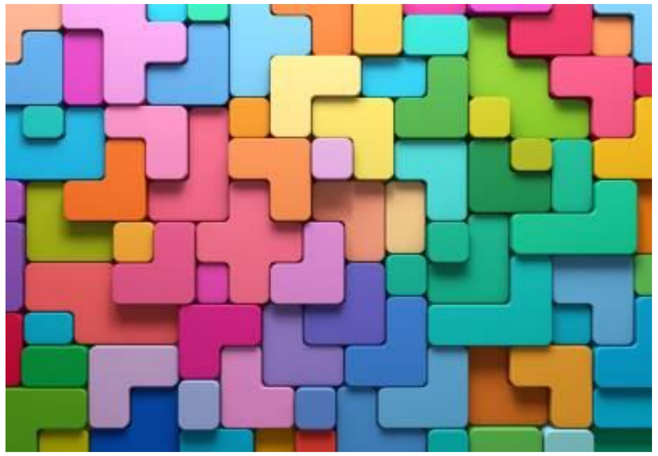
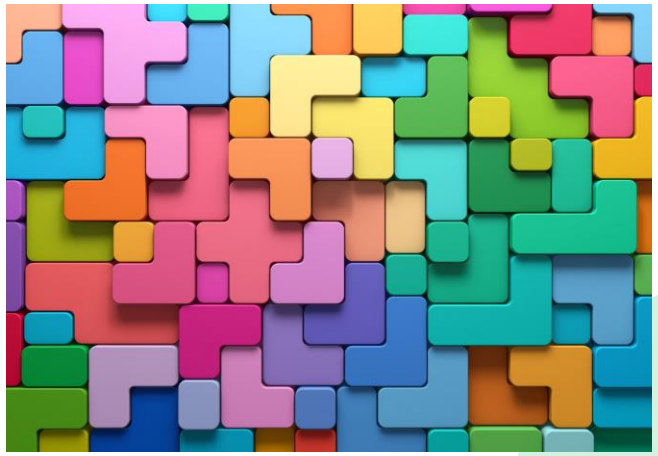
PARALLELL-PROGRAMMERING

FORELESNING 22

ONSDAG 13/11





(bilder generert av bing image creator)

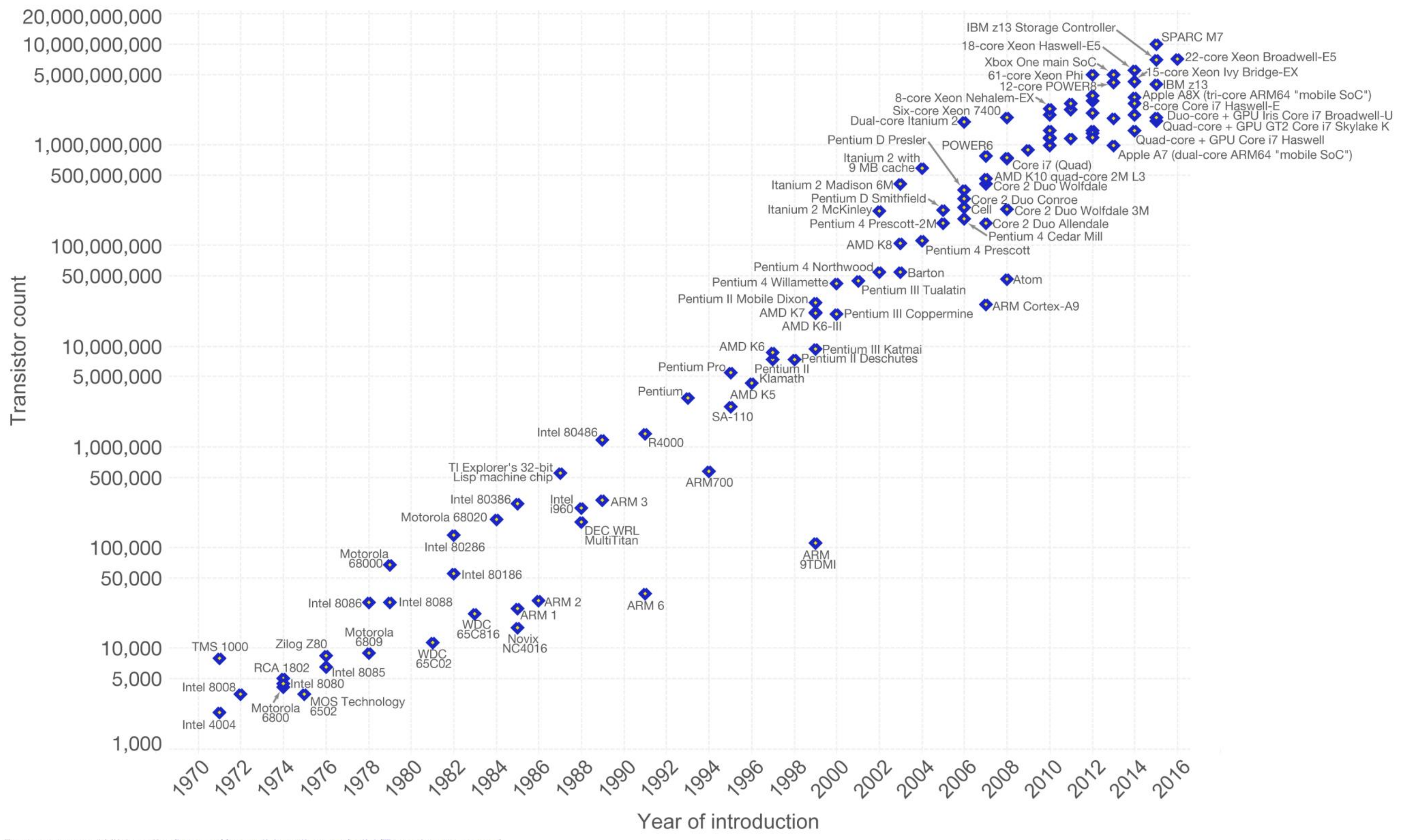
Læremål: Algoritmeanalyse og optimalisering

- Å kunne analysere hvor rask en algoritme er (viktig når vi får enorme mengder data)
- Profilering: finne flaskehalser (10% av koden bruker vanligvis 90% av tiden)
- Parallellprogrammering: la datamaskinen gjøre flere ting samtidig (uten å rote det til og gjøre ting i feil rekkefølge)

Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Our World in Data

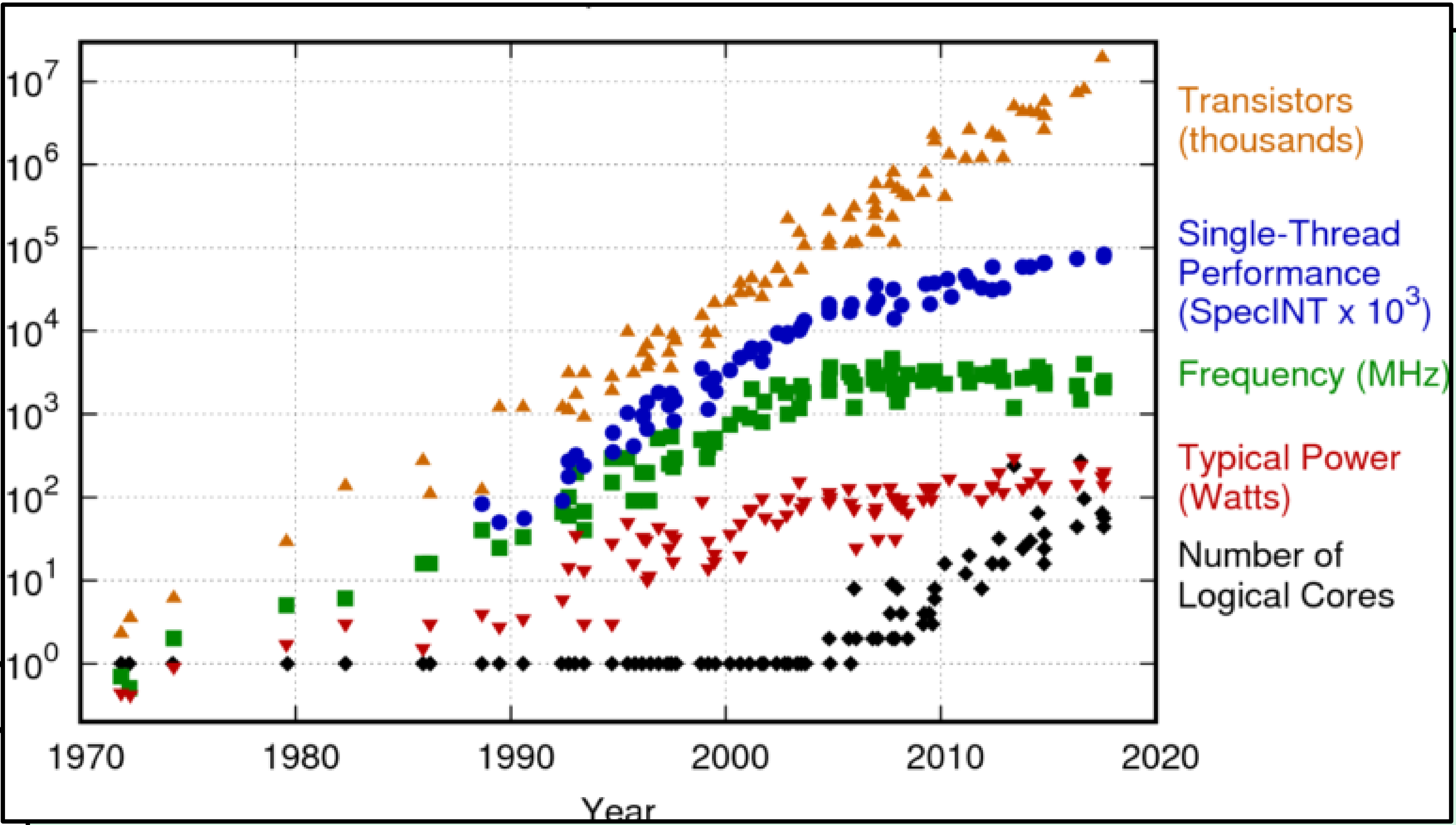
Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

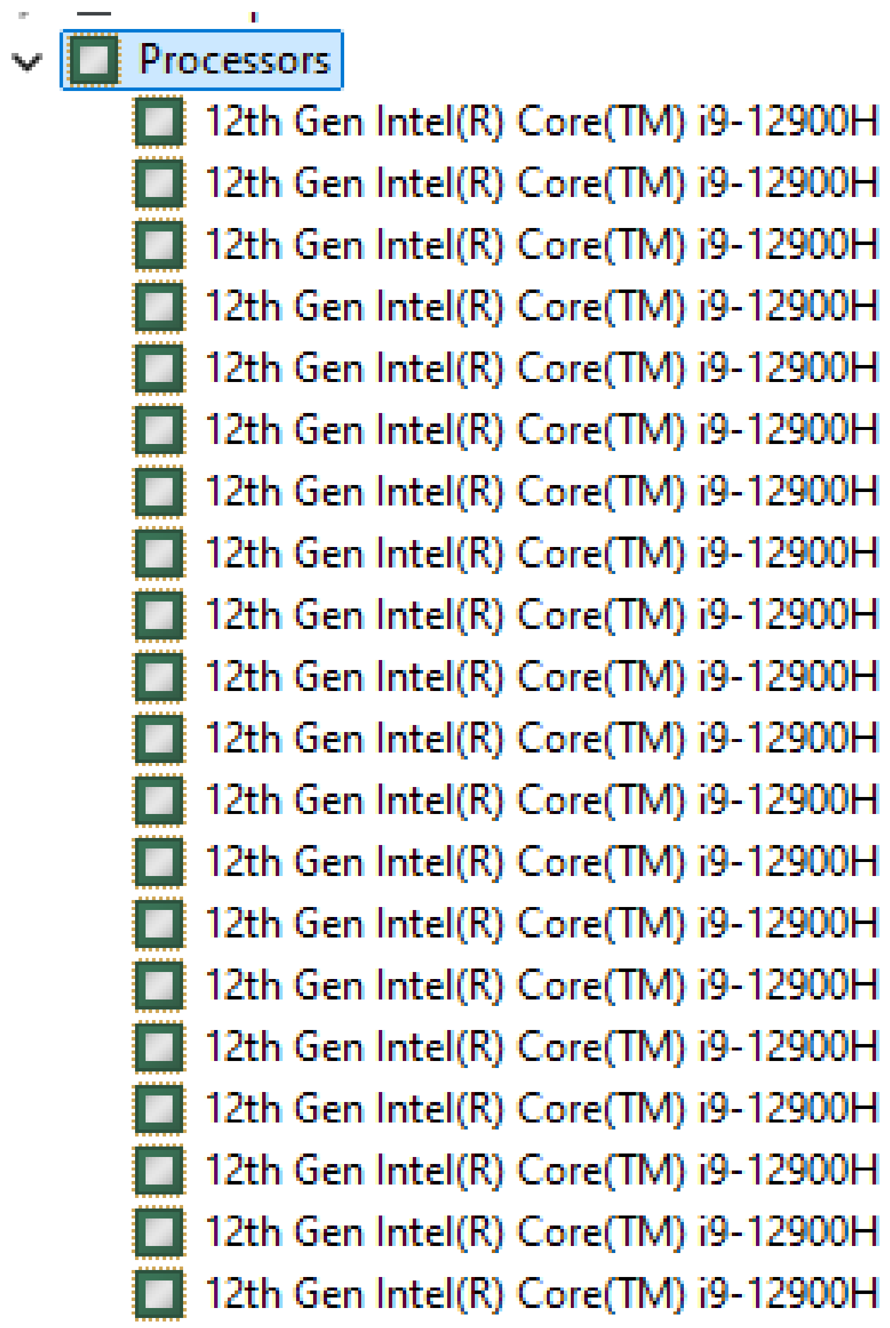
Alternativ til effektiv kode: Raskere datamaskin?

- Lenge gjorde man maskiner raskere ved å øke klokkefrekvensen til prosessorene deres (hvor fort de kan gjøre instruksjoner etter hverandre)
- Problem: Høyere frekvens → høyere strømforbruk (mer energi tapt som varme)
- Ca. 2004: Gradvis overgang *mange* prosessorer som samarbeider (flere kjerner) i en og samme maskin

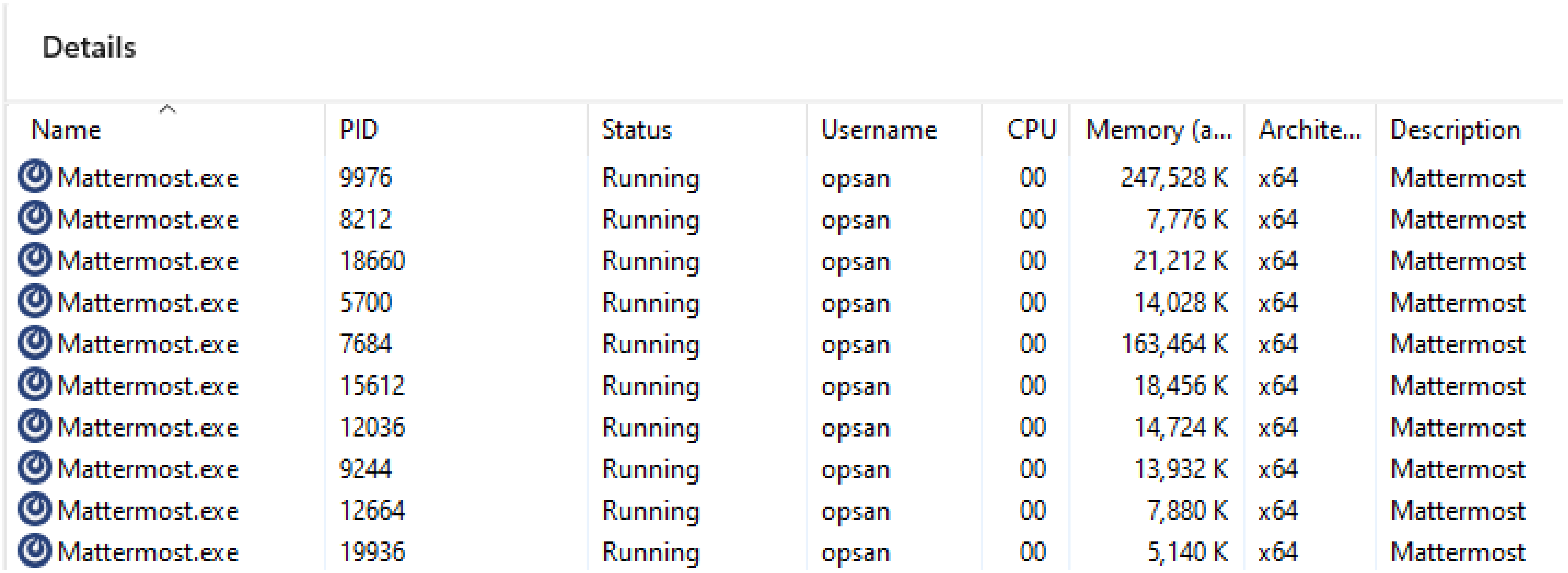


Transistors (thousands)
Single-Thread Performance (SpecINT x 10^3)
Frequency (MHz)
Typical Power (Watts)
Number of Logical Cores
Year
1970
1980
1990
2000
2010
2020
10^0
10^1
10^2
10^3
10^4
10^5
10^6
10^7

- Min maskin:


Processors
12th Gen Intel(R) Core(TM) i9-12900H
12th Gen Intel(R) Core(TM) i9-12900H
12th Gen Intel(R) Core(TM) i9-12900H
12th Gen Intel(R) Core(TM) i9-12900H
12th Gen Intel(R) Core(TM) i9-12900H
12th Gen Intel(R) Core(TM) i9-12900H
12th Gen Intel(R) Core(TM) i9-12900H
12th Gen Intel(R) Core(TM) i9-12900H
12th Gen Intel(R) Core(TM) i9-12900H
12th Gen Intel(R) Core(TM) i9-12900H
12th Gen Intel(R) Core(TM) i9-12900H
12th Gen Intel(R) Core(TM) i9-12900H
12th Gen Intel(R) Core(TM) i9-12900H
12th Gen Intel(R) Core(TM) i9-12900H
12th Gen Intel(R) Core(TM) i9-12900H
12th Gen Intel(R) Core(TM) i9-12900H
12th Gen Intel(R) Core(TM) i9-12900H
12th Gen Intel(R) Core(TM) i9-12900H
12th Gen Intel(R) Core(TM) i9-12900H
12th Gen Intel(R) Core(TM) i9-12900H

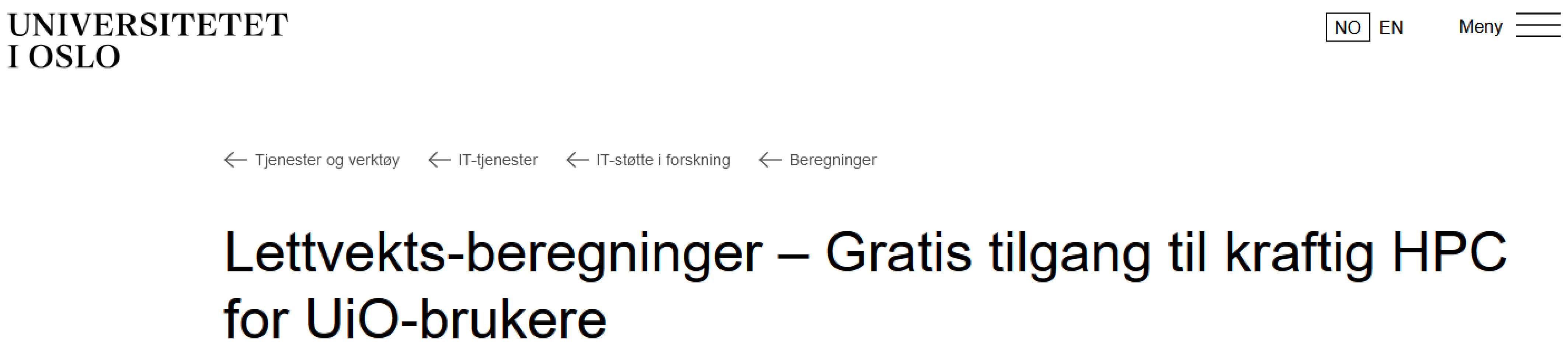
- Samme program (delt opp i *prosesser*) kan kjøre på flere kjerner samtidig



Details

Name	PID	Status	Username	CPU	Memory (a...	Archite...	Description
Mattermost.exe	9976	Running	opsan	00	247,528 K	x64	Mattermost
Mattermost.exe	8212	Running	opsan	00	7,776 K	x64	Mattermost
Mattermost.exe	18660	Running	opsan	00	21,212 K	x64	Mattermost
Mattermost.exe	5700	Running	opsan	00	14,028 K	x64	Mattermost
Mattermost.exe	7684	Running	opsan	00	163,464 K	x64	Mattermost
Mattermost.exe	15612	Running	opsan	00	18,456 K	x64	Mattermost
Mattermost.exe	12036	Running	opsan	00	14,724 K	x64	Mattermost
Mattermost.exe	9244	Running	opsan	00	13,932 K	x64	Mattermost
Mattermost.exe	12664	Running	opsan	00	7,880 K	x64	Mattermost
Mattermost.exe	19936	Running	opsan	00	5,140 K	x64	Mattermost

https://www.uio.no/tjenester/it/forskning/beregning/lettvekt.html

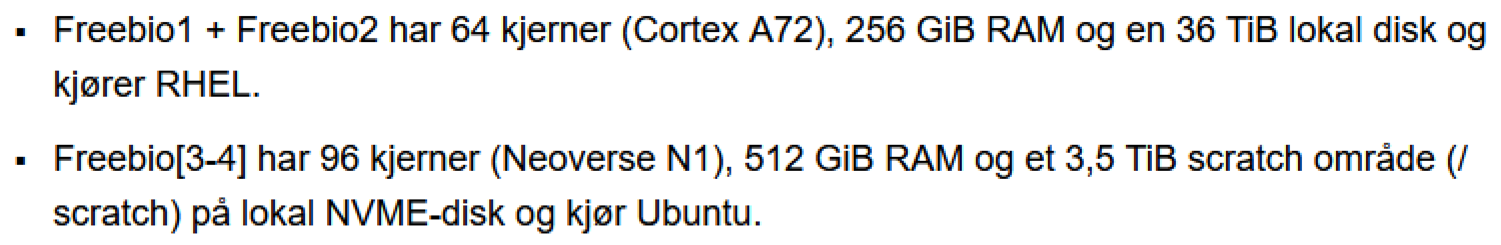

UNIVERSITETET
I OSLO

NO EN Meny

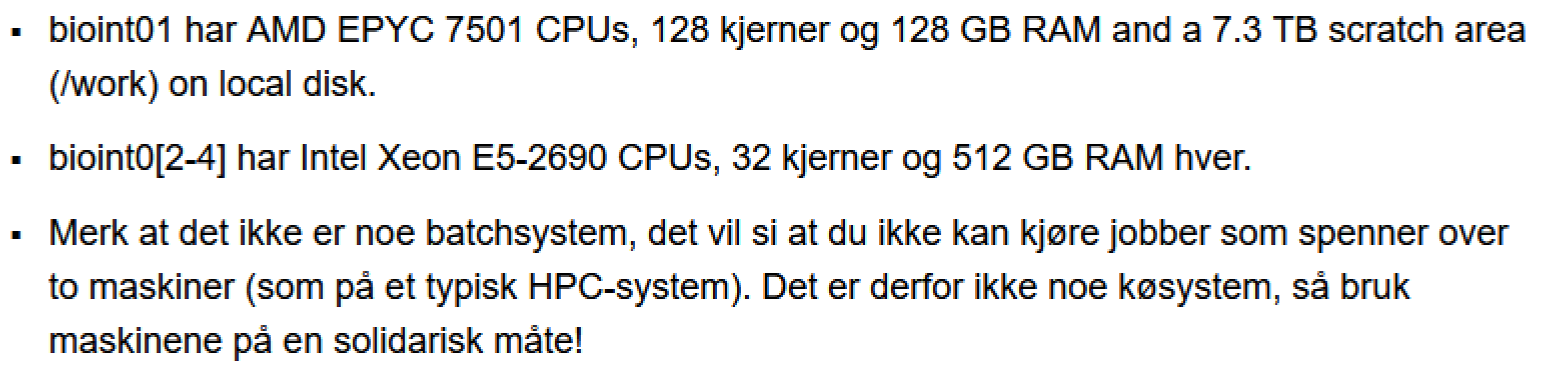
← Tjenester og verktøy ← IT-tjenester ← IT-støtte i forskning ← Beregninger

Lettvekts-beregninger – Gratis tilgang til kraftig HPC for UiO-brukere

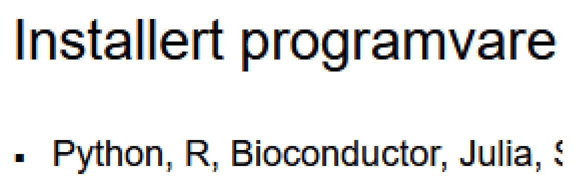
Tilgang: https://nettskjema.no/a/210629



- Freebio1 + Freebio2 har 64 kjerner (Cortex A72), 256 GiB RAM og en 36 TiB lokal disk og kjører RHEL.
- Freebio[3-4] har 96 kjerner (Neoverse N1), 512 GiB RAM og et 3,5 TiB scratch område (/scratch) på lokal NVME-disk og kjør Ubuntu.



- bioint01 har AMD EPYC 7501 CPUs, 128 kjerner og 128 GB RAM and a 7.3 TB scratch area (/work) on local disk.
- bioint0[2-4] har Intel Xeon E5-2690 CPUs, 32 kjerner og 512 GB RAM hver.
- Merk at det ikke er noe batchsystem, det vil si at du ikke kan kjøre jobber som spenner over to maskiner (som på et typisk HPC-system). Det er derfor ikke noe køsystem, så bruk maskinene på en solidarisk måte!



Installert programvare

- Python, R, Bioconductor, Julia,

Alt vi har gjort til nå har vært *sekvensielt*

- Vi har skrevet programmer som kjører på én kjerne i datamaskinen (de andre brukes ikke)
- I dag ser vi på *parallellisering* – å fordele programmets oppgaver mellom flere kjerner som jobber parallelt
- Krever litt ekstra kjøretid for maskinen å fordele og holde styr på de fordelte oppgavene, men lønner seg likevel i mange tilfeller

In 2016 the world champion in chess, Magnus Carlsen, played against 70 opponents simultaneously in Hamburg. We can imagine that we have been told to write a computer program that will instruct Magnus on how he should play in order to finish within the least amount of time. Suppose the following

- Magnus uses 10 seconds to make a move
- His opponents use 50 seconds to make a move
- In an average game, there are 30 moves

Fig. 66 Source: chessbase.com

The sequential program

The naive approach would be to write a sequential program, where Magnus plays against one opponent at a time, and when the game is over he plays against the next opponent. In this case, Magnus would use

$(10 + 50) \text{ seconds} \times 30 \text{ moves} \times 70 \text{ opponents} = 126000 \text{ seconds} = 35 \text{ hours}.$

Created copies of Magnus

Now say that we want to utilize all the processors we have available to solve this problem. Imagine that Magnus is a 7-core computer. This would be analogous to making 7 copies of Magnus Carlsen, each running the sequential program. Now we can distribute 10 opponents to each copy and therefore solve the problem in $\frac{35}{7} = 5$ hours.

Don’t wait

Of course, Magnus Carlsen did not spend 35 hours on these games. The way Magnus Carlsen would play is as follows: He goes to the first opponent, uses 10 seconds to make his move, and goes directly to the next opponent. After making a move against all the 70 opponents there is $10 \times 70 = 700$ seconds $= 11.67$ minutes since he started at the first opponents. The first opponent only used 50 seconds to make his/her move so Magnus doesn’t need to wait and can therefore continue in the same manner. In total he would now spend $\mathbf{10\ seconds \times 70\ opponents \times 30\ moves = 21000\ seconds = 5\ hours\ and\ 50\ minutes.}$

The multi-core and multi-threaded approach

The multi-core approach is analogous to making copies of Magnus, while the multi-thread approach is analogous to Magnus not waiting for his opponent to make a move. By combining these approaches we could even go faster. This would be analogous to each copy of Magnus would not wait for their opponents. With this approach, Magnus can finish all 70 games in only 50 minutes.

I/O Bounded vs CPU bounded problems

Depending on the type of problem, one should use different techniques to speed up the program. In the example of Magnus Carlsen playing against multiple opponents, there are two ways we can reduce the total time

1. Use the time that Magnus has to wait for the opponent to move to do other work
2. Create copies of Magnus that do the same work

Multithreading - Don’t wait

The first approach would be analogous to running the program across 70 threads. At any time, there is only one process (i.e. Magnus Carlsen), but as soon as he has to wait for an opponent, another thread takes over. These types of problems are I/O-bounded problems. I/O bounded problems are problems where you spend a lot of time waiting for input/output (I/O) from some external source that is slower than the CPU. An example of this is when downloading content from the web.

Note that if the opponents playing against Magnus would make their move instantaneously, we would not gain any speedup by running Magnus along multiple threads.

Technically speaking, a multi-threaded program is not a parallel program, since it is not running multiple things at the same time. Instead, the operating system determines which task are performed at a given time, so that the program spend the least amount of time waiting. Notice that it is the operating system that decides which thread is being run.

Multiprocessing - Making copies

The other approach would be analogous to running the program on multiple cores. Here we make copies of Magnus, with each copy representing one processor executing the work independently of the other processors.

The example with Magnus Carlsen playing chess is an example where the tasks (i.e. the chess games) are completely independent. However, in scientific computing, one would often like to exploit multiprocessing also where there is some dependence across the cores. For example, when solving equations on a large domain, one approach is to split the domain into smaller subdomains and then solve the equation on each subdomain on a separate core.

Noen begreper

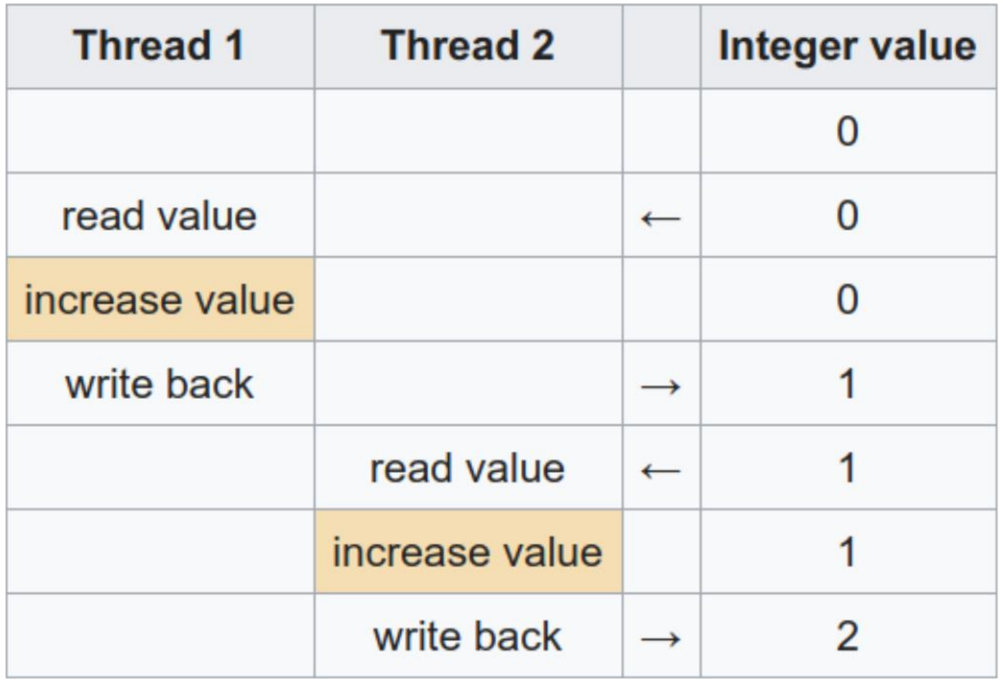
- **Tråder** er styrt av operativsystemet: Lar en kjerne bytte mellom oppgaver så fort den må vente på et eller annet ("utålmodig prosessor") – hver tråd er da en oppgave
- **Parallellisering** er når en (stor) oppgave fordeles på flere prosessorer som må koordineres (samarbeide) for å løse den
- Disse kan kombineres (og vi skal se på begge deler)

LIVEKODING: PARALLEL-LISERING

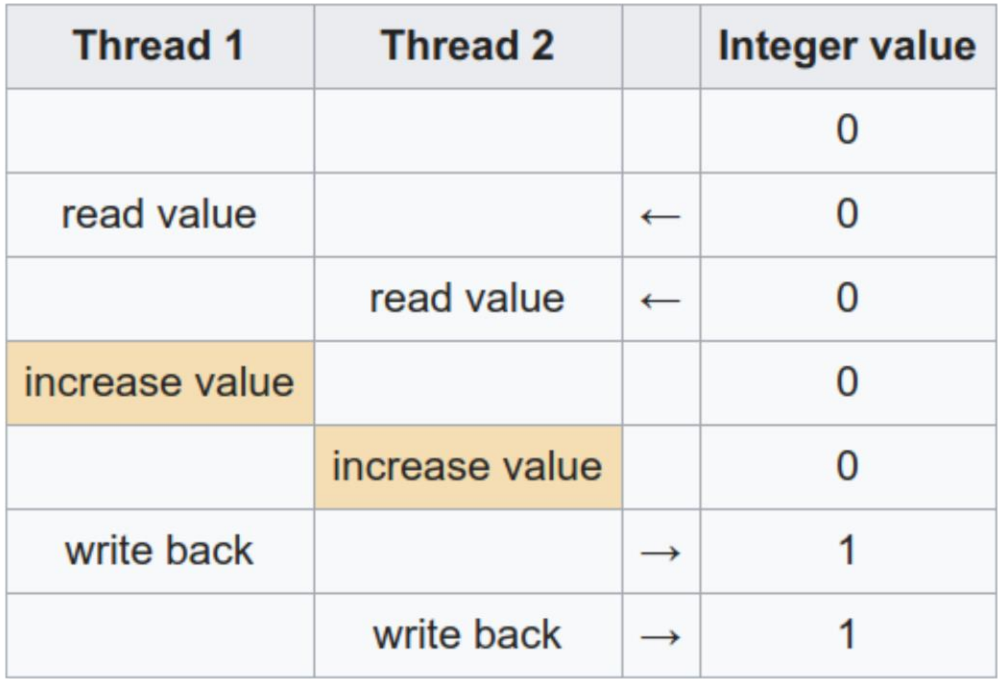
Ulember med parallellisering

- Kan være vanskeligere å lese koden
- Ekstra arbeid med å holde styr på prosessene som kjører parallelt (flere ikke alltid bedre)
- Hvis 100 000 personer skal male et hus, må alle få hver sin lille oppgave, hvert sitt utstyr, og vi må passe på at de ikke kolliderer med hverandre – dette kan da ta lenger tid enn å la en person male hele huset

“Race conditions”: Hvilken tråd kommer først?



Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

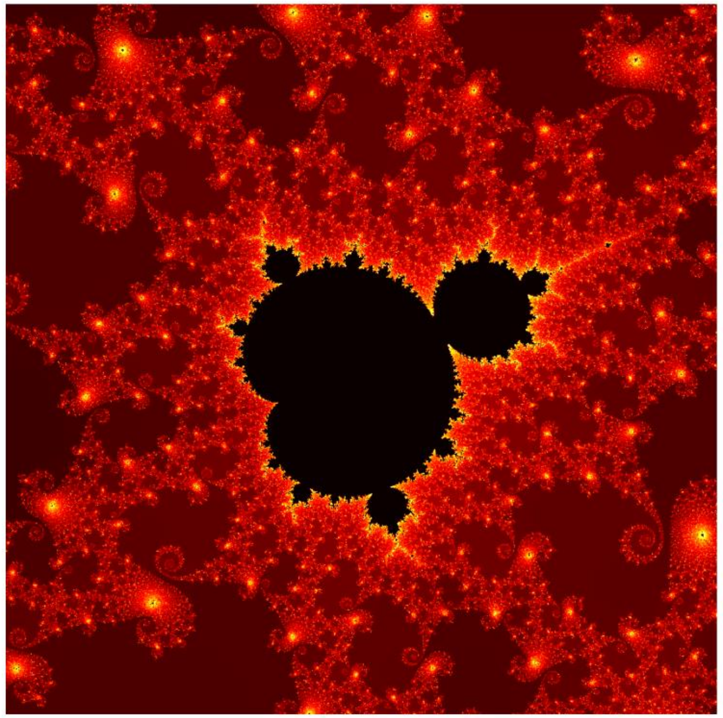


Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Alternativ til prosessorkjerner: Kjerner i skjermkort

- Prosessoren (CPU) er en altmuligmann
- Skjermkortet (GPU) er en spesialist på utregninger (som trengs for å vise ting på skjermen)
- GPUen har også flere kjerner – hver med ansvar for sine piksler på skjermen (bygd mtp. parallellisering)
- Men dette er lavnivå-programmering mot en bestemt type maskinvare, med de utfordringer det medfører

Eksempel: Mandelbrot-settet



- Standard Python: 206 s
- numpy: 34.5 s
- numba: 7.34 s
- C++: 5.2 s
- C++ (OMP): 264 ms
- numba (OMP): 230 ms
- GPU (med numba): < 20 ms

Etter forelesningen

- Ingen forelesning på fredag – vi er i mål!
- Onsdag 20/11: Oppsummering av hele semesteret mtp. muntlig eksamen
- Fredag 22/11: Spørretime og eksamensinfo