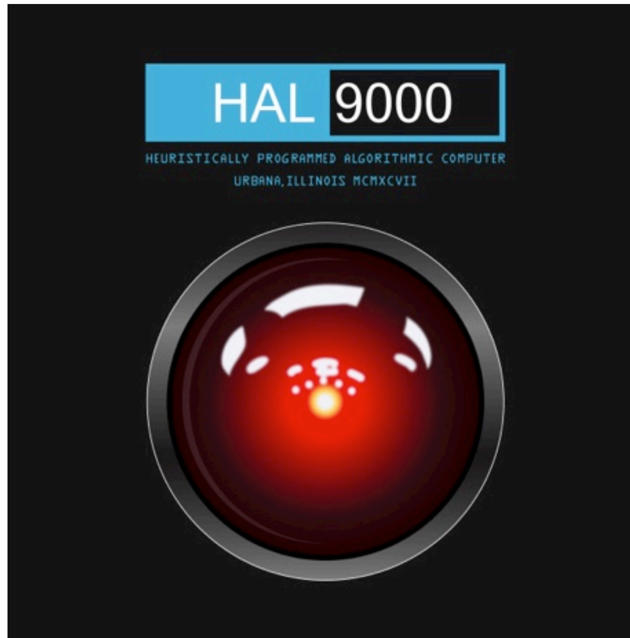
# Beräkningar och Prestanda

**Dr. Johan Hagelbäck**

johan.hagelback@lnu.se

http://aiguy.org




The 9000 series is the most reliable computer ever made. We are all, by any practical definition of the words, foolproof and incapable of error.

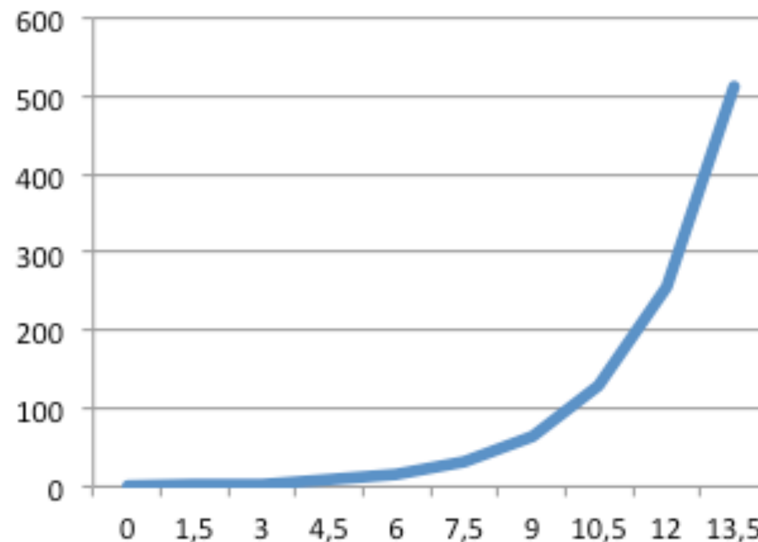
-HAL 9000

# HAL 9000

- HAL 9000 är datorn i filmen 2001: A Space Odyssey från 1968, regisserad av Stanley Kubrick efter en roman av Arthur C. Clarke
- HAL 9000 anses vara helt idiotsäker och kan inte göra något fel, och den har i princip inga begränsningar
- Den är intelligent och kan kommunicera i tal och text
- En bit in i historien drabbas dock HAL av problem på grund av tvetydiga instruktioner om uppdraget, och börjar döda rymdskeppets besättning

# Prestanda

- Våra datorer har begränsningar, dom är inte som HAL 9000
- Hårdvaran har dock förbättrats i en otrolig hastighet enligt Moores Lag – hastigheten dubbleras efter 18 månader



# Prestanda

- Hur kan vi mäta prestanda?
- Minne (RAM och disk) är enkelt då de har en bestämd storlek
- Generellt kan man säga att applikationer blir snabbare om det finns ”gott om” minne
- Beräkningshastighet är svårare att mäta
- Olika processorer har olika klockfrekvens (mätt i GHz), men eftersom det finns olika typer av processorer går det inte att jämföra klockfrekvens

# Prestanda

- Superdatorer brukar i stället mätas i MIPS (miljoner instruktioner per sekund) eller MFLOPS (miljoner flyttalsberäkningar per sekund)
- För att ytterligare komplicera det hela har vi oftast både en CPU och en GPU, och GPUn kan markant förbättra prestandan för vissa typer av applikationer
- Vi har också oftast mer än en kärna, och vissa applikationer är lättare att parallellisera än andra

# Benchmark

- Oftast är inte själva siffrorna på hårdvaran som är det mest intressanta
- Vi är mer intresserade av hur bra datorn är till det vi vill använda den till, t.ex. 3D spel
- Det bästa sättet att mäta prestanda är benchmarks – mäta hur snabbt datorn utför en specifik uppgift:
  - Framerate (FPS) i ett spel
  - Tid för att sortera en lista med 100000 tal
  - Svarstid för en nätverksapplikation
  - …
- Empiriskt experiment

# Benchmark

- Det stora problemet med benchmarks är att de är svåra att generalisera
- En benchmark mäter prestanda i en specifik miljö och under specifika omständigheter
- Beroende på vad operativsystemet har för sig i bakgrunden kan prestanda skilja sig åt mellan olika körningar, på samma dator
- För att få ett säkrare värde mäter man flera gånger och beräknar ett medelvärde

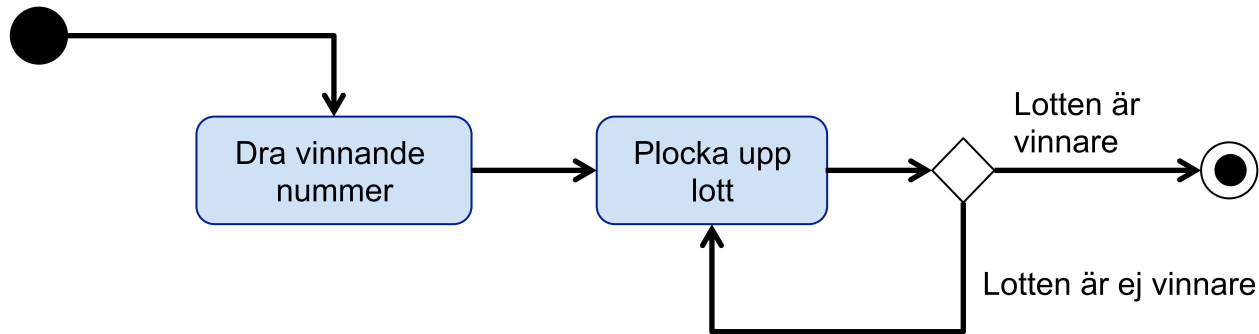
# Datamängd

- En sak benchmarks har svårt för är hur prestanda varierar med storleken på indata:
  - Hur varierar svarstiden om vår webbserver hanterar 10 eller 1000 förfrågningar i sekunden?
  - Hur varierar körtiden om vi sortera 100 eller 100000 siffror?
  - …
- För att ta reda på detta måste vi köra flera benchmarks med olika storlekar på indata
- Ett annat verktyg vi kan använda är att *analytiskt* beräkna hur prestandan på en algoritm varierar med storleken på indata

# Lotteri

- I ett lotteri dras ett vinnande nummer
- Alla sålda lotter ligger i en stapel
- Vi plockar upp en lott i taget och jämför med vinnande numret, tills vi hittat vinnaren

# Lotteri




Vilken del av algoritmen upprepar sig?

# Lotteri

- För varje lott i högen:
  - Plocka upp lotten
  - Jämför med vinnande numret
- Körtiden för algoritmen är alltså beroende av antalet lotter i högen, kallat N
- Hur många lotter måste vi plocka upp och jämföra med vinnande numret?
  - Bästa fallet: 1
  - Värsta fallet: N
  - Medel: N/2

# Lotteri

- Vi säger att algoritmen är av typen O(N)
  - O betyder *ordo*
  - O(N/2) är egentligen mer korrekt, men vanligtvis tas konstanter bort i ordo-uttryck
- Det är en *linjär* algoritm – körtiden är direkt proportionell mot storleken på indata

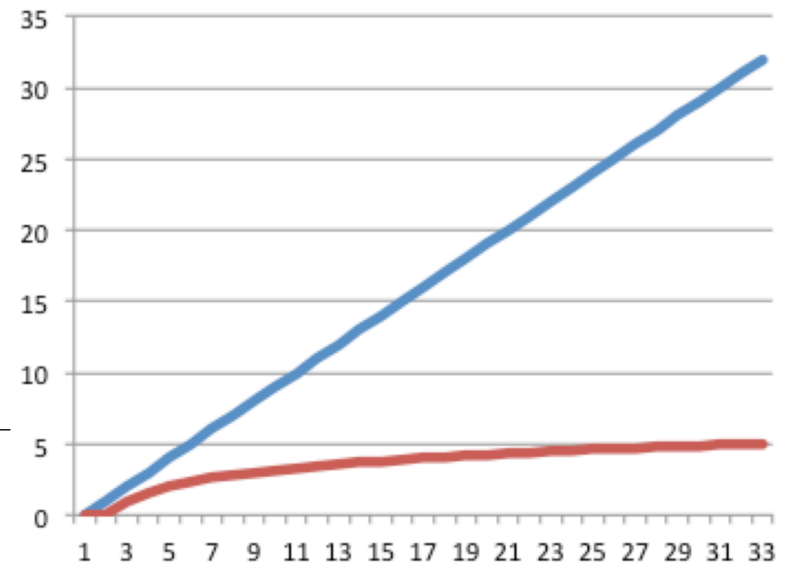
- Lotterialgoritmen är en vanlig *linjärsökning*
- Om lotterihögen är sorterad, kan vi förbättra algoritmen?

# Linjär vs. binär sökning

- Om högen är sorterad kan vi använda *binär* sökning:
  - Börja på mitten
  - Kontrollera om vinnande numret är högre eller lägre
  - Fortsätt på ¼ eller ¾
  - …
- Blir prestandan bättre?

# Linjär vs. binär sökning

| Antal lotter | Linjärsökning | Binärsökning |
|---|---|---|
| 0 | 0 | 0 |
| 4 | 4 | 2 |
| 8 | 8 | 3 |
| 32 | 32 | 5 |
| N | N | $\log_2 N$ |

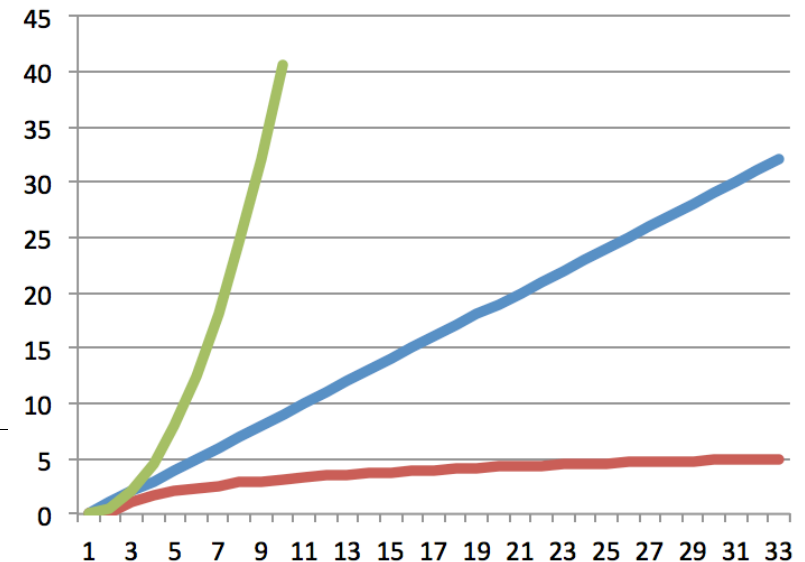


# Sortering

- Vi vill sortera högen med lotter och väljer urvalssortering:
  - För varje lott i högen:
    - Leta upp lotten med lägst nummer i resten av högen
    - Byt plats på nuvarande lott och minsta lotten
- Algoritmen kräver två nästlade loopar
- En yttre som går igenom samtliga lotter (N) och en inre som går igenom N/2 lotter
- Algoritmen är av typen $O(N^2)$
- Den är av typen *polynoma* algoritmer

# Sortering

| Antal lotter | Linjärsökning | Binärsökning | Sortering |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 4 | 4 | 2 | 8 |
| 8 | 8 | 3 | 32 |
| 32 | 32 | 5 | 512 |
| N | N | $\log_2 N$ | $N^2/2$ |



# Algoritmtyper

- Algoritmer av typen $O(\log_2 N)$ och $O(N)$ är snabba, såvida vi inte har extremt mycket indata
- Prestandan för polynoma algoritmer växer betydligt snabbare med storleken på indatan
- Moderna dator klarar dock dessa algoritmer även på relativt stora datamängder
- Det finns dock algoritmer med ännu värre prestanda…

# Kombinationslås

- Koden är tre siffror mellan 0 och 39
- Först vrids hjulet medsols till första siffran, sedan motsols till andra siffran, och medsols igen till tredje siffran
- Vet vi inte rätt kod måste vi söka efter den
- En algoritm måste alltså gå igenom alla möjliga kombinationer:
  - 40 möjliga siffror (0-39)
  - Kombination av 3 siffror
  - Antal möjliga koder blir därmed $40*40*40 = 40^3 = 64000$
- Antag att det tar 5 sek att testa en kombination. Hur lång tid tar det då att testa alla?

# Kombinationslås

| Antal möjliga siffror | Kombinationer | Tid (5 sek/ kombination) |
| --- | --- | --- |
| 40 | 64000 | 88,89 h |
| 41 | 68921 | 95,72 h |
| 42 | 74088 | 102,9 h |
| 43 | 79,507 | 110,42 h |

# Kombinationslås

- För att göra låset säkrare kan vi:
  - Inkludera fler möjliga siffror
  - Ha koder med mer än 3 siffror
- Det första alternativet är en polynom algoritm:
  - $40*40*40 = 40^3 = N^3$
- Det andra alternativet då?
- Då ändras i stället exponenten:
  - $40*40*40*40 = 40^4 = 40^N$

# Exponentiell algoritm

- Exponentiella algoritmer är riktigt elaka:

| Antal rotationer | Kombinationer | Tid (5 sek/ kombination) |
|---|---|---|
| 3 | 64000 | 88,89 h |
| 4 | 2560000 | 296,26 dagar |
| 5 | 102400000 | 16,22 år |

- Dessa ska vi alla sätt vi kan försöka undvika

# Exponentiell algoritm

| Datamängd (N) | $N^3$ | $3^N$ |
|---|---|---|
| 1 | 1 sek | 1 sek |
| 5 | 2,08 min | 4,05 min |
| 10 | 16,67 min | 16,40 h |
| 15 | 56,25 min | 166 dagar |
| 20 | 2,22 h | 110 år |
| 25 | 4,34 h | 26849 år |

# Exponentiell algoritm

- Typiska exponentiella algoritmer är så kallade *brute-force* algoritmer
- De söker igenom alla möjliga kombinationer av indata
- Typiska användningsområden är att knäcka lösenord eller koder
- Eftersom algoritmernas tidsåtgång ökar exponentiellt kan vi välja ”säkra” lösenord eller koder

| Antal siffror | Kombinationer |
|---|---|
| 2 | 100 |
| 3 | 1000 |
| 4 | 10000 |
| 5 | 100000 |
| 6 | 1000000 |

# Praktiska algoritmer

- Oftast kan ett problem lösas med flera olika algoritmer
- Det är då viktigt att vi utvärderar alla alternativ, då någon algoritm kan vara betydligt snabbare än andra
- Om vi till exempel vet att ett lösenord består av ett engelskt ord kan vi knäcka det på några sekunder
- Vi kan som exempel jämföra körtiden för olika sorteringsalgoritmer:

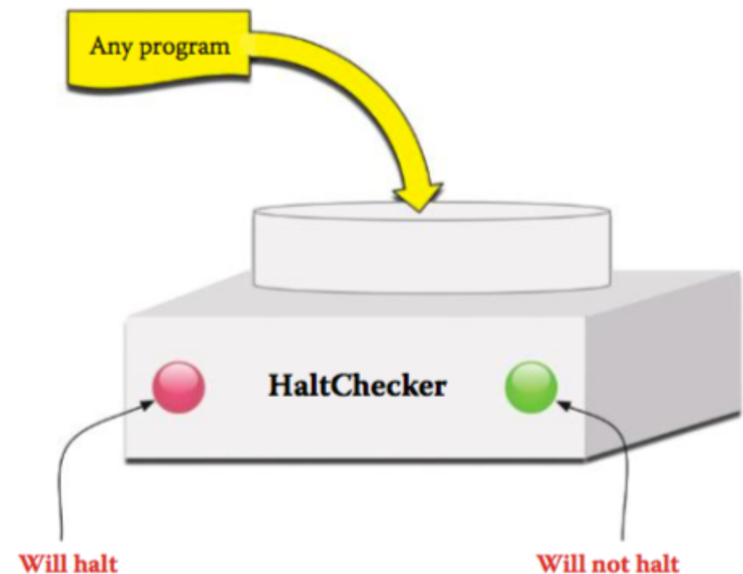
# Sorteringsalgoritmer

| Körning | Bubble | Quick | Selection | Insertion | Merge |
|---|---|---|---|---|---|
| 1 | 17384 | 24 | 3258 | 3 | 30 |
| 2 | 17559 | 21 | 3386 | 3 | 27 |
| 3 | 17795 | 19 | 3344 | 4 | 28 |
| 4 | 17484 | 20 | 3417 | 3 | 28 |
| 5 | 17642 | 19 | 3358 | 3 | 30 |
| Medel | 17572,8 | 20,6 | 3352,6 | 3,2 | 28,6 |

100000 slumptal mellan 0 och 10000

# Omöjliga algoritmer

- 1936 beskrev Alan Turing ett teoretiskt program kallat *Halt Checker*
- Halt Checker är ett program som tar vilket annat program som helst som indata
- Halt Checker kontrollerar om programmet det får som indata kommer att stanna (halt) eller fortsätta köra



# Omöjliga algoritmer

- Ett annat program kallat *InterestingProgram* använder *HaltChecker*:

```
10    haltchecker = new HaltChecker(InterestingProgram)
20    if (haltchecker.check() == true)
30      while (true)
40        //Do something
50    else
60      halt()
```

- Om HC säger att IP ska stanna, stannar det inte
- Om HC säger att IP inte ska stanna, då stannar det
- Algoritmen är ”omöjlig” för IP

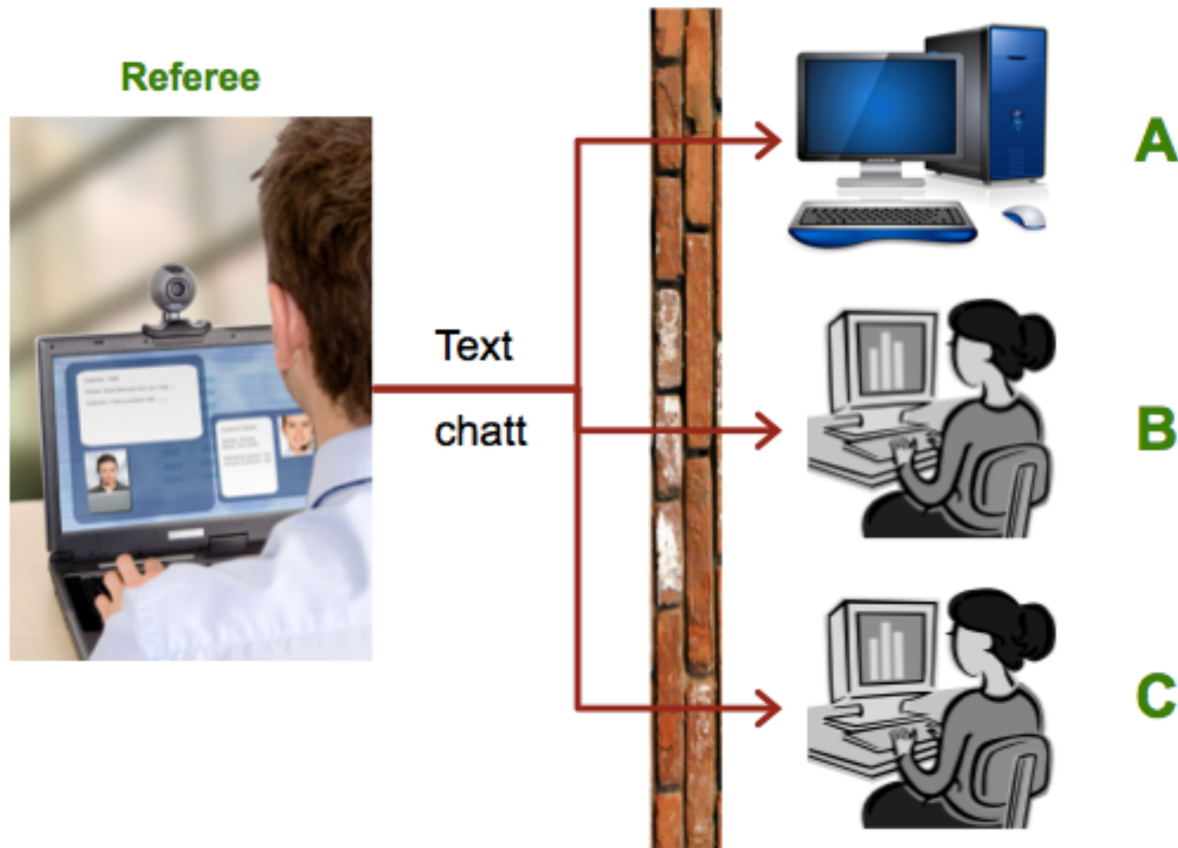
# Omöjliga algoritmer

- Att avgöra om ett program stannar eller inte känns inte som en omöjlig uppgift
- Det fungerar för de flesta program
- Men antagandet att det ska fungera för ”alla program” stämmer inte, då InterestingProgram lurar det

# Turing testet

- 1950 publicerade Alan Turing artikeln *Computing Machinery and Intelligence* i tidsskriften *Mind*
- I artikeln resonerade Turing kring frågan Kan maskiner tänka?
- I artikeln beskrevs Turing testet som historiskt sätt varit, och är, en milstolpe i artificiell intelligens

# Turing testet



# Turing testet

**Är det en dator eller människa jag kommunicerar med?**

# Turing testet

- Har vi klarat Turing testet ännu?
- Det har gjorts många försök, men än anses inte Turing testet vara slaget
- Ett av de bästa försöken är CleverBot
- I ett försök 2011 blev CleverBot 59,3% mänsklig jämfört med 63,3% för riktiga människor
- Ett av de första försöken var chattboten ELIZA från 1966

# ELIZA

| | |
|---|---|
| ELIZA: | Please tell me what's been bothering you. |
| YOU: | I am scared of spiders. |
| ELIZA: | Is it because you are scared of spiders that you came to me? |
| YOU: | Yes. I need help. |
| ELIZA: | I see. |
| YOU: | Can you help me? |
| ELIZA: | Really, I help you? |
| YOU: | You are the psychiatrist. |
| ELIZA: | What makes you think I am the psychiatrist? |
| YOU: | It says so on your office door. |
| ELIZA: | Why are you concerned over my office door? |
| YOU: | Because it says you are a psychiatrist and now you are telling me that you are not. |
| ELIZA: | Does it please you to believe I am not? |
| YOU: | No. Now I wonder if you can help me or not? |
| ELIZA: | You are being a bit negative. |

# ELIZA

- ELIZA försöker hitta nyckelord i frågorna och konstruerar ett svar utifrån dem
- Om inget nyckelord hittas, svarar ELIZA med något generellt svar eller upprepar en tidigare fråga
- ELIZA har ingen förståelse för vad en mening betyder
- Ända trodde man seriöst på slutat av 60-talet att system som ELIZA kunde användas för terapeutisk behandling…

- IBM Watson  är ett QA (question-answering) system som kan tolka innebörden i en fråga och sen leta upp svaret bland en stor mängd lagrade artiklar, böcker, tidskrifter, wikis, …
- Watson blev känd när den ställde upp i en stormästarmatch i Jeopardy mot de två bästa tävlanden någonsin i USA

IBM Watson

# Parallellism och samtidighet

# Samtidighet

- Hittills har vi pratat om *sekventiell* exekvering – instruktioner körs en efter en i sekvens
- Men precis som vi kan gå och prata samtidigt kan datorer multitaska
- När mjukvara gör flera saker samtidigt kallar vi det *samtidighet* (*concurrency*)
- Vi är intuitivt bättre på att förstå sekventiell exekvering så samtidighet kan vara svårt att greppa
- Samtidighet leder också till ett antal problem som kan vara svåra att felsöka och korrigera

# Varför samtidighet?

- Hårdvaran blir snabbare och snabbare men det finns tekniska och fysiska begränsningar på hur snabbt data kan skickas mellan t.ex. processor och minne
- Tätare kretsar ger snabbare dataöverföring och processorer, men är kostsamt att utveckla och tillverka
- Det finns en gräns, som visserligen flyttas fram hela tiden, där ännu mindre kretsar inte är kostnadseffektiva
- En lösning är flera processorer och/eller processorkärnor
- För att kunna utnyttja detta krävs att programvaran kan exekvera olika delar parallellt

# Distribuerade beräkningar

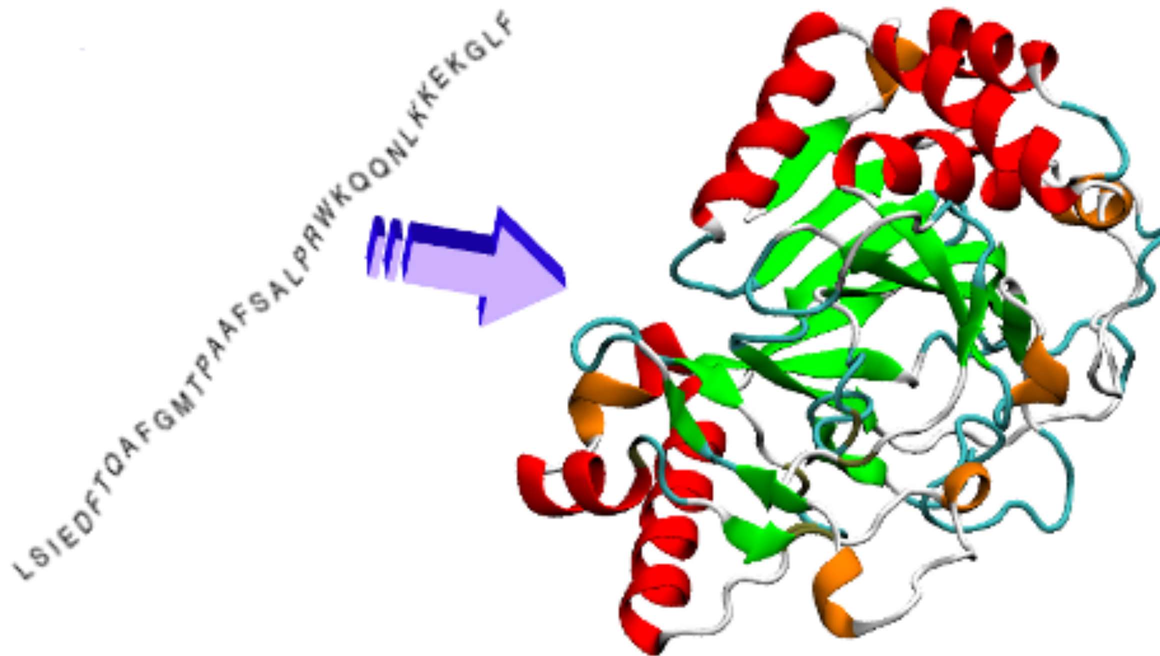
- En annan form av parallellism är distribuerade beräkningar (*distributed computing*)
- Dessa görs i ett nätverk, t.ex. Internet, där varje fysisk dator/server ansvarar för en oberoende del av en större uppgift
- Internet i sig självt kan ses som ett distribuerat system eftersom många enheter tillsammans hjälps åt med kommunikationen
- Vi använder idag oftast termen *cloud computing* för distribuerade system över Internet

# Exempel: FOLDING @HOME

- Folding@home är ett exempel på en distribuerad applikation
- Projektet startade år 2000 av Stanford University med syftet att vem som helst som har en dator kopplad till Internet kan ”låna ut” beräkningsresurser till forskning
- När du inte aktivt använder din dator till något krävande hämtar applikationen data från en server, utför beräkning och laddar upp resultatet

# Protein folding

- Protein folding innebär att utifrån en sekvens av aminosyror (som utgör ett protein) avgöra hur proteinet ser ut i 3D:



# Exempel: FOLDING @HOME

- Att veta 3D strukturen på ett protein hjälper till att förstå olika sjukdomar och hur mediciner/vaccin kan utvecklas för att bekämpa dem
- Exempel på sjukdomar som tros bero på proteiner med felaktig 3D struktur är Alzheimers, Parkinsons, Huntingtons, cystisk fibros, olika typer av cancer, med mera…
- 2009 kom Folding@home med i Guiness rekordbok för världens kraftfullaste nätverk för distribuerade beräkningar
- Projektet har idag strax under en miljon användare

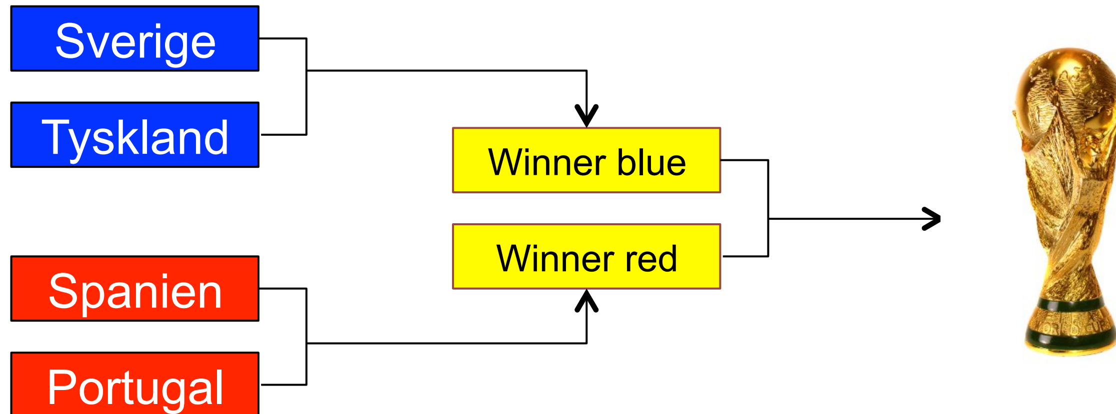
# Parallellism

- Distribuerade system, cloud computing, multicore/multiprocessor system och superdatorer använder sig av parallella beräkningar
- En annan form av samtidighet kan köras på en processor/processorkärna
- Då turas olika processer om att använda systemets resurser, så för användaren ser det ut som att saker körs parallellt
- Samtidighet brukar beskriva alla former av samtidiga beräkningar, medan parallellism används för hårdvarubaserade samtidiga beräkningar

# Schemaläggning

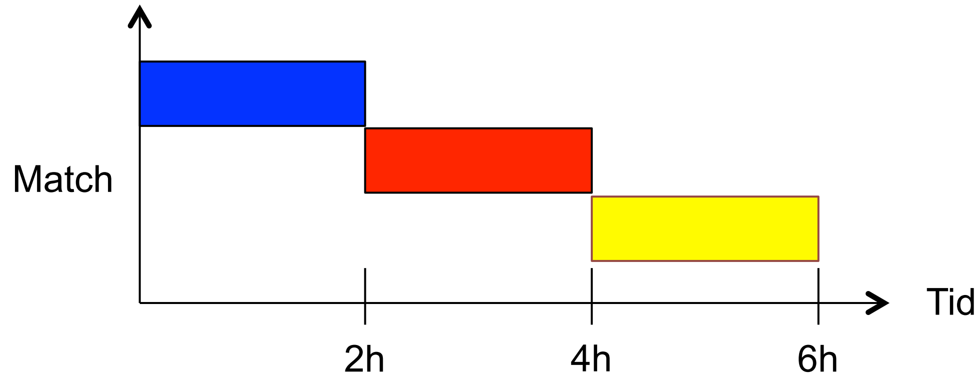
- Ett problem med samtidighet är att vissa deluppgifter är beroende av resultatet från andra beräkningar
- Samtliga delar av problemet kan då inte köras parallellt
- Ett intuitivt exempel är en fotbollsturnering:

# Schemaläggning

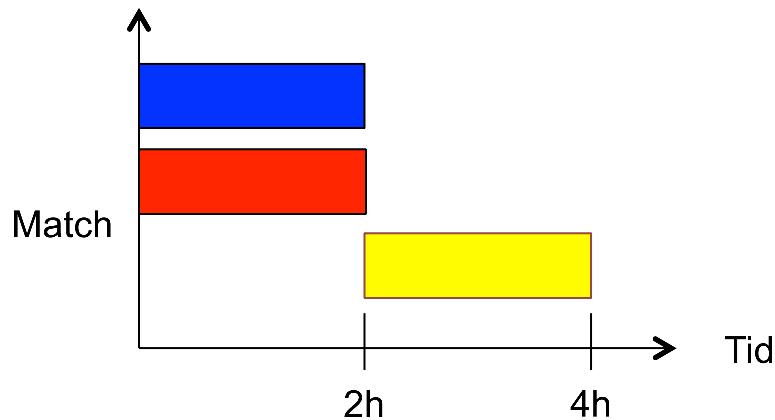



- Totalt måste tre matcher spelas
- Gula matchen är dock beroende av resultaten från blå och röd match, så endast dessa båda kan spelas samtidigt
- Vi kan dock få ner tiden till 4h (antag varje match tar 2h) i stället för 6h om alla matcher spelas sekventiellt:

# Schemaläggning




Sekventiell

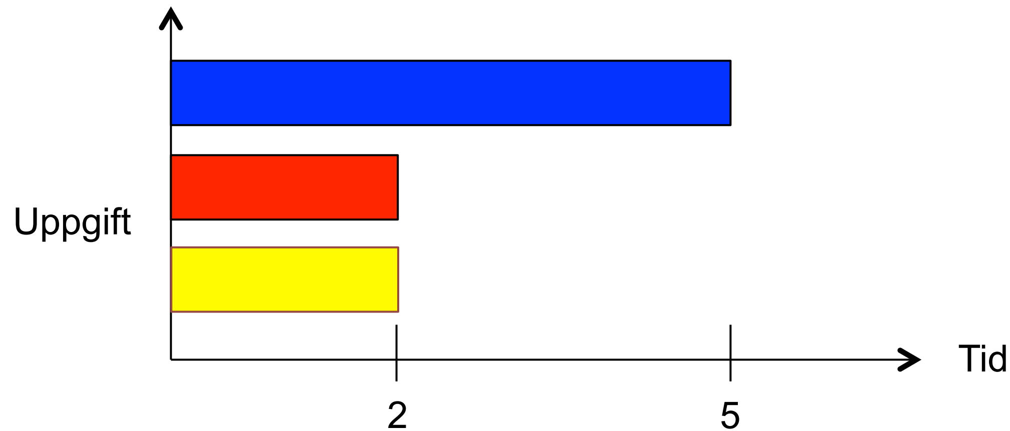



Parallell

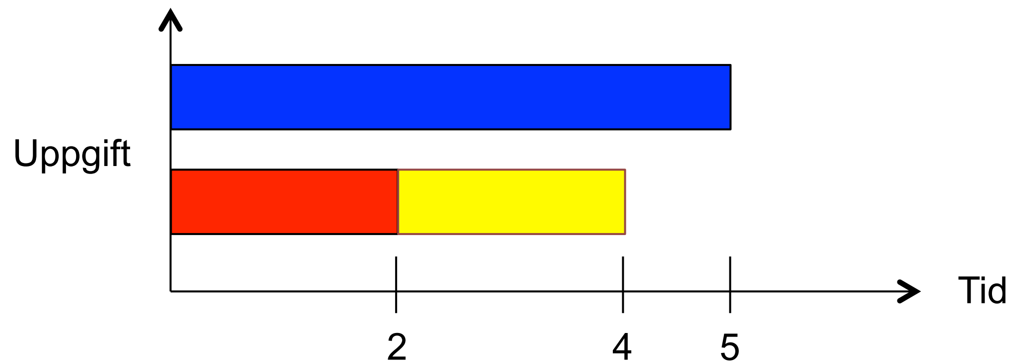
# Schemaläggning

- Schemaläggning innebär att avgöra vilken deluppgift som ska köras på vilken processor/core och när den ska köras
- Schemaläggning av en fotbollsturnering är enkelt, men för många verkligen problem kan det vara svårare
- Antag att vi har tre deluppgifter som tar olika tid att slutföra:

# Schemaläggning




Alternativ 1:
Tidsåtgång: 5 t
Kärnor: 3




Alternativ 2:
Tidsåtgång: 5 t
Kärnor: 2

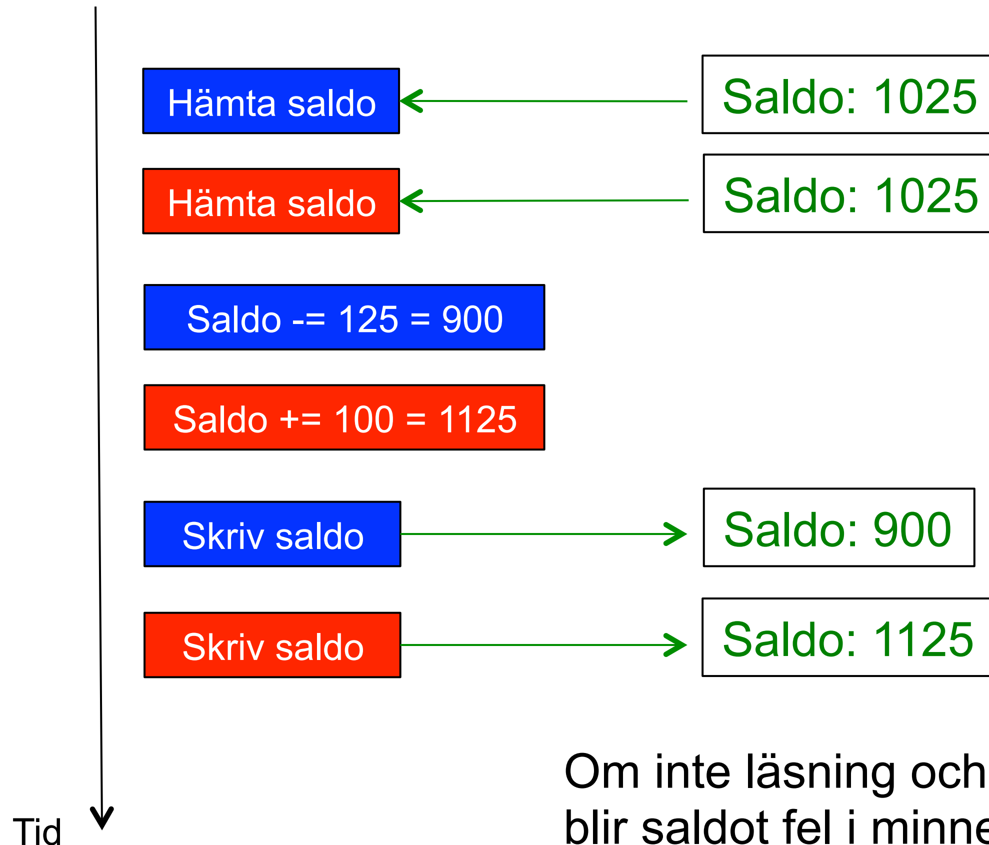
# Prestanda

- Fler kärnor/processorer ger inte alltid bättre prestanda
- En turnering med fyra lag går inte snabbare om vi ökar antalet fotbollsplaner från 2 till 3
- Synkronisering, när en process väntar på att en annan process ska bli färdig, är tekniskt ett ganska svårt problem att lösa effektivt
- Gemensam tillgång till data i minnet och administrering av processer (overhead) ställer också till problem med prestandan

# Prestanda

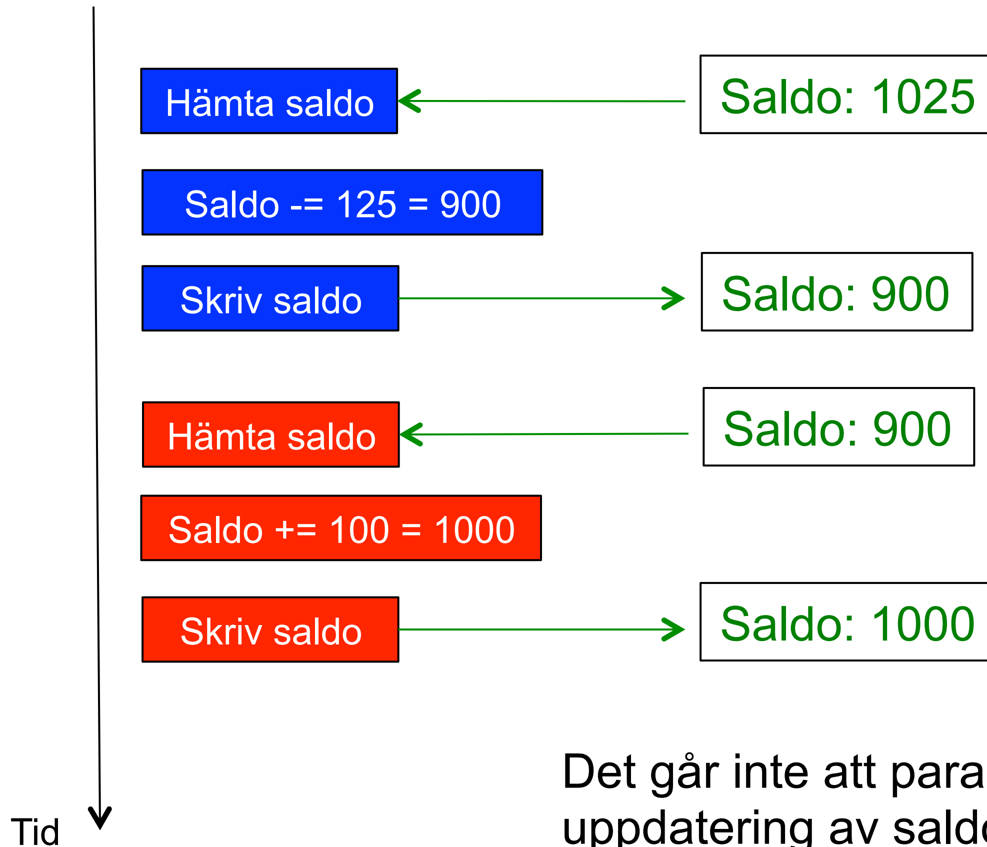
- Om varje deluppgift går väldigt snabbt att slutföra blir overhead för stor och prestandan kan till och med bli sämre än med sekventiell exekvering
- Prestandaproblem kan också uppstå om tiden det tar att slutföra olika deluppgifter varierar stort (= komplex schemaläggning)
- Om deluppgifter kräver tillgång till samma data, t.ex. saldot på ett bankkonto, kan datasynkronisering leda till dålig prestanda

# Datasynkronisering




Om inte läsning och skrivning synkroniseras blir saldot fel i minnet (kallas *race condition*)

# Datasynkronisering




Det går inte att parallellisera läsning och uppdatering av saldot

# Datasynkronisering

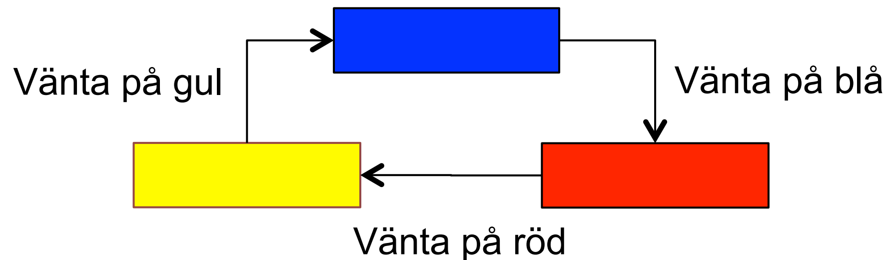
- För att undvika felaktig data i minnet används två typer av lås: läs- och skrivlås (*read lock, write lock*)
- För att hämta ett värde skapar processen ett läslås, värdet hämtas, och läslåset släpps
- För att uppdatera ett värde skapar processen ett skrivlås, värdet hämtas och ändras, skrivs tillbaks till minnet, och skrivlåset släpps

# Datasynkronisering

- Regler:
  - Läslås tillåts förutom om en annan process har ett aktivt skrivlås
  - Skrivlås tillåts bara om ingen annan process har ett läs- eller skrivlås
- Läsning kan göras parallellt
- Skrivning tillåts bara av en enda process i taget

# Deadlock

- Deadlock uppstår om två eller fler processer har ett cirkulärt beroende mellan varandra:




- Ingen av processerna blir klara – systemet hänger sig
- Dessa situationer måste detekteras och en eller flera processer måste avbrytas

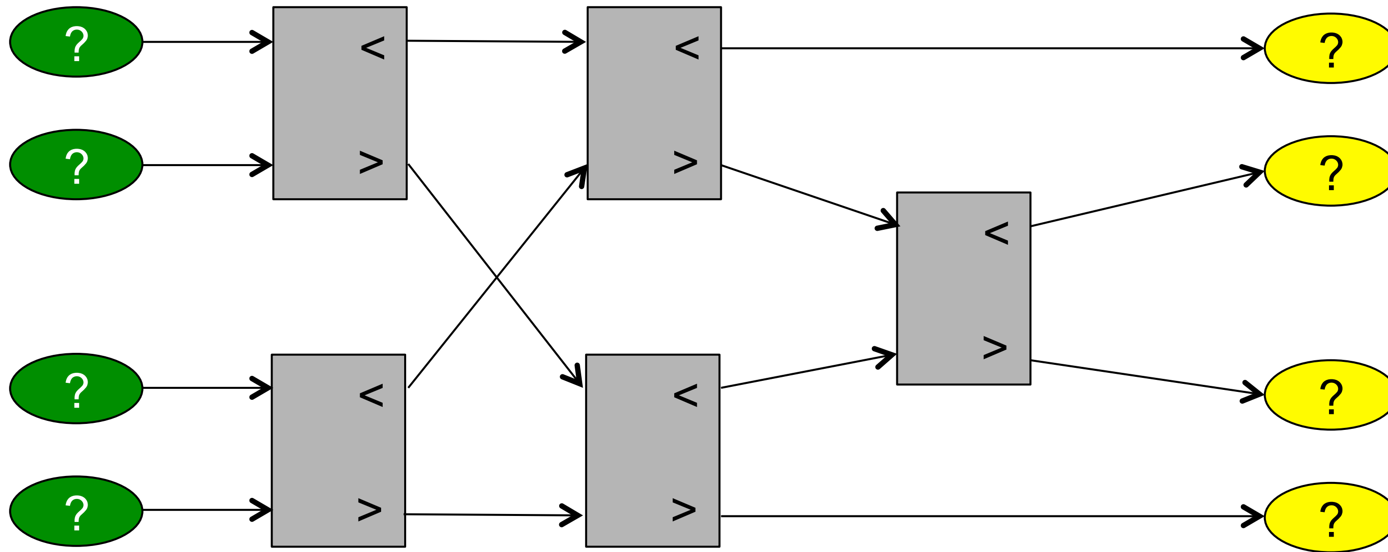
# Prestanda

- Det bästa sättet att avgöra om parallellisering är bra eller ej är att implementera både en sekventiell och en parallell version, köra de båda upprepade gånger och mäta prestanda
- Prestandan beror inte bara på hur problemet kan delas upp i deluppgifter, utan även hur operativsystemet administrerar processer
- Det är väldigt svårt att teoretiskt avgöra hur stor, om någon, prestandavinst som görs med parallellisering
- Generellt kan sägas att du behöver stora datamängder (>100000) för att parallellisering ska löna sig

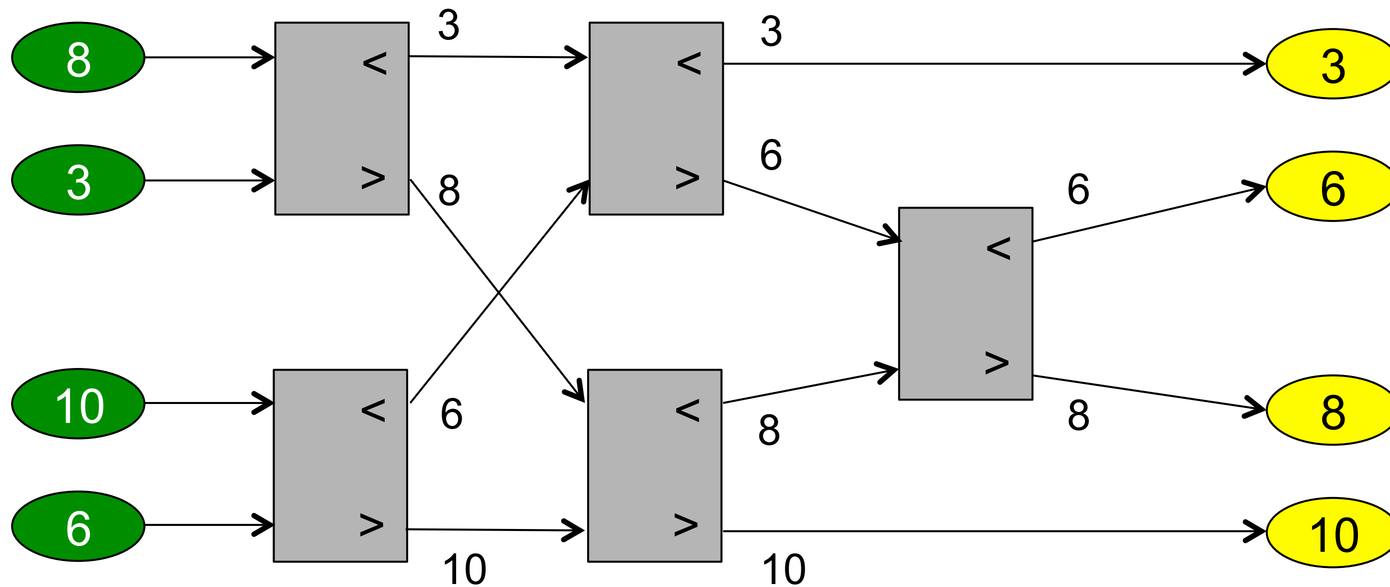
# Parallell sortering

- Sortering av värden kan göras parallellt med ett sorteringsnätverk (*sorting network*)
- Varje enhet i nätverket tar två värden som indata, jämför dessa och returnerar det minsta respektive högsta värdet
- De båda sorterade värden blir indata till andra enheter
- Enheterna kan köras parallellt på flera kärnor

# Sorteringsnätverk



# Sorteringsnätverk

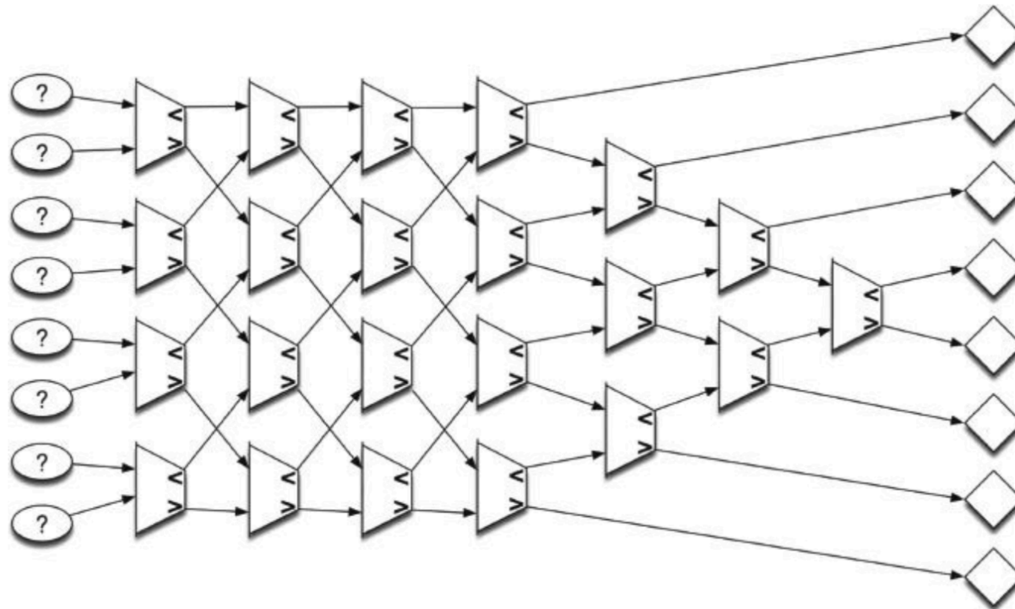



Prestandan blir inte bättre om fler än två kärnor används!

# Sorteringsnätverk

- I exemplet med fyra värden att sortera krävs 5 enheter
- Sekventiellt krävs alltså 5 tidsenheter, och parallellt 3 tidsenheter
- Beroende på overhead är det tveksamt om vi i praktiken tjänar på parallelliseringen

# Sorteringsnätverk



- Sortera 8 värden kräver 22 enheter
- Parallellt krävs 7 tidsenheter jämfört med 22 sekventiellt
- Cirka tre gånger snabbare, om man inte tar hänsyn till overhead