27. Parallel Programming I

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

[Task-Scheduling: Cormen et al, Kap. 27]

The Free Lunch

The free lunch is over [36]

[36] "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

Microprocessor Transistor Counts 1971-2011 & Moore's Law

Transistor count: 2,600,000,000; 1,000,000,000; 100,000,000; 10,000,000; 1,000,000; 100,000; 10,000; 2,300

curve shows transistor count doubling every two years

Date of introduction: 1971; 1980; 1990; 2000; 2011

By Wgsimon, https://commons.wikimedia.org/w/index.php?curid=15193542

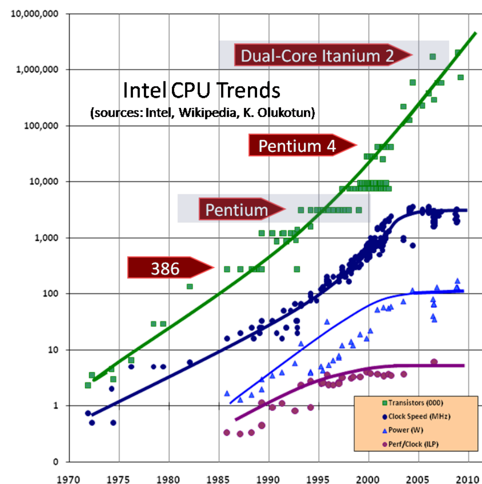
Für eine lange Zeit...

- 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)

Trends



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

Multicore

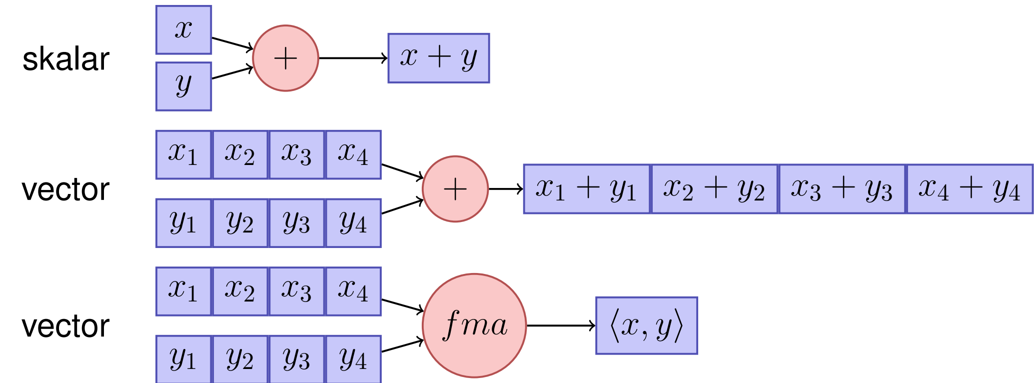
- Verwende die Transistoren für mehr Rechenkerne
- Parallelität in der Software
- Implikation: Programmierer müssen parallele Programme schreiben, um die neue Hardware vollständig ausnutzen zu können

Formen der Parallelen Ausführung

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

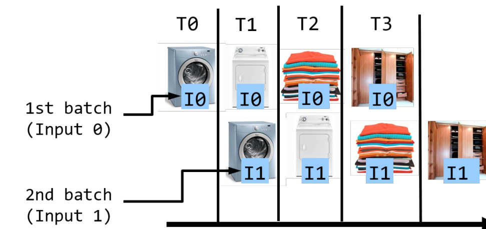
Vektorisierung

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

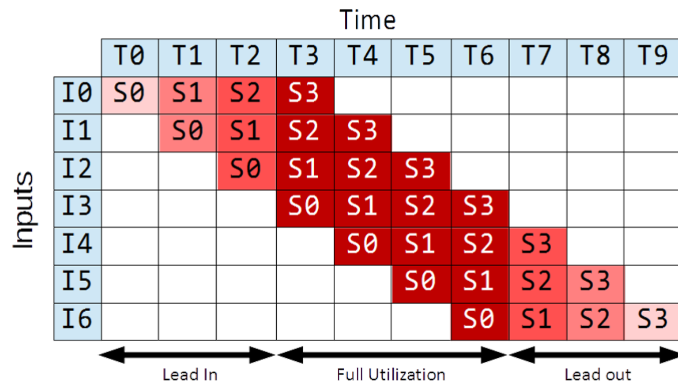


Hausarbeit

Effizienter



Pipeline



Throughput (Durchsatz)

- Throughput = Rate der ein- oder ausgehenden Daten
- Anzahl Operationen pro Zeiteinheit
- Je grösser, desto besser
- Approximation

$$\text{throughput} = \frac{1}{\max(\text{Berechnungszeit}(\text{Stufen}))}$$

ignoriert lead-in und lead-out Zeiten

Latenz

- Zeit zum Ausführen einer Berechnung
- Pipeline-Latenz ist nur konstant, wenn die Pipeline balanciert ist: Summe aller Operationen über die Stufen
- Unbalancierte Pipeline
  - Erster Durchlauf wie bei der balancierten Pipeline
  - Balancierte Version, Latenz= $\#\text{stufen} \cdot \max(\text{Berechnungszeit}(\text{Stufen}))$

Beispiel Hausarbeit

Waschen $T_0 = 1h$, Trocknen $T_1 = 2h$, Bügeln $T_2 = 1h$, Versorgen $T_3 = 0.5h$

- Latenz Erster Durchlauf: $L = T_0 + T_1 + T_2 + T_3 = 4.5h$
- Latenz Zweiter Durchlauf: $L = T_1 + T_1 + T_2 + T_3 = 5.5h$
- Langfristiger Durchsatz: 1 Ladung alle $2h$ $(0.5/h)$.

Throughput vs. Latency

- Erhöhen des Throughputs kann Latenz erhöhen
- Stufen der Pipeline müssen kommunizieren und synchronisieren: Overhead

Pipelines in CPUs

Fetch | Decode | Execute | Data Fetch | Writeback

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

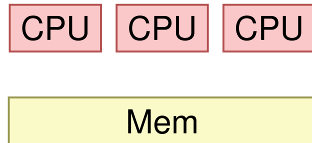
Moderne CPUs führen unabhängige Instruktionen intern auf mehreren Einheiten parallel aus

- Pipelining
- Superscalar CPUs (multiple instructions per cycle)
- Out-Of-Order Execution (Programmer observes the sequential execution)
- Speculative Execution

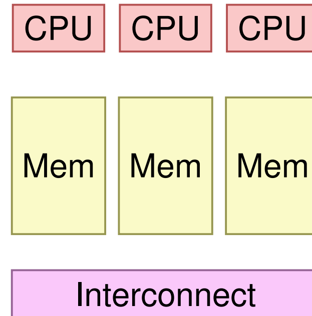
27.2 Hardware Architekturen

Gemeinsamer vs. verteilter Speicher

Gemeinsamer Speicher



Verteilter Speicher



Shared vs. Distributed Memory Programming

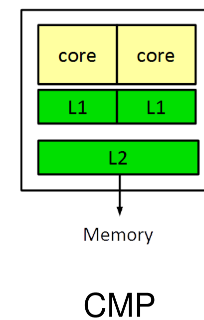
- Kategorien des Programmierinterfaces
  - Kommunikation via Message Passing
  - Kommunikation via geteiltem Speicher
- Es ist möglich:
  - Systeme mit gemeinsamen Speicher als verteilte Systeme zu programmieren (z.B. mit Message Passing Interface MPI)
  - Systeme mit verteiltem Speicher als System mit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

Architekturen mit gemeinsamen Speicher

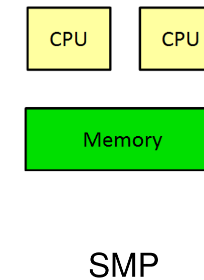
- Multicore (Chip Multiprocessor - CMP)
- Symmetric Multiprocessor Systems (SMP)
- Simultaneous Multithreading (SMT = Hyperthreading)
  - ur ein physischer Kern, Mehrere Instruktionsströme/Threads: mehrere virtuelle Kerne
  - Zwischen ILP (mehrere Units für einen Strom) und Multicore (mehrere Units für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

Gleiches Programmierinterface!

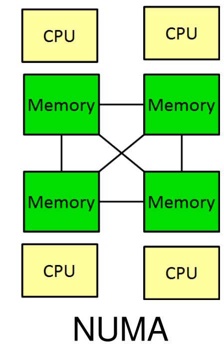
Übersicht



CMP



SMP

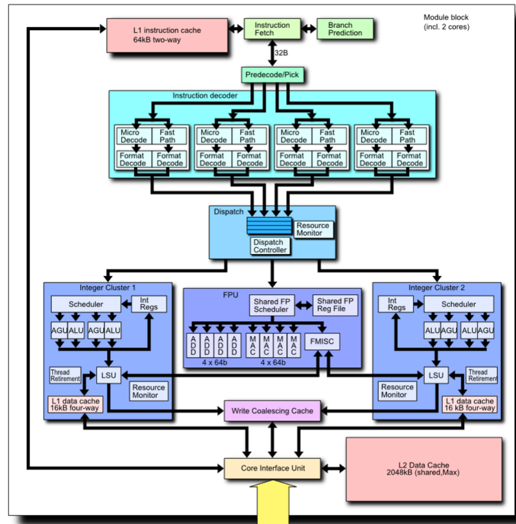


NUMA

Ein Beispiel

AMD Bulldozer: Zwischen CMP und SMT

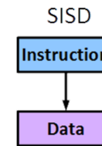
- 2x integer core
- 1x floating point core



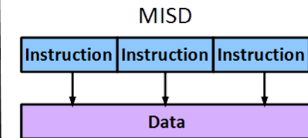
Wikipedia

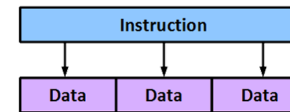
Klassifikation nach Flynn

Single-Core

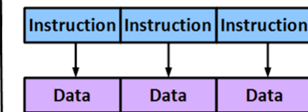


Fault-Tolerance





Vector Computing / GPU

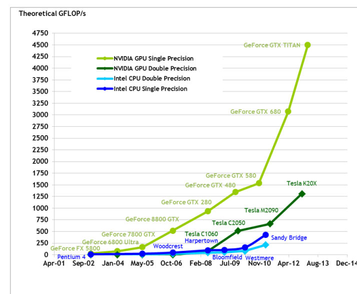


Multi-Core

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



27.3 Multi-Threading, Parallelität und Nebenläufigkeit

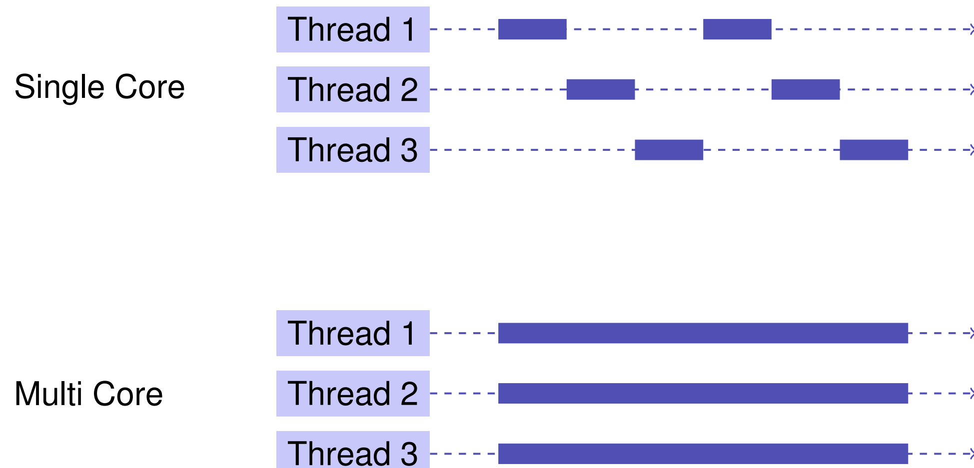
Prozesse und Threads

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

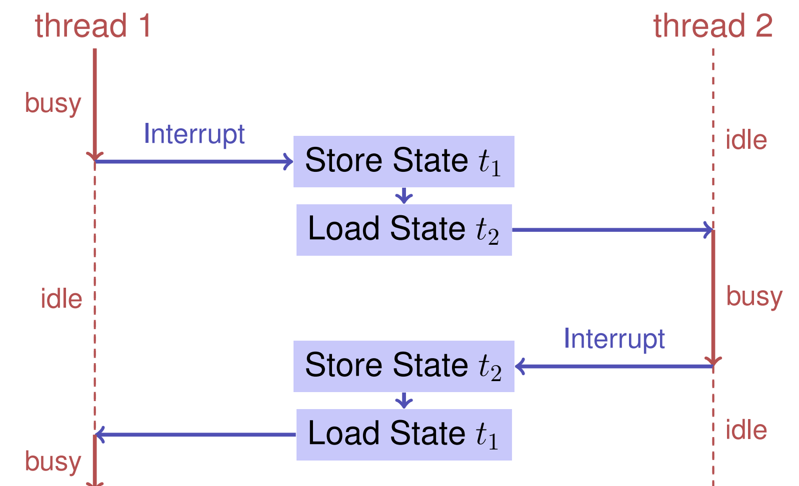
Warum Multithreading?

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

Multithreading konzeptuell

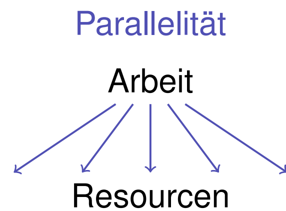


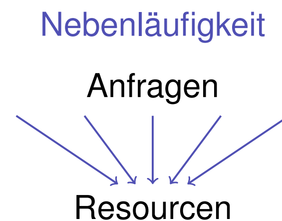
Threadwechsel auf einem Core (Preemption)



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

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 eingesetzen System abhängig.

27.4 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)

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 $T_p > T_1/p$ (der übliche Fall)
- Hexerei: superlinearer Speedup $T_p < T_1/p$

Effizienz:$E_p = S_p/p$

Erreichbarer Speedup?

Paralleles Programm

Paralleler Teil	Seq. Teil
80%	20%

$$T_1 = 10$$
$$T_8 = ?$$
$$T_8 = \frac{10 \cdot 0.8}{8} + 10 \cdot 0.2 = 1 + 2 = 3$$
$$S_8 = \frac{T_1}{T_8} = \frac{10}{3} = 3.33$$

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

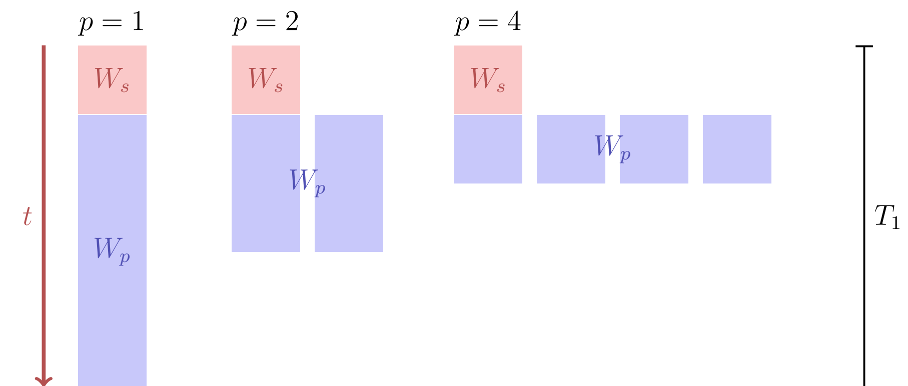
Mit seriellem, nicht parallelisierbaren Anteil λ: $W_s = \lambda W$, $W_p = (1-\lambda)W$:

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

Somit

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

Illustration Amdahl's Law



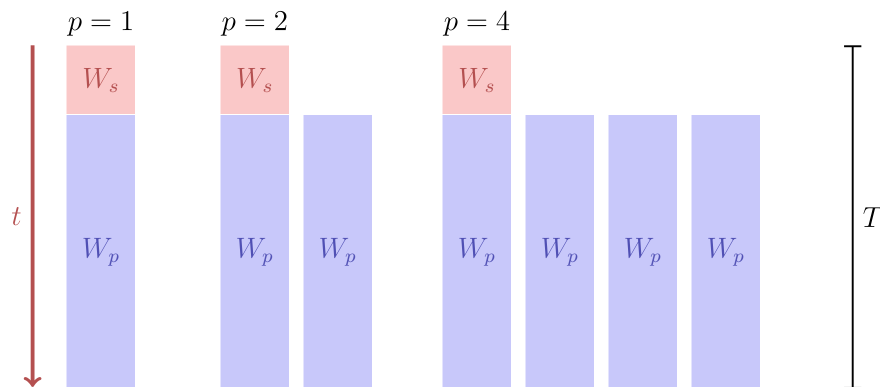
Amdahl's Law ist keine gute Nachricht

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:

$$W_s + W_p = T$$

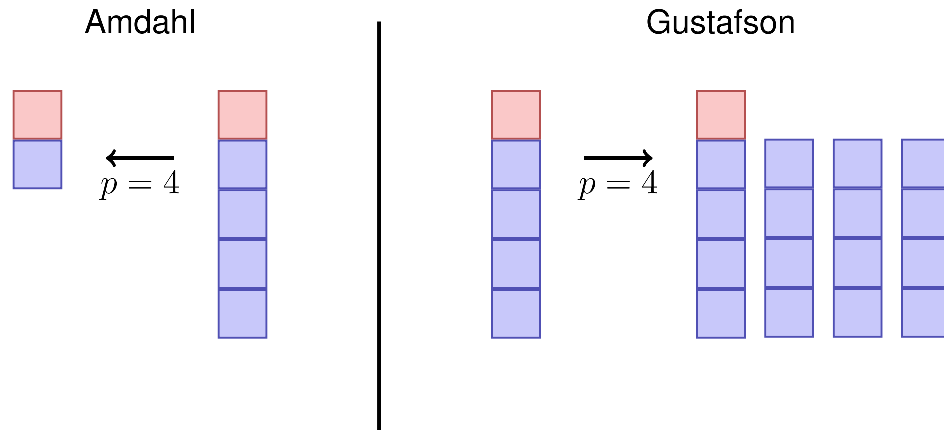
Arbeit, die mit p Prozessoren in der Zeit T erledigt werden kann:

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

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



27.5 Task- und Datenparallelität

Paradigmen der Parallelen Programmierung

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

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;
```

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);
}
```

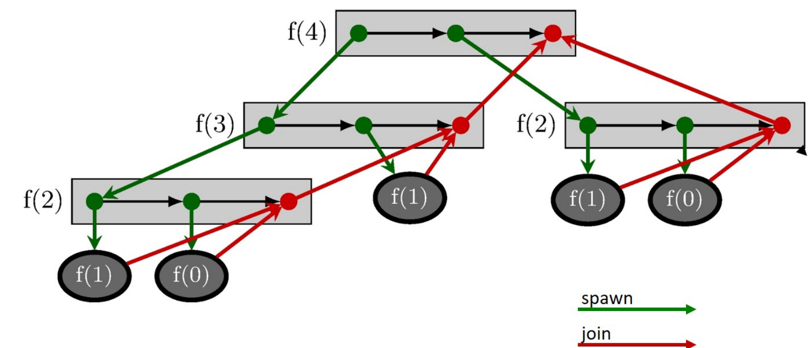

Partitionierung und Scheduling

- 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 Resourcennutzung bei wenig Overhead

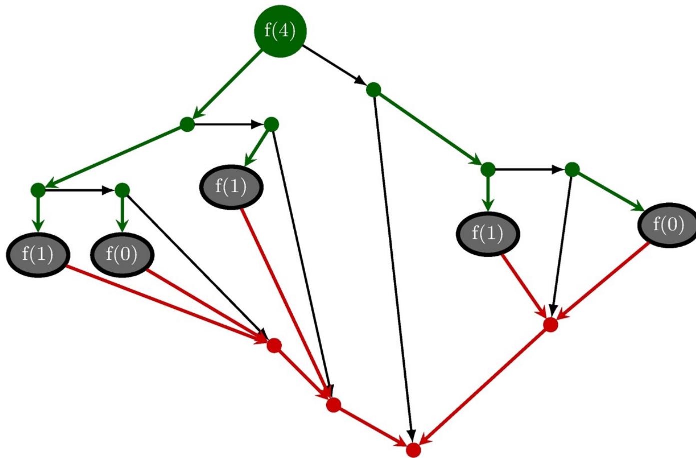
Beispiel: Fibonacci P-Fib

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

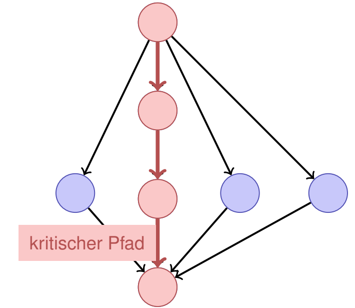


P-Fib Task Graph



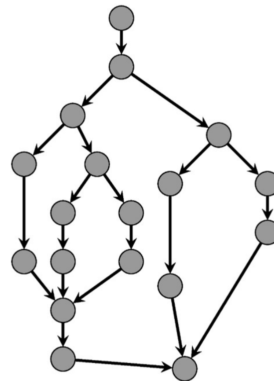
Frage

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



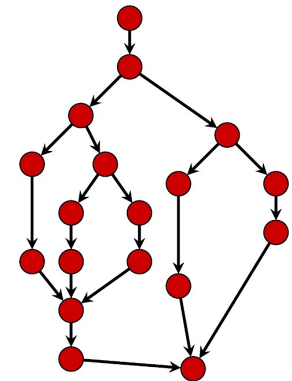
Performanzmodell

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



Performanzmodell

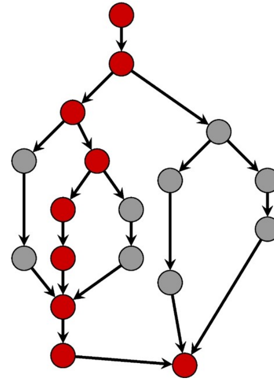
- T_p: Ausführungszeit auf p Prozessoren
- T_1: *Arbeit:* Zeit für die gesamte Berechnung auf einem Prozessor
- T_1/T_p: Speedup



Performanzmodell

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

$T_p \geq T_1/p$ Arbeitsgesetz
$T_p \geq T_\infty$ Zeitspannengesetz



Greedy Scheduler

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

Theorem

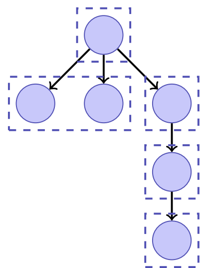
Auf einem idealen Parallelrechner mit p Prozessoren führt ein Greedy-Scheduler eine mehrfädige Berechnung mit Arbeit T_1 und Zeitspanne T_∞ in Zeit

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

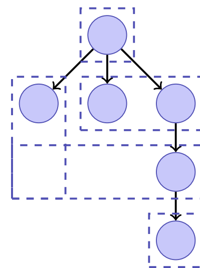
aus.

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 $\geq P \cdot (\lfloor T_1/p \rfloor \cdot p) = T_1 - T_1 \mod p + p \geq T_1$. Widerspruch. Also maximal $\lfloor T_1/p \rfloor$ vollständige Schritte.

Jeder unvollständige Schritt führt zu jedem Zeitpunkt alle vorhandenen Tasks t mit $\deg^-(t) = 0$ aus und verringert die Länge der Zeitspanne. Andernfalls wäre die gewählte Zeitspanne nicht maximal. Anzahl unvollständige Schritte also maximal T_∞.

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:

P1	s1
P2	s2
P3	s3

Ausführungszeit: 3 Einheiten

Fremder Thread "stört":

P1	s1	
P2	s2	s1
P3	s3	

Ausführungszeit: 5 Einheiten

Granularität: Wie viele Tasks?

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

Exklusive Auslastung:

P1	s1	s4	s7
P2	s2	s5	s8
P3	s3	s6	s9

Ausführungszeit: $3 + \varepsilon$ Einheiten

Fremder Thread "stört":

P1	s1			
P2	s2	s4	s5	s8
P3	s3	s6	s7	s9

Ausführungszeit: 4 Einheiten. Volle Auslastung.

Granularität: Wie viele Tasks?

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

P1

P2

P3

Ausführungszeit: dominiert vom Overhead.

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)