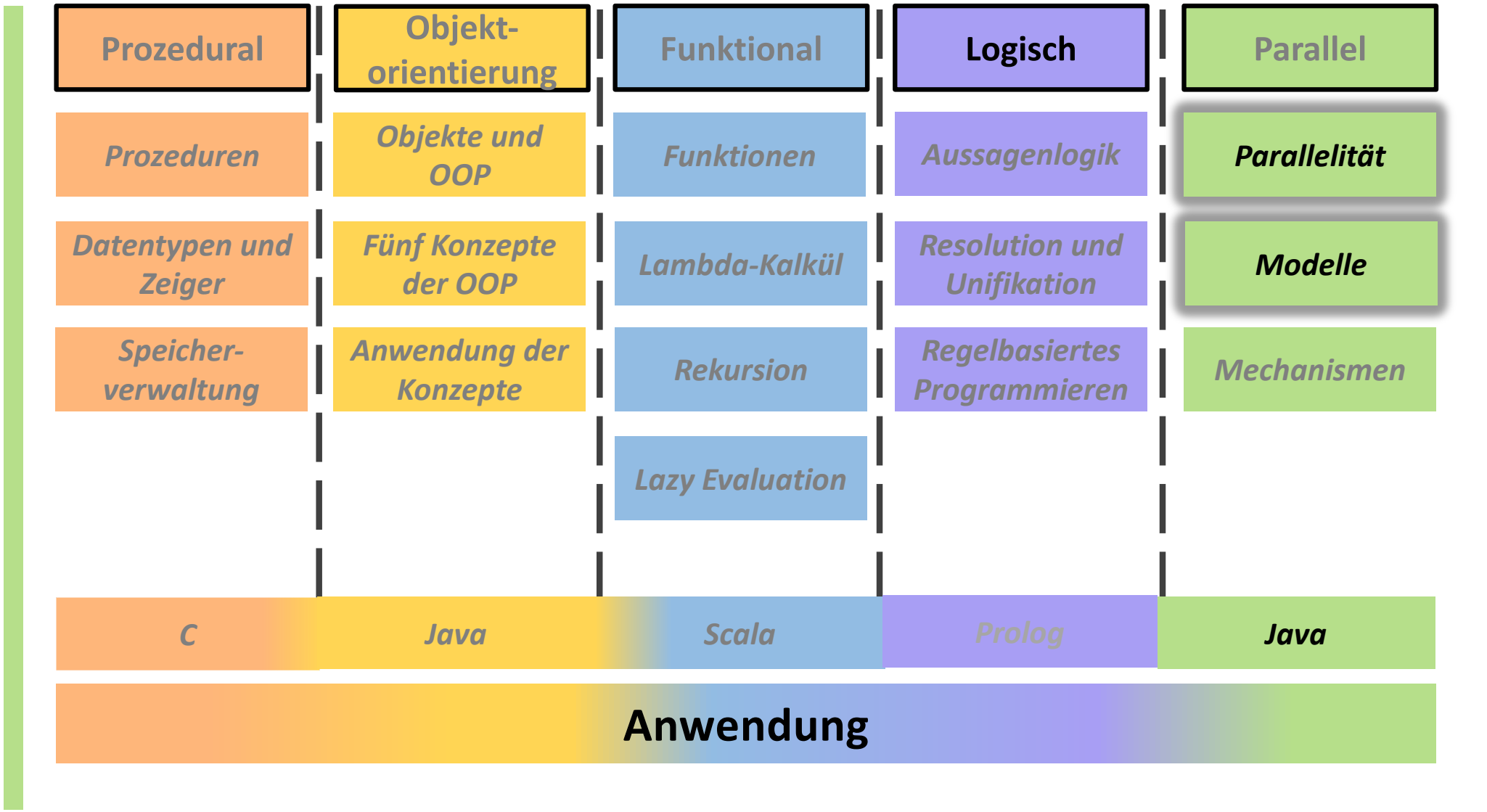
Grundlagen paralleler Strukturen

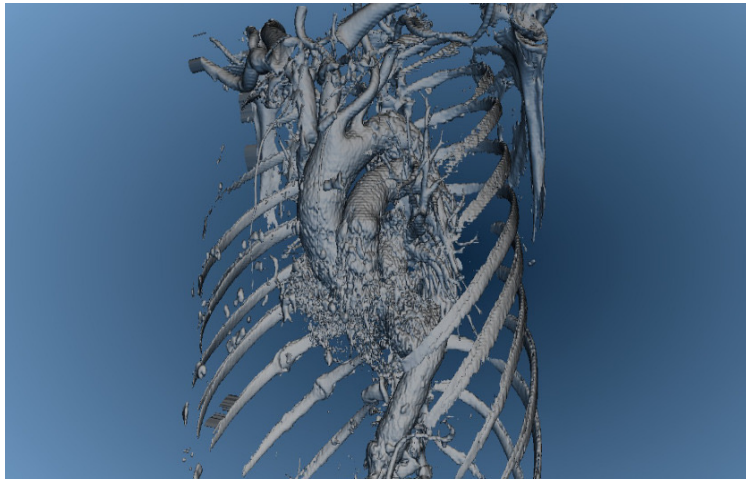
Paralleles Paradigma

Übersicht

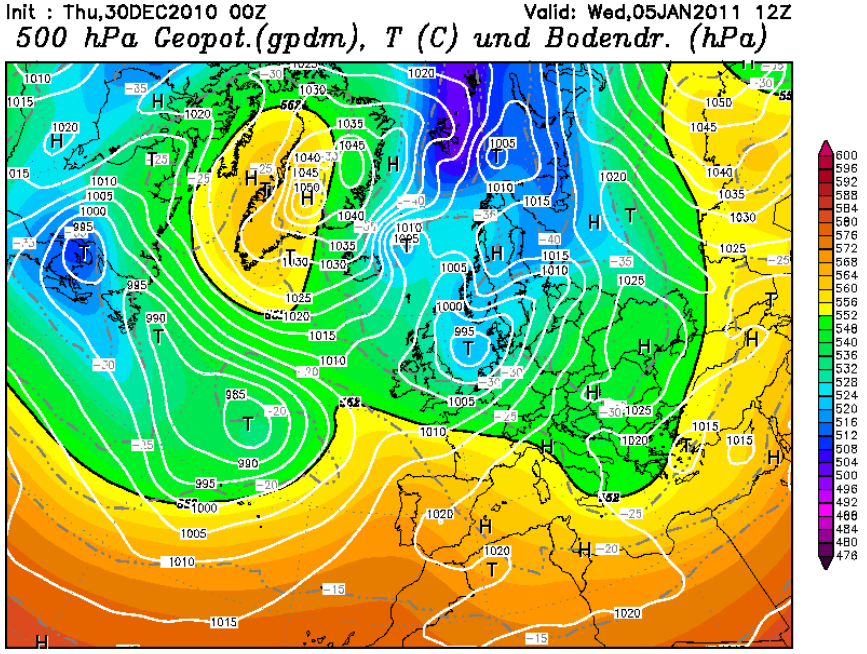

Prozedural
Objekt-orientierung
Funktional
Logisch
Parallel
Prozeduren
Objekte und OOP
Funktionen
Aussagenlogik
Parallelität
Datentypen und Zeiger
Fünf Konzepte der OOP
Lambda-Kalkül
Resolution und Unifikation
Modelle
Speicher-verwaltung
Anwendung der Konzepte
Rekursion
Regelbasiertes Programmieren
Mechanismen
Lazy Evaluation
C
Java
Scala
Prolog
Java
Anwendung

Warum Parallele Programmierung

- Anwendungsszenarien benötigen immer mehr Leistung
  - Daten-Visualisierungen (Medizintechnik)
  - Computerspiele
  - Datamining (z.B. Genomanalysen, Überwachung von Bankdaten)
  - Wetter-Simulationen
  - …



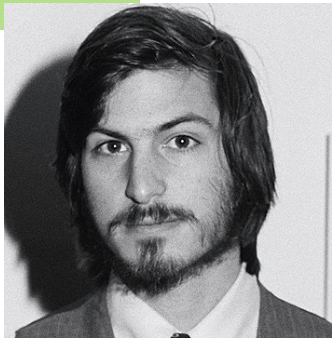
(c) http://www.aec.at/



Zitate

"We stand at the threshold of a many core world. The hardware community is ready to cross this threshold.
The parallel software community is not." **Tim Mattson**, principal engineer at Intel (2008)

"The way the processor industry is going, is to add more and more cores, but nobody knows how to program those things. I mean, two, yeah; four, not really; eight, forget it." **Steve Jobs, Apple (2008)**

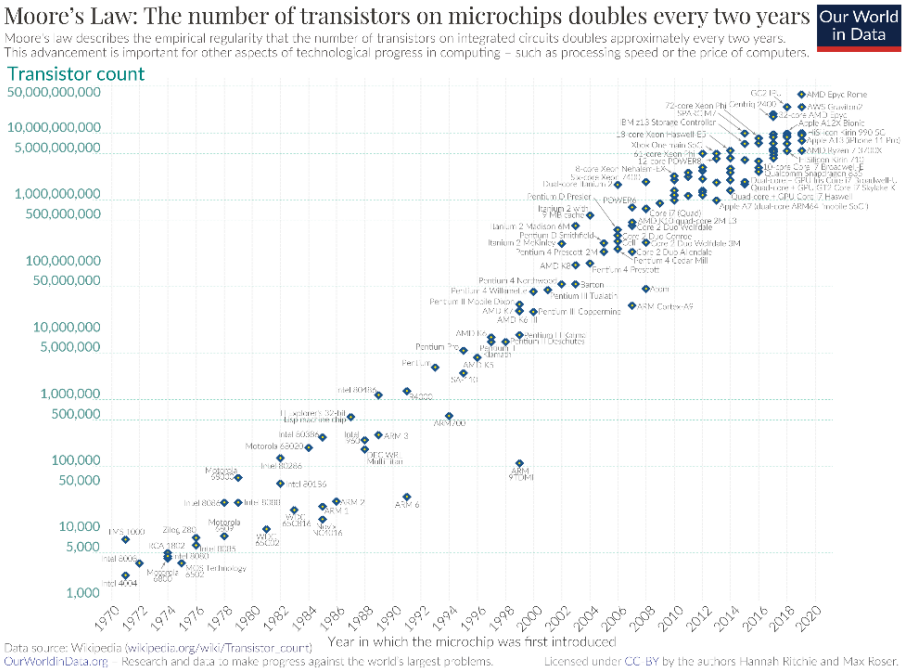


“Everybody who learns concurrency thinks they understand it, ends up finding mysterious races they thought weren’t possible, and discovers that they didn’t actually understand it yet after all.” **Herb Sutter**, chair of the ISO C++ standards committee, Microsoft (2005).

Entwicklung der Rechenkapazität

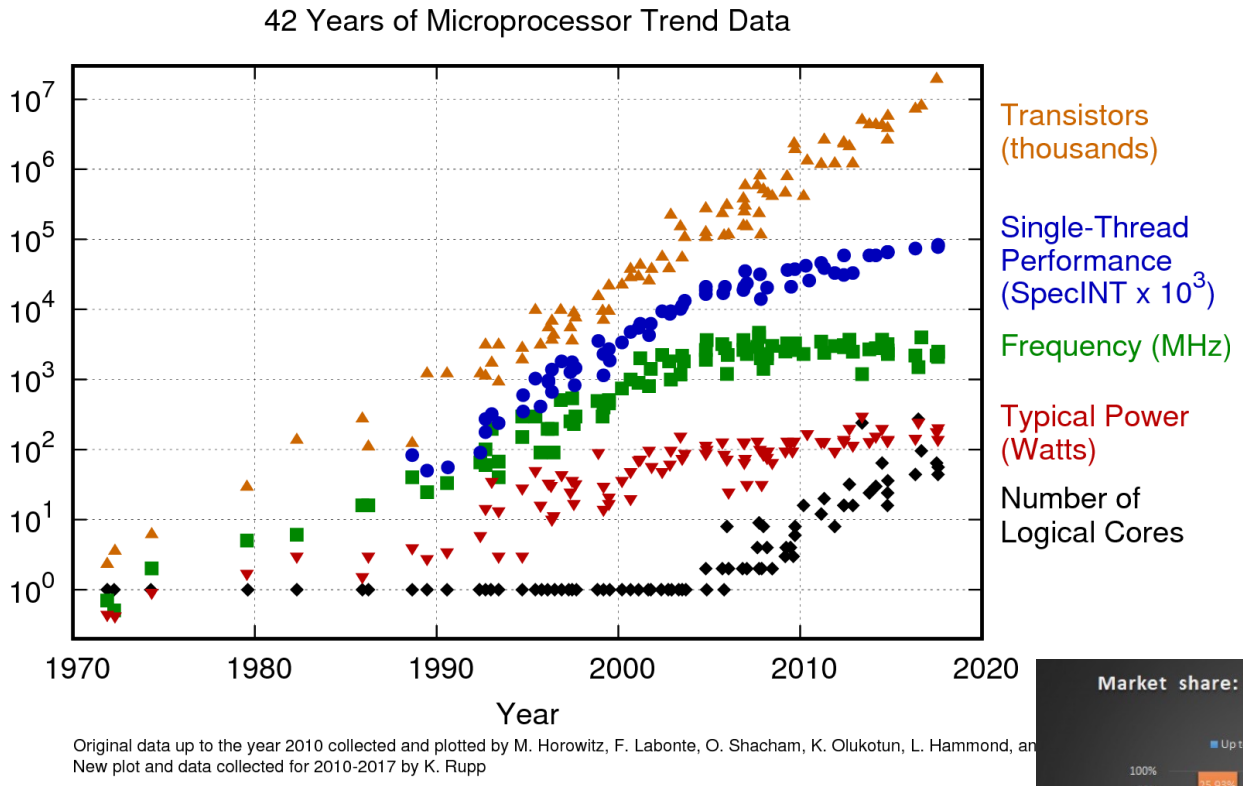
- Moores Gesetz:

  … Transistorendichte auf Computerchips verdoppelt sich alle zwei Jahre

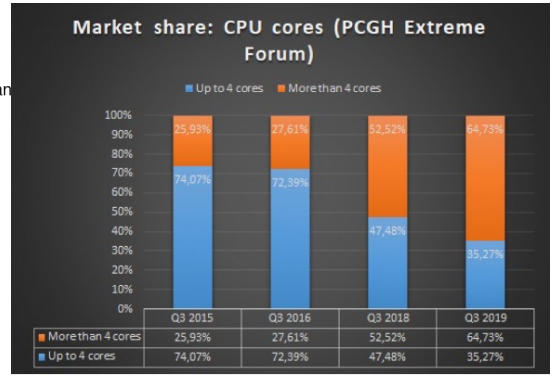


Moore‘s physikalische Grenzen

- physikalische Grenzen für Silizium gegen 2020 erreicht
- „normale“ max. Taktfrequenzen stagnieren bei ca. 5GHz



…CPU Speed stagniert (fast)
…Anzahl der Kerne wächst stark



Gründe der physikalischen Grenzen

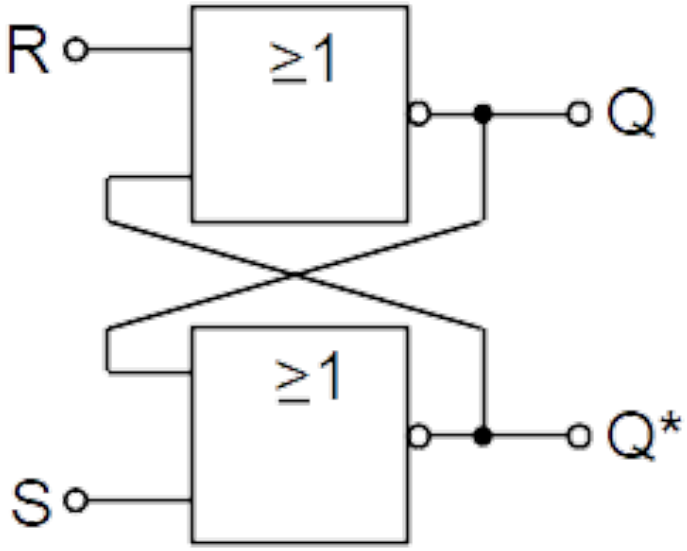
- Beschränkung der Taktfrequenzen:
  - Lichtgeschwindigkeit: 300.000 km/s
  - 10 GHz-Takt = Lichtweg nur noch 3cm
  - thermische Effekte
  - Front-Side-Bus automatisch beschränkt

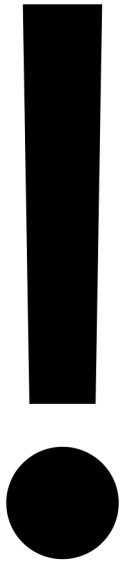
- Beschränkung durch Chip:
  - Aufbauen eines stabilen Zustandes beim Schalten jedes FlipFlops erfordert jeweils Tausende von Elektronenbewegungen

Flip Flops?

?

Flip Flops!





- RS-FlipFlop aus NOR-Gattern:

$$Q = \overline{R \vee Q^*} = \bar{R} \wedge \overline{Q^*} = \bar{R} \wedge (S \vee Q)$$

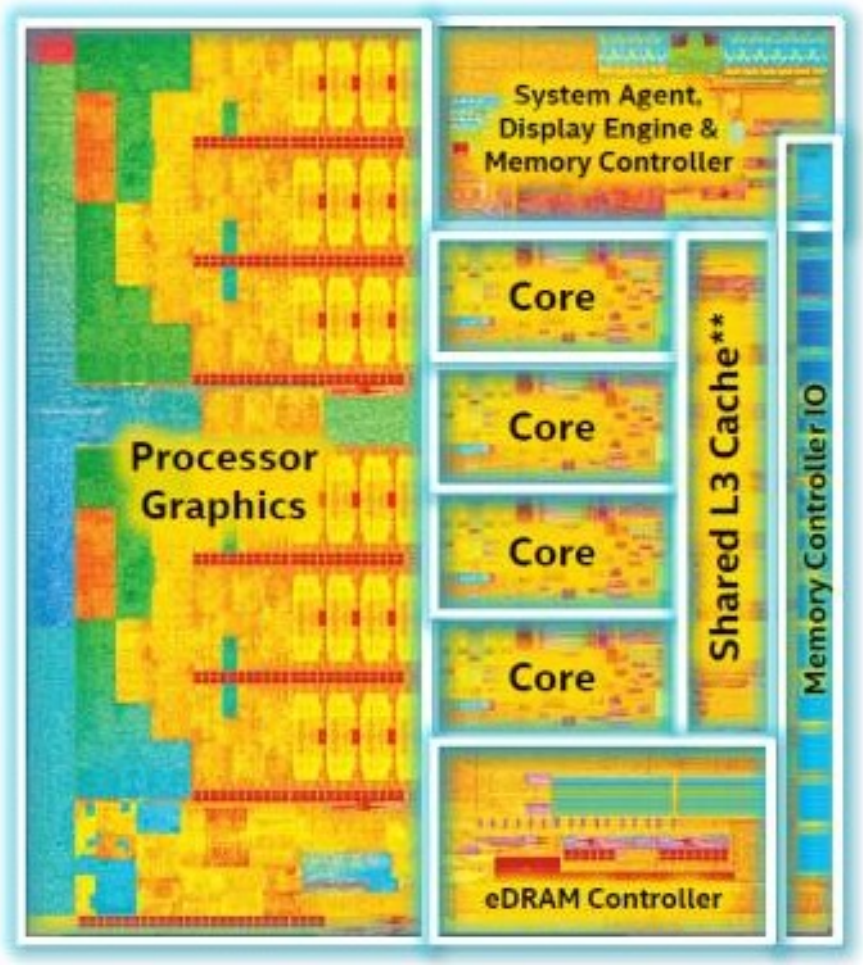
FF_NOR-RS by: https://commons.wikimedia.org/wiki/User:Saure

Probleme beim Speicherzugriff

- Taktfrequenz ist nicht das einzige Problem
- Begrenzung der Bandbreite für den Speicherzugriff:
  - Verarbeitung von immer mehr Daten pro Prozess
  - hochfrequenter Zugriff und Abruf größerer Datenmengen
- Single-Core-Prozessoren keine Lösung

Hardwarelösung

- neue Lösungen benötigt, um Anforderungen zu entsprechen
- **Idee:**
  Verteilung der Aufgaben auf mehrere Rechenkerne
- Bsp.:
  5th Gen Intel Core Processor



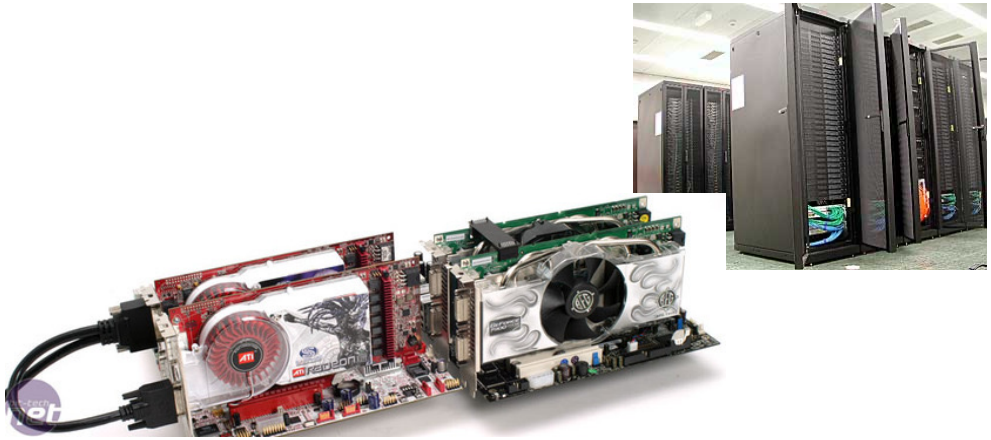
(C) Intel Corp.

Leistungssteigerungen durch Parallelisierung

Aufgaben für Programmierer

- keine per se Synchronisierung von Programmen möglich
- Beachtung von:

➢ **Parallelisierung**
  - Multicore Rechner
  - Grafikkarten/GPUs
  - Rechencluster



➢ **Skalierbarkeit**
  - Cloud Services

Zielstellung

Ziel dieser Vorlesung ist, dass Ihr:

- **Lernt**, wie parallele Anwendungen funktionieren
  - Parallele Architekturen
  - Ebenen der Parallelität
  - Threads vs. Prozesse
- **Wisst**, was bei parallelen Anwendungen zu beachten ist
  - Parallelisierung von Programmen
  - Threads zur Parallelisierung
  - Informationsaustausch
  - Message Passing Interface (MPI)

Parallele Architekturen

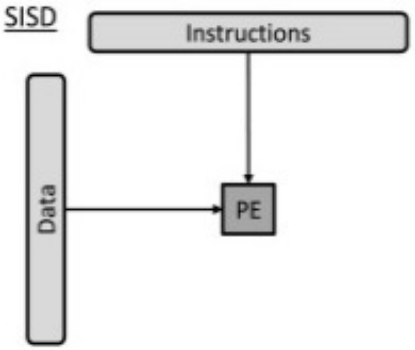
Flynn‘sche Klassifizierung

- = „theoretische Klassifizierung von Parallelrechnern“
- Flynn charakterisiert Parallelrechner nach:
  - Organisation der globalen Kontrolle
  - Daten- und Kontrollflüssen
- Vier Klassen:
  - SISD – Single Instruction Single Data
  - MISD – Multiple Instruction Single Data
  - SIMD – Single Instruction Multiple Data
  - MIMD – Multiple Instruction Multiple Data

Single Instruction – Single Data

- Herkömmlicher Rechner
- Eine Operation wird auf jeweils ein Datum angewendet

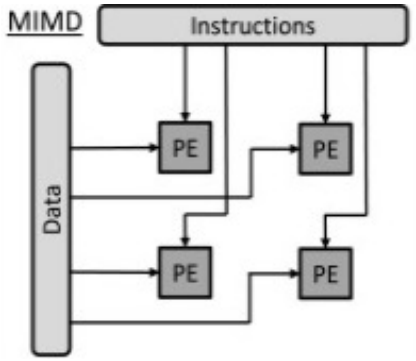
= „von-Neumann Architektur“



[Quelle: Rajkumar Buyya, Christian Vecchiola and S. Thamarai Selvi; Mastering Cloud Computing *Foundations and Applications Programming; 2013]*

Multiple Instruction – Multiple Data

- Verschiedene Operationen werden auf verschiedenen Daten ausgeführt

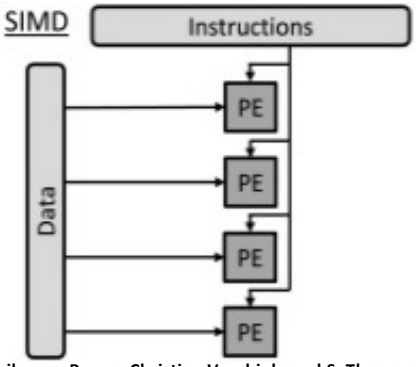


[Quelle: Rajkumar Buyya, Christian Vecchiola and S. Thamarai Selvi; Mastering Cloud Computing *Foundations and Applications Programming; 2013]*

= „klassischer multi core“

Single Instruction – Multiple Data

- Spezialisierter Rechner, der dieselbe Operation auf verschiedenen Datenbereichen gleichzeitig ausführt



[Quelle: Rajkumar Buyya, Christian Vecchiola and S. Thamarai Selvi; Mastering Cloud Computing *Foundations and Applications Programming; 2013]*

- Beispiel: 512Bit Vektorrechner könnte z.B. gleichzeitig 16 Additionen von Floats durchführen (1 Float = 16Bit; 2 Floats je Addition)

  = „Vektorrechner“

Einer fehlt…

- Multiple Instruction, Single Data
- Anwendung nur in sehr speziellen Bereichen
  - Fehlertoleranz
  - z.B. Space Shuttle

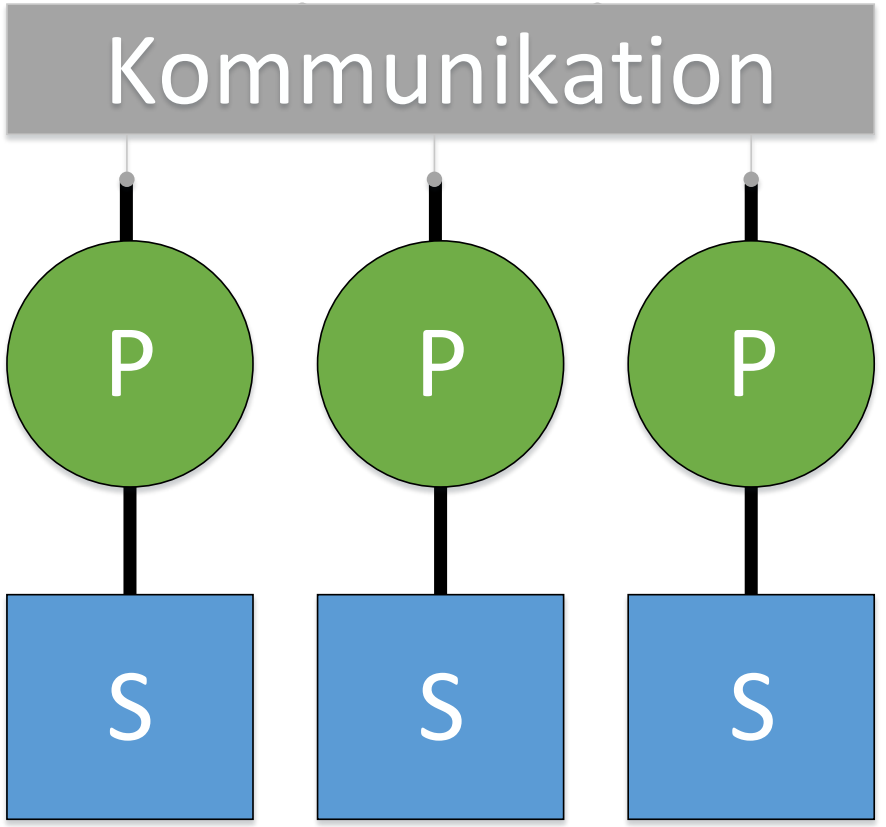
Speicherzugriffe

Parallele Speicherung

- Drei mögliche Herangehensweisen:
  - separate Datenspeicherung (**Distributed Memory**)
  - gemeinsame Datenspeicherung (**Shared Memory**)
  - Mischformen (**Hybride Systeme**)

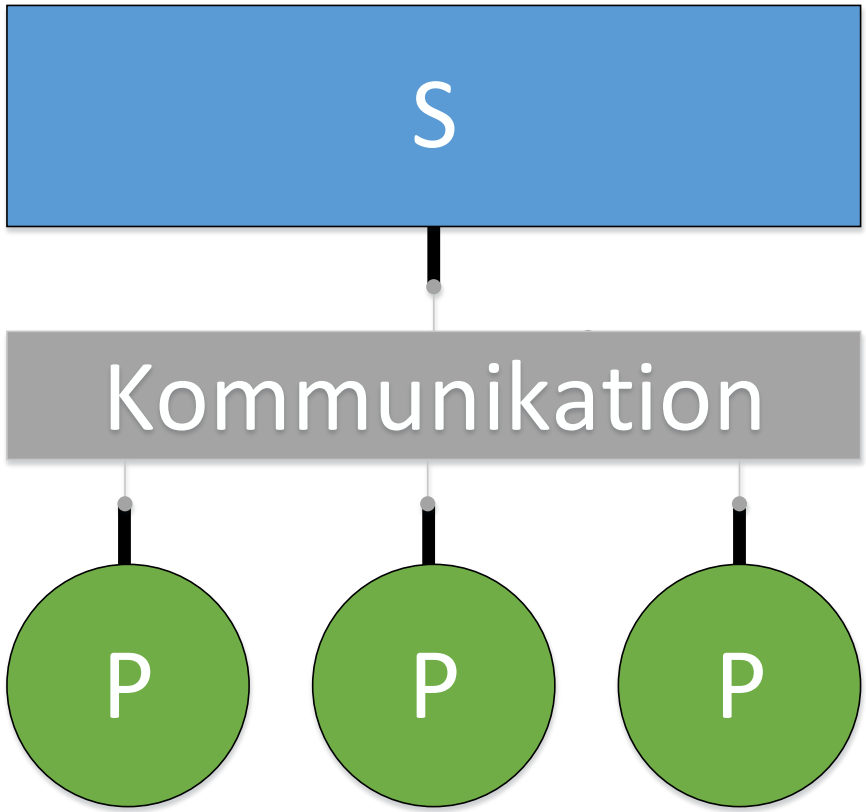
Distributed Memory

- eigenen lokalen Speicher für jeden Prozessorkern
- Kommunikation über internes Netzwerk
  - ➢ z.B. via Message-Passing
- Anwendung im klassischen Cloud-Computing



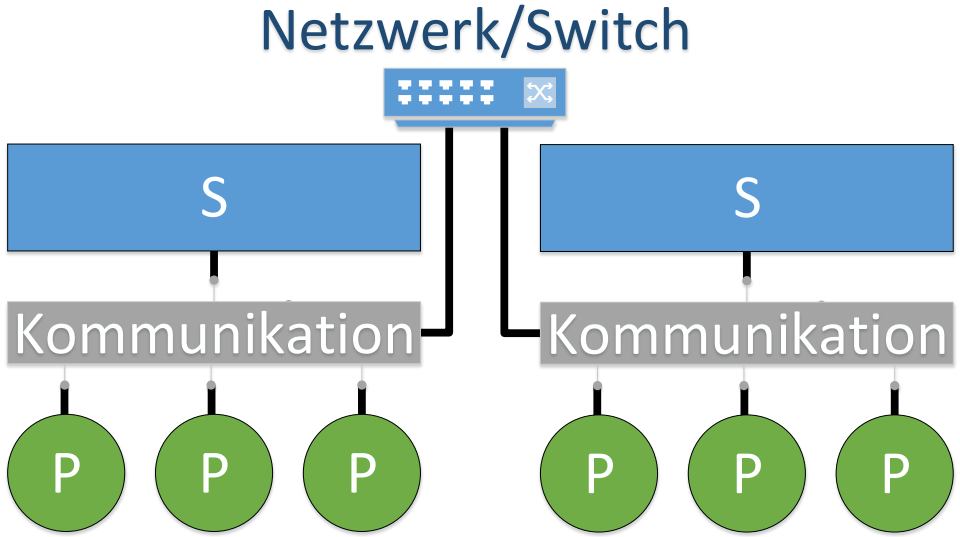
Shared Memory

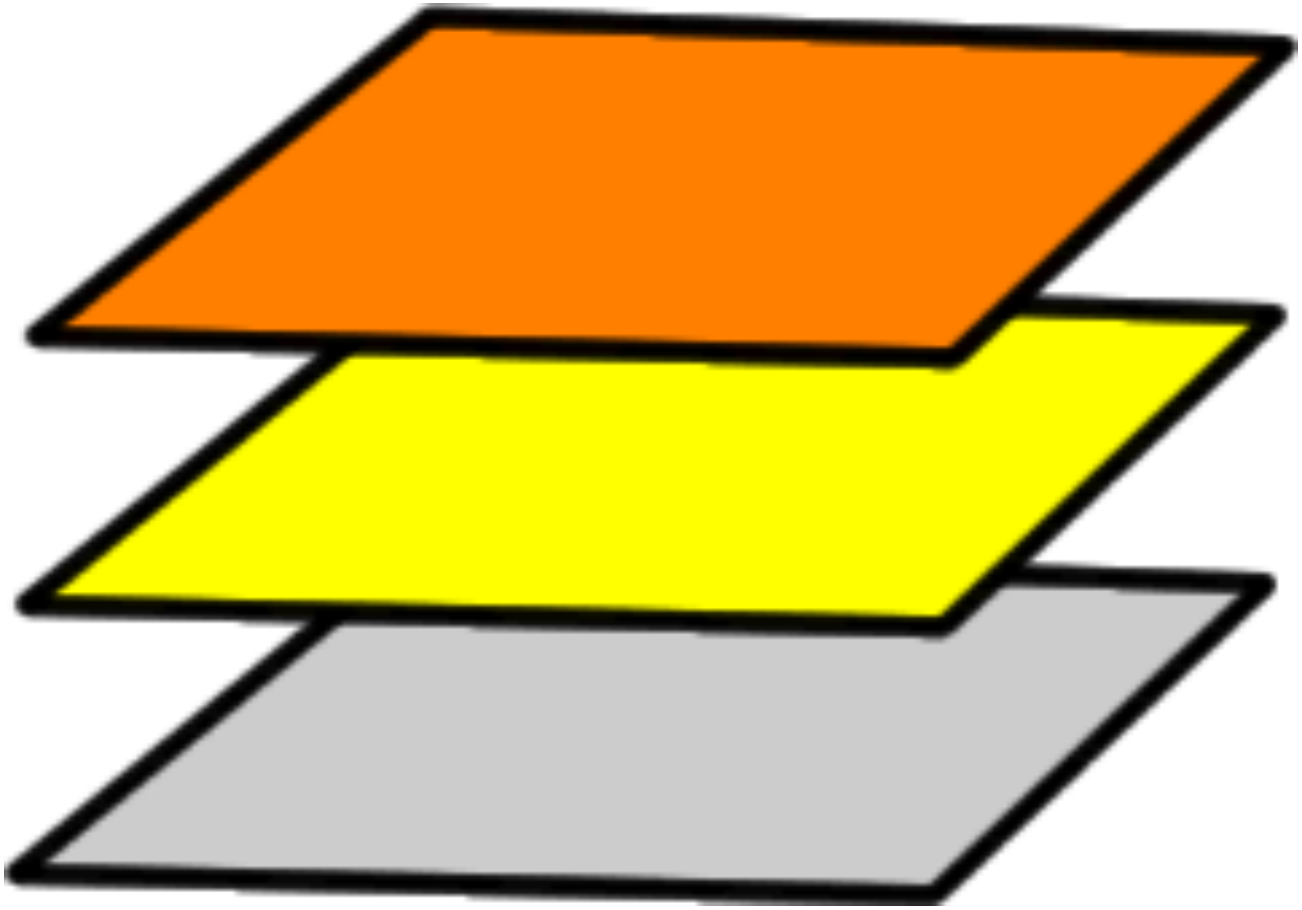
- Gleichzeitiger Zugriff auf gemeinsamen Speicher
- Kommunikation über gemeinsam genutzten Speicher
  - ➢ Multi-Threading
- homogene Prozessoren
  - ➢ einzelne oder parallele Ansprache von Prozessoren
- typische Multicore Prozessor-Architektur (außer L1/L2-Cache)



Hybride Systeme

- einzelne Knoten mit Shared Memory
- Knotenverbindung als Distributed Memory
- Programmiermodell:
  - **Message-Passing** zwischen den Knoten
  - **Multi-Threading** innerhalb der einzelnen Knoten
- Anwendung bei Rechenclustern und auf Skalierbarkeit ausgelegten Cloud-Diensten





Ebenen der Parallelität in der Software

Ebenen

- Hier zu betrachtende Ebenen:
  - **Instruktionsebene** – Parallelisierung auf Basis von Instruktionen
  - **Funktionsebene** – Parallelisierung auf Basis von Funktionalitäten
  - **Datenebene** – Parallelisierung auf Basis von Daten

Parallelität auf Instruktionsebene

CPU-Zyklus

- Ausführung einer Instruktion in mehreren vordefinierten Phasen; z.B. klassische RISC Pipeline
  - InstructionFetch (IF)
  - InstructionDecode (ID)
  - Execute (EX)
  - Memory Access (MEM)
  - Writeback (WB)

➔ Bei sequentieller Ausführung sehr viel ungenutzte Resourcen

Idee: Pipelining

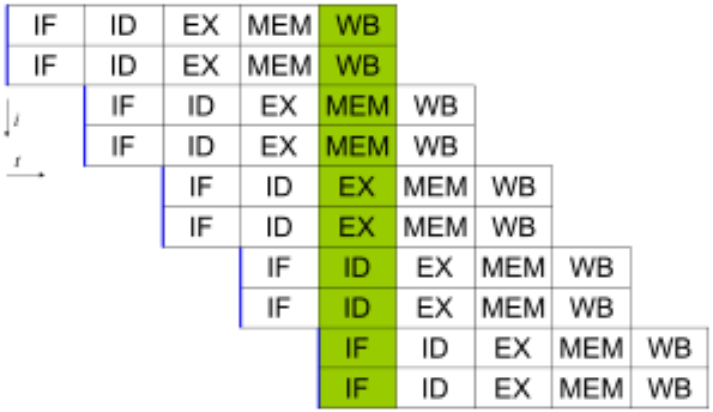
- Ausführung jeder Phase parallel möglich, da realisiert durch unterschiedliche Bauteile

- Prinzip des Pipelinings:
  - Idealismus: Abarbeitung so vieler Instruktionen „parallel“, wie Schritte in einem CPU-Zyklus vorhanden sind
  - Realität: Beachtung von Abhängigkeiten

Abhängigkeiten von Instruktionen

- Drei Arten der Abhängigkeit:
  - **Fluss-Abhängigkeit**: Instruktion1 berechnet ein Ergebnis, welches von Instruktion2 als Operand genutzt wird
  - **Anti-Abhängigkeit**: Instruktion1 verwendet ein Register als Operand, indem von Instruktion2 ein Ergebnis abgelegt wird
  - **Ausgabe-Abhängigkeit**: Instruktion1 und Instruktion2 verwenden dasselbe Register zur Ablage ihres Ergebnisses
- meist automatische Ausführung auf Prozessoren:
  - keine Interaktion durch Anwendungsprogrammierer

- parallele Ausführung mehrerer Instruktionen parallel



- Notwendige Voraussetzungen:
  - mehrere gleiche Ausführungseinheiten (z.B. ALUs)
  - größere Register z.B. zum gleichzeitigen Laden mehrerer Instruktionen
  - keine Abhängigkeiten zwischen parallel ausgeführten Instruktionen ➔ aufwendige Abhängigkeitsbestimmungen

Pipelining Wartezeiten

- Ziel: jede Phase möglichst ausnutzen (=Effizienz)

- **Aber:**

```
        if a>b
      /        \
then             else
  c:=a             c:=b
```

Welche Instruktion wird als nächstes in Pipeline geladen?

```
while !goOn
  sleep
```

Wollen wir wirklich so lange warten?

```
1 | a:= 5*c
2 | b:= a*4
```

Was machen wir bei Abhängigkeiten?

Parallelisierungsstrategien

- Zur Vermeidung von Wartezeiten zwei mögliche Parallelisierungtechniken:
  - Out-of-order Execution
  - Speculative Execution
- Gemeinsame Grundidee ist immer, den Prozessor-Zyklus zu füllen und somit die volle Kapazität auszunutzen

Out-of-Order Execution

- Schleifenausführung dauert immer Zeit, sollte aber terminieren:

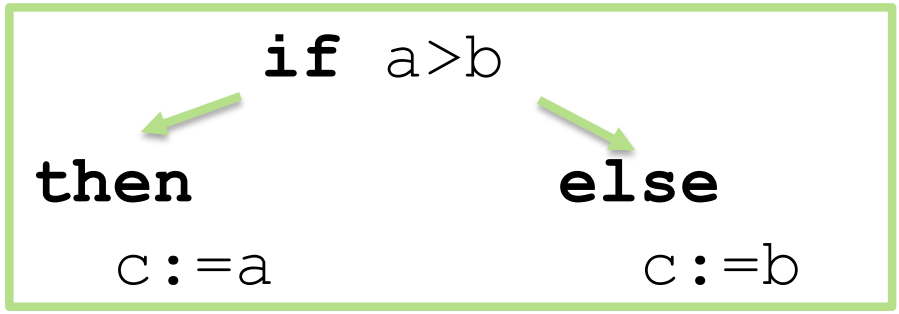
```
while !goOn
  sleep
```

- Idee: Vorziehen von Instruktionen aus dem weiteren Programmverlauf

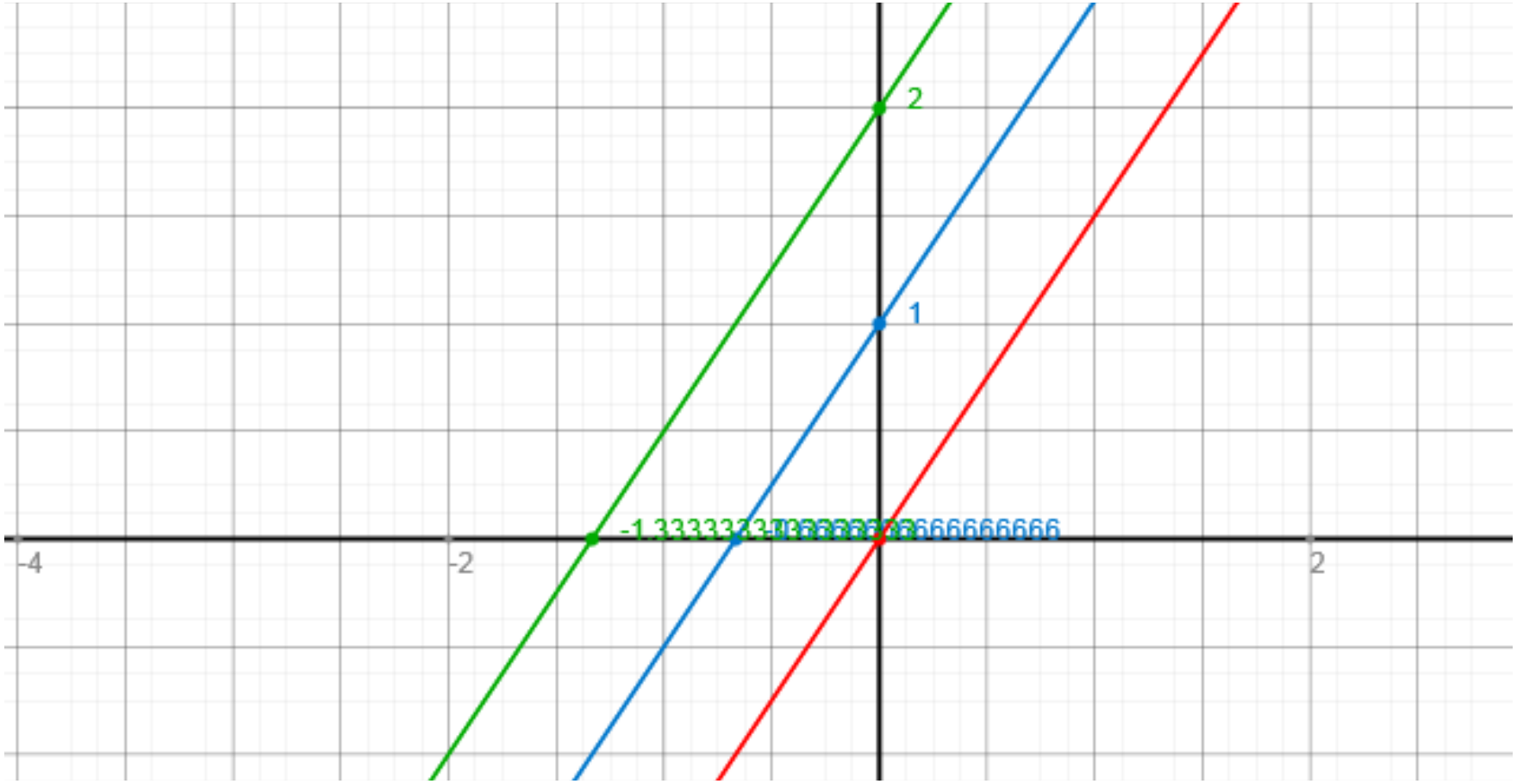
- Beachtung von Fluss-, Daten- und Ausgabeabhängigkeiten

Speculative Execution

- unklar, ob Ausführung des `then`- oder `else`-Zweiges:



- Idee: Spekulieren/Raten des Ausführungzweiges (**Speculative Execution**):
  - **Eager Execution -** Berechnung beider Zweige und Verwerfen des überflüssigen Ergebnisses
  - **Predictive Execution -** Auswahl des Zweiges anhand früherer Entscheidungen
  - **Lazy Execution -** via Out-of-Order Execution: Verschiebung der Ausführung der Zweige bis zur Auswertung der Conditions und Ausführung anderer Instruktionen



Parallelität auf Funktionseben

Parallelität auf Funktionsebene

- Parallelisierung von untereinander unabhängigen **Funktionen** oder **Funktionsblöcke**
- Notwendigkeit der Identifizierung von unabhängigen Funktionsblöcken
- Wichtig: ggf. benötigte Reihenfolgen der Abarbeitung muss vorher bekannt sein
  - ➢ Umsetzung durch später vorgestellte Synchronisierungsverfahren

Beispiel Parallelität auf Funktionsebene

```
int[] data = new int[100];
getStochasticData(data);
estimation = calculateEstimator(data);
median = calculateMedian(data);
```

Operation unabhängig
=> können parallelisiert werden

```
variance = calculateVar(data,estimation);
```

Ausführung erst nach Berechnung des Schätzwertes

```
print(estimation, variance, median);
```

Beispiel – Parallelisierung Bubble Sort

- Anwendungsbeispiel: Bubblesort
- Ziel: Sortierung einer Liste hinsichtlich eines Vergleichskriteriums
- beim Durchlaufen der Liste
  - Iteration über die komplette Liste
  - Vergleich aller benachbarten Elemente
  - Vertauschen der Elemente, bei Fehlschlag des Sortierkriteriums
  - n-maliges Durchlaufen der Liste (n = Länge der Liste)
- Komplexität $O(n^2)$

Pseudocode - Bubble Sort

$O(n)$

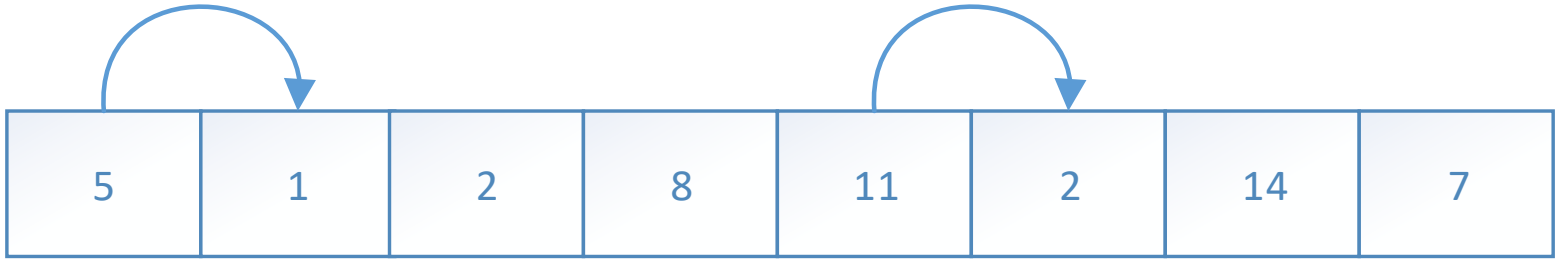
$O(n)$

```
for i=0 to n
  for j=0 to n-1
    if inputList[j]>inputList[j+1]
      swap(inputList[j], inputList[j+1])
    EndIf
  EndFor
EndFor
```

Java - Bubble Sort

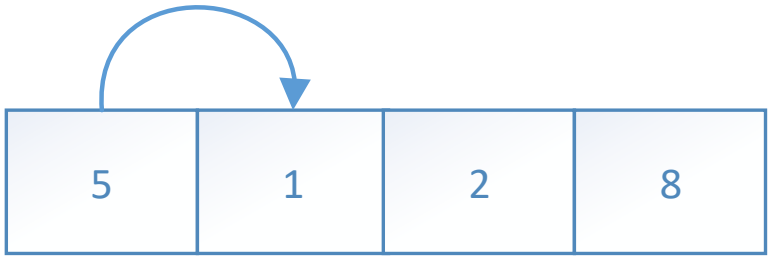
```
for(int i=0; i<n; i++){
  for(int j=0; j<n-1; j++){
    if(inputList[j]>inputList[j+1]){
      //swap neighbour items
      int temp = inputList[j];
      inputList[j] = inputList[j+1];
      inputList[j+1] = temp;
    }
  }
}
```

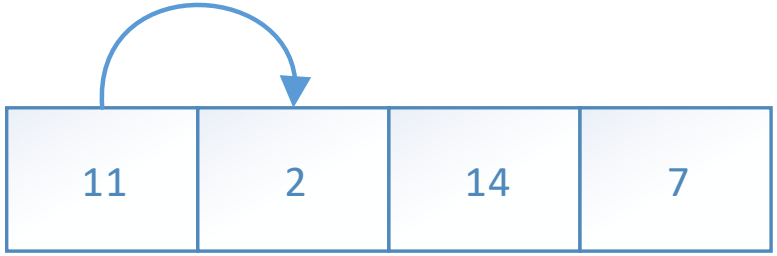
Parallelisierung von Bubble Sort I



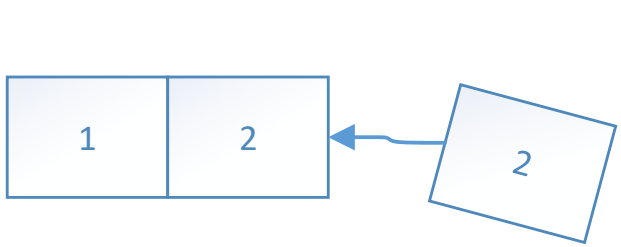
Parallelisierung von Bubble Sort II

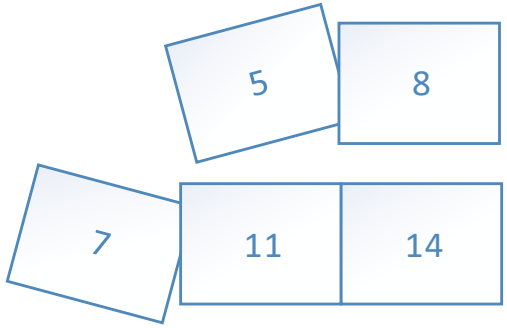
- **Divide and Conquer** -> Datenparallelität





- Sortierung auf zwei Teillisten





- dann Sortierung der beiden Teillisten

Parallelisierung von Bubble Sort III

$$O(\frac{n^2}{4})$$

```
BubbleSortParallel(inputList)
  inputLen <- inputList.length
  list1 <- Bubble(inputList.sublist(0, inputLen/2))

  list2 <- Bubble(inputList.sublist(inputLen /2 +1,
                    inputLen)

  sortedList <- merge(list1, list2)
  return sortedList
```

$$O(n)$$

```
Bubble(List)
  for i=0 to n/2
    for j=0 to n/2
      if inputList[j]>inputList[j+1]
        swap(inputList[j], inputList[j+1])
```

Ab wann Geschwindigkeitsgewinn?

$$n^2 > \frac{n^2}{4} + n$$

$$0 > \left(\frac{-3n}{4} + 1\right) n$$

Für alle Listen mit Länge $n \geq 2$ sinkt der Aufwand.

Theorie: Speedup und Amdahls Gesetz

- Speedup eines Programmes gegeben durch:

$$S(n) = \frac{T_1}{T_n} = \frac{Ausführungszeit\ 1\ Prozessor}{Ausführungszeit\ n\ Prozessoren}$$

- Amdahls Gesetz zur Berechnung der Beschleunigung

$$S(p) = \frac{1}{1 - X + \frac{X}{p}}$$

X: parallelisierbarer Programmteil

p: Anzahl der Prozessoren

- Speedup mit Amdahls Law

$$S(p) = \frac{T_1}{T_1 \cdot (1 - X + \frac{X}{p})}$$

Grenzen des Speedup

- Amdahls Gesetz

$$S(p)_{max} \leq \frac{1}{1 - X + \frac{X}{p}}$$

- maximal möglicher Speedup für unbegrenzte Prozessoranzahl

$$\lim_{p \to \infty} \left(\frac{1}{1 - X + \frac{X}{p}} \right) = \frac{1}{1 - X} \geq S(p)_{max}$$

Effizienz und Parallelisierungs-Overhead

- Effizienz E(p) – Güte der Ausnutzung zusätzlicher Prozesse (geringe Wartezeiten etc.)

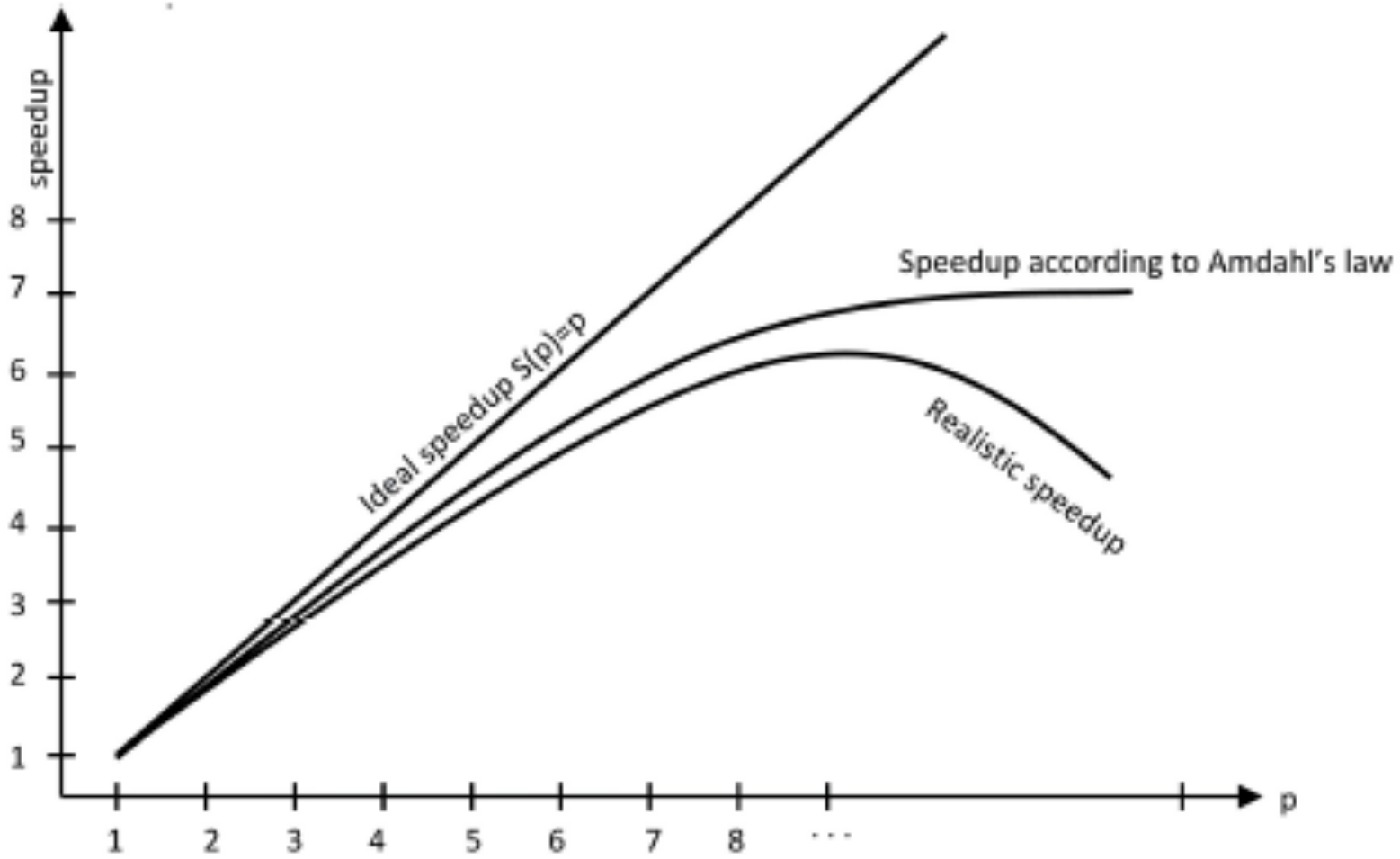
$$\mathrm{E}(p) = \frac{S(p)}{p}$$

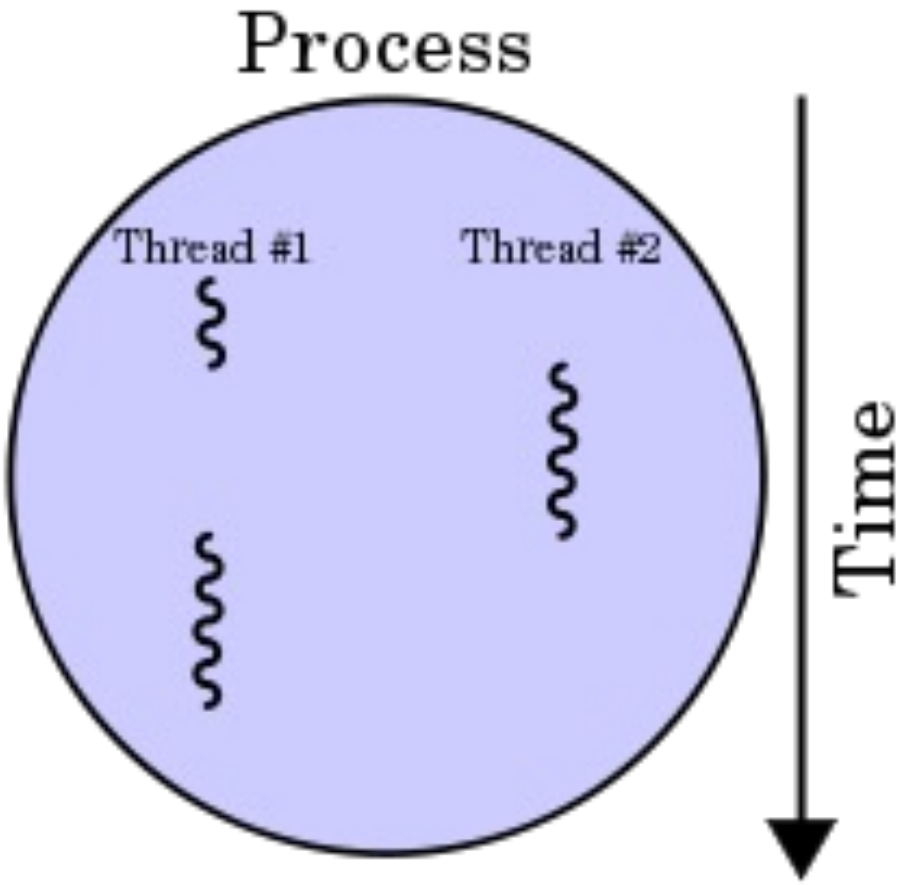
- es gilt $0 \leq E(p) \leq 1$ mit Optimum $S(p) = p \rightarrow E(p) = 1$
- aber **Parallelisierungs-Overhead**:
  - durch zusätzlichen Verwaltungsaufwand der Prozesse
  - ggf. Wartezeiten bei nicht optimaler Lastverteilung

$$T_{overhead} = p \cdot T_{parallel} - T_{sequenziell}$$

Realistischer Speedup

- große Anzahl an Prozessoren $T_{overhead} > T_1 - T_p$





Prozesse & Threads

Prozesse I

- Anforderung der Multitaskingfähigkeit an Systeme
  - Bsp.: Während des Herunterladens einer Datei, soll nicht die komplette Anwendung blockieren
  - Andere Funktionen, z.B. das Öffnen einer Datei sollen trotzdem möglich sein

- Auslagerung von einzelnen, voneinander unabhängigen (concurrent) Anwendungen in separate **Prozesse**

Prozesse II

- Prozess - einzeln ausführbare Einheiten
  - i.d.R. jedes gestartete Programm ein einzelner Prozess
- umfassen neben dem Programm auch alle für die Ausführung notwendigen Informationen:
  - Runtime-Stack bzw. Heap
  - aktuelle Registerinhalte des Prozessors
  - Programmcounter (nächste auszuführende Instruktion)
- Jeweils Zuweisung der erforderlichen Informationen bei Erstellung eines Prozesses (aufwendig)

Zuweisung von Prozessen (Scheduling)

- begrenzte Ressourcen (CPUs/APUs), müssen diese auf Prozesse verteilt werden
  - meistens mehr Prozesse als Recheneinheiten
  - begrenzte Rechenzeit auf CPU für einzelne Prozesse
  - Abwechslung zwischen Prozessen
- Zuweisung der Rechenzeiten durch Scheduler
  - Entscheidung, wann welcher Prozess Rechenzeit bekommt
  - Entscheidung, wie viel Rechenzeit jeder Prozess bekommt
  - bei Prozesswechsel, Sicherung aller Daten (Registerinhalte, PC etc.) von CPU (Dispatching)
- Anwendung des Multitasking-Prinzips sowohl für Einprozessor- als auch für Mehrprozessor-Systeme

Threads

- Erzeugung von Prozessen aufwendig - Zeit für Zuweisung Adressraum und notwendiger Daten

- Einführung des weniger aufwendigen **Thread-Modells:**

  - Threads - separate (atomare) Kontrollflüsse
  - leichtgewichtiger als Prozesse – weniger Erzeugungsaufwand
  - mehrere Threads, d.h. separate Kontrollflüsse, in einem Prozess,
  - Teilung des Adressraums und somit der Daten von Threads innerhalb eines Prozesses

Java Thread

- Möglichkeit in Java zur Erzeugung von Threads auf Sprachebene
- Erzeugte Threads automatisch Kind-Threads des Threads, in dem sie erzeugt wurden
- Haupt-Thread der JVM ist immer der Thread mit der main()-Funktion
- um Objekt als Thread zu markieren, muss entweder:
  - die **Klasse Thread** erweitert oder
  - das **Interface Runnable** implementiert werden
  - in beiden Fällen enthält **Funktion run()** das Thread-Verhalten

Erweiterung der Klasse Thread I

- Erweitern der Klasse **`Thread`**
- Implementierung der **`run()`** Methode
- Erzeugung des Objektes der Thread-Klasse
- mittels **`start()`** Thread ausführen
  - Thread-Klasse besitzt `start()` Methode die die jeweilige `run()` Methode ausführt
  - startet `BubbleParThread.run()` in neuem Thread

Erweiterung der Klasse Thread II

```
public class BubbleParThread extends Thread {
  int[] inputList;
  int index, step;
  public BubbleParThread(int[] inputList,
    int index, int step){
    … //set globals
  }

  public void run(){
    int n = inputList.length;
    for(int i=index;i < n-step; i+=step){
      if (inputList[i] > inputList[i+step]){
       … //swap neighbour items
      } // ende if
    } // ende for
  }
}

public class BubbleSeq {
  public static void main(String[] args){
    int[] inputList = getRVals(100,100000);
    BubbleParThread bubbleTh = new
        BubbleParThread(inputList, 0, 2);

    bubbleTh.start();
  }
}
```

Implementierung des Interface Runnable I

- Implementierung des Interface **`Runnable`**
- Implementierung der **`run()`** Methode
- Erzeugung Objekt `BubbleParThread`

- Erzeugung Thread-Objekt
  - Übergeben der Referenz des auszuführenden Threads (`bubbleTh`) bei Erzeugung

- Ausführung mittels **`start()`** Hilfsthread
  - Anlegen eines neuen Threads
  - Ausführung von `run()` des übergebenen Threads

Implementierung des Interface Runnable II

```
public class BubbleParThread implements Runnable {
  int[] inputList;
  int index, step;
  public BubbleParThread(int[] inputList,
    int index, int step) {
    … //set globals
  }
  public void run() {
    int n = inputList.length;
    for(int i=index;i<n-step;i+=step) {
      if (inputList[i]>inputList[i+step]) {
       … //swap neighbour items
      } // ende if
    } // ende for}}

public class BubbleSeq{
  public static void main(String[] args){
    int[] inputList = getRVals(100,100000);
    BubbleParThread bubbleTh = new
        BubbleParThread(inputList,0,2);

    Thread thread = new Thread(bubbleTh);

    thread.start();
  }
}
```

weitere nützliche Threadfunktionen

- in Java existieren weitere nützliche Threadfunktionen:

- `Thread.join();`
  - ➢ wartet auf die Beendigung des aufrufenden Threads

- `Thread.sleep(long time);`
  - ➢ Thread wird die angegebene Zeit nicht ausgeführt

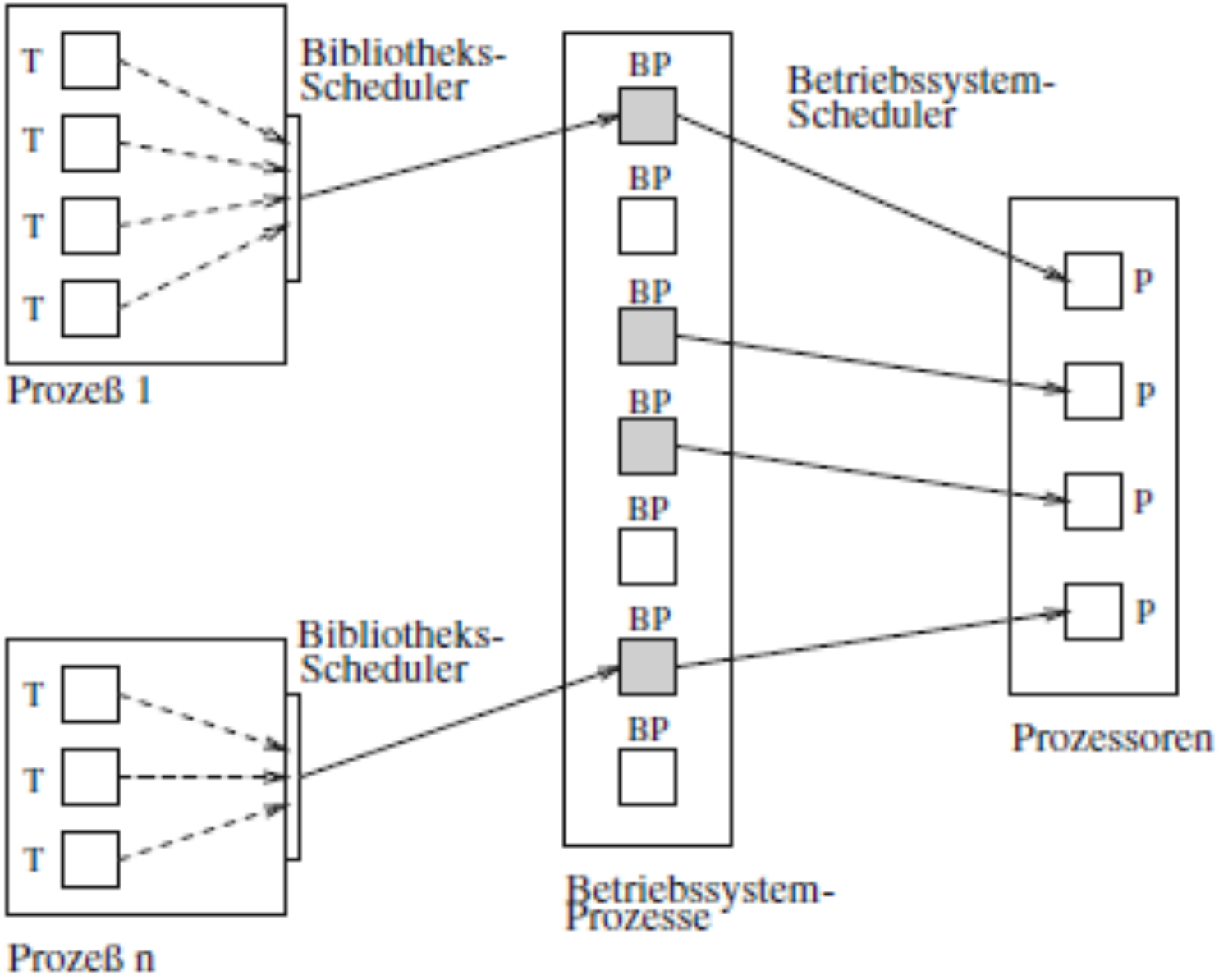
- `Thread.yield()`
  - ➢ Hinweis an Scheduler, dass Prozess mit gleicher Priorität vorgezogen werden kann

Thread Ausführung

- leichtgewichtigere Erzeugung, da
  - nur Zuweisung von PC und Registerspeicher nötig
  - keine Neuzuweisung zu einem Adressraum nötig

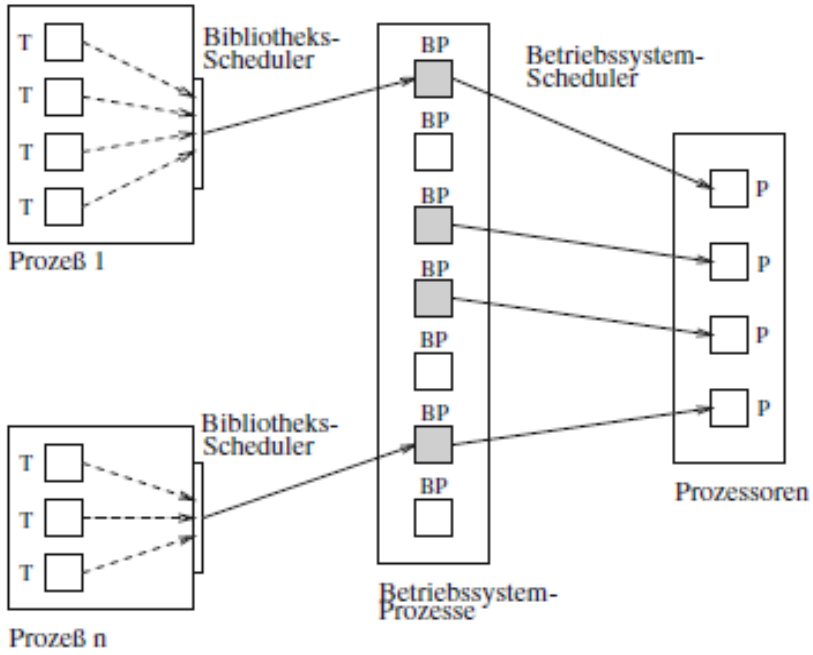
- Zuordnung von Threads zu unterschiedlichen Betriebssystemprozessen (Abarbeitung auf der CPU)
  - Möglichkeit der Ausführung unterschiedlicher Threads eines Prozesses auf mehreren Prozessoren bei Multicore-Rechnern
  - unterschiedliche Zuteilungsmodelle (Thread:OS-Prozess) möglich

N:1-Abbildung

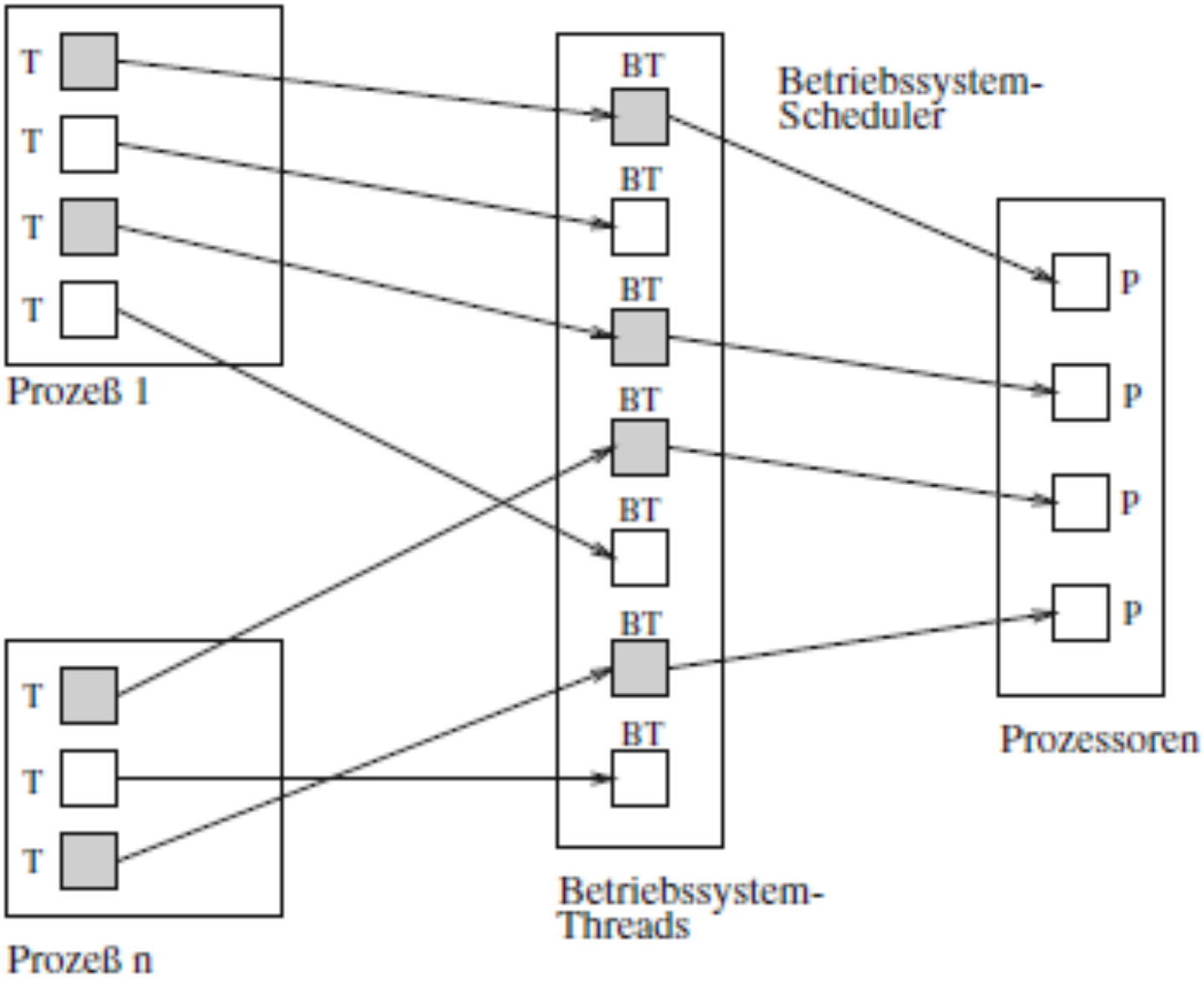


N:1-Abbildung

- eindeutige Abbildung zwischen Anwendungs-und BS-Prozess
- Zuweisung des Threads an BS-Prozess durch Bibliotheks-Scheduler
- Ausführung mehrerer Prozesse gleichzeitig möglich
- keine parallele Abarbeitung mehrerer Threads eines Prozesses

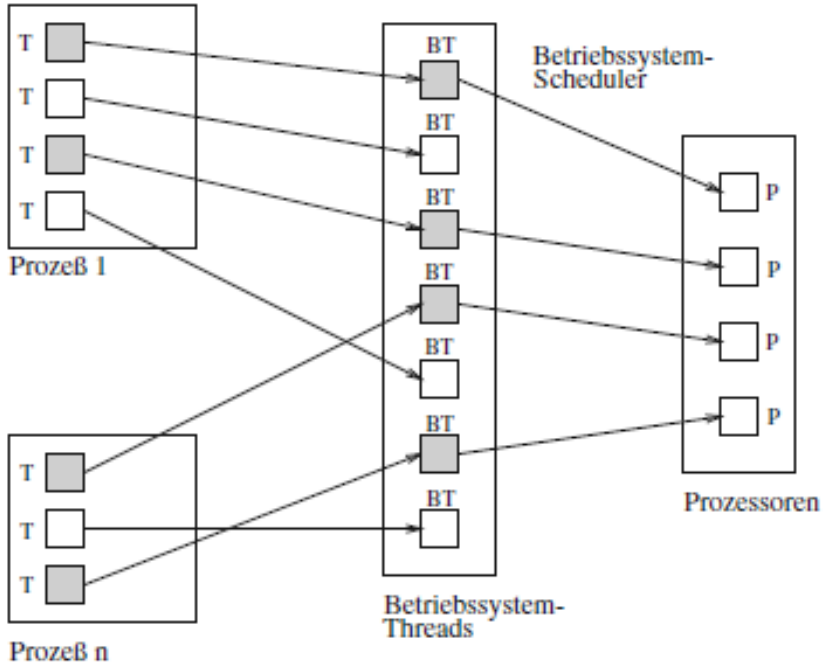


1:1-Abbildung

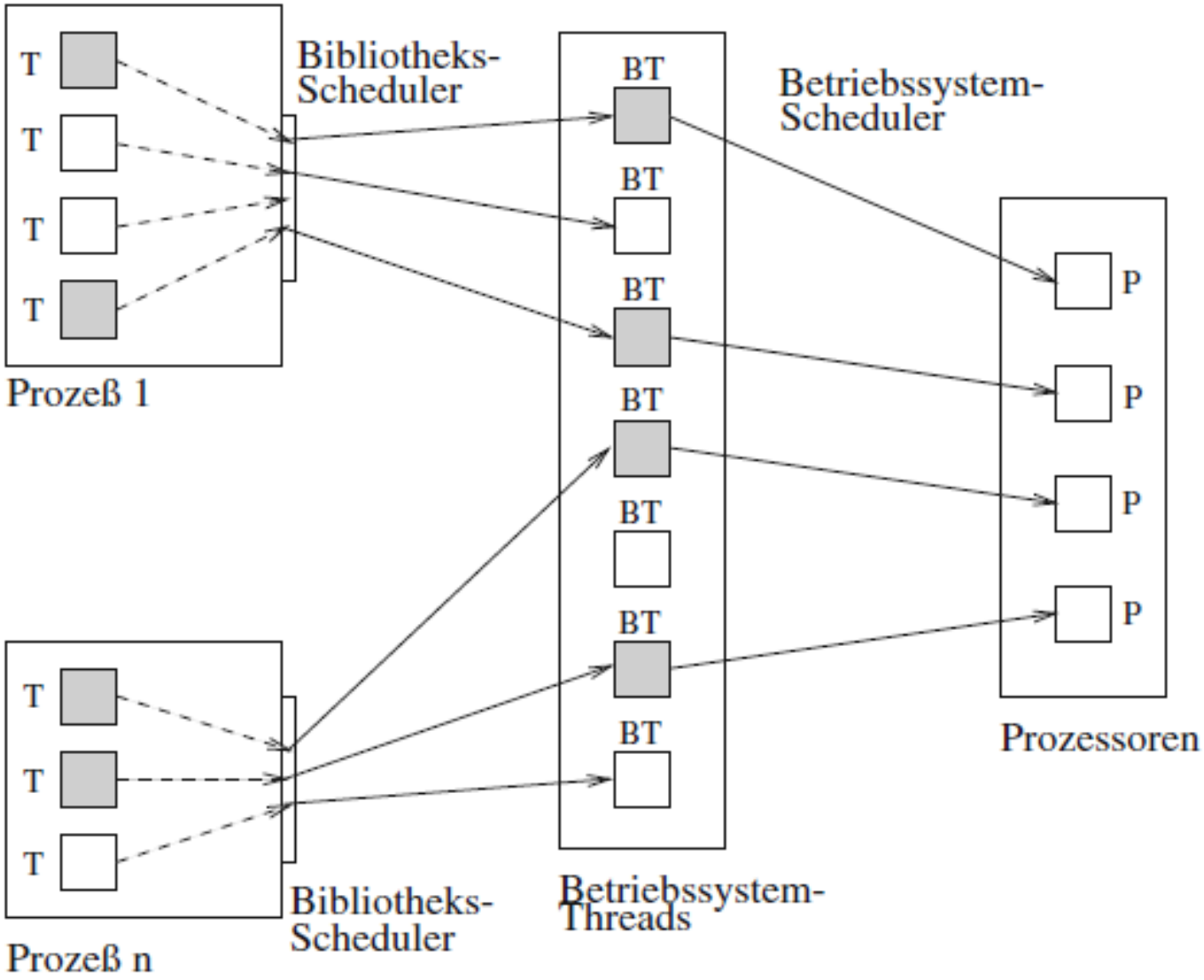


1:1-Abbildung

- exakte Abbildung jedes Threads auf einen Betriebssystem-Thread
  - ➢ nicht auf BS-Prozess!
  - ➢ kein Bibliotheks-Scheduler nötig
- Abarbeitung und Zuweisung der Threads durch BS
- parallele Abarbeitung mehrerer Threads eines Prozesses möglich

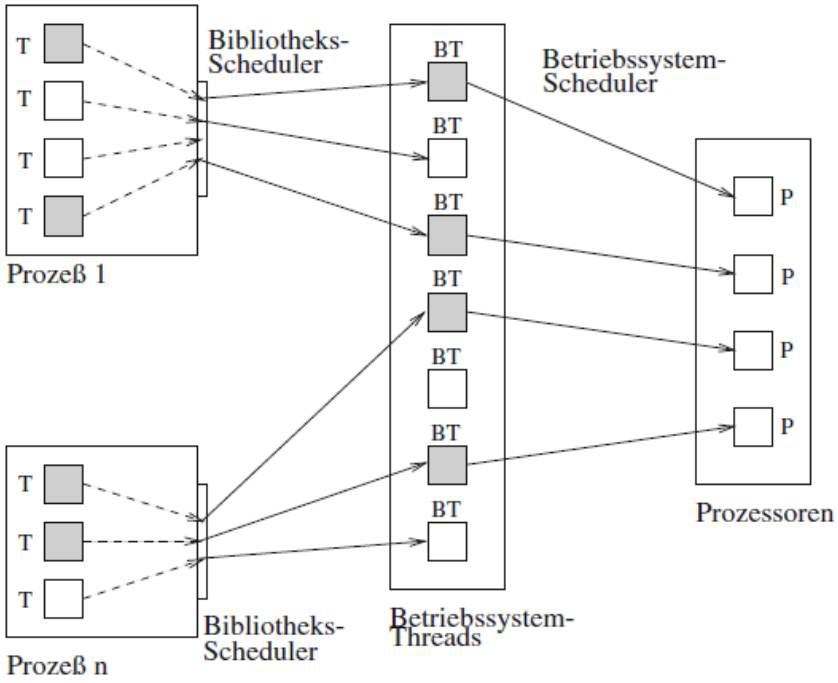


N:M-Abbildung



N:M-Abbildung

- Verteilung von mehreren Threads auf BS-Threads via Bibliotheks-Scheduler
- Beeinflussung des Bibliotheks-Schedulers durch Programmierer
  - ➢ Vorgabe der Abarbeitungsreihenfolge bzw. Prioritäten der Threads
- Erhöhung der Effizienz durch variable Anzahl an BS-Threads



Prozess vs. Thread

Prozess

- Objekte des OS zur Ausführung von Programmen
- Instanz eines Computerprogramms
- eigener Adressraum
- besitzt 1 bis n Threads

Thread

- Codestücke eines Programms
- in meisten Systemen kleinste verwaltbare Einheit
- besitzt Register, Stack und Program-Counter
- Nutzung des Adressraum des Elternprozesses mit anderen Threads