Final Paper

An Investigation into Multi-Pivot Quicksort Algorithms

Authors:
Paymahn Moghadasian [umpaymah@myumanitoba.ca]

Joshua Hernandez [umhern23@myumanitoba.ca]

Instructor: Dr. S. Durocher

COMP 4420
Advanced Design and Analysis of Algorithms

Due : April 9, 2014

Contents

Sort Method	Space	Average Case Time	Worst Case Time
Selection	$O(1)$	$O(n^2)$	$O(n^2)$
Insertion	$O(1)$	$O(n^2)$	$O(n^2)$
Merge	$O(n)$	$O(n\log(n))$	$O(n\log(n))$
Quicksort	$O(1)$	$O(n\log(n))$	$O(n^2)$
Radix	$O(n)$	$O(n*k)$	$O(n*k)$
Counting	$O(m)$	$O(n+m)$	$O(n+m)$

Table 1: Summary of space and time complexities of various sorts where n represents the number of elements, k represents the number of digits in the largest value and m represents the maximum value to be sorted.

1 Abstract

The quicksort is a thoroughly studied sorting algorithm and is commonly among the first efficient sorts learned by students of computer science. Many variants of the quicksort have been proposed, from the classic quicksort introduced by Tony Hoare in 1961 [1] to Yaroslavskiy's dual pivot quicksort introduced in 2009 and used by the Java 7 Standard Library [2]. Since Hoare's first proposal, much research has gone into attempting to minimize the total number of swaps done by the sort, the total number of comparisons done by the sort and minimizing the worst case runtime. We aim to experimentally validate the swap and comparison count of several variants of the quicksort and compare the runtimes and various optimizations. Our results SUMMARIZE THE RESULTS

2 Introduction

Sorting is a fundamental concept of computer science wherein a totally ordered multiset is modified such that the elements of the multiset are rearranged (permuted) in either non-decreasing or non-increasing order. A broad range of applications benefit from sorting such as organizing an MP3 library by song title to quickly identifying duplicates in a list to more advanced applications such as load balancing, data compression and computer graphics [3]. It is well known that all comparison based sorting algorithms are lower bound by $\Omega(nlogn)$ comparisons [4] and quicksort is no exception to this rule. Interestingly, there are non-comparison based sorts such as the counting sort and the radix sort which take advantage of certain properties of the data set and get around the lower bound of comparison base sorts. A summary of space and time complexities can be found in Table 1.

The quicksort was first introduced by Tony Hoare in [1] and was quickly adopted by the field as a standard sorting algorithm alongside the mergesort [5] and several other sorting algorithms. Quicksorts simplicity, high average case performance, poor

worst case performance and low use of additional memory has made it one of the most highly studied "efficient" sorting algorithms. van Emden [6] proposed an optimization to the quicksort in 1970 which resulted in a 15% improvement on the efficiency of the quicksort. In 1971 Hoare and Foley [7] provided a formal proof of other correctness of the quicksort. In 1977, Sedgewick did a thorough analysis of quicksort [8] and then in 1978 Sedgewick again published a paper on the quicksort exploring various implementations and proposing several optimizations [9]. The academic interest in the quicksort has not slowed since the late 1970s; papers regarding the quicksort continue to be published for adapting the quicksort to GPUs [10], providing optimal pivot selection [11] and using more than one pivot [12].

In this paper we explore, compare and contrast the classic implementation of the quicksort against various multipivot quicksort implementations and their respective optimizations. We are particularly interested in comparing the classic quicksort against the standard dual pivot quicksort, Yaroslavskiy's dual pivot quicksort[13], Aumüller and Dietzfelbinger's optimized dual-pivot quicksort [14], Edmonson's M-Pivot quicksort [12] and the three pivot quicksort introduced by Kushagra et al [2]. We will first introduce the various quicksorts by providing a brief overview of their behaviour, implementation and potential optimizations. Next we will discuss the experiments and provide their results. Following this we provide an analysis, discuss future work and conclude the paper.

3 Quicksort

The following descriptions will assume that the algorithms are being used to sort a list A in non-decreasing order where n denotes the number of elements in A. Additionally, all algorithms are assumed to perform the sort in place.

3.1 Classic Quicksort

The classic quicksort, first introduced by Hoare [1] in 1962 is a recursive sorting algorithm which falls into the category of divide and conquer algorithms. All of following variations of the quicksort have these qualities as well. Quicksort can be summarized as follows:

1. Select a pivot in the current range of elements to be sorted.
2. Partition the list such that all elements less than the pivot appear to the left of the pivot while all elements greater than or equal to the pivot appear to the right of the pivot.
3. Recursively call quicksort on the left and right halves of the currently partitioned range.

This naïve implementation does not handle it's worst case gracefully. The classic quicksort degrades to a time complexity of $O\left(n^2\right)$ when the data being processed is in nearly sorted or already sorted order. Due to the simplicity of this implementation there are some minor changes which can be made to speed up the average case and aid the worst case runtimes. It was shown by Cook and Kim [15] that insertion sorts are the fastest sort for small lists and nearly sorted lists. This optimization does not make the quicksort asymptotically faster than it's naï variation in either the average or worst case but does provide a meaningful speedup. Cook and Kim also showed that a hybrid of insertion sort and quicksort outperforms either of the sorts used individually. A perfectly selected pivot (the median of the range) would ensure that the worst case space complexity is bound by $O\left(n\log(n)\right)$. Perfect pivot selection cannot be achieved with an increase in time complexity; however, selecting pivot to be the median of first, middle and last elements is a good enough estimation of the true median of the range [9].

3.2 Dual Pivot Quicksort

The dual pivot quicksort is a simplified version of Yarslavskiy's quicksort and can be though of as the classic quicksort with an extra pivot. Due to the extra pivot, this quicksort uses Yaroslavskiy's partitioning algorithm. The dual pivot quicksort suffers from the same worst-case space complexity as the classic quicksort and can be similarly optimized. Performing an insertion sort for small subarrays and selection the pivot intelligently can both provide meaningful and measurable improvements. Similar to the classic quicksort, the best pivots to select are the first and third quartiles. These can be estimated by selecting the first and third quartiles of a selection of five evenly spaced values.

3.3 Yaroslavskiy Quicksort

Yaroslavskiy wished to make further optimizations to the dual pivot quicksort. Yaroslavskiy partition algorithm, even though based of the same abstract algorithmic idea produces different results [16]. The algorithm uses a five element sorting network to pick the two best elements then partitions intelligently by picking the "better" comparison first [16].

3.4 Optimal Dual Pivot Quicksort

To further build on on the dual pivot quicksort and the Yaroslavskiy quicksort [14]. The pivots are selected as one sees fit, but when partitioning we keep track of how many elements that have been placed into the small list and the big list, denote these N_s and N_l respectively. This idea is very similar to Yaroslavskiy's quicksort algorithm. Note that there are three category of elements in the dual pivot quicksort.

When $N_s > N_l$ during the partition process, you first compare the the small pivot, otherwise compare to the large pivot. Although this is a simple optimization, the paper expends most of it's effort in proving that is the case[14]. With this small optimization the number of comparisons is drastically decreases relative to most quicksort implementations.

3.5 Three Pivot Quicksort

With the optimizations introduced from the Yaroslavskiy's quicksort algorithm and the optimal dual pivot quicksort algorithm, work has been done to see if similar benefits can achieved with three pivots. The three pivot quicksort algorithm as described by Kushagra-Ortiz-Qiau-Munro. They present an algorithm that intelligently partitions the list into four segments **??**. To select the pivots we select the first, third and fifth sextiles of a selection of seven evenly spaced values, similar to that of the dual pivot quicksort.

3.6 M Pivot Sort

In 2007 a patent was published for the M-pivot quicksort **??**. The algorithm uses ideas from probability and is considered to be a natural extension to the dual pivot quicksort and the three pivot quicksort. The concept hopes to take the best parts of quicksort and also deal with problems that the quicksort faces. M-pivot sort select M pairs of elements positioned linearly throughout the list (or sub-list). Thus $2M$ elements have been selected as potential pivots. The entries are moved to the end of the list then sorted with an algorithm that works well on small list sizes (like insertion sort). This is shown in Figure 1. After the M pivots are selected partitioning occurs

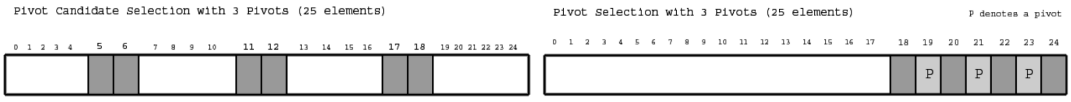


Figure 1: Diagram of how pivot selection is done in the M-pivot sort algorithm.

as demonstrated in Figure 2. For each pivot we cycle through the array and group elements together less than the current pivot. Following this, each of the sublists are recursed into. There are a few optimizations that can be made before starting any part of the algorithm (within the stack frame)[**?**]. Before the call to the M-pivot sort, we can enforce that the min-heap property is applied. One call to the min-heapify function would only add $O(n)$ time and would not change the asymptotic runtime of the algorithm.

There are two more optimizations that could be applied. If one or more pivots end up having the same value then a different pivot is selected**??**. The final proposed

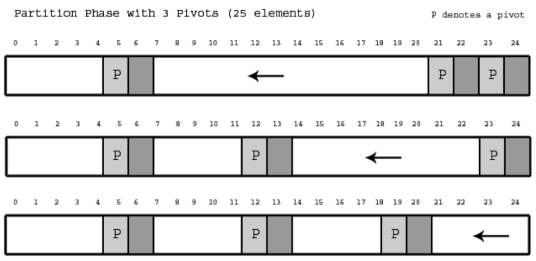


Figure 2: How partitioning is done in the M-pivot sort algorithm.

Sort Method	Comparisons
Classic	$2n \log n - 1.51n + O(\log(n))$
Dual Pivot	$2.13n \log n - 2.57n + O(\log(n))$
Optimal Dual Pivot	$1.8n \log n + O(n)$
Three Pivot	$1.846n \log n + O(n)$
Yaroslavskiy	$1.9n \log n - 2.46n + O(\log(n))$
M Pivot	$O(n \log n)$

Table 2: Summary table of theoretical comparisons.**??**[2][14][16]

optimization that could be made is that if all the pivots are found to be close to one side of the list, then the number of pivots is changed**??**. The previously two stated optimizations are not further studied in this paper.

3.7 Summary

Sort Method	Swaps
Classic	$0.33n \log n - 0.58n + O(\log(n))$
Dual Pivot	$0.8n \log n - 0.3n + O(\log(n))$
Optimal Dual Pivot	$0.33n \log n + O(n)$
Three Pivot	$0.615n \log n + O(n)$
Yaroslavskiy	$0.6n \log n + 0.08n + O(\log(n))$
M Pivot	$O(n \log n)$

Table 3: Summary table of theoretical swaps.**??**[2][14][16]

Sort	Pivot Selection methods	Number of Pivots
Classic	3	1
Dual Pivot	2	2
Optimal Dual Pivot	2	2
Yaroslavskiy	1	2
Three Pivot	1	3
M-Pivot	1	3,4,5,6
Heap Optimized M-Pivot	1	3,4,5,6

Table 4: List of all the variation of quicksorts executed.

4 Experiment

4.1 Overview

The experiments consisted of generating lists of integers ranging in size from four to fifty million. The integers were chosen uniformly at random between zero and 10^{12}. Consecutive lists were 9/7 the size of the previous rounded down to the nearest integer. Each quicksort and each of it's variations sorted the same list. The number of comparisons, swaps and time taken by each quicksort were recorded. A summary of the conditions can be found in Table 4 for a total of 17 conditions. A total of 69 iterations were run resulting in 109296 data points. During each iteration, the data gathering for each list size was done in parallel using four threads.

4.2 Platform

The experiments were run on a virtual machine with OSX 10.9 as the guest operating system and Windows 8.1 as the host operating system. The host operating system has 16 GB of ram and an i5-2500k CPU overclocked to 4.3 GHz. The guest OS had access to 8 GB of RAM and access to all 4 cores of the host CPU.

4.3 Results

Figure 3 provides the legend for all of the plots. Note that for the plots we overlay the function:

$$A \cdot n \log(n) + B \cdot n + C \log(n) \tag{1}$$

The nomenclature for the legend is as follows:

(Algorithm Name, Pivot Selection Method Index, Number of Pivots, is Insertion Sort used).

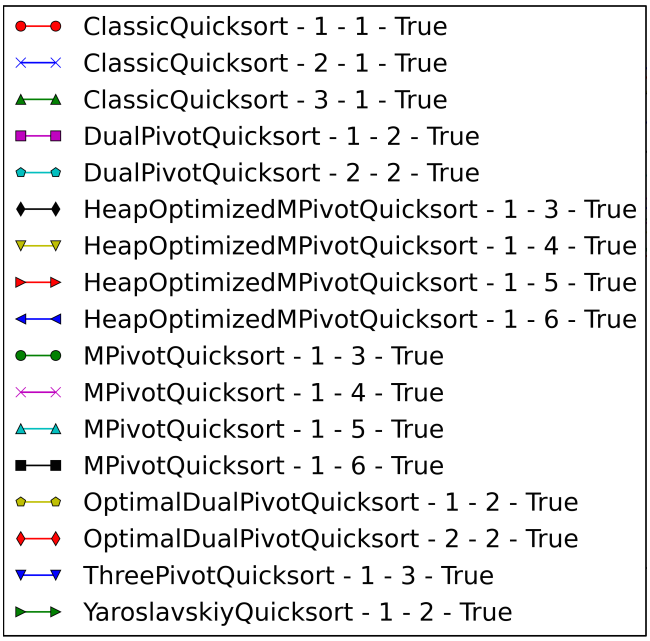


Figure 3: Plot legend

5 Analysis

5.1 Non-Linear Curve Fit

With the data that has been collected we wish to be able to look at a whole data set and then quantitatively compare to other data sets. From our introduction of the different quicksort algorithms in section , the asymptotic runtime of any of the described algorithms of the had one of the following forms:

$$A \cdot n \log(n) + B \cdot n + O(\log(n)) \tag{2}$$

$$A \cdot n \log(n) + O(n) \tag{3}$$

$$O(n \log(n)) \tag{4}$$

In correspondence of this we will fit the data to the following equation:

$$A \cdot n \log(n) + B \cdot n + C \log(n) \tag{5}$$

With the function parameters A, B and C were found using a non-linear curve fitting tool in the SciPy module in Python. Quantitatively, the fit function that was picked seems to result in good fits visually. Though the fit has less meaning do to the fact we don't have error bars on the data. A closer look at the reduced chi squared values of the fits leads us to believe that the fits are not good [17]. Thus we will treat the results from the best fit as qualitative.

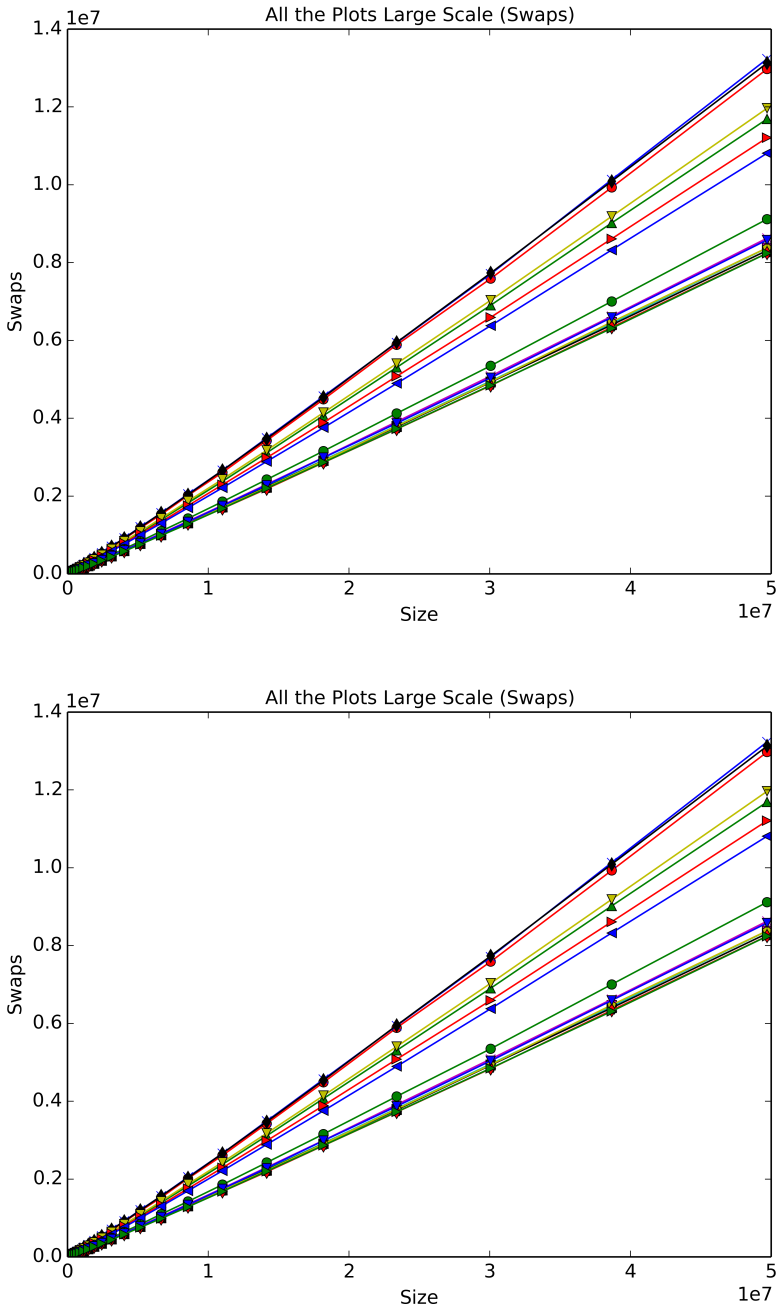


Figure 4: A plot of the data from all the sorting algorithm against the number of comparisons.

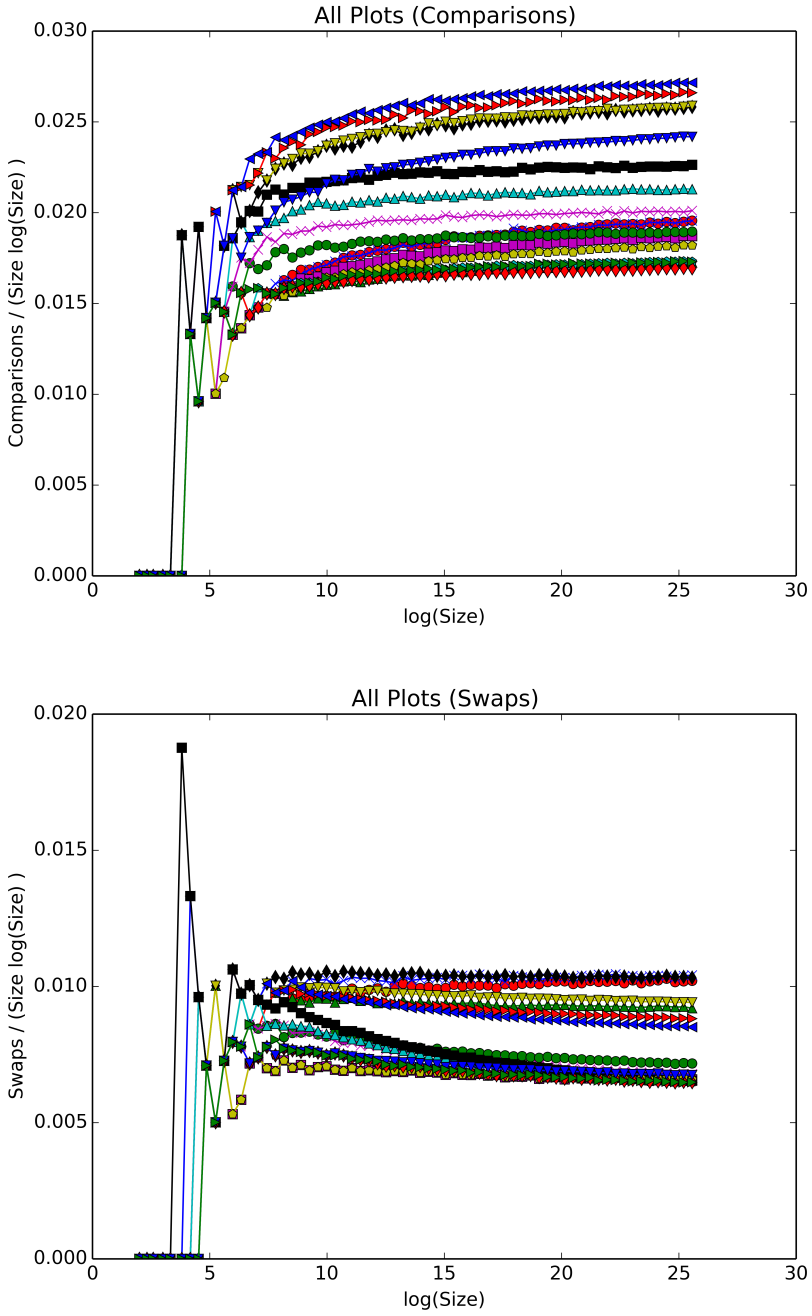


Figure 5: A plot of the data from all the sorting algorithm

5.2 Effect of Pivot Selection

In Figure 6 we can see that picking the pivot as the median of the fist, middle and last elements (green line) provides a significant decrease in comparisons and swaps

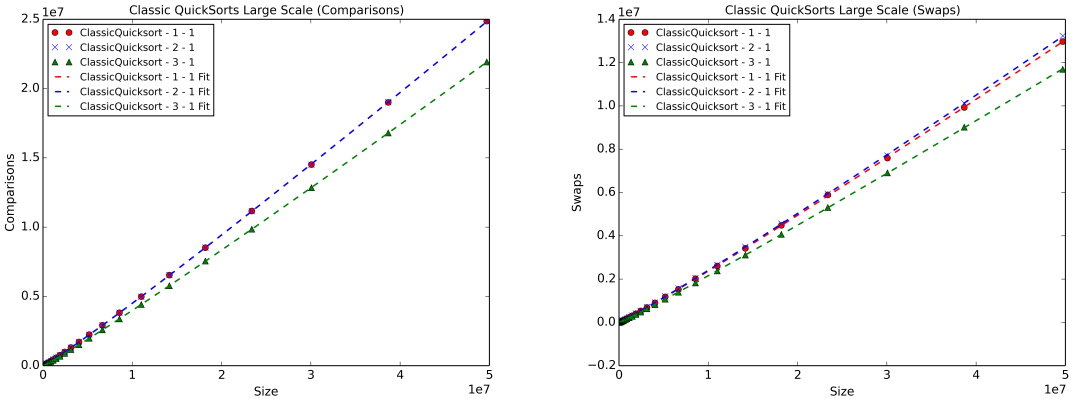


Figure 6: Swap and comparison counts for the classic quicksort.

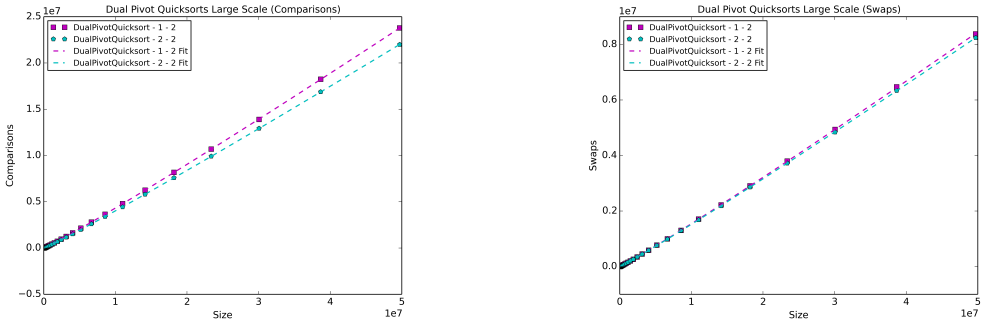


Figure 7: Swap and comparison counts for the dual pivot quicksort.

compared to picking the first element (red line) or last element (blue line). We expect that picking any single element should provide no difference but we see that selecting the last element as the pivot is beneficial over picking the first element. We believe that this may be an artifact of the random number generator in Python's random module which documents that it should not be used for cryptographically secure applications suggesting the generator is in some way subpar.

Similar to the classic quicksort, we see in Figures 7 and 8 that selecting pivots intelligently provides a benefit for the dual pivot quicksort and it's optimal counterpart. Interestingly, smart pivot selection has a large effect on the number of comparisons for both of the dual pivot quicksorts while having a less pronounced effect on the number of swaps. We believe that due to the larger number of pivots and the nature of the data we sorted there is a higher probability that each element will already be in it's correct partition and thus the effect of "good" pivots is less pronounced. Had the data been almost-sorted we believe that the effect of smart pivot selection would be more conspicuous.

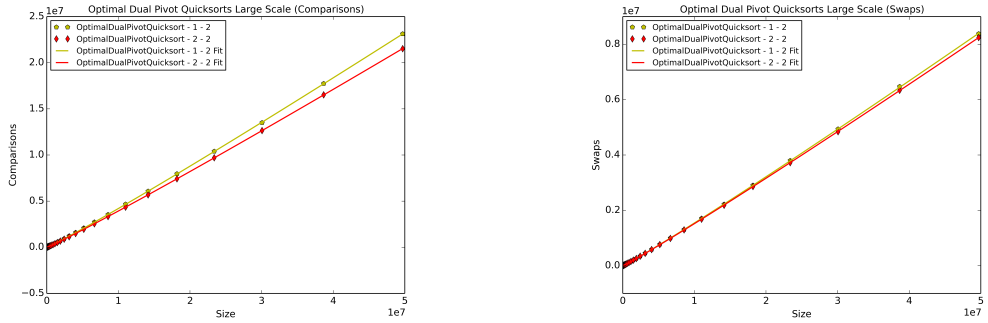


Figure 8: Swap and comparison counts for the optimal dual pivot quicksort.

6 Discussion

6.1 $n\log(n)$ Trend

A basic question to ask is if we are seeing the $nlog(n)$ trend in all of the sort data we see. From Figure 5 we see plots of $y/n\log(n)$ (where y is either the number of comparisons or swaps). As n gets larger you can see that each curve is tending to a constant. Since the $y/n\log(n)$ approaches a constant, this indicates that y is of $O(n)$. Which is consistent with the theory described in the introduction.

6.2 Comparisons among Quicksorts

Preliminary observations on Figure 4 show that the optimal dual pivot quicksort had the lowest number of comparisons. This is as expected since it was designed to minimize the number of comparisons. Also the Yaroslavskiy and M-Pivot sort (for $M = 3, 4, 5, 6$) have a minimal number of comparisons. In contrast, the number of comparisons for all of the heap optimized M-pivot sorts are poor. Unexpectedly, the three pivot quicksort and classic quicksort both have performance comparable to that of the heap optimized M-pivot sorts. The thee pivot quicksort performs surprisingly few swap operations in comparison to the M-pivot quicksorts.

An observation from the two pivot quicksorts with pivot selection that selects the first and last element as the pivots don't do as well as the selection of the quartiles in the number of comparisons. This can be seen from figure 9. When looking at size vs. swaps in the two pivot there are no noteworthy differences.

In comparisons between all the M-pivot sort algorithms, the ones without the heap optimization always did better. This is due to an implementation error. In the original description of the algorithm the heapfication of the array only happens on the very first call whereas in our implementation it occurs for every recursive call.

6.3 Fit Functions

From tables 5, 6, 7, 8, 9 and 10 the fit coefficients for the data collected is summarized based of equation 5. For the sake of asymptotic comparison between the algorithms we only focus on the fit parameter A. Based on these coefficients the optimal dual pivot quicksort with quartile pivot selection is shown to be the best quick sort algorithm based in regards to the number of comparisons. Conversely, the heap-optimized M pivot quicksort for $M = 6$, is the worst algorithm based in regards to the number of comparisons. The classic quicksort algorithms using the median-of-three for pivot selection has incredibly poor swap performance. The best algorithm for reducing the number of swaps is the basic M-pivot sort algorithm with $M = 6$. But the only algorithm with competitive runtimes for both metrics is Yaroslavskiy's dual pivot quicksort algorithm.

6.4 Future Work

While our tests only examined the performance of these quicksorts on lists of nearly unique numbers, we believe that examining their comparison and swap counts on different "kinds" of lists provides just as much, if not more, insight. This is especially true for concerns of their worst-case performance. Lists of numbers which can be modelled by a Gaussian or Poisson distribution may provide some insight into the behaviour of these quicksorts on real world data. Additionally, sorting lists in pseudo-sorted or sorted order will provide data for the worst-case performance of these sorts. Another potentially interesting may be one where elements of the list are chosen uniformly at random but where the range of the selected elements is less than the length of the list, thus guaranteeing value repetition.

Comparing these sorts against other sorts may also be of use. Other popular "efficient" sorts such as the merge sort or timsort can provide good points of reference for the performance of the quicksort. This can be especially useful if more "kinds" of lists are used as mentioned previously. We also believe that comparing the M-Pivot quicksort with $M = 1$ and $M = 2$ with the other one and two pivot sorts can result in potentially interesting graphs and analysis. We also believe that implementing the optimizations presented by Edmonson [12] on the quicksorts other than the M-Pivot Quicksort can prove useful in determining their true effectiveness.

7 Conclusion

We have implemented several variations of the quicksort and explored their efficiencies in regards to total number of swaps and comparisons. Our results show that Yaroslavskiy's quicksort is the highest overall performer while the heap optimized M-pivot quicksorts performed the worse. Our results also show that the optimal dual pivot quicksort performs better than Yaroslavskiy's quicksort in regards to total

Sort Method	$A_{\text{comparisons}}$
Classic Quicksort - 1 - 1	0.02151 ± 0.00019
Classic Quicksort - 2 - 1	0.02135 ± 0.00017
Classic Quicksort - 3 - 1	0.01807 ± 0.00006
DualPivot Quicksort - 1 - 2	0.02014 ± 0.00018
DualPivot Quicksort - 2 - 2	0.01772 ± 0.00007
Heap Optimized M-Pivot Quicksort - 1 - 3	0.02778 ± 0.00014
Heap Optimized M-Pivot Quicksort - 1 - 4	0.02788 ± 0.00015
Heap Optimized M-Pivot Quicksort - 1 - 5	0.02842 ± 0.00025
Heap Optimized M-Pivot Quicksort - 1 - 6	0.02869 ± 0.00011
M-Pivot Quicksort - 1 - 3	0.01970 ± 0.00009
M-Pivot Quicksort - 1 - 4	0.02076 ± 0.00015
M-Pivot Quicksort - 1 - 5	0.02165 ± 0.00010
M-Pivot Quicksort - 1 - 6	0.02386 ± 0.00014
Optimal Dual Pivot Quicksort - 1 - 2	0.01959 ± 0.00019
Optimal Dual Pivot Quicksort - 2 - 2	0.01744 ± 0.00007
Three Pivot Quicksort - 1 - 3	0.02587 ± 0.00009
Yaroslavskiy Quicksort - 1 - 2	0.01796 ± 0.00010

Table 5: Summary table coefficients of the non-linear fit for the parameter A on the comparison data.

number of comparisons as claimed by [14] while the two sorts are nearly identical in swap count. Overall, the M-Pivot quicksorts performed well. The M-Pivot quicksort with $M = 3$ performed better than the three pivot quicksort of Kushagra et al. [2].

Our results also validate the idea that intelligent pivot selection provides a measurable and significant effect on the overall performance of quicksorts. We find that for lists of nearly unique numbers, selecting a median of three and first and third quartiles of five provide good performance to developer time ratios. This paper serves as a single point of reference for several quicksort algorithms using a varying number of pivots and several different optimizations. We believe that the work done here can be easily used and extended to compare quicksorts among themselves and against other sorts such as the merge sort and the timsort.

8 Fit Coefficients

9 Misc Plots

Sort Method	$B_{\text{comparisons}}$
Classic Quicksort - 1 - 1	-0.05025 ± 0.00469
Classic Quicksort - 2 - 1	-0.04634 ± 0.00428
Classic Quicksort - 3 - 1	-0.02126 ± 0.00163
DualPivot Quicksort - 1 - 2	-0.03650 ± 0.00465
DualPivot Quicksort - 2 - 2	-0.01045 ± 0.00183
Heap Optimized M-Pivot Quicksort - 1 - 3	-0.05055 ± 0.00355
Heap Optimized M-Pivot Quicksort - 1 - 4	-0.05152 ± 0.00368
Heap Optimized M-Pivot Quicksort - 1 - 5	-0.04639 ± 0.00626
Heap Optimized M-Pivot Quicksort - 1 - 6	-0.03889 ± 0.00280
M-Pivot Quicksort - 1 - 3	-0.01917 ± 0.00220
M-Pivot Quicksort - 1 - 4	-0.01716 ± 0.00391
M-Pivot Quicksort - 1 - 5	-0.00902 ± 0.00244
M-Pivot Quicksort - 1 - 6	-0.03206 ± 0.00366
Optimal Dual Pivot Quicksort - 1 - 2	-0.03580 ± 0.00490
Optimal Dual Pivot Quicksort - 2 - 2	-0.01266 ± 0.00165
Three Pivot Quicksort - 1 - 3	-0.04282 ± 0.00232
Yaroslavskiy Quicksort - 1 - 2	-0.01636 ± 0.00251

Table 6: Summary table coefficients of the non-linear fit for the parameter B on the comparison data.

Sort Method	$C_{\text{comparisons}}$
Classic Quicksort - 1 - 1	121.34341 ± 99.97322
Classic Quicksort - 2 - 1	78.33233 ± 91.31275
Classic Quicksort - 3 - 1	22.52742 ± 34.75034
DualPivot Quicksort - 1 - 2	52.22569 ± 99.17037
DualPivot Quicksort - 2 - 2	-53.25104 ± 39.07336
Heap Optimized M-Pivot Quicksort - 1 - 3	94.09226 ± 75.71998
Heap Optimized M-Pivot Quicksort - 1 - 4	124.92512 ± 78.46872
Heap Optimized M-Pivot Quicksort - 1 - 5	56.97527 ± 133.52044
Heap Optimized M-Pivot Quicksort - 1 - 6	29.84293 ± 59.67899
M-Pivot Quicksort - 1 - 3	51.60730 ± 46.87649
M-Pivot Quicksort - 1 - 4	49.35746 ± 83.31083
M-Pivot Quicksort - 1 - 5	4.76647 ± 52.03322
M-Pivot Quicksort - 1 - 6	136.38815 ± 77.96382
Optimal Dual Pivot Quicksort - 1 - 2	56.95127 ± 104.39452
Optimal Dual Pivot Quicksort - 2 - 2	-26.20004 ± 35.22097
Three Pivot Quicksort - 1 - 3	-17.79746 ± 49.38183
Yaroslavskiy Quicksort - 1 - 2	0.73204 ± 53.59187

Table 7: Summary table coefficients of the non-linear fit for the parameter C on the comparison data.

Sort Method	A_{swap}
Classic Quicksort - 1 - 1	0.01026 ± 0.00017
Classic Quicksort - 2 - 1	0.01095 ± 0.00016
Classic Quicksort - 3 - 1	0.00848 ± 0.00012
DualPivot Quicksort - 1 - 2	0.00629 ± 0.00010
DualPivot Quicksort - 2 - 2	0.00606 ± 0.00006
Heap Optimized M-Pivot Quicksort - 1 - 3	0.01004 ± 0.00009
Heap Optimized M-Pivot Quicksort - 1 - 4	0.00898 ± 0.00004
Heap Optimized M-Pivot Quicksort - 1 - 5	0.00809 ± 0.00004
Heap Optimized M-Pivot Quicksort - 1 - 6	0.00759 ± 0.00005
M-Pivot Quicksort - 1 - 3	0.00672 ± 0.00006
M-Pivot Quicksort - 1 - 4	0.00605 ± 0.00003
M-Pivot Quicksort - 1 - 5	0.00535 ± 0.00003
M-Pivot Quicksort - 1 - 6	0.00513 ± 0.00003
Optimal Dual Pivot Quicksort - 1 - 2	0.00629 ± 0.00010
Optimal Dual Pivot Quicksort - 2 - 2	0.00606 ± 0.00006
Three Pivot Quicksort - 1 - 3	0.00635 ± 0.00006
Yaroslavskiy Quicksort - 1 - 2	0.00586 ± 0.00005

Table 8: Summary table coefficients of the non-linear fit for the parameter A on the swap data.

Sort Method	B_{swap}
Classic Quicksort - 1 - 1	-0.00132 ± 0.00442
Classic Quicksort - 2 - 1	-0.01411 ± 0.00417
Classic Quicksort - 3 - 1	0.01903 ± 0.00298
DualPivot Quicksort - 1 - 2	0.00824 ± 0.00264
DualPivot Quicksort - 2 - 2	0.01080 ± 0.00153
Heap Optimized M-Pivot Quicksort - 1 - 3	0.00774 ± 0.00233
Heap Optimized M-Pivot Quicksort - 1 - 4	0.01111 ± 0.00097
Heap Optimized M-Pivot Quicksort - 1 - 5	0.01881 ± 0.00113
Heap Optimized M-Pivot Quicksort - 1 - 6	0.02363 ± 0.00129
M-Pivot Quicksort - 1 - 3	0.01160 ± 0.00156
M-Pivot Quicksort - 1 - 4	0.01890 ± 0.00069
M-Pivot Quicksort - 1 - 5	0.03171 ± 0.00065
M-Pivot Quicksort - 1 - 6	0.03601 ± 0.00077
Optimal Dual Pivot Quicksort - 1 - 2	0.00824 ± 0.00264
Optimal Dual Pivot Quicksort - 2 - 2	0.01080 ± 0.00153
Three Pivot Quicksort - 1 - 3	0.01030 ± 0.00153
Yaroslavskiy Quicksort - 1 - 2	0.01592 ± 0.00127

Table 9: Summary table coefficients of the non-linear fit for the parameter B on the swap data.

Sort Method	C_{swap}
Classic Quicksort - 1 - 1	-36.52075 ± 94.28550
Classic Quicksort - 2 - 1	102.34036 ± 88.94077
Classic Quicksort - 3 - 1	-103.21696 ± 63.57056
DualPivot Quicksort - 1 - 2	-38.02577 ± 56.38166
DualPivot Quicksort - 2 - 2	42.34411 ± 32.52471
Heap Optimized M-Pivot Quicksort - 1 - 3	-53.81114 ± 49.78070
Heap Optimized M-Pivot Quicksort - 1 - 4	-3.24127 ± 20.75454
Heap Optimized M-Pivot Quicksort - 1 - 5	-16.07787 ± 24.00667
Heap Optimized M-Pivot Quicksort - 1 - 6	-20.99394 ± 27.47669
M-Pivot Quicksort - 1 - 3	41.76450 ± 33.26955
M-Pivot Quicksort - 1 - 4	16.90493 ± 14.70501
M-Pivot Quicksort - 1 - 5	-17.36329 ± 13.95680
M-Pivot Quicksort - 1 - 6	-5.54593 ± 16.46902
Optimal Dual Pivot Quicksort - 1 - 2	-38.02577 ± 56.38166
Optimal Dual Pivot Quicksort - 2 - 2	42.34411 ± 32.52471
Three Pivot Quicksort - 1 - 3	14.68107 ± 32.72260
Yaroslavskiy Quicksort - 1 - 2	6.27071 ± 27.13874

Table 10: Summary table coefficients of the non-linear fit for the parameter C on the swap data.

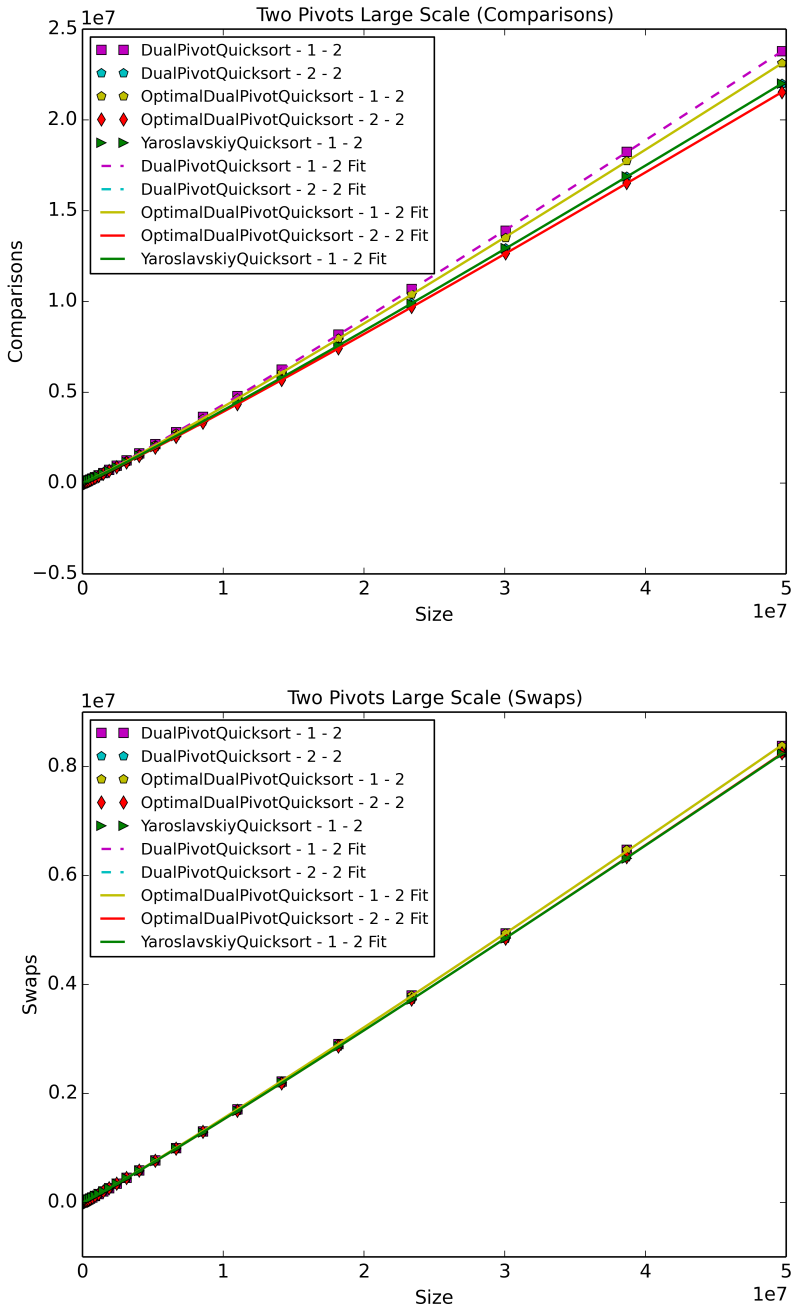
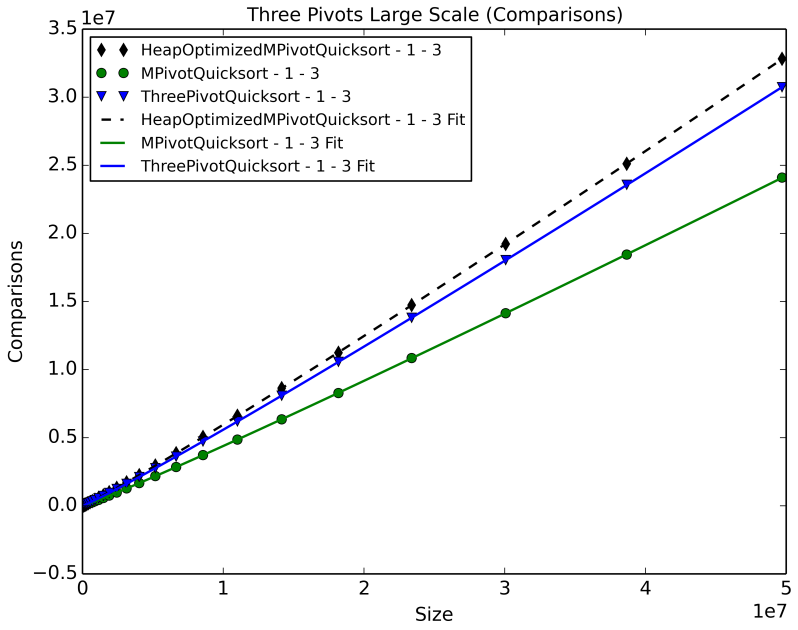


Figure 9: A plot of the data from all the sorting algorithm with two pivots



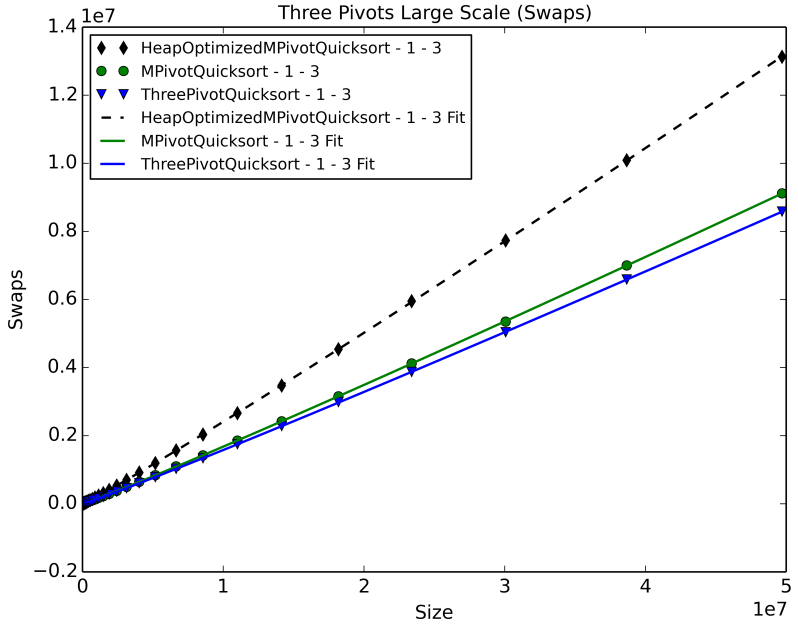


Figure 10: A plot of the data from all the sorting algorithm with three pivots

References

[1] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 1962. 1, 2

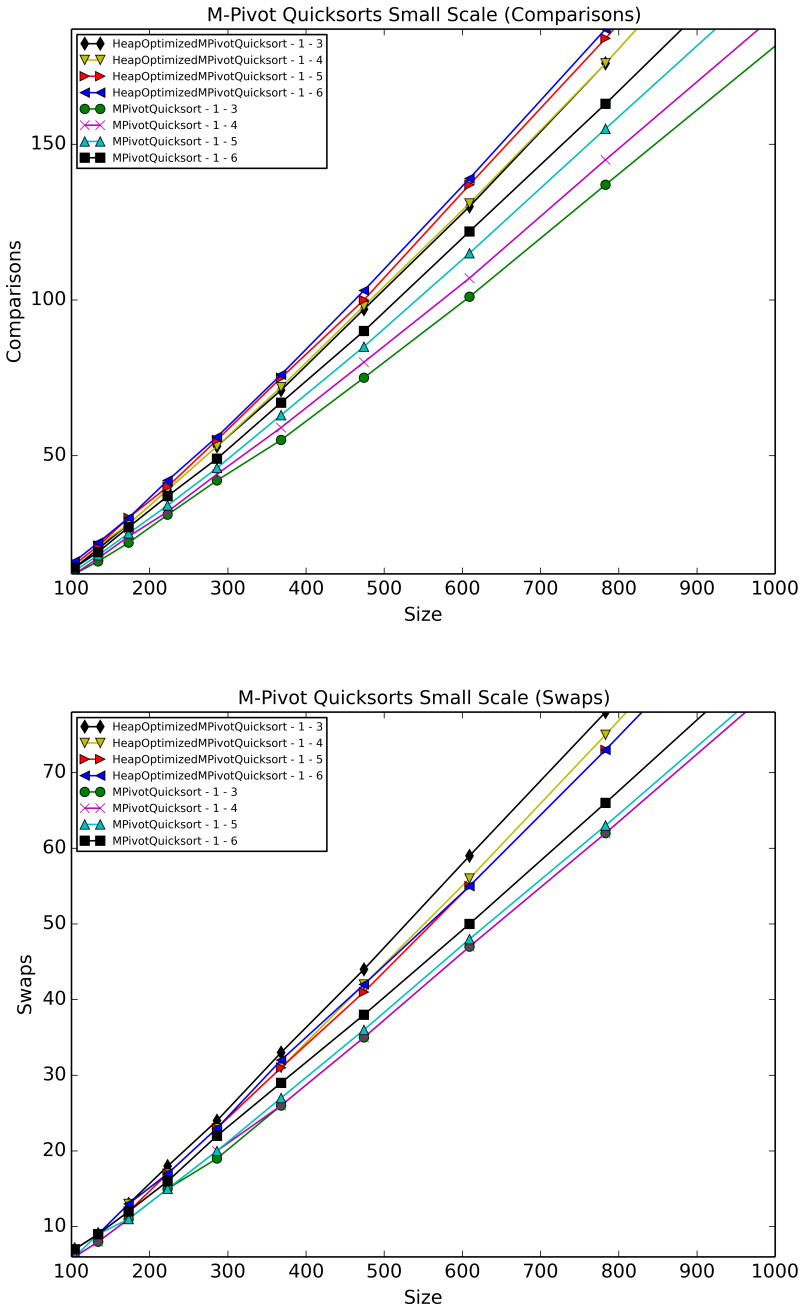


Figure 11: Data from all the version of M-Pivot Sort.

[2] S. Kushagra, A. López-Ortiz, A. Qiao, and J. I. Munro, "Multi-pivot quicksort: Theory and experiments," 2013. 1, 2, 5, 13

[3] R. Sedgewick and K. Wayne, "Advanced topics in sorting," 2007. 1

[4] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001. 1

[5] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998. 1

[6] M. H. van Emden, "Increasing the efficiency of quicksort," *Communications of the ACM*, vol. 13, no. 9, pp. 563–567, 1970. 2

[7] M. Foley and C. A. R. Hoare, "Proof of a recursive program: Quicksort," *The Computer Journal*, vol. 14, no. 4, pp. 391–395, 1971. 2

[8] R. Sedgewick, "The analysis of quicksort programs," *Acta Informatica*, vol. 7, no. 4, pp. 327–355, 1977. 2

[9] R. Sedgewick, "Implementing quicksort programs," *Communications of the ACM*, vol. 21, no. 10, pp. 847–857, 1978. 2, 3

[10] D. Cederman and P. Tsigas, "Gpu-quicksort: A practical quicksort algorithm for graphics processors," *Journal of Experimental Algorithmics (JEA)*, vol. 14, p. 4, 2009. 2

[11] C. Martínez and S. Roura, "Optimal sampling strategies in quicksort and quickselect," *SIAM Journal on Computing*, vol. 31, no. 3, pp. 683–705, 2001. 2

[12] J. R. Edmondson, "M pivot sort-replacing quick sort.," in *AMCS*, pp. 47–53, 2005. 2, 12

[13] V. Yaroslavskiy, "Dual-pivot quicksort," *Research Disclosure RD539015 (Sept., 2009), http://iaroslavski. narod. ru/quicksort/DualPivotQuicksort. pdf.* 2

[14] M. Aumüller and M. Dietzfelbinger, "Optimal partitioning for dual pivot quicksort," in *Proceedings of the 40th International Conference on Automata, Languages, and Programming - Volume Part I*, ICALP'13, (Berlin, Heidelberg), pp. 33–44, Springer-Verlag, 2013. 2, 3, 4, 5, 13

[15] C. R. Cook and D. J. Kim, "Best sorting algorithm for nearly sorted lists," *Communications of the ACM*, vol. 23, no. 11, pp. 620–624, 1980. 3

[16] S. Wild and M. E. Nebel, "Average case analysis of java 7's dual pivot quicksort," in *Proceedings of the 20th Annual European Conference on Algorithms*, ESA'12, (Berlin, Heidelberg), pp. 825–836, Springer-Verlag, 2012. 3, 5

[17] P. Bevington, *Data reduction and error analysis for the physical sciences*. McGraw-Hill, 1969. 7

A GitHub Repository

Project GitHub Repository Link