Discussion of *Multi-Pivot Quicksort: Theory and Experiments*

Paper by Kushagra, López-Oritz, Munro, and Qiao
Discussion by Nisha Masharani

Multi-pivot quicksort is a modification of a canonical sorting algorithm, quicksort. Quicksort has two steps: a partition step and a recursive step. In the partition step, a pivot is chosen, and then all elements that are smaller than that pivot are moved to one side of that pivot, and all elements that are larger than that pivot are moved to the other side. In the recursive step, the two parts of the array that are on either side of the pivot are recursively partitioned and similarly undergo the recursive step. The runtime of quicksort, O(n log n), is the theoretical lower bound for sorting, so it is not an inefficient algorithm. However, quicksort does not take advantage of modern computer architecture. Specifically, in recent years, there have been vast improvements in caching behavior that quicksort does not utilize fully. By using a technique that does take advantage of this type of caching behavior, the constant factor in the run time can be improved upon. In their paper, Kushagra et al. demonstrate a multi-pivot quicksort algorithm that takes advantage of modern caching, and therefore demonstrates both theoretical and practical performance improvements.

The history of multi-pivot quicksorts is a fairly short one. In 2009, a man named Vladimir Yaroslavskiy posted on an open source Java forum detailing a dual-pivot quicksort algorithm that outperformed traditional single-pivot quicksort, primarily by reducing the number of swaps by 20%.[1] The algorithm quickly became famous as having upset the status quo, and was integrated into Java 7 as the default built-in sorting algorithm. Soon after Yaloslavskiy's

[1] http://permalink.gmane.org/gmane.comp.java.openjdk.core-libs.devel/2628

results were published, other people began asking if more pivots meant faster sorting in general. During my research, I found several blog and stack overflow posts detailing informal studies of n-pivot quicksorts, but none of these studies were ever thoroughly analyzed or published.[2]

Kushagra, et al. present a thorough description and analysis of a three-pivot quicksort algorithm, as well as a rigorous analysis of the caching behavior of single-pivot, dual-pivot, and three-pivot quicksorts. The authors claim that it is the caching behavior of the three-pivot quicksort algorithm that causes it to perform better than single-pivot and dual-pivot quicksorts in experiments, although they also note that these performance improvements are architecture dependent. They prove four types of performance for the algorithm theoretically, and then present the results of their experiments as further evidence for the strong performance of their algorithm.

The three pivot quicksort algorithm works as such: firstly, three pivots (p, q, and r) are chosen and sorted in increasing order. Then, the array is traversed and partitioned into four sub-arrays such that, for any element x in the array:

1. If $x < p$, then x is swapped to the left of p.
2. If $p \leq x < q$, then x is swapped to the space between p and q.
3. If $q \leq x < r$, then x is swapped to the space between q and r.
4. If $r \leq x$, then x is swapped to the right of r.

To do the actual partition, four pointers are used to separate the unsorted region and the four sub-arrays. These pointers traverse from the ends of the array to the center until they cross, at which point the pivots are put into place at the end of the partition step. An important intuition to

[2] For instance, http://porg.es/blog/n-pivot-quicksort

maintain, and one that we will revisit, is that, at each step, the current element is compared to the central pivot, q, and is then compared to either pivot p or pivot r depending on whether the element is less than q or greater than q. This algorithm is very similar to Yaroslavskiy's dual-pivot quicksort algorithm, which uses similar techniques to traverse the array and partition the elements. However, the authors demonstrate that this algorithm has theoretical improvements upon both single-pivot quicksort and Yaroslavskiy's dual-pivot quicksort.

In the theoretical analysis of the performance of the three-pivot quicksort, the authors discuss four different measures of cost by which the performance can be measured. The first two types of cost are familiar to anybody who has done an analysis of single-pivot quicksort: the number of comparisons made during the course of running the algorithm and the number of swaps made during the algorithm. The third and fourth types of cost are more unfamiliar, but are vital to the authors' analysis: the number of cache misses and the number of recursive calls to a subproblem larger than the size of a cache block.

In calculating the cost, or the expected number of occurrences, for each of these categories, the authors use a cost function that has two parts. The first part is the cost of performing a partition on an array of size n. The second part is the cost of performing the recursive calls on each of the subarrays. The resulting equation is as follows:

$$f_n = p_n + \binom{n}{3}^{-1} \sum_{i=0}^{n-3} \sum_{j=i+1}^{n-2} \sum_{k=j+1}^{n-1} (f_i + f_{j-i-1} + f_{k-j-1} + f_{n-k-1})$$

where the f_x elements are the costs of the recursive calls on the subarrays delineated by the pivots. Using summation identities, this simplifies to:

$$f_n = p_n + \binom{n}{3}^{-1} \sum_{i=0}^{n-3} (n-i-1)(n-i-2) f_i$$

In these equations, f_n is the total cost of running the algorithm on an array of length n, p_n is the cost of the partition step, and the summation expression is the total cost of running the algorithm on the subarrays created by the partition. In the first equation, i, j, and k are the indices of the three pivots; j and k are simplified out in the second equation. Since the size of the subarrays affects the cost of the recursive calls, the cost function uses probabilistic expectation for the cost of running the algorithm on the subarrays. To calculate this expectation, Kushagra, et al. must make an important assumption: that any combination of three pivots is chosen with equal probability—meaning the probability of any three pivots being chosen is $C(n, 3)^{-1}$. While this assumption works for a theoretical analysis and holds for many of the authors' experiments, this assumption does not necessarily work in every practical case, so the costs in applications can vary from the theoretical predictions and experimental results found in this paper. However, with that assumption, this equation means that calculating the cost means figuring out the cost of the partition step only, then plugging that expression in for p_n and simplifying.

The first type of cost, the number of comparisons, is one of the two types of cost thoroughly analyzed in the canonical proofs of running time for single-pivot quicksort. For single-pivot quicksort, the number of comparisons is known to be $\theta(n \log n)$, which is a theoretical lower bound and is therefore the same for dual-pivot and three-pivot quicksort. However, the constant factor in front of the $n \log n$ is different for the three variations of the sort. For single-pivot quicksort, the actual cost is $2.0n \log n$, while for Yaroslavskiy's dual-pivot quicksort, the number of comparisons is $1.9n \log n$. To prove the cost for three-pivot quicksort, the authors use the idea that, for every partition step, every element other than the pivots

undergoes two comparisons: first, to pivot q, and then either to pivot p or pivot r. The three pivots undergo a constant number of comparisons to sort them, leading to

$$p_n = 2(n-3) + \frac{8}{3}$$

overall comparisons during the partition step. By using this equation for p_n in the cost function described above, the authors derive a cost of 1.846*n log n,* which is clearly lower than the number of comparisons for both single-pivot and dual-pivot quicksort.

The second type of cost, the number of swap operations, is also a canonically studied method of run-time analysis for quicksort. Since a swap operation really refers to three operations—the copying of the first element into a temporary variable, the replacement of the first element by the second, and then the replacement of the second element by a temporary variable—this type of cost can actually be more indicative of the overall running time of the algorithm. Intuitively, the number of swaps can be thought of as a the number of swaps in a two step process: the number of swaps to partition around pivot q, and then the number of swaps to partition around p and r. The number of swaps to partition around pivot q was proven in prior work on single-pivot quicksort to be:

$$\frac{n-2}{6}$$

Then, the number of swaps to partition around pivots p and r is:

$$i + n - k$$

where *i* is the location of pivot p and *k* is the location of pivot r. Since the locations of i and k vary, like in the overall cost function, the authors use the expected number of swaps during the partition around pivots p and r, leading to an overall partition cost of:

$$p_n = \frac{n-2}{6} + \binom{n}{3}^{-1} \sum_{i=0}^{n-3} \sum_{j=i+1}^{n-2} \sum_{k=j+1}^{n-1} i + n - k$$

which simplifies to:

$$p_n = \frac{4n+1}{6}$$

which is then plugged into our cost function, yielding a cost of 0.62*n log n.*The second part of this partition requires more swaps than the traditional single-pivot quicksort, meaning that the overall number of swaps is actually greater than in single-pivot quicksort. Specifically, in single-pivot quicksort, the number of swaps is (1/3)*n log n*, while in dual-pivot quicksort the number of swaps is 0.6*n log n*.

The third type of cost, cache misses, is critical to the authors' argument. The authors claim that the three-pivot quicksort is experimentally faster than the other forms of quicksort because of the caching behavior of the algorithm. Intuitively, the number of cache misses can be thought of as proportional to the number of times at which each element is examined. For a single partition step in three-pivot quicksort, about half of the elements in the array, the elements between pivots p and r, are looked at only once, while half of the elements, namely, the ones to

the left of pivot p and to the right of pivot r, are looked at twice. Therefore, the entire array is, in expectation, looked at 1.5 times, yielding a partition step cost of:

$$p_n = \left(\frac{3}{2}\right)\frac{n+1}{B}$$

where B is the size of a cache block. To partition an array into four sub-arrays using single-pivot quicksort, each element is looked at once during the first partition step, and then each element gets looked at once again during the second partitions of the sub-arrays. In this case, the entire array is looked at twice. In the case of a perfect split (perfect quarters), the number of cache misses is 1.5 times the number of pages in the cache for three-pivot quicksort, and 2 times the number of pages in the cache for single-pivot quicksort. In the worst case, both algorithms go over the entire array twice, so the number of cache misses is twice the number of pages for both algorithms. Therefore, on average, the three-pivot quicksort results in fewer cache misses. Numerically, the number of cache misses in the single-pivot algorithm has a leading coefficient of (2), while the number of cache misses in the three-pivot algorithm has a leading coefficient of (18/13 = 1.38). The authors also perform a similar analysis of the behavior of Yaroslavskiy's dual pivot quicksort, and find that it is somewhere in the middle, with a coefficient of (8/5 = 1.6).

The last type of cost that the authors explore is the number of recursive calls to subproblems that are larger than the size of a cache block, which is closely related to the number of cache misses. Intuitively, an algorithm with smaller subproblems will have fewer recursive calls to subproblems larger than the size of a cache block, and this intuition holds in the authors' analysis. The partition cost, in this case, is simply $p_n = 1$, which is then similarly used in our cost

function to find the total cost. The cost for single-pivot quicksort has a leading coefficient of (2), while three-pivot quicksort has a leading coefficient of (12/13 = 0.92), which intuitively makes sense because three-pivot quicksorts splits an array into four subproblems, each of which are smaller than the two subproblems in single-pivot quicksort (outside of the worst case).

The authors claim that the improvements in the cost of cache misses and the number of large subproblems is responsible for an improvement in the running time of three-pivot quicksort over single-pivot quicksort. The three-pivot variant shows a 7-8% improvement over Yaroslavskiy's two-pivot algorithm, which the authors attribute to a 25% reduction in the number of cache misses. The authors then go on to verify these findings via rigorous experimentation, which they detail at length in the paper. To summarize, the authors tested six algorithms: single-pivot quicksort, dual-pivot quicksort, and three-pivot quicksort, as well as optimized versions of the algorithm. They wrote the tests in C, to prevent effects from the JVM and other complicated code environments, and then ran those tests multiple times and averaged the results. The arrays upon which the algorithm was run were random permutations of the numbers $1,\ldots,n$, where n is the size of the array. It is important to note that these conditions are the optimal conditions for all of the algorithms, and they saw performance decreases when they ran their algorithm on less than optimal arrays. As expected, the authors found that the dual-pivot variant was an improvement over the single-pivot quicksort, and that the three-pivot variant was an improvement over the dual-pivot variant. The optimized versions showed similar performance improvements.

The authors then ran several experiments to learn about the effects of computer architecture on the performance of the algorithm. Firstly, they ran the algorithm on multiple

platforms, and found that the performance on non-optimal arrays was sometimes dramatically different on different computers. They also ran the tests under different GCC optimization levels, and similarly found that those optimizations would result in different algorithms becoming faster on different computers. Lastly, they ran their experiments on a multi-core architecture using threading, and discovered that all three algorithms showed similar speedups when concurrency was added. The overall result of these tests was one of the authors' key points—that computer architecture can make a huge difference in the performance of sorting algorithms.

At the end of the paper, in addition to summarizing the successes of three-pivot quicksort, Kushagra, et al. give a bigger picture conclusion for the reader to consider. The authors postulate that, due to the advances in computer architecture that have been made in the last decade or so, canonical results in computer science may no longer hold. The authors use this claim to foreshadow further work in developing more multi-pivot quicksorts, but this claim is also applicable to other canonical algorithms. For example, throughout the course of this class, we have seen algorithms to approximate well-studied problems in graphs, for example, finding the min-cut or figuring out the connectivity of a graph, in new graph environments. I find this, perhaps, to be the most interesting question raised in the paper—if we can make such improvements on quicksort, which is one of the most well-studied, basic algorithms, what other basic algorithms can we improve upon given new computer architecture?