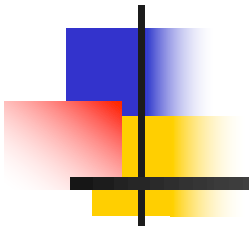


IN3030 – Effektiv parallellprogrammering
Uke 1, våren 2020

Eric Jul

Professor

PSE

Institutt for Informatikk

Litt om Eric

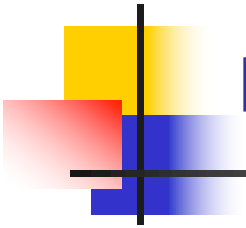
- **Ph.D. University of Washington, 1989**
- **Dansk-amerikaner**
- **Bor i Danmark – pendler til Oslo ca 3-4 gange/måned**
- **1989-2000: Førsteamauensis Københavns Universitet**
- **2000-2009: Professor Københavns Universitet**
- **2009-2015: Bell Labs, Dublin & Professor II, IfI**
- **2016- Professor IfI**

En beslutning: hvilket språk?

- **Skal jeg forelese på dansk?**
- **Or English?**
- **Recording lectues: I will attempt BUT I make no promises – my recoding last Fall did not work ☹**
- **Lecture slides are in Norwegian (perhaps a little Danish intermixed in a few places…)**
- **Please keep up with the messages on the web site**
- **Use the Hjeplelærer ☺**
- **Ask questions when in doubt**

Bakgrunn IN3030

- **Kurs er fra 2014 – tidligere INF2440**
- **Lavet av Arne Maus – nu Emeritus**
- **Overtaket av Eric 2018**
- **I 2019: INF2440 -> IN3030**



Motivation for Parallelle Programmer

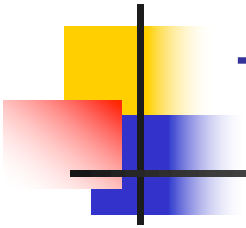
- **Maskiner kan bestå av flere CPU-er**
  - Hver CPU kan utføre programmer uavhengig av de andre CPUer
- **Hver CPU kan have flere kjerner**
  - Hver kjerne kan utføre programmer
- Vi vil gjerne utnytte disse muligheter for parallellisme

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).



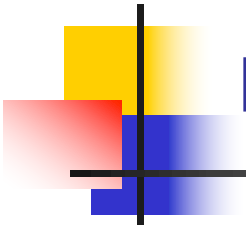
Tre grunner til å lage parallelle programmer

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 - et **nytt** kurs – og dog

- IN3030 er 90% baseret på INF2440.
- Et relativt UNIKT kurs – set internationalt.
- Planlagt fem obliger - den første legges ut 23. jan. Så ca. 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
- Kurs under utvikling betyr at også dere selv vil være med på forme kurset, og at ikke alt vil være perfeekt.



Pensum

- Ingen dekkende lærebok er funnet, men:
  - **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

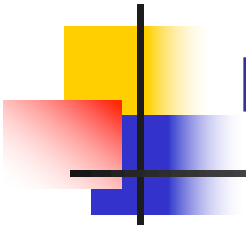
(Hvis leses utenfor Ifi, så koster det $!)

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:

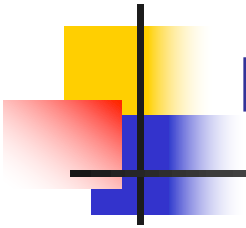
$$S = \frac{tid\ (sekvensiell\ algoritme)}{tid\ (parallell\ algoritme)}$$

- 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?



Lineær speedup analogier

- Selvsagt ønsker vi lineær speedup, MEN:
- Nogle problemer 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 umulige at parallellisere: Eksempel: hvis 1 kvinne kan lage et barn på 9 måneder, kan 9 kvinner så lage et barn på 1 måned?

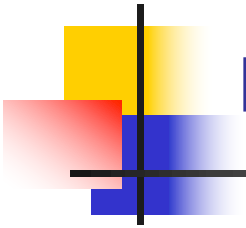
Flynns klassifikasjon av datamaskiner:

	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**

Eksempel på en SSSE 3-instruksjon (Intel i7)

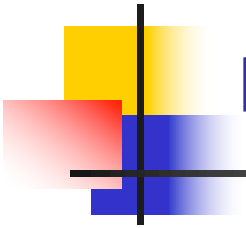
PHADDW, PHADDD	Packed Horizontal Add (Words or Doublewords)	takes registers A = [a0 a1 a2 …] and B = [b0 b1 b2 …] and outputs [a0+a1 a2+a3 … b0+b1 b2+b3 …]

- Det er mange registre (opp til 32) på en kjerne for bl.a. slike instruksjoner
- Et register kan sees på som en cache 0 – det tar bare én instruksjons-sykel å aksessere et register mot 3 sykler i cache1
- (i en 3 GHz CPU er en sykel 1/3 ns = 1/3 milliard dels sekund)



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

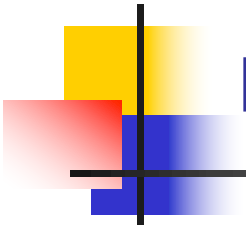


Flere mulige synsvinkler

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

INF2440: Parallellprogrammering av ulike algoritmer med tråder på multikjerne CPU i Java – empirisk vurdert ved tidsmålinger.



Learning-by-doing

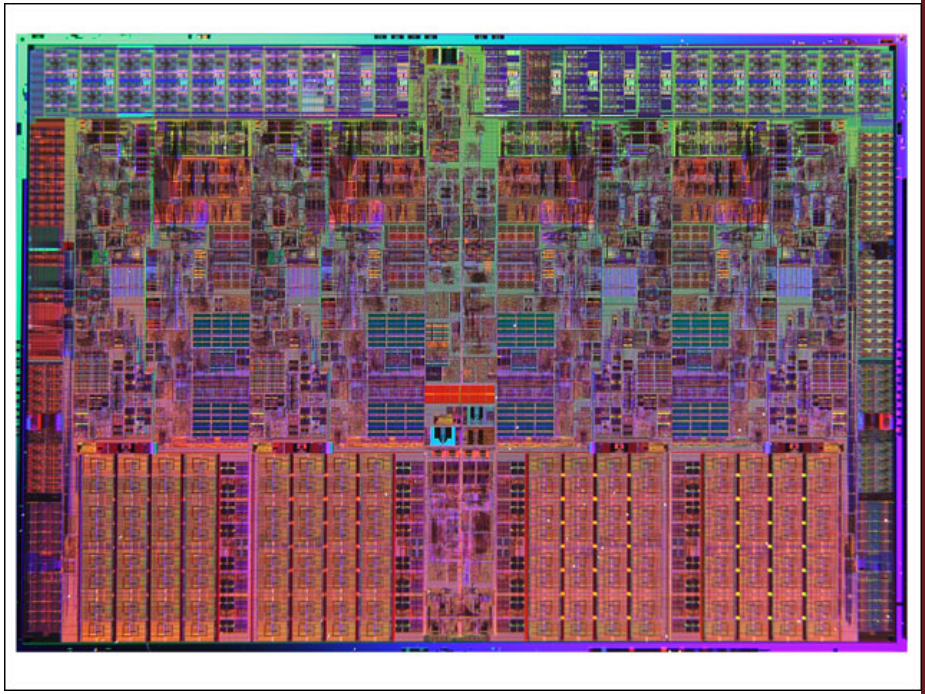
Dette kurs er et Learning-by-doing kurs!

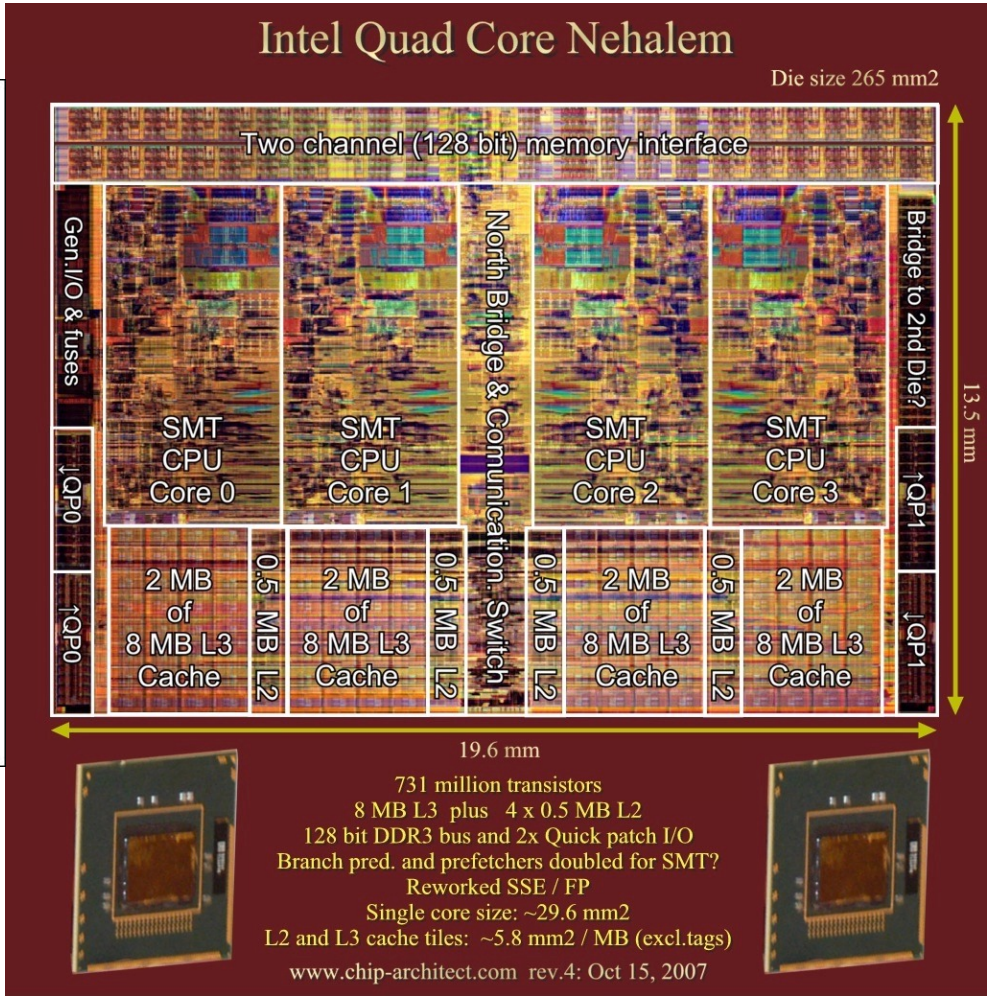
Utbyttet ditt er avhengig av DIN innsats!

Maskinvare og språk for parallelle beregninger og Ifi-kurs

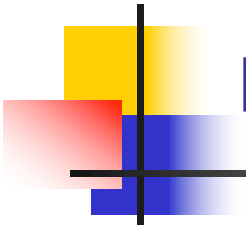
1. Klynger av datamaskiner - **INF3380**
   - jfr. Abelklyngen på USIT med ca 10 000 kjerner. 640 maskiner (noder), hver med 16 kjerner
   - C, C++, Fortran, MPI –de ulike programbitene i kjernene sender meldinger til hverandre
2. Grafikkkort GPU med 2000+ små-kjerner, **INF5063**
   - Eks Nvidia med flere tusen småkjerner (SIMD – maskin)
3. Multikjerne CPU (2-100 kjerner per CPU) **INF2440**
   - AMD, Intel, ARM, Mobiltelefoner,..
   - De fleste programmeringsspråk: Java, C, C++,..
4. Mange maskiner løst koblet over internett. **INF5510**
   - Planetlab – world-wide testbed
   - Emerald – språk til distribuert programmering
5. Teoretiske modeller for beregningene **INF4140**
   - 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

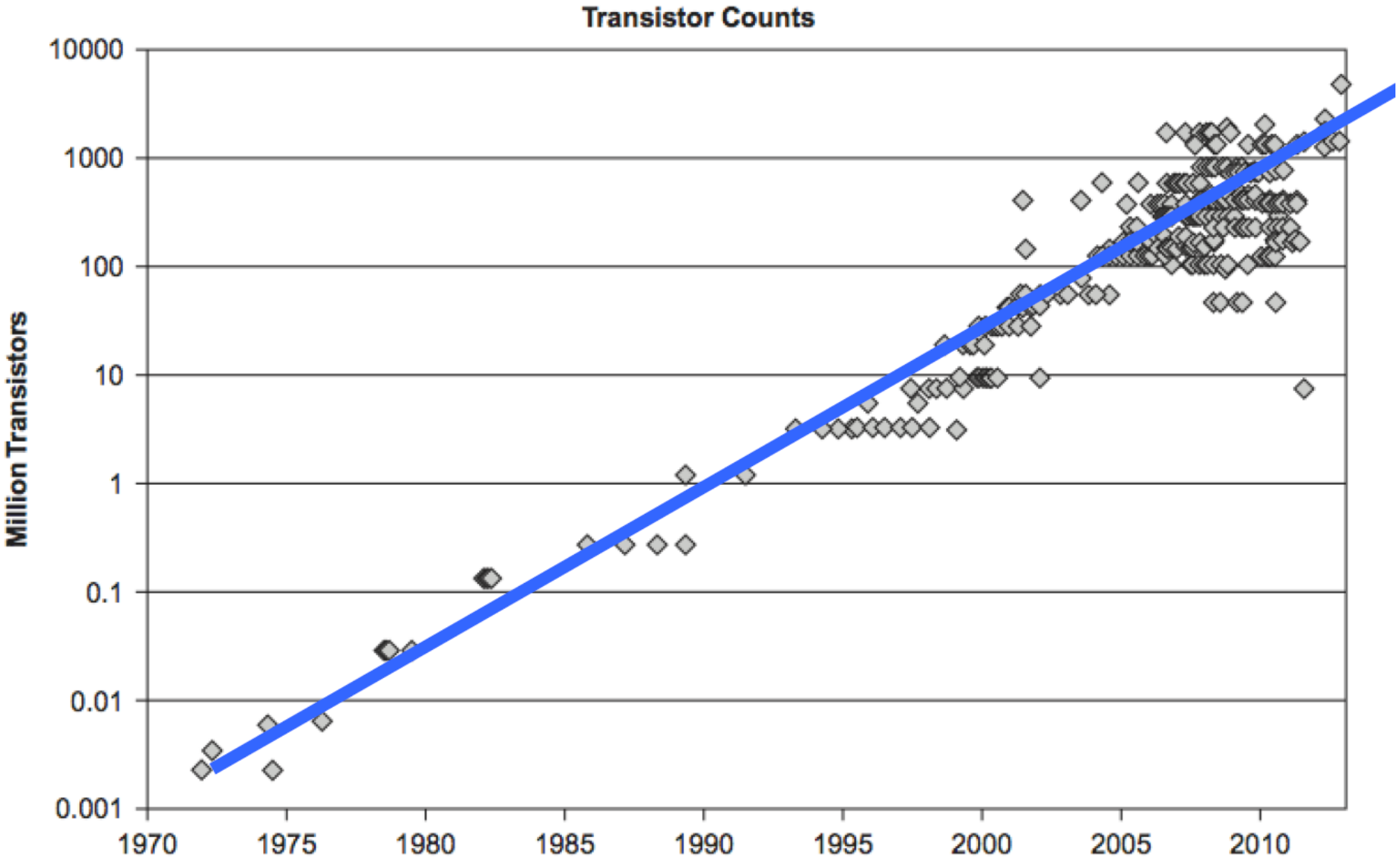


Hvorfor får vi multikjerne CPUer ?

- 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 !

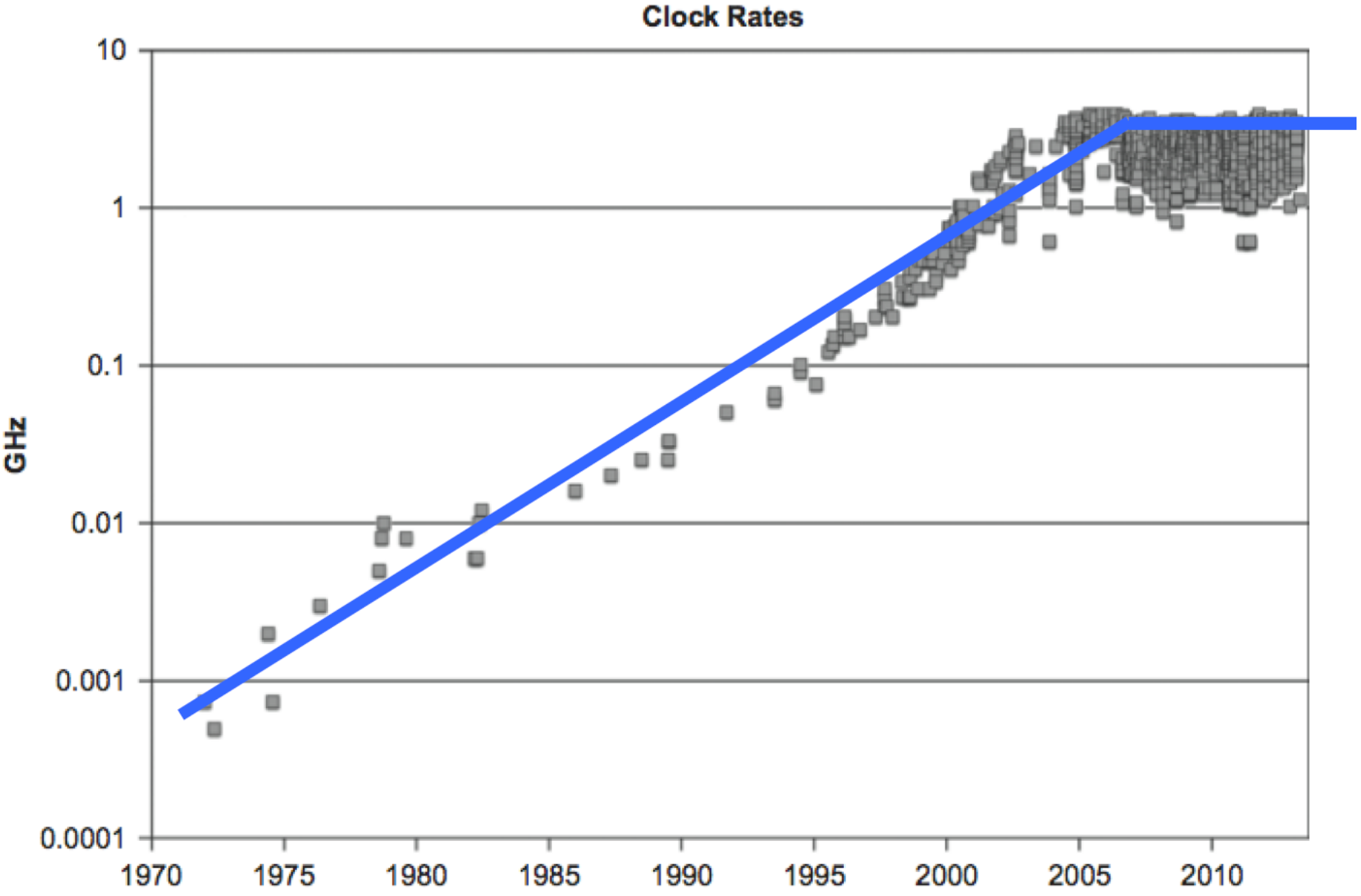
Transistors per Processor over Time

Continues to grow exponentially (Moore's Law)



Processor Clock Rate over Time

Growth halted around 2005



Helt avgjørede for oss – cache-hukommelse

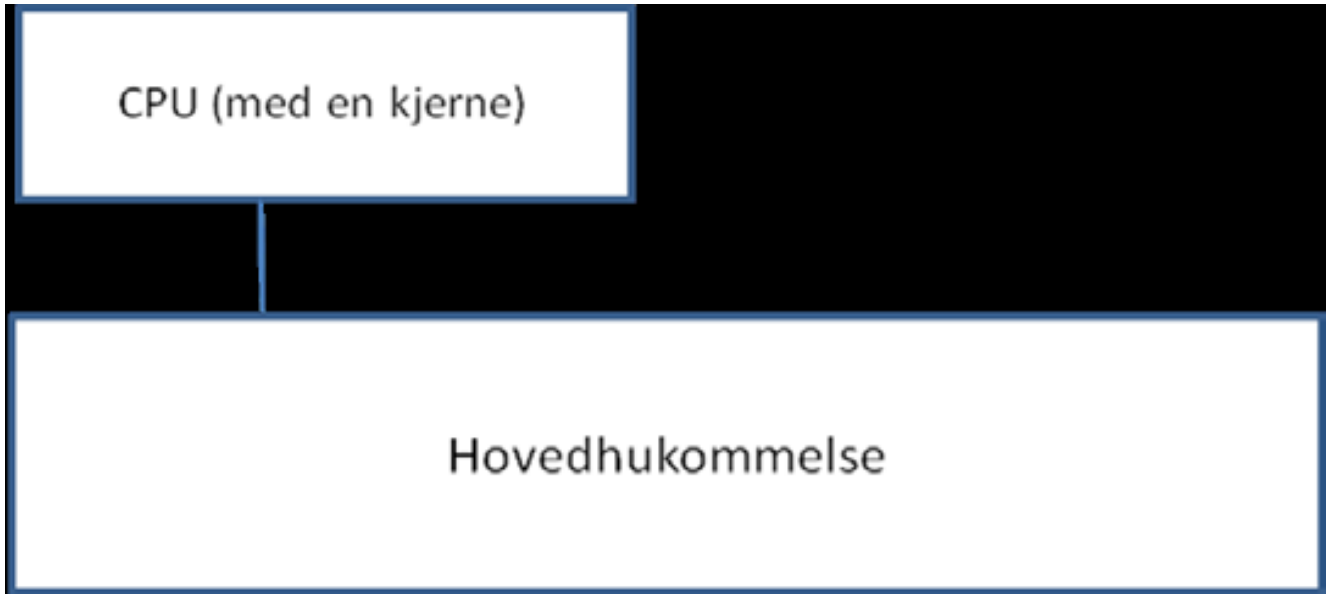
- Hva er cache
  - Raskere (men også dyrere) hukommelse mellom hovedlageret og kjernene.
  - Vi må ha cache fordi det er så store hastighetsforskjeller mellom en CPU-kjerne og hovedlageret ('main memory')
  - Ofte nå 3-4 lag med cache hukommelser + et antall registere i kjernen (enda raskere enn cache-hukommelsene) 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)

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,..)

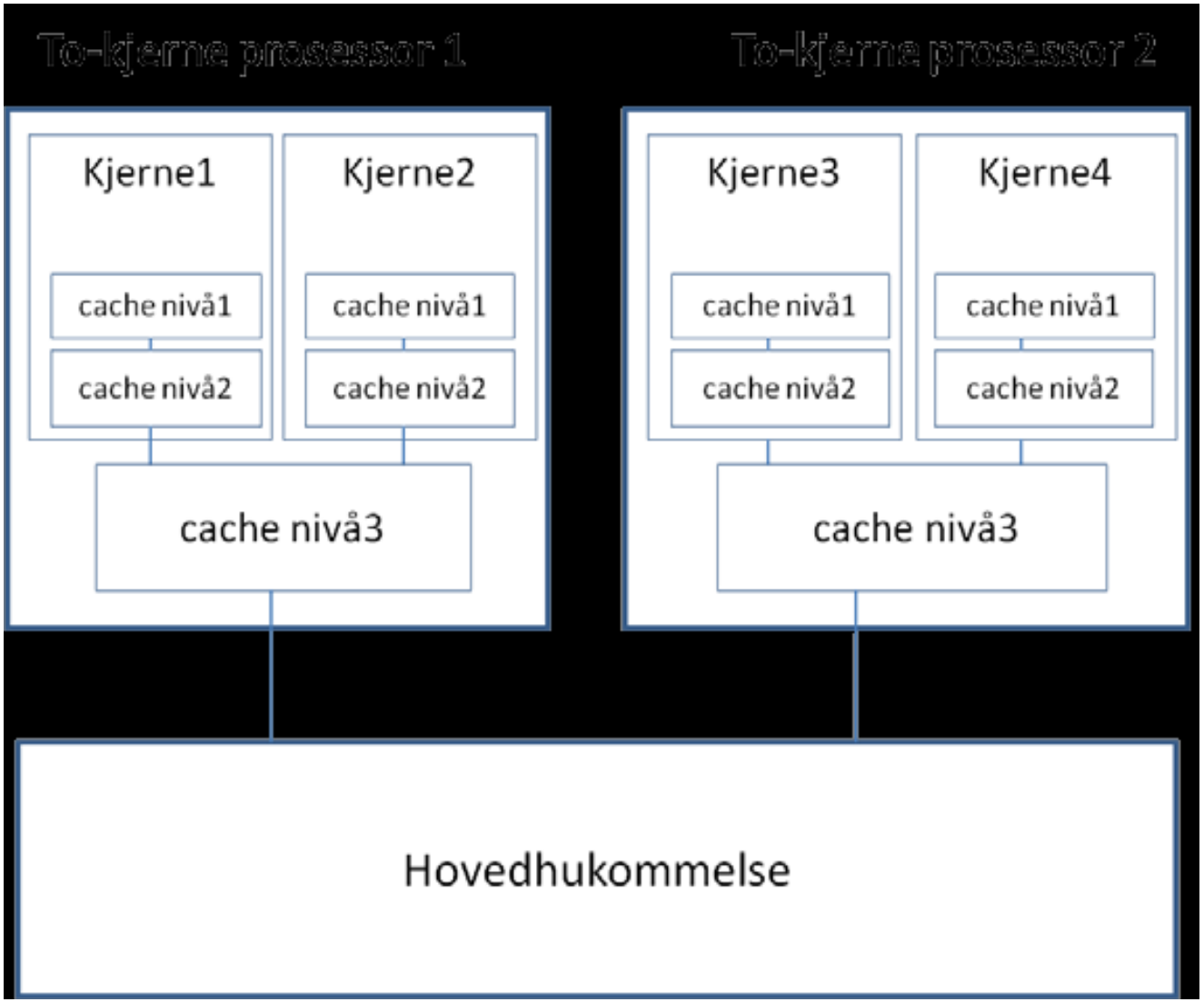
Vi må lage slike cache-vennlige programmer !

Maskin 1980 (uten cache)

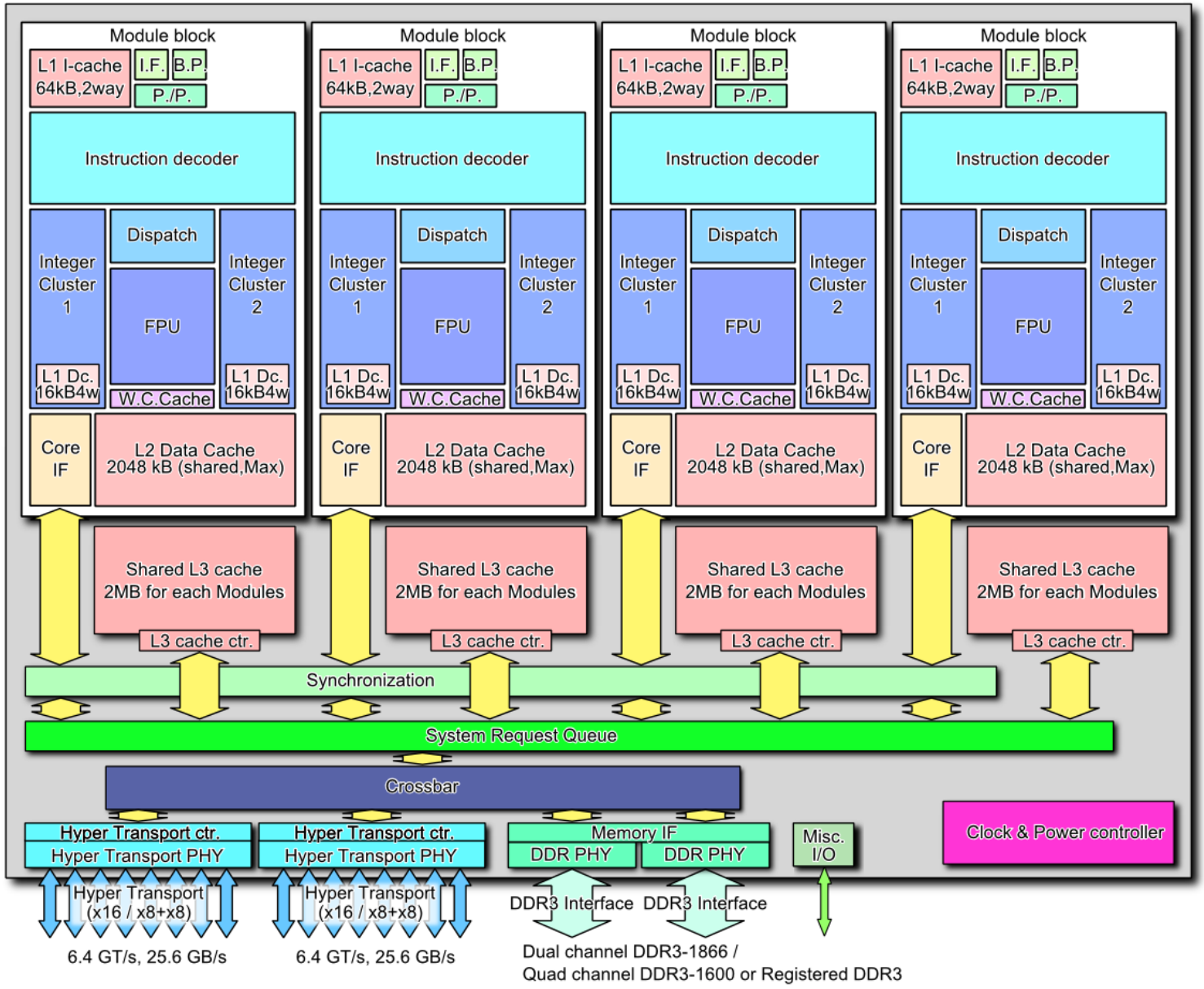


Figur 19.1 Skisse av en datamaskin i ca. 1980 hvor det bare var én beregningsenhet, en CPU, som leste sine instruksjoner og både skrev og leste data (variable) direkte i hovedhukommelsen. Intel 8080: 1 MHz CPU

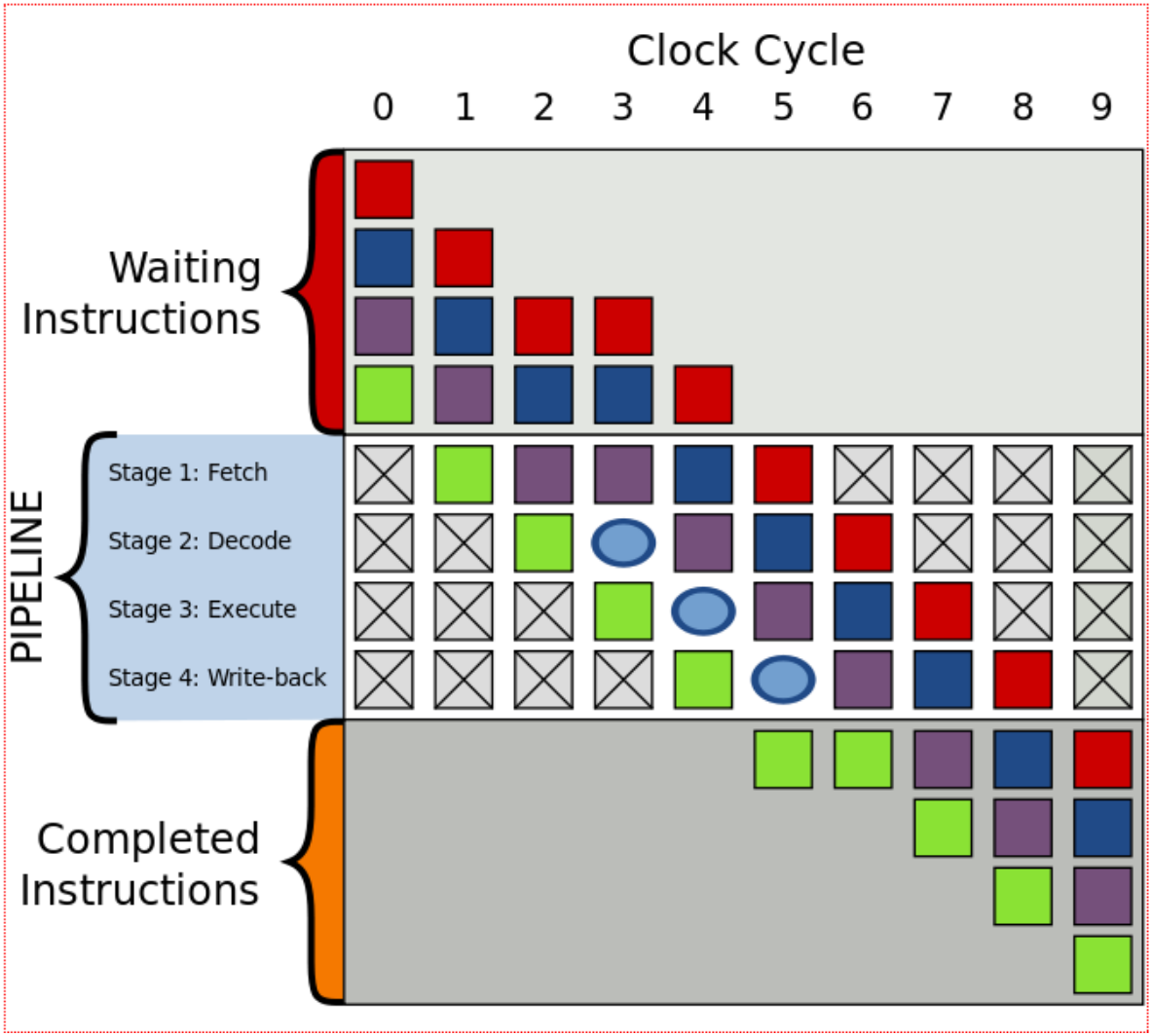
Maskin ca. 2010 med to dobbeltkjerne CPU-er



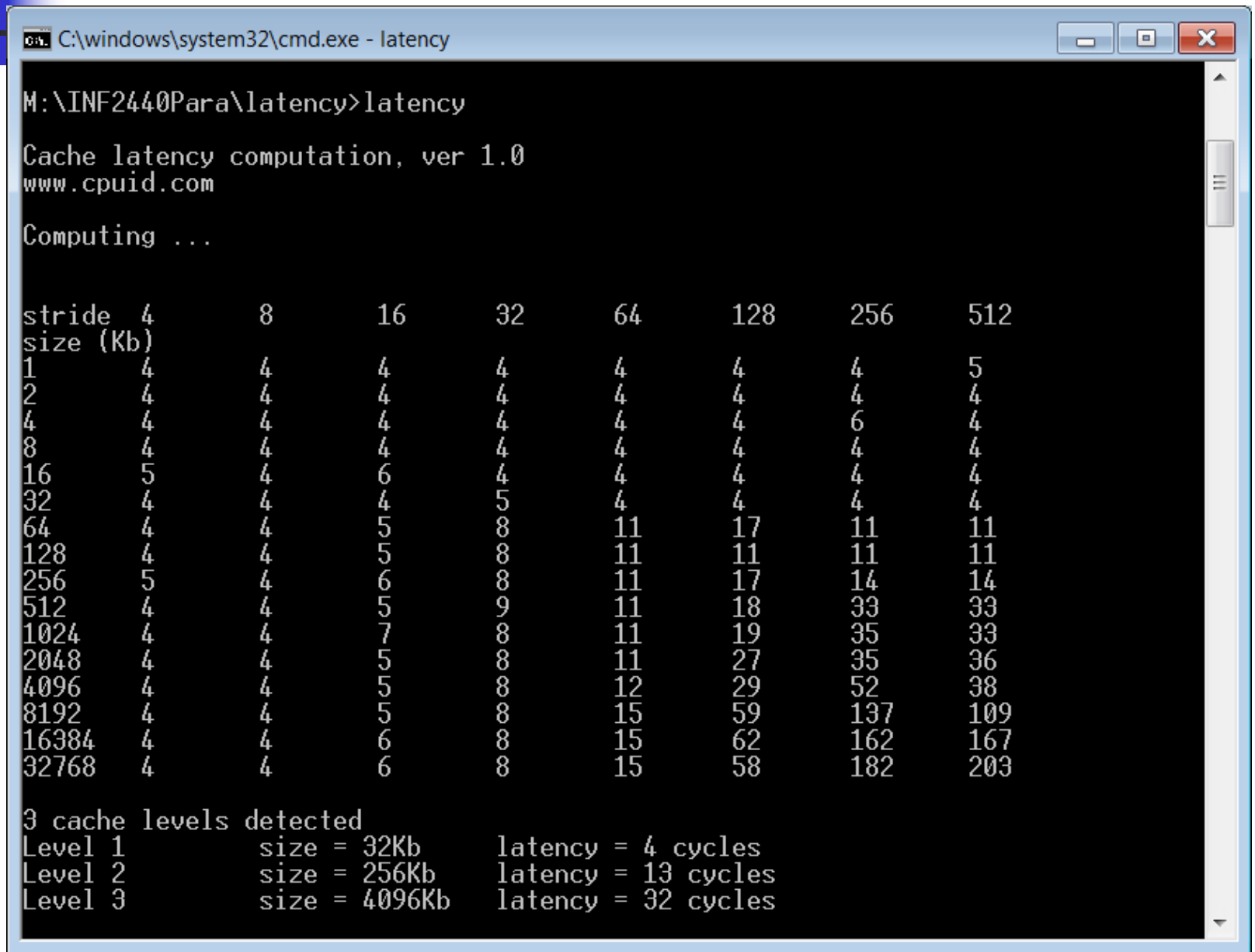
Hukommelses-systemet i en 4 kjerne CPU – mange lag og flere ulike beregningsmoduler i hver kjerne.:



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

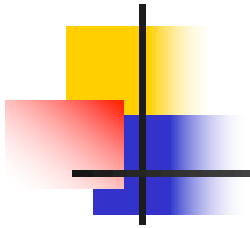
- Hukommelses-systemet i en multicore CPU ,Intel Core i5-459 3.3 GHz, – mange lag (typisk aksesstid i instruksjonssykler):

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

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.



END OF LECTURE F1, 2020