Dual-Pivot and Beyond: The Potential of Multiway Partitioning in Quicksort

Sebastian Wild

Abstract: Since 2011 the Java runtime library uses a Quicksort variant with two pivot elements. For reasons that remained unclear for years it is faster than the previous Quicksort implementation by more than 10%; this is not only surprising because the previous code was highly-tuned and is used in many programming libraries, but also since earlier theoretical investigations suggested that using several pivots in Quicksort is not helpful.
In my dissertation I proved by a comprehensive mathematical analysis of all sensible Quicksort partitioning variants that (a) indeed there is hardly any advantage to be gained from multiway partitioning in terms of the number of comparisons (and more generally in terms of CPU costs), but (b) multiway partitioning does significantly reduce the amount of data to be moved between CPU and main memory. Moreover, this more efficient use of the memory hierarchy is not achieved by any of the other well-known optimizations of Quicksort, but only through the use of several pivots.

ACM CCS: Theory of computation → Design and analysis of algorithms → Data structures design and analysis → Sorting and searching

Keywords: Quicksort, multiway partitioning, average-case analysis, cache misses, external-memory model

1 Introduction

Sorting is one of the basic stepping stones for solving more interesting tasks, and thus used in some form or another in any software application. How to sort most efficiently might be the most well-studied and best-understood algorithmic problem and programmers can rely on robust and efficient implementations in programming libraries for sorting an array of elements in main memory. Yet, a silent revolution took place in the practical side of this well-understood problem: *All* sorting methods in Oracle's Java runtime library have been rewritten entirely within the last ten years [10, 11]! As this code forms the base of the Android runtime library, these new sorting methods might easily be among the most executed algorithms in existence. (Estimates speak of 2 billion active Android devices (2016 worldwide) [2].)

For such widely used libraries, new trends are adopted very conservatively; any change has the potential to affect existing use cases. Indeed – until ten years ago – all major programming libraries used a sorting method based on the Quicksort Implementation developed by Jon Bentley and Douglas McIlroy in the early 1990s for the C standard library [6]. Nevertheless, the running time

```
DualPivotQuicksort(A, left, right)    // Sorts A[left..right]
 1  if right − left ≥ 1
 2      P = min{A[left], A[right]} // Pivot 1
 3      Q = max{A[left], A[right]} // Pivot 2
 4      k = left + 1;  ℓ = k;  g = right − 1
 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
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
```

Figure 1: The algorithmic core of dual-pivot Quicksort as used in Java 7. Figure 3 shows the state of the array after line 18.

improvements by the new methods Dual-Pivot Quicksort and Timsort (a Mergesort variant used for stable sorting) eventually became too big to ignore.

Remarkably, this dual-pivot Quicksort was not proposed by an algorithms expert; quite the contrary: young Russian software developer and puzzle enthusiast Vladimir Yaroslavskiy, at the time working for Sun Microsystems, played with faster sorts in his free time and discovered the potential of multiway partitioning. Later joined by Jon Bentley and Java expert Joshua Bloch, Yaroslavskiy developed the implementation now used in the Java library. It contains a handful of clever mechanisms for robust performance on all kinds of input distributions; for random permutations of distinct elements the Java code is essentially equivalent to the pseudocode in Figure 1.

Arguably, using two pivots is a natural generalization of Quicksort and Figure 1 is a quite straight-forward implementation of that idea. How could its potential have escaped the eyes of so many researchers worldwide? And more importantly: if two pivots are good, could more pivots be even better?

These questions were the starting point of my research. Before we can analyze in how far more pivots help or harm, we have to understand *why* dual-pivot Quicksort is faster than the "classic" single-pivot version. We can now give a plausible explanation for that: The assumptions of traditional models of running time are not fulfilled on modern computers to the extend they were fulfilled 20 years ago, when Bentley and McIlroy designed their classic Quicksort implementation [6]. Dual-pivot Quicksort is not a new idea, but when it was studied in the past [16, 8], it was correctly found to *not* save comparisons and it was – for the machines of the time adequately – concluded to not improve Quicksort's running time. So the idea of multiway partitioning is not new; the news is that it nowadays makes Quicksort faster. And the reason for that is a continuing trend in computer hardware design, that now also affects sorting in internal memory.

2 The "Memory Wall"

Moore's Law predicts the number of transistors per area in integrated circuits to *double* roughly every two years. Although details and the sustainability of this growth rate are certainly debatable, it reflects past improvements of CPU peak performance to within a reasonable accuracy. To profit from that applications also need to get their data faster; but access times of main memory and (net) transfer speeds of the connecting bus systems could by far not keep up with this rate of growth: According to the historical data of John McCalpin's STEAM benchmark [13], CPU peak performance grew by 46% per year over the last 25 years, whereas the observable memory bandwidth (the amount of data transferable between CPU and RAM in one time unit) grew only by an annual 37% – the imbalance is growing exponentially!

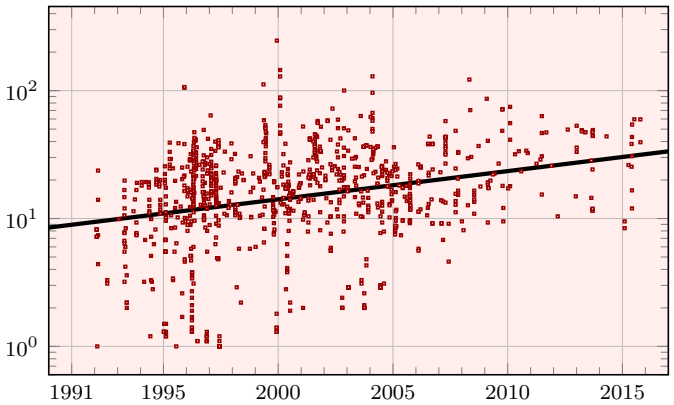


Figure 2: Development of machine balance over the last 25 years. Each point is a result of the STEAM benchmark: x is time, y is quotient of CPU peak performance (MFLOPS) divided by net memory bandwith (MWps in "triad" benchmark) on logarithmic scale. The line is the regression line of all points. (Further details on the data are given in [17, Fig. 1].

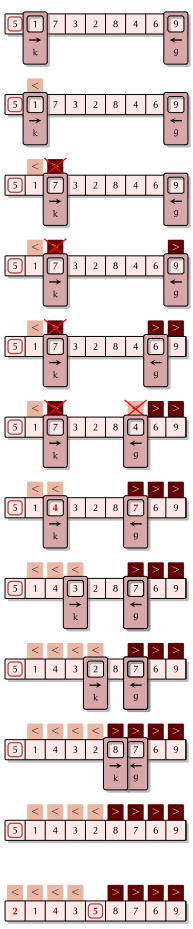
Figure 2 shows the quotient of CPU peak performance and memory bandwidth, the "machine balance", for all reported benchmark results: (These numbers average over various types of machines and timestamps are not uniform, but the qualitative trend is undeniable.)

Improving processing power and memory capacity does not speed up a program if it has to wait for data to be transfered. Hierarchies of faster caches and automatic prefetching alleviate the problem only partially – if the (net) bus bandwidth itself becomes the limiting factor, they do not help either. John Backus recognized this issue already in 1977; he called it the *von-Neumann-Bottleneck* [4]; in 1995, William Wulf und Sally McKee [19] coined the more drastic phrase to "hit a memory wall" when a system's overall performance is dominated by memory speed. This extreme state is certainly not (yet?) reached for most applications, but the balance is shifting: in 1993, when Bentley and McIlroy published their Quicksort implementation, a typical imbalance of CPU speed vs. bandwidth was 7:1. Today it is rather 30:1.

3 Quicksort and Memory Bandwidth

What does it mean for sorting methods that memory accesses became more expensive in relation to CPU time? This is depends on details of the algorithms, so let us have a closer look at Quicksort. Its core idea is to determine the *rank* of an (arbitrary) pivot element by comparing it with all others. That determines the position of the pivot in the sorted array and we can deal recursively with the smaller resp. larger elements. While determining the rank, we simultaneously *partition* the array into segments: smaller elements left and larger elements right.

The most-used partitioning strategy is illustrated to the right. It is due to C. A. R. Hoare [9] and Robert Sedgewick [15] and was used up to version 6 of the Java runtime library (and in many other libraries). The method works in place and uses sequential scans that guarantee maximal locality of reference for caches. Indeed all practically relevant partitioning strategies do that. By moving outside-in, the scanning indices k and g together scan the whole array exactly once. Simpler codes like Lumoto's method [5, 7] do *not* have this property.



To analyze and compare bandwidth consumption we need a precisely defined model for the costs of an execution. The model I propose for that purpose allows access to the array only via *iterators*. An iterator points to a certain array position and allows to read and write the value there. Unlike general pointers, we can only move iterators to *neighboring* positions. In the illustration to the right, iterators are shown as rectangles with a "window"; note that we have 2 active iterators here.

The cost of an execution is the number of *scanned elements:* the number of visited elements summed over all iterators, or equivalently, the overall number of iterator movements. This counts areas twice if they are scanned twice. The number of scanned elements was demonstrated to be roughly proportional to the number of *(Level 1) Cache Misses* [14, 3]. For each active iterator, we need one cache line and only every B movements, a cache miss occurs (for B the cache's block size). Different iterators cause cache misses independently unless very close in space *and* time. Our model is similar to the classic external-memory model [1], but I want to avoid the latter's terminology to not suggest that data movements are the *only* important operation; indeed, CPU time and scanned elements need to be balanced.

4 Dual-Pivot Partitioning

The classic Hoare-Sedgewick partitioning needs n scanned elements since iterators k and g together visit each element once; this coincides with the number of key comparisons used in the process. The known results for comparisons in classic Quicksort (e. g., [16]) thus also give the scanned elements. For Yaroslavskiy-Bentley-Bloch (YBB) partitioning (Figure 1), the average number of scanned elements is $\frac{4}{3}n$ as illustrated in Figure 3. In contrast, the average number of comparisons is $\frac{19}{12}n$ [18].

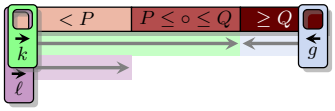


Figure 3: State after YBB partitioning. k and g together scan n elements, ℓ on average the first third a second time.

We can obtain the same subdivision into three segments by *two rounds of classic partitioning*, see Figure 4. Compared to YBB, we now scan the middle segment *twice* and need $\frac{5}{3}n$ scanned elements instead of $\frac{4}{3}n$ – YBB gets the same work done with less overall bandwidth consumption!

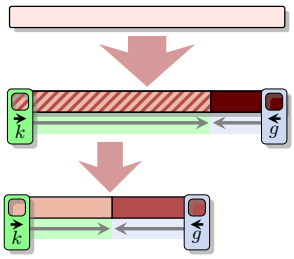


Figure 4: Simulating 3-way by 2-way partitioning: first all elements around Q, then the left segment around P.

Comparing multiway Quicksort to a simulation by classic partitioning is an insightful point of view, but one has to account for the pivot distributions (for more complex measures of costs). Assuming all input permutations to be equally likely, the rank of the pivot in classic Quicksort is uniformly distributed. In the simulation above this is no longer true: the larger pivot Q is the maximum of two elements, so its ranks tend to be larger. Effectively, we draw Q as an order statistic of a sample (max of 2). P in the second step happens to be uniform in its range, though.

5 Dual-Pivot and Beyond

To assess the potential of multiway partitioning in general, my dissertation extends the above ideas in several ways. I describe a parametric template algorithm for partitioning with any given number of pivots that generalizes all practically relevant partitioning methods, and I analyze this generic method in different models of costs (including scanned elements and key comparisons). The typical optimizations used in fast implementations are also taken into account, in particular the choice of the pivots from a small sample (e. g., median-of-3 for classic Quicksort). All results are precise asymptotic approximations including constant factors. Apart from those results themselves, my dissertation also surveys many techniques for the analysis of algorithms that help to better understand Quicksort's behavior.

The result of the analysis allows to predict the costs (in different cost models) for any given Quicksort variant. Interesting is also the influence of the algorithmic parameters, in particular s, the number of segments (corresponding to $s-1$ pivots). If we consider the number of comparisons, the traditional cost model, a larger s seems to save comparisons, but most of this improvement is actually spurious: it is due to better pivots that we get automatically from sorting the $s-1$ pivots. (A three-pivot method must sort its pivots, but the cost of that is not reflected in the leading-term approximation. We thereby give it an unfair advantage over Quicksort with one pivot.) If we compare s-way partitioning and its

simulation by classic binary partitioning, the number of comparisons is almost the same. Multiway partitioning does not reduce the number of comparisons significantly.

When we consider the number of scanned elements, however, multiway partitioning leads to a significant reduction (20% in the case of YBB partitioning as shown above). If we do not allow pivots to be chosen from a sample, the optimal number of pivots with respect to scanned elements is 5 (leading to $s = 6$ segments), closely followed by 3 pivots (supporting the partitioning method of Kushagra et al. [12]). In the case of choosing pivots from a large sample (using appropriate order statistics), there is no finite optimal s: any further pivot improves the leading-term approximation of costs and the number of scanned elements converge to $n \log_3(n)$. These theoretically optimal variants have limited use for realistic input sizes, but are an important benchmark for practical parameter choices.

6 Summary

Quicksort had a lively youth with new variants and tweaks being proposed regularly. Robert Sedgewick studied many of them in his Ph. D. thesis [16] and by means of mathematical analysis reduced the pool of options to a few genuinely helpful optimizations, the most effective ones being pivot sampling and a cutoff to Insertionsort for small subproblems. Multiway partitioning was not among them.

I showed that – similar to multiway Mergesort – multiway partitioning has genuine potential to save memory transfers, and it is indeed the only optimization of Quicksort that does so. My dissertations gives a reassessment of the relative merits of various Quicksort variants in the spirit of Sedgewick's thesis, but taking the increasing relative costs of memory bandwidth into account, and thus guiding the search for 21st century library sorting methods.

Literature

[1] Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, August 1988.

[2] Tomi T. Ahonen. 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.

[3] Martin Aumüller, Martin Dietzfelbinger, and Pascal Klaue. How good is multi-pivot quicksort? *ACM Transactions on Algorithms*, 13(1):1–47, 2016.

[4] John Backus. 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.

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

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

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

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

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

[10] Java Core Library Development Mailing List. Replacement of quicksort in java.util.arrays with new dual-pivot quicksort, 2009.

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

[12] Shrinu Kushagra, Alejandro López-Ortiz, Aurick Qiao, and J. Ian Munro. Multi-pivot Quicksort: Theory and experiments. In *Meeting on Algorithm Engineering and Experiments (ALENEX)*, pages 47–60. SIAM, 2014.

[13] John D. McCalpin. Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. continually updated technical report.

[14] Markus E. Nebel, Sebastian Wild, and Conrado Martínez. Analysis of pivot sampling in dual-pivot Quicksort. *Algorithmica*, 75(4):632–683, August 2016.

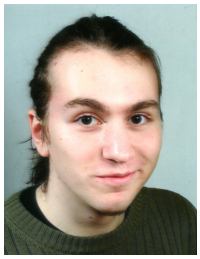
[15] Robert Sedgewick. Implementing Quicksort programs. *Communications of the ACM*, 21(10):847–857, 1978.

[16] Robert Sedgewick. *Quicksort*. Reprint of the author's Ph. D. thesis, Garland Publishing, 1980.

[17] Sebastian Wild. *Dual-Pivot Quicksort and Beyond: Analysis of Multiway Partitioning and Its Practical Potential*. Doktorarbeit (Ph. D. thesis), Technische Universität Kaiserslautern, 2016. ISBN 978-3-00-054669-3.

[18] Sebastian Wild and Markus E. Nebel. Average case analysis of Java 7's dual pivot Quicksort. In Leah Epstein and Paolo Ferragina, editors, *European Symposium on Algorithms (ESA)*, volume 7501 of *LNCS*, pages 825–836. Springer, 2012.

[19] William Allen Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, March 1995.



Dr. Sebastian Wild studied computer science at *Technische Universität Kaiserslautern* on a scholarship by *Studienstiftung des deutschen Volkes* and graduated in 2012 with a Master of Science. After that he did his Ph. D. as *wissenschaftlicher Mitarbeiter* (employed doctoral candidate with teaching duties) in the research group of Prof. Dr. Markus Nebel. His findings in the field of analysis of algorithms soon led to publications and international collaborations, including a *Best Paper Award* [18] at the European Symposium on Algorithms 2012. Sebastian was continually involved in teaching. During his studies he was a student tutor and during his Ph. D. years, he was responsible for organizing tutorials and involved in the development of new courses. Sebastian is married and father of three children.

Address: University of Waterloo, David R. Chariton School of Computer Science, DC 2332, University Avenue 200 East, Waterloo, ON N2T 2K6, Canada. E-Mail: wild@uwaterloo.ca