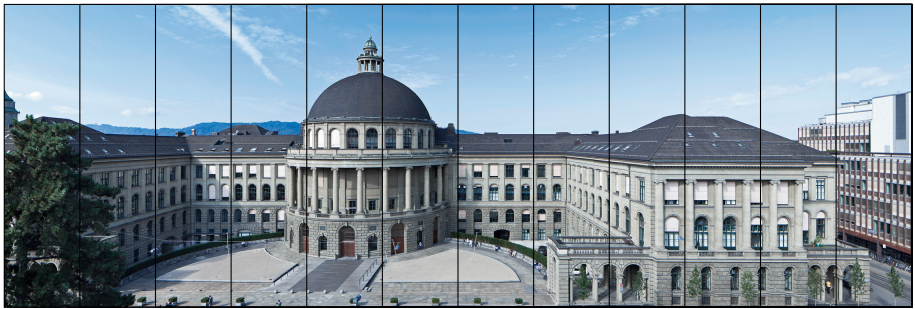
30. Parallel Programming I

Moore's Law, Multi-Threading, Parallelität und Nebenläufigkeit, C++ Threads, Skalierbarkeit: Amdahl und Gustafson Scheduling

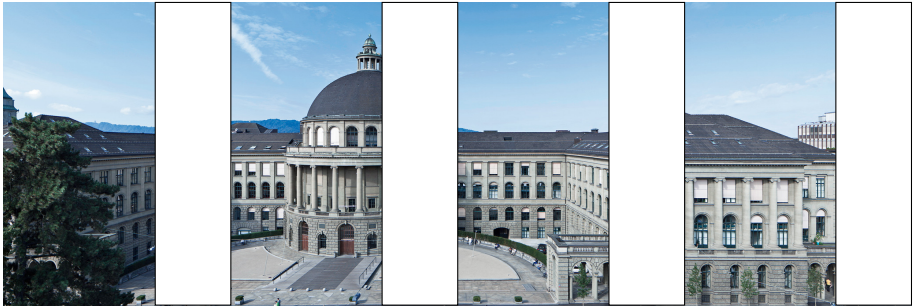
[Task-Scheduling: Cormen et al, Kap. 27] [Concurrency, Scheduling: Williams, Kap. 1.1 – 1.2]

Motivation: Ein Bild malen (1 Maler)



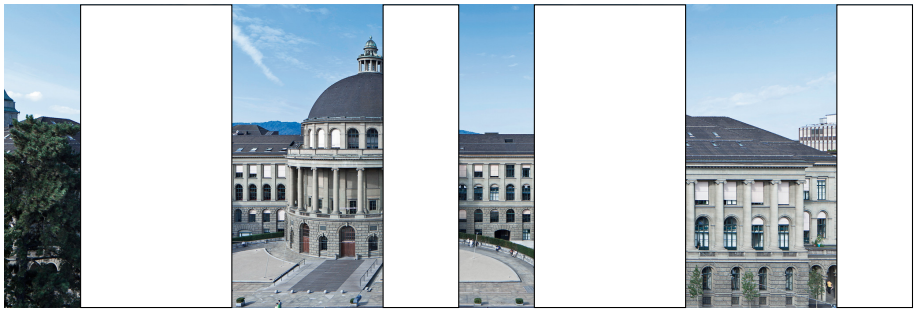
Modell: sequentielle Ausführung

Motivation: Ein Bild malen (4 Maler)



Modell: parallele Ausführung

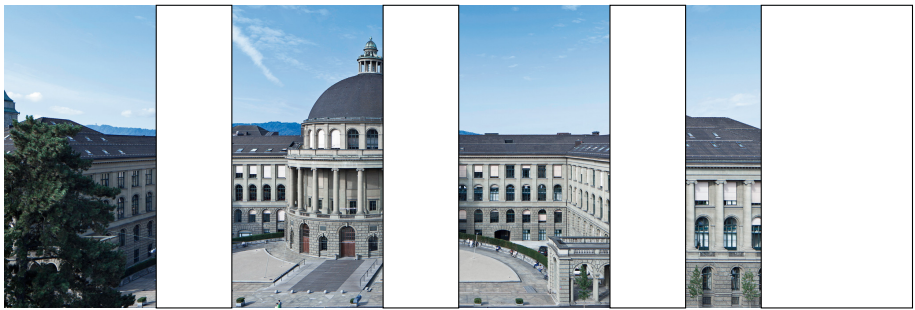
Motivation: Ein Bild malen (4 Maler, 3 Pinsel)



P P P

Modell: nebenläufige Ausführung

Motivation: Ein Bild malen (4 Maler, 1 Pinsel)



Modell: nebenläufige Ausführung

Modelle

- **Sequentiell:** das Programm wird schrittweise abgearbeitet, in vorgegebener Anordnung
- **Parallel:** Aufgaben werden parallel ausgeführt. Es besteht kein Synchronisationsbedarf und keine Resourcenknappheit.
- **Nebenläufig:** Aufgaben werden parallel ausgeführt. Es besteht aber Synchronisationsbedarf bzw. Resourcenknappheit: Aufgaben müssen unterbrochen werden.

Warum Parallelität / Concurrency

- **Reaktive / Interaktive / Mehrbenutzer- Systeme** insbesondere grafische Benutzeroberflächen. ⇒ Nebenläufigkeit
- Aufwändige Berechnungen: Datenverarbeitung- und -Analyse, Machine-Learning, **Performanz** ist wichtig. ⇒ Parallelität
- **Natürliche Parallelität / Nebenläufigkeit**: verteilte Systeme, Gerätetreiber.

Technischer Hintergrund: CPUs

Heutige Computer (Desktops, Telefone, ...) haben typischerweise (mindestens)

- **CPU**: zentrale Recheneinheit, Allzweckrecheneinheit
- GPU: Grafikverarbeitungseinheit, unglaublich effizient für lineare Algebra-Berechnungen (Spiele, Grafik, maschinelles Lernen).

Heutige CPUs sind in der Regel mehrkernig (**multi-core**):

- Jeder Kern ist im Grunde eine eigene CPU, die Code ausführen kann.
- Beispiele:
  - Intel i7-8700K hat 6 Kerne (von 2017)
  - Intel i9-12900KF hat 16 Kerne (ab 2021)
  - AMD Ryzen 9 5950X hat 16 Kerne (ab 2020)
  - Apple M1 Max hat 10 Kerne (ab 2021)

Intel i7 (2017)

Moore’s Law

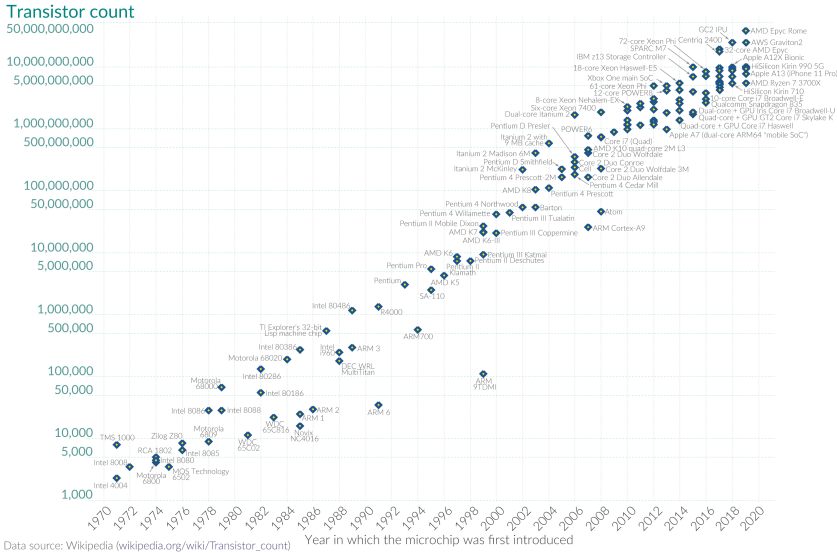
Beobachtung von Gordon E. Moore:
Die Anzahl Transistoren in integrierten Schaltkreisen verdoppelt sich ungefähr alle zwei Jahre.

Gordon E. Moore (1929)

Moore's Law: The number of transistors on microchips doubles every two years

Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.



Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

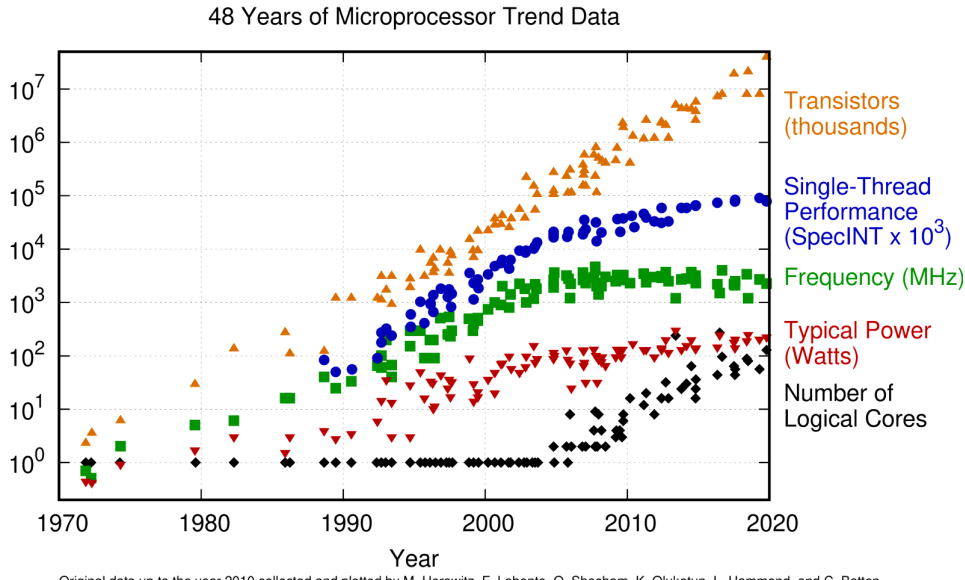
OurWorldinData.org – Research and data to make progress against the world's largest problems.

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 Rechnereneration und ihre Programme wurden schneller.

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)



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 2010-2019 by K. Rupp

https://github.com/karlrupp/microprocessor-trend-data

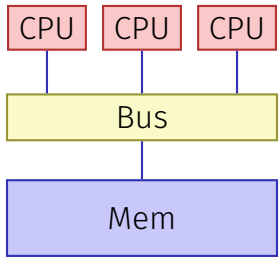
Multicore

- Verwende die Transistoren für mehr Rechenkerne
- Parallelität in der Software

⇒ Programmierer müssen **parallele Programme** schreiben, um die neue Hardware vollständig ausnutzen zu können

(Vereinfachte) Annahme: Rechenmodell

Unabhängige Rechenkerne



Gemeinsamer Speicher

30.1 Multi-Threading, Parallelität und Nebenläufigkeit

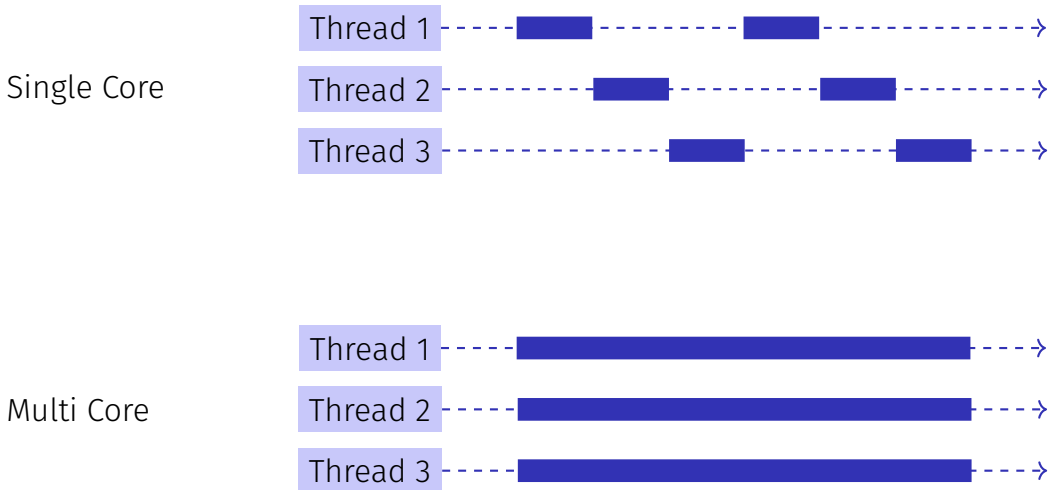
Prozesse und Threads

- **Prozess**: Instanz eines Programmes
  - jeder Prozess hat seinen eigenen Kontext, sogar eigenen Addresraum („sieht nur seinen eigenen Speicher”)
  - OS verwaltet Prozesse (Resourcenkontrolle, Scheduling, Synchronisierung)
- **Thread**: Ausführungsfaden eines Programmes
  - Threads teilen sich einen Addressraum
  - Schneller Kontextwechsel zwischen Threads

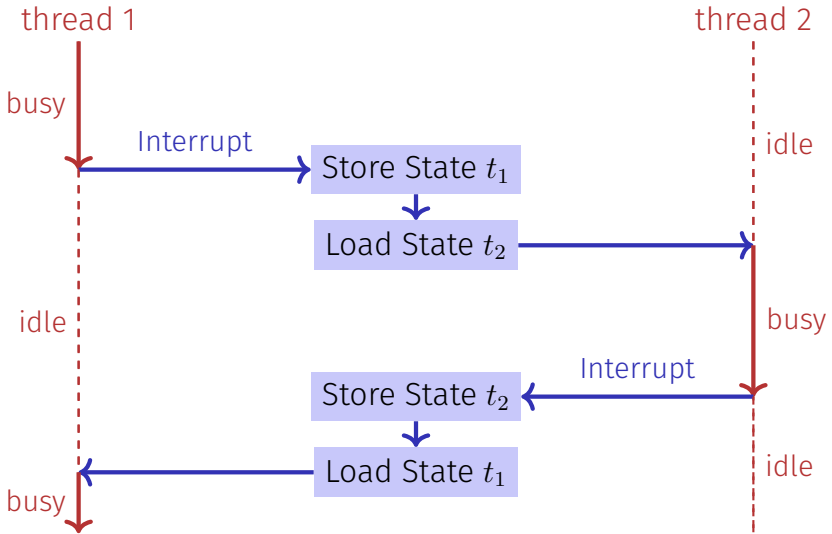
Warum Multithreading?

- Verhinderung vom “Polling” auf Resourcen (Files, Netwerkzugriff, Tastatur)
- Interaktivität (z.B. Responsivität von GUI Programmen)
- Mehrere Applikationen / Clients gleichzeitig instanzierbar
- Parallelität (Performanz!)

Multithreading konzeptuell

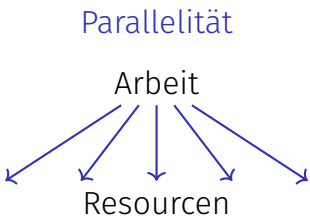


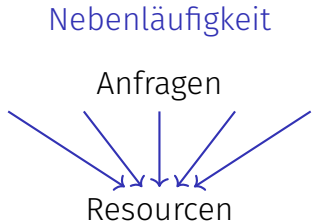
Threadwechsel auf einem Core (Preemption)



Parallelität vs. Nebenläufigkeit (Concurrency)

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





Thread-Sicherheit

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

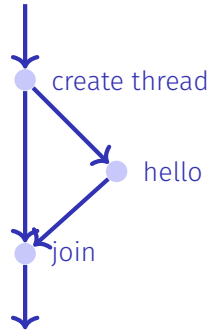
Viele Optimierungen (Hardware, Compiler) sind darauf ausgerichtet, dass sich ein *sequentielles* Programm korrekt verhält.

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

30.2 C++ Threads

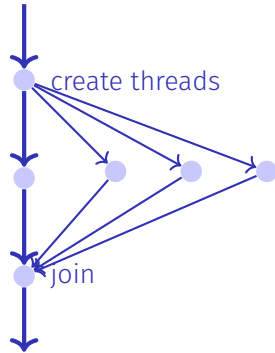
C++11 Threads

```
#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;
}
```



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;
}
```



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
```

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
```

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.

Was kann man (nicht) parallelisieren?

```
int pow8(int b){
  int b2 = b * b;
  int b4 = b2 * b2;
  return b4 * b4;
}

int main(){
  int x;
  std::cin >> x;
  x = pow8(x);
  std::cout << x;
}
```

- Programm berechnet x^8
- Alle Teile des Programmes müssen in einer festen Reihenfolge ausgeführt werden
  - Eingabe $\rightarrow$ Berechnung $\rightarrow$ Ausgabe
  - b2 $\rightarrow$ b4 $\rightarrow$ b8
- Berechnungen sind alle abhängig
- und müssen sequentiell ausgeführt werden

Was kann man (nicht) parallelisieren?

```
int pow8(int b){
  int b2 = b * b;
  int b4 = b2 * b2;
  return b4 * b4;
}

int main(){
  std::vector<int> v = ....;
  for (int& x : v)
    x = pow8(x);
  ...
}
```

- Programm berechnet x^8 für jedes $x \in v$
- Die Berechnung von x_i^8 hängt nicht von der Berechnung von x_j^8 ab.
- Wir können die Berechnung aller x^8 parallelisieren
- Parallelisierung *kann* die Laufzeit reduzieren wenn
  - die parallelisierte Berechnung lange genug dauert
  - genügend CPUs verfügbar sind

Was kann man (nicht) parallelisieren?

```
int main(){
  std::vector<int> v = ....;
  for (int& x : v)
    x = pow8(x);
  ...
}
```

- Beispiel ist offensichtlich parallelisierbar
- Solche Probleme nennt man **embarrassingly parallel** (beschämend parallel ☺)

- Viele Probleme und Algorithmen sind leider anders:
  - Können parallelisiert werden, aber eine tiefergehende Analyse ist nötig.
  - Müssen vor- bzw- nachbearbeitet werden, um das Problem in parallelisierbare Teilprobleme zu zerlegen bzw. um Teillösungen wieder zu kombinieren
- Beispiele: Matrix-Multiplikation, Mergesort

30.3 Skalierbarkeit: Amdahl und Gustafson

Skalierbarkeit

In der parallelen Programmierung:

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

Parallele Performanz

Gegeben fixierte Rechenarbeit W (Anzahl Rechenschritte)
T_1: Sequentielle Ausführungszeit sei
T_p: Parallele Ausführungszeit 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:

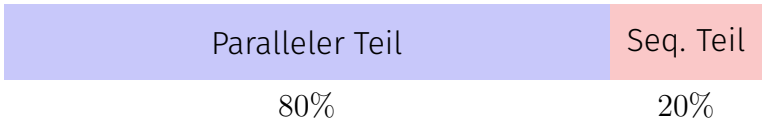
$$S_p = \frac{W/T_p}{W/T_1} = \frac{T_1}{T_p}.$$

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

Effizienz:$E_p = S_p/p$

Erreichbarer Speedup?

Paralleles Programm



$$T_1 = 10$$

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

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

Amdahl's Law: Zutaten

Zu leistende Rechenarbeit W fällt in zwei Kategorien

- Parallelisierbarer Teil W_p
- Nicht paralleliserbarer, sequentieller Teil W_s

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

$$T_1 = W_s + W_p$$
$$T_p \geq W_s + W_p/p$$

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

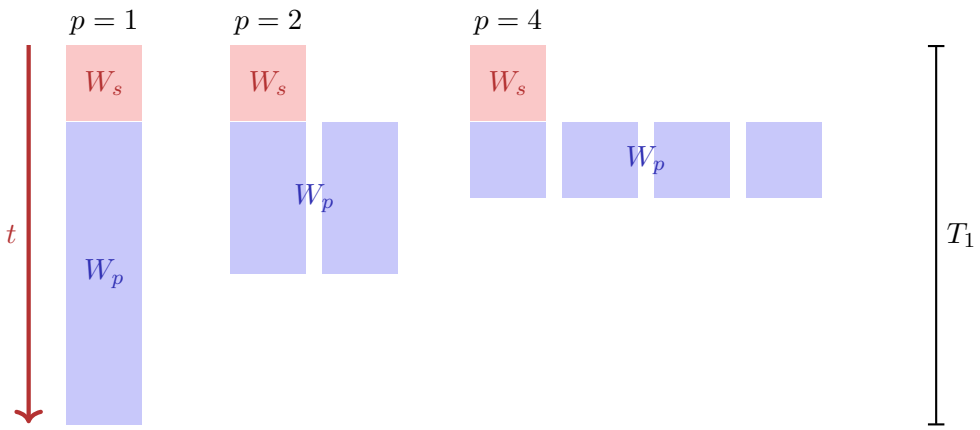
Mit seriellem, nicht parallelisierbaren Anteil λ: $W_s = \lambda W$, $W_p = (1-\lambda)W$:

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

Somit

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

Illustration Amdahl's Law



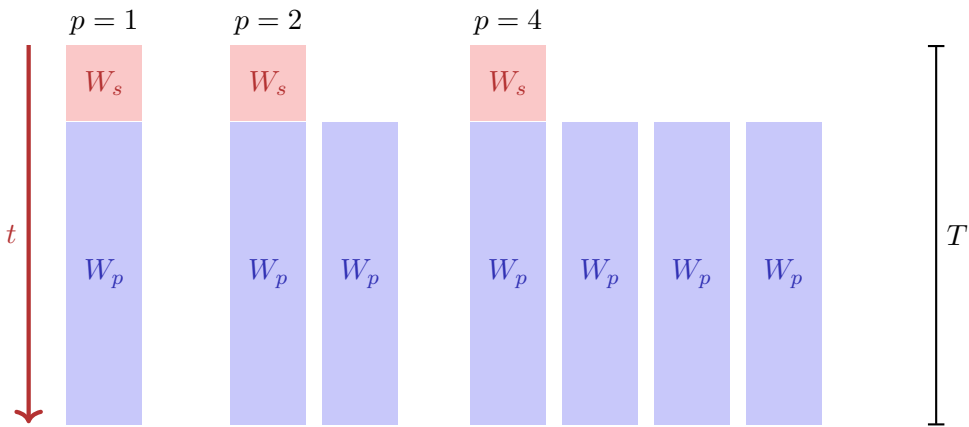
Amdahl’s Law ist keine gute Nachricht

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



Gustafson's Law

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

$$W_s + W_p = T$$

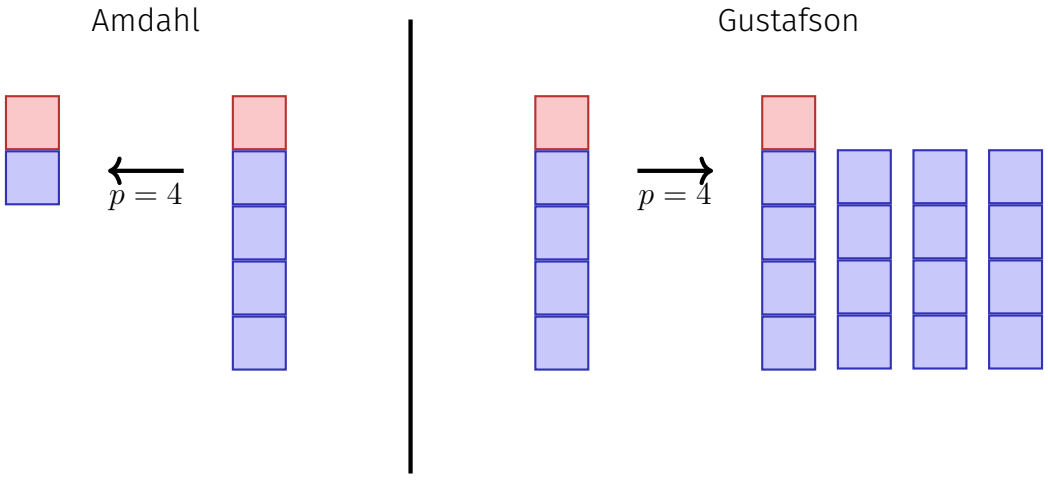
Arbeit, die mit p Prozessoren in der Zeit T erledigt werden kann:

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

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 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).

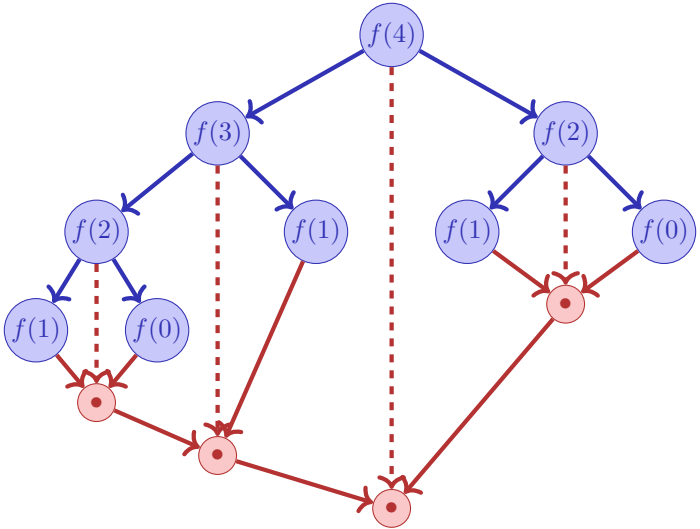
Die beiden Modelle widersprechen sich nicht, sondern beschreiben die Laufzeitverbesserung verschiedener Probleme und Algorithmen.

30.4 Scheduling

Beispiel: Fibonacci

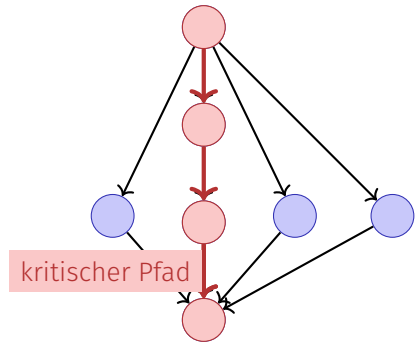
```
int fib_task(int x){
    if (x < 2) {
      return x;
  } else {
      auto f1 = std::async(fib_task, x-1);
      auto f2 = std::async(fib_task, x-2);
      return f1.get() + f2.get();
    }
}
```

Task-Graph



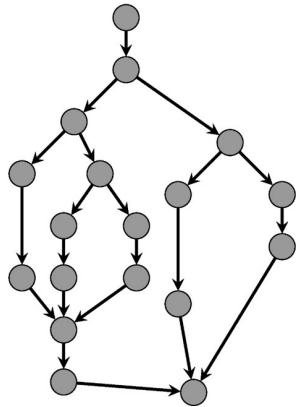
Frage

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



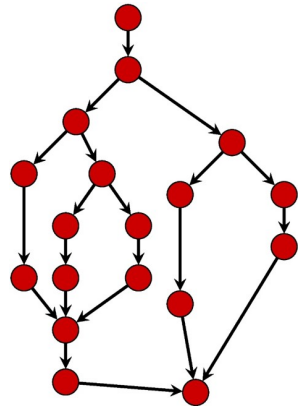
Performanzmodell

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



Performanzmodell

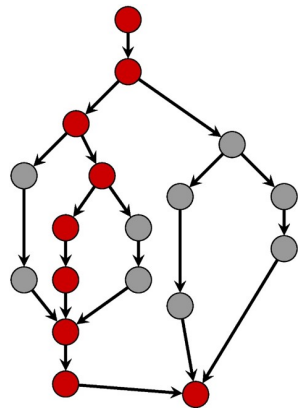
- T_p: Ausführungszeit auf p Prozessoren
- T_1: **Arbeit:** Zeit für die gesamte Berechnung auf einem Prozessor
- T_1/T_p: Speedup



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 38

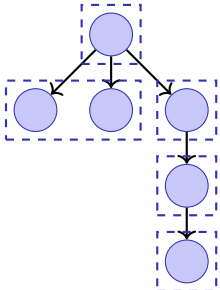
Auf einem idealen Parallelrechner mit p Prozessoren führt ein Greedy-Scheduler eine mehrfädige Berechnung mit Arbeit T_1 und Zeitspanne T_∞ in Zeit

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

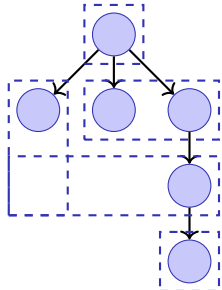
aus.

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 $\geq \lfloor T_1/p \rfloor \cdot p + p = T_1 - T_1 \mod p + p > T_1$. 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_∞.

Konsequenz

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

$$T_p \leq T_1/p + T_\infty \quad \Rightarrow \quad T_p \lesssim T_1/p$$

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.

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)