

The Free Lunch

27. Parallel Programming I

Moore's Law und The Free Lunch, Hardware Architekturen, Parallele Ausführung, Klassifikation nach Flynn, Multi-Threading, Parallelität und Nebenläufigkeit, C++ Threads, Skalierbarkeit: Amdahl und Gustafson, Daten- und Taskparallelität, Scheduling

[Task-Scheduling: Cormen et al, Kap. 27] [Concurrency, Scheduling: Williams, Kap. 1.1 – 1.2]

The free lunch is over ⁵⁴

⁵⁴"The Free Lunch is Over", a fundamental turn toward concurrency in software, Herb Sutter, Dr. Dobb's Journal, 2005

Moore's Law



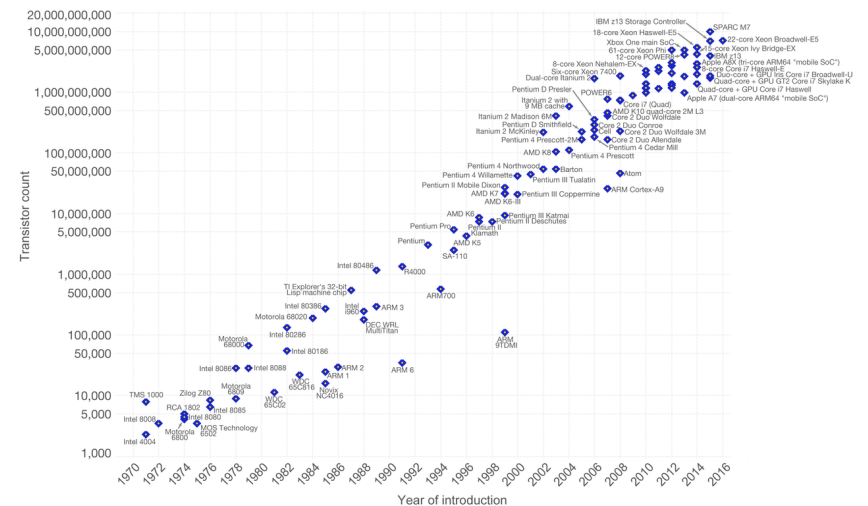
Gordon E. Moore (1929)

Beobachtung von Gordon E. Moore:

Die Anzahl Transistoren in integrierten Schaltkreisen verdoppelt sich ungefähr alle zwei Jahre.

Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Licensed under CC-BY-SA by the author Max Roser.

Für eine lange Zeit...

- wurde die sequentielle Ausführung schneller ("Instruction Level Parallelism", "Pipelining", Höhere Frequenzen)
- mehr und kleinere Transistoren = mehr Performance
- Programmierer warteten auf die nächste schnellere Generation

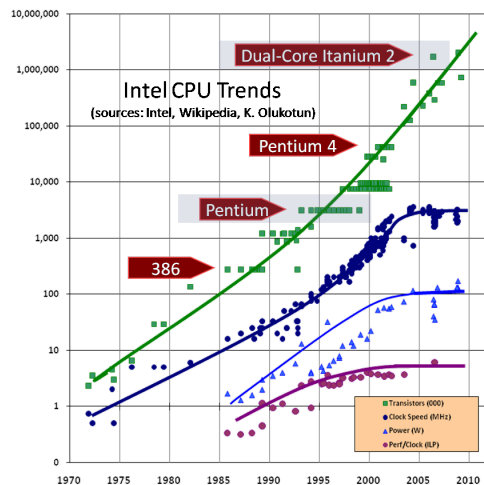
Heute

- steigt die Frequenz der Prozessoren kaum mehr an (Kühlproblematik)
- steigt die Instruction-Level Parallelität kaum mehr an
- ist die Ausführungsgeschwindigkeit in vielen Fällen dominiert von Speicherzugriffszeiten (Caches werden aber immer noch grösser und schneller)

848

849

Trends



<http://www.gotw.ca/publications/concurrency-ddj.htm>

Multicore

- Verwende die Transistoren für mehr Rechenkerne
- Parallelität in der Software
- Implikation: Programmierer müssen parallele Programme schreiben, um die neue Hardware vollständig ausnutzen zu können

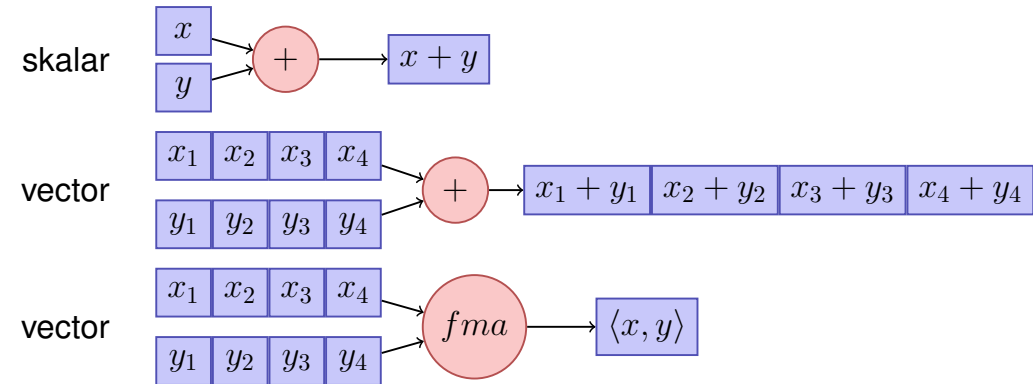
851

Formen der Parallelen Ausführung

- Vektorisierung
- Pipelining
- Instruction Level Parallelism
- Multicore / Multiprocessing
- Verteiltes Rechnen

Vektorisierung

Parallele Ausführung derselben Operation auf Elementen eines Vektor(Register)s



852

853

Pipelining in CPUs



Mehrere Stufen

- Jede Instruktion dauert 5 Zeiteinheiten (Zyklen)
- Im besten Fall: 1 Instruktion pro Zyklus, nicht immer möglich ("stalls")

Parallelität (mehrere funktionale Einheiten) führt zu *schnellerer Ausführung*.

ILP – Instruction Level Parallelism

Moderne CPUs führen unabhängige Instruktionen intern auf mehreren Einheiten parallel aus

- Pipelining
- Superskalare CPUs (Mehrere Instruktionen pro Zyklus)
- Out-Of-Order Execution (Programmierer sieht die sequentielle Ausführung)
- Speculative Execution (Instruktionen werden spekulativ ausgeführt und unterbrochen, wenn die Bedingung zu deren Ausführung nicht erfüllt ist.)

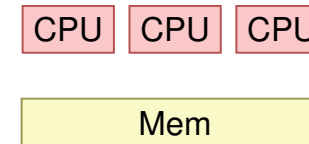
854

855

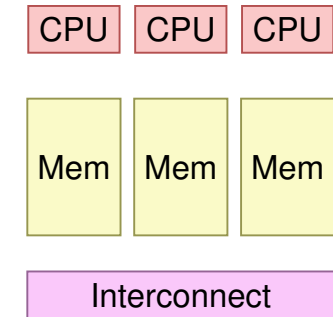
27.2 Hardware Architekturen

Gemeinsamer vs. verteilter Speicher

Gemeinsamer Speicher



Verteilter Speicher



856

857

Shared vs. Distributed Memory Programming

- Kategorien des Programmierinterfaces
 - Kommunikation via Message Passing
 - Kommunikation via geteiltem Speicher
- Es ist möglich:
 - Systeme mit gemeinsamen Speicher als verteilte Systeme zu programmieren (z.B. mit Message Passing Interface MPI)
 - Systeme mit verteiltem Speicher als System mit gemeinsamen Speicher zu programmieren (z.B. Partitioned Global Address Space PGAS)

Architekturen mit gemeinsamen Speicher

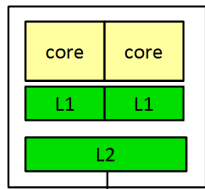
- Multicore (Chip Multiprocessor - CMP)
- Symmetric Multiprocessor Systems (SMP)
- Simultaneous Multithreading (SMT = Hyperthreading)
 - nur ein physischer Kern, Mehrere Instruktionsströme/Threads: mehrere virtuelle Kerne
 - Zwischen ILP (mehrere Units für einen Strom) und Multicore (mehrere Units für mehrere Ströme). Limitierte parallele Performance
- Non-Uniform Memory Access (NUMA)

Gleiches Programmierinterface!

858

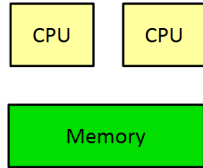
859

Übersicht

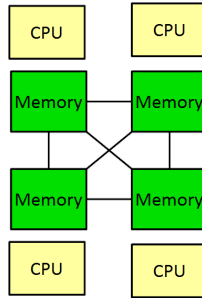


Memory

CMP



SMP

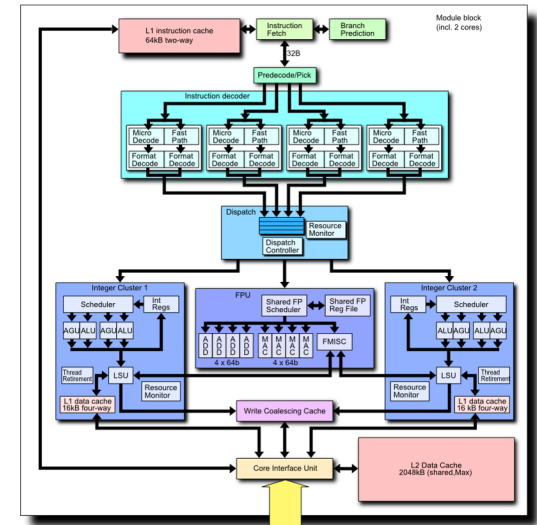


NUMA

Ein Beispiel

AMD Bulldozer: Zwischen CMP und SMT

- 2x integer core
- 1x floating point core



860

Wikipedia
861

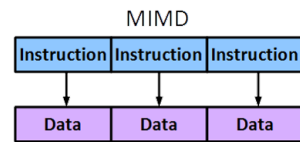
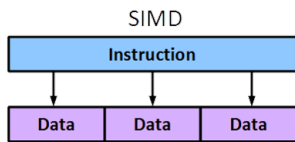
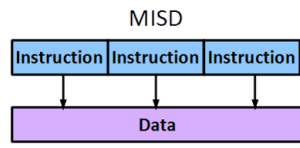
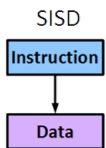
Klassifikation nach Flynn

Single-Core

Fault-Tolerance

SI = Single Instruction
MI = Multiple Instructions

SD = Single Data
MD = Multiple Data



Vector Computing / GPU

Multi-Core

862

Massiv Parallele Hardware

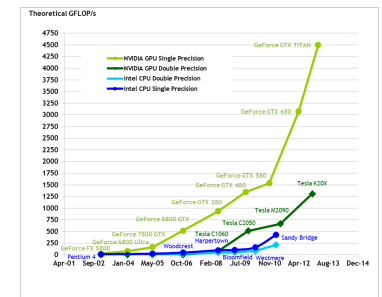
[General Purpose] Graphical Processing Units ([GP]GPUs)

■ Revolution im High Performance Computing

- Calculation 4.5 TFlops vs. 500 GFlops
- Memory Bandwidth 170 GB/s vs. 40 GB/s

■ SIMD

- Hohe Datenparallelität
- Benötigt eigenes Programmiermodell. Z.B. CUDA / OpenCL



863

Wikipedia
861

27.3 Multi-Threading, Parallelität und Nebenläufigkeit

Prozesse und Threads

- Prozess: Instanz eines Programmes
 - jeder Prozess hat seinen eigenen Kontext, sogar eigenen Adressraum
 - OS verwaltet Prozesse (Ressourcenkontrolle, Scheduling, Synchronisierung)
- Threads: Ausführungsfäden eines Programmes
 - Threads teilen sich einen Adressraum
 - Schneller Kontextwechsel zwischen Threads

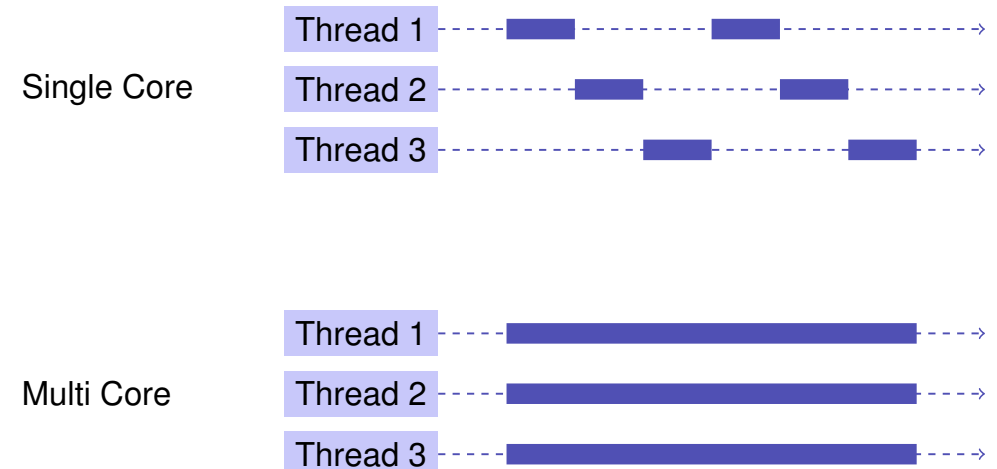
864

865

Warum Multithreading?

- Verhinderung vom "Polling" auf Ressourcen (Files, Netzwerkzugriff, Tastatur)
- Interaktivität (z.B. Responsivität von GUI Programmen)
- Mehrere Applikationen / Clients gleichzeitig instanzierbar
- Parallelität (Performanz!)

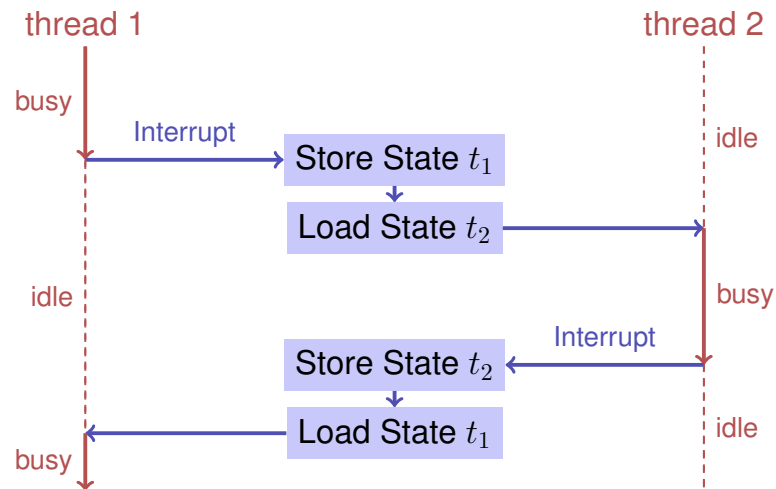
Multithreading konzeptuell



866

867

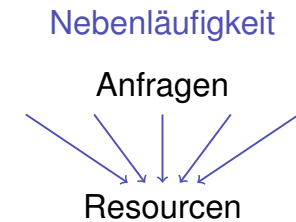
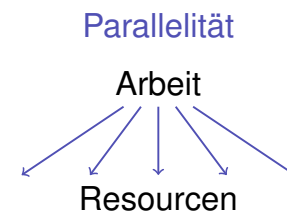
Threadwechsel auf einem Core (Preemption)



868

Parallelität vs. Nebenläufigkeit (Concurrency)

- **Parallelität:** Verwende zusätzliche Ressourcen (z.B. CPUs), um ein Problem schneller zu lösen
- **Nebenläufigkeit:** Verwalte gemeinsam genutzte Ressourcen (z.B. Speicher) korrekt und effizient
- Begriffe überlappen offensichtlich. Bei parallelen Berechnungen besteht fast immer Synchronisierungsbedarf.



869

Thread-Sicherheit

Thread-Sicherheit bedeutet, dass in der nebenläufigen Anwendung eines Programmes dieses sich immer wie gefordert verhält.

Viele Optimierungen (Hardware, Compiler) sind darauf ausgerichtet, dass sich ein *sequentielles* Programm korrekt verhält.

Nebenläufige Programme benötigen für ihre Synchronisierungen auch eine Annotation, welche gewisse Optimierungen selektiv abschaltet

870

Beispiel: Caches

- Speicherzugriff auf Register schneller als auf den gemeinsamen Speicher
- Prinzip der Lokalität
- Verwendung von Caches (transparent für den Programmierer)

Ob und wie weit die Cache-Kohärenz sichergestellt wird ist vom eingesetzten System abhängig.



871

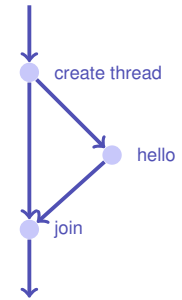
27.4 C++ Threads

C++11 Threads

```
#include <iostream>
#include <thread>

void hello(){
    std::cout << "hello\n";
}

int main(){
    // create and launch thread t
    std::thread t(hello);
    // wait for termination of t
    t.join();
    return 0;
}
```



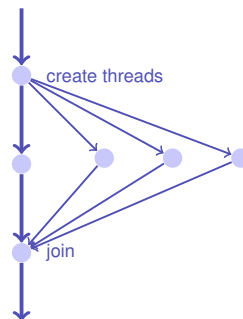
872

873

C++11 Threads

```
void hello(int id){
    std::cout << "hello from " << id << "\n";
}

int main(){
    std::vector<std::thread> tv(3);
    int id = 0;
    for (auto & t:tv)
        t = std::thread(hello, ++id);
    std::cout << "hello from main \n";
    for (auto & t:tv)
        t.join();
    return 0;
}
```



874

Nichtdeterministische Ausführung!

Eine Ausführung:

```
hello from main
hello from 2
hello from 1
hello from 0
```

Andere Ausführung:

```
hello from 1
hello from main
hello from 0
hello from 2
```

Andere Ausführung:

```
hello from main
hello from 0
hello from hello from 1
2
```

875

Technisches Detail

Um einen Thread als Hintergrundthread weiterlaufen zu lassen:

```
void background();

void someFunction(){
    ...
    std::thread t(background);
    t.detach();
    ...
} // no problem here, thread is detached
```

27.5 Skalierbarkeit: Amdahl und Gustafson

Mehr Technische Details

- Beim Erstellen von Threads werden auch Referenzparameter kopiert, ausser man gibt explizit `std::ref` bei der Konstruktion an.
- Funktoren oder Lambda-Expressions können auch auf einem Thread ausgeführt werden
- In einem Kontext mit Exceptions sollte das `join` auf einem Thread im `catch`-Block ausgeführt werden

Noch mehr Hintergründe im Kapitel 2 des Buches *C++ Concurrency in Action*, Anthony Williams, Manning 2012. Auch online bei der ETH Bibliothek erhältlich.

876

877

Skalierbarkeit

In der parallelen Programmierung:

- Geschwindigkeitssteigerung bei wachsender Anzahl p Prozessoren
- Was passiert, wenn $p \rightarrow \infty$?
- Linear skalierendes Programm: Linearer Speedup

878

879

Parallele Performanz

Gegeben fixierte Rechenarbeit W (Anzahl Rechenschritte)

Sequentielle Ausführungszeit sei T_1

Parallele Ausführungszeit T_p auf p CPUs

- Perfektion: $T_p = T_1/p$
- Performanzverlust: $T_p > T_1/p$ (üblicher Fall)
- Hexerei: $T_p < T_1/p$

Paralleler Speedup

Paralleler Speedup S_p auf p CPUs:

$$S_p = \frac{W/T_p}{W/T_1} = \frac{T_1}{T_p}$$

- Perfektion: Linearer Speedup $S_p = p$
- Verlust: sublinearer Speedup $S_p < p$ (der übliche Fall)
- Hexerei: superlinearer Speedup $S_p > p$

Effizienz: $E_p = S_p/p$

880

881

Erreichbarer Speedup?

Paralleles Programm

Paralleler Teil	Seq. Teil
80%	20%

$$T_1 = 10$$

$$T_8 = \frac{10 \cdot 0.8}{8} + 10 \cdot 0.2 = 1 + 2 = 3$$

$$S_8 = \frac{T_1}{T_8} = \frac{10}{3} \approx 3.3 < 8 \quad (!)$$

Amdahl's Law: Zutaten

Zu leistende Rechenarbeit W fällt in zwei Kategorien

- Parallelisierbarer Teil W_p
- Nicht parallelisierbarer, sequentieller Teil W_s

Annahme: W kann mit *einem* Prozessor in W Zeiteinheiten sequentiell erledigt werden ($T_1 = W$):

$$T_1 = W_s + W_p$$

$$T_p \geq W_s + W_p/p$$

882

883

Amdahl's Law

$$S_p = \frac{T_1}{T_p} \leq \frac{W_s + W_p}{W_s + \frac{W_p}{p}}$$

Amdahl's Law

Mit seriellem, nicht parallelisierbarem Anteil λ : $W_s = \lambda W$,
 $W_p = (1 - \lambda)W$:

$$S_p \leq \frac{1}{\lambda + \frac{1-\lambda}{p}}$$

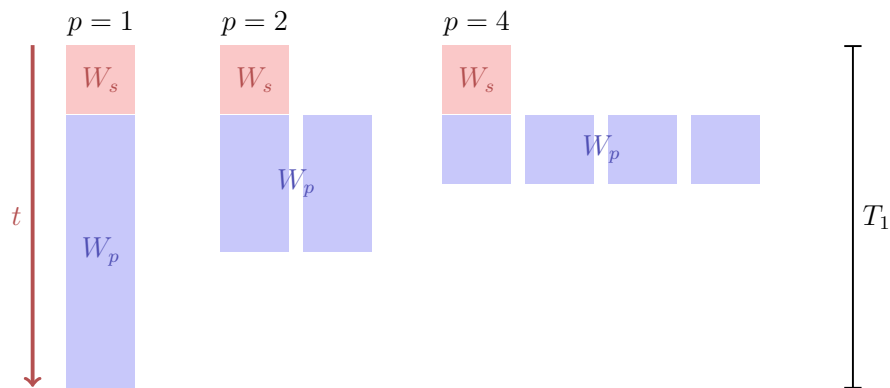
Somit

$$S_\infty \leq \frac{1}{\lambda}$$

884

885

Illustration Amdahl's Law



886

887

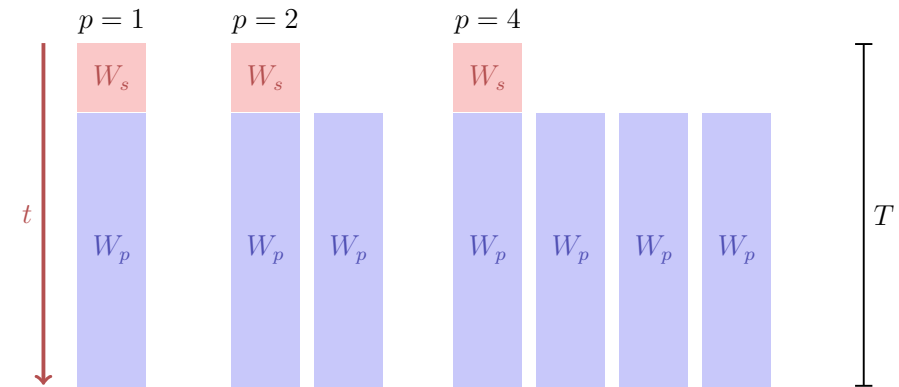
Amdahl's Law ist keine gute Nachricht

Alle nicht parallelisierbaren Teile können Problem bereiten und stehen der Skalierbarkeit entgegen.

Gustafson's Law

- Halte die Ausführungszeit fest.
- Variiere die Problemgröße.
- Annahme: Der sequentielle Teil bleibt konstant, der parallele Teil wird grösser.

Illustration Gustafson's Law



888

889

Gustafson's Law

Arbeit, die mit einem Prozessor in der Zeit T erledigt werden kann:

$$W_s + W_p = T$$

Arbeit, die mit p Prozessoren in der Zeit T erledigt werden kann:

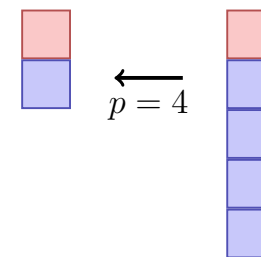
$$W_s + p \cdot W_p = \lambda \cdot T + p \cdot (1 - \lambda) \cdot T$$

Speedup:

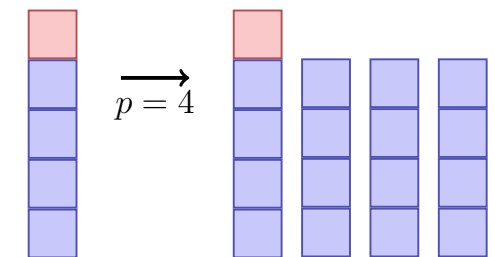
$$\begin{aligned} S_p &= \frac{W_s + p \cdot W_p}{W_s + W_p} = p \cdot (1 - \lambda) + \lambda \\ &= p - \lambda(p - 1) \end{aligned}$$

Amdahl vs. Gustafson

Amdahl



Gustafson



890

891

Amdahl vs. Gustafson

Die Gesetze von Amdahl und Gustafson sind Modelle der Laufzeitverbesserung bei Parallelisierung.

Amdahl geht von einem festen *relativen* sequentiellen Anteil der Arbeit aus, während Gustafson von einem festen *absoluten* sequentiellen Teil ausgeht (der als Bruchteil der Arbeit W_1 ausgedrückt wird und bei Zunahme der Arbeit nicht wächst).

Die beiden Modelle widersprechen sich nicht, sondern beschreiben die Laufzeitverbesserung verschiedener Probleme und Algorithmen.

892

27.6 Task- und Datenparallelität

893

Paradigmen der Parallelen Programmierung

- *Task Parallel*: Programmierer legt parallele Tasks explizit fest.
- *Daten-Parallel*: Operationen gleichzeitig auf einer Menge von individuellen Datenobjekten.

894

Beispiel Data Parallel (OMP)

```
double sum = 0, A[MAX];
#pragma omp parallel for reduction (+:ave)
for (int i = 0; i < MAX; ++i)
    sum += A[i];
return sum;
```

895

Beispiel Task Parallel (C++11 Threads/Futures)

```
double sum(Iterator from, Iterator to)
{
    auto len = from - to;
    if (len > threshold){
        auto future = std::async(sum, from, from + len / 2);
        return sumS(from + len / 2, to) + future.get();
    }
    else
        return sumS(from, to);
}
```

896

Partitionierung und Scheduling

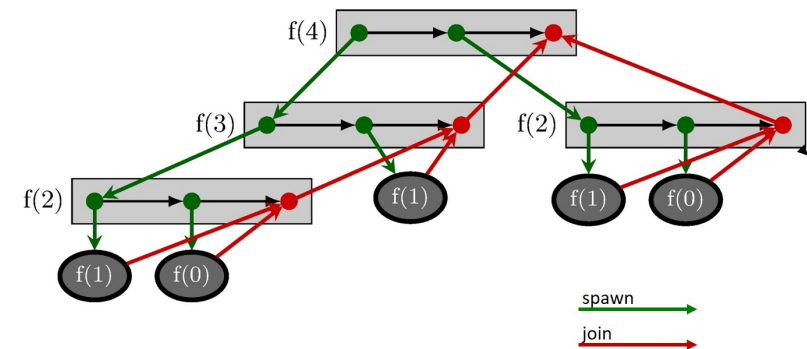
- Aufteilung der Arbeit in parallele Tasks (Programmierer oder System)
 - Ein Task ist eine Arbeitseinheit
 - Frage: welche Granularität?
- Scheduling (Laufzeitsystem)
 - Zuweisung der Tasks zu Prozessoren
 - Ziel: volle Ressourcennutzung bei wenig Overhead

897

Beispiel: Fibonacci P-Fib

```
if  $n \leq 1$  then
    return  $n$ 
else
     $x \leftarrow$  spawn P-Fib( $n - 1$ )
     $y \leftarrow$  spawn P-Fib( $n - 2$ )
    sync
    return  $x + y$ ;
```

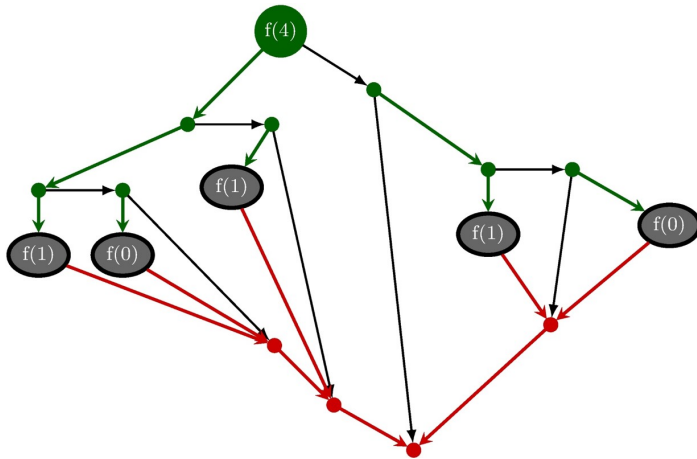
P-Fib Task Graph



898

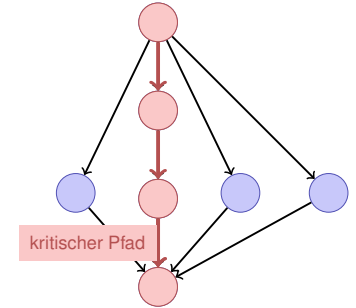
899

P-Fib Task Graph



Frage

- Jeder Knoten (Task) benötigt 1 Zeiteinheit.
- Pfeile bezeichnen Abhängigkeiten.
- Minimale Ausführungseinheit wenn Anzahl Prozessoren = ∞ ?

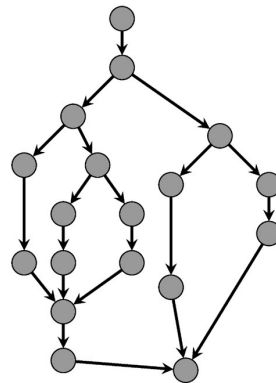


900

901

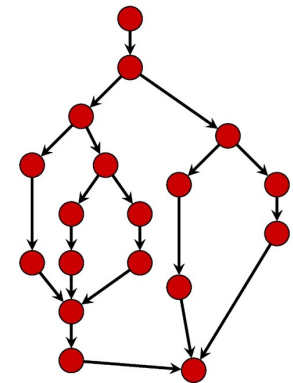
Performanzmodell

- p Prozessoren
- Dynamische Zuteilung
- T_p : Ausführungszeit auf p Prozessoren



Performanzmodell

- T_p : Ausführungszeit auf p Prozessoren
- T_1 : *Arbeit*: Zeit für die gesamte Berechnung auf einem Prozessor
- T_1/T_p : Speedup



902

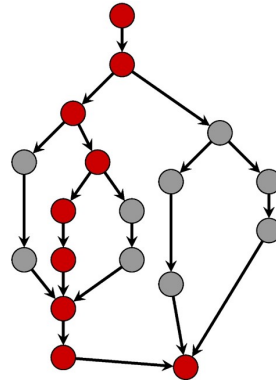
903

Performanzmodell

- T_∞ : **Zeitspanne**: Kritischer Pfad. Ausführungszeit auf ∞ Prozessoren. Längster Pfad von der Wurzel zur Senke.
- T_1/T_∞ : **Parallelität**: breiter ist besser
- Untere Grenzen

$$T_p \geq T_1/p \quad \text{Arbeitsgesetz}$$

$$T_p \geq T_\infty \quad \text{Zeitspannengesetz}$$



Greedy Scheduler

Greedy Scheduler: teilt zu jeder Zeit so viele Tasks zu Prozessoren zu wie möglich.

Theorem

Auf einem idealen Parallelrechner mit p Prozessoren führt ein Greedy-Scheduler eine mehrfädige Berechnung mit Arbeit T_1 und Zeitspanne T_∞ in Zeit

$$T_p \leq T_1/p + T_\infty$$

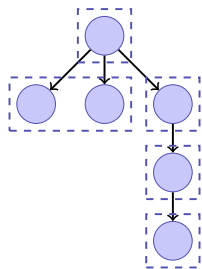
aus.

904

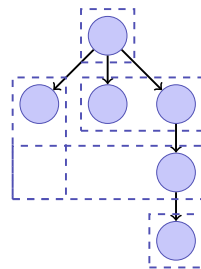
905

Beispiel

Annahme $p = 2$.



$$T_p = 5$$



$$T_p = 4$$

Beweis des Theorems

Annahme, dass alle Tasks gleich viel Arbeit aufweisen.

- Vollständiger Schritt: p Tasks stehen zur Berechnung bereit
- Unvollständiger Schritt: weniger als p Tasks bereit.

Annahme: Anzahl vollständige Schritte grösser als $\lfloor T_1/p \rfloor$. Ausgeführte Arbeit $\geq \lfloor T_1/p \rfloor \cdot p + p = T_1 - T_1 \bmod p + p > T_1$. Widerspruch. Also maximal $\lfloor T_1/p \rfloor$ vollständige Schritte.

Betrachten nun den Graphen der ausstehenden Tasks. Jeder maximale (kritische) Pfad beginnt mit einem Knoten t mit $\deg^-(t) = 0$. Jeder unvollständige Schritt führt zu jedem Zeitpunkt alle vorhandenen Tasks t mit $\deg^-(t) = 0$ aus und verringert also die Länge der Zeitspanne. Anzahl unvollständige Schritte also begrenzt durch T_∞ .

906

907

Konsequenz

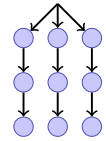
Wenn $p \ll T_1/T_\infty$, also $T_\infty \ll T_1/p$, dann $T_p \approx T_1/p$.

Beispiel Fibonacci

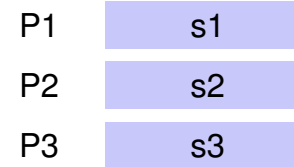
$T_1(n)/T_\infty(n) = \Theta(\phi^n/n)$. Für moderate Größen von n können schon viele Prozessoren mit linearem Speedup eingesetzt werden.

Granularität: Wie viele Tasks?

- #Tasks = #Cores?
- Problem: wenn ein Core nicht voll ausgelastet werden kann
- Beispiel: 9 Einheiten Arbeit. 3 Cores. Scheduling von 3 sequentiellen Tasks.

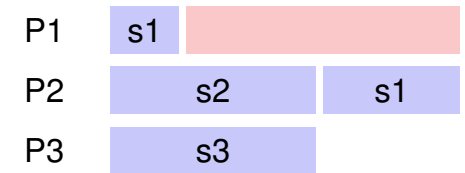


Exklusive Auslastung:



Ausführungszeit: 3 Einheiten

Fremder Thread "stört":



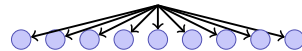
Ausführungszeit: 5 Einheiten

908

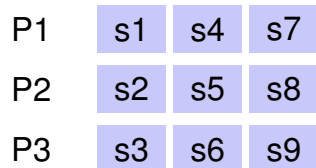
909

Granularität: Wie viele Tasks?

- #Tasks = Maximum?
- Beispiel: 9 Einheiten Arbeit. 3 Cores. Scheduling von 9 sequentiellen Tasks.

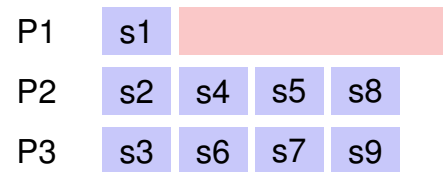


Exklusive Auslastung:



Ausführungszeit: $3 + \varepsilon$ Einheiten

Fremder Thread "stört":

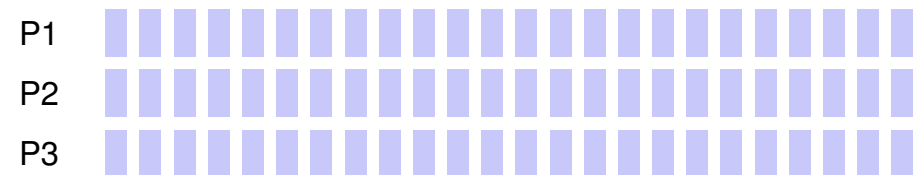


Ausführungszeit: 4 Einheiten.
Volle Auslastung.

910

Granularität: Wie viele Tasks?

- #Tasks = Maximum?
- Beispiel: 10^6 kleine Einheiten Arbeit.



Ausführungszeit: dominiert vom Overhead.

911

Granularität: Wie viele Tasks?

Antwort: so viele Tasks wie möglich mit sequentiellem Cut-off, welcher den Overhead vernachlässigen lässt.

Beispiel: Parallelität von Mergesort

- Arbeit (sequentielle Laufzeit) von Mergesort $T_1(n) = \Theta(n \log n)$.
- Span $T_\infty(n) = \Theta(n)$
- Parallelität $T_1(n)/T_\infty(n) = \Theta(\log n)$ (Maximal erreichbarer Speedup mit $p = \infty$ Prozessoren)

