

# Datenstrukturen und Algorithmen

## Übung 13

FS 2021

# Programm von heute

- 1 Feedback letzte Übung
- 2 Wiederholung Theorie
- 3 Nächste Übung

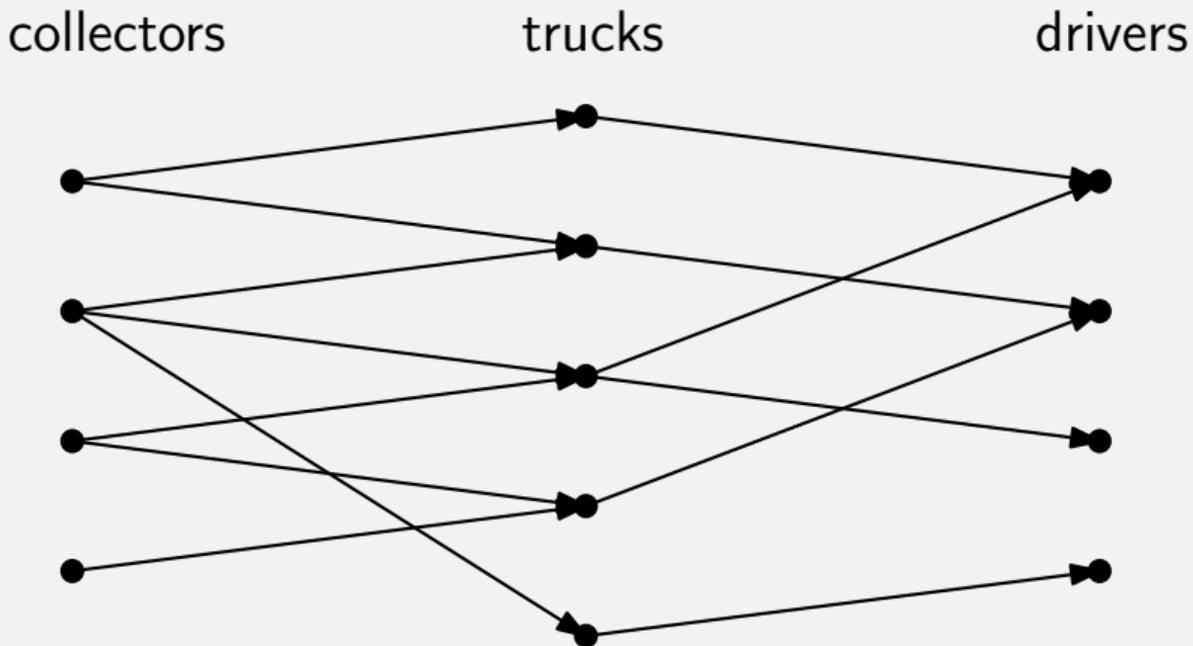
# 1. Feedback letzte Übung

# Aufgabe Applying Maximum Flow

- Knotenkapazität: Knoten durch Eingangs- und Ausgangsknoten ersetzen.
- Verbinde Knoten durch Kante mit dieser Kapazität.

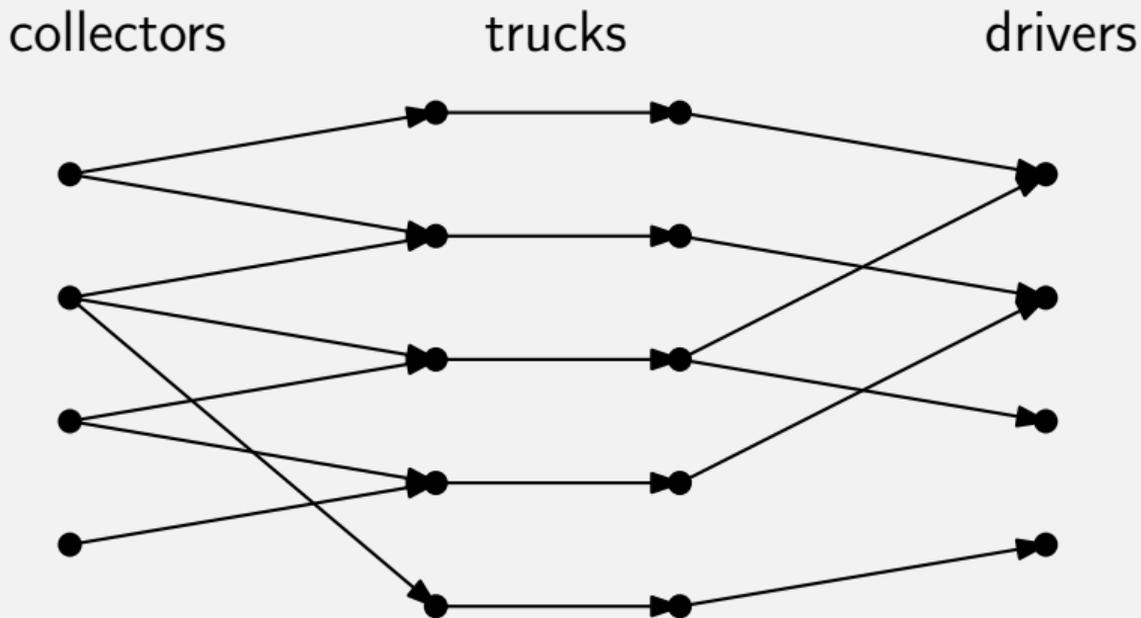
# Aufgabe Applying Maximum Flow

- Wir haben Auflader, Fahrer und Fahrzeuge.



# Aufgabe Applying Maximum Flow

- Wir haben Auflader, Fahrer und Fahrzeuge.







# Aufgabe: Summe eines Vektors

```
void sum_par( Iterator beg, Iterator end, int& result ) {
    const int nThreads = std::thread::hardware_concurrency();
    std::vector<std::thread> myThreads;
    std::vector<int> sums( nThreads, 0 );
    const int partSize = (end-beg)/nThreads;

    for( int i=0; i<nThreads-1; ++i ){
        myThreads.emplace_back(
            std::thread(sum_ser, beg, beg + partSize, std::ref(sums[i])));
        beg += partSize;
    }
    // ...
    for( auto& t:myThreads ) t.join();
    sum_ser( sums.begin(), sums.end(), result );
}
```

# Aufgabe: Summe eines Vektors

```
void sum_ser(  
    Iterator from,  
    Iterator to,  
    int& result ) {  
  
    int local = 0;  
    for( ;from != to; ++from )  
        local += *from;  
    result = local;  
}
```

```
void sum_ser(  
    Iterator from,  
    Iterator to,  
    int& result ) {  
  
    result = 0;  
    for( ;from != to; ++from )  
        result += *from;  
}
```

# Aufgabe: Summe eines Vektors

```
void sum_ser(  
    Iterator from,  
    Iterator to,  
    int& result ) {  
  
    int local = 0;  
    for( ;from != to; ++from )  
        local += *from;  
    result = local;  
}
```

```
void sum_ser(  
    Iterator from,  
    Iterator to,  
    int& result ) {  
  
    result = 0;  
    for( ;from != to; ++from )  
        result += *from;  
}
```

Difference?

# Aufgabe: Summe eines Vektors

```
void sum_ser(  
    Iterator from,  
    Iterator to,  
    int& result ) {  
  
    int local = 0;  
    for( ;from != to; ++from )  
        local += *from;  
    result = local;  
}
```

```
void sum_ser(  
    Iterator from,  
    Iterator to,  
    int& result ) {  
  
    result = 0;  
    for( ;from != to; ++from )  
        result += *from;  
}
```

Difference?

execution time: 0.468879 ms

execution time: 0.944031 ms

# Aufgabe: Summe eines Vektors – False Sharing!

```
void sum_ser(  
    Iterator from,  
    Iterator to,  
    int& result ) {  
  
    int local = 0;  
    for( ;from != to; ++from )  
        local += *from;  
    result = local;  
}
```

```
void sum_ser(  
    Iterator from,  
    Iterator to,  
    int& result ) {  
  
    result = 0;  
    for( ;from != to; ++from )  
        result += *from;  
}
```

Difference?

execution time: 0.468879 ms

execution time: 0.944031 ms

# Aufgabe: Mergesort (2-threads)

```
void mergesort_par( std::vector<int> & v ) {  
    int n = v.size();  
    int partSize = n / 2;  
  
    std::thread t1( mergesort, std::ref(v), 0, partSize-1 );  
    std::thread t2( mergesort, std::ref(v), partSize, n-1 );  
    t1.join();  
    t2.join();  
    merge( v, 0, partSize-1, n-1 );  
}
```

Analog mit  $n$  threads

# Aufgabe: Mergesort Rekursiv

```
void mergesort_par(std::vector<int> & v, int cutoff, int l, int r) {
    if (r-l < cutoff){ // sequential base case
        mergesort( v, l, r );
    } else {
        int m = ( l+r )/2 ;
        std::thread t (mergesort_par,std::ref(v),cutoff,l,m);
        mergesort_par(v,cutoff,m+1,r); // avoid forking another thread
        t.join();
        merge(v,l,m,r);
    }
}
```

## **2. Wiederholung Theorie**

# Speedup, Performanz und Effizienz

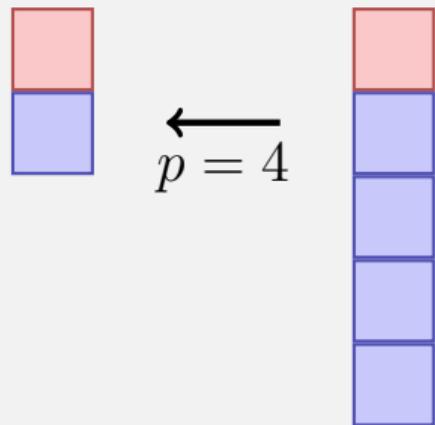
Gegeben

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

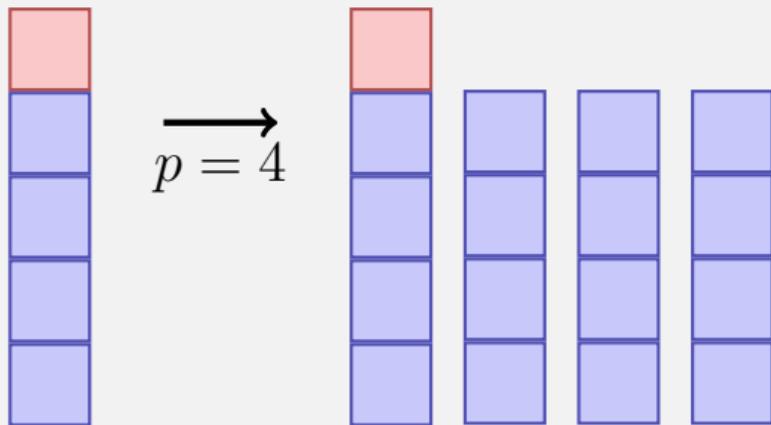
	Ausführungszeit	Speedup	Effizienz
Perfektion (linear)	$T_p = T_1/p$	$S_p = p$	$E_p = 1$
Verlust (sublinear)	$T_p > T_1/p$	$S_p < p$	$E_p < 1$
Hexerei (superlinear)	$T_p < T_1/p$	$S_p > p$	$E_p > 1$

# Amdahl vs. Gustafson

Amdahl



Gustafson



# Amdahl vs. Gustafson, or why do we care?

<b>Amdahl</b>	<b>Gustafson</b>
Pessimist	Optimist
starke Skalierung	schwache Skalierung

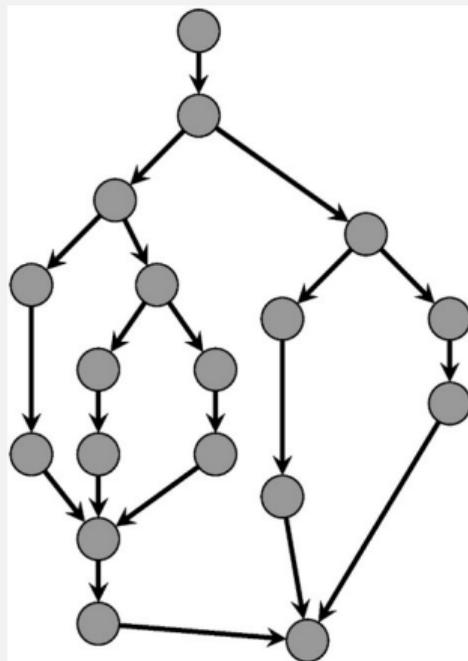
# Amdahl vs. Gustafson, or why do we care?

<b>Amdahl</b>	<b>Gustafson</b>
Pessimist	Optimist
starke Skalierung	schwache Skalierung

⇒ Methoden müssen entwickelt werden so dass sie einen möglichst kleinen sequenziellen Anteil haben.

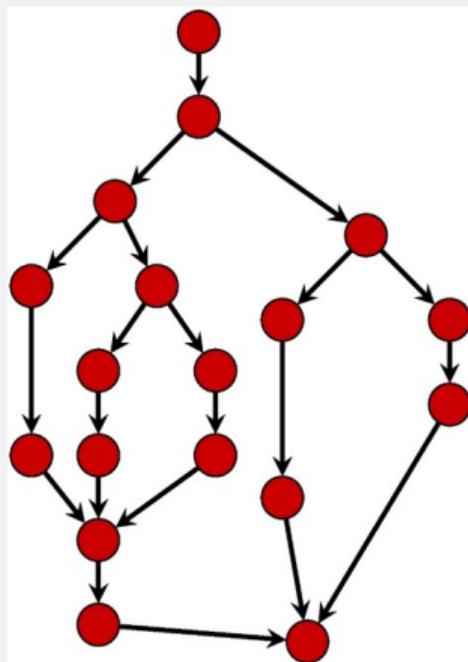
# Task Parallelität: Performanzmodell

- $p$  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





# Greedy Scheduler

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

## Theorem

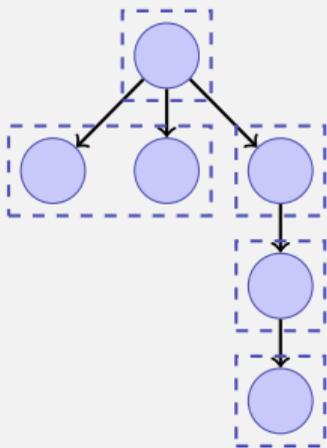
*Auf einem idealen Parallelrechner mit  $p$  Prozessoren führt ein Greedy-Scheduler eine mehrfädige Berechnung mit Arbeit  $T_1$  und Zeitspanne  $T_\infty$  in Zeit*

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

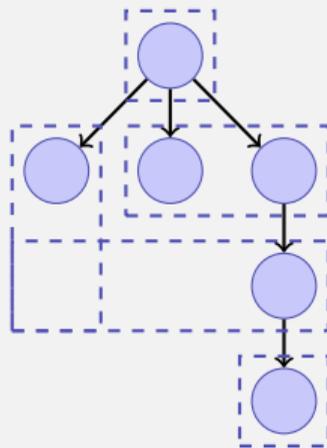
*aus.*

# Beispiel

Annahme  $p = 2$ .



$$T_p = 5$$



$$T_p = 4$$

# Race Conditions (Wettlaufsituationen)

*Data Race* (low-level Race-Conditions) Fehlerhaftes Programmverhalten verursacht durch ungenügend synchronisierten Zugriff zu einer gemeinsam genutzten Resource, z.B. gleichzeitiges Lesen/Schreiben oder Schreiben/Schreiben zum gleichen Speicherbereich.

*Bad Interleaving* (High Level Race Condition) Fehlerhaftes Programmverhalten verursacht durch eine unglückliche Ausführungsreihenfolge eines Algorithmus mit mehreren Threads, selbst dann wenn die gemeinsam genutzten Ressourcen anderweitig gut synchronisiert sind.

# Speichermodelle

Wann und ob Effekte von Speicheroperationen für Threads sichtbar werden, hängt also von Hardware, Laufzeitsystem und der Programmiersprache ab.

Ein *Speichermodell* (z.B. das von C++) gibt Minimalgarantien für den Effekt von Speicheroperationen.

- Lässt Möglichkeiten zur Optimierung offen
- Enthält Anleitungen zum Schreiben Thread-sicherer Programme

C++ gibt zum Beispiel *Garantien, wenn Synchronisation mit einer Mutex verwendet* wird.

# Counter Problem

```
std::vector<std::thread> tv(10);
int counter {0};
for (auto & t:tv)
    t = std::thread([&]{
        for (int i =0; i<100000; ++i){counter++;} // race!!
    });
for (auto & t:tv)
    t.join();
std::cout << "count= " << counter << std::endl;
```

# Counter Lösung 1

```
std::vector<std::thread> tv(10);
std::mutex lock;
int counter {0};
for (auto & t:tv)
    t = std::thread([&]{
        for (int i =0; i<100000; ++i){
            mutex.lock(); counter++; mutex.unlock(); // synchronized!
        });
for (auto & t:tv)
    t.join();
std::cout << "count= " << counter << std::endl;
```

# Counter Lösung II

```
std::vector<std::thread> tv(10);
std::atomic<int> counter {0};
for (auto & t:tv)
    t = std::thread([&]{
        for (int i =0; i<100000; ++i){counter++;} // atomic!!
    });
for (auto & t:tv)
    t.join();
std::cout << "count= " << counter << std::endl;
```

# Quiz: Was ist hier falsch?

```
void exchangeSecret(Person & a, Person & b) {  
    a.getMutex()->lock();  
    b.getMutex()->lock();  
    Secret s = a.getSecret();  
    b.setSecret(s);  
    a.getMutex()->unlock();  
    b.getMutex()->unlock()  
}
```

# Deadlock (Verklemmung)

Thread 1:

```
exchangeSecret(p1, p2);
```

Thread 2:

```
exchangeSecret(p2, p1);
```

# Deadlock (Verklemmung)

Thread 1:

```
exchangeSecret(p1, p2);
```

Thread 2:

```
exchangeSecret(p2, p1);
```

Was tun?

# Mögliche Lösung

```
void exchangeSecret(Person & a, Person & b) {
    std::mutex* first;
    std::mutex* second;
    if (a.name < b.name){
        first = a.getMutex(); second = b.getMutex();
    } else {
        first = b.getMutex(); second = a.getMutex();
    }
    first->lock();
    second->lock();
    Secret s = a.getSecret();
    b.setSecret(s);
    first->unlock();
    second->unlock();
}
```

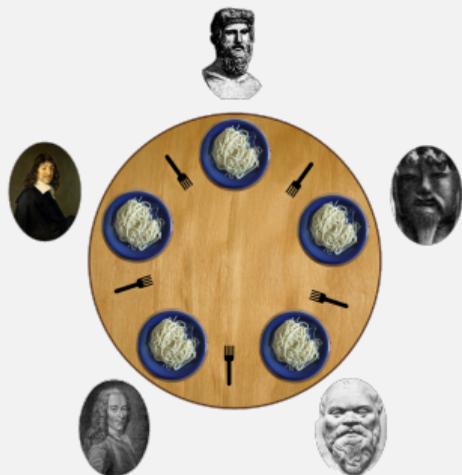
# Deadlocks und Races

- Nicht einfach zu sehen
- Schwierig zu debuggen
- Treten u.U. selten auf
- Testing genügt nicht
- Eigentlich muss man die Korrektheit des Codes formal beweisen

Vorsicht und Sorgfalt ist gefragt beim Programmieren mit Locks!

## **3. Nächste Übung**

# Dining Philosophers



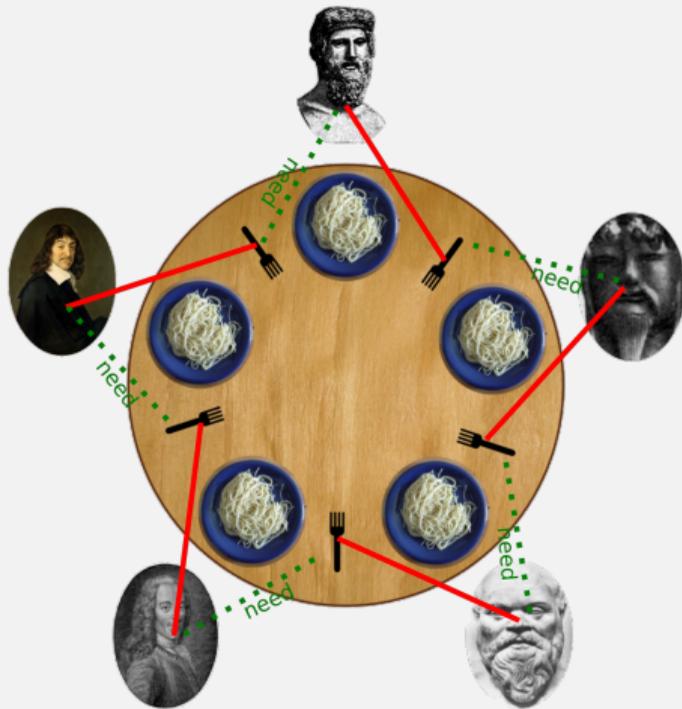
- Philosophen denken und essen abwechslungsweise. Zum Essen brauchen sie zwei Gabeln.
- Philosoph = Thread, Gabel = Lock.

# Dining Philosophers - Pseudocode

```
while(true) {  
    think();  
    acquire_fork_on_left_side();  
    acquire_fork_on_right_side();  
    eat();  
    release_fork_on_right_side();  
    release_fork_on_left_side();  
}
```

- Problem mit diesem Program?

# Dining Philosophers - Deadlock



■ Lösung?

# Dining Philosophers

- Zyklische Abhängigkeit brechen
- Beispielsweise: Philosoph fünf nimmt erste **rechte** Gabel.
- Allgemeine Möglichkeit: Lock Ordnung definieren. Dann immer in dieser Reihenfolge locken.

Fragen?