

Programmierparadigmen

Kapitel 4a Einführung in parallele Programmierung

(Diese Folien beruhen auf einem Foliensatz von Prof. Dr. Kai Uwe Sattler)



Überblick

- Grundlagen
 - Motivation
 - Architekturen
 - Grundbegriffe
 - Programmiermodelle
- Parallele Programmierung in Erlang
- Parallele Programmierung in C++
- Parallele Programmierung in Java



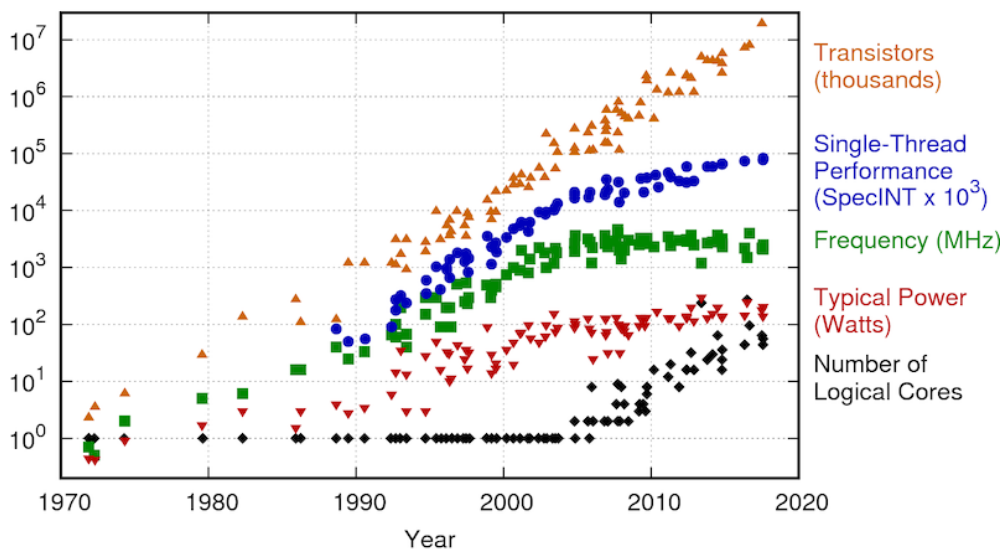
- Programmierung paralleler Algorithmen und Verfahren als Paradigma
- Verständnis grundlegender Architekturen und Modelle
- Praktische Erfahrungen mit Erlang, C++ und Java



Motivation: The free lunch is over 1

- Herb Sutter: Dr. Dobbs' Journal 30(3), 2005.

42 Years of Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

<https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data>



Motivation: The free lunch is over 2

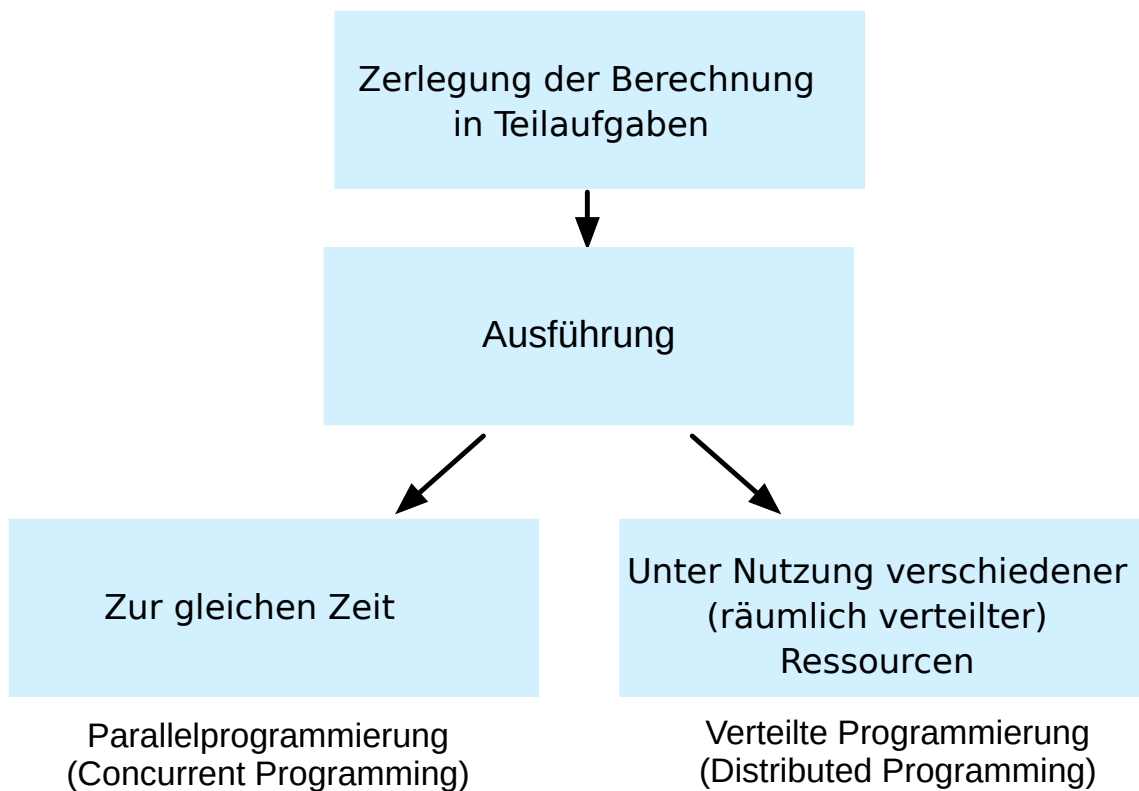
- Grundproblem: Die Taktfrequenz wächst nur noch langsam
 - Physikalische Gründe: Wärmeentwicklung, Energiebedarf, Kriechströme, Signallaufzeiten (~ Distanzen)...
→ 10 Ghz-CPU realisierbar/sinnvoll???
- Mögliche Auswege:
 - Hyperthreading:
 - Abarbeitung mehrerer Threads auf einer CPU (5..15% Performanzgewinn)
 - Einfache Hardwareunterstützung (einige Register)
 - Multicore:
 - Mehrere CPUs auf einem Chip
 - Billiger als echte Mehrprozessorsysteme
 - Caching:
 - Vergrößerung L1, L2, L3-Cache
 - Speicherzugriff 10..50 × teurer als Cache

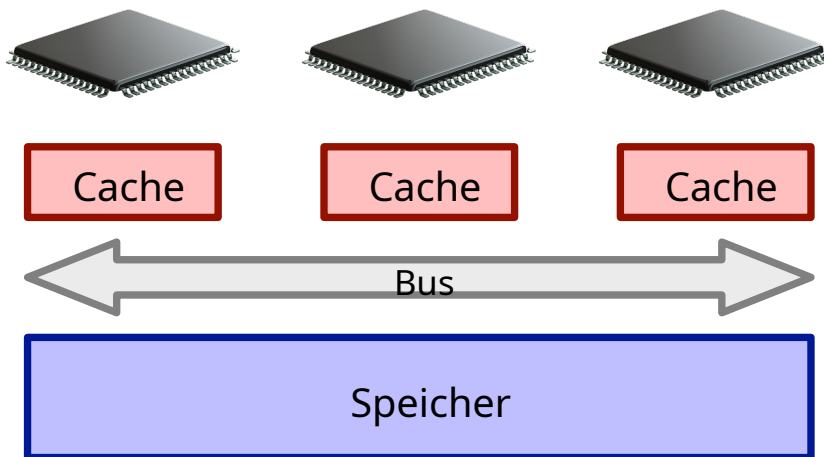


Konsequenzen und Trends

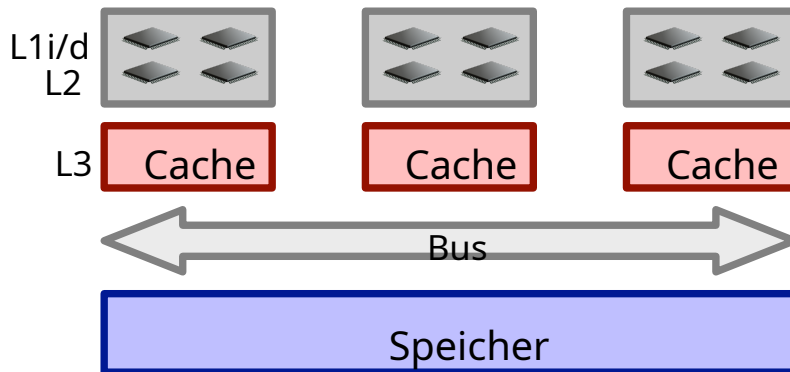
- Applikationen müssen nebenläufig programmiert werden, um Prozessor (CPU) auszunutzen → Many-Core-Systeme
- CPU-Begrenzung von Applikationen
- Effizienz und Performanzoptimierung werden immer wichtiger
- Unterstützung von Nebenläufigkeit/Parallelität durch Programmiersprachen







- Zugriff über Bus auf gemeinsamen Speicher
- Jeder Prozessor verfügt über eigenen Cache



- Mehrere Prozessorkerne auf einem Chip
- Kerne verfügen meist über jeweils eigene L1/L2-Caches und einen gemeinsamen L3-Cache
 - Layer-1-Cache ist am schnellsten, aber dafür ziemlich klein (damit alle Speicherstellen physikalisch nah an der Recheneinheit sind)
 - Layer-2-Cache ist weniger schnell, dafür etwas größer
 - Layer-3-Cache ist noch größer und noch langsamer

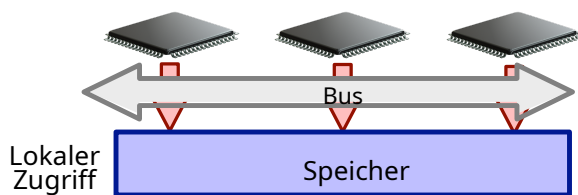


- Zur Erreichung einer hohen Energieeffizienz (insbesondere in mobilen Geräten wie Handys und Laptops) wird mittlerweile oft noch zwischen zwei Arten von CPU-Cores unterschieden:
 - Performance Cores:
 - Höhere maximale Taktrate
 - Hyperthreading
 - Bei Intel: AVX 512 (siehe später; nicht bei allen Intel CPUs)
 - Efficiency Cores:
 - Niedrigere maximale Taktrate
 - Kein Hyperthreading; kein AVX 512
 - Geringerer Energieverbrauch & kleinere Chipfläche
 - Zum Teil teilen sich mehrere E-Cores gemeinsame L2-Caches:
 - Bei Intel 12 Gen Core Desktop CPUs teilen sich je 4 E-Cores einen 2 MB L2-Cache während jeder P-Core auf eigenen 1,25 MB L2-Cache zugreifen kann



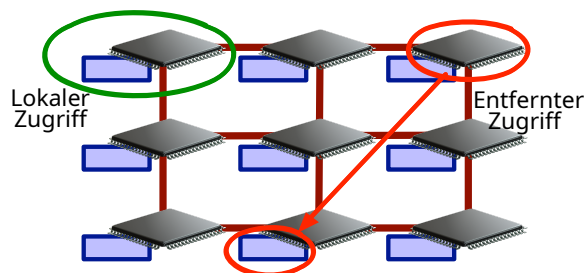
SMP:

Symmetric Multi Processing



NUMA:

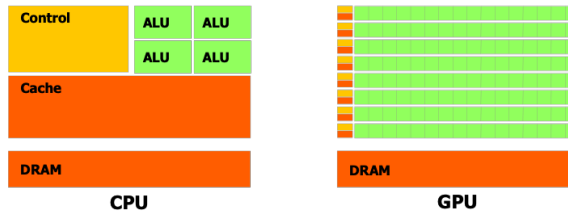
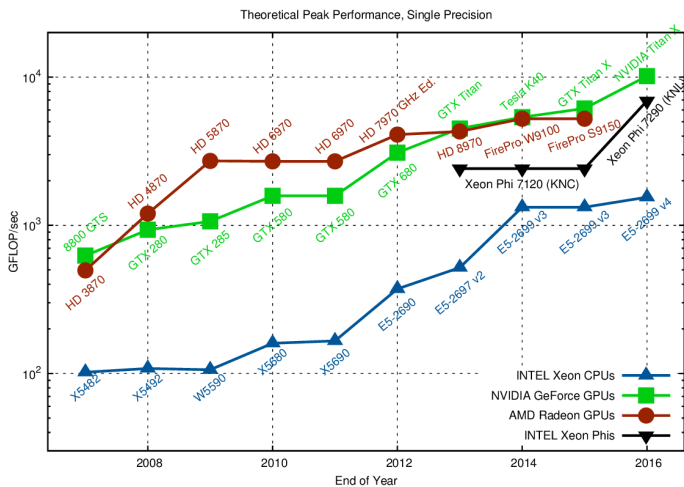
Non-Uniform Memory Access



- | | |
|---|--|
| <ul style="list-style-type: none"> ■ Speicherbandbreite ist begrenzt und wird von allen CPUs gemeinsam genutzt ■ Skalierbarkeit begrenzt ■ Single (Memory-)Socket-Lösung | <ul style="list-style-type: none"> ■ Jedem Prozessor sind Teile des Speichers zugeordnet ■ „Lokaler“ Speicherzugriff ist schneller als auf entfernte Bereiche ■ Mehr-Socket-Board |
|---|--|



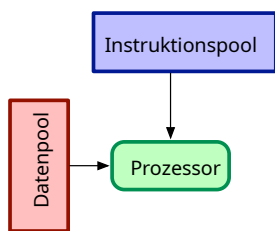
- GPU = Graphics Processing Unit
- Hochparallele Prozessorarchitekturen (nicht nur) für Grafik-Rendering



<https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/>

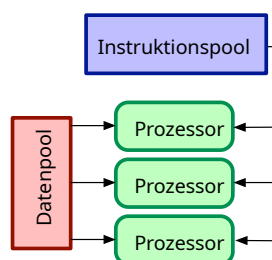


- SISD: Von Neumann



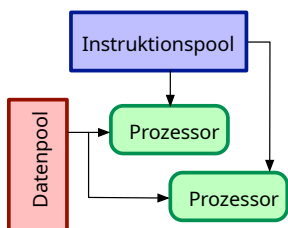
Single Instruction Single Data

- SIMD: Vektorprozessor



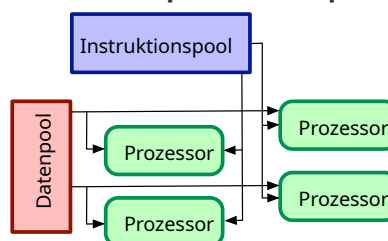
Single Instruction Multiple Data

- MISD: Fehlertoleranz



Multiple Instruction Single Data

- MIMD: Supercomputer



Multiple Instruction Multiple Data



- Wie bewertet man den Laufzeitgewinn durch Parallelisierung?
- T_n = Laufzeit des Programms mit n Prozessoren/Kernen
- Speedup:**

$$\text{Speedup} = \frac{T_1}{T_n}$$

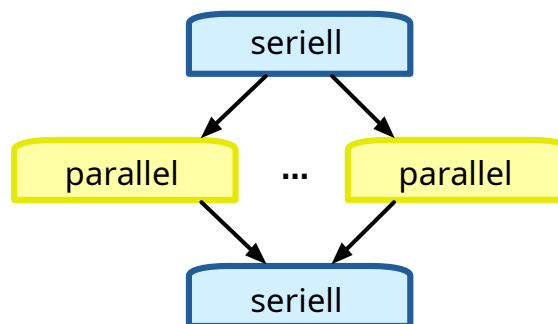
- Effizienz:**

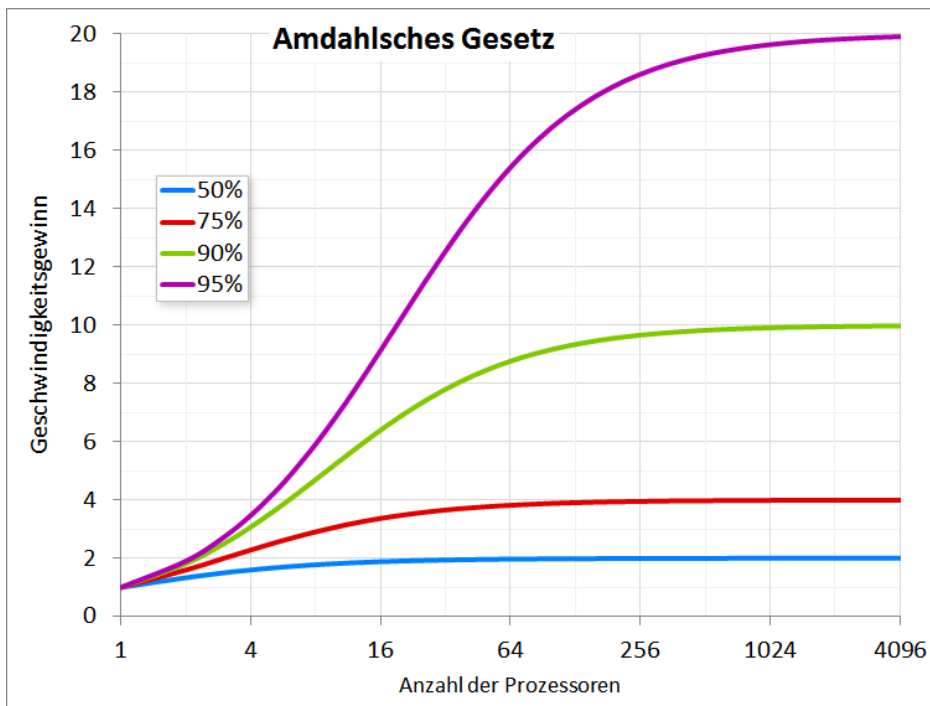
$$\text{Effizienz} = \frac{\text{Speedup}}{n}$$



- Berücksichtigung parallelisierbarer und serieller Anteile im Programmablauf
- n Anzahl Prozessoren
 - p paralleler Anteil
 - s serieller Anteil
 - $p + s = 1$ (Normierung für Vergleichbarkeit)
- Maximaler Speedup:

$$\begin{aligned} \text{Speedup}_{max} &= \frac{1}{T_n} \\ &= \frac{s + p}{s + \frac{p}{n}} \\ &= \frac{1}{s + \frac{p}{n}} \end{aligned}$$





CC BY-SA 4.0 Frank Klemm



- **Prozess** := Programm in Ausführung;
Ausführungsumgebung für ein Programm
 - Hat eigenen Adressraum
 - Ein Prozessor(kern) kann immer nur einen Prozess zu einer Zeit ausführen
- **Thread** := Leichtgewichtige Ausführungseinheit oder Kontrollfluss (= Folge von Anweisungen) innerhalb eines sich in Ausführung befindlichen Programms
 - „Leichtgewichtig“ im Vergleich zu Betriebssystemprozess
 - Threads eines Prozesses teilen sich einen Adressraum
 - Thread kann von CPU bzw. Core einer CPU ausgeführt werden
 - Wörtlich übersetzt: Thread = „Faden“



- Art der Kommunikation zwischen Prozessen bzw. Threads
- **Shared Memory:**
 - Kommunikation über (Variablen im) gemeinsamen Speicher
 - Prozess kann direkt auf Speicher eines anderen Prozesses zugreifen
 - Erfordert explizite Synchronisation, z.B. über kritische Abschnitte
- **Message Passing:**
 - Prozesse mit getrennten Adressräumen; Zugriff nur auf eigenen Speicher
 - Kommunikation durch explizites Senden/Empfangen von Nachrichten
 - Implizite Synchronisation durch Nachrichten



Shared Memory

Produzent

```
flag = false;  
x = 42;  
flag = true;
```

Konsument

```
flag = false;  
while (!flag) /* wait */;  
y = x + 13;
```

Achtung: Ergebnis hängt von zeitlicher Reihenfolge ab!

Message Passing

Produzent p

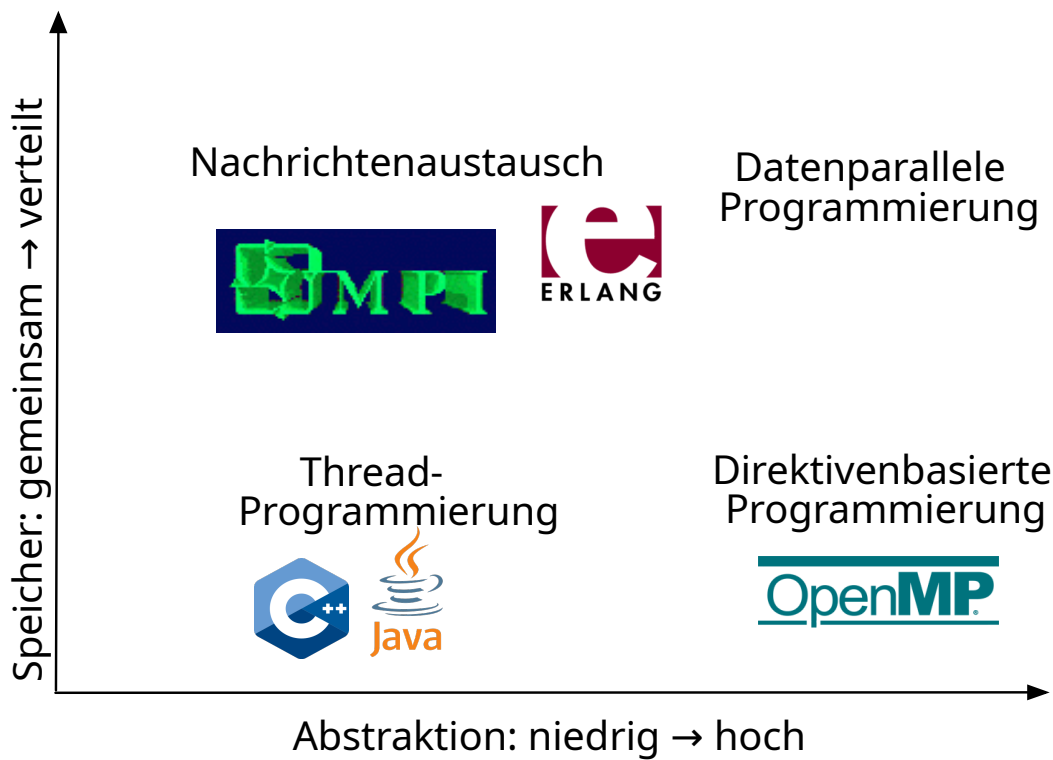
```
x = 42;  
send(k, x);
```

Konsument k

```
x = receive(p);  
y = x + 13;
```

Konsument kann erst nach Empfang mit Berechnung fortfahren.





■ Instruktionsparallelität:

- Parallele Ausführung mehrerer Operationen durch eine CPU-Instruktion (Instruktion = Anweisung, Befehl)
- Implizit: „Pipelining“ von Instruktionen (hier nicht weiter behandelt)
- Explizit („Vectorized“): Vektorinstruktionen ~ SIMD

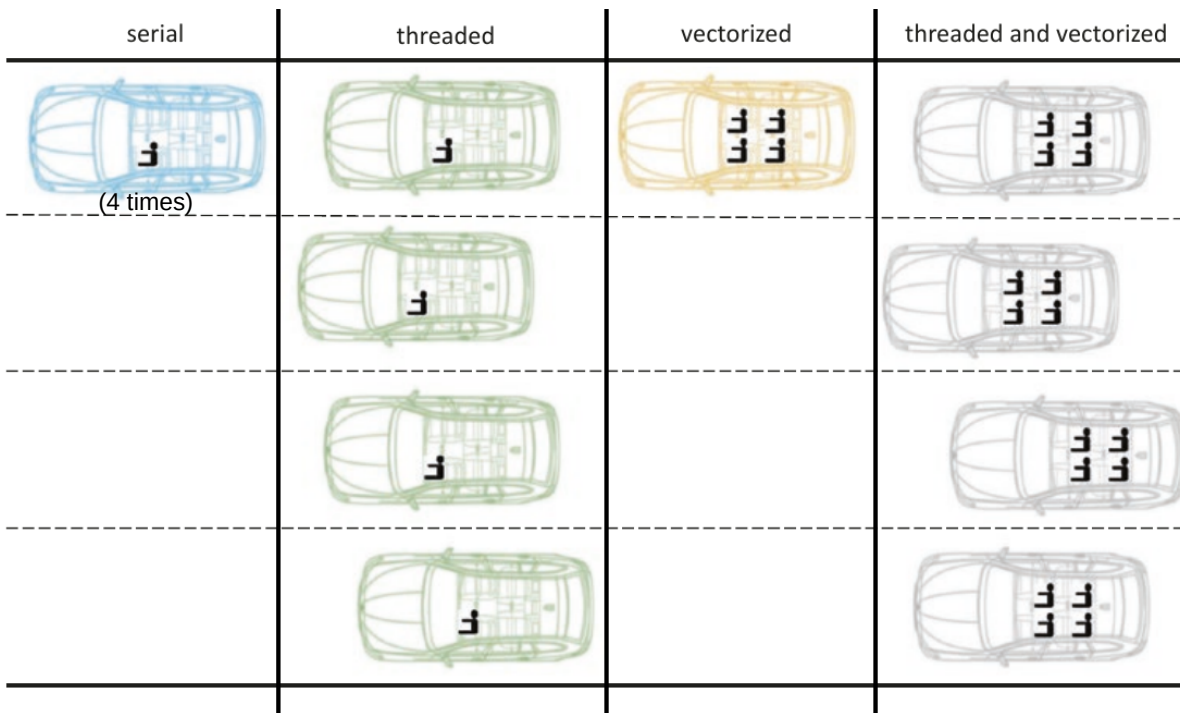
■ Datenparallelität:

- Gemeinsame Operationen auf homogener Datenmenge
- Zerlegung eines Datensatzes in kleinere Abschnitte
- Beispiele: Vektorinstruktionen, OpenMP (später)

■ Task-Parallelität („Threaded“):

- Ausnutzung inherenter Parallelität durch gleichzeitige Ausführung unabhängiger (Teil-)Aufgaben





- Quelle: M. Voss, R. Asenjo, J. Reinders. *Pro TBB – C++ Parallel Programming with Threading Building Blocks*. p 113, Apress Open, 2019. (hier leicht angepasst)



- Varianten:
 - Autovektorisierung durch Compiler (in Übung)
 - Explizite Programmierung mit Vektorinstruktionen
- Beispiel: Intel Advanced Vector Extensions (AVX)
 - Spezieller (zusätzlicher) Befehlssatz zur Beschleunigung von Anwendungen, in denen große Mengen von Daten primitiver Datentypen auf gleichartige Weise verknüpft werden müssen
 - Im Jahr 2011 eingeführt, existiert mittlerweile in unterschiedlichen Bitbreiten: 256, 512 Bit
 - Grundidee: Führe einen Befehl gleichzeitig auf mehreren gleichartigen elementaren Daten (Ganzzahl, Gleitkommazahl) aus
 - Z.B. bei AVX mit 256 Bit auf x86-64-Architektur können mit einer Operation gleichzeitig vier Gleitkommaoperationen mit doppelter Genauigkeit bzw. acht Operationen mit einfacher Genauigkeit ausgeführt werden



- Exemplarischer Ablauf am Beispiel Addition mehrerer Zahlenpaare:
 - Fülle zwei AVX-Register mit den zu verknüpfenden Daten
 - Führe Operation aus
 - Auslesen der Resultate aus Ergebnisregister (hier nicht gezeigt)

```
1 #include <immintrin.h>
2
3 __m256 first = _mm256_setr_ps(10.0, 11.0,
4                               12.0, 13.0, 14.0, 15.0,
5                               16.0, 17.0);
6 __m256 second = _mm256_setr_ps(5.1, 5.1, 5.1,
7                                 5.1, 5.1, 5.1, 5.1, 5.1);
8 __m256 result = _mm256_add_ps(first, second);
```



- Suche erstes Vorkommen eines Werts in einem Array:
 - Anstatt jede Zahl einzeln zu prüfen, sollen in einer CPU-Instruktion mehrere Zahlen des Vektors gleichzeitig überprüft werden
 - Erst Variable („Suchvektor“) definieren, die einem 256-bit SIMD-Register entspricht und mit der gesuchten Zahl gefüllt wird
 - Dann "klug" über den Vektor iterieren und in jeder Iteration mehrere Zahlen auf einmal überprüfen:
 - Jeweils mehrere Werte des Vectors in eine 256-bit SIMD-Registervariable laden (z.B. direkt aus Array mehrere Zahlen als Block)
 - Suchvektor mit den aktuell zu testenden Zahlen vergleichen
 - Vergleichsergebnis auswerten und bei Erfolg abbrechen und gesuchten Index zurückgeben
 - Dieses Beispiel wird in der Übung vertieft



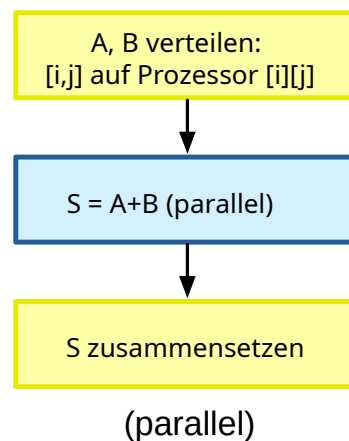
- Hohe Leistungssteigerung möglich, wenn zu lösendes Problem gut mit den zur Verfügung stehenden Befehlen parallelisiert werden kann
- Allerdings kann es je nach Problemstellung zu „Reibungsverlusten“ bei der Vorbereitung der Berechnung oder dem Auslesen des Ergebnisses kommen (siehe Übung), die einen Teil der erhofften Leistungssteigerung wieder aufzehren
- Weiterhin führt die Ausführung von AVX-Befehlen teilweise zur Reduktion der Taktrate des jeweiligen CPU Kerns:
 - Da Änderungen der Taktrate selber auch Zeit kosten, werden nachfolgende Threads teilweise kurzfristig langsamer ausgeführt
 - Daher empfehlen manche Quellen explizit die Deaktivierung von AVX 512 im BIOS, um solche Nachteile („unerwünschte Drosselung“) für andere Threads auszuschließen
- Portierung von Software zwischen AVX-fähigen und anderen Hardware-Architekturen erfordert zusätzlichen Aufwand



- Homogene Datenmenge: Felder, Listen, Dokumentenmenge, ...
- Verteilung der Daten
- Alle Prozessoren führen gleiches Programm auf jeweils eigenen Daten aus
- Beispiel: Matrix-Addition $S = A + B$

```
1 for (i=0; i < Z; i++)  
2   for (j=0; j < S; j++)  
3     S[i][j] = A[i][j]+B[i][j];
```

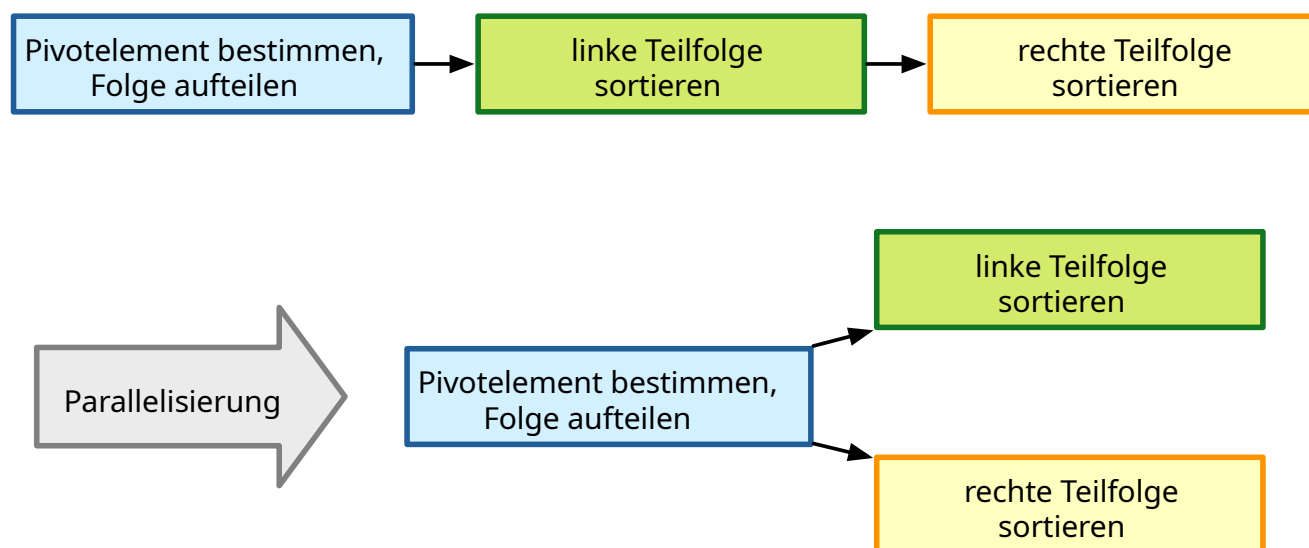
(sequentiell)



- **Zerlegung eines Problems in parallel verarbeitbare Teile**
 - Beispiel: Suche in einer Datenbank mit 1 TB Größe
 - Annahme: 100 MB/Sekunde mit einem Prozessor \approx 175 Minuten
 - Bei paralleler Suche mit 10 Prozessoren \approx 17.5 Minuten
 - Übertragbar auf andere Probleme, z.B. Sortieren, Suche in Graphen?
- **Synchronisation konkurrierender Zugriffe auf gemeinsame Ressourcen**
 - Beispiel: Produzent-Konsument-Beziehung
 - Annahme: Datenaustausch über gemeinsame Liste
 - Fragestellungen: Benachrichtigung über neues Element in Liste; Konsument entnimmt während Produzent einfügt
 - Erfordert oft **wechselseitigen Ausschluss** (Gift für Performance!)
- Außerdem: **Fehlersuche, Optimierung, ...**



- Sind Teilprozesse unabhängig voneinander, dann können sie **de-sequentialisiert** werden
- Beispiel: QuickSort



- Parallele Verarbeitung als wichtiges Paradigma moderner Software
- Vorstellung verschiedener paralleler:
 - **Hardware-Architekturen** und
 - **Programmiermodelle**
- Herausforderungen:
 - Problemzerlegung
 - Synchronisation
 - ...
- Im weiteren: konkrete Methoden und Techniken in Erlang und C++



- M. Herlihy, N. Shavit: *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012
- *Getting Started with Erlang – User’s Guide, Chapter 3: Concurrent Programming*. https://erlang.org/doc/getting_started/conc_prog.html
- J. Armstrong: *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007
- M. Voss, R. Asenjo, J. Reinders: *Pro TBB – C++ Parallel Programming with Threading Building Blocks*. Apress Open, 2019.
- A. Williams: *C++ Concurrency in Action – Practical Multithreading*. 2nd Edition, Manning, 2019

