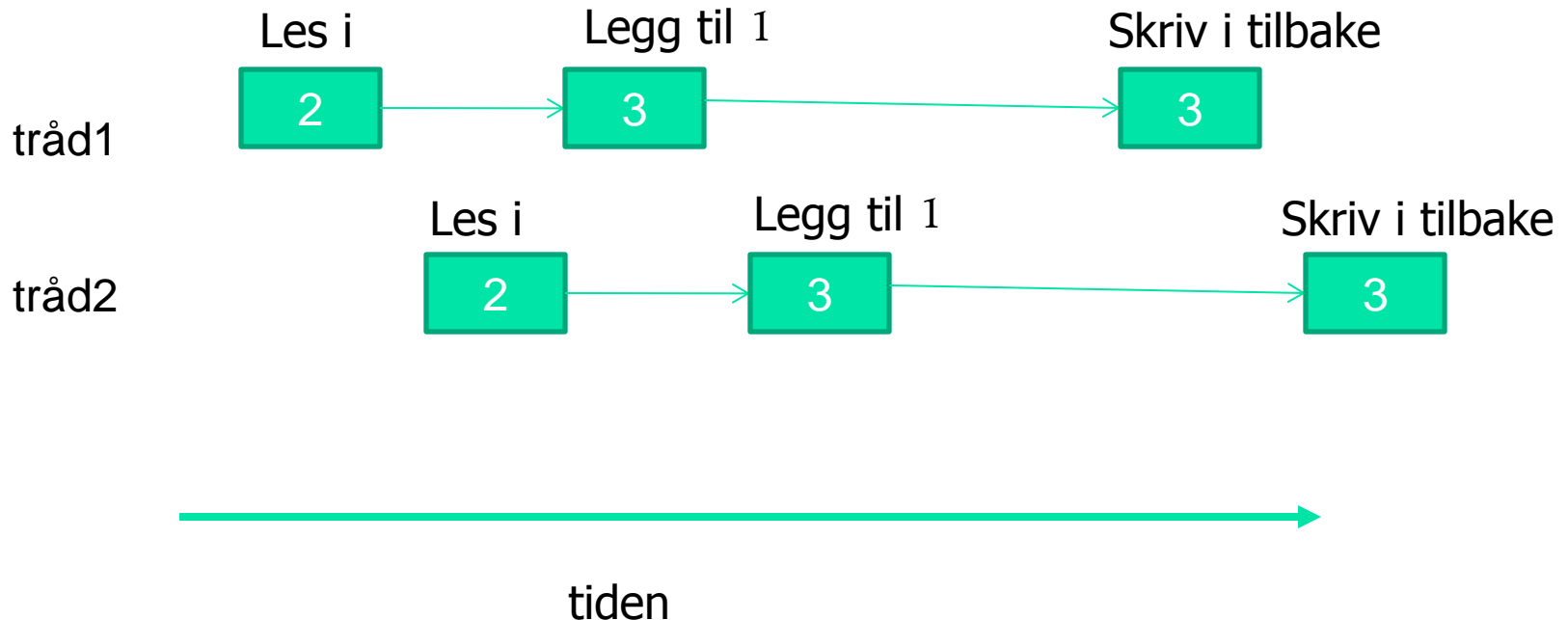


Flere problemer parallellitet med tråder i Java

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

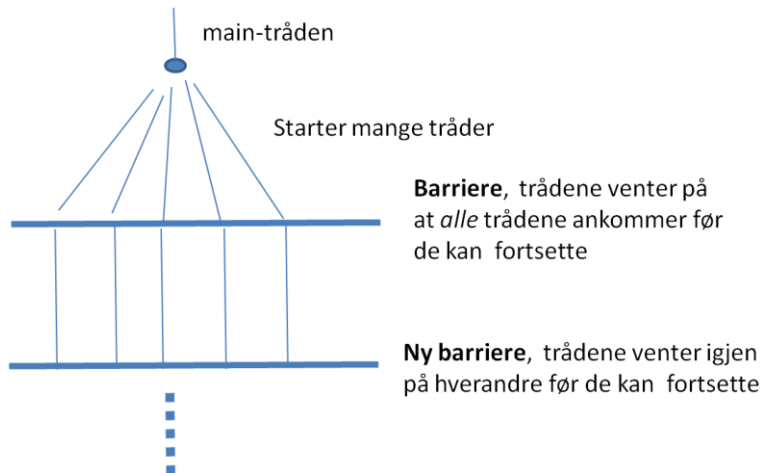
- Setter i gang **n tråder** (på en 2 kjerner CPU) som alle prøver å øke med 1 en felles variabel int i; 100 000 ganger uten synkronisering;

```
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 et objekt **b** av klassen CyclicBarrier med et tall **ant** til konstruktoren = 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** 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

    //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, ogsaa 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

```



Oppsummering – Uke1

- Vi har gjennomgått hvorfor vi får flere-kjerne CPUer
- Tråder er måten som et Javaprogram bruker for å skape flere uavhengige parallelle programflyter i tillegg til main-tråden
- Tråder deler felles adresserom (data og kode)
- Stygg feil vi kan gjøre: Samtidig oppdatering (skriving) på delte data (eks: i++)
 - Dette løses ved synkronisering .
 - Alle tråder som vil skrive må køes opp på **samme** synkroniseringsvariabel/objekt slik at bare én tråd slipper til av gangen.
 - Alle objekter kan nyttes som en synkroniseringsvariabel, og da kan vi bruke enten en synchronized metode,
 - eller objekter av spesielle klasser som:
 - CyclicBarrier
 - Semaphore (undervises senere)
 - De inneholder metoder som `await()`, som gjør at tråder venter.