Datenstrukturen und Algorithmen

Übung 12

FS 2018

Programm von heute

1. Feedback letzte Übung
2. Wiederholung Theorie
3. Programmieraufgaben

1. Feedback letzte Übung

Fussballmeisterschaft

	Verein	Punkte	Gegner
1)	FC St. Gallen (FCSG)	37	FCB, FCW
2)	BSC Young Boys (YB)	36	FCW, FCB
3)	FC Basel (FCB)	35	FCSG, YB
4)	FC Luzern (FCL)	33	FCZ, GCZ
5)	FC Winterthur (FCW)	31	YB, FCSG

Fussballmeisterschaft

	Verein	Punkte	Gegner
1)	FC St. Gallen (FCSG)	37	FCB, FCW
2)	BSC Young Boys (YB)	36	FCW, FCB
3)	FC Basel (FCB)	35	FCSG, YB
4)	FC Luzern (FCL)	33	FCZ, GCZ
5)	FC Winterthur (FCW)	31	YB, FCSG

Alte *Zwei-Punkte-Regel*

In jedem Spiel werden genau 2 Punkte vergeben: $2+0, 1+1, 0+2$

Fussballmeisterschaft

	Verein	Punkte	Gegner
1)	FC St. Gallen (FCSG)	37	FCB, FCW
2)	BSC Young Boys (YB)	36	FCW, FCB
3)	FC Basel (FCB)	35	FCSG, YB
4)	FC Luzern (FCL)	33	FCZ, GCZ
5)	FC Winterthur (FCW)	31	YB, FCSG

Alte *Zwei-Punkte-Regel*

In jedem Spiel werden genau 2 Punkte vergeben: $2+0, 1+1, 0+2$

Frage: Kann der FC Luzern noch Meister werden?

Fussballmeisterschaft

	Verein	Punkte	Gegner	höchstens...
1)	FC St. Gallen (FCSG)	37	FCB, FCW	
2)	BSC Young Boys (YB)	36	FCW, FCB	
3)	FC Basel (FCB)	35	FCSG, YB	
4)	FC Luzern (FCL)	33	FCZ, GCZ	
5)	FC Winterthur (FCW)	31	YB, FCSG	

Alte *Zwei-Punkte-Regel*

In jedem Spiel werden genau 2 Punkte vergeben: $2+0, 1+1, 0+2$

Frage: Kann der FC Luzern noch Meister werden?

unter der **Annahme:** Der FCL gewinnt beide Spiele $\rightarrow 37p$.

Fussballmeisterschaft

	Verein	Punkte	Gegner	höchstens...
1)	FC St. Gallen (FCSG)	37	FCB, FCW	$+0$ Punkte
2)	BSC Young Boys (YB)	36	FCW, FCB	$+1$ Punkt
3)	FC Basel (FCB)	35	FCSG, YB	$+2$ Punkte
4)	FC Luzern (FCL)	33	FCZ, GCZ	
5)	FC Winterthur (FCW)	31	YB, FCSG	

Alte *Zwei-Punkte-Regel*

In jedem Spiel werden genau 2 Punkte vergeben: $2+0, 1+1, 0+2$

Frage: Kann der FC Luzern noch Meister werden?
unter der **Annahme:** Der FCL gewinnt beide Spiele $\rightarrow 37p$.

Fussballmeisterschaft

	Verein	Punkte	Gegner	höchstens...
1)	FC St. Gallen (FCSG)	37	FCB, FCW	$+0$ Punkte
2)	BSC Young Boys (YB)	36	FCW, FCB	$+1$ Punkt
3)	FC Basel (FCB)	35	FCSG, YB	$+2$ Punkte
4)	FC Luzern (FCL)	33	FCZ, GCZ	
5)	FC Winterthur (FCW)	31	YB, FCSG	egal

Alte *Zwei-Punkte-Regel*

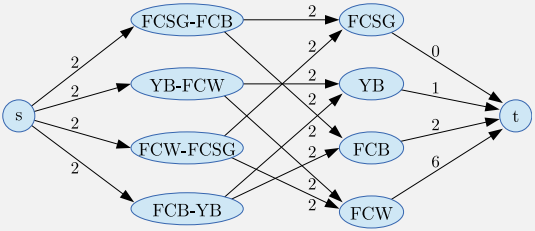
In jedem Spiel werden genau 2 Punkte vergeben: $2+0, 1+1, 0+2$

Frage: Kann der FC Luzern noch Meister werden?

unter der **Annahme:** Der FCL gewinnt beide Spiele $\rightarrow 37p$.

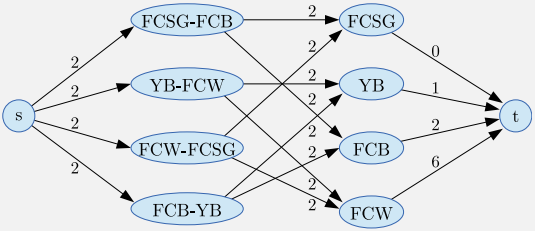
Fussballmeisterschaft

Annahme: Der FCL kann noch Meister werden.



Fussballmeisterschaft

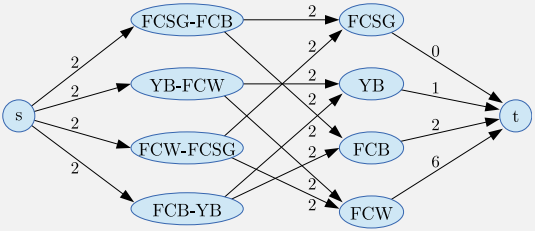
Annahme: Der FCL kann noch Meister werden.

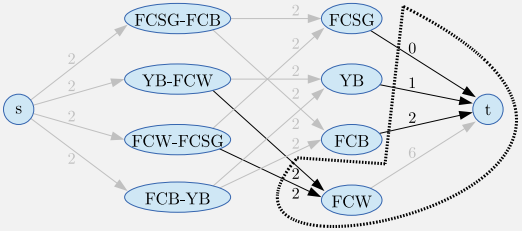


4 Spiele $\Rightarrow$ Fluss muss Grösse 8 haben.

Fussballmeisterschaft

Annahme: Der FCL kann noch Meister werden.

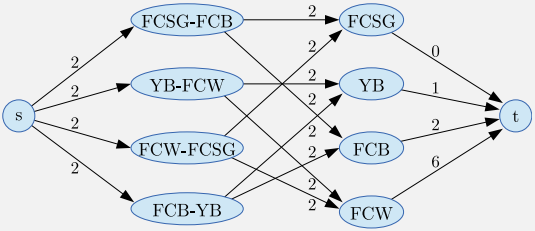


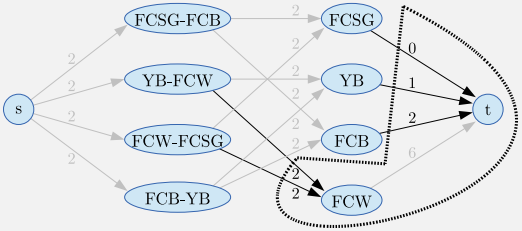


4 Spiele $\Rightarrow$ Fluss muss Grösse 8 haben.

Fussballmeisterschaft

Annahme: Der FCL kann noch Meister werden.





4 Spiele $\Rightarrow$ Fluss muss Grösse 8 haben.

Aber: MinCut ist 7. $\Rightarrow$ **Widerspruch**.

2. Wiederholung Theorie

Speedup, Performanz und Effizienz

Gegeben

- fixierte Rechenarbeit W (Anzahl Rechenschritte)
- Sequentielle Ausführungszeit sei T_1
- Parallele Ausführungszeit T_p auf p CPUs

	Ausführungszeit	Speedup	Effizienz
Perfektion (linear)	$T_p = T_1/p$	$S_p = p$	$E_p = 1$
Verlust (sublinear)	$T_p > T_1/p$	$S_p < p$	$E_p < 1$
Hexerei (superlinear)	$T_p < T_1/p$	$S_p > p$	$E_p > 1$

Gustafson's Law

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

Gustafson's Law

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

$$W_s + W_p = T$$

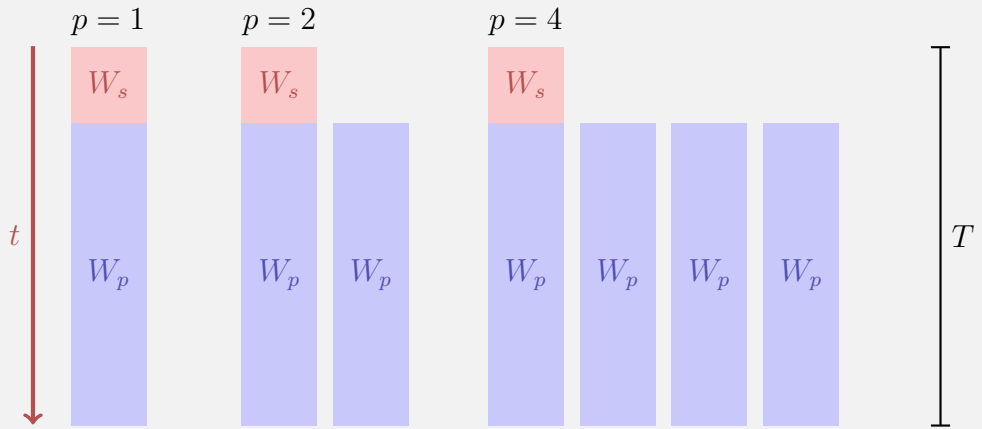
Arbeit, die mit p Prozessoren in der Zeit T erledigt werden kann:

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

Speedup:

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

Illustration Gustafson's Law



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

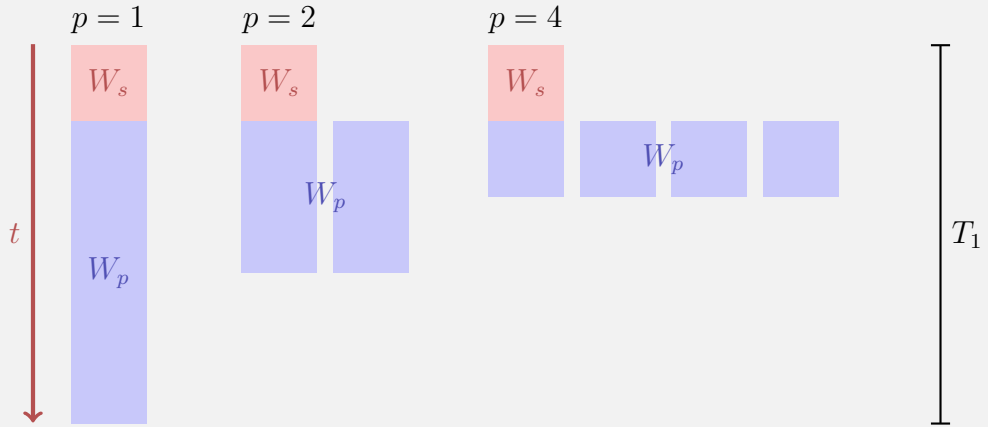
Mit seriellem, nicht parallelisierbaren Anteil λ: $W_s = \lambda W$, $W_p = (1 - \lambda)W$:

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

Somit

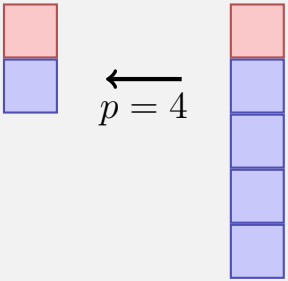
$$S_\infty \leq \frac{1}{\lambda}$$

Illustration Amdahl's Law

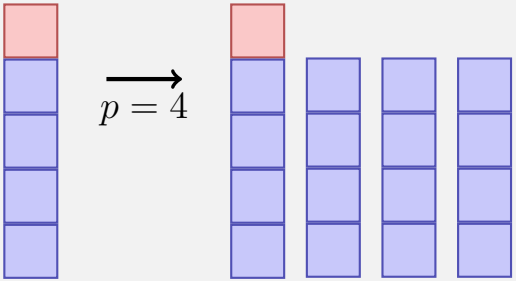


Amdahl vs. Gustafson

Amdahl



Gustafson



Amdahl vs. Gustafson, or why do we care?

Amdahl	**Gustafson**
Pessimist	Optimist
starke Skalierung	schwache Skalierung

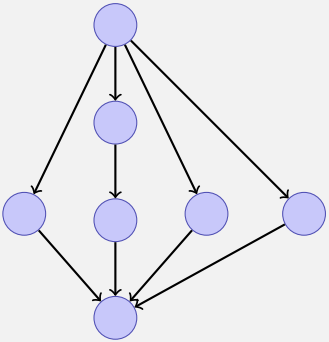
Amdahl vs. Gustafson, or why do we care?

Amdahl	**Gustafson**
Pessimist	Optimist
starke Skalierung	schwache Skalierung

$\Rightarrow$ Methoden müssen entwickelt werden so dass sie einen möglichst kleinen sequenziellen Anteil haben.

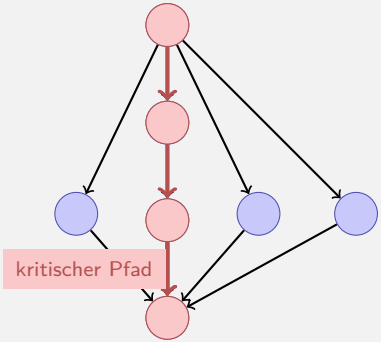
Frage

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



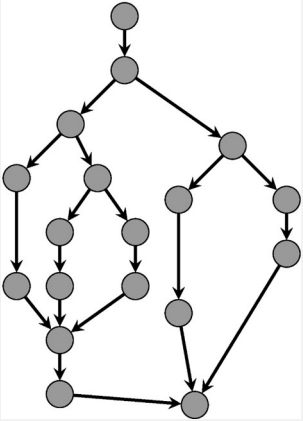
Frage

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



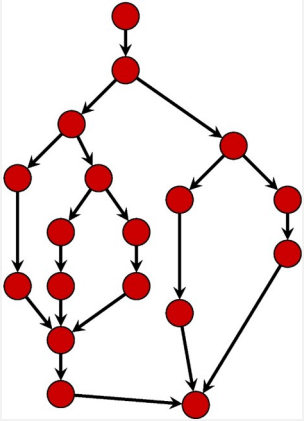
Performanzmodell

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



Performanzmodell

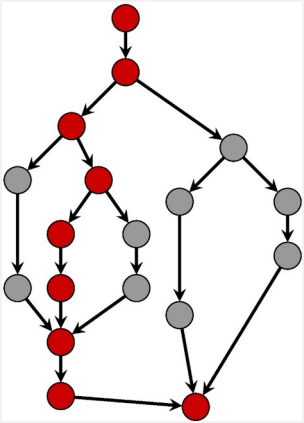
- T_p: Ausführungszeit auf p Prozessoren
- T_1: *Arbeit:* Zeit für die gesamte Berechnung auf einem Prozessor
- T_1/T_p: Speedup



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

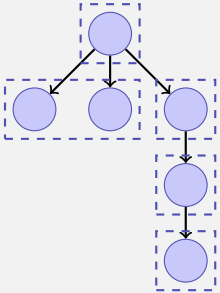
Auf einem idealen Parallelrechner mit p Prozessoren führt ein Greedy-Scheduler eine mehrfädige Berechnung mit Arbeit T_1 und Zeitspanne T_∞ in Zeit

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

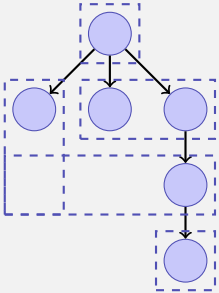
aus.

Beispiel

Annahme $p = 2$.



$T_p = 5$

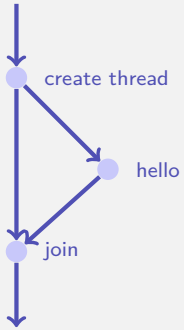


$T_p = 4$

3. Programmieraufgaben

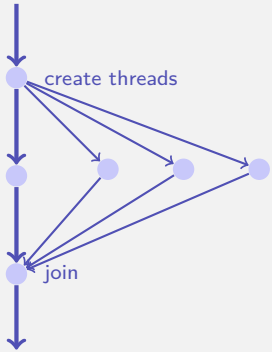
C++11 Threads

```
#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
```

Technische Details I

- Beim Erstellen von Threads werden auch Referenzparameter kopiert, ausser man gibt explizit std::ref bei der Konstruktion an.

Technische Details I

- Beim Erstellen von Threads werden auch Referenzparameter kopiert, ausser man gibt explizit std::ref bei der Konstruktion an.

```
void calc( std::vector<int>& very_long_vector ){
  // doing funky stuff with very_long_vector
}
int main(){
  std::vector<int> v( 1000000000 );
  std::thread t1( calc, v );          // bad idea, v is copied
  // here v is unchanged
  std::thread t2( calc, std::ref(v) ); // good idea, v is not copied
  // here v is modified
  std::thread t2( [&v]{calc(v)}; } ); // also good idea
  // here v is modified
  // ...
```

Technische Details II

- Threads können nicht kopiert werden.

Technische Details II

- Threads können nicht kopiert werden.

```
{
  std::thread t1(hello);
  std::thread t2;
  t2 = t1; // compiler error
  t1.join();
}
{
  std::thread t1(hello);
  std::thread t2;
  t2 = std::move(t1); // ok
  t2.join();
}
```

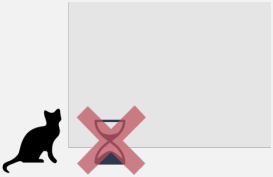
Garantien einer Mutex (=Lock).

Korrektheit (Safety)

- Maximal ein Prozess in der kritischen Region

Fortschritt (Liveness)

- Das Betreten der kritischen Region dauert nur endliche Zeit dauern, wenn kein Thread in der kritischen Region verweilt.



Locks: RAII Ansatz

```
class BankAccount {
  int balance = 0;
  std::mutex m;
public:
  ...
  void withdraw(int amount) {
    std::lock_guard<std::mutex> guard(m);
    int b = getBalance();
    setBalance(b - amount);
  } // Destruction of guard leads to unlocking m
};
```

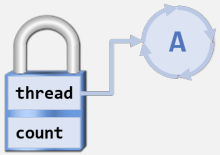
Locks: RAII Ansatz

```
class BankAccount {
  int balance = 0;
  std::mutex m;
public:
  ...
  void withdraw(int amount) {
    std::lock_guard<std::mutex> guard(m);
    int b = getBalance();
    setBalance(b - amount);
  } // Destruction of guard leads to unlocking m
};
```

Was ist mit getBalance / setBalance?

Reentrante Locks

Reentrantes Lock (rekursives Lock)



- merkt sich den betroffenen Thread;
- hat einen Zähler
  - Aufruf von lock: Zähler wird inkrementiert
  - Aufruf von unlock: Zähler wird dekrementiert. Wenn Zähler $= 0$, wird das Lock freigeben

Konto mit reentrantem Lock

```
class BankAccount {
  int balance = 0;
  std::recursive_mutex m;
  using guard = std::lock_guard<std::recursive_mutex>;
public:
  int  getBalance(){ guard g(m); return balance;
  }
  void setBalance(int x) { guard g(m); balance = x;
  }
  void withdraw(int amount) { guard g(m);
    int b = getBalance();
    setBalance(b - amount);
  }
};
```