Multi-pivot QuickSort Analysis

Jeff XING

April 23, 2019

1 INTRODUCTION

QuickSort [2] has been implemented widely in practise since 1960s. Utilizing the divided and conquer, QuickSort gains good performance by choose one pivot to seperate the input array into two parts (one larger than pivot and another is smaller than pivot) in each iteration. It was thought that one pivot scheme was better than the multiple pivot scheme until dual pivot was proved better by implementation [4] in 2009. Then, 3-pivot QuickSort algorithm [3] was put up by introducing a smart way to swap the elements. In this report, we will explain and implement QuickSort, 2-pivot QuickSort and 3-pivot QuickSort. Through the experiment, we try to show the gains of more pivot. We will also try to get some insights. Finally, we will explore the impact of data pattern (many duplicate elements or almost sorted) on the algorithm.

2 BACKGROUNDS

2.1 QuickSort

Let $A = [a_1, a_2, \cdots, a_N]$ denote an unsorted array. Sort function can make A into a sorted one. In QuickSort, there are two basic operations: `swap(A,m,n)` and `partition(A, start, end)`.

`swap(A,m,n)` is used to swap the elements $A[m]$ and $A[n]$ in A. `partition(A, start, end)` is used to swap the elements between `start` and `end` so that the elements are divided into two sections in Figure 1. The elements in left section are smaller than the pivot. The elements in right section are no smaller than the pivot. Note that the pivot is well picked between A[`start`] and A[`end`].

QuickSort is just partition the unsorted array into two sections and Quick-Sort the two sections recursively until the whole array is sorted.

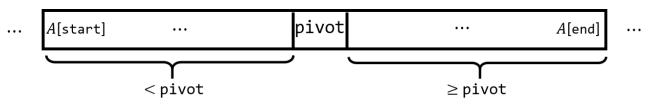


Figure 1: Partition for one-pivot QuickSort

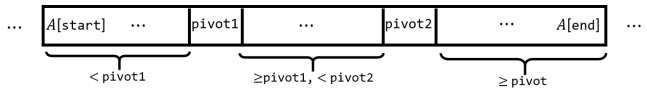


Figure 2: Partition for 2-pivot QuickSort

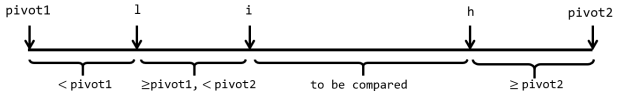


Figure 3: Partition process for 2-pivot QuickSort

2.2 2-pivot QuickSort

2-pivot QuickSort is almost the same as QuickSort except the `partition(A, start, end)`. In the partition function in 2-pivot QuickSort, we use 2 pivots instead of one and define the function as `partition2(A, start, end)`. We can get the result as in Figure 2.

`partition2(A, start, end)` is a little more complex than `partition(A, start, end)`. Here, we pick A[`start`] and A[`end`] as `pivot1` and `pivot2`. Here, we let `pivot1` is smaller than `pivot2`. We use three pointers l, i, h to point the position the next low element, middle element and high element to place. In each step as Figure 3, we compare $A[i]$ with `pivot1` and `pivot2`. There are three cases we need to deal with:

- $A[i] <$ `pivot1`: `swap(A,i,l)`, `i++`, `l++`
- `pivot1` $\leq A[i] <$ `pivot12`: `i++`
- $A[i] \geq$ `pivot2`: `swap(A,i,h)`, `h--`

Finally, we swap the `pivot1` and `pivot2` to right places.

2.3 3-pivot QuickSort

The partition process of 3-pivot QuickSort is shown in Figure 4. There are four cases of $A[i]$. As there are neighbouring sections, swapping the elements may influence the neighbouring sections. Extra swaps are needed. Thus, how to smartly swap the elements should be well designed. Luckily, a good 3-pivot partition algorithm has been put in Algorithm A.1.1 in [3]. Here, we use the partition algorithm in [3].

3 EXPERIMENTS

3.1 Gains from Pivot

In this part, we try to answer: How many pivots is optimal? Why do more pivots help? Why do too many pivots hurt?

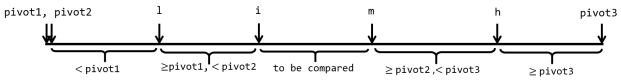
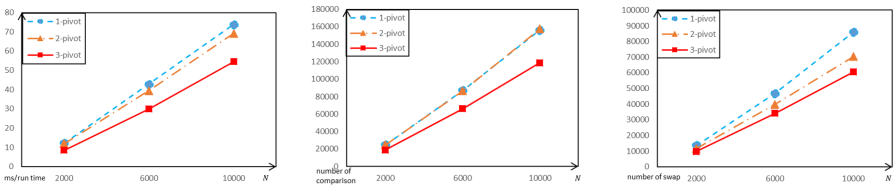


Figure 4: Partition process for 3-pivot QuickSort

We compare the run time, comparison numbers and swap numbers of 1-pivot QuickSort, 2-pivot QuickSort and 3-pivot QuickSort. We generate 2000-length, 6000-length and 10000-length arrays each time. Each elements are randomly assigned between 0 and 100000. Thus, the arrays are totally unsorted and has little duplicated elements. The QuickSort functions process the same array. For accuracy, we run 30 times. The arrays of different time are different. finally, we focus on the average performance.



(a) run time vs. length of array (b) number of comparison vs. length of array (c) number of swap vs. length of array

Figure 5: Results of 1-pivot QuickSort, 2-pivot QuickSort and 3-pivot QuickSort

Figure 5 shows the run time, number of comparison and number of swap versus the length of array for 1-pivot QuickSort, 2-pivot QuickSort and 3-pivot QuickSort. We can see that as the number of pivots increases, the the run time and number of swap both decrease. The numbers of comparison fr single-pivot and dual-pivot are almost the same while the decrease of the number of comparison of 3-pivot QuickSort can be observed. From this results, it seems that more pivots can bring benefit of performance if rearrangement the element in the partition function can be wisely designed.

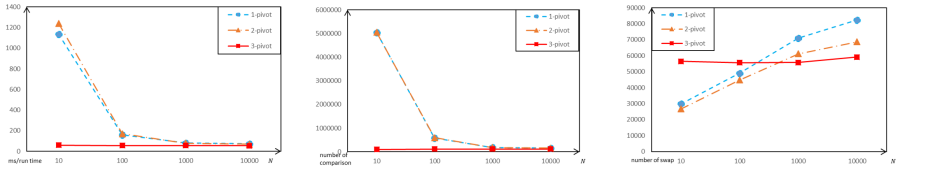
As for how many pivots are optimal, there are also some references working on this question. [1] concludes that 3 to 5 pivots are best choice for they visit the fewest array cells with good cache behavior by the experiment using a proposed generalized partition algorithm in multi-pivot Quicksort.

From the experience, we can see that adopting more pivot can reduce the number of comparison and the number of swap. This may be the reason why more pivots help. However, when the number of pivot grow large, more extra swap operation (or some other array accesses) are needed for the array rearrangement in partition function, which hurts the performance of QuickSort.

3.2 Impact of Duplicate Elements

In this part, we try to answer: What if the array has many duplicate elements? How does that change the situation? Does that change the optimal number of pivots?

Here, we change the range of the elements of the array to control how many duplicate elements are in. The length of the array is 10000.



(a) run time vs. variant range (b) number of comparison vs. variant range (c) number of swap vs. variant range

Figure 6: Results of 1-pivot QuickSort, 2-pivot QuickSort and 3-pivot QuickSort

From Figure 6, we can see that the run time and the number of comparison increase with the number of duplicate elements and the number of swap decreases with the number of duplicate elements. We can see that the comparison dominate the time consumption. If there are many same elements, some of the partitions can be large, more comparisons are needed to sort the same element. From the picture, we can guess the optimal pivot number will increase.

References

[1] Martin Aumüller, Martin Dietzfelbinger, and Pascal Klaue. How good is multi-pivot quicksort? *ACM Transactions on Algorithms (TALG)*, 13(1):8, 2016.

[2] Charles AR Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[3] Shrinu Kushagra, Alejandro López-Ortiz, J Ian Munro, and Aurick Qiao. Multi-pivot quicksort: Theory and experiments. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 47–60. Society for Industrial and Applied Mathematics, 2014.

[4] Vladimir Yaroslavskiy. Dual-pivot quicksort. *Research Disclosure*, 2009.