Begriffe und Grundlagen

M. Thaler, TG208, tham@zhaw.ch
www.zhaw.ch/~tham

- Literartur
  - [McCool] pp. 39-75
  - [Mattson] pp. 17-23
  - [Grama] pp. 197 - 231
  - [Gove] pp. 333 - 382

Lehrziele

- **Sie können**
  - die Begriffe Concurrent und Parallel erklären
  - die wichtigsten Begriffe im Zusammenhang mit Parallel Computing aufzählen und erklären
  - die wichtigsten Grössen zur Performance Evaluation aufzählen und erklären
  - das Gesetz von Amdhal und Gustafson erklären und diskutieren
  - Datentransfer mit Hilfe von Latenz und Bandbereite beschreiben
  - Skalierbarkeit bei parallelen Systemen erklären
  - können die O-Notation erklären

Inhalt

- Parallel Computing
- Wichtige Begriffe
- Performance Evaluation
- Skalierbarkeit
- Komplexität

What is parallel computing?

- **Concurrent computing**
  - traditionell auf Uniprozessor-Systemen
  - Betriebssystem → schaltet zwischen Tasks um
    - bessere Ausnutzung der Rechnerressourcen
    - Tasks "scheinen" gleichzeitig verarbeitet zu werden
    - gilt auch wenn mehrere Prozessoren zur Verfügung stehen
  - nicht notwendigerweise "gleichzeitg"
- **Parallel computing**
  - mehrere Prozessoren arbeiten gleichzeitig, um ein Programm schneller zu berarbeiten als mit einem Prozessor
  - treibende Kräfte
    - Performance (faster, more, less power)
    - aktuelle HW-Plattformen

- Concurrent in Sinne von
  - das Betriebssystem kann keine Tasks parallelisieren sondern "verschränkt" die Ausführung der Tasks
- Anmerkung zu den Begriffen
  - concurrent: alle Tasks machen "Fortschritte"
  - parallel: alle arbeiten echt gleichzeitig (parallel)

- Zitat Tanenbaum (Operating Systems, 3rd ed.) zum Thema Shared Memory Multiprozessoren

  " A program running on any of the CPU's sees a normal (usually paged) virtual address space. The only unusal property this system has is that the CPU can write some value into a me-mory word and then read the word back and get a different value (because another CPU has changed it)."

Wichtige Begriffe

- **Task**
  - Programm oder Algorithmus: in Tasks aufgeteilt
    - eine Sequenz von Instruktionen (Teilaufgaben)
    - "logischer" Teil eines Programms oder Algorithmus
  - z.B.
    - eine Funktion
    - Bearbeitung eines Datenblocks
    - Update eines Datenwertes
    - etc.

Wichtige Begriffe

- **Unit of Execution: UE**
  - Ausführung eines Tasks: Task wird UE zugeteilt
    - Thread: gemeinsame Ressourcen → Shared Memory
    - Prozess: keine gemeinsamen Ressourcen → IPC
- **Processing Element: PE**
  - generisches Hardware-Element, das Instruktionsstrom ausführt
    - Workstation
    - Prozessor, CPU, ALU, ...

... wichtige Begriffe

- **Load Balance / Load Balancing**
  - Zuteilung: Task → UE → PE
    - beeinflusst gesamte Performance (PE Nutzung)
  - Zuteilung: UE → PE
    - statisch oder dynamisch
- **Synchronisation**
  - stellt sicher, dass Ereignisse und Abläufe in der richtigen Reihenfolge stattfinden, falls gefordert
  - verhindert Race Conditions

- Zuteilung UE → PE
  - statisch: zur Compilationszetit
  - dynamisch: zur Laufzeit

- Race Condition
  - mindestens zwei Tasks greifen auf gemeinsame Daten zu
  - mindestens ein Zugriff ist ein Schreibzugriff

Analytische Modellierung

- **Performance Measures**
  - Speedup
  - Effizienz
  - Kosten
- **Modelle**
  - Amdhal
  - Gustafson
  - Work-Span
- **Weitere Faktoren**
  - Skalierbarkeit
  - Daten Transfers
  - Arithmetische Dichte

Performance Measures

- **Ausführunszeiten**
  - $T(1)$: Ausführungszeit auf einem Prozessor
  - $T(P)$: Ausführungszeit auf P Prozssoren
  - $T_i(P)$: Ausführungszeit auf Prozessor i bei P Prozessoren
- **Speedup[1)]**
  - beantwortet: wie viel mal schneller als auf einem Prozessor

$$S(P) = \frac{T(1)}{T(P)}$$

- **Effizienz[1)]: mittlere Auslastung**

$$E(P) = \frac{S(P)}{P}$$

1) Speedup und Effizienz: relative Masse

- Annahme
  - die P Prozessoren stehen während der gesamten Ausführungszeit als "eine Parallel-Ressource" zur Verfügung
  - diese Ressource ist nicht "sharable"
- Ausführungszeiten
  - Ausführungszeiten → Wall-Clock Time
  - nur Tasks, die zum Programm gehören
- Effizienz
  - gibt Hinweis auf Rechnerauslastung

... Performance Measures

- **Kosten**
  - gesamter Rechenaufwand (Zeit) auf P Prozessoren

$$C(P) = P \cdot \max(T_i(P)) \quad i = 0 \ldots P\text{-}1$$

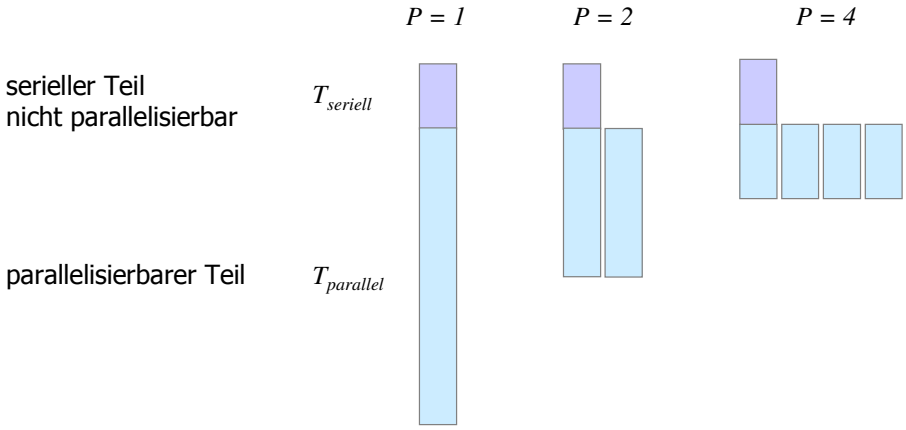
  - kostenoptimal für $C(P) = C(1)$

- **Rechenleistung & Leistungsverbrauch**
  - Power $\sim P \cdot f^3$
  - Performance $\sim P \cdot f$

- Kosten
  - Parallel-Ressource ist während längster $T_i(P)$ belegt
  - beinhalten die gesamten Prozessor-Ressource-Kosten
  - Overhead "erzeugt" Kosten

Amdahls's Law (1967)

- **Grundidee**
  - Problemgrösse N konstant
  - Anzahl Prozessoren P variabel



- Hinweis zu P
  - Anzahl Prozessoren
  - auch Anzahl Tasks, die von Hardware parallel ausgeführt werden können
  - meist Verdoppelung, weil i.d.R. Anzahl Prozessoren Zweierpotenz

... Amdahls Law

Ansatz

- Rechenzeit für paralleles Programm
  - $T(1)$: Ausführungszeit auf einem Prozessor
  - γ: serieller, nicht parallelisierbarer Anteil von $T(1)$
  - P: Anzahl Prozessoren

$$T(P) = T_{seriell} + T_{parallel} = \gamma \cdot T(1) + \frac{(1-\gamma) \cdot T(1)}{P}$$

Speedup und Effizienz

$$S(P) = \frac{T(1)}{T(P)} = \frac{1}{\gamma + \frac{1-\gamma}{P}}$$

$$E(P) = \frac{S(P)}{P} = \frac{1}{\gamma \cdot P + 1 - \gamma}$$

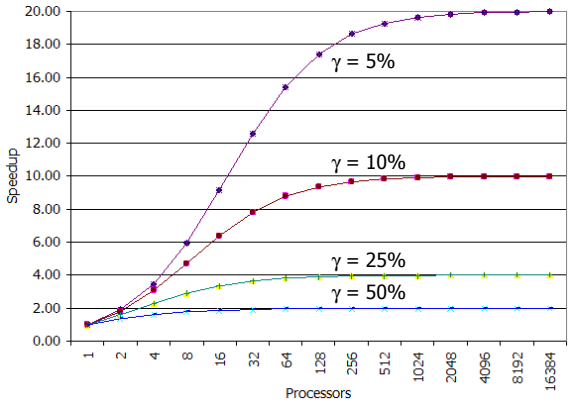
- Hinweise
  - berücksichtigt nicht
    - Overhead durch Parallel Computing: z.B. Erzeugen und Beenden von Threads
    - "schlechte Load Balance"
    - Overhead für Kommunikation
  - wichtigste Annahme
    - Problemgrösse bleibt konstant

... Performance

- **Diskussion Amdhal**
  - serieller Anteil bestimmt maximalen Speedup

$$S(P \to \infty) = \frac{1}{\gamma}$$

$$E(P \to \infty) = ?$$

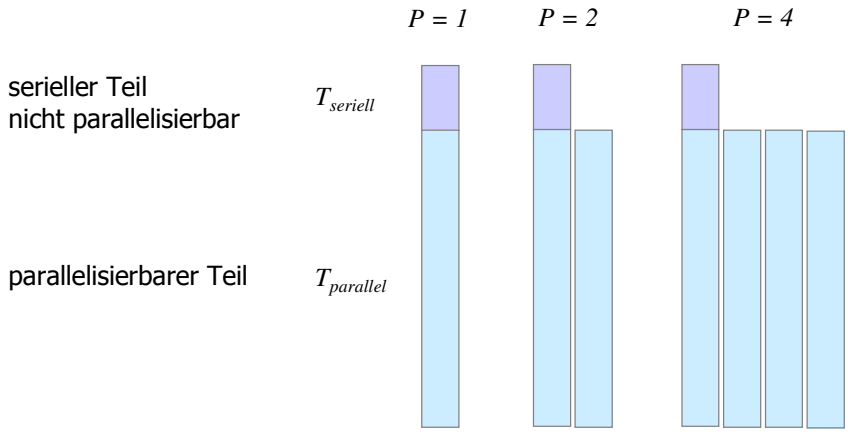


  - gleiche Problemstellung → ändern der Anzahl Prozessoren
    - für gewisse Fälle eine unrealistische Annahme
  - zu pessimistisch, wenn Problemgrösse und Anzahl Prozessoren wachsen, Rechenzeit aber konstant bleibt → Gustafson's Law

- In gewissen Fällen ist Amdhal aber auch zu optimistisch
  - keine Berücksichtigung des Overheads
  - Kurven beginnen in diesem Fall mit steigender Anzahl Prozessoren wieder zu sinken

Gustafson's Law

- **Grundidee**
  - Anzahl Prozessoren P und Problemgrösse N nimmt zu
  - Rechenzeit bleibt konstant: $T(P) = T_{seriell} + T_{parallel} = const.$



... Gustafson's Law

- **Gustafson's Law (1988)**
  - Annahme
    - Problemgrösse N und Anzahl Prozessoren P nehmen zu
    - Rechenzeit $T_{total}(P)$ bleibt konstant
  - für $S(P)$ gilt:

$$S(P) = P - (P-1)\cdot\gamma$$

  - Zunahme Speedup → proportional zur Anzahl Prozessoren
    - Speedup bezieht sich hier auf "Menge" der Berechnung
  - für $E(P)$ gilt:

$$E(P) = \frac{S(P)}{P} = 1 - \left(1 - \frac{1}{P}\right)\cdot\gamma$$

- Gustafson's Law: bei Problemstellungen mit sehr viel Parallelität
  - Speedup, wenn Anzahl P und Problemgrösse N steigen
  - Annahme: $T_{seriell}$ konstant für steigende Anzahl P
    $T_{total}(P)$ bleibt gleich gross

- Berechnungen

$$T(P) = T_{seriell} + T_{parallel} = const = T(1)$$

$$\rightarrow \quad S(P) = \frac{T_{seriell} + P\cdot T_{parallel}}{T_{seriell} + T_{parallel}} = \frac{T_{seriell}}{T_{seriell} + T_{parallel}} + \frac{P\cdot T_{parallel}}{T_{seriell} + T_{parallel}}$$

$$S(P) = \frac{T_{seriell}}{T_{seriell} + T_{parallel}} + \frac{P}{T_{seriell}}\cdot\frac{T_{seriell}\cdot T_{parallel}}{T_{seriell} + T_{parallel}}$$

$$S(P) = \frac{T_{seriell}}{T_{seriell} + T_{parallel}}\cdot\left(1 + \frac{P}{T_{seriell}}\cdot T_{parallel}\right)$$

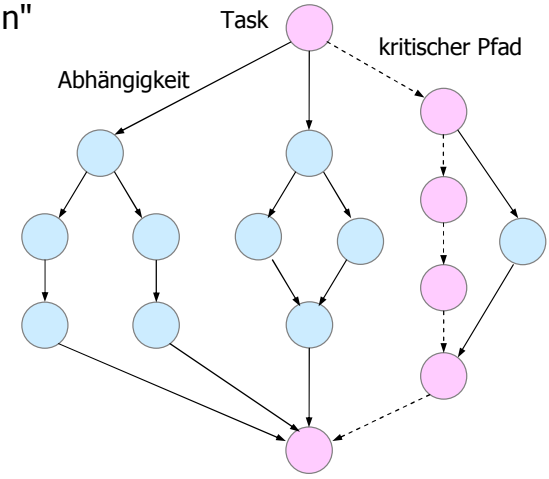
mit $\gamma = \frac{T_{seriell}}{T_{seriell} + T_{parallel}}$ bzw. $T_{parallel} = T_{seriell}\frac{1-\gamma}{\gamma}$

$$\rightarrow \quad S(P) = \gamma\cdot\left(1 + P\cdot\frac{1-\gamma}{\gamma}\right) = \gamma + P\cdot(1-\gamma) = P - (P-1)\cdot\gamma$$

- Effizienz $E(P) = 1 - \left(1 - \frac{1}{P}\right)\cdot\gamma \underset{P>>1}{\approx} 1-\gamma$

Work-Span Model

- **Grunidee**
  - Algorithmus als Graph (DAG)
  - Länge kritischer Pfad: "span"



Annahme:
alle Tasks gleiche Rechenzeit $T = 1$

→ $T_{seriell} = 2$ und $T(1) = 16$

... Work-Span Model

- **Span** (Länge kritischer Pfad)
  - definiert kürzest mögliche Rechenzeit

$$T(P \to \infty) = T_\infty$$

- **Speedup**
  - **obere Grenze** durch "Struktur" der Anwendung

$$S_{upper}(P \to \infty) = \frac{T(1)}{T(P \to \infty)} \leq \frac{T(1)}{T_\infty} = S_\infty$$

- **Beispiel (vorne)**

$$T(1) = 16 \quad T_\infty = 6 \quad \to \quad S_\infty = \frac{16}{6} \approx 2.66 \qquad S_{Amdahl}(\infty) = \frac{1}{\gamma} = 8$$

... Work-Span

- **Brennt's Lemma (1974)**
  - obere Grenze für Rechenzeit

$$T(P) \leq \frac{T(1) - T_\infty}{P} + T_\infty$$

perfekt parallelisiebar : $T(1) - T_\infty$

nicht parallelisierbar : T_∞

  - untere Grenze für Speedup

$$S(P) = \frac{T(1)}{T(P)} \geq \frac{T(1)}{\frac{T(1) - T_\infty}{P} + T_\infty} = \frac{P \cdot T(1)}{T(1) + (P-1) \cdot T_\infty}$$

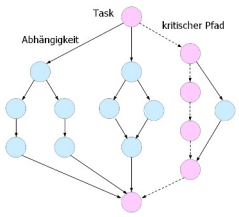
- **Speedup-Bereich**

$$\frac{T(1)}{\frac{T(1)}{P} - \frac{T_\infty}{P} + T_\infty} \leq S(P) \leq \frac{T(1)}{T_\infty}$$

mit $T_\infty << T(1)$

$\rightarrow$ T_∞ / P vernachlässigbar

... Work-Span



- **Beispiel**

Work Span & Parallel Slack

- **Mit Brent's Lemma und** $\frac{T(1)}{P} >> T_\infty \quad \rightarrow \quad \frac{T(1)}{P \cdot T_\infty} >> 1$
  - linearer Speedup, wenn Programm mehr Parallelität hat, als parallele Hardware verarbeiten kann

$$S(P) \approx \frac{P \cdot T(1)}{T(1) + (P-1) \cdot T_\infty} \underset{T_1 >> P \cdot T_\infty}{\approx} \frac{P \cdot T(1)}{T(1)} = P$$

- **Parallel Slack → potentieller Parallelismus**

$$S_\infty = \frac{T(1)}{T_\infty} \quad \rightarrow \quad \frac{S_\infty}{P} = \frac{T(1)}{P \cdot T_\infty}$$

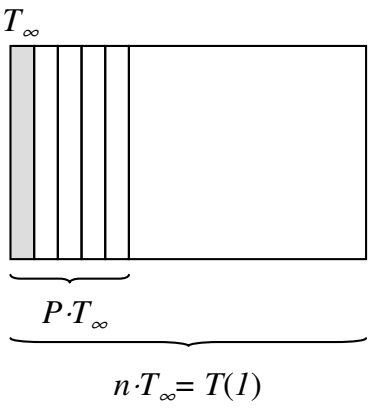
  - sollte > 8 sein [McCool]
  - Motivation für "Over-decomposition" mit "Greedy Scheduling"

- Greedy Scheduler

  "A greedy scheduler is a scheduler in which no processor is idle if there is more work it can do"

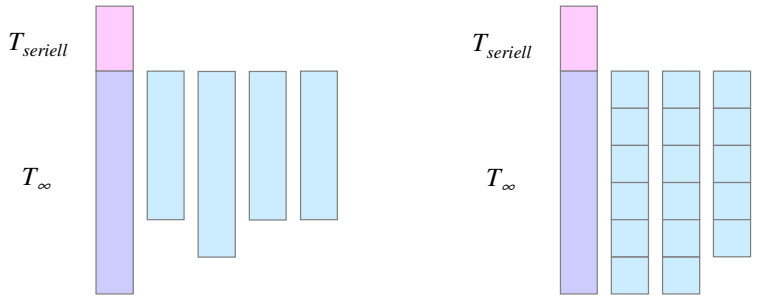
  Guy Blelloch

- $T(1) >> P \cdot T_\infty$
  - auch wenn $P \cdot T_\infty$ als minimale Rechnenzeit verwendet wird, gibt es immer noch viele Tasks bzw. "Rechenzeit", die gleichzeitig ausgeführt werden kann



Over-decomposition

- **Idee**
  - mehr "parallele Arbeit" zur Verfügung stellen als Hardware verarbeiten kann → kleinere "Arbeitspakete"
  - verbessert Load Balance → bessere Nutzung der Hardware



- Over-decomposition
  - macht nur Sinn wenn Support von Laufzeitumgebung
  - z.B. Taskpool mit Greedy Scheduling, etc.

- Task Queues
  - es werden soviele Threads gestartet wir die Hardware parallel verarbeiten kann
  - Tasks werden dynamische diesen Threads zugewiesen
  - ermöglicht Greedy Scheduling

- Greedy Scheduling
  - keine Prozessor ist idle, solange noch Arbeit zur Verfügung steht

Skalierbarkeit (Scaling)

- **Parallele Programme**
  - Entwurf und Implementation i.A. mit "kleinen Systemen"
    - Anzahl Prozessoren
    - Problemgrösse (Daten, Umfang, etc.)
- **Frage**
  - wie verhält sich Performance
    - mit mehr Prozessoren P
    - und grösserem Problem N
  - ist das System immer noch korrekt
  - d.h. wie skaliert das System
- **Skalierbarkeit**
  - Vergrösserung des Systems (HW/SW)
    → proportionale Vergrösserung des Resultats

- Skalierbarkeit
  - Zunahme nicht notwendigerweise linear
  - nicht skalierbar wenn
    - z.B. zusätzliche Prozessoren → Abnahme des Speedups (Kommunikation nimmt überproportional zu)

- Skalierbarkeit nach [Grama]
  - für konstanten Effizienzwert $E(N)$ existiert ein Paar "Anzahl PE's" und "Problemgrösse"
  - Anzahl PE's und Problemgrösse nehmen monton zu

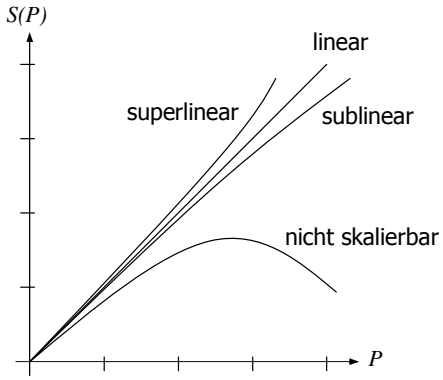
Weak and Strong Scaling

- **Weak Scaling**
  - die Problemgrösse wird erhöht
  - die Anzahl Prozessoren wird erhöht
  - die Arbeitsmenge pro Prozessor bleibt konstant
  - Gustafson gehört in diese Katergorie
- **Strong Scaling**
  - die Problemgrösse bleibt konstant
  - die Anzahl Prozessoren wird erhöht
  - die Arbeitsmenge pro Prozessor nimmt ab
  - schwieriger als Weak Scaling → weniger Data Re-Use und mehr Kommunikation
  - Amdhal gehört in diese Kategorie

- Weak Scaling
  - interessant für O(n) Algorithmen
  - e.g. GPU computing
- Strong Scaling
  - schwieriger zu realisieren
  - weniger Arbeit pro Prozessor
    - weniger Daten pro Prozessor und Re-Use
    - mehr Overhead für Task-Verwaltung

Skalierbarkeit

- **Speedup** $S(P)$
  - linear → optimal, möglich
  - sublinear → üblich
    - Amdhal
    - Zusatzaufwand
  - nicht skalierbar
    - Overhead durch
      - Datenaustausch / Kommunikation
      - Synchronisation
  - superlinear
    - Cache Effekte (selten)



- Speedup
  - linear → SIMD und Vektorprozessing → möglich
  - sublinear
    → serieller Anteil fast immer vorhanden
    → Zusatzaufwand durch
      - Parallelisierung
      - Datenaustausch / Kommunikation
      - Synchronisation
  - nicht skalierbar
    - wenn Kommunikation und/oder Synchronisation über-proportional zunehmen
    - z.B. zu geringe Busbandbreite resp. zu viele Prozessoren am gleichen Bus
  - superlinear ... sehr selten
    - möglich, vor allem wegen Cache Effekten
      mehr Prozessoren → Problem hat in Cache Platz

Daten Transfers

- **Latenz und Bandbreite**
  - beschreiben Modell für "Message Transfers"

$$T_{transfer} = T_{latenz} + \frac{M}{Bandbreite}$$

  - T_{latenz} Zeit um eine Meldung der Länge 0 zu transferieren
  - M Anzahl Bytes pro Meldung
  - B Bandbreite → Anzahl Bytes pro Zeiteinheit

- **Diameter**
  - Anzahl Abschnitte in einem "Netzwerk" zwischen zwei Rechnerknoten (Sender und Empfänger)

- Transferzeit bei mehreren "Hops"
  - Diameter x $T_{transfer}$

... Daten Transfer

- **Beispiel: Zugriff auf Speicher (Cache Line)**
  - DDR3 (typ. Werte): $T_{latenz} \approx 60ns, \quad B \approx 20GB/s$
  - Cache Line: 64 Bytes

$$T_{transfer} = T_{latenz} + \frac{M}{Bandbreite} = 60ns + \frac{64}{20GB/s} = 63.2ns$$

  - einfacher Benchmark: pointer chasing (linked list in array)

$$T_{benchmark} \sim 75ns$$

```
accessMemWhile:             /* pointer chasing             */
   mov rdi, [rdi]           /* get pointer to next element */
   dec rsi                  /* decrement counter           */
   ja  accessMemWhile       /* repeat until done           */
   ret
```


- Benchmark: Pointer Chasing
  - eine "linked list" mit Arrays
  - a[i] enthält Pointer auf a[j]
  - a[i] und a[j] in verschiedenen Cache Lines
  - Distanz i und j: d = (N-1)/4
  - Berechnung j: j = (i + d) % N
  - Zugriff in C: ptr = (char **)(*p)
  - Realisierung in Assembler: siehe oben

- Sequentielle Adressierung des Arrays
  - 4.0 ns pro Cache line
  - 0.6 ns pro double / long / pointer
  - 0.4 ns pro int
  - 0.4 ns pro byte

- Daten zu Speicherzugriffe: Intel i7

  http://software.intel.com/en-us/forums/topic/287236

  http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

Arithmetic Density

- **Anzahl Instruktionen / Anzahl Datenzugriffe**

$$D = \frac{i}{r + w}$$

i: instructions, r: reads, w: writes

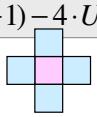
Caching nicht berücksichtigt

- **Beispiel: Laplace Operator**

$$\nabla^2 U = \left(\frac{d^2 U(x,y)}{dx^2} + \frac{d^2 U(x,y)}{dy^2} \right)$$

- diskretisiert

$$\nabla^2 U(x,y) = U(x-1,y) + U(x+1,y) + U(x,y-1) + U(x,y-1) - 4 \cdot U(x,y)$$



"stencil"

- arithmetische Dichte D

$$D = \frac{5}{5+1} = \frac{5}{6} = 0.83$$

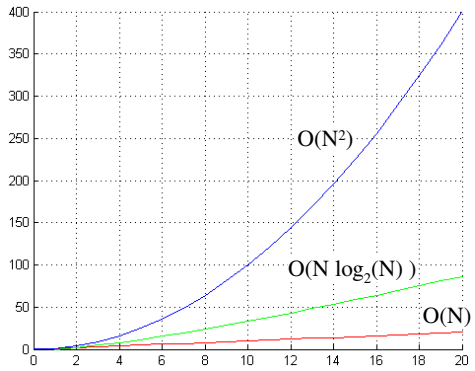
- Diskussion
  - arithmetische Dichte → ein relativ ungenaues Mass
  - Performance: abhängig davon, ob Datenzugriffe auf Daten in Cache oder Memory
  - gibt Hinweis, ob Anwendung "computation bound" oder "data bound"
  - je grösser, desto besser
- Zugriffsmuster → Stencil (Schablone)
  - häufig bei der Lösung von Differentialgleichungen anzutreffen
  - z.B. Heat Diffusion, Wettersimulationen, etc.

- Hinweis zu Dichte:
  - Caching "pre-loads" Data
    - exakte Berücksichtigung schwierig
  - obiges Beispiel (best case): 1 r → 1 w => Dichte D = 2.5

Komplexität: O-Notation

- **O-Notation beschreibt**
  - die asymptotische Komplexität eines Problems
  - Zusammenhang zwischen Problemvergrösserung und Laufzeitzunahme
  - z.B. Verdoppelung des Problems → doppelte Laufzeit: linear
- **Wichtige Klassen**

Klasse	Name
O(1)	konstant
O(N)	linear
O(log N)	logarithmisch
$O(N^2)$	quadratisch
$O(2^N)$	exponentiell



- Hinweise
  - die Notation gilt für grosse N
    - was gross ist, muss von Fall zu Fall entschieden werden
  - Komplexität → guter "Hinweis"
    - deckt nicht alle Aspekte ab
    - berücksichtigt Anzahl der Instruktionen , **aber**
      → nicht Komplexität
      → nicht Cache Verhalten
      → Skalierbarkeit
      → Anzahl Prozessoren
      → etc.
  - für Anzahl Instruktionen → korrekte Funktionen verwenden (z.B. log2 statt log)
- Beispiele
  - O(N): komponentenweise Mul zweier Vektoren
  - $O(\log_2 N)$: Summe der Vektorelemente
  - $O(N^2)$: Filterung eines Bildes
  - $O(N \log_2 (N))$: Quicksort (Mittelwert)
- Siehe auch Theoretische Informatik (W.Weck)

Take Home