Downloaded 01/30/21 to 5.18.247.84. Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

# Multi-Pivot Quicksort: Theory and Experiments

Shrinu Kushagra
skushagr@uwaterloo.ca
University of Waterloo

Alejandro López-Ortiz
alopez-o@uwaterloo.ca
University of Waterloo

J. Ian Munro
imunro@uwaterloo.ca
University of Waterloo

Aurick Qiao
a2qiao@uwaterloo.ca
University of Waterloo

November 7, 2013

**Abstract**

The idea of multi-pivot quicksort has recently received the attention of researchers after Vladimir Yaroslavskiy proposed a dual pivot quicksort algorithm that, contrary to prior intuition, outperforms standard quicksort by a a significant margin under the Java JVM [10]. More recently, this algorithm has been analysed in terms of comparisons and swaps by Wild and Nebel [9]. Our contributions to the topic are as follows. First, we perform the previous experiments using a native C implementation thus removing potential extraneous effects of the JVM. Second, we provide analyses on cache behavior of these algorithms. We then provide strong evidence that cache behavior is causing most of the performance differences in these algorithms. Additionally, we build upon prior work in multi-pivot quicksort and propose a 3-pivot variant that performs very well in theory and practice. We show that it makes fewer comparisons and has better cache behavior than the dual pivot quicksort in the expected case. We validate this with experimental results, showing a 7-8% performance improvement in our tests.

## 1 Introduction

**1.1 Background** Up until about a decade ago it was thought that the classic quicksort algorithm [3] using one pivot is superior to any multi-pivot scheme. It was previously believed that using more pivots introduces too much overhead for the advantages gained. In 2002, Sedgewick and Bentley [7] recognised and outlined some of the advantages to a dual-pivot quicksort. However, the implementation did not perform as well as the classic quicksort algorithm [9] and this path was not explored again until recent years.

In 2009, Vladimir Yaroslavskiy introduced a novel dual-pivot partitioning algorithm. When run on a battery of tests under the JVM, it outperformed the standard quicksort algorithm [10]. In the subsequent release of Java 7, the internal sorting algorithm was replaced by Yaroslavskiy's variant. Three years later, Wild and Nebel [9] published a rigorous average-case analysis of the algorithm. They stated that the previous lower bound relied on assumptions that no longer hold in Yaroslavskiy's implementation. The dual pivot approach actually uses less comparisons ($1.9n \ln n$ vs $2.0n \ln n$) on average. However, the difference in runtime is much greater than the difference in number of comparisons. We address this issue and provide an explanation in §5.

Aumüller and Dietzfelbinger [1] (ICALP2013) have recently addressed the following question: If the previous lower bound does not hold, what is really the best we can do with two pivots? They prove a $1.8n \ln n$ lower bound on the number of comparisons for all dual-pivot quicksort algorithms and introduced an algorithm that actually achieves that bound. In their experimentation, the algorithm is outperformed by Yaroslavskiy's quicksort when sorting integer data. However, their algorithm does perform better with large data (eg. strings) since comparisons incur high cost.

**1.2 The Processor-Memory Performance Gap** Both presently and historically, the performance of CPU registers have far outpaced that of main memory. Additionally, this performance gap between the processor and memory has been increasing since their introduction. Every year, the performance of memory improves by about 10% while that of the processor improves by 60% [5]. The performance difference grows so quickly



Copyright © 2014.
by the Society for Industrial and Applied Mathematics.

Downloaded 01/30/21 to 5.18.247.84. Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

that increasingly more levels of cache (L1, L2 and L3) have been introduced to bridge the gap. This results in an ever-changing computer architecture where cache effects in programs gradually grow more significant.

**1.3 Our Work** We provide evidence that the recent discovery of fast multi-pivot quicksort algorithms is driven by the aforementioned cache effects. Generally, these algorithms perform more steps of computation but also incur fewer cache faults in order to break down the problem into smaller subproblems. With computation performance improving much more quickly, it is intuitive that these multi-pivot schemes would, over time, gain an advantage over the classic one pivot algorithm. Thus, we believe that if the trend continues, it will become advantageous to perform more computation to use more pivots.

We present a multi-pivot quicksort variant that makes use of three pivots. We prove that our approach makes, on average, fewer comparisons ($1.84n\ln(n)$ vs $1.9n\ln(n)$) and more swaps than the dual pivot approach. However, in our experiments, the 3-pivot algorithm is about 7-8% faster than Yaroslavskiy's 2-pivot algorithm. Similar to Yaroslavskiy's quicksort, our algorithm performs much better in practice than the differences in comparisons and moves would predict. We present analyses of the cache behaviors of the various quicksort schemes. The results of our analyses give strong evidence that caching is in fact causing the performance differences observed.

With the increasing processor-memory performance gap in mind, we consider the technique of *presampling pivots*. This technique performs a significant amount of computation to precompute many pivots, with the goal of reducing cache faults. Our experiments show that, on modern architectures, this idea achieves a 1.5-2% improvement in performance.

## 2 Multi-Pivot Quicksort: 3-pivot

We introduce a variant of quicksort that makes use of three pivots $p < q < r$. At each iteration, the algorithm partitions the array around the three pivots into four subarrays and recursively sorts each of them. At first glance, this algorithm seems to be performing the same work as two levels of regular 1-pivot quicksort in one partition step. However, note that the middle pivot $q$ is of higher quality since it is a median of three pivots. This is the same as a regular quicksort that picks a median-of-3 pivot for every recursive call at alternating depths. Thus, we expect the performance of the 3-pivot variant to be between classic quicksort and classic quicksort using a median-of-3 pivot. Later, we shall see that it actually outperforms median-of-3 quicksort in practice by a significant margin.

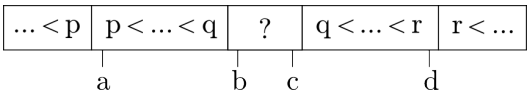


Figure 1: Invariant kept by the partition algorithm. All elements before pointer $b$ are less than $q$, all elements before pointer $a$ are less than $p$, all elements after pointer $c$ are greater than $q$, and all elements after pointer $d$ are greater than $r$. All other elements (between pointers $b$ and $c$ inclusive) have not yet been compared.

**2.1 Partition Algorithm** The partition algorithm uses four pointers: $a$, $b$, $c$, and $d$, which keep the invariant shown in Figure 1. Pointers $a$ and $b$ initially point to the first element of the array while $c$ and $d$ initially point to the last element of the array. The algorithm works by advancing $b$ and $c$ toward each other and moving each element they pass through into the correct subarray, terminating when $b$ and $c$ pass each other ($b > c$).

When $A[b] < q$, if $A[b] < p$, it swaps $A[b]$ and $A[a]$ and increments $a$ and $b$, or else does nothing and increments $b$. This case is symmetric to the case when $A[c] > q$. When $A[b] > q$ and $A[c] < q$ then the algorithm swaps both elements into place using one of the four cases ($A[b] < r$ and $A[c] > p$, etc.), then increments/decrements $a$, $b$, $c$, and $d$ accordingly. Refer to algorithm A.1.1 for pseudocode.

## 3 Analysis

In the next few subsections, we give analyses for the 3-pivot quicksort algorithm, as well as cache behavior analyses for 1- and 2-pivot quicksorts. We show that the 3-pivot algorithm makes, on average, fewer comparisons and cache misses than the 1- or 2-pivot algorithms.

**Assumptions for 3-pivot quicksort** Throughout the next few sections we make the following assumptions:

1. The input array is a random permutation of $1, \ldots, n$
2. The elements indexed at the first quartile, the median and the third quartile are chosen as the three pivots. On random permutations this is the same as choosing them at random. Hence each triplet appears with probablity $\binom{n}{3}^{-1}$.

Given these assumptions, the expected value (or cost) of each of the 3-pivot quantities being analysed can be


Copyright © 2014.
by the Society for Industrial and Applied Mathematics.

represented by the following recursive formula:

(3.1)

$$f_n = p_n + \frac{6}{n(n-1)(n-2)} \sum_{i=0}^{n-3} \sum_{j=i+1}^{n-2} \sum_{k=j+1}^{n-1} \Big( f_i + f_{j-i-1} + f_{k-j-1} + f_{n-k-1} \Big)$$
$$= p_n + \frac{12}{n(n-1)(n-2)} \sum_{i=0}^{n-3} (n-i-1)(n-i-2) f_i$$

where $f_n$ denotes the expected cost (or number of comparisons) and $p_n$ represents the expected partitioning cost of the property being analysed. The solutions to these recurrences can be found in Appendix A.2.

**Notation** In our analyses we shall use the following notation:

1. $C_p(n)$ – expected number of comparisons of the $p$-pivot quicksort algorithm sorting an array of $n$ elements
2. $S_p(n)$ – expected number of swaps of the $p$-pivot quicksort algorithm sorting an array of $n$ elements
3. $CM_p(n)$ – expected number of cache misses of the $p$-pivot quicksort algorithm sorting an array of $n$ elements
4. $SP_p(n)$ – expected number of recursive calls to a subproblem greater in size than a block in cache invoked by the $p$-pivot quicksort algorithm sorting an array of $n$ elements

### 3.1 Number of Comparisons

THEOREM 3.1. $C_3(n) = \frac{24}{13} n \ln n + O(n) \approx 1.846\, n \ln n + O(n)$

*Proof.* The algorithm chooses three pivots and sorts them. This costs $\frac{8}{3}$ comparisons on average. Let the three pivots chosen be $p$, $q$ and $r$ with $p < q < r$. It is easy to see that each element is compared exactly twice to determine its correct location. First with $q$ and depending upon the result of this comparison either with $p$ (if less) or $r$ (if greater). Thus the expected number of comparisons in a single partition step is given by $p_n = 2(n-3) + \frac{8}{3}$. Using the above value of $p_n$ and plugging it in equation (3.1) gives,

$$C_3(n) = f_n = \frac{24}{13} n \ln n + O(n)$$

The mathematical details are omitted here for brevity. Full details can be found in Appendix A.2. The same result can be derived using the ideas presented in the PhD thesis of Hennequin [2], who took a more general approach and showed that if the partitioning costs are of the form $p_n = \alpha n + O(1)$ then a 3-pivot quicksort would have a total cost of $\frac{12}{13}\alpha\, n \ln n + O(n)$.

This is a lower number of comparisons than both the 1-pivot algorithm ($2.0 n \ln n$) and the 2-pivot algorithm ($1.9 n \ln n$). This theoretical result is validated by our experiments as well. Figure 4 in §4 clearly shows that the 3-pivot variant makes much fewer comparisons than its 1- and 2-pivot counterparts. One more point to note here is that in Yaroslavskiy's 2-pivot partitioning algorithm, $p_n$ depends upon whether the algorithm compares with $p$ or $q$ first [9]. This is not the case in 3-pivot algorithm because of its symmetric nature.

Tan in his PhD thesis [8] had also analysed the number of comparisons for a 3-pivot quicksort variant. He had also obtained an expected count of $1.846\, n \ln n + O(n)$ for the number of comparisons. However, his algorithm made 3 passes of the input. First pass to partition about the middle pivot, then for the left pivot and finally for the right pivot. However, our algorithm saves on these multiple passes and hence makes fewer cache faults. This behavior is rigorously analysed in §3.3.

### 3.2 Number of Swaps

THEOREM 3.2. $S_3(n) = \frac{8}{13} n \ln n + O(n) \approx 0.615 n \ln n + O(n)$

*Proof.* The 3-pivot algorithm makes two kinds of swaps. Thus the partitioning process can be viewed as being composed of two parts. The first part partitions the elements about $q$ (the middle pivot). This step is the same as a 1-pivot partition. In the second part, the two parts obtained are further subdivided into two more partitions leading to a total of four partitions. However, the second part is different from the normal 1-pivot partitioning process. Here the partition is achieved only by the way of swaps. This process is detailed in Figure 2.

The algorithm maintains four pointers $a$, $b$, $c$ and $d$ as shown in Figure 1. The left pointer $a$ is incremented when an element is found to be less than $p$ in which case it is swapped to the location pointed to by $a$. Similar analysis holds for the rightmost pointer $d$. The swaps made in the second part can be given by $i + n - k$ where $i$ and $k$ are the final positions of the pivots $p$ and $r$. Hence, the total number of swaps in given by:

$$S_3(n) = i + n - k + \text{swaps made partitioning about } q$$

The swaps made during partitioning using single pivot was analysed by Sedgewick in 1977 [6] and their number


Copyright © 2014.
by the Society for Industrial and Applied Mathematics.

Downloaded 01/30/21 to 5.18.247.84. Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

Downloaded 01/30/21 to 5.18.247.84. Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

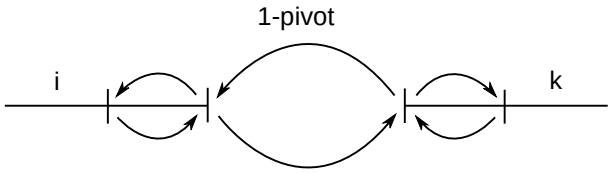


Figure 2: Swaps made in the partitioning process. Two types of swaps are made. The ones shown in bigger arrows are similar to the swaps made in the 1-pivot case. The ones shown in smaller arrows are made everytime an element is placed in the leftmost or rightmost buckets respectively.

is given by $\frac{n-2}{6}$. Hence the expected number of swaps in the partitioning process is given by:

$$p_n = \frac{n-2}{6} + \frac{6}{n(n-1)(n-2)} \sum_{i=0}^{n-3} \sum_{j=i+1}^{n-2} \sum_{k=j+1}^{n-1} i + n - k$$
$$= \frac{4n+1}{6}$$

Plugging the value of $p_n$ in Equation (3.1) and solving the recurrence gives the expected number of swaps for the 3-pivot quicksort as:

$$S_3(n) = \frac{8}{13} n \ln(n) + O(n)$$

This is greater than the number of swaps by the 1-pivot algorithm ($\frac{1}{3} n \ln(n)$ [6]) and the 2-pivot algorithm ($0.6n \ln(n)$ [9]) whereas 3-pivot makes $0.62n \ln(n)$ swaps.

**3.3 Cache Performance** We claimed in §1 that our 3-pivot algorithm has better cache performance than previous variants. First we provide an intuitive argument comparing with the 1-pivot algorithm. In one partition step of the 3-pivot algorithm, the array is split into four subarrays. Two pointers start at either end and stop when they meet each other. Thus these two pointers touch every page once. Assuming a perfect split, the other two pointers start at either end and scan one quarter of the array. They touch half of the pages in the array. Thus, assuming a perfect split, the 3-pivot algorithm incurs page faults equal to 1.5 times the number of pages. The 1-pivot partition algorithm touches every page in the subarray being sorted. In order for the 1-pivot algorithm to split the array into four subarrays, it must partition the array once, and the two subarrays each once. Thus it touches every page twice and incurs twice as many page faults as pages in the array. However, this performance is the *worst* case for the 3-pivot partition scheme. Thus, a 3-pivot algorithm intuitively incurs less cache faults.

Let $M$ denote the size of the cache, $B$ denote the size of a cache line. In this section, for simplicity, we will obtain upper bounds on $CM_p(n)$, cache misses of the $p$-pivot quicksort on $n$ elements, and $SP_p(n)$, number of recursive calls to a subproblem of size greater than block size by a $p$-pivot quicksort.

**1-pivot Quicksort** The upper bound for the 1-pivot case was obtained by LaMarca and Ladner [4]. They showed the following:

$$CM_1(n) \leq \frac{2(n+1)}{B} \ln\left(\frac{n+1}{M+2}\right) + O\left(\frac{n}{B}\right)$$

$$SP_1(n) \leq \frac{2(n+1)}{M+2} - 1$$

where $CM_{(1/3)}$ and $SP_{(1/3)}$ denote the same quantities for median-of-3 1-pivot quicksort.

THEOREM 3.3. *$CM_{(1/3)}(n) \leq \frac{12}{7}\left(\frac{n+1}{B}\right) \ln\left(\frac{n+1}{M+2}\right) + O\left(\frac{n}{B}\right)$ and $SP_{(1/3)}(n) \leq \frac{12}{7}\left(\frac{n+1}{M+2}\right) - 2 + O\left(\frac{1}{n}\right)$*

*Proof.* Refer to appendix A.3.

**2-pivot Quicksort**

THEOREM 3.4. *$CM_2(n) \leq \frac{8}{5}\left(\frac{n+1}{B}\right) \ln\left(\frac{n+1}{M+2}\right) + O\left(\frac{n}{B}\right)$ and $SP_2(n) \leq \frac{12}{10}\left(\frac{n+1}{M+2}\right) - \frac{1}{2} + O\left(\frac{1}{n^4}\right)$*

*Proof.* This algorithm uses three pointers to traverse through the array. Hence the total number of cache misses during partitioning will be at most the total number of elements scanned by these pointers divided by $B$. This gives rise to the following recurrence relations:

$$CM_2(n) \leq \frac{4n+1}{3B} + \frac{6}{n(n-1)} \sum_{i=0}^{n-2} (n-i-1) CM_2(i)$$

$$SP_2(n) \leq 1 + \frac{6}{n(n-1)} \sum_{i=0}^{n-2} (n-i-1) NS_2(i)$$

The recurrence for number of subproblems is self-explanatory. A minor point is that the above relation holds for $n > M$. For $n \leq M$, $CM_2(n) = 0$. Solving the above recurrences we get,

$$CM_2(n) \leq \frac{8}{5}\left(\frac{n+1}{B}\right) \ln\left(\frac{n+1}{M+2}\right) + O\left(\frac{n}{B}\right)$$

$$SP_2(n) \leq \frac{12}{10}\left(\frac{n+1}{M+2}\right) - \frac{1}{2} + O\left(\frac{1}{n^4}\right)$$



Copyright © 2014.
by the Society for Industrial and Applied Mathematics.

Downloaded 01/30/21 to 5.18.247.84. Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

### 3-pivot Quicksort

THEOREM 3.5. $CM_3(n) \leq \frac{18}{13}\left(\frac{n+1}{B}\right)\ln\left(\frac{n+1}{M+2}\right)+O\left(\frac{n}{B}\right)$ *and* $SP_3(n) \leq \frac{12}{13}\left(\frac{n+1}{M+2}\right)-\frac{1}{3}+O\left(\frac{1}{n}\right)$

*Proof.* This algorithm uses four pointers to traverse through the array. Hence the total number of cache misses during partitioning will be at most the total number of elements scanned by these pointer divided by $B$. Hence the partitioning costs for $CM_3(n)$ is given by, $\frac{3n+1}{2B}$ and for $SP_3(n)$ by 1. Solving we get,

$$CM_3(n) \leq \frac{18}{13}\left(\frac{n+1}{B}\right)\ln\left(\frac{n+1}{M+2}\right)+O\left(\frac{n}{B}\right)$$

$$SP_3(n) \leq \frac{12}{13}\left(\frac{n+1}{M+2}\right)-\frac{1}{3}+O\left(\frac{1}{n}\right)$$

One point to note is that we are overestimating (upper-bounding) the number of cache misses. This is because some of the elements of the left sub-problem might still be in the cache when the subproblem for that subarray is solved. But for the purposes of this analysis we have ignored these values. Additionally, these cache hits seem to affect only the linear term as was analysed by LaMarca and Ladner in [4]. Hence the asymptotic behaviour is still accurately approximated by these expressions. Note that 3-pivot quicksort algorithm has 50% and 25% less cache faults than 1- and 2-pivot algorithms, respectively.

## 4 Experiments

The goal for our experiments is to simplify the environment the code is running in by as much as possible to remove extraneous effects from the JVM. This way, it is simpler to identify key factors in the experimental results. As such, we wrote all tests in C.

We ran rigorous experiments comparing classic quicksort, Yaroslavskiy's 2-pivot variant, our 3-pivot variant, as well as optimized versions of them. Optimized 1-pivot quicksort picks a pivot as a median of three elements. Optimized 2-pivot quicksort picks two pivots as the second and fourth of five elements. Optimized 3-pivot quicksort picks three pivots as the second, fourth, and sixth of seven elements. In addition, all three switch to insertion sort at the best subproblem size determined experimentally for each. The unoptimized versions do none of these.

For the experiments shown, we ran each algorithm on arrays containing a random permutation of the 32-bit integers $1 \ldots n$, where $n$ is the size of the array. Tests on the smallest array sizes were averaged over thousands of trials, which is gradually reduced to 2-10 trials for the largest array sizes. All experiments were run on the machine specified in Table A.4.1.

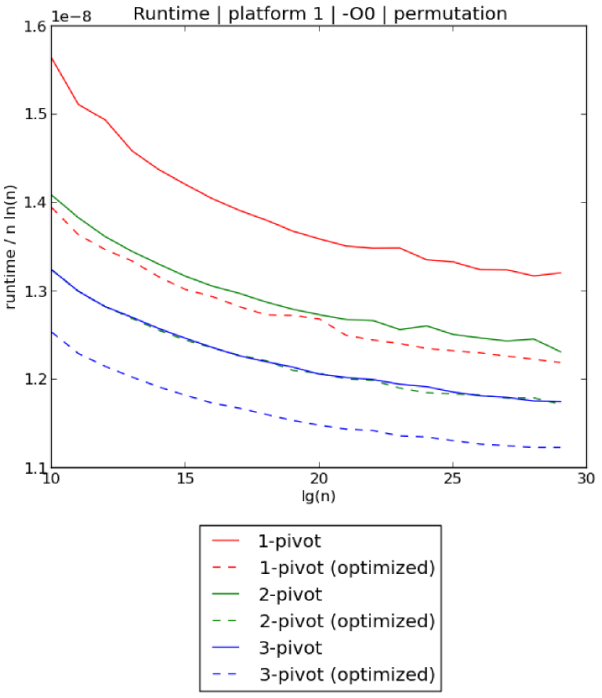


Figure 3: Plot of runtime against the size of array for the various quicksort algorithms. The size, $n$, is plotted logarithmically and the runtime is divided by $n \ln n$.

### 4.1 Runtime, Comparisons, and Assignments

Figure 3 shows the experiment in runtime. The unoptimized 3-pivot variant is faster than both the optimized and unoptimized versions of the 1-pivot and 2-pivot quicksort algorithms. Recall that 3-pivot quicksort is similar to a mix between 1-pivot quicksort and optimized 1-pivot quicksort, yet it significantly outperforms both of them. The graph also shows that the performance is consistent, doing as just as well for small numbers of elements as for large numbers of elements.

Figure 4 shows the experiment in comparisons. The graph confirms the results of our analysis. The 3-pivot version uses fewer comparisons than the 2-pivot version. Note here that the optimized 3-pivot algorithm uses more comparisons on small input sizes but still outperforms the others in runtime.

Swaps are implemented with three assignment operations using a temporary variable. In our implementations, multiple overlapping swaps are optimized to use fewer assignments. for example, $swap(a, b)$ followed by $swap(b, c)$ can be done with only four assignments. Thus, instead of counting swaps, we count the number


Copyright © 2014.
by the Society for Industrial and Applied Mathematics.

Downloaded 01/30/21 to 5.18.247.84. Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

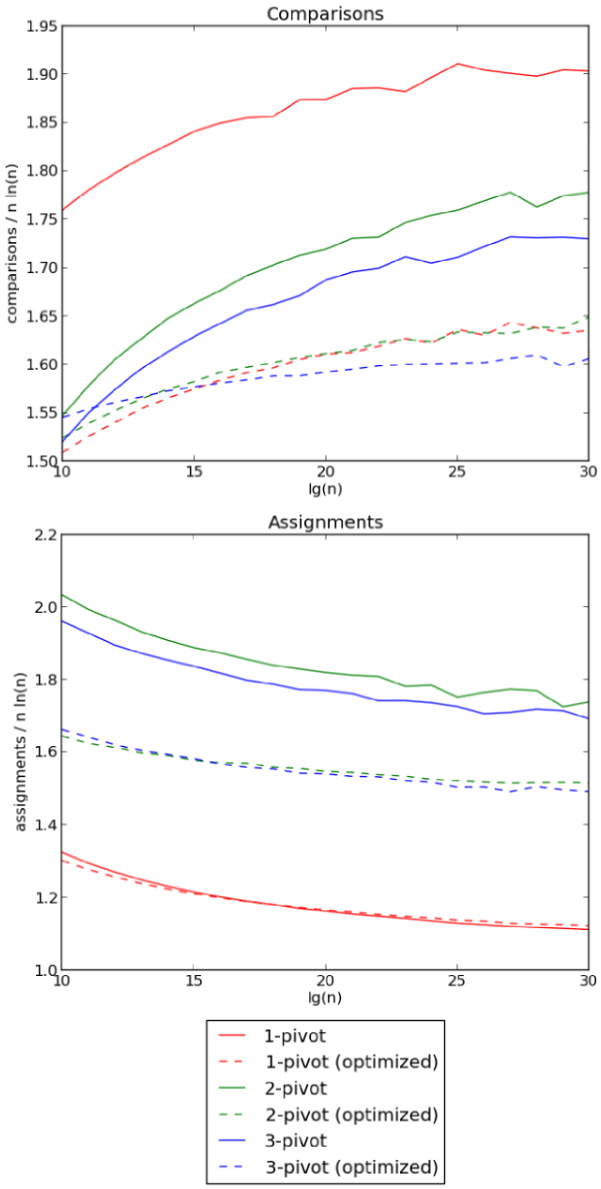


Figure 4: Plots of number of comparisons and assignments against the size of array for the various quicksort algorithms. The size, $n$, is plotted logarithmically and the comparisons/assignments are divided by $n \ln n$. Note that the graphs for number of comparisons appear to approach the correct coefficients calculated in the analysis.

of assignment operations done. Figure 4 shows these results. The classic 1-pivot algorithm uses far fewer assignments than the other variants. Our 3-pivot algorithm uses slightly fewer assignments than the 2-pivot algorithm. It is expected that the graphs look slightly different from our swap analysis.

**4.2 Comprehensive Tests** In addition to the simple tests shown, we also ran two sets of comprehensive tests. These tests were ran on two different platforms in order to highlight artifacts from differing computer architectures. The low level details of the platforms are described in Appendix A.4. The compiler used for all tests is:

gcc (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3

The first set of these tests evaluated runtime performance on different input distributions (see Appendix A.5). The different input types we considered are:

1. *Permutation:* A random permutation of the integers from 1 to $n$ (see Figure A.5.2 and Figure A.5.7)
2. *Random:* $n$ random elements were selected from $1 \ldots \sqrt{n}$ (see Figure A.5.3 and Figure A.5.8)
3. *Decreasing:* The integers $n$ to 1, in decreasing order (see Figure A.5.6 and Figure A.5.11)
4. *Increasing:* The integers 1 to $n$, in increasing order (see Figure A.5.5 and Figure A.5.10)
5. *Same:* $n$ equal integers (see Figure A.5.4 and Figure A.5.9)

The 3-pivot algorithm performs well in all the tests except for the "same" distribution on platform 1. Since this is not observed in platform 2, we conclude that artifacts due to architecture play a significant role in performance.

The second set of tests evaluated runtime performance under different GCC optimization levels (see Appendix A.6). The graphs show runtimes of the algorithms compiled with the -O0, -O1, -O2 and -O3 flags. We see that the results are much less uniform and are dependent on the platform and optimization flag. However, in most cases, the 3-pivot algorithm still outperforms the others. Using -O0, 3-pivot is faster on both platforms. Using -O1, 3-pivot is faster on one platform and only slightly slower than 2-pivot on the other. Using -O2 and -O3, the standard version of 3-pivot is faster than the standard version of 2-pivot quicksort, while the reverse is true for the optimized versions. Better understanding of the algorithms and fine-tuning them under compiler optimizations is an area which we mark for future work.


Copyright © 2014.
by the Society for Industrial and Applied Mathematics.

| Variant | Cache Misses | Comparisons | Swaps |
|---|---|---|---|
| 1-pivot | $2\left(\frac{n+1}{B}\right)\ln\frac{n+1}{M+2}$ | $2n\ln n$ | $0.333n\ln n$ |
| 1-pivot (median of 3) | $1.71\left(\frac{n+1}{B}\right)\ln\frac{n+1}{M+2}$ | $1.71n\ln n$ | $0.343n\ln n$ |
| 2-pivot (Yaroslavskiy) | $1.6\left(\frac{n+1}{B}\right)\ln\frac{n+1}{M+2}$ | $1.9n\ln n$ | $0.6n\ln n$ |
| 3-pivot | $1.38\left(\frac{n+1}{B}\right)\ln\frac{n+1}{M+2}$ | $1.85n\ln n$ | $0.62n\ln n$ |

Table 1: Summary of previous results [6, 9] and results of our analysis. Each value has lower order terms that have been omitted for conciseness.

**4.3 Other Experiments** Other multi-pivot algorithms are also of interest. In particular, we also ran tests on a 7-pivot approach. However, these tests concluded that the 7-pivot algorithm runs more slowly than the 2- and 3-pivot variants.

Another feature of consequence is the behavior of algorithms under a multi-core architecture. Thus we performed a set of tests on these versions of quicksort running on four threads on a machine with four cores. The scheme we used to split work is as follows: Use a very large sample to perform a 4-way/3-pivot partition of the array into four subarrays of (probably) very similar sizes. Then run an instance of a quicksort algorithm on each of the four subarrays. One fact to note here is that under this scheme, the runtime of the entire algorithm is the *max* of the runtimes of the four instances. Thus, a fast algorithm with high variance in runtime may actually perform worse than a slower algorithm that has a consistent runtime. Our tests concluded that all three of the 1-, 2-, and 3-pivot approaches showed comparable speedups (about three times faster than single threaded) when run under these conditions.

## 5 Theory and Experiments: Explained

The dramatic speed increase of Yaroslavskiy's algorithm is not fully explained by previous analyses of the number of comparisons and swaps. We see a 7-9% increase in performance but the average number of comparisons is only 5% less, and almost twice as many swaps! This disparity between theory and practice is highlighted even more clearly with the results of our 3-pivot algorithm (refer Table 1). Our algorithm uses *more* comparisons and swaps than the median-of-3 quicksort algorithm yet we see about a 7% *reduction* in runtime.

After analysing the cache performance of each of the algorithms, we can finally explain the disparity we see between theory and practice. Even though our algorithm uses more swaps and comparisons than median-of-3 quicksort, it make almost 20% fewer cache misses. This explains why our algorithm performs better even though traditional analyses say that it should do much worse. It also explains why we see such a speed increase for Yaroslavskiy's dual-pivot quicksort.

## 6 Further Cache Improvements

With the insights from caching in modern architectures, we design a modification based on the idea of *presampling pivots*. Given an initial unsorted sequence $a_1, \ldots, a_n$ of size $n$, the main ideas of this algorithm can be summarised as follows:

1. Sample $\sqrt{n}$ elements to be used as pivots for the partitioning process and sort them. This is done just once at the start and not for each recursive call.

2. For every recursive call, instead of choosing a pivot from the subarray, choose an appropriate element from the above array as a pivot. Partition the array about the chosen pivot.

3. Once we run out of pivots, fall back to the standard quicksort algorithm (1-pivot, 2-pivot, etc. as the case may be).

This strategy has some nice properties. By choosing pivots out of a sample of $\sqrt{n}$ elements, the initial pivots are extremely good with very high probability. Hence, we expect that using presampling would bring down the number of subproblems below the cache size more quickly.

We implemented this approach and carried out some experiments, the details of which have been omitted due to space constraints. In practice, it leads to about a $1.5 - 2\%$ gain in performance when comparing the running times of the standard 1-pivot quicksort against that of 1-pivot quicksort with presampling. For larger array sizes, the presampled version was on average about 2% faster than the standard version. Similar results were obtained when comparing the presampled and standard versions of 2-pivot quicksort.

We believe that fine tuning this approach further, such as varying the sample size and choosing when to fall back to the standard algorithm, would lead to even


Copyright © 2014.
by the Society for Industrial and Applied Mathematics.

Downloaded 01/30/21 to 5.18.247.84. Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

more gains in performance. Analysing this approach mathematically is another avenue which needs more investigation. We mark these as areas for future work.

## 7 Conclusions and Future Work

First, we have confirmed previous experimental results on Yaroslavskiy's dual-pivot algorithm under a basic environment thus showing that the improvements are not due to JVM side effects. We designed and analysed a 3-pivot approach to quicksort which yielded better results both in theory and in practice. Furthermore, we provided strong evidence that much of the runtime improvements are from cache effects in modern architecture by analysing cache behavior.

We have learned that due to the rapid development of hardware, many of the results from more than a decade ago no longer hold. Further work in the short term can be directed at discovering, analysing, and implementing more interesting multi-pivot quicksort schemes.

## References


[1] Martin Aumüller and Martin Dietzfelbinger. Optimal partitioning for dual pivot quicksort. *CoRR*, abs/1303.5217, 2013.

[2] Pascal Hennequin. Combinatorial analysis of quicksort algorithm. *Informatique théorique et applications*, 23(3):317–333, 1989.

[3] C. A. R. Hoare. Quicksort. *Comput. J.*, 5(1):10–15, 1962.

[4] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of sorting. *J. Algorithms*, 31(1):66–104, 1999.

[5] D.A. Patterson and J.L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.

[6] Robert Sedgewick. The analysis of quicksort programs. *Acta Inf.*, 7:327–355, 1977.

[7] Robert Sedgewick and Jon Bentley. Quicksort is optimal. `http://www.cs.princeton.edu/~rs/talks/QuicksortIsOptimal.pdf`, 2002. [Online; accessed 21-April-2013].

[8] Kok-Hooi Tan. An asymptotic analysis of the number of comparisons in multipartition quicksort. 1993.

[9] Sebastian Wild and Markus E. Nebel. Average case analysis of java 7's dual pivot quicksort. In Leah Epstein and Paolo Ferragina, editors, *ESA*, volume 7501 of *Lecture Notes in Computer Science*, pages 825–836. Springer, 2012.

[10] Vladimir Yaroslavskiy. Dual-pivot quicksort. `http://iaroslavski.narod.ru/quicksort/DualPivotQuicksort.pdf`, 2009. [Online; accessed 21-April-2013].


## A Appendix

### A.1 Partition Algorithm

**Algorithm A.1.1** 3-Pivot Partition

**Require:** $A[left] < A[left+1] < A[right]$ are the three pivots

1: **function** PARTITION3($A$, $left$, $right$)
2: $a \leftarrow left + 2$, $b \leftarrow left + 2$
3: $c \leftarrow right - 1$, $d \leftarrow right - 1$
4: $p \leftarrow A[left]$, $q \leftarrow A[left + 1]$, $r \leftarrow A[right]$
5: **while** $b \leq c$ **do**
6: **while** $A[b] < q$ and $b \leq c$ **do**
7: **if** $A[b] < p$ **then**
8: SWAP($A[a]$, $A[b]$)
9: $a \leftarrow a + 1$
10: **end if**
11: $b \leftarrow b + 1$
12: **end while**
13: **while** $A[c] > q$ and $b \leq c$ **do**
14: **if** $A[c] > r$ **then**
15: SWAP($A[c]$, $A[d]$)
16: $d \leftarrow d - 1$
17: **end if**
18: $c \leftarrow c - 1$
19: **end while**
20: **if** $b \leq c$ **then**
21: **if** $A[b] > r$ **then**
22: **if** $A[c] < p$ **then**
23: SWAP($A[b]$,$A[a]$), SWAP($A[a]$,$A[c]$)
24: $a \leftarrow a + 1$
25: **else**
26: SWAP($A[b]$,$A[c]$)
27: **end if**
28: SWAP($A[c]$,$A[d]$)
29: $b \leftarrow b + 1$, $c \leftarrow c - 1$, $d \leftarrow d - 1$
30: **else**
31: **if** $A[c] < p$ **then**
32: SWAP($A[b]$,$A[a]$), SWAP($A[a]$,$A[c]$)
33: $a \leftarrow a + 1$
34: **else**
35: SWAP($A[b]$,$A[c]$)
36: **end if**
37: $b \leftarrow b + 1$, $c \leftarrow c - 1$
38: **end if**
39: **end if**
40: **end while**
41: $a \leftarrow a - 1$, $b \leftarrow b - 1$, $c \leftarrow c + 1$, $d \leftarrow d + 1$
42: SWAP($A[left + 1]$,$A[a]$), SWAP($A[a]$,$A[b]$)
43: $a \leftarrow a - 1$
44: SWAP($A[left]$,$A[a]$), SWAP($A[right]$,$A[d]$)
45: **end function**



Copyright © 2014.
by the Society for Industrial and Applied Mathematics.

Downloaded 01/30/21 to 5.18.247.84. Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

Downloaded 01/30/21 to 5.18.247.84. Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

### A.2 Solving Recurrences for 3-pivot quicksort

All the quantities analysed in this paper satisfy a recurrence relation of the following form

$$f_n = an+b+\frac{12}{n(n-1)(n-2)}\sum_{i=0}^{n-3}(n-i-1)(n-i-2)f_i$$

Multiplying by $n(n-1)(n-2)$ throughout gives:

$$\begin{aligned} n(n-1)(n-2)f_n = {} & an^2(n-1)(n-2) \\ & + bn(n-1)(n-2) \\ & + 12\sum_{i=0}^{n-3}(n-i-1)(n-i-2)f_i \end{aligned}$$

Substituting $n-1$ in the above equation and then subtracting gives:

$$\begin{aligned} (n-1)(n-2)(n-3)f_{n-1} = {} & \\ & a(n-1)^2(n-2)(n-3) \\ & + b(n-1)(n-2)(n-3) \\ & + 12\sum_{i=0}^{n-4}(n-i-1)(n-i-2)f_i \end{aligned}$$

$$\begin{aligned} n(n-1)(n-2)f_n = {} & (n-1)(n-2)(n-3)f_{n-1} \\ & + a(n-1)(n-2)(4n-3) \\ & + 3b(n-1)(n-2) \\ & + 24\sum_{i=0}^{n-3}(n-i-2)f_i \end{aligned}$$

The idea is to get rid of the summation by subtracting equations. Repeating the process twice on the above equation gives the following equation:

(A.1)

$$\begin{aligned} n(n-1)(n-2)f_n = {} & 3(n-1)(n-2)(n-3)f_{n-1} \\ & - 3(n-2)(n-3)(n-4)f_{n-2} \\ & + (n-3)(n-4)(n-5)f_{n-3} \\ & + 24f_{n-3} + 6a(4n-9) + 6b \end{aligned}$$

We use standard linear algebra software to solve this recursive equation giving it the appropriate initial conditions. All the equations analysed in this paper have the above form. Only the value of $a$ and $b$ changes. For the case of comparisons $a=2$ and $b=\frac{-10}{3}$. For the analysis of swaps $a=\frac{2}{3}$ and $b=\frac{1}{6}$. Similarly for other cases. We will show the detailed solution for the analysis of comparisons. Other analysis are very similar. The solution to (A.1) for comparisons is of the form:

$$C_3(n) = \frac{24}{13}(n+1)H_n - \frac{311}{117} + \frac{190}{117} + G(n)$$

where $H_n$ is the harmonic function, $G(n)$ is a large expression on $n$ output by our recurrence solver which contains complex numbers and the gamma function. Hence the analysis of $G(n)$ is very important. We first prove that the $G(n)$ is indeed real and that it is of $O(\frac{1}{n})$.

Define $d=\frac{5}{2}+\frac{1}{2}i\sqrt{23}$, $z=10097+i1039\sqrt{23}$. Then $G(n)$ for the analysis of comparisons is:

$$\begin{aligned} & G(n) = \\ & -\frac{1}{34983\pi\Gamma(n+1)}\left\{\cosh(\frac{1}{2}\pi\sqrt{23})\Gamma(n-d)\Gamma(d)z \right. \\ & \left. + \frac{48\pi\Gamma(n-\overline{d})\overline{z}}{\Gamma(d)}\right\} - \frac{10097}{34983}(n+1) \end{aligned}$$

Using the properties of gamma function, $\Gamma(n-d) = -d(-d+1)(-d+2)(-d+3)\cdots(-d+n-1)\Gamma(-d)$. Hence, we get the following equations:

$$\Gamma(n-d) = z_1\Gamma(-d)$$

$$\Gamma(n-\bar{d}) = \bar{z_1}\Gamma(\bar{d})$$

where $z_1 = -d(-d+1)\cdots(-d+n-1)$. Substituting these values in the above equation, we get:

$$\begin{aligned} & G(n) = \\ & -\frac{1}{34983\pi\Gamma(n+1)}\left\{\cosh(\frac{1}{2}\pi\sqrt{23})\Gamma(-d)\Gamma(d)zz_1 \right. \\ & \left. + \frac{48\pi\Gamma(\overline{d})\overline{zz_1}}{\Gamma(d)}\right\} - \frac{10097}{34983}(n+1) \end{aligned}$$

Now using the properties of gamma function we get:

$$\begin{aligned} \Gamma(-d)\Gamma(d) = \frac{-\pi}{d\sin(\pi d)} &= \frac{-\pi}{d\cos(i\frac{\pi}{2}\sqrt{23})} \\ &= \frac{-\pi}{d\cosh(\frac{\pi}{2}\sqrt{23})} \end{aligned}$$

or

$$\cosh(\frac{\pi}{2}\sqrt{23})\Gamma(-d)\Gamma(d) = \frac{-\pi}{d}$$

$$\frac{48\Gamma(\bar{d})}{\Gamma(d)} = \frac{-\pi}{\overline{d}}$$

Substituting these values in the above equation we get:

$$\begin{aligned} G(n) = & -\frac{1}{34983\pi\Gamma(n+1)}\left\{-\frac{\pi zz_1}{d} - \frac{\overline{zz_1}\pi}{\overline{d}}\right\} \\ & - \frac{10097}{34983}(n+1) \\ = & \frac{2}{34983\Gamma(n+1)}Re\Big(\frac{zz_1}{d}\Big) - \frac{10097}{34983}(n+1) \\ = & O\Big(\frac{1}{n}\Big) - \frac{10097}{34983}(n+1) \end{aligned}$$



Copyright © 2014.
by the Society for Industrial and Applied Mathematics.

Downloaded 01/30/21 to 5.18.247.84. Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

Equation (A.1) hence solves to:

$$\begin{aligned}C_3(n) =& \frac{24}{13}(n+1)\ln(n) \\ &+ \left\{-\frac{311}{117} + \frac{24}{13}\gamma - \frac{10097}{34983}\right\}(n+1) \\ &+ \frac{190}{117} + O\Big(\frac{1}{n}\Big) \\ \approx& \frac{24}{13}(n+1)\ln(n) - 1.88(n+1) + \frac{190}{117} + O\Big(\frac{1}{n}\Big)\end{aligned}$$

Here, we have shown the exact derivations for the number of comparisons. The analysis for number of swaps and cache misses are very similar to the above analysis and hence they have been omitted.

**A.3 Solving recurrences for median-of-3 1-pivot quicksort** This algorithm uses two poniters to traverse through the array. Hence the total number of cache misses during partitioning will be at most the total number of elements scanned by these pointers divided by $B$. This gives rise to the following recurrence relations:

$$\begin{aligned}CM_{(1/3)}(n) \leq & \frac{n}{B} + \frac{12}{n(n-1)(n-2)} \sum_{i=0}^{n-1} \Big( (n-i)(i-1) \\ & (CM_{(1/3)}(i) + CM_{(1/3)}(n-i)) \Big) \\ \leq & \frac{n}{B} + \frac{12}{n(n-1)(n-2)} \sum_{i=0}^{n-1} i(n-i-1) CM_{(1/3)}(i)\end{aligned}$$

$$\begin{aligned}SP_{(1/3)}(n) \leq & \\ & 1 + \frac{12}{n(n-1)(n-2)} \sum_{i=0}^{n-1} i(n-i-1) SP_{(1/3)}(i)\end{aligned}$$

The recurrence for number of subproblems is self-explanatory. A minor point is that the above relation holds for $n > M$. For $n \leq M$, $CM_{(1/3)}(n) = 0$. Both of the above recurrence relations can be written in a more general form as,

$$f_n = an + b + \frac{6}{n(n-1)(n-2)} \sum_{i=1}^{n-1} (i-1)(n-i) f_i$$

$$f_n = an + b + \frac{12}{n(n-1)(n-2)} \sum_{i=0}^{n-1} i(n-i-1) f_i$$

where $a = \frac{1}{B}$ and $b = 0$ for the 1st recurrence and $a = 0$ and $b = 1$ for the second one. Multiplying by $n(n-1)(n-2)$ throughout gives:

$$\begin{aligned}n(n-1)(n-2) f_n = & \, an^2(n-1)(n-2) \\ & + bn(n-1)(n-2) + 12 \sum_{i=0}^{n-1} i(n-i-1) f_i\end{aligned}$$

Substituting $n-1$ in the above equation and then subtracting gives:

$$\begin{aligned}(n-1)(n-2)(n-3) f_{n-1} = & \, a(n-1)^2(n-2)(n-3) \\ & + b(n-1)(n-2)(n-3) \\ & + 12 \sum_{i=0}^{n-2} i(n-i-2) f_i\end{aligned}$$

$$\begin{aligned}n(n-1)(n-2) f_n = & \, (n-1)(n-2)(n-3) f_{n-1} \\ & + a(n-1)(n-2)(4n-3) \\ & + 3b(n-1)(n-2) + 12 \sum_{i=0}^{n-1} i f_i\end{aligned}$$

The idea is to get rid of the summation by subtracting equations. Repeating the process gives the following equation:

$$\begin{aligned}n(n-1)(n-2) f_n = & \, 3(n-1)(n-2)(n-3) f_{n-1} \\ & - 3(n-2)(n-3)(n-4) f_{n-2} \\ & + (n-3)(n-4)(n-5) f_{n-3} \\ & + 24 f_{n-3} + 6a(4n-9) + 6b\end{aligned}$$

Substituting values for $a$ and $b$ and solving the above recurrence using standard linear algebra software we get:

$$CM_{(1/3)}(n) \leq \frac{12}{7}\left(\frac{n+1}{B}\right)\ln\left(\frac{n+1}{M+2}\right) + O\left(\frac{n}{B}\right)$$

$$SP_{(1/3)}(n) \leq \frac{12}{7}\left(\frac{n+1}{M+2}\right) - 2 + O\left(\frac{1}{n}\right)$$

The analysis for 2-pivot quicksort is also very similar to the one shown above. We do not show it here.

**A.4 Test Machine Specifications**

| | |
|---|---|
| CPU | Intel(R) Core(TM)2 Quad CPU Q9650 @ 3.00 GHz |
| L1d cache | d: 32K i: 32K |
| L2 cache | 6144K |
| Cache line size | 64 |
| Memory | 4 × 2048 MB DDR @ 800 MHz |
| OS Kernel | GNU/Linux 3.2.0-53-generic |

Table A.4.1: Specifications of the system the tests were run on (**platform 1**)



Copyright © 2014.
by the Society for Industrial and Applied Mathematics.

Downloaded 01/30/21 to 5.18.247.84. Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

| | |
|---|---|
| CPU | Intel(R) Core(TM) i5-3570K CPU @ 3.40GHz |
| L1d cache | d: 32K i: 32K |
| L2 cache | 256K |
| L3 cache | 6144K |
| Cache line size | 64 |
| Memory | 2 × 8192 MB DDR3 @ 1333 MHz |
| OS Kernel | GNU/Linux 3.8.0-31-generic |

Table A.4.2: Specifications of the system the additional comprehensive tests were run on (**platform 2**)

**A.5 Experiments by Distribution Type** These tests were run on the distribution types outlined in §4.2, on platform 1 and 2 using -O0 optimization flag.

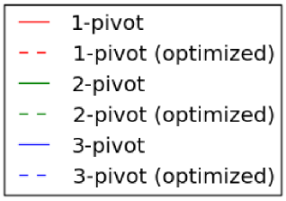


Figure A.5.1: Legend for the following graphs

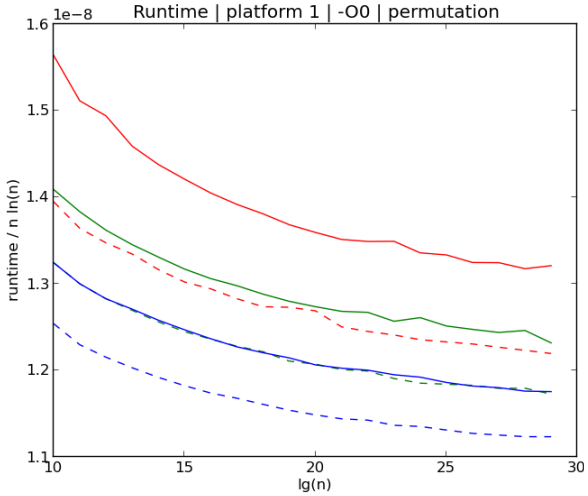


Figure A.5.2

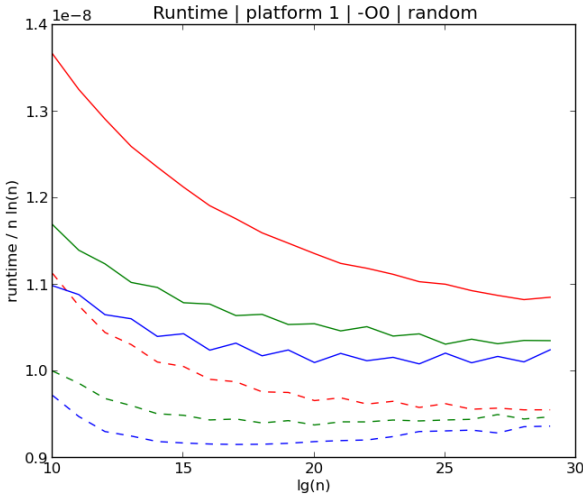


Figure A.5.3

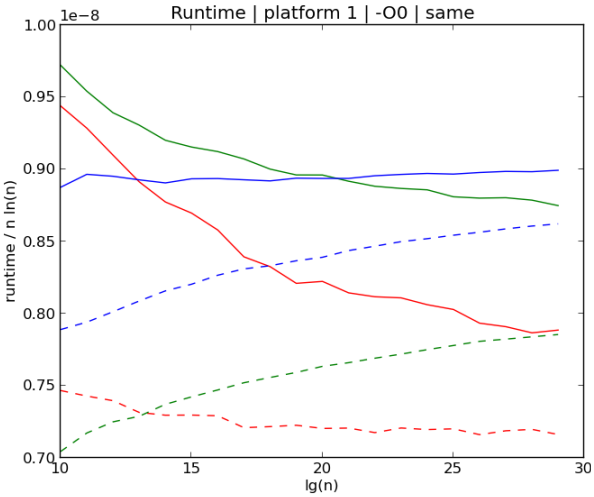


Figure A.5.4

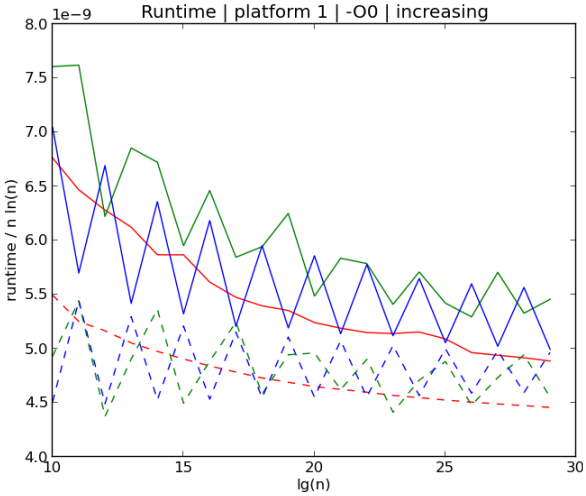


Figure A.5.5



Copyright © 2014.
by the Society for Industrial and Applied Mathematics.

Downloaded 01/30/21 to 5.18.247.84. Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

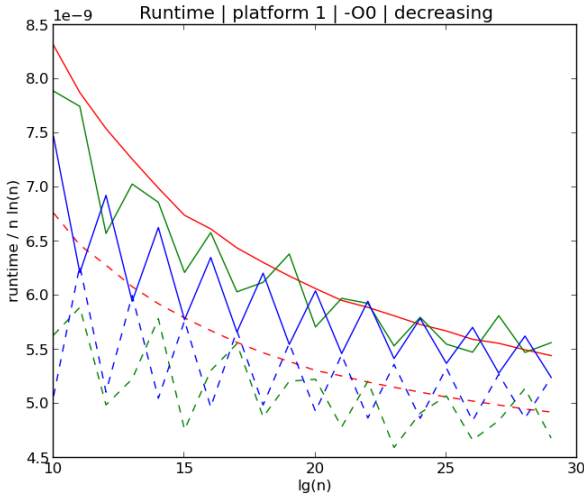


Figure A.5.6

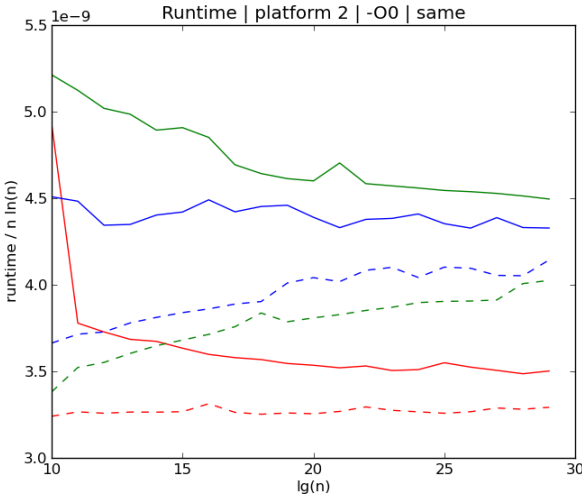


Figure A.5.9

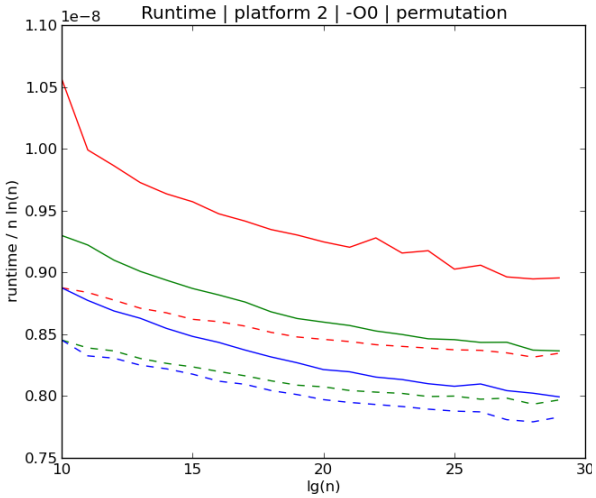


Figure A.5.7

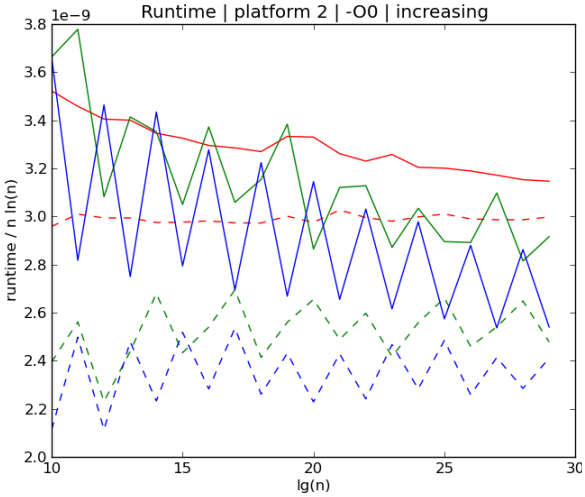


Figure A.5.10

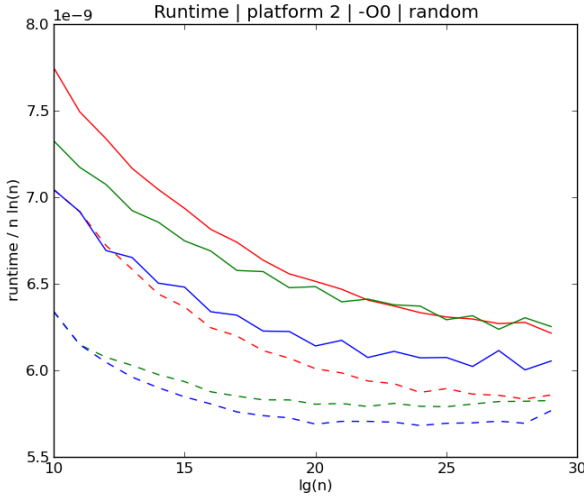


Figure A.5.8

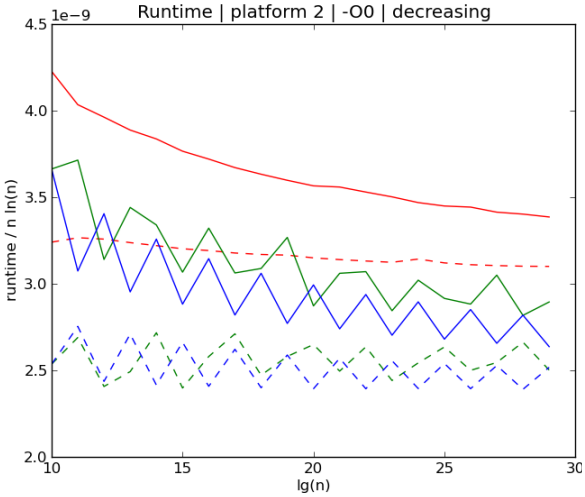


Figure A.5.11



Copyright © 2014.
by the Society for Industrial and Applied Mathematics.

### A.6 Experiments by Compiler Optimization

These tests were run on the permutation distribution, on platform 1 and 2 using -O0, -O1, -O2, and -O3 optimization flags.

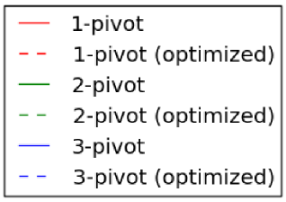


Figure A.6.1: Legend for the following graphs

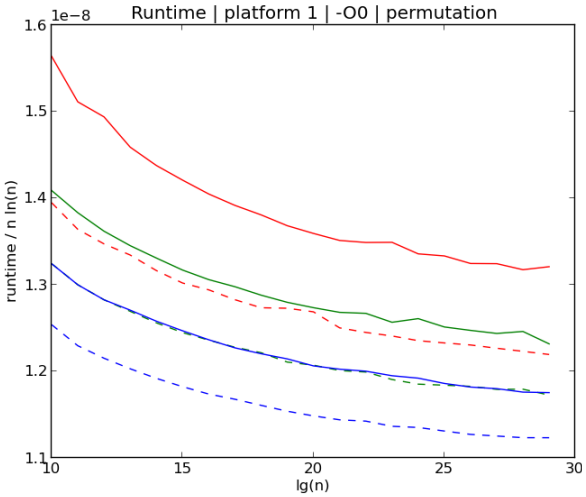


Figure A.6.2

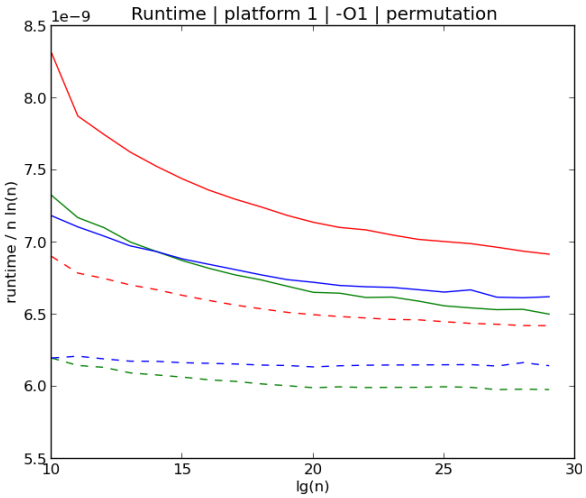


Figure A.6.3

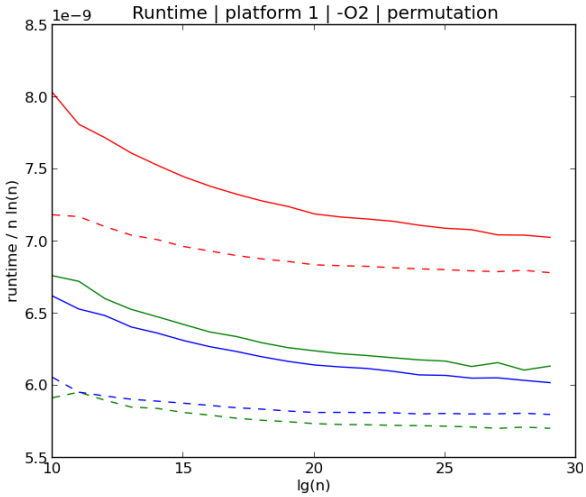


Figure A.6.4

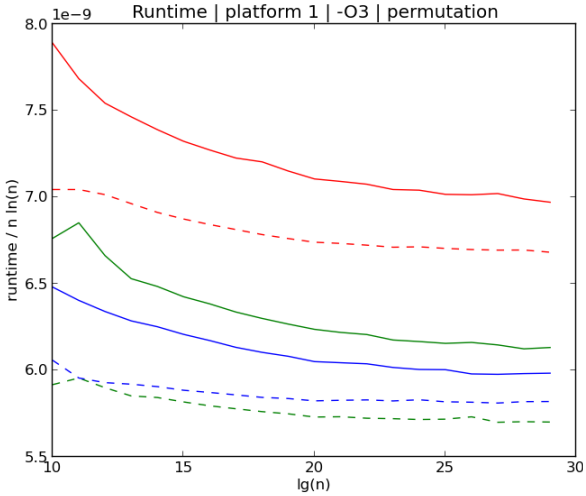


Figure A.6.5

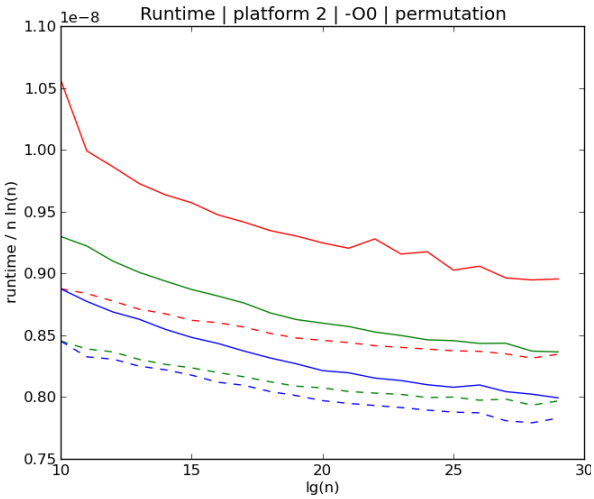


Figure A.6.6



Copyright © 2014.
by the Society for Industrial and Applied Mathematics.

Downloaded 01/30/21 to 5.18.247.84. Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

Downloaded 01/30/21 to 5.18.247.84. Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/page/terms

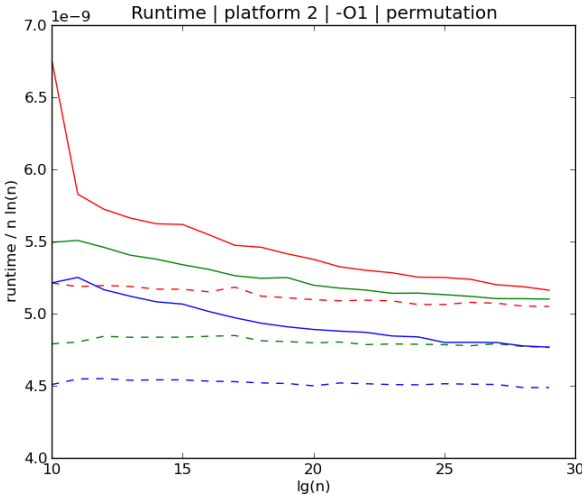


Figure A.6.7

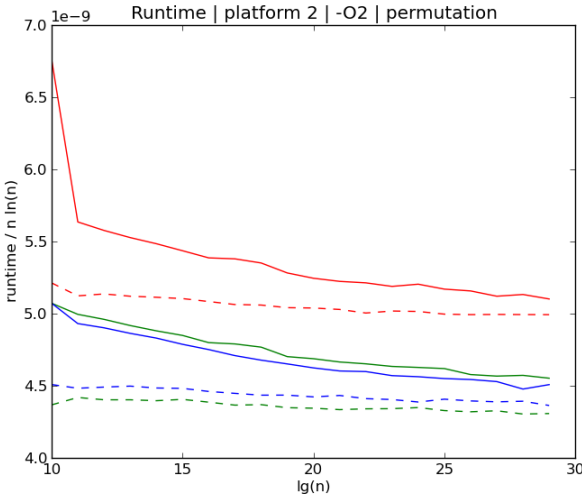


Figure A.6.8

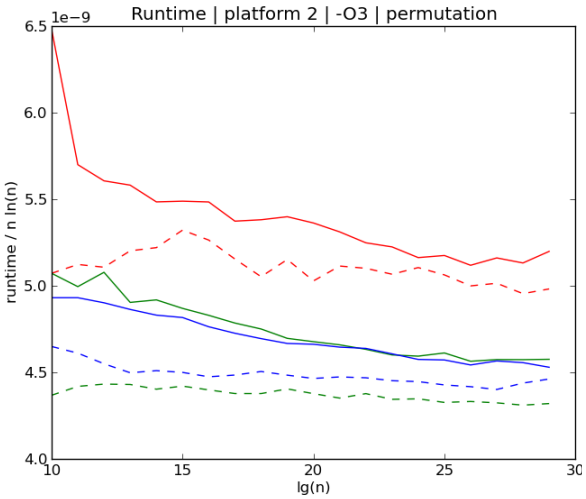


Figure A.6.9



Copyright © 2014.
by the Society for Industrial and Applied Mathematics.