

# Externspeicher- Algorithmen

VO Algorithm Engineering

Professor Dr. Petra Mutzel

Lehrstuhl für Algorithm Engineering, LS11

12. VO, 2. Teil

5.12.2005

## Literatur für diese VO

- U. Meyer, P. Sanders und J. Sibeyn (Eds.), Algorithms for Memory Hierarchies, Advances Lectures, Lecture Notes in Computer Science 2625, Springer 2003:
  - Kapitel 1: P. Sanders: Memory Hierarchies --- Models and Lower Bounds, S. 1-13
  - Kapitel 2: R. Pagh: Basic External Memory Data Structures, S. 14-35

# Überblick

- Motivation Externspeicher-Algorithmen
- Das Externspeichermodell
- Grundlegende Externe Datenstrukturen
  - Stacks
  - Queues
  - Lineare Listen

## Durchwandern eines Arrays

```
for (i=0; i<N; i++) D[i]=i;  
C=Permute(D)
```

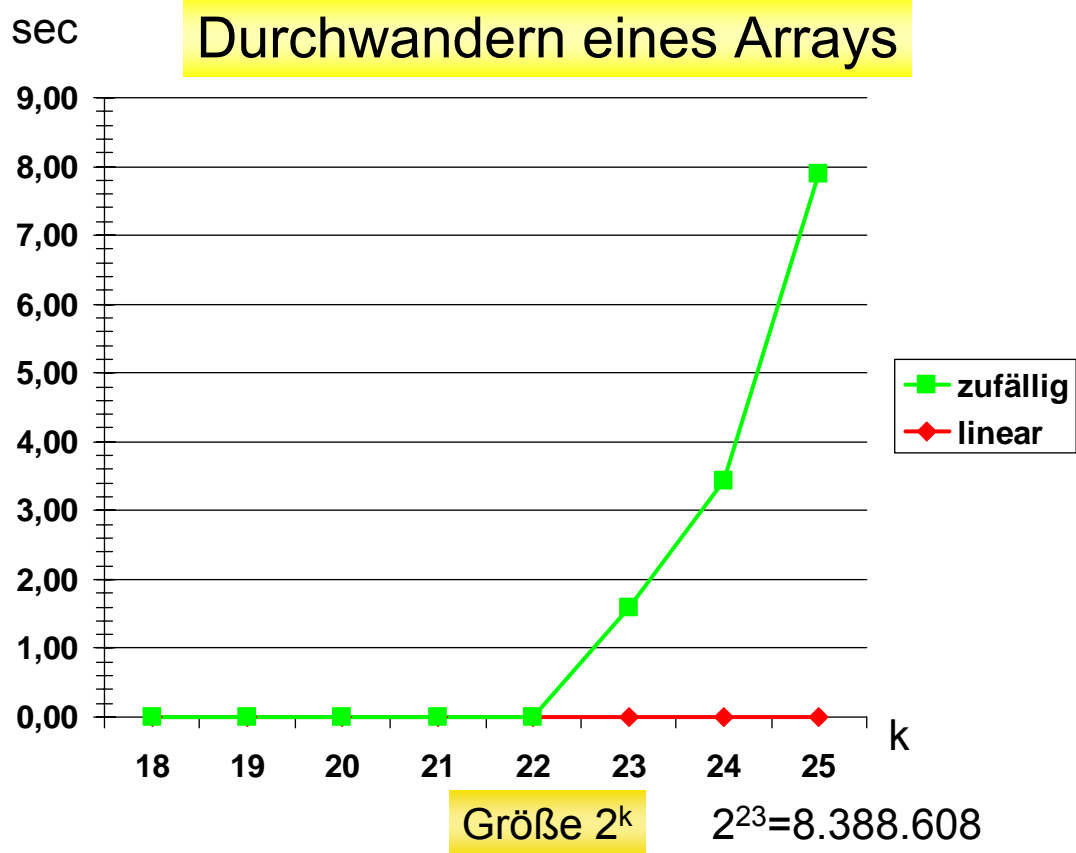
Lineares Durchlaufen:

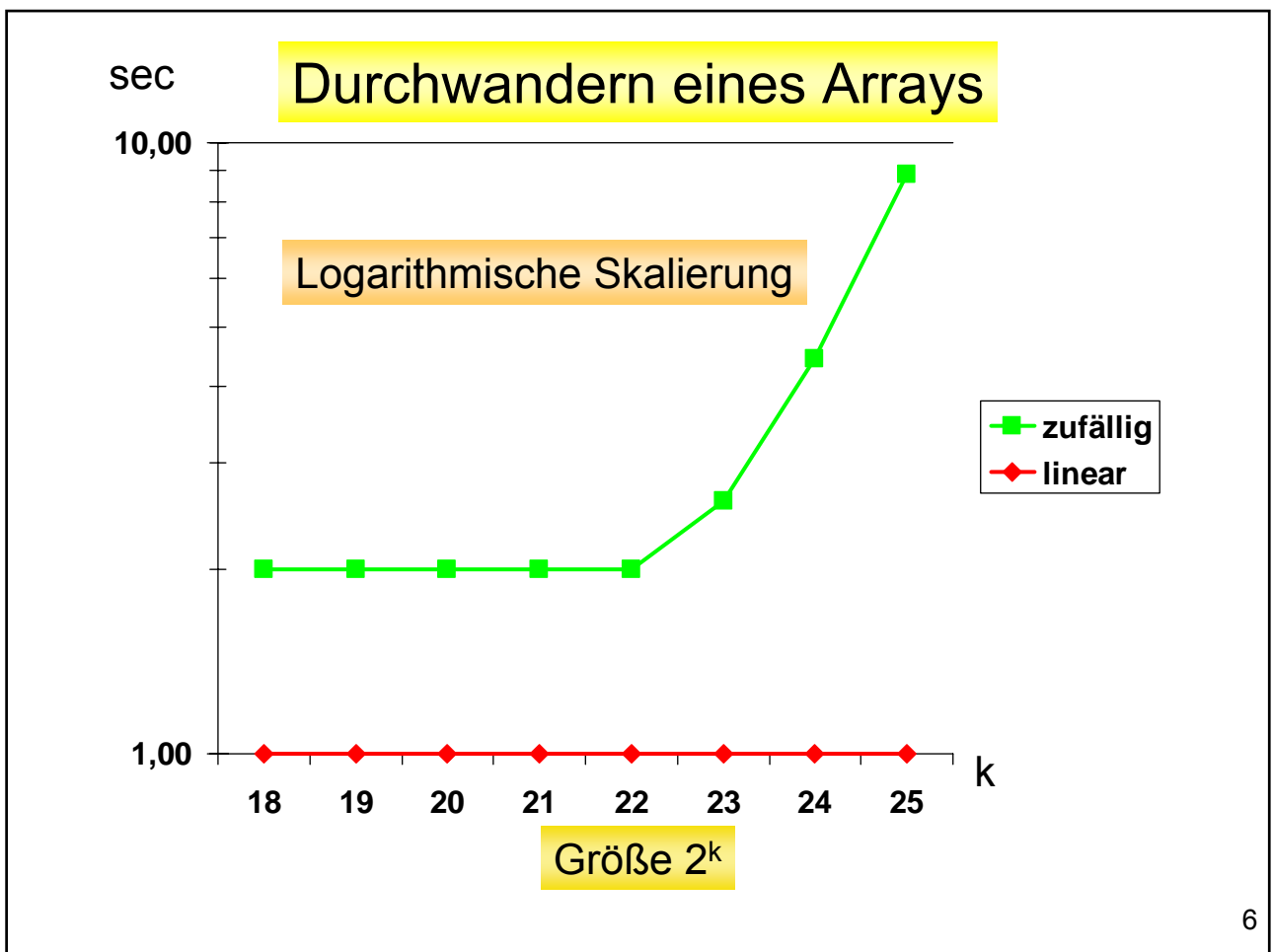
```
for (i=0; i<N; i++) A[D[i]]=A[D[i]]+1;
```

Zufälliges Durchlaufen:

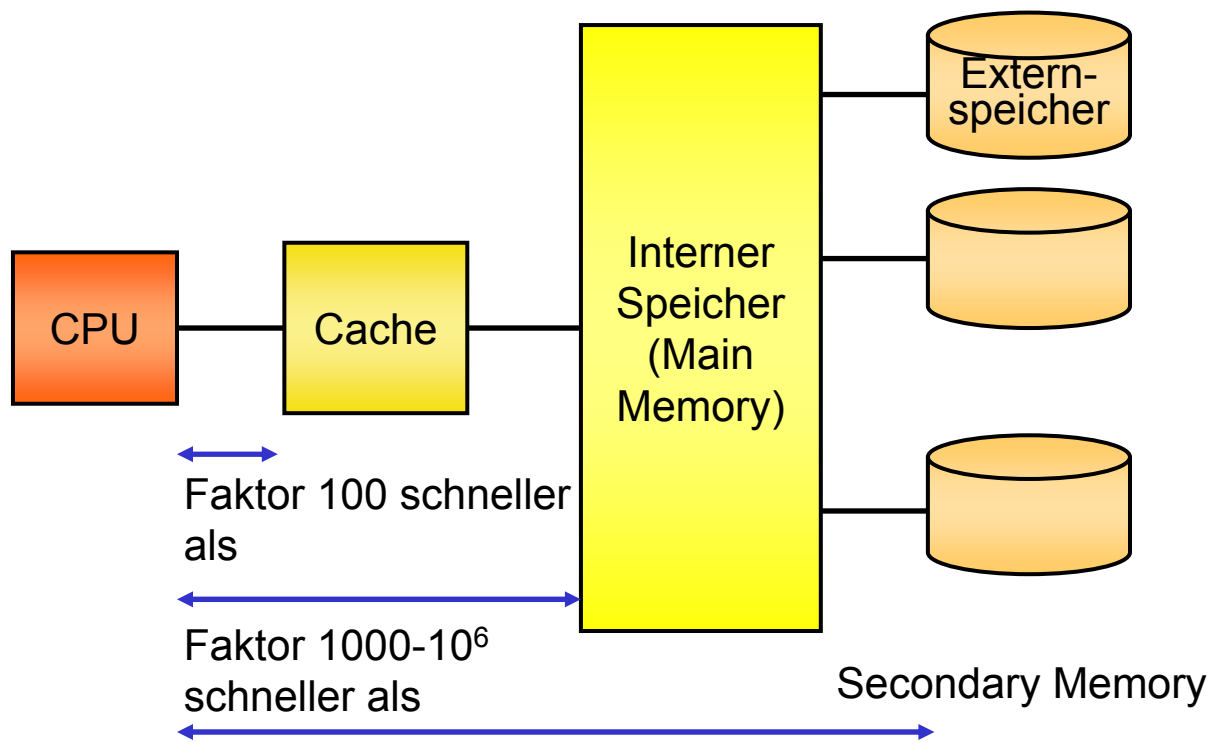
```
for (i=0; i<N; i++) A[C[i]]=A[C[i]]+1;
```

## Durchwandern eines Arrays





## Hierarchisches Speichermodell moderner Computer



## Probleme klassischer Algorithmen

- Ein Zugriff im Hauptspeicher spricht jeweils eine Speicherzelle an und liefert jeweils eine Einheit zurück
- Ein Zugriff im Externspeicher (ein I/O) liefert jeweils einen ganzen Block von Daten zurück
- Meist keine Lokalität bei Speicherzugriffen, und deswegen mehr Speicherzugriffe als nötig

## Problem ist aktueller denn je, denn

- Geschwindigkeit der Prozessoren verbessert sich zwischen 30%-50% im Jahr;
  - Geschwindigkeit des Speichers nur um 7%-10% pro Jahr
- 
- „One of the few resources increasing faster than the speed of computer hardware is the amount of data to be processed.“

Donald E. Knuth: The Art of Computer Programming  
1967 (Neuaufgabe 1998):

„When this book was first written, magnetic tapes were abundant and disk drives were expensive. But disks became enormously better during the 1980s,... . Therefore the once-crucial topic of patterns for tape merging has become of limited relevance to current needs. Yet many of the patterns are quite beautiful, and the associated algorithms reflect some of the best research done in computer science during its early years;

The techniques are just too nice to be discarded abruptly onto the rubbish heap of history. ...

Therefore merging patterns are discussed carefully and completely below, in what may be their last grand appearance before they accept a final curtain call.“

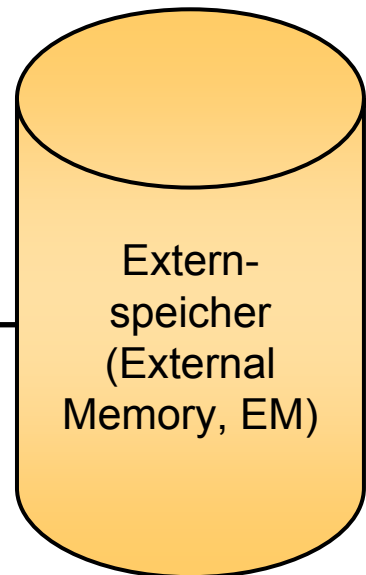
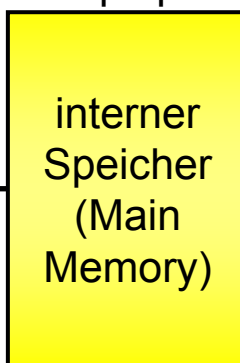
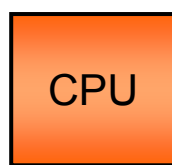
Pavel Curtis in Knuth: ``The Art of Computer Programming'' 1967 (Neuaufgabe 1998):

„For all we know now, these techniques may well become crucial once again.“

## Das Externspeichermodell

Modell von Aggarwal, Vitter  
und Shriver 1994

$M$  = Anzahl der  
Elemente im  
Hauptspeicher



Rechenoperationen können  
nur mit Daten im Haupt-  
speicher getätigt werden

1 I/O

Annahme:  $B < M/2$



= Anzahl der Elemente,  
die in einen Block passen

12

## Analyse von Externen Algorithmen

- Anzahl der ausgeführten I/O-Operationen
- Anzahl der ausgeführten CPU-Operationen im RAM-Modell
- Anzahl der belegten Blöcke auf dem Sekundärspeicher

## Ziele beim Entwurf Externer Algorithmen

- **Interne Effizienz:**
  - Anzahl der RAM-Operationen vergleichbar zu den besten internen Algorithmen
- **Örtliche Lokalität:**
  - Ein r/w Block sollte möglichst viele nützliche Daten enthalten
- **Zeitliche Lokalität:**
  - Daten, die im internen Speicher sind, sollten möglichst verarbeitet werden, bevor sie wieder herausgeschrieben werden.

## Externe Datenstrukturen: Stacks

- Ein Stack  $S$  repräsentiert eine dynamische Menge von Elementen (maximal  $N$ )

- **Operationen:**

- Insert( $x$ ): Einfügen eines neuen Elements in  $S$
- Delete: Ausgabe und Entfernung des letzten eingefügten Elements aus  $S$

Interne Algorithmen für Stack der Größe  $N$ :  
Array der Länge  $N$  und zwei Zeiger

Im Worst Case: 1 I/O per Insert und Delete  
Operation

## Externe Stacks

- **Buffer für externe Stacks:**
  - Array im internen Speicher der Länge  $2B$
  - enthält zu jedem Zeitpunkt die letzten  $k$  eingefügten Elemente, wobei  $k \leq 2B$
- **Insert(x):**
  - Meistens 0 I/Os benötigt, außer: wenn Buffer voll ist
  - Dann: die  $B$  ältesten Elemente werden in EM ausgelagert.
- **Delete:**
  - Meistens 0 I/Os benötigt, außer: wenn Buffer leer
  - Dann: 1 I/O holt die nächsten  $B$  Elemente aus dem EM (diejenigen, die als letztes ausgelagert wurden)

## Externe Stacks

- **Analyse der Operationen:**
  - Insert(x):  $1/B$  I/Os amortisiert
  - Delete:  $1/B$  I/Os amortisiert
- **Dies ist bestmöglich!**
- Denn: mit einer I/O können nicht mehr als  $B$  Elemente gleichzeitig gespeichert oder gelesen werden
- **ZIEL für externe Datenstrukturen: 1 I/O**

# Externe Datenstrukturen: Queues

- **Operationen:**
  - Insert(x): Einfügen eines neuen Elements in S
  - Delete: Ausgabe und Entfernung des ältesten Elements aus S
- **Zwei Buffer: R und W Buffer:**
  - Zwei Arrays im internen Speicher der Länge jeweils B
- **Insert(x):** Analyse:  $1/B$  I/Os
  - zu W Buffer, außer: wenn voll
  - Dann: schreibe alle B Elemente in EM
- **Delete:** Analyse:  $1/B$  I/Os
  - aus R Buffer, außer: wenn Buffer leer
  - Dann: 1 I/O holt die nächsten B Elemente aus dem EM (wenn keine dort, dann aus W Buffer (Buchführung!))

18

## Externe Datenstrukturen: Lineare Listen

- Eine lineare Liste  $L$  liefert eine effiziente Implementierung für dynamische geordnete Listen von Elementen

- **Operationen:**

- $\text{Search}(x)$ : Sucht ein Element in  $L$  mit Schlüssel  $x$
- $\text{Insert}(x)$ : Einfügen in  $L$  eines neuen Elements mit Schlüssel  $x$  an die richtige Stelle
- $\text{Delete}(x)$ : Entfernung des Elements mit Schlüssel  $x$  aus  $L$

Interne Algorithmen für Lineare Listen:

Im Worst Case: 1 I/O für jeden einzelnen Zeiger, der verfolgt wird

## Lineare Listen

- **Idee: Erhalte räumliche Lokalität:**
  - Füge jeweils ungefähr  $k$  konsekutive Listenelemente in einem Block zusammen
  - Verlinke jeweils benachbarte Blöcke
- **Konkrete Umsetzung:**
  - **Invariante:** In jedem Paar konsekutiver Blöcke existieren mehr als  $2/3B$  konsekutive Listen-Elemente
- **Search(x):**
  - Traversiere die Liste bis zur richtigen Stelle:  $O(N/B)$  I/Os
  - Die Einführung der Invariante führt zu einer Erhöhung der benötigten I/Os auf höchstens  $3N/B$ .

## Lineare Listen

- **Insert(x):** Analyse:  $O(1+N/B)$  I/Os
  - Traversiere die Liste bis zur richtigen Stelle:  $O(N/B)$  I/Os
  - Meistens 1 I/Os benötigt, außer: wenn Block voll ist
  - Dann: Falls ein benachbarter Block noch Platz hat: tausche 1 Element aus:  $O(1)$  I/Os
  - Sonst: Teile den Block in 2 Teile der Größe  $B/2$ :  $O(1)$  I/Os

Danach sind mind.  $B/6$  Delete-Operationen notwendig, damit L an dieser Stelle wieder verschmolzen werden muß

- **Delete(x):** Analyse:  $O(1+N/B)$  I/Os
  - Traversiere die Liste bis zur richtigen Stelle:  $O(N/B)$  I/Os
  - Meistens 1 I/Os benötigt, außer: wenn B danach mit benachbarten Blöcken  $\leq 2/3B$  Elemente besitzt:
  - Dann: Verschmelze die beiden Blöcke:  $O(1)$  I/Os

ENDE