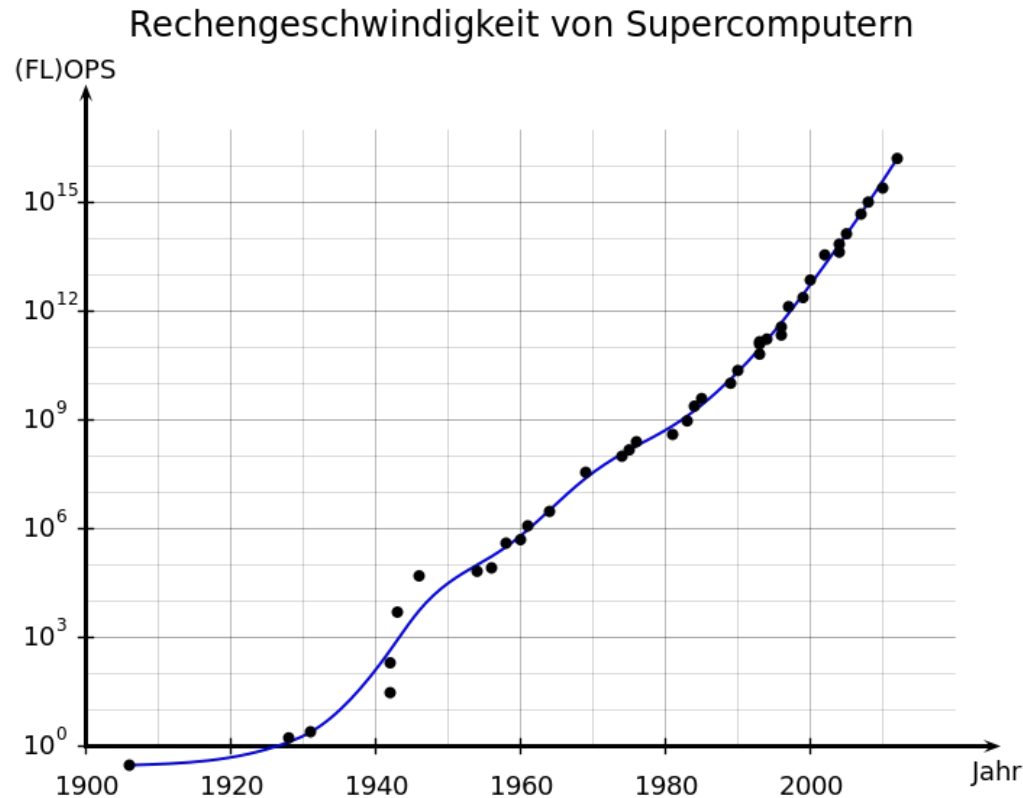


Tag der Informatiklehrer

Paralleles Programmieren für moderne Multicore-Prozessoren

Prof. Dr. Hans Jürgen Ohlbach

Supercomputing,
ca. ab 1975 mit Parallelrechnern (ILLIAC IV)



SuperMUC

- derzeit 3.2 Pflops
- demnächst 6.4 Pflops
- 155,656 Prozessorkerne
- 300 TB Ram
- 14 PB Plattenplatz



- derzeit 33,86 PFlops
 - 3,2 Mio Prozessorkerne
 - 1,3 TB Ram
 - 12,4 PB Plattenplatz
- Siehe <http://www.top500.org/>



Angenommen, der Rechner würde nicht
30 PFlops/sec machen, sondern
30 PHüpf/sec (jeweils 1 mm).

Wie schnell wäre er dann?

Angenommen, der Rechner würde nicht
30 PFlops/sec machen, sondern
30 PHüpf/sec (jeweils 1 mm).

Wie schnell wäre er dann?

Lichtgeschwindigkeit:

$$300.000 \text{ km/sec} = 3 \cdot 10^8 \text{ m/sec} = 3 \cdot 10^{11} \text{ mm/sec.}$$

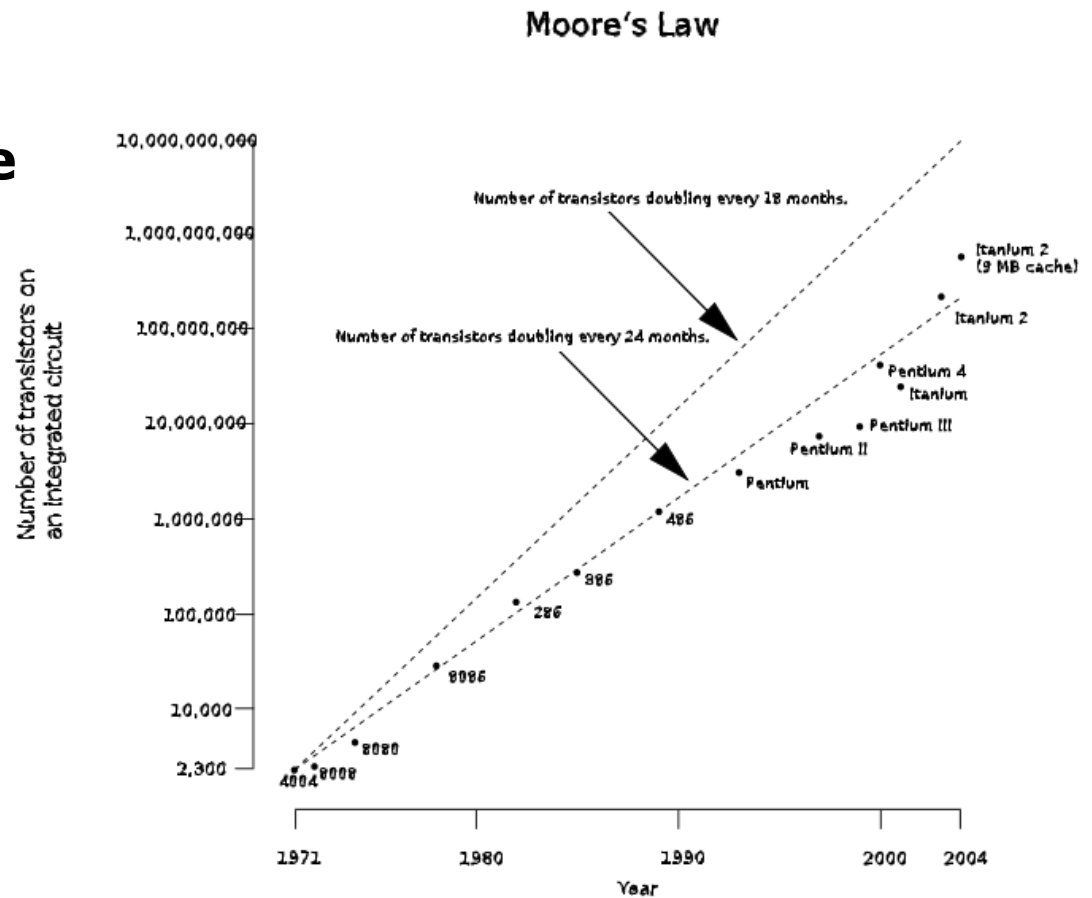
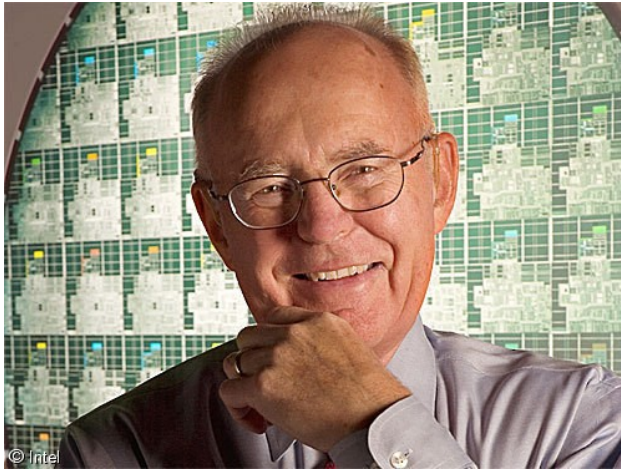
Tianhe-2-Hüpf:

$$30 \text{ PHüpf/sec} = 30 \cdot 10^{15} \text{ Hüpf/sec} = 3 \cdot 10^{16} \text{ Hüper/sec.}$$

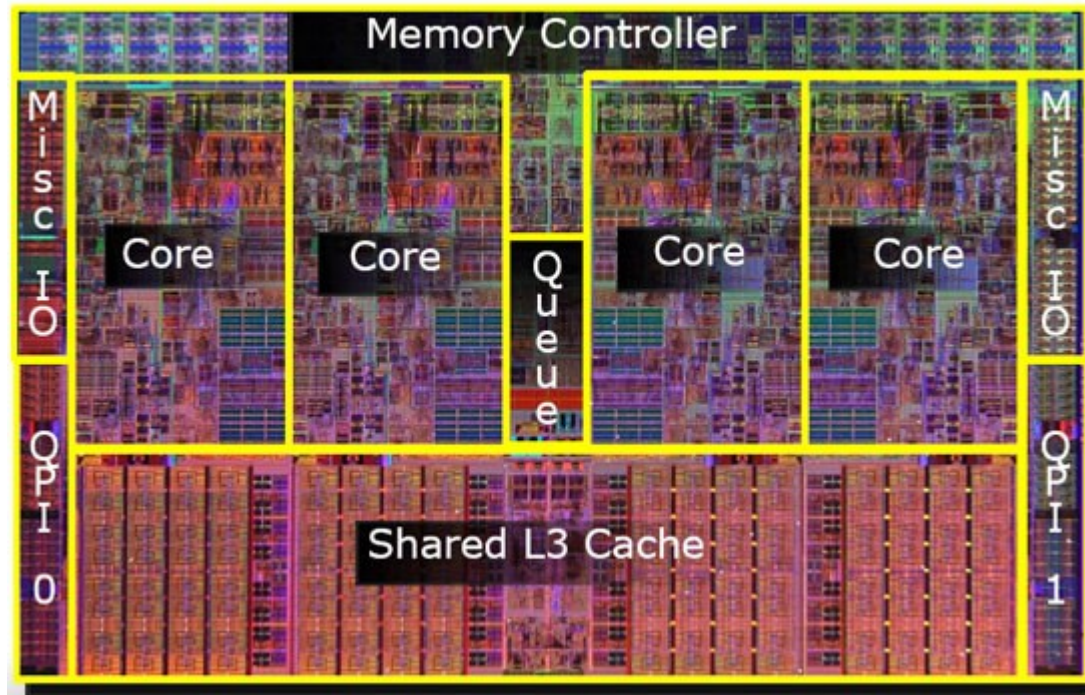
Das entspricht 100.000 facher Lichtgeschwindigkeit (Warp 9,999)

Mooresches Gesetz

Die Anzahl der Transistoren auf einem Chip verdoppelt sich alle 1.5-2 Jahre (Gordon Moore 1965)



über 730 Millionen Transistoren



Trotz des Mooreschen Gesetzes erhöht sich die Arbeitsgeschwindigkeit der Prozessoren in den letzten Jahren deutlich weniger.

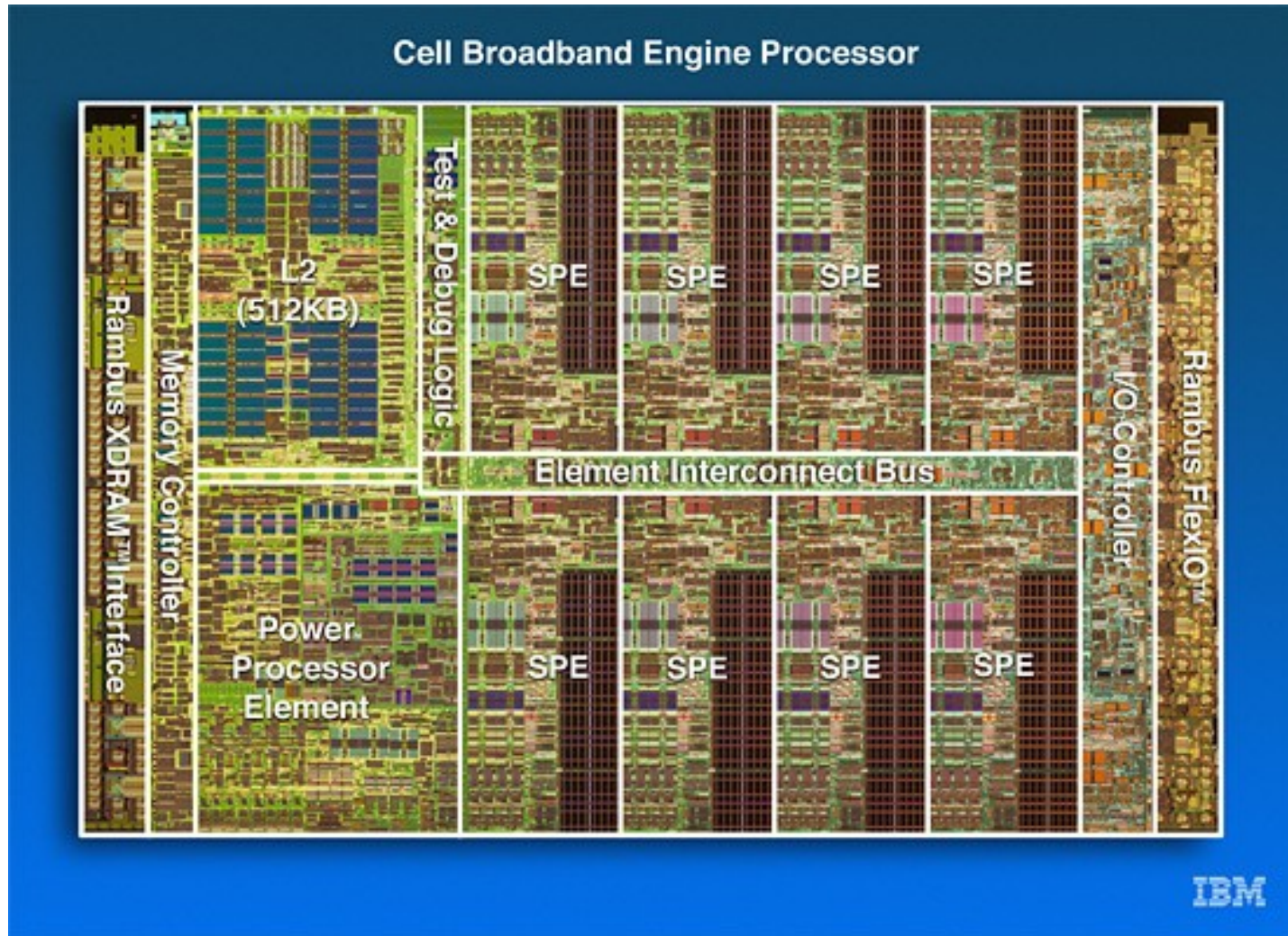
Gründe:

erhöhte Taktfrequenzen bewirken höheren Stromverbrauch und mehr Hitzeentwicklung.
Hier scheint eine Grenze des Machbaren erreicht.

Ausweg:

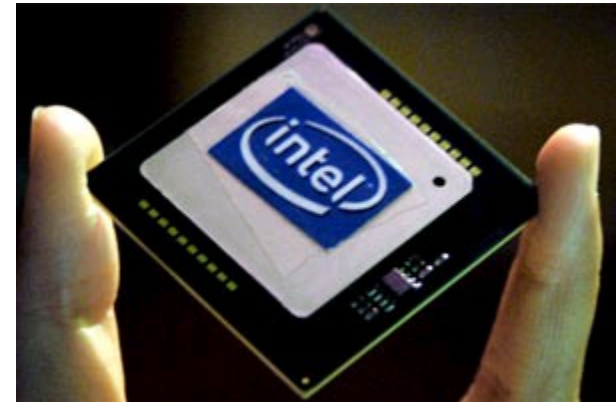
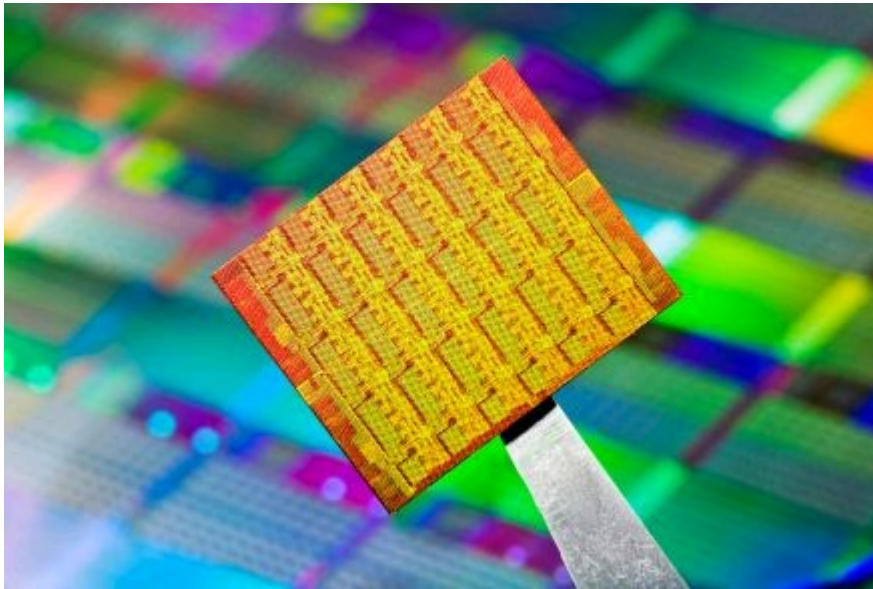
echte Parallelität durch Mehrkernprozessoren.

IBM Cell 8-Kern Prozessor



u.a. in Braunschweig entwickelt für Systementwickler freigegeben

<http://derstandard.at/1269449363706/Video-Intel-experimentiert-mit-48-Kern-CPU>



Mehrere Prozessorkerne werden vom Betriebssystem genutzt, um **unterschiedliche Programme parallel laufen zu lassen. Die allermeisten Programme für normale Computer sind aber so geschrieben, dass sie **nur einen** Prozessorkern benutzen. Rechenintensive Programme nutzen daher nur einen Bruchteil der tatsächlichen Kapazität.**

Prozesse: in Ausführung befindliche Programme

Threads: unterschiedliche (quasi)parallele Durchläufe durch ein Programm

Beispiel: Suchmaschine (Google):

Ohne Threads:

- entweder ein Benutzer muss warten bis alle anderen vor ihm fertig sind.
- oder für jeden Benutzer wird eine eigene Kopie des Suchprogramms gestartet.

Mit Threads:

für jeden Benutzer wird nur ein neuer Thread gestartet, der parallel zu den anderen läuft

Analogie: Küche

Prozesse: mehrere Küchen mit je einem Koch (verschiedene Rezepte)

Prozess mit *mehreren Threads*:

eine Küche mit mehreren Köchen und einem Rezept



2 Prozessorkerne für $n!$

Kern 1: $f1 = 1*3*5*\dots$

Kern 2: $f2 = 2*4*6*\dots$

$n! = f1*f2.$

Verallgemeinerung auf mehr Prozessorkerne ist offensichtlich.

Sortieren großer Listen (Bsp. Telefonbuch)

Serieller Bubblesort: analog Staffellauf

Paralleler Bubblesort: Phasenmethode

Parallelisierung:

manche Probleme lassen sich ganz einfach parallelisieren, bei manchen Problemen sind die Abläufe so verzahnt, dass sich Parallelisierung wegen des Koordinationsaufwands nicht lohnt. Bei anderen Problemen ist kreatives Algorithmendesign gefordert.

Wenn mehrere Prozesse/Threads auf die gleichen Ressourcen zugreifen (Speicher, Peripheriegeräte), können sie sich gegenseitig stören:

- Race Conditions
- Deadlocks

Kontoführung



Race Conditions im Beispiel

Kontostand: 1000

Thread für Ehemann

Abheben(Betrag) (300)

Lade Kontostand

Threadwechsel

Kontostand = Kontostand - Betrag

Sichere Kontostand

Thread für Ehefrau

Abheben(Betrag) (500)

Lade Kontostand

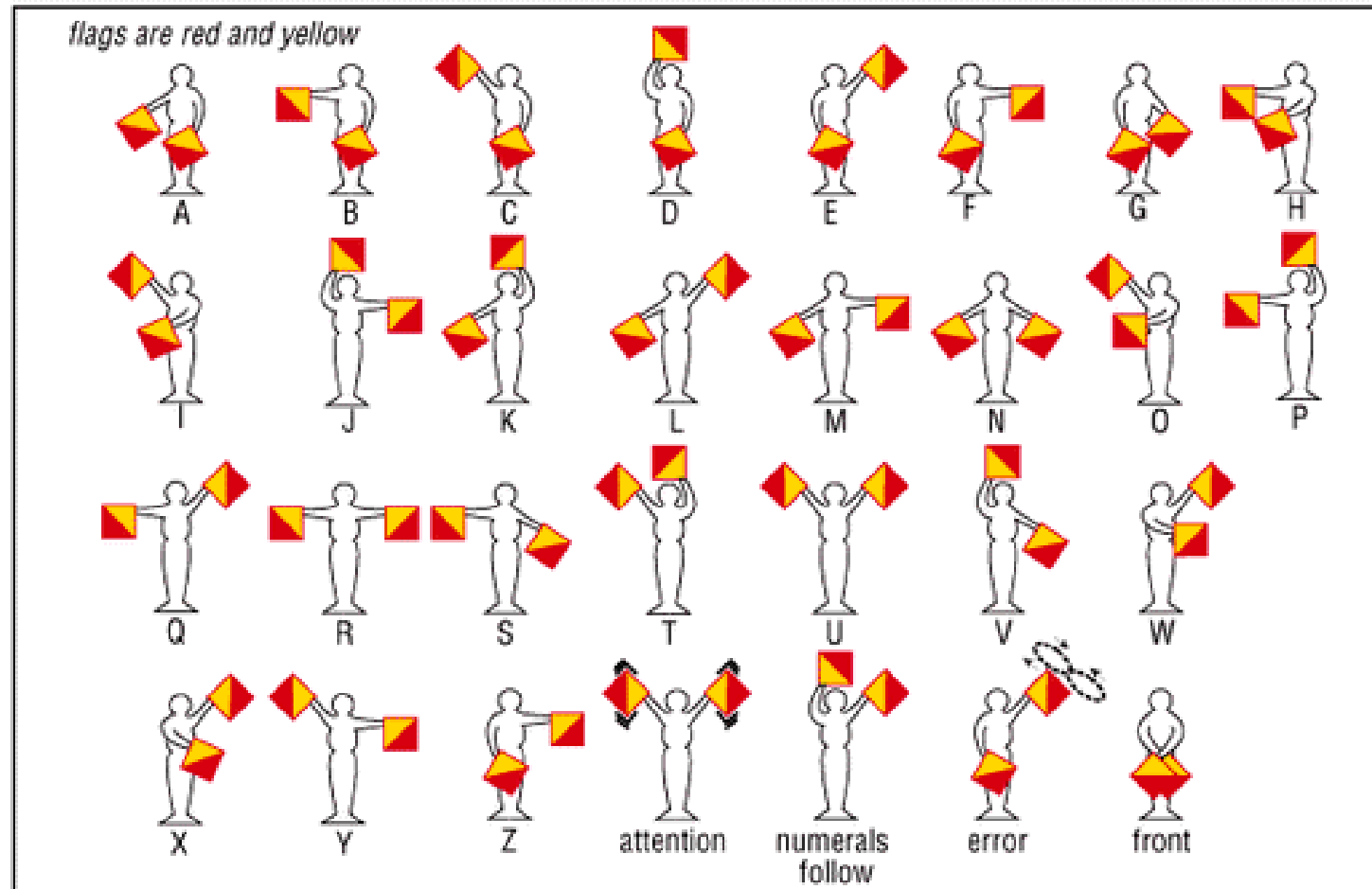
Kontostand = Kontostand - Betrag

Sichere Kontostand

Kontostand: 700

Kritischer Bereich

Hat nichts mit dem Semaphore Alphabet zu tun



Semaphore sind **Zählvariable**, die die Anzahl von freien Zugängen zählen.

3 atomare Funktionen:

`init(s,n)`: initialisiert Semaphore mit einer positiven Zahl

`up(s)`: (oder **signal(s)**) erhöht die Semaphore

`down(s)`: (oder **wait(s)**) erniedrigt die Semaphore bis auf 0;
blockiert wenn sie 0 ist.

Semaphore zum Schutz kritischer Bereiche

init(s,1); (binäre Semaphore)

Prozess A

down(s)

<kritischer Bereich>

up(s)

Prozess B

down(s)

<kritischer Bereich>

up(s)

Wenn Prozess A in den kritischen Bereich kommt, ist die Semaphore auf 0 und down() von Prozess B blockiert

Wenn Prozess A up() aufruft, ist die Semaphore wieder auf 1 und das down() von Prozess B wacht wieder auf.



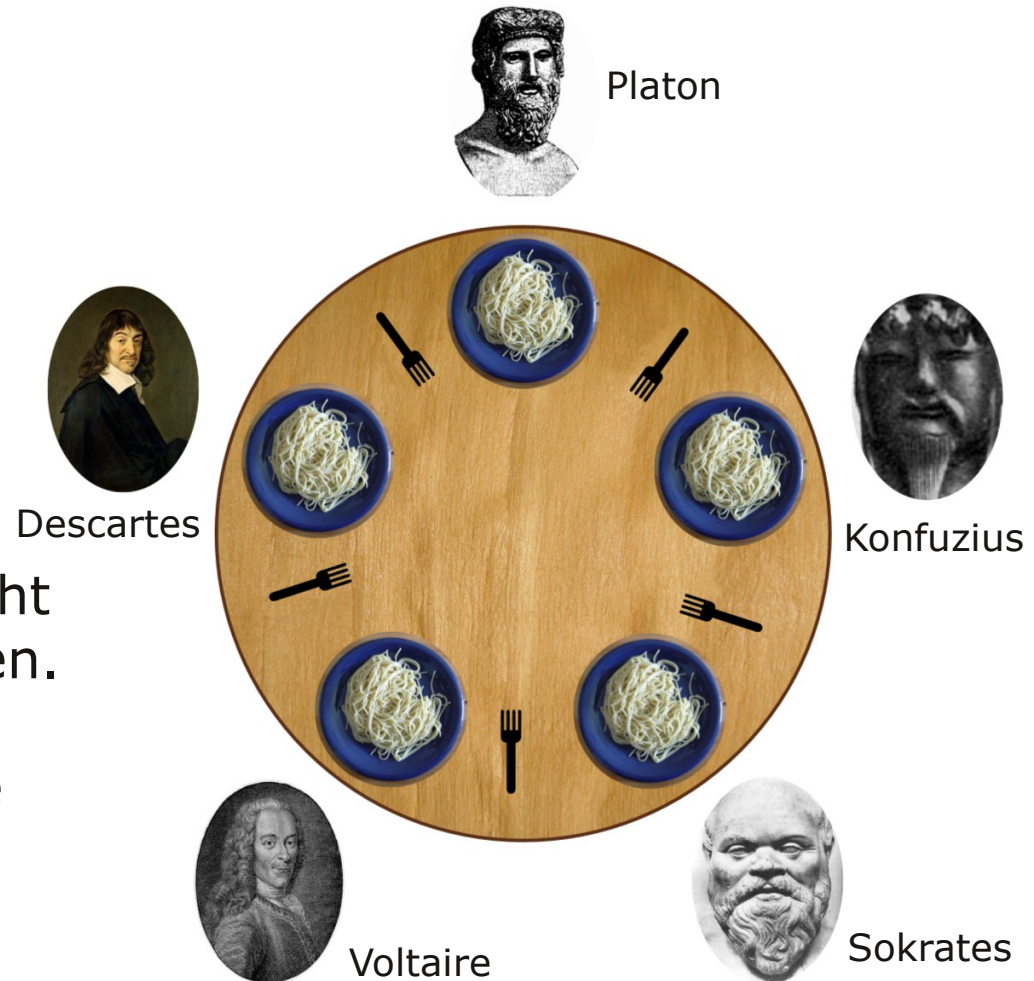
Das Kontobeispiel in Java

```
public class Konto {  
    public synchronized void einzahlen(int Betrag) {  
        int Kontostand = LadeKontostand();  
        Kontostand = Kontostand + Betrag;  
        sichereKontostand();  
    }  
  
    public synchronized void auszahlen (int Betrag) {  
        int Kontostand = LadeKontostand();  
        Kontostand = Kontostand - Betrag;  
        sichereKontostand();  
    }  
    ...  
}
```

Deadlocks: das Philosophenproblem

Jeder Philosoph braucht
zwei Gabeln zum Essen.

Wenn jeder die rechte
Gabel nimmt, kann
keiner essen.



Deadlocks bei Prozessen

Mindestens zwei Prozesse befinden sich in einem gegenseitigen Deadlock (Verklemmung), falls jeder Prozess Ressourcen hält, die ein anderer braucht, und Ressourcen braucht die ein anderer hält.

Beispiel: elektronisches Bankensystem:

Prozess P will Geld von Konto A nach Konto B buchen.

Prozess Q will Geld von Konto B nach Konto A buchen.

Prozess P öffnet Konto A und sperrt es für andere Prozesse

Prozess Q öffnet Konto B und sperrt es für andere Prozesse

-> Deadlock!

Lösung: Ordnung auf den Ressourcen, Zugriff nach der Ordnung

Beispiel: Fibonacci-Zahlen: $f_0 = 0$; $f_1 = 1$; $f_n = f_{n-1} + f_{n+1}$

```
class Fib extends FJTask {  
    static final int threshold = 13;  
    volatile int number;  
    Fib(int n) { number = n; }  
  
    public void run() {  
        int n = number;  
        if (n <= threshold) number = seqFib(n);  
        else {  
            Fib f1 = new Fib(n - 1);  
            Fib f2 = new Fib(n - 2);  
            coInvoke(f1, f2);  
            number = f1.number + f2.number;}  
    }  
}
```

übergibt die Aufgaben an
„worker threads“

Windows:

Task Parallel Library (TPL) (.NET-Framework)

Java:

Java Parallel Programming Framework

C/C++:

pThread-Bibliothek

OpenMP (C, Fortran, C++)

Schleifen werden automatisch verteilt.

Multi-Core Standard Template Library (MCSTL)
(Parallelisierung der C++ STL)

Bisher konnte man sich darauf verlassen, dass ein (serielles) Verfahren mit der nächst schnelleren Rechnergeneration automatisch auch entsprechend schneller wurde.

Mit Mehrkernprozessoren ist das nicht mehr so! Nahezu alle bisherigen Problemlösungen müssen überarbeitet werden, um sie effizient zu parallelisieren.
Das ist noch ziemlich am Anfang.