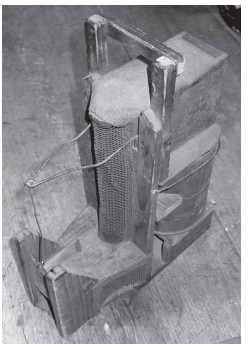
Quicksort mit zwei Pivots und mehr: Eine mathematische Analyse von Mehrwege-Partitionierungsverfahren und der Frage, wie Quicksort dadurch schneller wird[1]

Sebastian Wild[2]

It is tempting to try to develop ways to improve quicksort: a faster sorting algorithm is computer science's "better mousetrap," and quicksort is a venerable method that seems to invite tinkering.

R. Sedgewick and K. Wayne, Algorithms 4th ed.



Bildquelle: Massel tow (https://commons.wikimedia.org/wiki/File:Historical_Mousetrap.jpg), https://creativecommons.org/licenses/by-sa/2.0/de/legalcode

Historische Mausefalle
(Museum der Schwalm, Schwalmstadt)

Abstract: Seit 2011 kommt in der Java runtime library eine Quicksort-Variante mit zwei Pivots zum Einsatz, von der lange unklar war, warum sie gegenüber der vorherigen, ausoptimierten Quicksort-Implementierung nochmals über 10% Laufzeit einsparen kann. Das verwundert umso mehr, da frühere Untersuchungen keinen Vorteil in der Verwendung mehrerer Pivots erwarten ließen. Durch eine umfassende mathematische Analyse aller sinnvollen Quicksort-Varianten konnte ich beweisen, dass durch die Verwendung mehrerer Pivots tatsächlich keine wesentliche Ersparnis bei der Anzahl an Vergleichen, und allgemeiner dem Aufwand innerhalb der CPU, zu erwarten ist, wohl aber eine deutliche Reduktion des Datenvolumens, das zwischen Hauptspeicher und CPU transferiert werden muss. Diese effizientere Nutzung der Speicherhierarchie ist dabei *nur* durch den Einsatz mehrerer Pivots, und durch keine andere der vielen bekannten Optimierungen von Quicksort zu erreichen.

1 Einführung

Im Laufe der vergangenen zehn Jahre hat sich ein technologischer Wandel mit weitreichenden Konsequenzen vollzogen, der eine Vielzahl neuer Informatik-Anwendungen auf neuen Gerätetypen hervorgebracht hat. In solch turbulenten Zeiten sind gründlich analysierte Algorithmen als Inbegriff plattformunabhängiger Programme und dauerhaft gültiger Konzepte wertvolle Pfeiler der Stabilität. Doch gerade bei dem vielleicht am meisten untersuchten und am besten verstandenen aller algorithmischen Probleme, dem Sortieren eines Arrays von Elementen, hat sich eine verstecke Revolution abgespielt: *Alle* Sortiermethoden in Oracles Java runtime library wurden in den vergangenen zehn Jahren von Grund auf neu geschrieben [Ja09, JD09]. Da die neuen Methoden auch in die Android runtime library übernommen wurden, gehören sie wohl zu den am häufigsten ausgeführten Algorithmen überhaupt.[3] So weit verbreitete Bibliotheken werden sehr konservativ gemanagt, ist doch

[1] Englischer Titel der Dissertation: "Dual-Pivot Quicksort and Beyond: Analysis of Multi-Way Partitioning and Its Practical Potential"

[2] TU Kaiserslautern, Fachbereich Informatik, Postfach 3049, 67663 Kaiserslautern, wild@cs.uni-kl.de

[3] Aktuelle Schätzungen sprechen von weltweit über 2 Mrd. aktiven Android-Geräten [Ah16]!

jede Änderung mit der Gefahr verbunden, existierende Anwendungen zu beeinflussen: So basierten (bis vor knapp zehn Jahren) die Quicksort-Implementierungen aller großen Bibliotheken auf dem von Jon Bentley und Douglas McIlroy Anfang der 1990er für die C-Standardbibliothek entwickelten Code [BM93]. Trotzdem war die Laufzeitersparnis der neuen Methoden – Dual-Pivot Quicksort und Timsort – schließlich so groß, dass ihre Überlegenheit nicht ignoriert werden konnte.

Bemerkenswerterweise stammt die Idee von Java's Dual-Pivot Quicksort nicht etwa von einem Algorithmik-Experten; im Gegenteil: Der junge russische Softwareentwickler Vladimir Yaroslavskiy, damals Angestellter bei Sun Microsystems, tüftelte in seiner Freizeit an schnellen Sortiermethoden und entdeckte dabei eher zufällig das Potential von Mehrwege-Partitionierungsverfahren. Mit Hilfe der erfahrenen Algorithmiker und Entwickler Jon Bentley und Joshua Bloch entwickelte Yaroslavskiy schließlich die jetzige Bibliotheksmethode. Diese enthält eine ganze Reihe von geschickten Mechanismen, um für möglichst viele Eingaben effizient zu sein; auf uniform zufälligen Permutationen ist der Code aber im Wesentlichen äquivalent zu nebenstehendem Algorithmus.

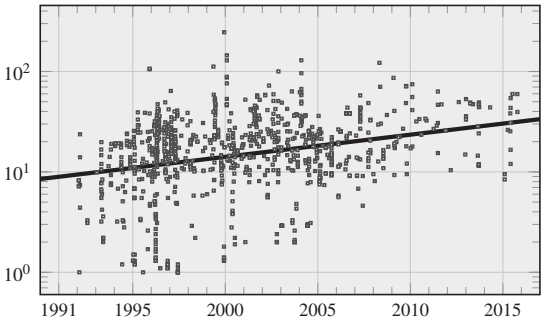
```
DUALPIVOTQUICKSORT(A, left, right)
    // Sortiere A[left..right]
 1  if right − left ≥ 1
        // Wähle äußerste Elemente als Pivots
 2      P := min {A[left], A[right]}
 3      Q := max{A[left], A[right]}
 4      ℓ := left + 1;  g := right − 1;  k := ℓ
 5      while k ≤ g
 6          if A[k] < P
 7              Swap A[k] and A[ℓ];  ℓ := ℓ + 1
 8          else if A[k] ≥ Q
 9              while A[g] > Q and k < g
10                  g := g − 1
11              end while
12              Swap A[k] and A[g];  g := g − 1
13              if A[k] < P
14                  Swap A[k] and A[ℓ];  ℓ := ℓ + 1
15              end if
16          end if
17          k := k + 1
18      end while
19      ℓ := ℓ − 1;  g := g + 1
        // Bringe Pivots an ihren Platz
20      A[left] := A[ℓ];  A[ℓ] := P
21      A[right] := A[g];  A[g] := Q
22      DUALPIVOTQUICKSORT(A, left , ℓ − 1)
23      DUALPIVOTQUICKSORT(A, ℓ + 1, g − 1)
24      DUALPIVOTQUICKSORT(A, g + 1, right)
25  end if
```

Wie konnte dieser so deutlich überlegene Dual-Pivot Quicksort den vielen Forschern weltweit über Jahrzehnte verborgen bleiben? Und wenn zwei Pivots so erfolgreich sind, sind dann noch mehr Pivots vielleicht noch besser?

Diese Fragen waren der Ausgangspunkt meiner Forschung. Der Schlüssel zu ihrer Beantwortung liegt in der zugrundeliegende Frage, *warum* Dual-Pivot Quicksort schneller als der bisherige, klassische Quicksort ist. In meiner Dissertation [Wi16] konnte ich eine plausible Antwort darauf geben: Die klassischen Modellannahmen für die Laufzeitvorhersage sind auf modernen Computern nicht mehr in dem Maße erfüllt, wie es vor 20 Jahren noch der Fall war. Dual-Pivot Quicksort wurde in der Vergangenheit durchaus schon untersucht [Se80, He91] – aber im damals angemessenen Modell, die Anzahl Schlüsselvergleiche und Vertauschungen zu zählen, korrekterweise für nicht hilfreich befunden! Die Idee der Mehrwege-Partitionierung ist also nicht neu; neu ist aber, dass Quicksort dadurch schneller wird. Der Grund dafür ist ein langanhaltender Trend im Hardwaredesign, der mittlerweile wohl auch beim Sortieren im Hauptspeicher ein Umdenken erforderlich macht.

2 Die „Speicher-Wand“

Die Herstellungsprozesse für Prozessoren haben sich seit den Anfängen der integrierten Schaltkreise Mitte des 20. Jahrhunderts enorm weiterentwickelt; nach der bekannten Vorhersage des *Mooreschen Gesetzes* etwa *verdoppelt* sich die Anzahl Transistoren pro Fläche etwa alle zwei Jahre. Auch wenn Auslegung und Gültigkeit dieser Aussage immer wieder Anlass zu Diskussionen geben, hat sich die tatsächlich messbare Durchsatz-Geschwindigkeit der CPUs durchaus in dieser Größenordnung gesteigert und tut es noch – die Geschwindigkeit der Speicher und ihrer Bussysteme konnte dagegen nicht mit dieser Entwicklung Schritt halten: Nach den Daten des STREAM-Benchmarks [Mc07] wuchs die CPU-Geschwindigkeit über die letzten 25 Jahre mit einer jährliche Rate von etwa 46%, während die *Speicherbandbreite,* die messbare Datenmenge, die pro Zeiteinheit zwischen CPU und Hauptspeicher übertragen werden kann, eine Wachstumsrate von etwa 37% jährlich aufwies.[4] Abb. 1 zeigt die Auswirkungen dieser unterschiedlich schnellen Weiterentwicklung: eine exponentiell wachsende Imbalance unserer Computer.



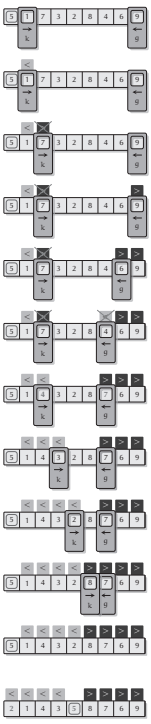
Jeder Punkt zeigt ein Ergebnis des STREAM-Benchmarks: die x-Achse das Datum, die logarithmische y-Achse die gemessene Balance, die sich aus *peak performance* (in MFLOPS) dividiert durch Netto-Bandbreite zum Hauptspeicher (in MW/s im “triad” Benchmark) zusammensetzt. Die Linie ist die Regressionsgerade der gezeigten Punktewolke.
(Für Details zu Daten und Aufbereitung, siehe Figure 1 in [Wi16].)

Abb. 1: Entwicklung der *Maschinenbalance,* dem Quotienten von CPU-Geschwindigkeit und der Speicherbandbreite über die letzten 25 Jahre.

Konsequenzen der Imbalance. Speicher- und Rechenkapazität im Überfluss sind nutzlos, wenn die Berechnungen ständig pausiert werden müssen, um die nächsten Daten aus dem Hauptspeicher zur CPU zu übertragen. Hierbei hilft auch die moderne Speicherhierarchie aus mehreren Levels schneller Cache-Speicher und automatischem *Prefetching* von zukünftig angefragten Speicherbereichen nicht weiter, wenn die Übertragungsgeschwindigkeit selbst der limitierende Faktor ist. John Backus erkannte dieses Problem schon 1977 und nannte es *von-Neumann-Bottleneck* [Ba78]; weit dramatischer formulierten es William Wulf und Sally McKee 1995 [WM95]: Wenn sich die in Abb. 1 gezeigte Entwicklung nur lange genug fortsetzt, laufen wir gegen eine *“memory wall”*, einen Zustand in dem die Geschwindigkeit des ganzen Systems vollständig von der Geschwindigkeit des Speichers dominiert wird. Diese drastische Situation ist beim Sortieren sicher (noch?) nicht gegeben, aber die Gewichte haben sich verschoben: entsprach das Verhältnis von CPU-Geschwindigkeit zu Bandbreite 1993, als die klassische Quicksort-Implementierungen entwickelt wurde, noch etwa 7:1, so liegt es heutzutage bei 30:1.

[4] Diese Werte sollten nicht direkt für quantitative Vorhersagen verwendet werden, da sie Durchschnittswerte über verschiedenste Geräteklassen darstellen und eine gleichmäßige Datenverteilung über die Zeit nicht gewährleistet werden kann. Der qualitative Trend ist aber bekannt und stabil gegen Variationen der Experimente.

3 Quicksort und die Speicherbandbreite



Um zu verstehen, welchen Einfluss diese relative Verteuerung von Speicherzugriffen auf das Sortieren hat, ist ein detaillierter Blick auf die Sortierverfahren nötig. Die Kernidee von Quicksort ist recht einfach: Wir bestimmen den *Rang* eines (beliebigen) Elementes, des *Pivots*, indem wir jedes andere Element mit dem Pivot vergleichen; der Rang des Pivots ist die Anzahl Elemente, die kleiner als das Pivot sind, und damit die *Position* des Pivots im sortierten Array. Indem man kleine Elemente direkt nach links, und große Elemente nach recht bringt, kann man das Array in einem Durchlauf *partitionieren*, also in drei Bereiche aufteilen, die alle kleinen, das Pivot bzw. alle großen Elemente beinhalten. Erste und letztere sortiert man anschließend unabhängig voneinander rekursiv mit dem gleichen Verfahren, solange das Teilarray mehr als ein Element umfasst.

Klassischer Quicksort. Das gängige Partitionierungsverfahren ist rechts illustriert. Es geht auf C. A. R. Hoare [Ho62] und Robert Sedgewick [Se78] zurück und kam bis Version 6 in der Java runtime library (und vielen weiteren Bibliotheken) zum Einsatz. Die Methode arbeitet in-place und basiert auf sequentiellen Scans; das ist optimal bzgl. der Anzahl *Cache Misses*. Tatsächlich arbeiten alle in der Praxis relevanten Partitionierungsmethoden nach diesem Prinzip. Obwohl a priori nicht feststeht, wo die beiden Laufindizes k und g sich treffen, scannen sie gemeinsam stets das gesamte Array genau einmal. (Einfachere Verfahren, z. B. Lomutos Methode [Be84, Co09], haben diese Eigenschaft nicht!)

Um den Bandbreitenbedarf mathematisch analysieren und mit anderen Algorithmen vergleichen zu können, müssen wir ein *Kostenmaß,* ein mathematisch präzise definiertes Modell der Kosten einer Ausführung, fixieren.

Iteratoren: Ein Modell für Scan-basierte Algorithmen. In unserem Modell geben wir die sequentielle Arbeitsweise fest vor; dadurch lässt sich die Kostenanalyse vereinfachen. Wir erlauben Zugriffe auf das Array nur durch *Iteratoren*, konzeptionelle Schreib-Lese-Köpfe, die sich stets an einer bestimmten Stelle im Array befinden und den Wert an dieser Stelle lesen und schreiben können. Iteratoren können darüber hinaus nur zu einer benachbarten Stelle weiterbewegt werden.[5] Die obige Illustration zeigt die Iteratoren als Rechtecke mit Fenster. Als Kosten der Ausführung definieren wir die Summe der zurückgelegten Distanzen aller Iteratoren; wir nennen dies kurz die Anzahl *gescannter Elemente*. Man beachte, dass ein Element als mehrfach gescannt gezählt wird, wenn es von mehreren Iteratoren besucht wird.

Die Anzahl gescannter Elemente ist i. W. proportional zur Anzahl der *(Level 1) Cache Misses* [NWM16, ADK16], da nach B Bewegungen eines Iterators ein neuer Cacheblock adressiert wird, wenn B die Blockgröße des Caches ist. Unser Modell ähnelt daher dem

[5] Zusätzlich benötigt man eine Möglichkeit, Iteratoren zu klonen, also ein Kopie zu behalten, die auf der aktuellen Stelle verbleibt. Die Gesamtzahl aktiver Iteratoren soll aber beschränkt bleiben. Des Weiteren kann man sich auf unidirektionale Iteratoren beschränken; beim Klonen gibt man dann die Scanrichtung des Klons mit an.

klassischen Externspeichermodell [AV88] mit Blockgröße $B = 1$.[6] Trotzdem halte ich es für angebracht, hier nicht die Terminologie des Externspeichermodells zu übernehmen, da das suggerierte, es käme auch beim Sortieren im Hauptspeicher *nur* noch auf die Datenbewegungen an. Tatsächlich spielt aber der Rechenaufwand auch weiterhin eine Rolle.

4 Dual-Pivot Quicksort gegen Klassischen Quicksort

Wie oben bereits erwähnt, kostet die klassische Hoare-Sedgewick-Partitionierung n gescannte Elemente, da die Iteratoren k und g gemeinsam jedes Elemente genau einmal besuchen. Das ist in diesem Fall gleichzeitig die Anzahl verwendeter Schlüsselvergleiche, sodass sich für den klassischen Quicksort die bekannten Resultate für Vergleiche (z. B. [Se80]) direkt übernehmen lassen. Die Anzahl gescannter Elemente für die Yaroslavskiy-Bentley-Bloch-Partitionierung (kurz YBB) ist im Mittel $\frac{4}{3}n$, da wieder zwei Iteratoren k und g gemeinsam alle Elemente ablaufen, aber zusätzlich ℓ alle kleinen überstreicht; siehe Abb. 2. Dagegen ist die Anzahl Vergleiche trickreicher zu bestimmen, als auf den ersten Blick erscheinen mag; im Mittel sind es $\frac{19}{12}n$ [WN12].

Abb. 2: Zustand des Arrays nach YBB-Partitionierung mit Angabe der gelaufenen Distanzen der Iteratoren.

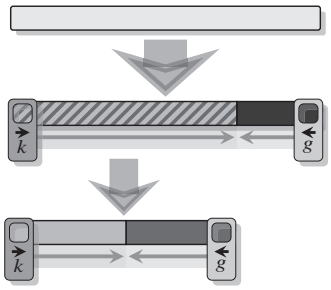


Abb. 3: Simulation einer 3-Wege- durch zwei 2-Wege-Partitionierungen.

Nun kann man die von der YBB-Partitionierung hinterlassene Dreiteilung des Arrays in kleine, mittlere und große Elemente auch erreichen, indem man *zweimal* die klassische anwendet: zuerst das ganze Array bzgl. Q, des größeren Pivots, wodurch die großen Elemente abgespalten werden, und dann das linke Segment nochmals bzgl. P, um auch kleine von mittleren Elemente zu trennen. Der Vorgang ist in Abb. 3 illustriert. Im Vergleich zu YBB wird dabei das mittlere Segment *zweimal* statt einmal durchlaufen, sodass im Mittel $\frac{5}{3}n$ statt der $\frac{4}{3}n$ gescannte Elemente anfallen. Um die gleiche Arbeit zu verrichten, benötigt YBB-Partitionierung also weniger Speicherbandbreite als die klassische Variante! Der Aufwand innerhalb der CPU steigt dagegen durch den komplexeren Partitionierungscode tatsächlich eher an.

Diese Simulationssichtweise ist sehr hilfreich um verschiedene Quicksort-Varianten miteinander zu vergleichen, sie bedarf aber etwas Sorgfalt. Während im üblichen Eingabemodell – zufällige Permutationen von n paarweise verschiedenen Schlüsseln – jeder Rang des Pivots im klassischen Quicksort gleich wahrscheinlich ist, gilt das für simuliertes Mehrwege-Partitionieren nicht mehr: Q ist stets das Maximum zweier Elemente, die Verteilung des Rangs ist also nach rechts verschoben! Der Rang von P ist dagegen innerhalb seines Teilbereiches uniform verteilt. Effektiv haben wir das Pivot Q als Ordnungsstatistik aus einem zufälligen Sample gewählt (max von 2), P dagegen als ein zufälliges Element.

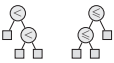
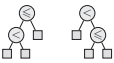
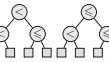
[6] Die interne Speichergröße M entspricht der erlaubten Anzahl aktiver Iteratoren, und es sind keine parallelen Transfers möglich, also $P = 1$.

5 Die Großfamilie der Quicksort-Varianten

Die Wahl der Pivots ist eine der vielen Stellschrauben von Quicksort, und in der Tat eine sehr effektive. Es ist *best practice* für Quicksort mit einem Pivot den Median von drei Elementen, oder sogar eine Approximation des Medians von neun Elementen ("ninther") zu wählen, da dies in Erwartung gut ein Viertel der Vergleiche – und damit gleichzeitig der gescannten Elemente, wie wir aus obiger Diskussion wissen! – einspart [NWM16]. Eine weitere erfolgreiche Optimierung ist, ab einer gewissen Teilproblemgröße die Rekursion abzubrechen und zu Insertionsort zu wechseln, weil dieser Algorithmus auf winzigen Eingaben schneller ist. Schließlich kann man die Partitionierungsmethode selbst in vielen Details variieren. Tatsächlich war es in Quicksorts Jugendzeit geradezu Volkssport, gewiefte neue Quicksort-Varianten zu ersinnen; auf diesen Umstand spielt das Zitat zu Beginn dieses Artikels an, das die Parallele zu Mausefallen zieht, der Erfindung, zu der es in den USA mit Abstand die meisten Patente gibt (weit über 4000 [Ja11]!). Erst durch seine mathematische Analyse konnte Robert Sedgewick 1975 etliche im Umlauf befindliche vermeintliche Verbesserungen als ineffektiv oder gar kontraproduktiv entlarven [Se80].

Vierzig Jahre später herrscht neuerliche Euphorie: Nach dem Erfolg von YBB-Quicksort wurden neue Mehrwege-Quicksort-Varianten vorgeschlagen [AD15, Ku14], und unzählige weitere sind denkbar, da sich die altbekannten Optimierungen, allen voran das *Pivot Sampling*, auch auf Mehrwege-Quicksort anwenden lassen. Meine Dissertation versucht im Geiste von Sedgewicks Arbeit eine Neubewertung vorzunehmen, die den Effekt auf den Bandbreitenverbrauch mit einbezieht.

Generisches one-pass in-place s-Wege-Partitionieren. Um der Vielzahl existierender und denkbarer Quicksort-Varianten Herr zu werden, analysiere ich einen generischen Partitionierungsalgorithmus, der eine ganze Reihe an Stellschrauben als Parameter offen lässt: s, die Anzahl *Segmente* (oder äquivalent: $s-1$ Pivots), m, die Anzahl Iteratoren, die von links nach rechts laufen (die restlichen $s-m$ laufen entgegengesetzt), sowie λ_k und λ_g, die beiden *Vergleichsbäume*. Vergleichsbäume sind binäre Suchbäume mit den Pivots als Schlüsseln, die bestimmen, in welcher Reihenfolge ein Element mit den $s-1$ Pivots verglichen wird, um seine *Klasse*, den Index seines Zielsegments, zu bestimmen. Ich erlaube dabei zwei verschiedene Vergleichsbäume zu verwenden, je nachdem, ob ein Element zuerst vom Links-nach-rechts-Iterator k oder vom Rechts-nach-links-Iterator g erreicht wird. Tatsächlich macht die YBB-Partitionierung davon Gebrauch; dort vergleicht λ_k zuerst mit P, aber λ_g zuerst mit Q (vgl. Zeile 6 bzw. Zeile 11 auf Seite 2).

Method	s	m	λ_k	λ_g
Classic [Se80]	2	1		
Lomuto [Be84]	2	2		
YBB [WN12]	3	2		
Sedgewick [Se80]	3	1.5		
Waterloo [Ku14]	4	2		

Tab. 1: Parameterwerte für bekannte Quicksort-Varianten

Pivot Sampling. Außerdem können die $s-1$ Pivots $P_1, \ldots, P_{s-1}$ als $\boldsymbol{\tau}$-Ordnungsstatistiken aus einem Sample der Größe $k \geq s-1$ gezogen werden; konkret ist P_1 das τ_1-Quantil des Samples, P_2 das $(\tau_1+\tau_2)$-Quantil, …, P_{s-1} das $(\tau_1+\cdots+\tau_{s-1})$-Quantil. Dabei ist

$\boldsymbol{\tau} = (\tau_1, \ldots, \tau_s) \in (0,1)^s$ ein Vektor mit $\tau_1 + \cdots + \tau_s = 1$, der die erwarteten relativen Größenanteile der s Segmente angibt. Für den Median von 3 ist $k = 3$ und $\boldsymbol{\tau} = \left(\frac{1}{2}, \frac{1}{2}\right)$, max von 2 entspricht $k = 2$ und $\boldsymbol{\tau} = \left(\frac{2}{3}, \frac{1}{3}\right)$; $k = s - 1$ bedeutet kein Pivot Sampling. Weitere Details und Pseudocode finden sich in Abschnit 4.3 von [Wi16].

Wir legen uns mit diesem Schema darauf fest, dass die Partitionierung in einem Durchlauf erreicht wird – nicht etwa in mehreren Phasen, die jeweils alle Elemente betrachten – und in-place ausgeführt wird, also mit konstant viel Zusatzspeicher; da alle gängigen Bibliotheksimplementierungen so arbeiten, stellt das keine echte Einschränkung dar.

6 Vereinheitlichte Average-Case Analyse

Der Beitrag meiner Dissertation ist die parametrische Average-Case Analyse dieses allgemeinen Partitionierungsverfahrens inklusive Pivot Sampling. Auch für die Kostenmaße konnte ich ein parametrisches Schema ausarbeiten, dass die konkreten Kostenmaße von Interesse als Spezialfall umfasst: Vergleiche, Schreibzugriffe, gescannte Elemente, und sogar *Branch Mispredictions.* Die Analyse verallgemeinert und vereinheitlicht damit mehrere Arbeiten, in denen jeweils einzelne Quicksort-Varianten und Kostenmaße getrennt untersucht wurden. Im Laufe der Zeit habe ich für diese Analyse eine Vielzahl mathematischer Fakten und Methoden zusammengetragen und teilweise weiterentwickelt. Dabei habe ich mich bemüht, neben den formalen Beweisen stets auch (m)eine Intuition zu vermitteln, warum eine gewisse Aussage gilt. Das resultierende Kapitel 2 in [Wi16] wurde so zu einer umfassenden Sammlung der mathematischen Methoden zur Analyse von Quicksort. Das Ergebnis der Analyse lässt sich schließlich als das folgende Theorem zusammenfassen.

Theorem 1 (Theorem 7.1 aus [Wi16], gekürzte Fassung): *Quicksort mit generischem one-pass in-place Partitionieren sortiert eine zufällige Permutation von n Elementen in Erwartung mit $C_n = \frac{a_C}{\mathcal{H}} n \ln n \pm O(n)$ Schlüsselvergleichen und $SE_n = \frac{a_{SE}}{\mathcal{H}} n \ln n \pm O(n)$ gescannten Elementen, wobei die Konstanten von den Algorithmus-Parametern abhängen:*

$$a_C = \sum_{r=1}^{s} \tau_r \cdot \left(\frac{\lambda_k(r) \cdot \left(\sigma_{\rightarrow} + \left[r \le \lfloor m \rfloor\right]\right)}{\kappa - \sigma_{\leftrightarrow} + \left[r \le \lfloor m \rfloor \vee r > \lceil m \rceil\right]} + \frac{\lambda_g(r) \cdot \left(\sigma_{\leftarrow} + \left[r > \lceil m \rceil\right]\right)}{\kappa - \sigma_{\leftrightarrow} + \left[r \le \lfloor m \rfloor \vee r > \lceil m \rceil\right]} \right), \quad (1)$$

$$a_{SE} = \sum_{i=1}^{\lceil m \rceil} i \cdot \tau_{\lceil m \rceil - i + 1} + \sum_{j=1}^{s - \lfloor m \rfloor} j \cdot \tau_{\lfloor m \rfloor + j} - \frac{\sigma_{\leftrightarrow}}{\kappa}, \quad (2)$$

$$\mathcal{H} = \sum_{r=1}^{s} \tau_r (H_\kappa - H_{\sigma_r}). \quad (3)$$

Symbol	*Bedeutung*
κ	$k+1$
$\boldsymbol{\sigma}$	$\boldsymbol{\tau} \cdot \kappa = (\tau_1 \cdot \kappa, \ldots, \tau_s \cdot \kappa)$
$\sigma_{\rightarrow}$	$\sigma_1 + \cdots + \sigma_{\lfloor m \rfloor}$
$\sigma_{\leftrightarrow}$	$\sigma_{\lfloor m \rfloor + 1} + \cdots + \sigma_{\lceil m \rceil}$
$\sigma_{\leftarrow}$	$\sigma_{\lceil m \rceil + 1} + \cdots + \sigma_s$
$\lambda(r)$	*Tiefe des r ten Blattes in* λ
H_n	$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$
[*Aussage*]	1 *falls Auss. wahr,* 0 *sonst*

Insbesondere die Möglichkeit zweier Vergleichsbäume erschwert die Analyse, da die Partitionierungskosten dann eine nicht-lineare Funktion der zufälligen Segmentgrößen sind. Darum hängt z. B. a_C nicht nur von $\boldsymbol{\tau}$, sondern von k direkt ab. Der erste Schritt der Analyse besteht darin, n unabhängige, uniform in $(0,1)$ verteilte reelle Zahlen statt der üblichen Permutation von $\{1, \ldots, n\}$ als Eingabe zu betrachten. Das entkoppelt den *Wert P* eines

Pivots von seinem *Rang* in der Eingabe; bedingt auf die Zufallsgröße P, sind die Klassen der anderen Elemente dann *unabhängig und identisch verteilt!* Die Pivotwerte definieren also eine Wahrscheinlichkeitsverteilung $\boldsymbol{D} = (D_1, \ldots, D_s)$ für die s Klassen. Daraus folgt, dass das Pivot Sampling nur noch die Verteilung $\boldsymbol{D}$ direkt beeinflusst, die Kosten des Partitionieren für festes $\boldsymbol{D}$ dagegen nicht.

Entropie-Intuition. Auf dieser Basis lässt sich ein Teil von Theorem 1 intuitiv nachvollziehen: Pro Element benötigen wir (im Mittel) a_C Vergleiche für seine Klassifizierung und „lernen" dadurch $\mathcal{H}(\boldsymbol{D}) = -\sum_{r=1}^{s} D_r \log_2(D_r)$ Bits an Information über die Eingabe, wobei $\mathcal{H}(\boldsymbol{D})$ die *Shannon-Entropie* der Klassenverteilung ist. Da $\boldsymbol{D}$ von den zufälligen Pivotwerten abhängt, müssen wir die *erwartete* Entropie berechnen. Diese ergibt sich schließlich gerade zu $\mathcal{H} = \mathbb{E}[-\sum_{r=1}^{s} D_r \ln(D_r)] = \ln(2)\mathbb{E}[\mathcal{H}(\boldsymbol{D})]$ aus Gleichung (3); der Informationsgewinn eines Vergleichs ist also $a_C/(\mathcal{H}\ln 2)$. Da wir insgesamt $\log_2(n!) \sim n\log_2(n)$ Bits an Information lernen müssen um zu sortieren, bedarf es dazu im Mittel einer Anzahl von $\frac{a_C}{\mathcal{H}} n \ln n$ Vergleichen. Auch wenn es noch einer rigorosen Fehlerabschätzung bedarf (siehe Kapitel 6 von [Wi16]), liefert diese Argumentation das korrekte Ergebnis.

7 Diskussion der Ergebnisse

Auf Basis von Theorem 1 lassen sich zwei gegebene Quicksort-Varianten vergleichen, doch aufgrund der wechselseitigen Abhängigkeiten ist der Einfluss einzelner Parameter unklar. Kapitel 7 in [Wi16] enthält eine umfassende Diskussion dieser Einflüsse und bestimmt optimale Parameterwerte. Oft sind die Ergebnisse auf den ersten Blick unerwartet: Beispielsweise ist eine leicht *asymmetrische* Pivotwahl, also $\boldsymbol{\tau} \neq \left(\frac{1}{s}, \ldots, \frac{1}{s}\right)$, meist *vorteilhaft* (siehe Abb. 4). Ohne Pivot Sampling sind 5 Pivots bzgl. gescannter Elemente optimal, für noch größere s steigt der Aufwand wieder. Mit passenden Sampling dagegen lassen sich aus jedem weiteren Pivot asymptotisch Vorteile ziehen; die Anzahl gescannter Elemente konvergiert dann gegen $n\log_3(n)$. Ebenso kann man durch passendes Sampling für jeden noch so unbalancierten Vergleichsbaum sicherstellen, dass Quicksort die optimale Vergleichszahl $n\log_2(n)$ im Grenzfall für große Samples erreicht – sofern man $\lambda_k = \lambda_g$ wählt. (Mit $\lambda_k \neq \lambda_g$ erreicht man $n\log_2(n)$ nie!) Für realistische Samplegrößen k gibt es aber durchaus bessere und schlechtere Vergleichsbäume.

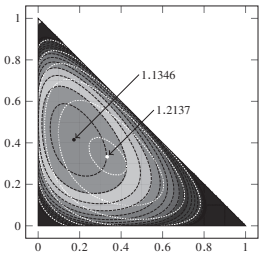


Abb. 4: Gescannte Elem. in YBB als Funktion von (τ_1, τ_2) für $k \to \infty$.

Faire Vergleiche. Obige asymptotisch optimale Varianten sind für realistische Eingabegrößen von begrenztem Nutzen, weil sich dort große Konstanten im $O(n)$-Fehlerterm verstecken: Da nur linear viele Partitionierungen stattfinden, verschwinden die Kosten des Pivot Sampling im $O(n)$-Term. Das bedeutet, dass man beim direkten Vergleich von z. B. klassischem und YBB-Quicksort selbst ohne Sampling Äpfel mit Birnen vergleicht: YBB darf (und muss) seine beiden Pivots sortieren, was sich aber nicht im führenden Term der Kosten niederschlägt, während wir dem klassischen Quicksort dies nicht zugestehen – kein Wunder, dass da YBB besser abschneidet. Um diesen Vergleich fair zu gestalten, habe ich das Vehikel des *Entropie-äquivalenten Samplings* entwickelt: Indem

man k und τ für den klassischen Quicksort *zufällig* nach einer gewissen Verteilung wählt (siehe Proposition 7.2 in [Wi16]), erhält man für beide Varianten den *gleichen* Nenner $\mathcal{H}$. Für das Beispiel aus Abschnitt 4 muss man mit Wahrscheinlichkeit $\frac{3}{5}$ das Maximum von 2 wählen und sonst gar nicht samplen. Weil eine YBB-Partitionierung günstiger als $\frac{5}{3}$ binäre Partitionierungen unter diesem Entropie-äquivalenten Sampling Schema ist, kann man von einem intrinsischen Vorteil des Drei-Wege-Partitionierens sprechen. Die Verallgemeinerung dieses Vergleichs liefert schließlich die Hauptaussage meiner Arbeit: *Bezüglich der Anzahl Vergleiche sind keine bzw. kaum Einsparungen möglich – die Vorteile, die in einigen Arbeiten ausgewiesen werden, sind ausschließlich der Übervorteilung durch bessere Pivots geschuldet – bezüglich der Anzahl gescannter Elemente hat Mehrwege-Quicksort aber originäre Vorteile.*

Gleiche Schlüssel. Neben diesen Ergebnissen zu zufälligen Permutationen, die stets n verschiedene Schlüssel betrachten, konnte ich wesentliche Teile der Analyse auch auf Eingaben mit vielen *gleichen* Schlüsseln erweitern (Kapitel 8 in [Wi16]) und eine lange bestehende, unbewiesene Vermutung von Robert Sedgewick und Jon Bentley bestätigen [SB02].

8 Fazit

Durch die Einbeziehung des Bedarfs an Speicherbandbreite in die mathematische Analyse konnte eine plausible Erklärung für die jüngsten Erfolge von Mehrwege-Quicksort-Varianten in der Praxis gegeben werden: Ähnlich zu Mehrwege-Mergesort sortiert Mehrwege-Quicksort mit weniger Durchläufen über die Eingabe und spart dadurch Speicherzugriffe; die geringen Kosten innerhalb der CPU bleiben dabei fast identisch erhalten. Durch die Analyse einer umfassenden Familie von Quicksort-Varianten können wir die Suche nach der Sortiermethode für das 21. Jahrhundert auf wenige Kandidaten eingrenzen.

Literatur

[AD15] Aumüller, Martin; Dietzfelbinger, Martin: Optimal Partitioning for Dual-Pivot Quicksort. ACM Transactions on Algorithms, 12(2):1–36, November 2015.

[ADK16] Aumüller, Martin; Dietzfelbinger, Martin; Klaue, Pascal: How Good Is Multi-Pivot Quicksort? ACM Transactions on Algorithms, 13(1):1–47, 2016.

[Ah16] Ahonen, Tomi T.: , Smartphone Bloodbath Market Share Update Q1: All the Top 10 brands plus OS shares plus installed base. http://communities-dominate.blogs.com/brands/2016/05/smartphone-bloodbath-market-share-update-q1-all-the-top-10-brands-plus-os-shares-plus-installed-base.html, 2016.

[AV88] Aggarwal, Alok; Vitter, Jeffrey S.: The Input/Output Complexity of Sorting and Related Problems. Communications of the ACM, 31(9):1116–1127, August 1988.

[Ba78] Backus, John: Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs. Communications of the ACM, 21(8):613–641, August 1978.

[Be84] Bentley, Jon: Programming pearls: how to sort. Communications of the ACM, 27(4):287–291, April 1984.

[BM93] Bentley, Jon L.; McIlroy, M. Douglas: Engineering a sort function. Software: Practice and Experience, 23(11):1249–1265, 1993.

[Co09] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford: Introduction to Algorithms. MIT Press, 3rd edition, 2009.

[He91] Hennequin, Pascal: Analyse en moyenne d'algorithmes : tri rapide et arbres de recherche. Thèse (Ph. D. Thesis), Ecole Politechnique, Palaiseau, 1991.

[Ho62] Hoare, C. A. R.: Quicksort. The Computer Journal, 5(1):10–16, January 1962.

[Ja09] Java Core Library Development Mailing List: , Replacement of Quicksort in java.util.Arrays with new Dual-Pivot Quicksort, 2009.

[Ja11] Jackson, Nicholas: Mousetraps: A Symbol of the American Entrepreneurial Spirit. The Atlantic, 2011.

[JD09] JDK Bug System: , Replace modified mergesort in java.util.Arrays.sort with timsort. https://bugs.openjdk.java.net/browse/JDK-6804124, 2009.

[Ku14] Kushagra, Shrinu; López-Ortiz, Alejandro; Qiao, Aurick; Munro, J. Ian: Multi-Pivot Quicksort: Theory and Experiments. In: Meeting on Algorithm Engineering and Experiments (ALENEX). SIAM, pp. 47–60, 2014.

[Mc07] McCalpin, John D.: Sustainable Memory Bandwidth in High Performance Computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. continually updated technical report.

[NWM16] Nebel, Markus E.; Wild, Sebastian; Martínez, Conrado: Analysis of Pivot Sampling in Dual-Pivot Quicksort. Algorithmica, 75(4):632–683, August 2016.

[SB02] Sedgewick, Robert; Bentley, Jon: , Quicksort is Optimal (Talk Slides), 2002.

[Se78] Sedgewick, Robert: Implementing Quicksort programs. Communications of the ACM, 21(10):847–857, 1978.

[Se80] Sedgewick, Robert: Quicksort. Reprint of the author's Ph. D. thesis, 1980.

[Wi16] Wild, Sebastian: Dual-Pivot Quicksort and Beyond: Analysis of Multiway Partitioning and Its Practical Potential. Doktorarbeit, Technische Universität Kaiserslautern, 2016.

[WM95] Wulf, William Allen; McKee, Sally A.: Hitting the Memory Wall: Implications of the Obvious. ACM SIGARCH Computer Architecture News, 23(1):20–24, March 1995.

[WN12] Wild, Sebastian; Nebel, Markus E.: Average Case Analysis of Java 7's Dual Pivot Quicksort. In (Epstein, Leah; Ferragina, Paolo, eds): European Symposium on Algorithms (ESA). volume 7501 of LNCS. Springer, pp. 825–836, 2012.

Sebastian Wild hat an der Technischen Universität Kaiserslautern mit einem Stipendium der Studienstiftung Informatik studiert und 2012 mit dem Master of Science abgeschlossen. Nach dem Studium blieb er an der TU Kaiserslautern als wissenschaftlicher Mitarbeiter tätig und promovierte bei Prof. Dr. Markus Nebel. Seine Forschungsergebnisse im Bereich der Analyse von Algorithmen führten schon zu Beginn der Promotion zu mehreren Publikationen mit internationalen Kooperationen und einem *Best Paper Award* [WN12]. Er war während des Studiums regelmäßig als Übungsleiter tätig und als wissenschaftlicher Mitarbeiter in der inhaltlichen und organisatorischen Betreuung, sowie der Konzeption neuer Lehrveranstaltungen involviert. Sebastian Wild engagierte sich als Vertreter der wissenschaftlichen Mitarbeiter im Prüfungsausschluss. Er ist verheiratet und Vater von drei Kindern.