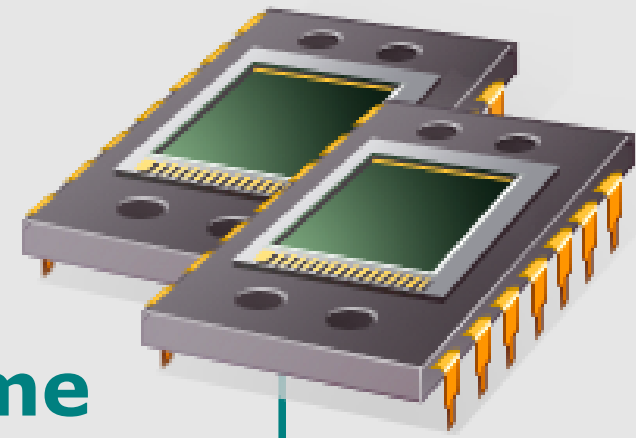


Hauptseminar Ausgewählte Kapitel der Systemsoftware: Multicore- und Manycore-Systeme



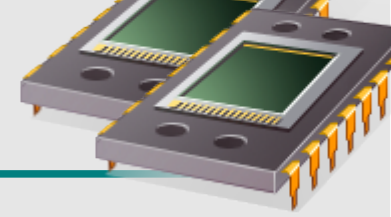
Herausforderungen von Tausendundeinem Kern

Glaubt man den Voraussagen einiger Experten, so werden innerhalb von zehn Jahren bis zu 1000 Prozessorkerne in einer CPU untergebracht sein. Dieser Vortrag wird die Herausforderungen bei der Hard- und Softwareentwicklung solcher Systeme erläutern, sowie die derzeit diskutierten Lösungsansätze vorstellen.

Vortragender: Alexander Münch
<Alexander.Muench@informatik.stud.uni-erlangen.de>

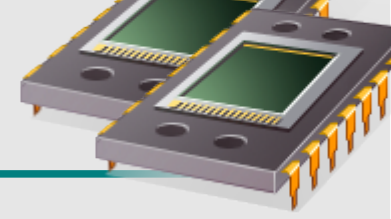
30. April 2009

Gliederung



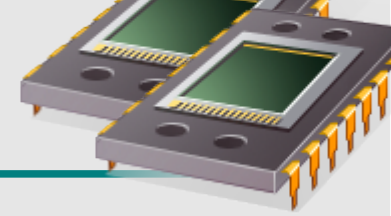
- Einführung
- Theoretischer Hintergrund
- Hardwareentwicklung
- Softwareentwicklung
 - Message Passing Interface
 - Open Multi-Processing
 - Threading Building Blocks
- Fazit
- Live-Beispiel

Warum Parallelität?

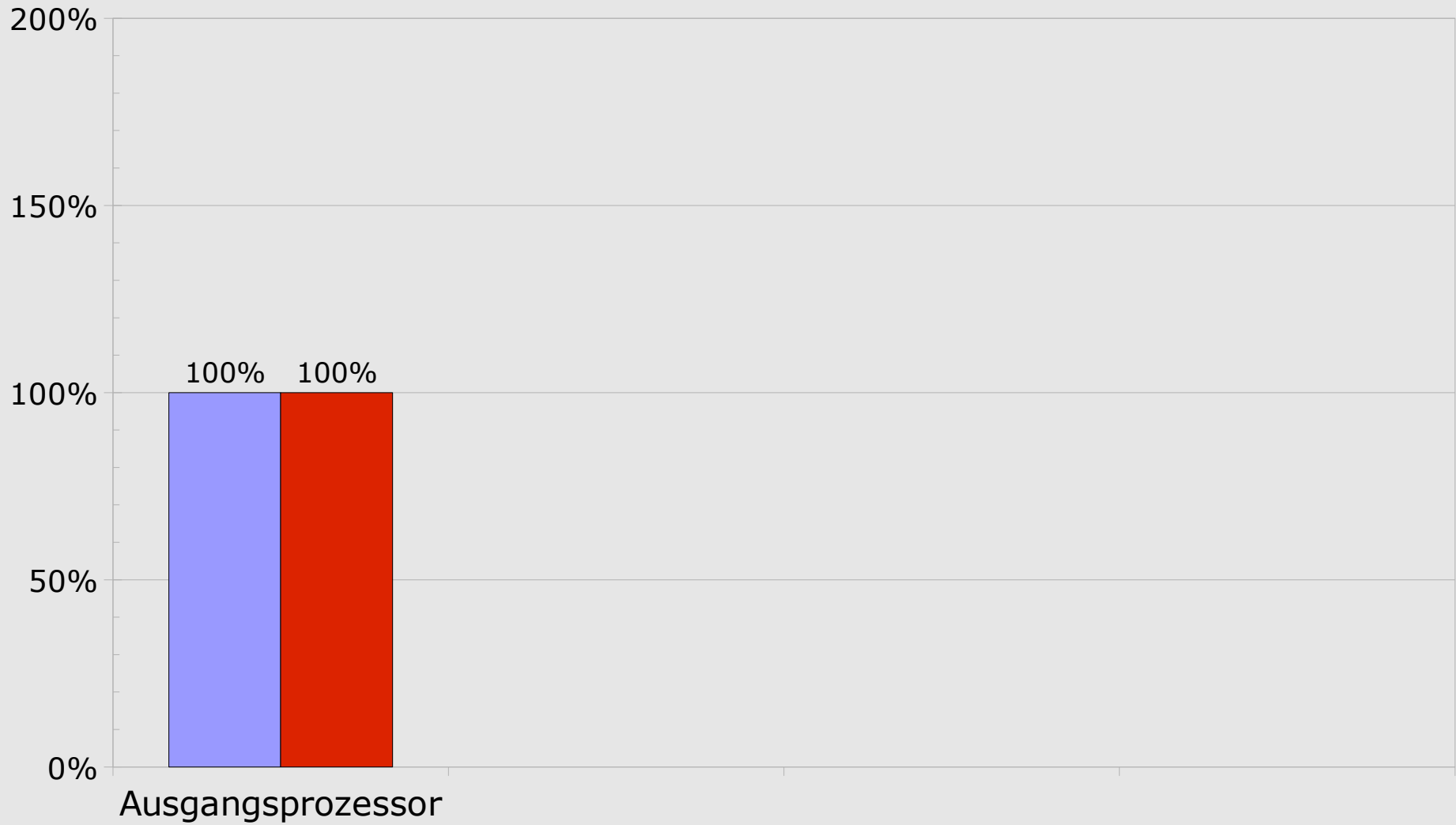


- bisher:
Performance-Steigerung durch
Taktsteigerung
- hoher Energieverbrauch, da exponentiell
gegenüber der Leistung
- weiteres Problem:
zu große Wärmeentwicklung
- besser:
Untertaktung und Zusammenschluss von
mehreren Prozessoren zu einer CPU

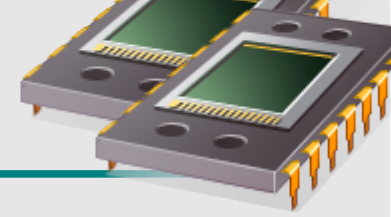
Leistung vs. Energieverbrauch



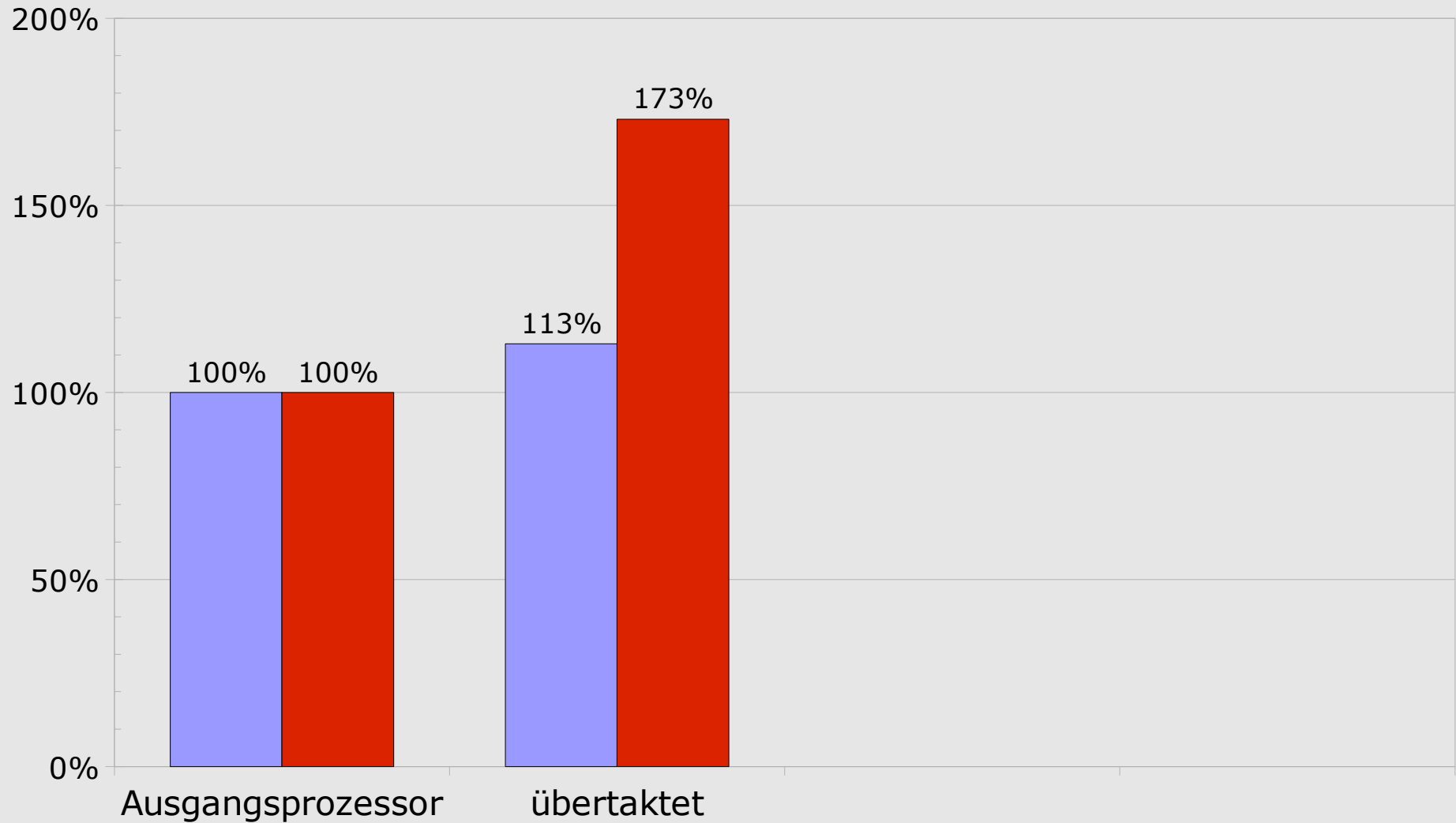
■ Leistung ■ Energieverbrauch



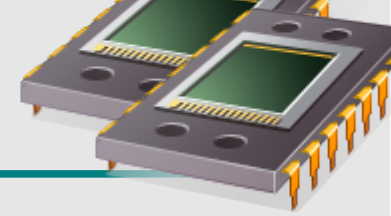
Leistung vs. Energieverbrauch



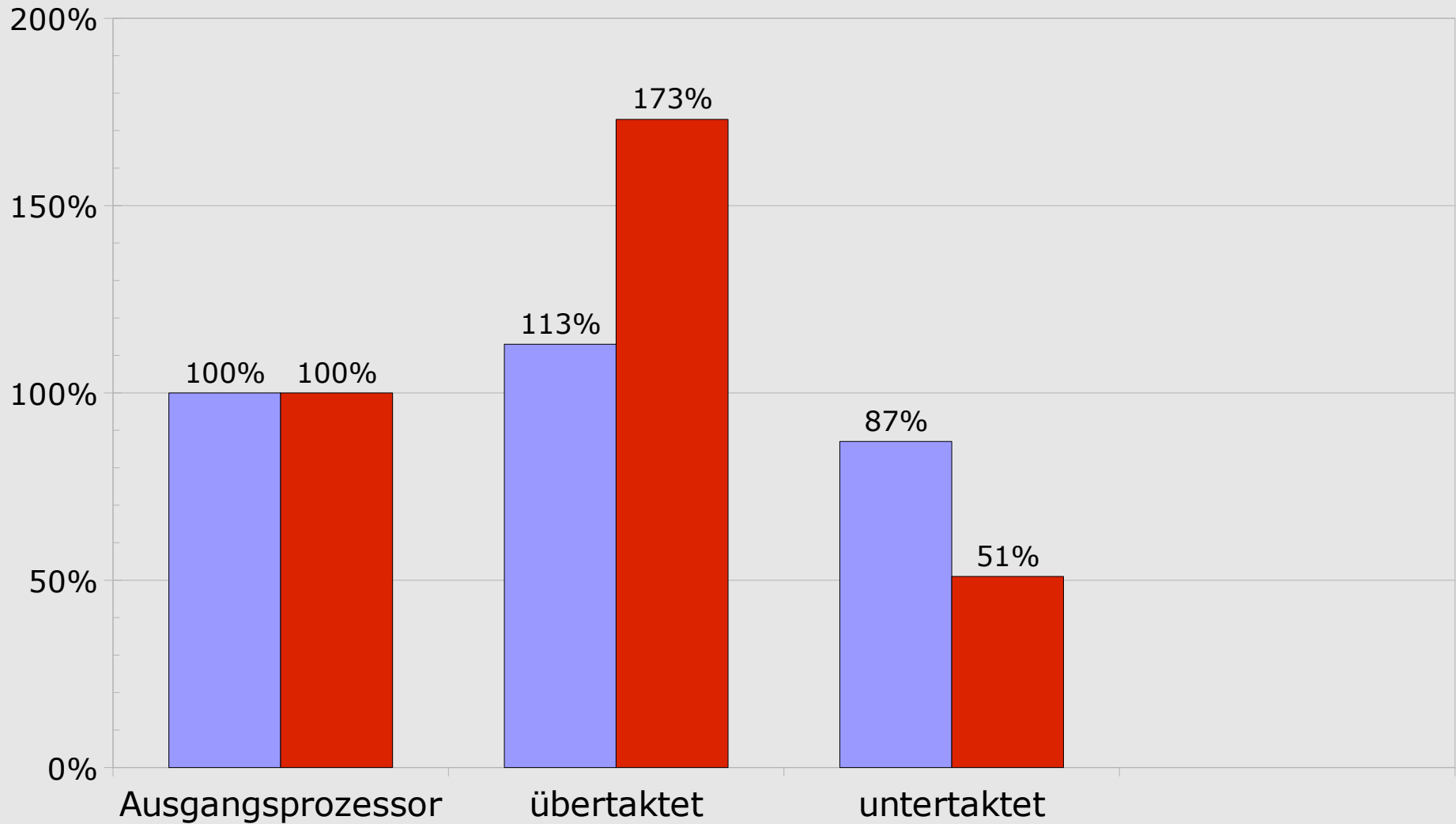
■ Leistung ■ Energieverbrauch



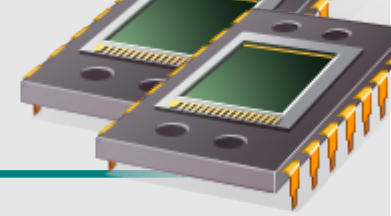
Leistung vs. Energieverbrauch



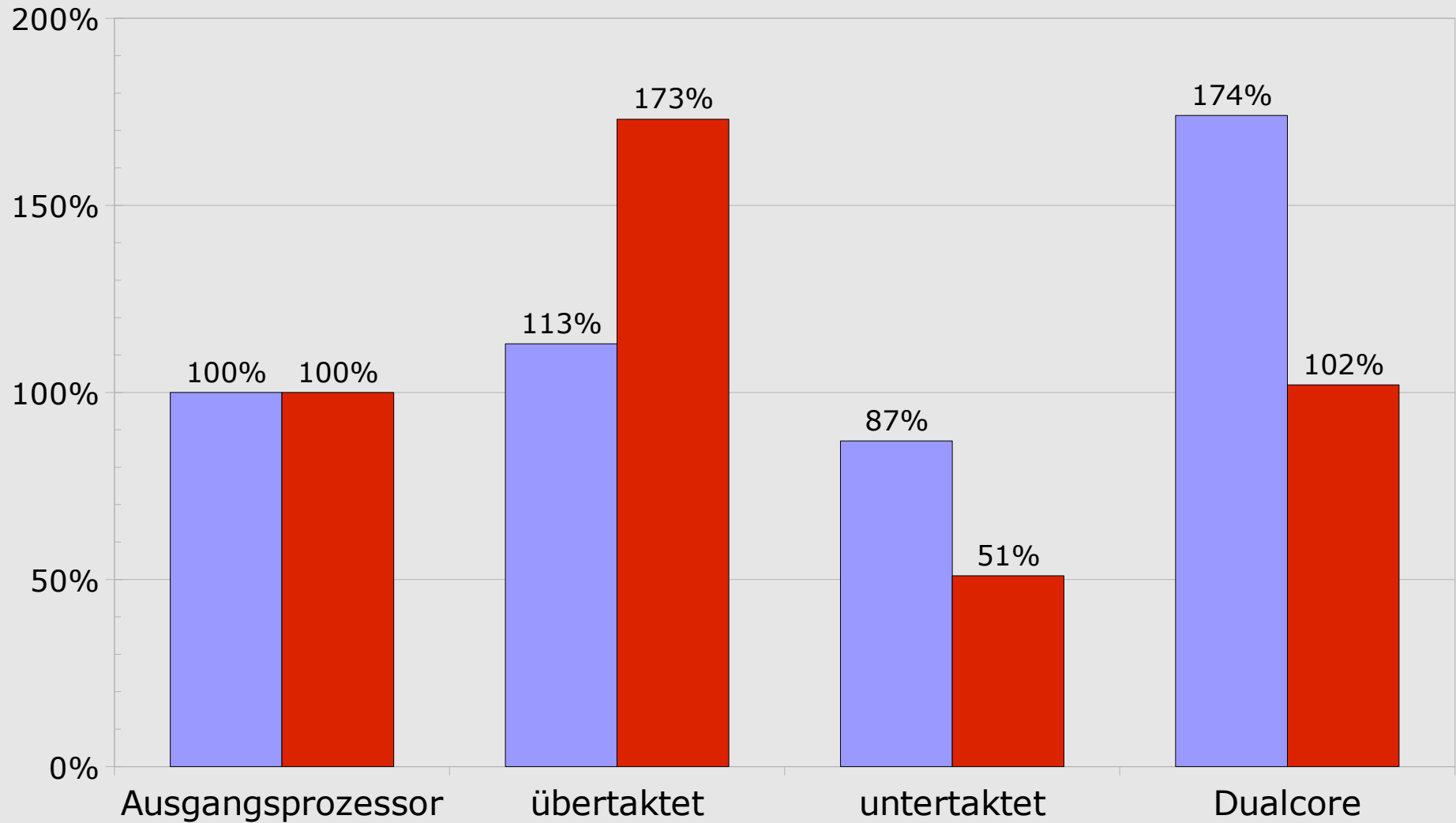
■ Leistung ■ Energieverbrauch



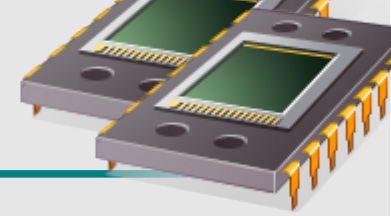
Leistung vs. Energieverbrauch



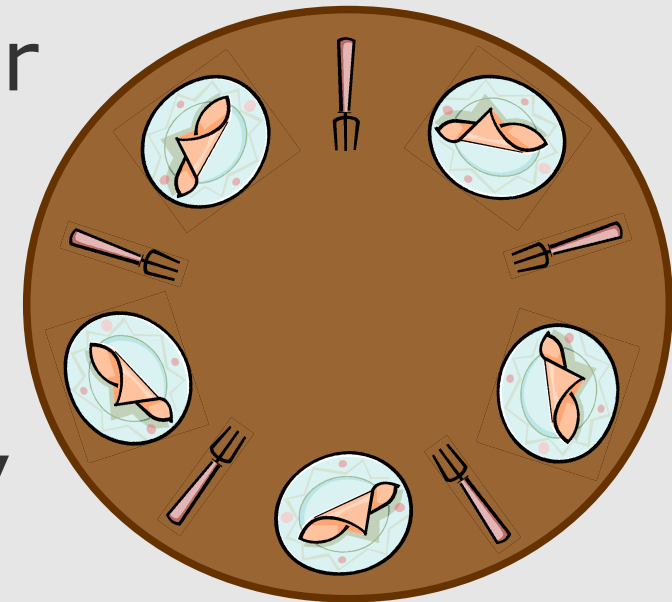
■ Leistung ■ Energieverbrauch



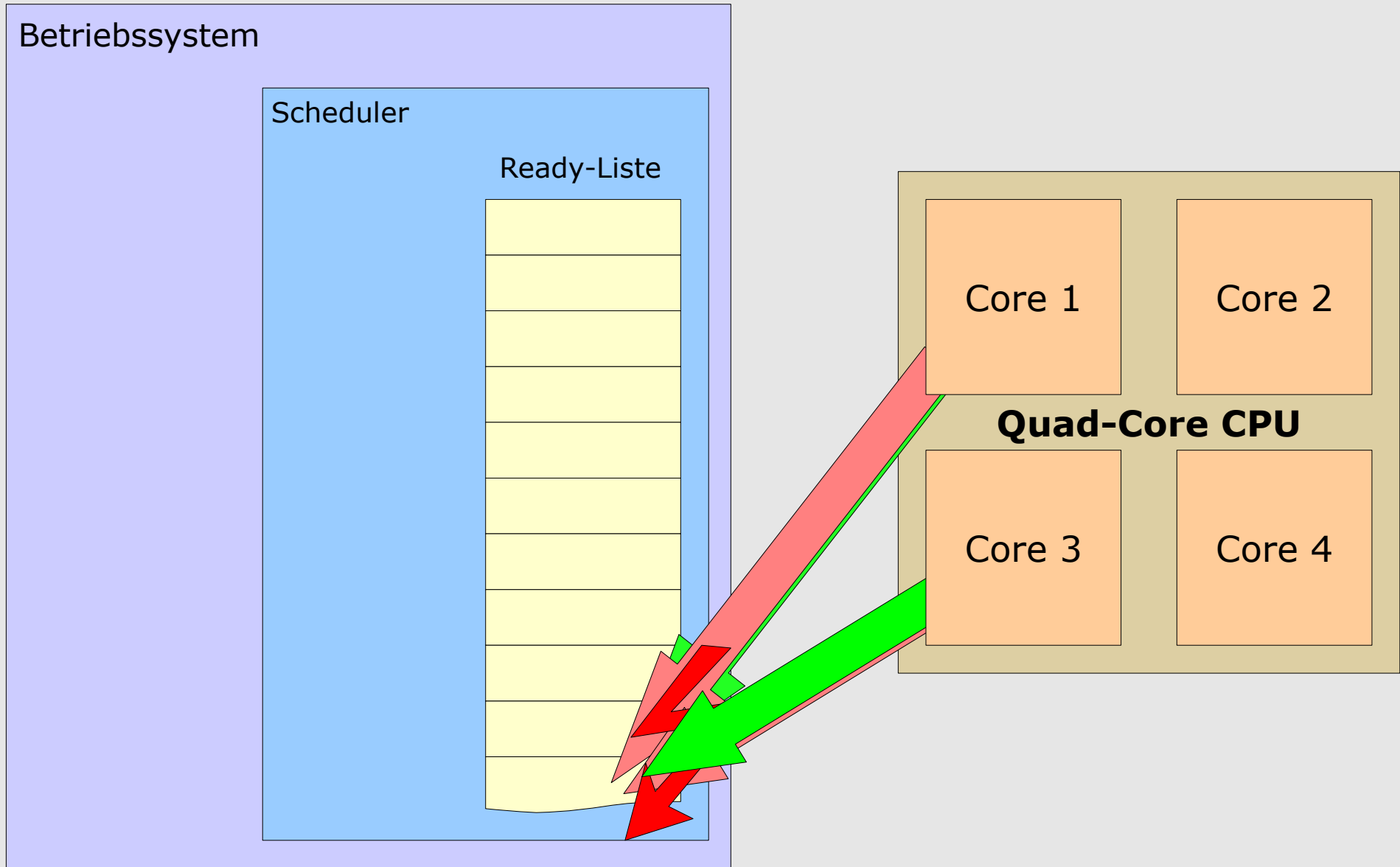
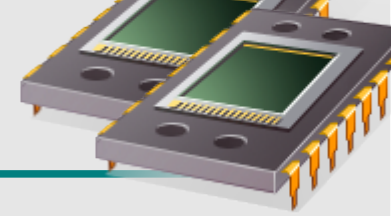
Das Philosophenproblem



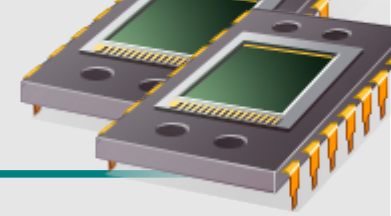
- 5 Philosophen sitzen am runden Tisch
- 5 Teller, dazwischen 5 Gabeln
- ein Philosoph kann entweder denken oder essen
- Will ein Philosoph essen,
 - nimmt er erst die linke Gabel,
 - dann die rechte Gabel,
 - isst, legt anschließend beide Gabeln zurück und denkt wieder.
- Was, wenn alle gleichzeitig essen wollen?



Philosophenproblem - übertragen



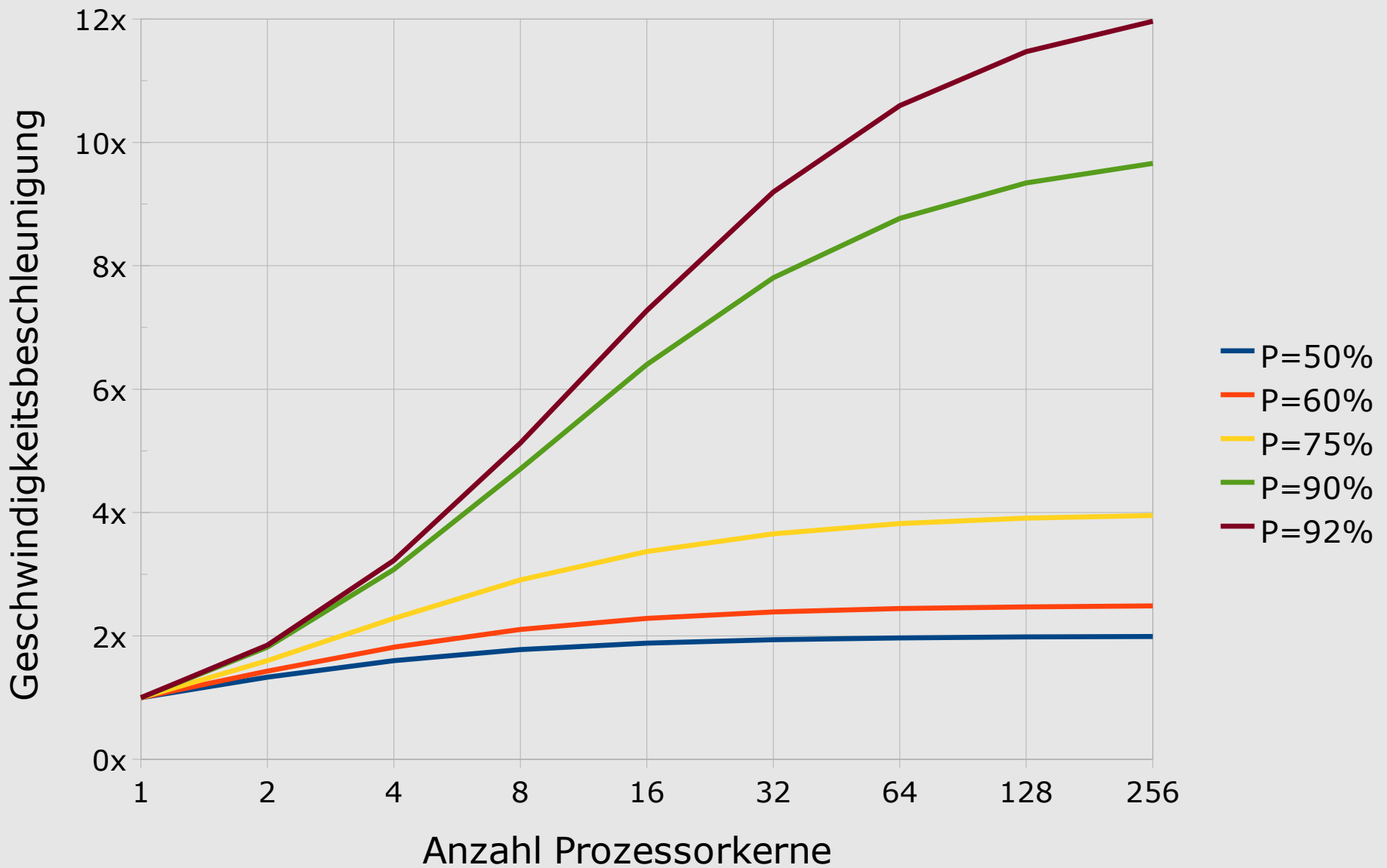
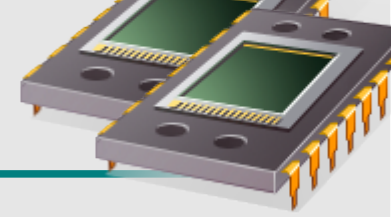
Amdahlsches Gesetz



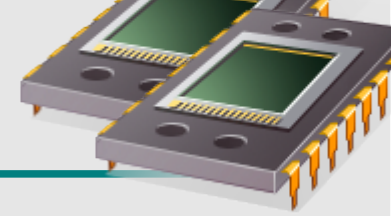
- Gene Amdahl, 1967:
- Sei P der Anteil an parallelem Code
- und $(1-P)$ der Anteil an seriellem Code in einem Programm,
- so ist die mögliche Beschleunigung S eines Programms bei N Prozessoren

$$S = \frac{1}{(1 - P) + \frac{P}{N}}$$

Amdahlsches Gesetz - grafisch



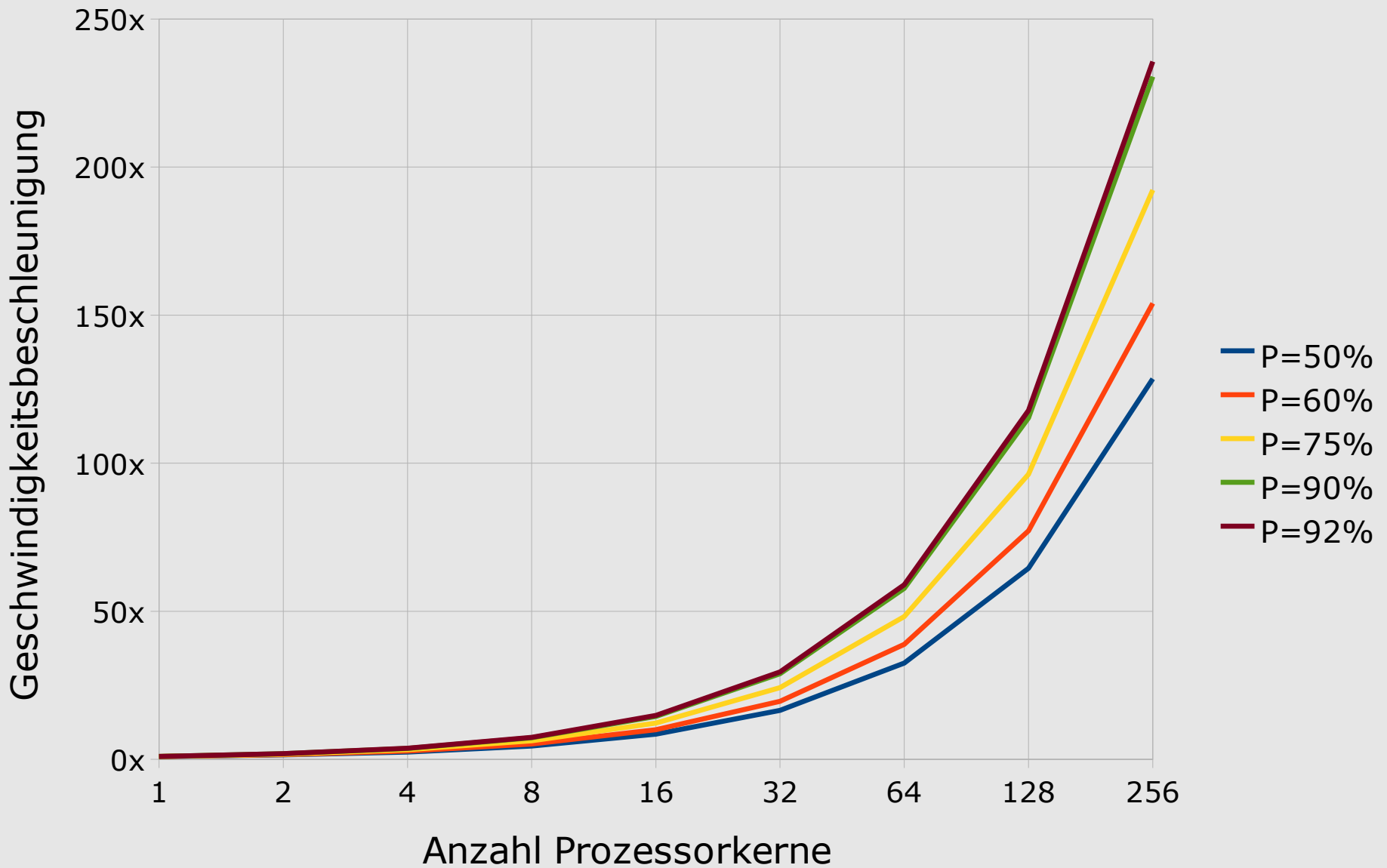
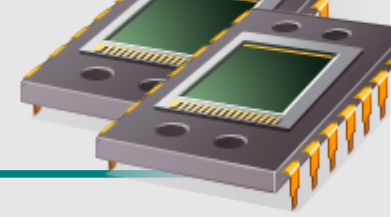
Gustafsons Gesetz



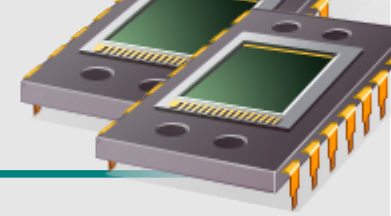
- John Gustafson, 1988:
- Sei P der Anteil an parallelem Code
- und $(1-P)$ der Anteil an seriellem Code in einem Programm,
- so ist die mögliche Beschleunigung S eines Programms bei N Prozessoren

$$S = N - (1 - P)(N - 1)$$

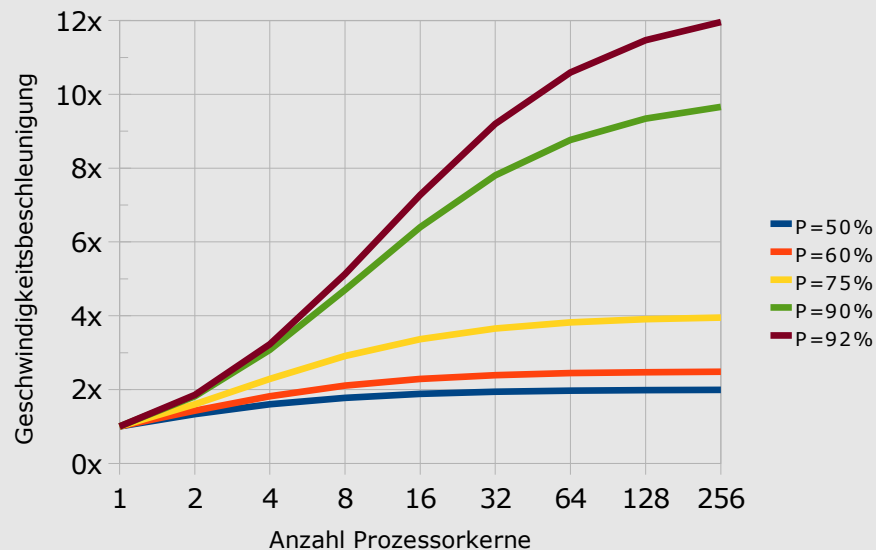
Gustafsons Gesetz - grafisch



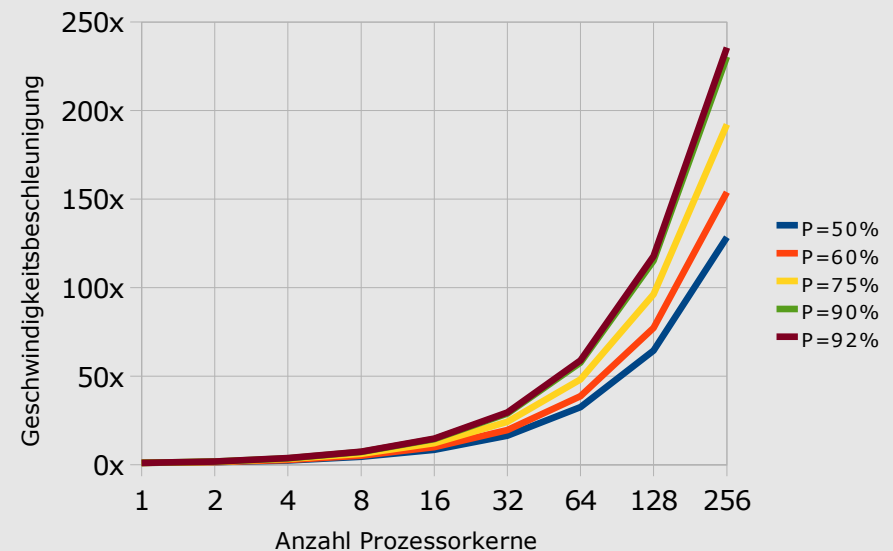
Amdahl vs. Gustafson



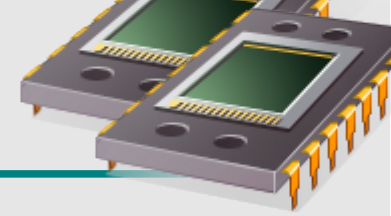
- Amdahl betrachtet den Performance-Gewinn bei konstanter Datenmenge



- Gustafson geht von konstantem Zeitbedarf aus und füttert das Programm mit mehr Daten

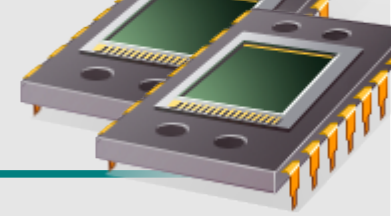


Symmetrische vs. asymmetrische Mehrkernprozessoren



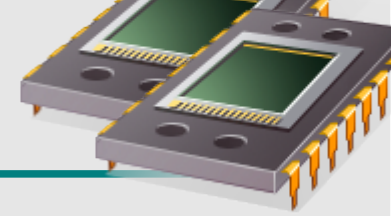
- symmetrischer Mehrkernprozessor:
alle Cores sind vom selben Typ
- asymmetrischer Mehrkernprozessor:
es werden Cores unterschiedlichen Typs verbaut
- Vorteile von Asymmetrie:
Einsparung von Chip-Fläche und Vermeidung
unnötiger Wärmeentwicklung
- hauptsächlich symmetrische
Mehrkernprozessoren im Umlauf
- bekannter asymmetrischer Vertreter:
IBM Cell-Prozessor (PLAYSTATION 3)

Transaktionaler Speicher



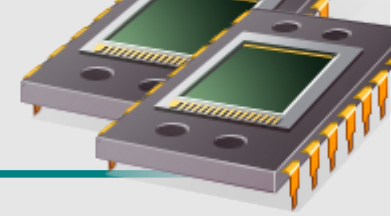
- Problem bei mehreren Prozessoren: konkurrierende Zugriffe auf gemeinsamen Speicher
- Idee aus Datenbanktechnik übernehmen: Transaktionen
- Speicher bietet Instruktionen für „begin transaction“ und „commit transaction“
- Änderungen werden semantisch atomar ausgeführt und sind erst für andere sichtbar, wenn komplett
- Hardware entscheidet, welche Zugriffe parallel ausgeführt werden können

Parallelität mittels Software

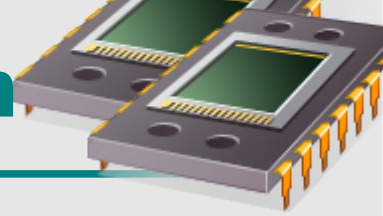


- Parallelisierbare Probleme
- Parallelität über das Betriebssystem
- andere Ansätze
 - Message Passing Interface
 - Open Multi-Processing
 - Threading Building Blocks
- Vergleich der Ansätze

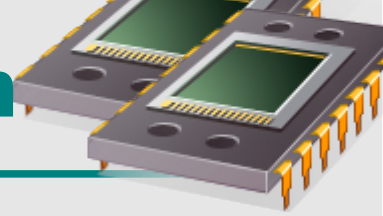
Parallelisierbare Probleme



- Mehr Kerne muss nicht unbedingt einen Performance-Gewinn bedeuten!
- Probleme und ihre Lösungsalgorithmen müssen parallelisierbar sein
- interaktive Prozesse sind das größtenteils nicht
- parallele Algorithmen, z.B. Filteroperationen auf Grafik-/Video-Daten versprechen große Performance-Gewinne

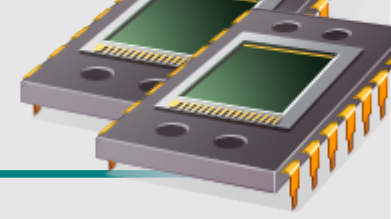


- das Betriebssystem nutzt mehrere Prozessoren automatisch aus: Prozesse und Threads werden parallel ausgeführt
- Problem: ein Prozess mit CPU-Burst lastet einen Kern 100% aus, die restlichen Kerne führen trotzdem nur den Idle-Thread aus
- Lösung: der Programmierer muss seine Anwendung so gestalten, dass sie ihre Arbeit mittels mehrerer Threads erledigt



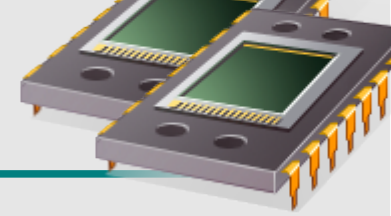
- Erstellen von Prozessen und Threads durch Systemcalls
 - Linux: `fork()`, `pthread_create()`
 - Windows: `CreateProcess()`, `CreateThread()`
- Nachteile
 - Code wird unübersichtlich
 - Programmierer muss sich genau überlegen, welche Threads auf welche Daten zugreifen und geeignete Synchronisierung durchführen

Message Passing Interface (MPI)

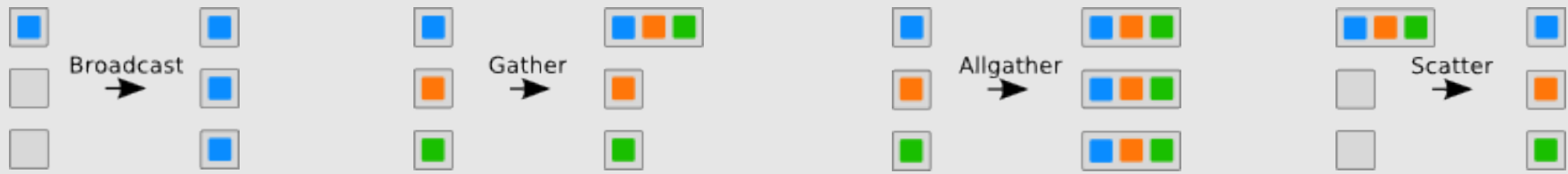


- Standard, der eine Programmierschnittstelle definiert
- Ziel: Nachrichtenaustausch zwischen Prozessen
- Es gibt einen Hauptprozess.
- `MPI_Send()`, `MPI_Recv()`:
Senden und Empfangen von Nachrichten (blockierend)
- `MPI_Isend()`, `MPI_Irecv()`, `MPI_Test()`
(nicht-blockierend)

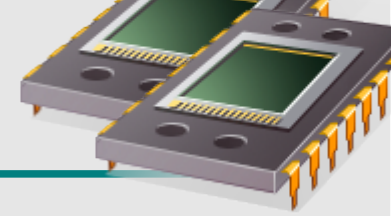
Message Passing Interface (MPI)



- Nachricht an alle: MPI_Bcast()
- Hauptprozess „sammelt“ Daten von allen anderen Prozessen: MPI_Gather(), MPI_Reduce()
- Jeder Prozess sendet seine Daten an alle anderen Prozesse: MPI_Allgather()
- Der Hauptprozess sendet an alle Prozesse eine Nachricht: MPI_Scatter()

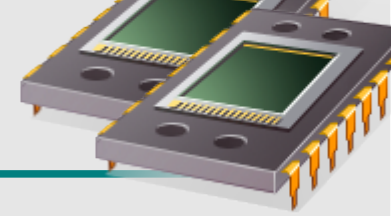


Message Passing Interface (MPI)



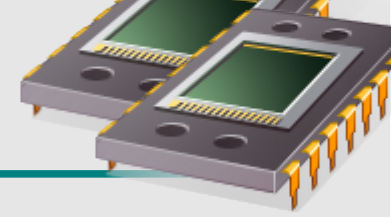
- MPI eignet sich besonders für Supercomputer, wo es keinen gemeinsamen Speicher gibt
- Performance-Gewinn durch exklusiven Zugriff auf den Speicher
- ggf. können die Prozesse sogar räumlich getrennt sein: z.B. Kommunikation über TCP/IP
- theoretisch auch für shared-memory Systeme geeignet, aber wegen dem zusätzlichen Overhead nicht rentabel

Open Multi-Processing (OpenMP)

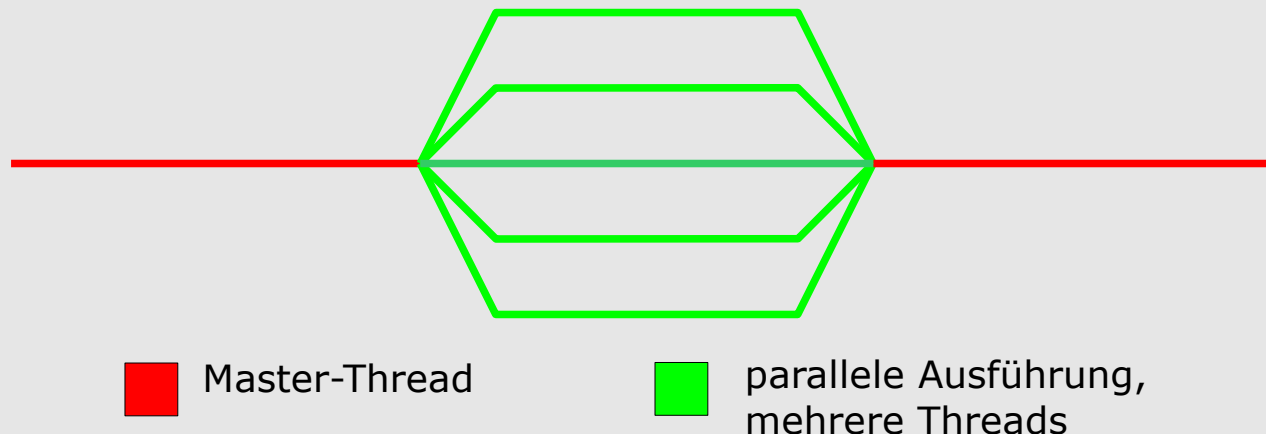


- Programmierschnittstelle für C/C++- und Fortran-Compiler
- Umfang
 - Compilerdirektiven
 - Bibliotheksfunktionen
 - Umgebungsvariablen
- Parallelität wird zur Übersetzung durch den Compiler erzeugt
- Auszeichnung paralleler Abschnitte im Code mittels Compilerdirektiven

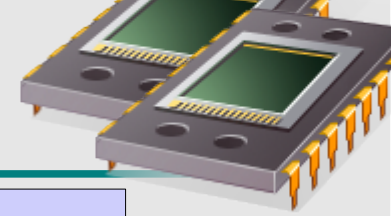
Open Multi-Processing (OpenMP)



- zu Beginn: initialer Thread
- Erreicht die Ausführung einen parallelen Abschnitt, erzeugt er zusätzliche Threads.
- Master-Thread und Team
- Am Ende des parallelen Abschnitts läuft nur der Master-Thread weiter.



OpenMP: parallele for-Schleife



```
void main(int argc, char** argv)
{
    double a[3][3], b[3][3], c[3][3];
    int i, j, k;
    double sum;

    for(int i = 0; i < 3; i++)
    {
        for(j = 0; j < 3; j++) {
            sum = 0;
            for(k = 0; k < 3; k++)
                sum += a[i][k] * b[k][j];
            c[i][j] = sum;
        }
    }
}
```

Eingabe:
a, b sind 3x3-Matrizen

Ausgabe:
c als 3x3-Matrix, die das
Matrizenprodukt a*b enthält

Variable, die das
Skalarprodukt halten wird

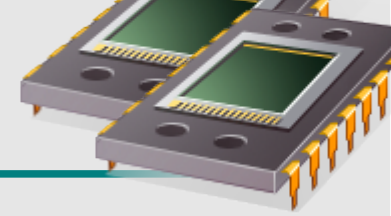
Für jede Zeile, ...

für jede Spalte, ...

berechne das
Skalarprodukt ...

und speichere es in
der Ergebnismatrix.

OpenMP: parallele for-Schleife



```
#include <omp.h>
```

Einbinden der notwendigen Definitionen für den Compiler

```
void main(int argc, char** argv)
```

parallel
Ein paralleler Abschnitt beginnt: Master-Thread erzeugt Team von mehreren Threads.

#pragma omp
Es folgt eine OpenMP-Anweisung.

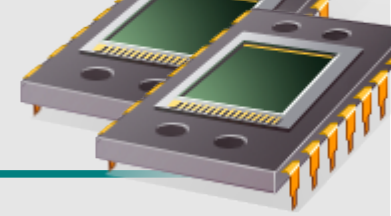
for
Die nächste for-Schleife wird parallel ausgeführt. Die Zählervariable i ist automatisch privat.

```
#pragma omp parallel for private(sum)
```

private(sum)
sum ist ebenfalls privat, d.h. jeder Thread hat seine eigene Kopie.

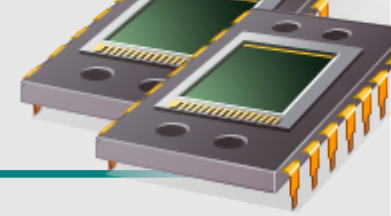
```
for(int i = 0; i < 3; i++)  
{  
    for(j = 0; j < 3; j++) {  
        sum = 0;  
        for(k = 0; k < 3; k++)  
            sum += a[i][k] * b[k][j];  
        c[i][j] = sum;  
    }  
}
```

OpenMP: weitere Konstrukte



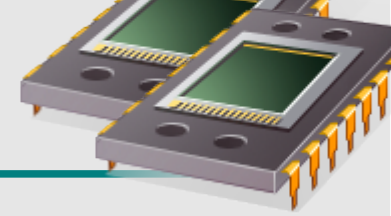
- sections: verschiedene Codeblöcke parallel ausführen
- barrier: an einer Barriere ist ein Synchronisationspunkt. Alle Threads warten dort, bis alle anderen auch an dieser Stelle angekommen sind
- single/master: ein Abschnitt wird nur einmalig/nur vom Master-Thread ausgeführt
- atomic: atomares Ausführen von Operationen

Threading Building Blocks (TBB)



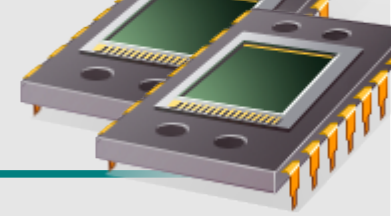
- von Intel entwickeltes freies Framework
- an die STL angelehnt:
 - Container
 - Algorithmen
 - Funktionsobjekte: „Funktork“
- Task-Scheduler koordiniert die Zuordnung von Tasks auf Threads

TBB: Range



- Abstraktion eines Wertebereichs
- mathematisch mit einem Intervall zu vergleichen, aber beliebige Datentypen möglich
- für Parallelisierung wichtig: Range sollte teilbar sein
- „Partitionskonstruktor“

TBB: Range - Beispiel



```
struct MyIntRange
```

```
{
```

```
    int from, to;
```

Variablen, die Anfang und Ende des Bereichs halten

```
    bool empty() const { return (from == to); }
```

```
    int size() const { return to - from; }
```

```
    int begin() const { return from; }
```

```
    int end() const { return to; }
```

Primitive Methoden für Bereich leer?
Größe des Bereichs?
Anfang? Ende?

```
    bool is_divisible() const {
```

```
        return (size() > 10);
```

```
    }
```

Gibt an, ob ein Bereich weiter unterteilbar ist

Nur Bereiche größer 10 sind teilbar.

```
    MyIntRange(int x1, int x2) {
```

```
        from = x1; to = x2;
```

```
    }
```

Konstruktor, der Bereich [x1, x2) erzeugt

Partitionskonstruktor

```
    MyIntRange(MyIntRange& r, split) {
```

```
        int middle = (r.from + r.to) / 2;
```

```
        from = r.from; to = middle;
```

```
        r.from = middle;
```

```
    }
```

```
};
```

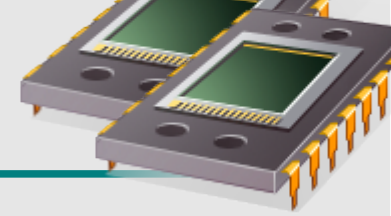
split wird von TBB definiert und unterscheidet diesen speziellen Konstruktor

Berechne Intervallmitte, ...

und verkleinere den übergebenen Bereich.

setze Anfang bis Mitte für den neuen Bereich ...

TBB: parallele for-Schleife



```
int numbers[100];
```

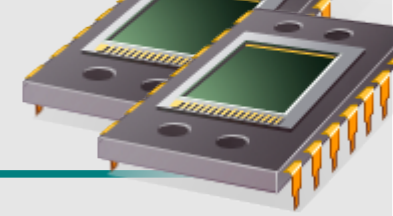
Array der Größe 100.
Eingabewerte sollen alle
verdoppelt werden.

```
for(int i = 0; i < 100; i++)  
    numbers[i] *= 2;
```

Für jedes Element
im Array ...

verdopple den Wert.

TBB: parallele for-Schleife



Kapselung durch Klasse

```
class DoubleInts
```

```
{  
    int* const numbers;
```

```
public:
```

```
    DoubleInts(int* array) : numbers(array) {}
```

```
    void operator()  
        (const blocked_range<int>& r) const
```

```
{  
    for(int i = r.begin(); i != r.end(); i++)  
        numbers[i] *= 2;  
};
```

```
parallel_for(blocked_range<int>(0, 100),  
            DoubleInts(data),  
            auto_partitioner());
```

Zeiger auf das Datenfeld

Zeiger auf das Datenfeld
in der Klasse speichern

Konstruktor

TBB-vordefiniertes Template für Ranges
hier: Integer-Range

Schleifenrumpf:
Iteriere durch den Range
und verdopple jedes
Element darin.

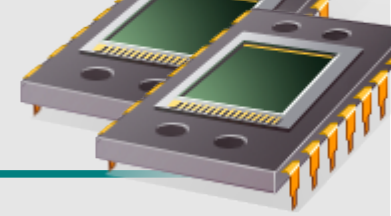
Neuer Task: for-Schleife

Erzeuge Bereich [0, 100).

Klasse instanzieren und
Datenfeld übergeben

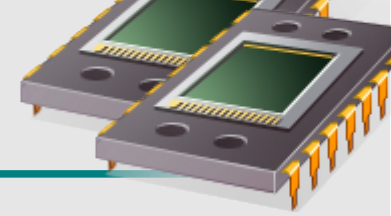
TBB kümmert sich um die Partitionierung des Bereichs.

TBB: weitere Funktionen



- vordefinierte Ranges von ein- bis dreidimensional
- komplexes Task-System
- Filter, Pipelines
- atomic-Template für atomare Operationen
- generische thread-safe Container:
 - Queue
 - Vector
 - HashMap

Vergleich der Ansätze



MPI

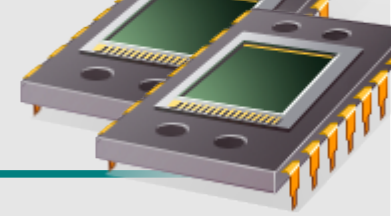
- Nachrichten basierend
- auch für Systeme ohne shared Memory
- Parallelismus auch über Rechengrenzen hinweg, z.B. TCP/IP
- für Probleme mit viel Rechenaufwand

OpenMP

- im Compiler integriert
- nur für C/C++ und Fortran
- „einfach“ nur Compiler-direktiven einsetzen
- Code bleibt nahezu unverändert
- Debugging: „Parallel-Modus“ ausschaltbar

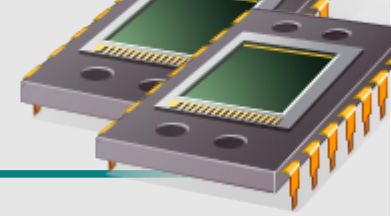
TBB

- Framework, an SLT angelehnt
- Kapselung
- Task-Scheduler koordiniert alle Tasks
- mehr Code zu schreiben, trotzdem bleibt der Code gut lesbar

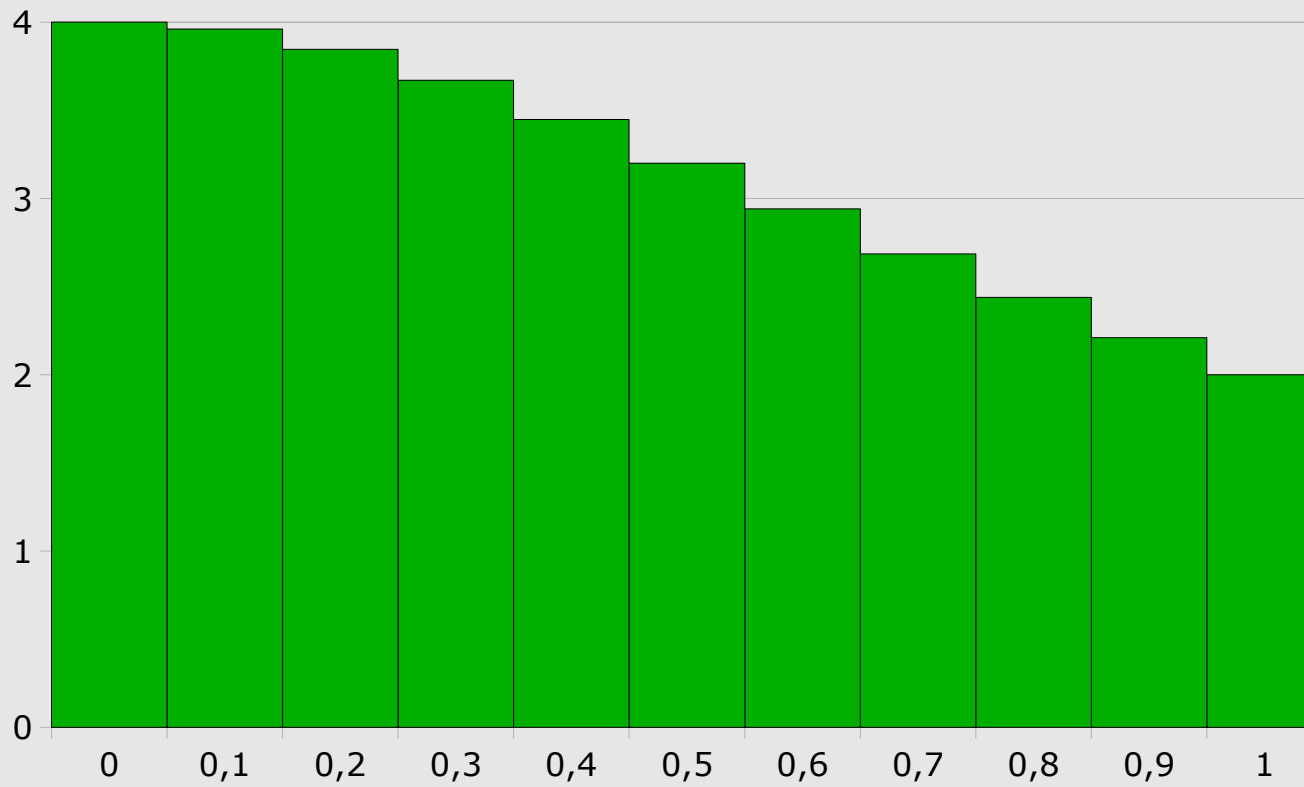


- Parallelität: Energieeinsparung, aber trotzdem ein Performance-Gewinn möglich
- unterschiedliche Ansätze für unterschiedliche Probleme:
 - MPI: Nachrichtenaustausch
 - OpenMP: Parallelität durch den Compiler
 - TBB: STL-angelehntes Framework
- alle vorgestellten Ansätze sind unabhängig von der Anzahl der Prozessorkerne
- Probleme müssen parallelisierbar sein, sonst nützen mehrere Kerne auch nichts

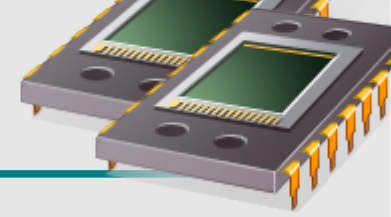
Live-Beispiel



$$\pi = \int_0^1 \frac{4}{(1+x^2)} dx$$



Noch Fragen?



42