Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 7b, Mittwoch, 14. Juni 2017

(Fortsetzung Verkettete Listen, Cache-Effizienz)

Prof. Dr. Hannah Bast
Lehrstuhl für Algorithmen und Datenstrukturen
Institut für Informatik
Universität Freiburg

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Inhalt
  - Verkettete Listen — Fortsetzung von gestern
  - Listen vs. Felder — Laufzeitvergleich + Analyse
  - Lokalität Speicherzugriffe — Definition + Hintergründe
  - Blockoperationen — Alternatives Effizienzmaß

Lokalität Speicherzugriffe 1/4

UNI FREIBURG

- Einfach(st)es Beispiel
  - Wir addieren die n Elemente eines Feldes auf

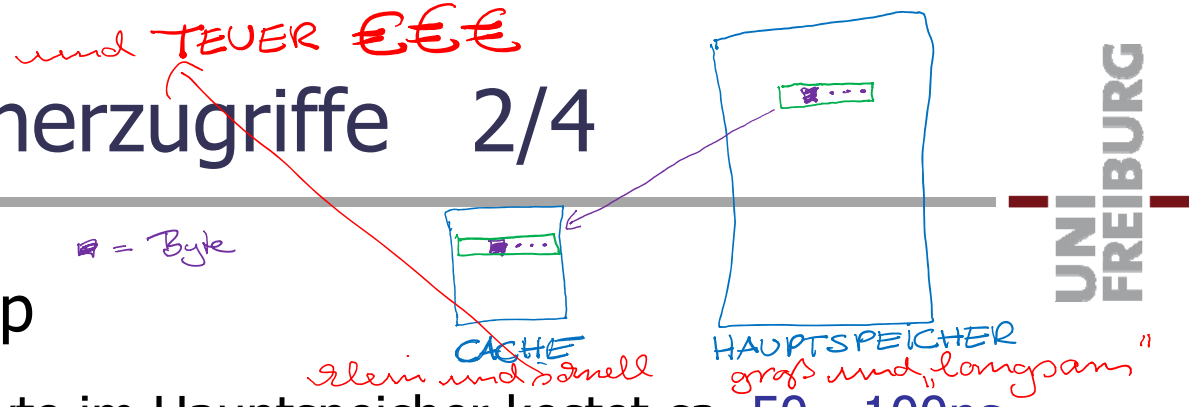
    … in der natürlichen Reihenfolge: 1 + 2 + 3 + 4 + 5

    … in einer zufälligen Reihenfolge: 2 + 5 + 3 + 1 + 4
  - Das Ergebnis ist in beiden Fällen **identisch**
  - Die Anzahl der Operationen ist ebenfalls **identisch**
  - Beobachtung:

$n = 10.000.000$: „random" ca. 10 mal LANGSAMER

$n = 100.000.000$: „random" ca. 20 mal LANGSAMER

Lokalität Speicherzugriffe 2/4



- CPU Cache, Prinzip
  - Zugriff auf ein Byte im Hauptspeicher kostet ca. 50 - 100ns
  - Zugriff auf ein Byte im Level-1 (L1) Cache kostet ca. 1ns
  - Bei Zugriff auf ein oder mehrere Bytes im Hauptspeicher holt man gleich einen ganzen Block (cache line) in den Cache

    Größe einer cache line für x86 Prozessoren: **64 Bytes**
  - Solange dieser Block im Cache ist, braucht man für Bytes aus diesem Block nicht mehr auf den Hauptspeicher zuzugreifen
  - Der Cache hat Platz für viele solcher Blöcke

    Typische Größe eines L1-Cache: **32 KB** (= 500 cache lines)

    Unter LINUX: getconf -a | grep CACHE

Lokalität Speicherzugriffe 3/4

- Disk Cache, Prinzip
  - "Seek Time" ist **~ 5ms** (HDD) bzw. **~ 0.1ms** (SSD)

    Bei HDD: Lesekopf muss an die Stelle bewegt werden
  - Transferrate ist **~ 50 MB/s** (HDD) bzw. **~ 200 MB/s** (SSD)

    Bei HDD: Kopf bleibt stehen, Platte dreht sich schnell weiter
  - Deshalb geht das Betriebssystem wie folgt vor

    Wird ein Byte von der Platte gelesen, wird gleich ein ganzer Block eingelesen … z.B. 128 KB auf einmal

    Solange dieser Block im Speicher ist, braucht man für Bytes aus diesem Block nicht mehr auf die Platte zugreifen

    Also dasselbe Prinzip wie beim CPU Cache

Lokalität Speicherzugriffe 4/4

UNI FREIBURG

- Wenn der Cache voll ist
  - … muss einer der Blöcke entfernt werden, dafür gibt es zahlreiche Strategien, zum Beispiel:

    LRU (Least Recently Used) = der Block, für den es am längsten her ist, das darauf zugegriffen wurde

    LFU (Least Frequently Used) = der Block, auf den am wenigsten zugegriffen wurde, seit er im Cache ist
  - Das ist aber nicht das Thema der Vorlesung heute

    Für unsere einfachen Analysen heute und für das ÜB7 spielt es keine Rolle, welche Strategie verwendet wird

Blockoperationen 1/8

UNI FREIBURG

- Abstraktion der bisherigen Beobachtungen
  - Es gibt einen **langsamen** und einen **schnellen** Speicher
  - Beide Speicher sind in Blöcke der Größe B unterteilt
  - Der schnelle Speicher ist M groß = Platz für M/B Blöcke
  - Stehen die Daten nicht im schnellen Speicher, wird der entsprechende Block in den schnellen Speicher geladen
  - Das Programm kann sich aussuchen, welche Blöcke im schnellen Speicher gehalten werden
  - Wir zählen nur die **Anzahl der Blockoperationen**

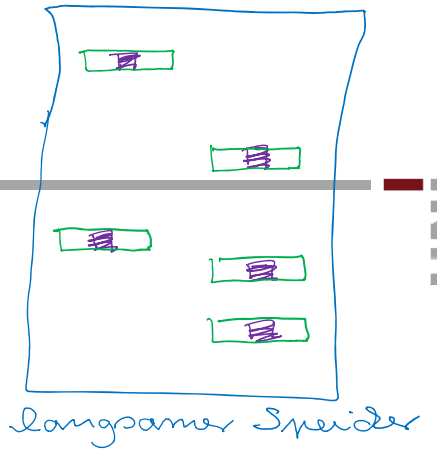
    Also +1 für jedes Mal, wenn ein Block in den schnellen Speicher geladen wird, der da gerade nicht drin steht

Blockoperationen 2/8

UNI FREIBURG

- Was wir alles vernachlässigen
  - Sämtliche Berechnung auf einem Block im schnellen Speicher
  - Kosten für das Verwalten der Blöcke im schnellen Speicher
  - Wie genau der langsame Speicher in Blöcke unterteilt ist
  - Ob eine Operation 1, 2, 4 oder 8 Bytes liest
  - Grund 1: die Zeit, die man braucht, um einen Block in den schnellen Speicher zu holen, dominiert oft alles andere
  - Grund 2: die anderen Sachen machen nur einen konstanten Faktor Unterschied → egal für $O(\ldots)$ oder $\Theta(\ldots)$ Schranken

Blockoperationen 3/8



- Gute vs. Schlechte Lokalität
  - Für B Operationen hat man also:

    Im "best case" nur 1 Blockoperation **gute Lokalität**

    Im "worst case" B Blockoperationen **schlechte Lokalität**
  - Sonderfall: wenn ein Algorithmus Platz $s \leq M$ braucht (dann passt insbes. die Eingabe ganz in den schnellen Speicher), hat man trivial $\lceil s/B \rceil$ Blockoperationen

    Eine Analyse der Anzahl Blockoperationen ist also erst für große bzw. sehr große Eingaben ($n \gg M$) interessant

Blockoperationen 4/8

- Typische Werte (für einen Server)
  - CPU L1-Cache: B = 64 Bytes, M = 32 KB — 512 blocks
    CPU L2-Cache: B = 64 Bytes, M = 256 KB — 4096 blocks
    CPU L3-Cache: B = 64 Bytes, M = 8 MB — 131072 blocks
  - Disk Cache: B = 64 KB, M = 64 GB — ≈ 1 million blocks

    Die meisten Betriebssysteme benutzen alles, was vom Hauptspeicher gerade nicht genutzt wird, als Disk Cache
  - Sinnvollerweise wählt man dabei B so, dass die transfer time für einen Block ungefähr gleich der seek time ist

    Wenn man schon die viele Zeit für ein "seek" aufwendet, kann man auch gerade noch mal so viel Zeit aufwenden, um möglichst viele Elemente auf einmal zu lesen

Blockoperationen 5/8

UNI FREIBURG

- Terminologie
  - Blockoperationen nennt man beim

    CPU Cache: in der Regel **cache misses**

    Weil sie dann nötig werden, wenn ein Stück vom (langsamen) Hauptspeicher nicht im (schnellen) Cache ist

    Disk Cache: oft einfach **IOs**

    IO oder I/O = Input/Output … eher historische Bezeichnung für Datentransfer von der oder auf die Platte
  - Wenn man die Anzahl Blockoperationen eines Algorithmus analysiert, spricht man deswegen oft von seiner

    Cache-Effizienz oder **IO-Effizienz**

Blockoperationen 6/8

Blockgröße B

$n >> M$

- IO-Effizienz von **ArraySumMain** 1/2
  - Wenn wir über die n Elemente in der Reihenfolge 1, 2, 3, ... iterieren, dann ist die Anzahl Blockoperationen

    im best case: $\lceil n/B \rceil$

    im worst case: $\lceil n/B \rceil$
  - Wenn wir über die n Elemente in einer zufälligen Reihenfolge iterieren, dann ist die Anzahl Blockoperationen

    im best case: $\lceil n/B \rceil$ ... aber sehr unwahrscheinlich bei $n >> M$

    im worst case: n

Blockoperationen 7/8

UNI FREIBURG

- IO-Effizienz von **ArraySumMain** 2/2
  - In der Praxis ist der Unterschied zwischen den beiden Varianten kleiner als die Blockgröße (B = 64)
  - Grund: der durchschnittliche Fall (average case) liegt irgendwo zwischen best case und worst case

    Wenn **n** nicht viel viel größer als **M** ist, steht das nächste Element manchmal schon zufällig im schnellen Speicher

    Außerdem wird auch bei zufälliger Reihenfolge pro Element auf 4 benachbarte Bytes (ein **int**) auf einmal zugegriffen

Blockoperationen 8/8

- IO-Effizienz von **MergeSort**
  - Kurze Wiederholung der Funktionsweise:

    Teile das Feld in zwei gleich große Teile

    Sortiere die beiden Teile rekursiv

    Mische die beiden sortierten Folgen zu einer sortierten Folge
  - Mischen von zwei Folgen der Gesamtlänge n geht mit $IO(n) \leq \lceil n/B \rceil$ Blockoperationen
  - Außerdem ist $IO(n) = 1$ für $1 \leq n \leq B$
  - Durch Auflösen der Rekursion kann man dann zeigen, dass $IO(n) = \Theta(n/B \cdot \log_2(n/B))$
  - Man kann sogar zeigen: $IO(n) = \Theta(n/B \cdot \log_{M/B}(n/B))$

Literatur / Links

- Cache-Effizienz / IO-Effizienz
  - In Mehlhorn/Sanders:

    2 Introduction 2.2.1 External Memory
  - In Wikipedia

    http://en.wikipedia.org/wiki/Cache

    http://de.wikipedia.org/wiki/Cache

    Da wird das Prinzip eines Caches beschrieben, es gibt aber keinen separaten Artikel zur Cache-Effizienz bei Algorithmen