Datenstrukturen und Algorithmen

Exercise 12

FS 2018

Program of today

1. Feedback of last exercise
2. Repetition theory
3. Programming Tasks

1. Feedback of last exercise

Football Championship

	Club	Points	Oppon.
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

Football Championship

	Club	Points	Oppon.
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

Historic *2-Point-Rule*

In each game, exactly 2 points are distributed: $2+0, 1+1, 0+2$

Football Championship

	Club	Points	Oppon.
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

Historic *2-Point-Rule*

In each game, exactly 2 points are distributed: $2+0, 1+1, 0+2$

Question: Can FCL still win the league?

Football Championship

	Club	Points	Oppon.	maximum...
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	

Historic *2-Point-Rule*

In each game, exactly 2 points are distributed: $2+0, 1+1, 0+2$

Question: Can FCL still win the league?

under the **Assumption:** The FCL wins both matches $\rightarrow$ 37p.

Football Championship

	Club	Points	Oppon.	maximum...
1)	FC St. Gallen (FCSG)	37	FCB, FCW	$+0$ points
2)	BSC Young Boys (YB)	36	FCW, FCB	$+1$ point
3)	FC Basel (FCB)	35	FCSG, YB	$+2$ points
4)	FC Luzern (FCL)	33	FCZ, GCZ	
5)	FC Winterthur (FCW)	31	YB, FCSG	

Historic *2-Point-Rule*

In each game, exactly 2 points are distributed: $2+0, 1+1, 0+2$

Question: Can FCL still win the league?

under the **Assumption:** The FCL wins both matches $\rightarrow$ 37p.

Football Championship

	Club	Points	Oppon.	maximum...
1)	FC St. Gallen (FCSG)	37	FCB, FCW	$+0$ points
2)	BSC Young Boys (YB)	36	FCW, FCB	$+1$ point
3)	FC Basel (FCB)	35	FCSG, YB	$+2$ points
4)	FC Luzern (FCL)	33	FCZ, GCZ	
5)	FC Winterthur (FCW)	31	YB, FCSG	never mind

Historic *2-Point-Rule*

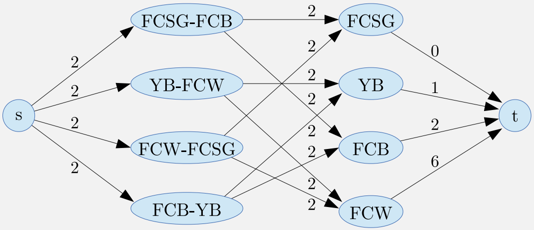
In each game, exactly 2 points are distributed: $2+0, 1+1, 0+2$

Question: Can FCL still win the league?

under the **Assumption:** The FCL wins both matches $\rightarrow$ 37p.

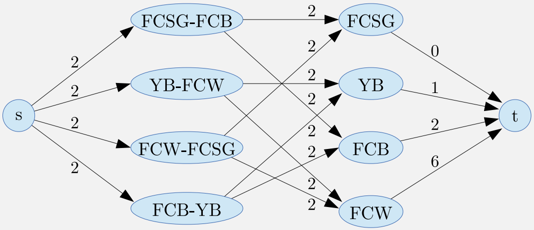
Football Championship

Assumption: FCL can still win the league.



Football Championship

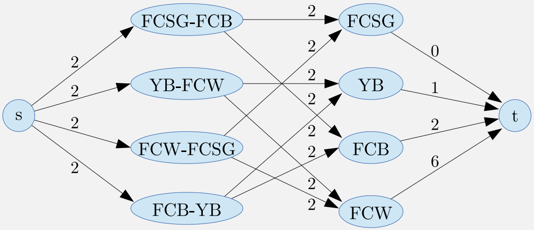
Assumption: FCL can still win the league.

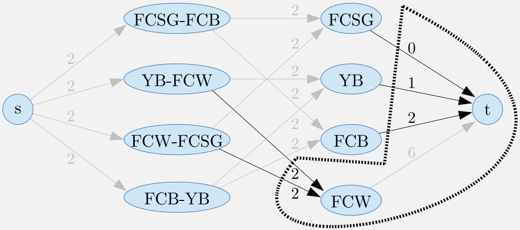


4 Games ⇒ We must have 8 Flow Units.

Football Championship

Assumption: FCL can still win the league.

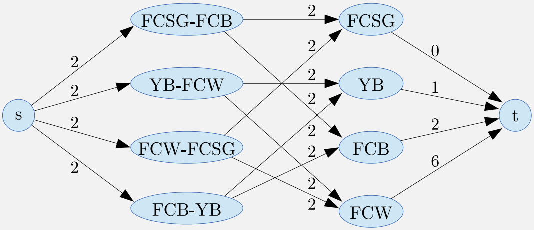


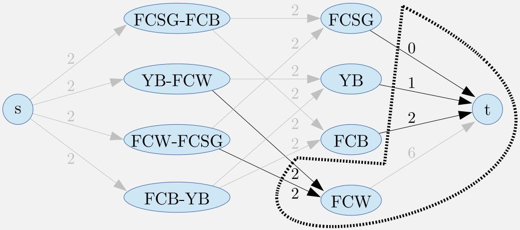


4 Games $\Rightarrow$ We must have 8 Flow Units.

Football Championship

Assumption: FCL can still win the league.





4 Games $\Rightarrow$ We must have 8 Flow Units.

But: MinCut has size 7. $\Rightarrow$ **Contradiction**.

2. Repetition theory

Parallel Performance

Given

- fixed amount of computing work W (number computing steps)
- Sequential execution time T_1
- Parallel execution time on p CPUs

	runtime	speedup	efficiency
perfection (linear)	$T_p = T_1/p$	$S_p = p$	$E_p = 1$
loss (sublinear)	$T_p > T_1/p$	$S_p < p$	$E_p < 1$
sorcery (superlinear)	$T_p < T_1/p$	$S_p > p$	$E_p > 1$

Gustafson's Law

- Fix the time of execution
- Vary the problem size.
- Assumption: the sequential part stays constant, the parallel part becomes larger

Gustafson's Law

Work that can be executed by one processor in time T:

$$W_s + W_p = T$$

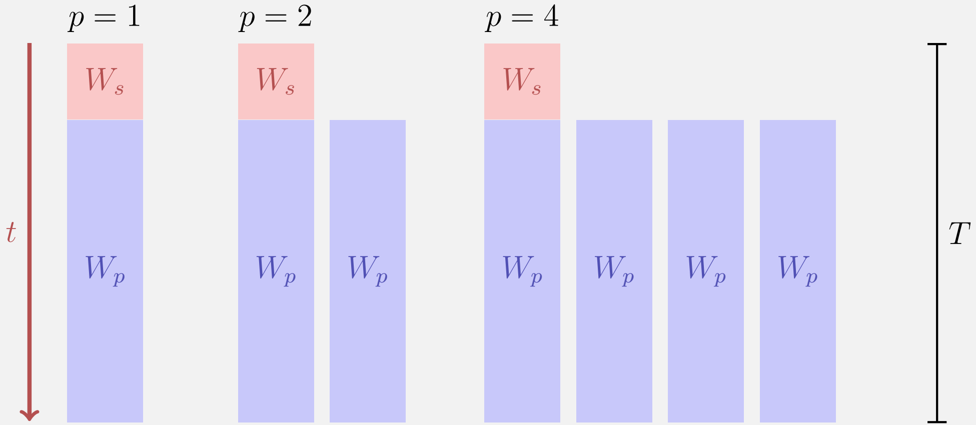
Work that can be executed by p processors in time T:

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

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: Ingredients

Computational work W falls into two categories

- Paralellisable part W_p
- Not parallelisable, sequential part W_s

Assumption: W can be processed sequentially by one processor in W time units ($T_1 = W$):

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

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

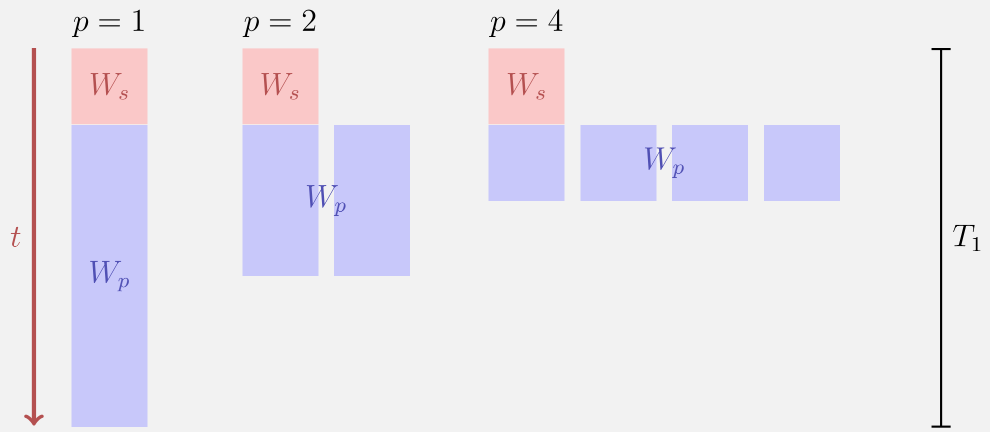
With sequential, not parallelizable fraction λ: $W_s = \lambda W$, $W_p = (1-\lambda)W$:

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

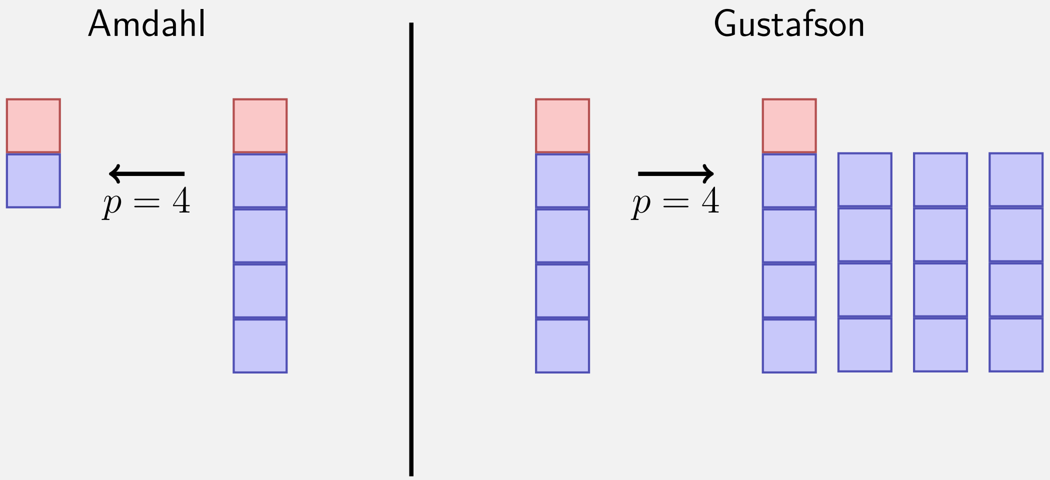
Thus

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

Illustration Amdahl's Law



Amdahl vs. Gustafson



Amdahl vs. Gustafson, or why do we care?

Amdahl	**Gustafson**
pessimist	optimist
strong scaling	weak scaling

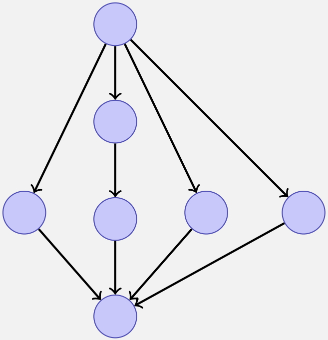
Amdahl vs. Gustafson, or why do we care?

Amdahl	**Gustafson**
pessimist	optimist
strong scaling	weak scaling

$\Rightarrow$ need to develop methods with small sequential protion as possible.

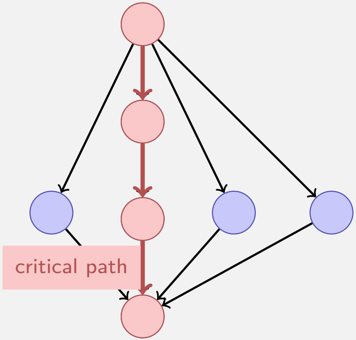
Question

- Each Node (task) takes 1 time unit.
- Arrows depict dependencies.
- Minimal execution time when number of processors $= \infty$?



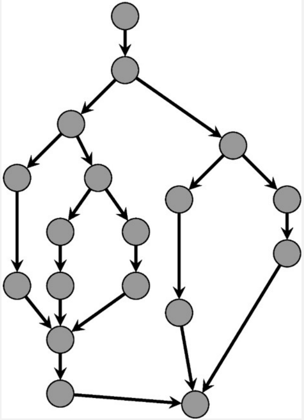
Question

- Each Node (task) takes 1 time unit.
- Arrows depict dependencies.
- Minimal execution time when number of processors $= \infty$?



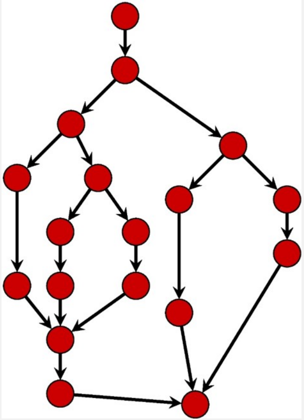
Performance Model

- p processors
- Dynamic scheduling
- T_p: Execution time on p processors



Performance Model

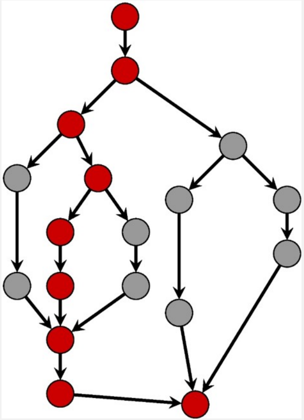
- T_p: Execution time on p processors
- T_1: *work*: time for executing total work on one processor
- T_1/T_p: Speedup



Performance Model

- T_∞: *span*: critical path, execution time on ∞ processors. Longest path from root to sink.
- T_1/T_∞: *Parallelism:* wider is better
- Lower bounds:

$$T_p \geq T_1/p \quad \text{Work law}$$
$$T_p \geq T_\infty \quad \text{Span law}$$



Greedy Scheduler

Greedy scheduler: at each time it schedules as many as availbale tasks.

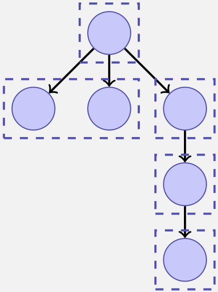
Theorem

On an ideal parallel computer with p processors, a greedy scheduler executes a multi-threaded computation with work T_1 and span T_∞ in time

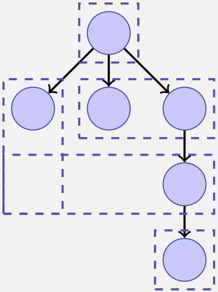
$$T_p \leq T_1/p + T_\infty$$

Beispiel

Assume $p = 2$.



$T_p = 5$

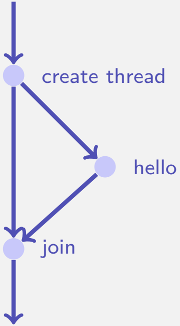


$T_p = 4$

3. Programming Tasks

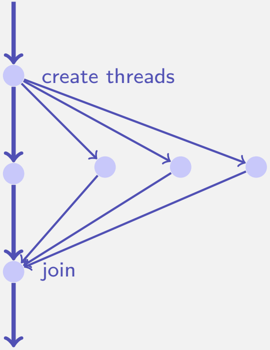
C++11 Threads

```cpp
#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;
}
```



Nondeterministic Execution!

One execution:

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

Other execution:

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

Other execution:

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

Technical Details I

- With allocating a thread, reference parameters are copied, except explicitly std::ref is provided at the construction.

Technical Details I

- With allocating a thread, reference parameters are copied, except explicitly std::ref is provided at the construction.

```
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
  // ...
```

Technical Details II

- Threads cannot be copied.

Technical Details II

- Threads cannot be copied.

```
{
  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();
}
```

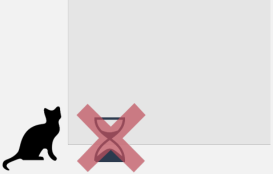
Guarantees of Mutual Exclusion (Lock)

Correctness (Safety)

- At most one process executes the critical section code

Liveness

- Acquiring the mutex terminates in finite time when no process executes in the critical section



Locks: RAII Approach

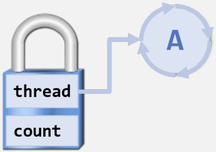
```
class BankAccount {
  int balance = 0;
  std::mutex m;
public:
  ...
  void withdraw(int amount) {
    std::lock_guard<std::mutex> guard(m);
    int b = getBalance();
    setBalance(b - amount);
  } // Destruction of guard leads to unlocking m
};
```

Locks: RAII Approach

```
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
};
```

What about getBalance / setBalance?

Reentrant Locks



Reentrant Lock (recursive lock)

- remembers the currently affected thread;
- provides a counter
    - Call of lock: counter incremented
    - Call of unlock: counter is decremented. If counter $= 0$ the lock is released.

Account with reentrant 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);
  }
};
```