

Kapitel 7

Algorithmen für große Datenmengen

In den letzten Jahren werden immer größere Datenmengen gesammelt und verarbeitet. Denken wir an die riesigen Mengen an DNA-Kodierungen, die jedes Jahr in der Molekularbiologie gesammelt werden, oder die Daten, die jeden Tag von Satelliten zur Erde gesendet werden. Andere Schlagwörter sind: Wissensdatenbank, Internet-Daten oder Verlagsdatenbanken. Die Datenmengen sind hierbei so riesig, dass sie nicht mehr im Arbeitsspeicher Platz haben. Deswegen liegen diese auf Externspeichern, wie z.B. magnetische Festplatten. Die Algorithmen, die wir bisher kennengelernt haben, werden für das RAM-Modell entwickelt. Dabei ist die Annahme, dass es unbegrenzten Speicher gibt und jeder Speicherzugriff gleich teuer ist.

Abbildung 7.1 zeigt das hierarchische Speichermodell moderner Computer und typische Kapazitäten der einzelnen Speichereinheiten. Hauptspeicherzugriffe sind ungefähr 100 Mal langsamer als Cache-Zugriffe, und Externspeicherzugriffe (im folgenden: I/O genannt) sind ungefähr 1000 Mal langsamer als Hauptspeicherzugriffe und somit bis zu 100.000 mal langsamer als Cache-Zugriffe.

Das Problem ist aktueller denn je, denn:

- (1) Die Geschwindigkeit der Prozessoren verbessert sich zwischen 30%-50% im Jahr; die Geschwindigkeit des Speichers hingegen verbessert sich nur um 7%-10% pro Jahr
- (2) Es gibt vermehrt wichtige large-scale Anwendungen, wie z.B. Geographische Informationssysteme, Bioinformatik (DNA, Datenbank), Suchen im Web

Hierzu ein aktuelles Zitat aus der Informatik-Community: “One of the few resources increasing faster than the speed of computer hardware is the amount of data to be processed.”¹

Ein Speicherzugriff vom Arbeitsspeicher in den Cache bzw. vom Externspeicher (Sekundärspeicher) in den Arbeitsspeicher liefert jeweils einen ganzen **Block** von Daten zurück. Dies wird jedoch von den klassischen Algorithmen nicht beachtet. Wendet man die

¹Aus dem Call-for-Papers für die IEEE InfoVis 2003

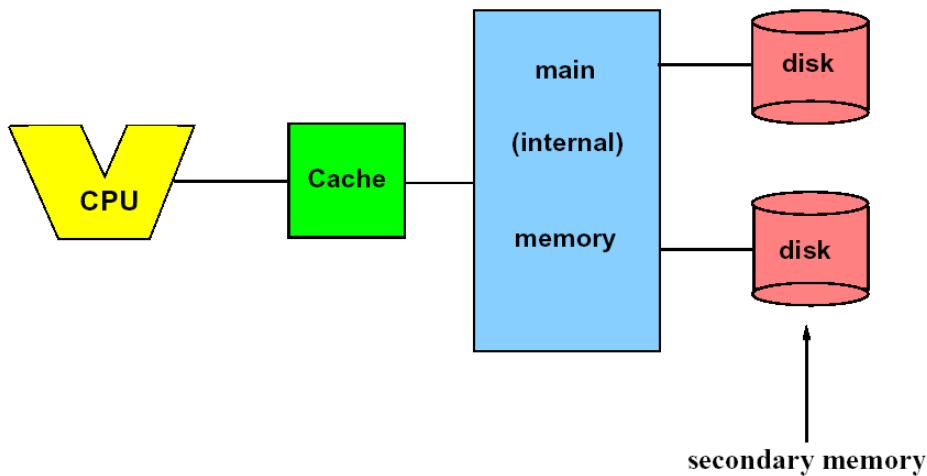


Abbildung 7.1: Hierarchisches Speichermodell moderner Computer

klassischen Algorithmen auf große Datenmengen an, dann erhält man meist keine Lokalität bei Speicherzugriffen, was zu sehr vielen unnötigen Speicherzugriffen führt. Das folgende Beispiel soll die Problematik verdeutlichen:

Beispiel: Wir betrachten die folgenden Programmstücke. Dabei seien B und C Felder der Größe N (N sehr groß), die als Indexarrays benutzt werden, um das Feld A durchzulau-
fen. `RandomPermute(B)` nimmt das Feld B und permutiert die Inhalte zufällig. D.h. nach dem Aufruf enthält das Feld C die gleichen Inhalte wie das Feld B , nur in einer zufälligen Reihenfolge.

- (1) Für $i = 0$ bis $N - 1$ tue: $B[i] = i$
- (2) $C = \text{RandomPermute}(B)$;
- (3) Für $i = 0$ bis $N - 1$ tue: $A[B[i]] = A[B[i]] + 1$;
- (4) Für $i = 0$ bis $N - 1$ tue: $A[C[i]] = A[C[i]] + 1$;

Die beiden Programmstücke (3) und (4) machen genau das gleiche: sie laufen jeweils alle Elemente des Feldes A ab und zählen 1 dazu. Allerdings läuft (3) das Feld A linear durch, während (4) das Feld A zufällig durchläuft. Abbildung 7.2 zeigt die Laufzeitunterschiede für die beiden Programmstücke. Die Laufzeittests wurden auf einem Rechner mit CPU 2.40GHz und Cache-Größe 512 KB ausgeführt. Obwohl die Komplexität beider Programmstücke äquivalent ist, so ist doch die zweite Version bis zu 20 Mal (!) langsamer als die erste. Zum Beispiel ist die Laufzeit für $N = 2^{25} = 33.554.432$ für das lineare Durchwandern 0,39 Sekunden, während sie für ds zufällige Durchwandern 7,89 beträgt.

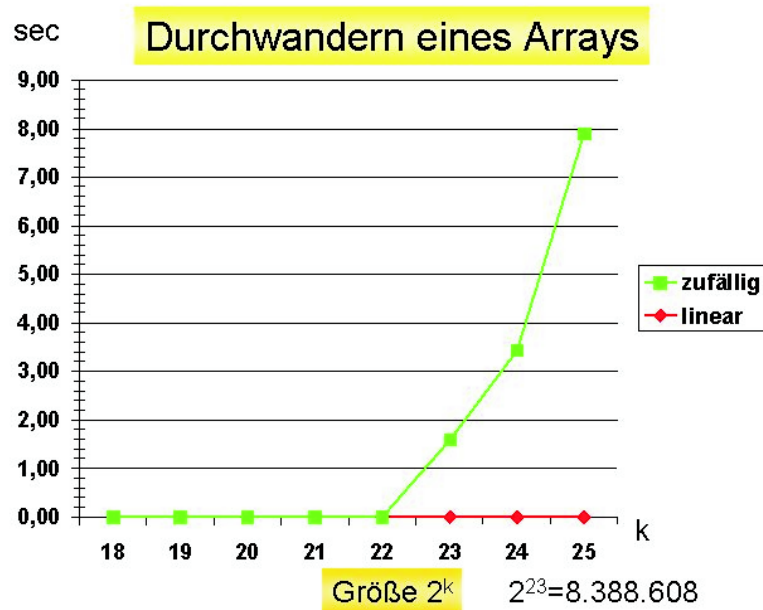


Abbildung 7.2: Laufzeitunterschiede zwischen linearem und zufälligem Durchwandern eines Feldes der Größe 2^k

Dies ist ein Faktor von 20 (!). Während man im ersten Programmstück für $N = 34$ Mio. ca. 66 Festplattenzugriffe hat (weil ein Zugriff nicht nur den Inhalt von $A[1]$, sondern auch die Inhalte von $A[2]$, $A[3]$, . . . , in den Cache bringt, so dass die folgenden Additionsaufrufe keine eigenen Zugriffe benötigen), hat man im zweiten Fall tatsächlich ungefähr $N = 34$ Mio. Zugriffe.

Bereits dieses einfache Programmstück zeigt, dass es bereits für einfache „FOR“-Schleifen wichtig ist, auf Lokalität der Speicherzugriffe zu achten, wenn man es mit großen Datenmengen zu tun hat.

In diesem Kapitel geht es also um die Entwicklung von Algorithmen und Datenstrukturen, die die Lokalität der Datenzugriffe beachten. In Kapitel 3.1 werden externe Speichermodelle und ein Beispiel eines Externspeicheralgorithmus diskutiert und analysiert. Kapitel 3.2 beschäftigt sich mit sogenannten *Cache-optimalen* Algorithmen.

7.1 Externspeicher-Algorithmen

7.1.1 Virtuelles Speichermanagement des Betriebssystems

Der virtuelle Speicher ist in *Seiten (pages)* fester Größe eingeteilt. Diese befinden sich teilweise im Hauptspeicher und teilweise im Sekundärspeicher. Sobald Seiten im Sekundärspeicher angesprochen werden, werden diese in den Hauptspeicher transportiert (*page in*), im Gegenzug muss dafür ein nicht mehr benötigtes Stück vom Hauptspeicher in den Sekundärspeicher kopiert werden (*page out*). Der Arbeitsspeicher wird als Pool von virtuellen Seiten gesehen, die entweder frei oder belegt sind.

Das Betriebssystem versucht nun durch intelligente Strategien (*caching and prefetching*), den I/O-Engpass zu minimieren. Jedoch sind dies sehr allgemeine Algorithmen, die nicht auf das jeweilige Problem optimiert sind. Wenn nun klassische Algorithmen auf Sekundärspeicher zugreifen, dann geschieht dies meist unstrukturiert, d. h. auf Daten wird “durcheinander” zugegriffen (keine Lokalität der Zugriffe) auszunützen. Das Problem dabei ist, dass die meiste Zeit damit verbracht wird, die Daten zwischen externem und internem Speicher hin- und herzutransportieren.

Die sogenannten *Externspeicheralgorithmen (External Memory Algorithms)* sollen dieses Problem lösen. Sie gehen davon aus, dass der Speicher in einen begrenzten Hauptspeicher und in eine gewisse Anzahl von Sekundärspeicherplatten geteilt ist, die jeweils unterschiedliche Zugriffszeiten sowie Zugriffseigenschaften besitzen.

Während ein Zugriff im Hauptspeicher immer jeweils 1 Einheit benötigt und eine Speicherzelle anspricht, geht man davon aus, dass ein Zugriff im Sekundärspeicher deutlich länger benötigt, und jeweils einen zusammenhängenden Datenblock zurückliefert.

7.1.2 Das theoretische Sekundärspeichermodell (EM-Modell)

Das EM-Modell benutzt die folgenden Parameter:

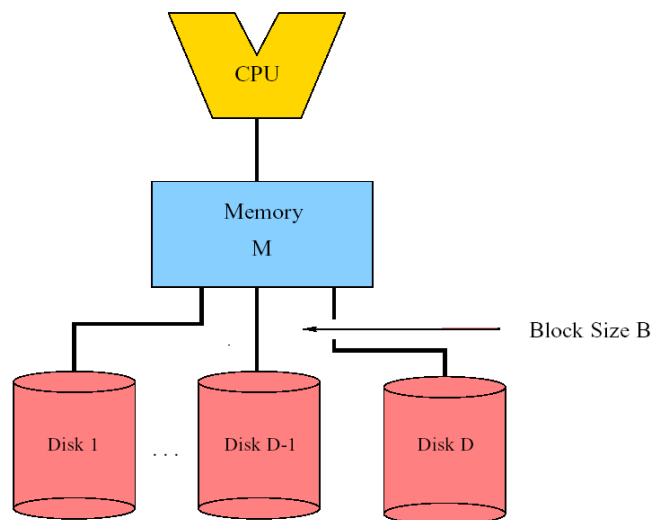
- N = Anzahl der Elemente in der Input-Instanz
- M = Anzahl der Elemente, die im Hauptspeicher passen
- B = Anzahl der Elemente, die in einen Block passen,

wobei $M < N$ und $1 \leq B \leq M/2$.

Das älteste theoretische Sekundärspeichermodell wurde 1988 von Aggerwal und Vitter vorgeschlagen. Es sieht vor, dass ein Rechner aus einer CPU mit einem schnellen Hauptspeicher der Größe M besteht. Der Sekundärspeicher wird als eine Platte (*disk*) modelliert, die durch

$P \geq 1$ voneinander unabhängige Zugriffsköpfe (Lese- und Schreibköpfe) angesprochen werden kann. Ein externer Zugriff (I/O) bewegt jeweils B Elemente von der Platte zum Hauptspeicher oder umgekehrt. Falls $P > 1$, dann können also $P * B$ Elemente in einer I/O-Operation bewegt werden. Dieses Modell ist jedoch nicht praxisnah, weil in der Regel bei Systemen mit mehr Zugriffsköpfen diese nicht unabhängig voneinander steuerbar sind.

Deswegen wurde dieses Modell 1994 von Vitter und Shriver folgendermaßen verändert. Dieses Modell ist momentan das Standardmodell in diesem Bereich und wird auch *Parallel Disk Modell* genannt. Denn es geht davon aus, dass D verschiedene, voneinander unabhängige Festplatten vorhanden sind. Nun ist es kein Problem, auch in der Praxis in einer I/O-Operation $D * B$ Elemente zu bewegen. Abb. 7.3(a) zeigt das *Parallel Disk Modell*.



(a)

Abbildung 7.3: Das *Parallel Disk* Speichermodell von Vitter und Shriver

Eine weitere Annahme beider EM-Modelle ist, dass Blöcke nicht geteilt werden können und dass Rechenoperationen nur mit Daten im Hauptspeicher getätigt werden können.

In diesem Modell basiert die Analyse von Algorithmen auf folgenden Parametern:

- (i) Anzahl der ausgeführten I/O-Operationen
- (ii) Anzahl der ausgeführten CPU-Operationen (RAM-Modell)
- (iii) Anzahl der belegten Blöcke auf dem Sekundärspeicher

In diesem Kapitel gehen wir immer von dem *Parallel Disk Modell* als unser EM-Modell aus und setzen der Einfachheit halber $D = 1$.

7.1.3 Untere Schranken im EM-Modell

Wir betrachten im folgenden die Mindestanzahl von notwendigen I/O-Operationen für simple Standardaufgaben (d.h. untere Schranken, $D = 1$).

Das Einlesen einer Menge von N Elementen benötigt mindestens $\Theta(N/B)$ I/O-Operationen.

Die Suche in dynamischen Daten von N Elementen benötigt mindestens $\Theta(\log N / \log B) = \Theta(\log_B N)$ I/O-Operationen.

Das Sortieren einer Menge von N Elementen benötigt mindestens $\Theta(\frac{N}{B} \log_{1+M/B}(1 + N/B))$ I/O-Operationen (ohne Beweis; Literaturtipp für Interessierte: A. Aggarwal und J.S. Vitter, The input/output complexity of sorting and related problems, *Communications of the ACM*, 1116–1127, 1988).

Bemerkung: Das EM-Standardmodell unterscheidet nicht zwischen zufälligen I/O-Plattenzugriffen und sequentiellen I/O-Zugriffen, obwohl letztere in der Praxis deutlich schneller sind. Es gibt EM-Modelle (z.B. von M. Farach, P. Ferragina und S. Muthakrishnan, Overcoming the memory bottleneck in suffix tree construction, *Proc. of the 39th Annual Symposium on Foundations of Computer Science*, 174–185, IEEE Computer Society 1998), die dies mit in Betracht ziehen.

7.1.4 Einfache Datenstrukturen

Im folgenden haben wir in der Vorlesung einfache externe Datenstrukturen wie Stack, Queue, und lineare Listen behandelt. Hierzu gibt es leider noch kein ausführliches Skript (s. Folien).

7.2 Externe Sortierverfahren

Hier haben wir in der Vorlesung die beiden wichtigsten externen Sortierparadigmen kennengelernt. Danach wurde ein externes Merge-Sort Verfahren vorgestellt und analysiert. Wir haben außerdem eine untere Schranke für externes Sortieren gezeigt. Auch hierzu gibt es leider noch kein ausführliches Skript (s. Folien).

7.3 Externspeicherdatenstruktur für Prioritätswarteschlangen

7.3.1 Prioritätswarteschlangen

Eine Prioritätswarteschlange *Priority Queue* ist eine Datenstruktur, die eine Menge von Elementen speichert, die aus einem Tupel *Information* und *Prioritätswert* (Schlüssel, *Key*) besteht. Diese Datenstruktur unterstützt die folgenden Operationen:

- *Get_Min*: Ausgabe der Elemente mit kleinstem Schlüssel
- *Del_Min*: Ausgabe und Entfernung des Elements mit kleinstem Schlüssel aus der Liste
- *Insert*: Einfügen eines neuen Elements in die Warteschlange

Prioritätswarteschlangen tauchen in einer Vielzahl von Anwendungen auf, z.B. in kombinatorischer Optimierung (z.B. bei Dijkstra's Algorithmus zur Bestimmung kürzester Wege), in Sortierverfahren (z.B. Heapsort), oder bei Scheduling bzw. Simulationen.

Es gibt eine Vielzahl von Realisierungen für Prioritätswarteschlangen. Wir haben z.B. in *Algorithmen und Datenstrukturen 1* einen binären Heap oder auch binäre Suchbäume wie AVL-Bäume kennengelernt.

Aufgabe: Überlegen Sie sich, wieviel Aufwand die Operationen *Get_Min*, *Del_Min*, und *Insert* jeweils in *theta*-Notation verursachen, wenn Sie für die Realisierung binäre Heaps bzw. binäre Suchbäume verwenden.

Als geeignete Datenstruktur für riesige Datenmengen haben wir bereits die B-Bäume kennengelernt. Im folgenden diskutieren wir die spezielle Externspeicherdatenstruktur *Externe Array-Heaps*. Wie wir sehen werden sind Externe Array-Heaps deutlich besser als die B-Baum-Datenstruktur als Realisierung für die Prioritätswarteschlange geeignet.

7.3.2 Externe Array-Heaps

Der externe Array-Heap besteht aus zwei Teilen: einer internen Datenstruktur (im folgenden Heap *H* genannt) im Arbeitsspeicher und einer externen Datenstruktur, die aus einer Menge von sortierten Feldern (*Arrays*) unterschiedlicher Länge besteht. Diese Felder sind in *L* Schichten $L_i, 1 \leq i \leq L$ eingeteilt; jede Schicht besteht aus $\mu = (cM/B) - 1$ Feldern (die im folgenden *Slots* genannt werden) der Länge $l_i = (cM)^i / B^{i-1}$ für $c < 1$ (z.B. $c = 1/7$). Jeder dieser Slots ist entweder leer oder er enthält eine sortierte Folge von höchstens l_i Elementen. Abbildung 7.4 zeigt die Darstellung eines Externen Array-Heaps.

Die folgenden Berechnungen dienen dazu, Ihnen eine Vorstellung der Größenordnungen zu geben. Z. B. gilt für $c = 1/7$

$$\mu = \frac{M}{7B} - 1, \quad l_1 = \frac{M}{7}, \quad l_2 = \frac{M^2}{49B} = l_1 \frac{M}{7B} = l_1[\mu + 1]$$

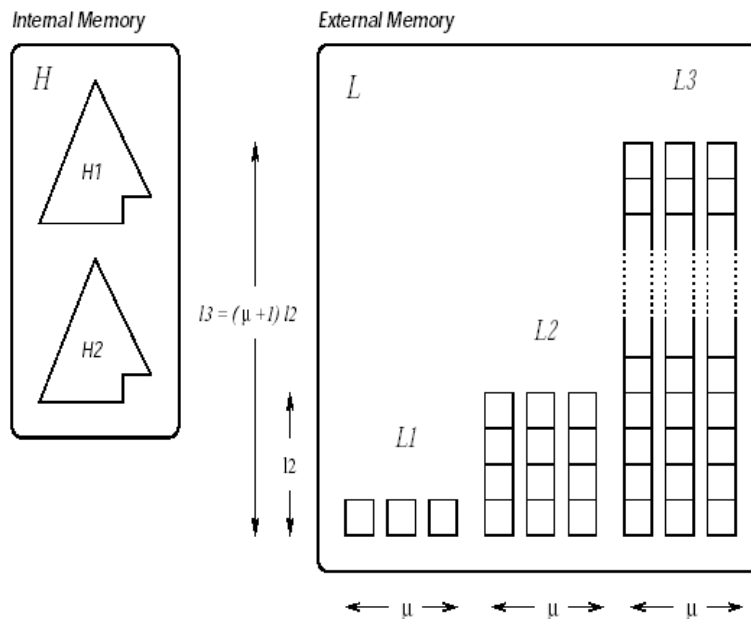


Abbildung 7.4: Externe Array-Heaps

Die letzte Beziehung kann verallgemeinert werden. Diese gilt für alle Schichten L_i .

LEMMA 7.1 Es gilt: $l_{i+1} = l_i(\mu + 1)$

Beweis:

$$l_{i+1} = (cM)^{i+1}/B^i = l_i(cM)/B = l_i(\mu + 1)$$

□

Daraus folgt, dass die Anzahl der Plätze für Elemente in L_{i+1} so gewählt wurde, dass sie genau der Anzahl aller möglichen Plätze von Elementen in L_i entspricht (das sind also $l_i * \mu$) plus zusätzlich l_i .

Die Operation `Insert` fügt die Elemente immer in den Heap H ein. Ist dafür kein Platz mehr in H vorhanden (d.h. H läuft über), dann werden zunächst $l_1 = cM$ dieser Elemente in den Sekundärspeicher bewegt. Zunächst wird versucht, sie in die erste Schicht L_1

7.3. EXTERNESPEICHERDATENSTRUKTUR FÜR PRIORITÄTSWARTESCHLANGEN 25

einzutragen. Dies geschieht in sortierter Folge, und zwar in einen noch freien Slot von L_1 . Falls es in L_1 keinen freien Slot gibt, dann werden **alle** Elemente in L_1 mit den $l_1 = cM$ Elementen aus H zu einer sortierten Liste gemischt, die dann in einen freien Slot von L_2 geschrieben werden. Falls auch dort kein freier Slot existiert, wird dieser Prozess solange wiederholt, bis ein freier Slot gefunden wurde.

Die Operation `Del_Min` erhält die Invariante, dass sich das kleinste Element immer in H befindet. Um dies effizient zu ermöglichen, wird der Heap H in zwei Heaps, H_1 und H_2 aufgeteilt. H_1 enthält maximal $2cM$ Elemente und speichert jeweils die neu eingefügten Elemente ab. H_2 speichert maximal die kleinsten B Elemente aus jedem belegten Slot j in L_i für alle $i = 1, \dots, L$.

Es befinden sich also maximal

$$2cM + B\mu L = 2cM + B\left(\frac{cM}{B}\right)L = cM(2 + L)$$

Elemente im Hauptspeicher. Weiterhin wird $(\mu + 1)B = cM$ zusätzlicher interner Speicher benötigt, um die μ Slots plus die *Overflow*-Folge zu mischen. Da M die Gesamtanzahl der Elemente im Hauptspeicher bezeichnet, muss also gelten

$$M \geq cM(3 + L), \text{ also } L \leq \frac{1 - 3c}{c}.$$

Das heißt, z.B. bei einem Wert von $c = 1/7$ muss $L \leq 4$ sein. L kann also im folgenden als Konstante betrachtet werden.

Die folgenden Operationen sind für Externe Array-Heaps nützlich:

- **Merge-Level** (i, S, S'): produziert eine sortierte Folge S' durch das Mischen der sortierten Folge der μ Slots in L_i (inklusive der jeweils kleinsten Elemente, die sich zu diesem Zeitpunkt in H_2 befinden) und der sortierten Sequenz S . Diese Operation benötigt $O((|S| + l_{i+1})/B + 1) = O(l_{i+1}/B)$ I/O-Operationen.
- **Store** (i, S): nimmt an, dass L_i einen leeren Slot enthält und die Folge S eine Länge im Bereich $[l_i/2, l_i]$ besitzt. S wird in einen leeren Slot von L_i geschrieben und seine kleinsten B Elemente werden nach H_2 bewegt. Diese Operation benötigt $O(|S|/B + 1) = O(l_i/B)$ I/O-Operationen.
- **Load** (i, j): holt die nächsten B kleinsten Elemente vom j -ten Slot aus L_i nach H_2 . Diese Operation benötigt eine I/O.
- **Compact** (i): nimmt an, dass mindestens zwei Slots in Level L_i existieren, deren Gesamtanzahl an Elementen, eingeschlossen diejenigen in H_2 , höchstens l_i ist. Diese beiden Slots (plus ihre kleinsten Elemente, die sich zum aktuellen Zeitpunkt in H_2 befinden) werden gemischt, und in einen freien Slot j' von L_i eingetragen. Damit ist nach dem Aufruf ein zusätzlicher Slot in L_i frei geworden. Die kleinsten B Elemente von j' werden nach H_2 übertragen. Dies benötigt insgesamt $O(l_i/B)$ I/O's.

Die Operationen Insert und Del_Min benutzen diese Operationen folgendermaßen:

Insert: Ein neues Element wird zunächst in H_1 eingefügt. Wenn H_1 voll wird, dann werden die größten $l_1 = cM$ Elemente von H_1 (Folge S) nach L_1 bewegt (die kleinsten B Elemente unter diesen bleiben im Hauptspeicher, sie werden nach H_2 kopiert). Sei $i = 1$. Falls L_i einen leeren Slot enthält, dann wird $\text{Store}(i, S)$ aufgerufen. Sonst enthalten alle Slots von L_i mindestens $l_i/2$ Elemente, sie werden mit S gemischt zur Sequenz S' ($\text{Merge-Level}(i, S, S')$). S' wird nun zur nächsten Schicht L_{i+1} transferiert, usw. Diese Schleife wird höchstens L Mal durchgeführt (bis wir den höchsten Level erreichen).

Del_Min: Das kleinste Element befindet sich entweder im internen Heap H_1 oder im internen Heap H_2 . In beiden Fällen entfernt man das Element. Im ersten Fall ist nichts weiter zu tun. Im zweiten Fall korrespondiert das entfernte Element mit einem Slot j einer Schicht L_i . Falls dieses Element das letzte in H_2 ist, das zu dem assoziierten Slot j der Schicht L_i gehört, dann werden die nächsten B Elemente von Slot j nach H_2 mittels der Operation $\text{Load}(i, j)$ bewegt. Nach jeder $\text{Load}(i, j)$ -Operation werden die Größen der Slots getestet und bei Bedarf wird $\text{Compact}(i)$ aufgerufen. Diese Operation mischt bei Bedarf zwei kleine Sequenzen und bewegt die kleinsten B Elemente der neuen Folge in den internen Heap H_2 .

Bemerkung: Es gilt zu jedem Zeitpunkt, dass in jeder Schicht L_i höchstens ein Slot existiert, der weniger als $l_i/2$ Elemente besitzt. Diese werden wir später beweisen (s. Lemma ...).

7.3.3 Korrektheit und Analyse

Zunächst analysieren wir die Korrektheit der beschriebenen Externen Array-Heap Datenstruktur.

LEMMA 7.2 Das kleinste Element ist immer im internen Speicher H_1 oder H_2 .

Beweis: Das kleinste Element wurde entweder neu eingefügt, und befindet sich deswegen in H_1 oder es gehört zu einem bestimmten Slot in einer Schicht der externen Struktur. Da die Folgen innerhalb der Slots sortiert sind, befindet sich das kleinste Element im ersten Block dieses Slots und wurde deswegen nach H_2 bewegt. Es ist leicht zu sehen, dass diese Invariante unter den verwendeten Operationen bestehen bleibt. \square

LEMMA 7.3 Bei der Ausführung von $\text{Store}(i, S')$ ist immer garantiert, dass S' zwischen $l_i/2$ und l_i Elementen enthält.

Beweis: Die obere Schranke gilt nach Definition der Längen l_i , $i = 1, \dots, L$. Wir zeigen die untere Schranke. Neue Elemente kommen entweder von einer unteren Schicht L_{i-1} durch

7.3. EXTERNSPERICHERDATENSTRUKTUR FÜR PRIORITÄTSWARTESCHLANGEN 27

die Operation $\text{Merge-Level}(i-1, S, S')$ verbunden mit der Operation $\text{Store}(i, S')$ oder durch die reine Operation $\text{Store}(i, S')$. Wir betrachten zunächst den letzteren Fall. Es ist also $i = 1$. Hier werden $cM = l_1$ Elemente ausgelagert, S' enthält also genau l_i Elemente in diesem Fall. Fall 2: S' ist durch $\text{Merge-Level}(i-1, S, S')$ entstanden. Beweis durch Induktion: die Behauptung ist korrekt für alle S bis $i-1$. Wir wollen zeigen: $|S'| \geq l_i/2$. Wir betrachten zunächst die Anzahl der Elemente, die beim Verschmelzen von μ Slots der Schicht L_{i-1} entstehen. Sei a_j die Anzahl der Elemente im Slot j der Schicht $i-1$ vor dem Verschmelzen. Da je zwei dieser μ Slots in Summe mehr als l_{i-1} Elemente enthalten (denn sonst wären sie mittels Compact zusammengefaßt worden), gilt also für jedes Paar j, k : $a_j + a_k > l_{i-1}$. Addiert über alle möglichen Paare ergibt dies:

$$(\mu - 1) \sum_{j=1}^{i-1} a_j > ((\mu - 1)\mu/2)l_{i-1}$$

S' besteht aus den μ Slots der Schicht $i-1$ und dem S , das bereits von den unteren Schichten herrührt. Für dieses S gilt die Induktionsbehauptung. Zusammen sind dies also

$$|S'| = \sum_{j=1}^{i-1} a_j + |S| > (\mu/2)l_{i-1} + l_{i-1}/2 = (\mu + 1)l_{i-1}/2 = l_i/2$$

□

Für die Analysen des folgenden Abschnitts nehmen wir an, dass $cM > 3B$ ist.

LEMMA 7.4 Nach N Operationen existieren höchstens $L \leq \log_{cM/B}(N/B)$ Schichten.

Beweis: Im schlimmsten Fall gibt es keine Element-Entfernungen. Deswegen bewegt jeder Überlauf die maximal mögliche Elementanzahl von einer Schicht in die nächste. Die Anzahl der Elemente in k Schichten ist somit

$$\sum_{i=1}^k \frac{(cM)^i}{B^{i-1}} \mu = \sum_{i=1}^k \left(\frac{(cM)^i}{B^{i-1}} \left(\frac{cM}{B} - 1 \right) \right) < \frac{(cM)^{k+2}}{B^{k+1}}$$

Wir müssen das kleinste k wählen, so dass

$$cM \left(\frac{cM}{B} \right)^{k+1} \geq N$$

ist. Wegen $cM > 3B$ gilt auch

$$cM(cM)^{k+1} \geq B(cM)^{k+1} \geq N.$$

Also folgt

$$k \geq \log_{cM/B}(N/B) - 1.$$

□

Im folgenden analysieren wir die I/O-Operationen für Externe Array-Heaps.

LEMMA 7.5 $\text{Store}(i, S)$ benötigt höchstens $3l_i/B$ I/O-Operationen. $\text{Compact}(i+1)$ und $\text{Merge-Level}(i, S, S')$ benötigen höchstens $3l_{i+1}/B$ I/O-Operationen.

Beweis: Wenn $\text{Store}(i, S)$ ausgeführt wird, wird S zunächst gelesen, und dann wird es in L_i gespeichert. Wegen $l_i/B > 3$ (da $l_i \geq l_1 = cM > 3B$) benötigt $\text{Store}(i, S)$ höchstens $2\lceil l_i/B \rceil \leq 3l_i/B$ I/O-Operationen. $\text{Merge-Level}(i, S, S')$ liest und schreibt jedes Element der Folge S und alle Elemente aller Slots in L_i höchstens ein Mal. Dies ergibt

$$2\lceil (|S| + \mu l_i)/B \rceil \leq 2\lceil l_{i+1}/B \rceil \leq 3l_{i+1}/B$$

I/O-Operationen. Seien j und k die beiden Slots (plus deren Blocks in H_2), die durch die Operation $\text{Compact}(i)$ gemischt werden, und seien e_j bzw. e_k die Anzahl der Elemente in j und k . $\text{Compact}(i)$ benötigt höchstens

$$\lceil e_j/B \rceil + \lceil e_k/B \rceil + \lceil l_i/B \rceil \leq (e_j + e_k)/B + l_i/B + 3 \leq 2l_i/B + 3 \leq 3l_i/B$$

I/O-Operationen, wegen $l_i/B > 3$. □

7.3.4 Amortisierte Analyse zur Abschätzung der I/O-Operationen

Für die Analyse der I/O-Operationen verwenden wir die sogenannte *amortisierte Analyse*. Amortisierte Analyse ist ein allgemeines Verfahren, das dazu dient, die durchschnittlichen Kosten pro Operation für eine beliebige Folge von Operationen nach oben hin abzuschätzen. Das Verfahren der *amortisierten Analyse* wird besonders gern im Zusammenhang mit dynamischen Datenstrukturen verwendet. Eine klassische Worst-Case-Analyse würde für jede Einzeloperation die schlechtestmögliche Schrittzahl abschätzen, die jedoch in einer Folge von Operationen eher selten auftaucht. Diese Abschätzung würde also für eine Folge von Operationen im Allgemeinen eine unrealistisch schlechte Abschätzung ergeben, während die amortisierende Analyse die verschiedenen Fälle gegeneinander aufrechnet.

Die amortisierte Analyse benützt eine Technik, die als Bankkonto-Paradigma bekannt geworden ist. Dabei wird jedem bei der Bearbeitung der Folge auftretenden Zustand ein Kontostand zugeordnet. Eine Einheit auf dem Konto repräsentiert eine Kosteneinheit bei der Abschätzung der Gesamtkosten. Ein Beispiel der Anwendung der Technik der amortisierten Analyse sehen wir im Beweis zum folgenden Satz.

SATZ 7.1 Sei $N \leq B(\frac{cM}{b})^{1/c-3}$, $0 < c < 1/3$ und $cM > 3B$. In einer Folge von N Operationen der Art Insert und Del_Min benötigt Insert amortisiert $\frac{18}{B}(\log_{cM/B}(N/B))$ I/O-Operationen und Del_Min $7/B$ amortisierte I/O-Operationen.

Beweis: Mit jedem nicht-leeren Slot j der Schicht L_i assoziieren wir einen Deposit D_{ij} , der für $6x/B$ Punkte zählt, wobei x die Anzahl der leeren Einträge ist. Anfangs sind alle Slots leer, d. h. die Konstostände sind alle leer. Mit dem internen Heap H assoziieren wir einen Kontostand, der immer dafür sorgen wird, dass mindestens eine Einheit Guthaben

vorhanden ist, sobald die `Load()`-Operation aufgerufen wird. Denn ein solches wird dazu benötigt, die `Load()`-Operation zu bezahlen.

Jedes Element erhält beim Einfügen ein Guthaben von $18L/B$ Kosteneinheiten. Wir zeigen, dass davon je $18/B$ Einheiten genügen, um von einer Schicht zur nächsten Schicht zu wandern (Operation `Merge-Level()` inklusive der `Store()`-Operation). Wir ordnen also jedem Element in jeder Schicht i , $i = 1, \dots, L$, $18/B$ Einheiten zur eventuellen Benutzung zu. Bei Entfernung eines Elements werden $7/B$ Einheiten Guthaben im internen Heap belassen.

Bei dem Fall eines Überlaufs, benutzen wir `Merge-Level(i, S, S')` und danach `Store($i + 1, S'$)`. Nach Lemma 7.5 benötigt jede Operation `Merge-Level(i, S, S')` und jede `Store($i + 1, S'$)`-Operation höchstens je $3l_{i+1}/B$ I/O-Operationen. Dies sind zusammen $6l_{i+1}/B$ I/O-Operationen.

Weil die Größe von S' zwischen $l_{i+1}/2$ und l_{i+1} ist, können diese Kosten durch je $12/B$ Einheiten aus den Guthaben jener Elemente genommen werden, die von dieser Operation betroffen sind. Denn: Wir bewegen mindestens $l_{i+1}/2$ Elemente von einer Schicht L_i zur nächsten Schicht L_{i+1} . (Dabei nehmen wir an, dass L_0 dem internen Heap H_1 entspricht.) Dabei benutzen wir deren Guthaben aus Schicht L_{i+1} . Dies ergibt also zusammen $(12/B)(l_{i+1}/2) = 6l_{i+1}/B$. Nach der `Store()`-Operation ist nun ein Slot j in Schicht L_{i+1} teilweise gefüllt, der vorher leer war. D. h. wir müssen dafür sorgen, dass das Guthaben $D_{i+1,j}$ auf den erforderlichen Wert $6x/B$ aufgefüllt wird, wobei x die Anzahl der leeren Einträge in j bezeichnet.

Wir wissen jedoch, dass x höchstens $l_{i+1}/2$ ist, d.h. wir müssen $D_{i+1,j}$ auf höchstens $6l_{i+1}/(2B) = 3l_{i+1}/B$ Einheiten auffüllen. Diese erhalten wir, indem wir von jedem der $|S'|$ Elemente, die gerade durch `Store($i + 1, S'$)` in den Slot j gewandert sind, $6/B$ Einheiten abziehen. Insgesamt haben wir also $18/B$ Einheiten Guthaben benötigt, um ein Element von L_i nach L_{i+1} zu bewegen; dies sind die amortisierten Kosten, für eine Einfüge-Operation für eine Schicht. Insgesamt über alle Schichten ergeben sich $18L/B$ Einheiten für eine `Insert`-Operation.

Im Falle einer `Del_Min`-Operation werden $7/B$ Einheiten Guthaben im internen Heap belassen. Die Kosten für jede benötigte `Load()`-Operation beträgt eine Einheit, die wir jeweils aus dem Guthaben von H bezahlen. Das geht sich aus, denn: Die `Load(i, j)`-Operation wird jeweils nach B erfolgreichen Entfernungen aus H_2 , die zu dem Slot j assoziiert waren, ausgeführt; von jedem der entfernten Elemente wird jeweils $1/B$ Einheiten für diese `Load()`-Operation zur Verfügung gestellt, dies ergibt zusammen $B(1/B) = 1$.

Die anderen $6/B$ Einheiten Guthaben je entferntem Element werden auf das Konto des Slots j , assoziiert mit der `Load(i, j)`-Operation, d.h. $D_{i,j}$, gutgeschrieben, um die Invariante zu erhalten. Denn in diesem Slot sind nach der `Load()`-Operation B nicht-gefüllte Plätze

hinzugekommen, was zur Folge hat, dass das Guthaben $D_{i,j}$ um $6B/B = 6$ Einheiten erhöht werden muss.

Die $\text{Compact}(i)$ -Operation benötigt $3l_i/B$ Einheiten. Dies kann allein durch die Guthaben der betroffenen Slots $D_{i,j}$ und $D_{i,k}$ bezahlt werden (es seien j und k die beiden betroffenen Slots). Denn die Gesamtanzahl der nicht-belegten Plätze vor der Ausführung der $\text{Compact}(i)$ -Operation in den Slots j und k ist mindestens l_i . Also beträgt das Gesamtguthaben von $D_{i,j}$ und $D_{i,k}$ mindestens $6l_i/B$ Einheiten. Nach dem Mischen der beiden Slots wird ein Slot der beiden, z.B. k leer, und Slot j erhält höchstens $l_i/2$ nicht-belegte Plätze. D.h., wir können $3l_i/B$ des Gesamtguthabens als Bezahlung für die $\text{Compact}(i)$ -Operation benutzen, die anderen $3l_i/B$ Einheiten bleiben als Guthaben bei $D_{i,j}$. Damit ist der Satz bewiesen. \square

Die obere Schranke für N kommt von Platzbeschränkungen her: der Heap benötigt $cM(3 + \log_{cM/B}(N/B))$ internen Speicherplatz (s. nächstes Theorem). Die Schranke folgt aus der Lösung der Ungleichung $cM(3 + \log_{cM/B}(n/B)) \leq M$ für N .

Als nächstes analysieren wir den Speicherplatzbedarf.

LEMMA 7.6 Jede Schicht L_i enthält höchstens einen Slot, der nicht-leer und aus weniger als $L_i/2$ Elementen besitzt.

Beweis: Ein Slot j kann durch die Operation $\text{Load}(i, j)$ Elemente verlieren. Wir nehmen als Induktionsannahme an, dass Slot j in Schicht L_i als einziger dieser Schicht nicht-leer ist und weniger als $l_i/2$ Elemente besitzt. Nach der $\text{Load}(i, j)$ -Operation wird von der DelMin -Operation getestet, ob zwei Slots existieren, deren Gesamtelementanzahl kleiner gleich l_i ist. Falls dies der Fall ist, dann wird die Operation Compact aufgerufen. Wir nehmen an, dass dies für Slot j und Slot k der Fall ist. Dann jedoch wird ein Slot, sagen wir Slot k geleert, und der andere, Slot j wird aufgefüllt. Andernfalls ist Slot j immer noch der einzige nicht-leere Slot in Schicht L_i mit weniger als $L_i/2$ Elemente. \square

SATZ 7.2 Wenn $cM > 3B$, dann gilt: Die Gesamtanzahl der benützten Blöcke ist höchstens $2(X/B) + L$, wobei X die Anzahl der Elemente im Heap H ist. Der Gesamtspeicherplatz im Arbeitsspeicher beträgt $cM(3 + L)$.

Beweis: In jedem Slot jeder Schicht existiert höchstens ein nur teilweise gefüllter Speicherblock, nämlich der oberste. Weiterhin wissen wir, dass maximal ein nicht-leerer Slot pro Schicht L_i existiert, der weniger als $l_i/2$ Elemente besitzt; dieser kann im schlechtesten Fall aus nur einem einzigen Block bestehen, der noch dazu nur teilweise gefüllt ist. Von diesen Fällen existieren zusammen höchstens L . Für die anderen Slots jedoch, die mindestens halb voll sind, gibt es mindestens auch einen voll beschriebenen Block (Aus $L_1 = cM$, $l_i > cM$ für $i \geq 2$, und $cM > 3B$ folgt, dass $l_i/2 > 3/2B$ für $i \geq 1$). Das heißt, dass die Anzahl dieser teilweise beschriebenen Blöcke nicht größer als die Anzahl der ganz belegten

Blöcke ist. Diese ist X/B . Daraus folgt die Beschränkung von $2(X/B)$. Zusammen ergibt dies $2(X/B) + L$. H_1 speichert $2cM$ Elemente und H_2 speichert einen Block pro Slot, das sind $\mu BL = (cM/B - 1)BL \leq cML$. Zusätzlich wird Platz für $(\mu + 1)B$ Elemente zum Mischen der Slots benötigt. Zusammen sind dies $2cM + cML + cM = cM(L + 3)$. \square

Was bedeutet nun die Schranke für N aus Theorem 7.2 in der Praxis?

Wir betrachten den Fall $M = 10^9$ und $B = 10^6$ genauer. In diesem Fall wäre

$$N \leq 10^6 (c10^3)^{1/c-3}.$$

Wenn wir $c = 1/7$ annehmen, dann folgt daraus

$$N \leq 10^6 \left(\frac{10^{12}}{7^4} \right) = 0,416 * 10^{15}$$

und $L \leq 4$ (s. oben).

Selbsttest-Aufgabe: Überlegen Sie sich eine genaue Realisierung für die beschriebene Datenstruktur Array-Heap. Welche Datenstruktur verwenden Sie für den Heap H_1 (H_1 und H_2 müssen keine echten Heap's sein)? Beschreiben Sie die Operation des Überlaufs im Heap beim Einfügen eines Elements, wenn die größten Elemente zu L_1 bewegt werden. Welche Datenstrukturen verwenden Sie für den Heap H_2 ? Welche Datenstruktur verwenden Sie für die Slots und Schichten? Analysieren Sie die Laufzeiten für die Operation Merge-Level (i, S, S') , Store (i, S) , Load (i, j) und Compact (i) im RAM-Modell. Beschreiben Sie eine Realisierung der Insert und der Del_Min Operation und analysieren Sie die Laufzeit.

7.3.5 Experimenteller Laufzeitvergleich

Andreas Crauser hat in seiner Dissertation acht verschiedene Implementierungen für Prioritätswarteschlangen experimentell getestet. Darunter waren vier Realisierungen, die speziell für Externspeicheranwendungen entwickelt wurden, wie Array-Heaps, Externe Radix-Heaps, Buffer Bäume, und B-Bäume. Die vier restlichen Realisierungen waren klassische Algorithmen im RAM-Modell, wie Fibonacci-Heaps, K -ary-Heaps, Pairing-Heaps und interne Radix-Heaps.

Eine Reihe von Experimenten fügt zunächst N Elemente in die Prioritätswarteschlange ein und entfernt diese am Ende wieder. Eine zweite Reihe von Experimenten führte die Insert- und Del-Min-Operationen in gemischter Reihenfolge durch, nachdem zunächst 20 Millionen Elemente eingefügt worden sind. Eine Insert-Operation erfolgte dabei mit Wahrscheinlichkeit $1/3$ und eine Del-Min-Operation mit Wahrscheinlichkeit $2/3$.

Tabelle 7.5 zeigt die experimentellen Resultate. Beim Vergleich zwischen den Externspeicher-algorithmen, schnitten B-Bäume, die wir bereits aus der Vorlesung *Algorithmen und Datenstrukturen 1* kennen, am schlechtesten ab. Es handelte sich dabei um eine Implementierung als B*-Baum, so dass ein Knoten mit allen notwendigen Ineration und Links genau

Insert/Delete _{min} time performance of the external queues (in secs)					
N [$\cdot 10^6$]	radix heap	array heap	buffer tree	buffer tree (orig.)	B-tree
1	6/24	18/11	56/34	62/43	11287/259
5	17/97	74/63	148/309	235/345	66210/1389
10	35/178	353/89	201/882	415/741	-
25	85/372	724/295	311/2833	1096/2302	-
50	164/853	1437/645	445/6085	3462/7592	-
75	246/1416	2157/1005	569/9880	7042/11907	-
100	325/1957	2888/1408	734/19666	12508/16909	-
150	478/3084	4277/2297	*	25051/27181	-
200	628/4036	5653/3234	*	*	-

Random/Total I/Os for external queues				
N [$\cdot 10^6$]	radix heap	array heap	buffer tree	buffer tree (orig.)
1	44/420	24/720	228/668	228/668
5	422/3550	120/4560	16722/21970	13381/15501
10	1124/8620	168/9440	35993/47297	30950/35374
25	2780/21820	570/29520	93789/123285	86495/97879
50	7798/56830	1288/66160	190147/249955	179809/201921
75	12466/89370	2016/102480	286513/376625	275713/310977
100	17736/124740	2776/139760	383518/504134	381023/426615
150	27604/192500	4216/210080	*	595157/659933
200	38284/211570	5712/284320	*	

Insert/Delete _{min} time performance of the internal queues (in secs)				
N [$\cdot 10^6$]	Fibonacci heap	k-ary heap	pairing heap	radix heap
1	3/32	4/33	3/19	3/11
2	6/73	8/75	6/45	5/27
5	17/208	21/210	14/126	11/71
7.5	172800*/-	32/344	22/207	18/124
10	-/-	43/482	30/291	23/162
20	-/-	172800*/-	172800*/-	172800*/-

Abbildung 7.5: Experimentelle Laufzeit für insert-all-delete-all

Time performance on mixed operations			
N [$\cdot 10^6$]	radix heap	array heap	buffer-tree
50	544	770	4996
75	609	945	5862
100	619	1027	6029

Random/Total I/Os on mixed operations			
50	2935/19615	22325/26997	153321/177201
75	5128/26752	24256/28384	171615/196647
100	5782/30094	24220/28380	171658/196578

Abbildung 7.6: Experimentelle Laufzeit und I/O-Operationen für die gemischten Operationen

auf eine Seite passt. Sie benötigen für $\text{Insert} \log_B N$ I/O-Operationen im schlechtesten Fall und besitzen experimentell die längste Rechenzeit (s. Abb. fi:expcruser1). Dies heißt nicht, dass B-Bäume schlechte Externspeicheralgorithmen sind; sie sind einfach nicht so gut als Realisierung der Datenstruktur Prioritätswarteschlange geeignet wie auf Heaps basierende Datenstrukturen.

Externe Radix Heaps führen zu den schnellsten Verfahren. Jedoch ist diese Datenstruktur nur beschränkt anwendbar, nämlich nur dann, wenn die Schlüssel ganzzahlig sind und die `Del_Min` Operationen absteigender Schlüssel beinhalten (wie z.B. bei Dijkstra's kürzestem Wege Algorithmus).

Die Implementierung von Herrn Crauser der Array-Heaps sind bei `Insert` bis zum Faktor 10 langsamer als Radix Heaps und Buffer Bäume, allerdings um einen Faktor ≤ 9 schneller bei der `Del_Min` Operation. Insgesamt sind sie um ca. 3 Mal schneller als Buffer Bäume und ca. 2,5 Mal langsamer als Radix Bäume. Array-Heaps besitzen nur wenig mehr I/O-Operationen als Radix Heaps.

Die vier internen Realisierungen brechen bereits bei $N = 20$ Mio. Operationen ein. Die Ergebnisse für die zweite Testserie sind ähnlich (s. Tabelle 7.6). Die Ergebnisse einer Dijkstra-Simulationsreihe sehen Sie auf den Folien zur Vorlesung.

7.4 Cache-Optimale Algorithmen

Der Cache befindet sich zwischen dem (schnellen) Prozessor und dem (langsamen) Arbeitsspeicher, und enthält jeweils solche Regionen des Arbeitsspeichers, die zuletzt referenziert wurden. Zugriffe auf Speicherbereiche, die sich im Cache befinden (sogenannte *Hits*), können mit Prozessorgeschwindigkeit bearbeitet werden. Zugriffe auf Speicherbereiche, die sich nicht in Cache befinden, (sogenannte *Misses*) ziehen, bevor sie ausgeführt werden können, zunächst eine Operation des Caches nach sich, diese Daten in den Cache zu holen.

Caches funktionieren, weil die meisten Programme eine hohe Lokalität besitzen:

Zeitliche Lokalität existiert, wenn der gleiche Speicherbereich öfters innerhalb einer kurzen Zeitspanne angesprochen wird. Caches nutzen zeitliche Lokalität aus, in dem sie versuchen, angesprochene Speicherbereiche möglichst lange im Cache zu lassen.

Örtliche Lokalität existiert, wenn öfter Speicherbereiche, angesprochen werden, die nahe beieinander liegen. Caches nutzen örtliche Lokalität aus, indem sie bei jedem *Miss* jeweils ganze Speicherblöcke in den Cache holen.

Caches können durch zwei Parameter charakterisiert werden:

- (i) Seitengröße C (in Byte)
- (ii) Kapazität Z (in Byte)

Ein Cache kann maximal Z Bytes aufnehmen. Diese sind unterteilt in Seiten der Größe L . Es gibt also auch hier — wie bereits im Externspeichermodell — eine Speicherhierarchie auf 2 Ebenen, bei dem die untere Ebene Kapazität Z besitzt, und die Daten zwischen den beiden Ebenen in Blöcken der Größe L hin- und herschoben werden.

Als Folgerung daraus, kann man auch hier das EM-Modell von Aggarwal und Vitter (1988) zugrunde legen. Das heißt, dass auch alle Externspeicheralgorithmen, die für das EM-Modell entwickelt wurden, grundsätzlich auch zur Cache-Optimierung verwendet werden können.

Allerdings enthalten moderne Computer tatsächlich nicht nur zwei Speicherschichten, sondern mehrere. Typische Komponenten sind: Register, Level 1 Cache, Level 2 Cache, Arbeitsspeicher und Externspeicher. Diesem Aspekt tragen Frigo, Leiserson, Prokop und Ramachandran (1995) Rechnung, indem sie das EM-Modell zum *Cache-Oblivious* Speichermodell (CO-Modell) erweitert haben.

Das *Cache-Oblivious* Speichermodell sieht vor, dass die Seitengröße L und die Kapazität Z unbekannt sind. Eine Algorithmen-Analyse im CO-Modell muss also für alle Blockgrößen und Kapazitäten gelten, folglich also auch für alle Ebenen der Speicherhierarchie.

Indem man also einen Algorithmus für unbekanntes C und Z optimiert, optimiert man automatisch die Speicherzugriffe auf jeder Ebene. Dies erhöht außerdem die Portabilität von Programmen, die im EM-Modell speziell für ein festes L und Z bzw. B und M entwickelt wurden.

7.4.1 Das ideale Cache-Modell von Frigo et al.

Das *ideale Cache-Modell* von Frigo et al. ist in Abbildung 7.7 dargestellt. Das Modell sieht eine 2-Ebenen Speicherhierarchie vor, die aus einem idealen Cache mit Z Bytes besteht und einem unendlich großen Arbeitsspeicher. Der Cache ist in Cache-Zeilen aufgeteilt, die jeweils aus L Bytes bestehen, und die jeweils in einem Schritt zwischen Cache und Arbeitsspeicher bewegt werden. Dabei wird angenommen, dass der Cache hoch ist, d. h. es gilt $Z = \Omega(L^2)$. Der Prozessor kann jeweils nur Speicherplätze ansprechen, die zu einer Linie im Cache gespeichert werden. Der ideale Cache benützt eine optimale off-line Strategie, indem er jeweils diejenigen Speicherzeilen im Cache ersetzt, deren Zugriff am weitesten in der Zukunft liegt.

Die Analyse erfolgt durch die beiden Parameter

- (i) Anzahl der ausgeführten CPU-Operationen im RAM-Modell
- (ii) Cache-Komplexität $Q(n, Z, L)$: die Anzahl der Cache-Misses, abhängig von Z und L .

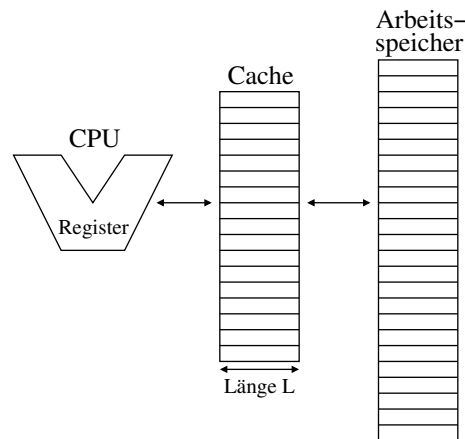


Abbildung 7.7: Das ideale Cache-Modell von Frigo et al.

Ein Algorithmus heißt „*Cache-Aware*“ (*Cache-bewusst*) wenn er Parameter enthält, mit denen die Cache-Komplexität für die gegebene Cache-Größe Z und Zeilenlänge L optimiert werden kann. Andernfalls heißt der Algorithmus „*Cache-Oblivious*“ (*Cache-ignorierend*).

Man kann zeigen, dass ein *Cache-Oblivious* Algorithmus, der für das 2-Ebenen Hierarchiespeichermodell optimiert wurde, auch für mehrere Ebenen optimal ist.

Im folgenden werden wir einen *Cache-Aware* Algorithmus sowie einen *Cache-Oblivious* Algorithmus für das Problem der Matrizen-Multiplikation kennenlernen, die beide bezüglich der Anzahl der I/O-Operationen optimal sind.

7.4.2 Ein Cache-Aware Algorithmus zur Matrix-Multiplikation

Wir betrachten die Multiplikation zweier $n \times n$ Matrizen A und B . Wir nehmen an, dass die Matrizen jeweils zeilenweise gespeichert sind, und dass n sehr groß ist, d.h. $n > L$. Der folgende Algorithmus ist *Cache-Aware*:

Dabei ist $MULT(A, B, C, s)$ die Standard-Prozedur, die $C = C + AB$ auf $s \times s$ Untermatrizen in Zeit $O(s^3)$ berechnet. (Dabei wird angenommen, dass s Teiler von n ist, ansonsten wäre der Code etwas aufwändiger).

Abhängig von der Cachegröße des Rechners kann der Parameter s so bestimmt werden, so dass die Cache-Komplexität optimiert werden kann. Es handelt sich also dabei um einen Algorithmus, der „*Cache-Aware*“ ist.

Algorithmus 1 BLOCK-MULT (A, B, C, n)

```

1: für  $i = 1$  bis  $n/s$  {
2:   für  $j = 1$  bis  $n/s$  {
3:     für  $k = 1$  bis  $n/s$  {
4:       MULT( $A_{ik}, B_{kj}, C_{ij}, s$ )
5:     }
6:   }
7: }
```

Um die Cache-Komplexität zu minimieren, wählen wir s als den größten Wert, so dass die drei $s \times s$ Untermatrizen zusammen in den Cache passen. Eine $s \times s$ Untermatrix benötigt $\Theta(s + s^2/L)$ Cache-Zeilen. Das liegt daran, dass die Matrix nicht unbedingt mit den Grenzen der Blöcke oder Zeilen abschließt, die gleichzeitig aus dem Hauptspeicher in den Cache geladen werden. Im schlimmsten Fall ist der Platzbedarf der Matrix im Cache deshalb

$$\Theta(s(\lceil s/L \rceil)) = \Theta(s(1 + s/L)) = \Theta(s + s^2/L)$$

Aus unserer Annahme, dass $Z = \Omega(L^2)$ folgt $s = \Theta(\sqrt{Z})$.

Folglich benötigt jeder Aufruf von $MULT()$ höchstens $Z/L = \Theta(s^2/L)$ Cache-Misses, um die drei Untermatrizen von A, B und C in den Cache zu bringen. Daraus folgt insgesamt eine Cache-Komplexität von

$$\Theta(1 + n^2/L + (n/\sqrt{Z})^3(Z/L)) = \Theta(1 + n^2/L + n^3/L\sqrt{Z}),$$

denn der Algorithmus muss n^2 Elemente einlesen, die auf $\lceil n^2/L \rceil$ Cache-Zeilen verteilt sind.

7.4.3 Ein Cache-Oblivious Algorithmus zu Matrixmultiplikationen

Wir wollen eine $m \times n$ Matrix A mit einer $n \times p$ Matrix B *cache-oblivious* multiplizieren. Der Algorithmus REC-MULT arbeitet rekursiv und folgt dem Divide&Conquer Prinzip:

Fall 1: Falls $m \geq \max\{n, p\}$, dann wird die Matrix A horizontal in Matrizen A_1 und A_2 mit jeweils $\lceil m/2 \rceil$ und $\lfloor m/2 \rfloor$ Zeilen gesplittet. Es folgen zwei Aufrufe der Form A_1B und A_2B . Hierbei wird das Gesetz

$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1B \\ A_2B \end{pmatrix}$$

ausgenutzt.

Fall 2: Andernfalls, falls $n \geq \max\{m, p\}$ gilt, dann werden beide Matrizen aufgeteilt, nämlich A vertikal in A_1 und A_2 mit jeweils $\lceil n/2 \rceil$ und $\lfloor n/2 \rfloor$ Spalten und B horizontal in B_1 und B_2 mit jeweils $\lceil n/2 \rceil$ und $\lfloor n/2 \rfloor$ Zeilen. Hierbei wird das Gesetz

$$\begin{pmatrix} A_1, A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2$$

genutzt.

Fall 3: Andernfalls, falls gilt $p \geq \max\{m, n\}$. In diesem Fall wird B vertikal aufgesplittet in B_1 und B_2 mit jeweils $\lceil p/2 \rceil$ und $\lfloor p/2 \rfloor$ Spalten. Es gilt

$$A(B_1, B_2) = (AB_1, AB_2).$$

Fall 4: Falls $m = n = p = 1$ gilt, dann werden die beiden Elemente multipliziert und zur resultierenden Matrix C hinzuaddiert.

Analyse des Algorithmus REC-MULT

SATZ 7.3 Der Algorithmus REC-MULT zur Multiplikation zweier Matrizen der Größe $m \times n$ und $n \times p$ benötigt $\Theta(mnp)$ Zeit im RAM-Modell. Die Cache-Komplexität beträgt $\Theta(m + n + p + (mn + np + mp)/L + mnp/L\sqrt{Z})$ Cache-Misses.

Beweis: [ohne Beweis]; für interessierte Leser, siehe Frigo, Leiserson, Prokop und Ramachandran. \square

KOROLLAR 7.1 Zur Multiplikation zweier $n \times n$ Matrizen benötigt der Algorithmus REC-MULT $\Theta(n^3)$ Zeit im RAM-Modell und $\Theta(n + n^3/L\sqrt{Z})$ Cache-Komplexität.

Intuitiv benutzt der Algorithmus den Cache effektiv, denn sobald ein Unterproblem ganz in den Cache passt, können dessen Unterprobleme ohne einen einzigen Cache-Miss gelöst werden. Deswegen sind Divide&Conquer Algorithmen grundsätzlich sehr gut für große Datenmengen geeignet.

Ein Verfahren mit besserer Laufzeit ist das Verfahren von Strassen (1968) zur Matrixmultiplikation. Auch dieses Verfahren beruht auf dem Divide&Conquer Prinzip. Jedoch wird hierbei jede Untermatrix in 4 Viertel der Höhe und Länge jeweils ungefähr $n/2$ zerlegt.

Die Komplexität von Strassen's Algorithmus zur Matrixmultiplikation zweier $n \times n$ Matrizen ist

$$\Theta(N^{\log_7 7}) = \Theta(N^{2,81}) \text{ mit } \Theta(n + n^2/L + n^{\log_7 7}/L\sqrt{Z}).$$

Auch dieser Algorithmus ist Cache-Oblivious und kommt theoretisch mit weniger Cache-Misses aus. Allerdings ist der Algorithmus von Strassen sehr aufwändig zu implementieren. In der Praxis ist der Algorithmus von Strassen für $n \leq 1.000.000$ noch ungefähr genauso langsam wie die besprochenen Algorithmen zu Matrixmultiplikation. Generell wird der Algorithmus von Strassen als ein eher theoretischer Beitrag gesehen, der damals allerdings

sehr überraschend war.

Inzwischen wurden viele Varianten des Algorithmus von Strassen gefunden. Allerdings gilt das Matrixmultiplikationsproblem als eines der interessantesten offenen Probleme, denn man geht davon aus, dass “der beste” Algorithmus für dieses Problem noch nicht gefunden wurde.

7.5 Weiterführende Literatur

Abschnitt 3.2 hält sich eng an *Kapitel 4: Priority Queues* aus der Dissertation von Andreas Crauser, *LEDA-SM: External Memory Algorithms and Data Structures in Theory and Practice*, Universität des Saarlandes, 2001

<http://www.mpi-sb.mpg.de/~crauser/diss.pdf>.

Eine Kurzfassung gibt es auch in: Klaus Brengel, Andreas Crauser, Paolo Ferragina, Ulrich Meyer: *An Experimental Study of Priority Queues in External Memory*, Proc. of the Workshop on Algorithmic Engineering (WAE '99), Lecture Notes in Computer Science 1668, 345-359, Springer-Verlag, 1999.

Abschnitt 3.3 hält sich an: Frigo, Leiserson, Prokop, Ramachandran, *Cache-Oblivious Algorithms*, MIT, Cambridge IEEE Transactions, 1999.

Originalliteratur zum Thema finden Sie unter: A. Aggarwal und J.S. Vitter: *The input/output complexity of sorting and related problems*, Communications of the ACM 31 (9), 1988, 1116-1127

oder auch: J.S. Vitter und E.A.M. Shriver: *Optimal algorithms for parallel memory I: two-level memories*, Algorithmica, 12(2-3), 1994