Datenstrukturen und Algorithmen

Übung 12

FS 2020

Programm von heute

1. Feedback letzte Übung
2. Parallele Programmierung
3. C++ Threads
4. In-Class Übung: Image Segmentation

1. Feedback letzte Übung

2. Parallele Programmierung

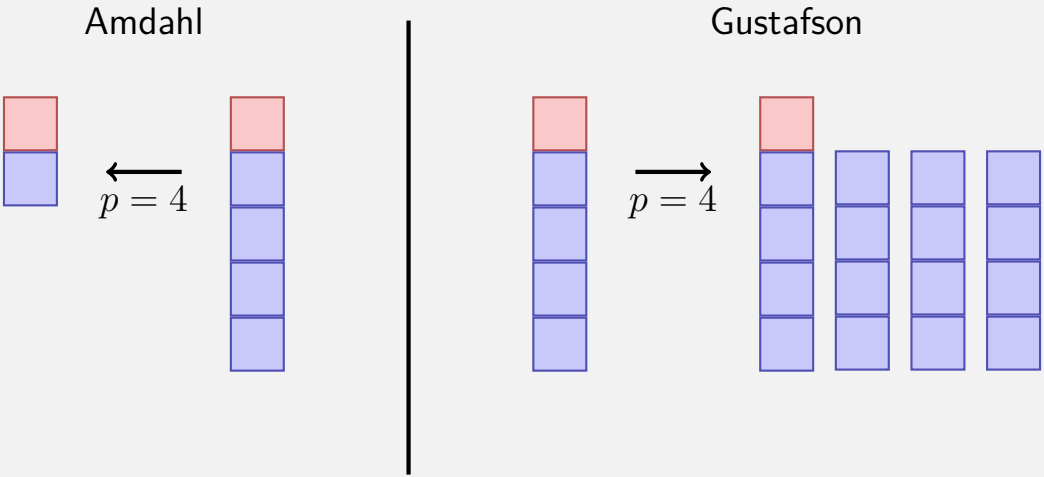
Speedup, Performanz und Effizienz

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 vs. Gustafson, or why do we care?

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

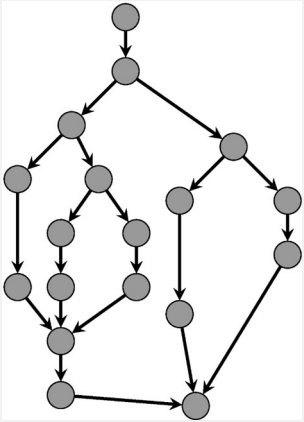
Amdahl vs. Gustafson, or why do we care?

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

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

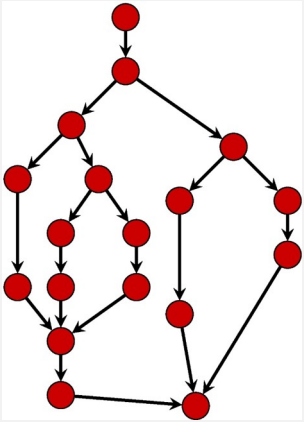
Task Parallelität: Performanzmodell

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



Performanzmodell

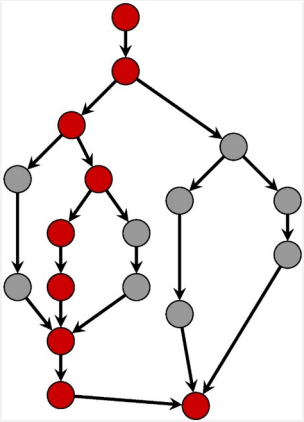
- T_p: Ausführungszeit auf p Prozessoren
- T_1: *Arbeit:* Zeit für die gesamte Berechnung auf einem Prozessor
- T_1/T_p: Speedup



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

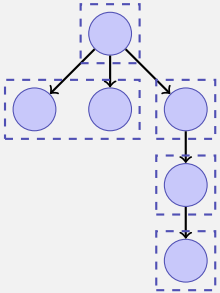
Auf einem idealen Parallelrechner mit p Prozessoren führt ein Greedy-Scheduler eine mehrfädige Berechnung mit Arbeit T_1 und Zeitspanne T_∞ in Zeit

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

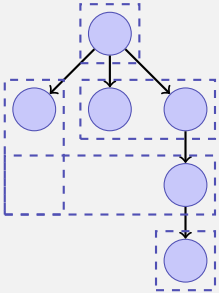
aus.

Beispiel

Annahme $p = 2$.



$T_p = 5$



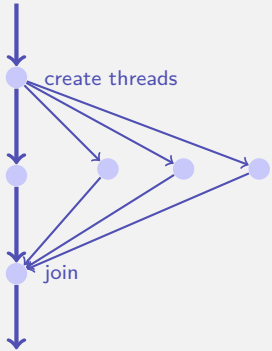
$T_p = 4$

3. C++ Threads

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

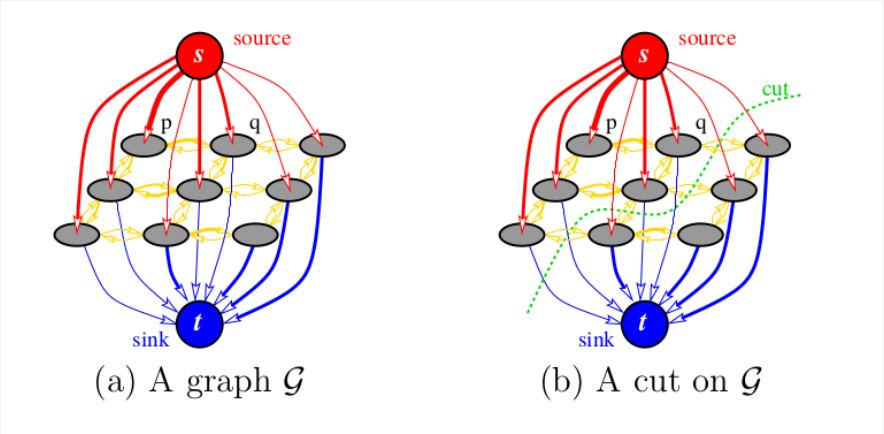
- Threads können nicht kopiert werden.

```
{
  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();
}
```

4. In-Class Übung: Image Segmentation

Max Flow / Edmonds-Karp / Push-Relabel

Idee: Max-Flow/Min-Cut



Quelle: An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision, Y.Boykov and V.Kolmogorov, IEEE Transactions on PAMI, Vol. 26, No. 9, pp. 1124-1137, Sept. 2004

Challenges

- Kapazitäten Quelle-Pixel/Pixel-Senke/ zwischen benachbarten Pixeln?
  - Ähnlichkeit zur Vordergrundfarbe/Hintergrundfarbe
  - Ähnlichkeit der Farbwerte

  $\Rightarrow$ Heuristik / Erfahrung (=Literatur)
- Edmonds-Karp Algorithmus zu langsam
  $\Rightarrow$ Push-Relabel Algorithmus

Implementation Push-Relabel?

Input: Flussnetzwerk $G = (V, E, c)$, Quelle s und Senke t. $n := |v|$

$h(s) \leftarrow n$
foreach $v \neq s$ **do** $h(v) \leftarrow 0$
foreach $(u, v) \in E$ **do** $f(u, v) \leftarrow 0$
foreach $(s, v) \in E$ **do** $f(s, v) \leftarrow c(s, v)$

while $\exists u \in V \setminus \{s, t\} : \alpha_f(u) > 0$ **do**
    wähle u mit $\alpha_f(u) > 0$ und maximalem $h(u)$ ← **in $\mathcal{O}(1)$?**
    if $\exists v \in V : c_f(u, v) > 0 \ \wedge \ h(v) = h(u) - 1$ **then**
        `push`(u, v) ←**Effizientes Finden der Kanten?** // push
    else
        $h(u) \leftarrow h(u) + 1$ // relabel

Möglichkeiten

Knotenverwaltung:

- Maximale Höhe $2n - 1 \Rightarrow$ Knotenlisten nach Höhe
  Algorithmus Laufzeit $\mathcal{O}(n^2\sqrt{m})$
- Abschwächen der Reihenfolge: verwende FIFO oder Relabel-to-front Heuristik für die Knoten mit Exzess.
  Algorithmus Laufzeit $\mathcal{O}(n^3)$

Kantenverwaltung:

- Merke die zuletzt besuchte Kante (=iterator) pro Knoten.
- Für Image Segmentation unnötig, da wenige Kanten pro Knoten

Fragen?