Performance Analysis of Multi-Pivots Quicksort

Wang Junming A0162553H

March 2019

1 Introduction

Quick sort has been know as one of the most classic sorting algorithms. Its primary method is randomly choosing a pivot, partition the input into two parts and solve the two parts recursively. Multi-pivots quick sort is a modification where we choose multiple pivots at a time and partition the input into multiple parts. Today by running a series of experiment, we hope to study and compare characteristic of multi-pivots quick sort against single-pivot quick sort.

2 Implementation

To be able to better compare the performance of quick sort with different number of pivots, I have implemented a general version of quick-sort in which the number of pivots can be given as a parameter. It ensures that the number of pivots is the only variable here. It is non-trivial to implement partition with multiple pivots efficiently and in-place. My final partition algorithm achieved (kn/2 + k^2) swaps. It works by one pass scan and maintaining boundaries of each partition. Due to its complexity and edge cases, the details of the algorithm is not discussed here. The partition has been optimized with a lot of efforts, the resulting quick sort finished sorting 20,000,000 elements in 2 seconds compared to the C++ standard library of 1.5 seconds, there are still possible optimization haven't being done due to lack of time.

3 Discussions and Experiments

A series of questions regarding multi-pivots quick sort has been discussed and testified in this section.

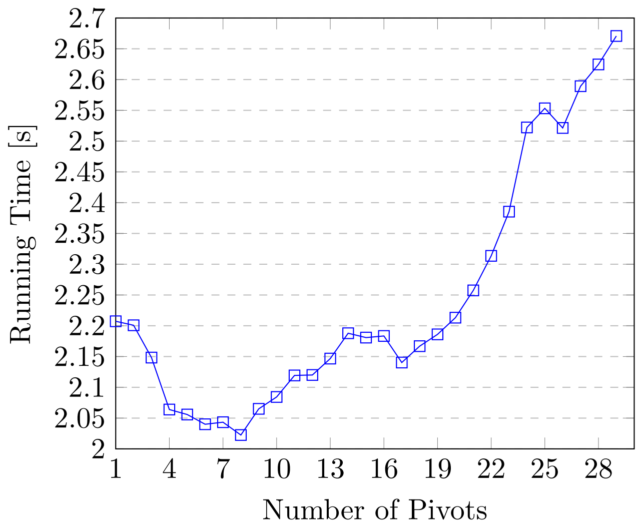
3.1 Question 1: How many pivots is optimal?

3.1.1 Hypothesis

The asymptotic running time is the same as single pivot quick sort, more pivots reduces the recurrence depth, but increases the number of comparisons and swaps in each partition. Thus my guess is the optimal number of pivots is when we find the balance between the two factors.

3.1.2 Experiment

The figure below shows the performance of quick sort with different number of pivots on a randomly generated array of size 20,000,000.



We firstly get improvements when number of pivots increases, but as the number of pivots keep increasing, the performance decreases. From the graph, we can see that we get good performance with 4 - 8 pivots. In which 8 pivots achieved best performance. But the difference is not very huge, thus I am not that confident to say 8 is optimal.

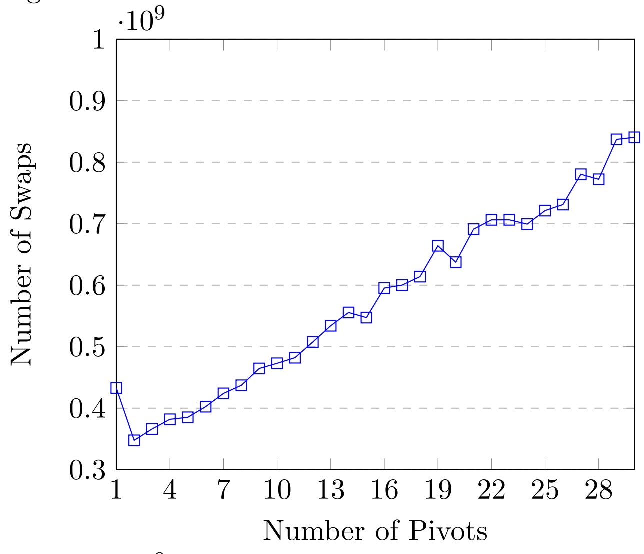
3.2 Question 2 and 3: Why do more pivots help? Why do more pivots hurt?

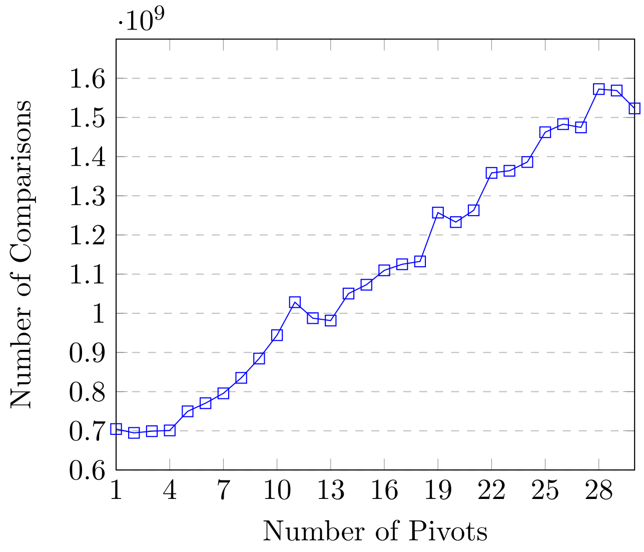
3.2.1 Hypothesis

As mentioned in Question 1, I think more pivots help as it reduced the recurrence depth, but in the meanwhile, it increases the number of swaps and comparisons in each partition.

3.2.2 Experiment

In order to understand the performance, I have modified the code to record the number of comparisons and swaps of sorting 20,000,000 elements, the two figures below shows the result.





The number of comparisons almost grows linearly as the number of pivots grows. However, with 2 - 7 pivots, the number of swaps is less than single pivot quick sort, this explains why 2 - 7(Notice in question 1 we said 2 - 8, because experiments with running time and swaps are conducted separately as recording the number of swaps will affect running time). I think the major reason to see the improvement is the decreasing of number of comparisons as we see in the figure, this is the combined result of decreased depth of recurrence and increased number of swaps and comparisons in each partition. Thus I think as the number of pivots grows, the increase in the number of swaps and comparisons in each partition dominates the decrease in the depth of recurrence, which hurts us in performance.

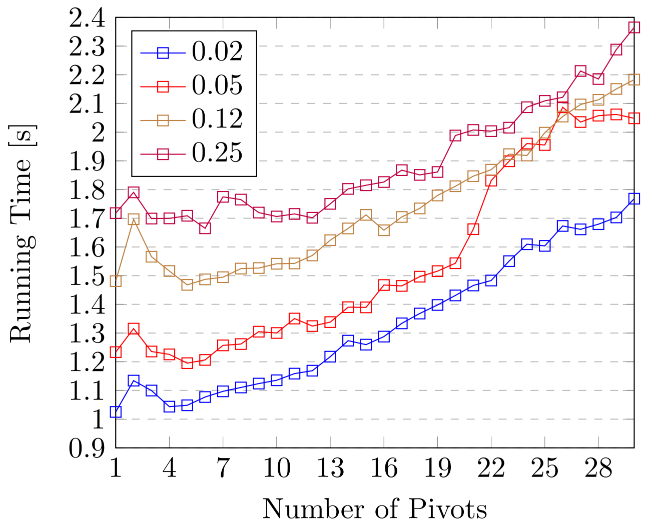
3.3 Question 4: Is multi-pivots QuickSort any better or worse on almost sorted data?

3.3.1 Hypothesis

Intuitively, almost sorted data could indicate less number of swaps in partition, if the partition algorithm has been implemented carefully. Since the number of swaps in each partition is affected, the optimal number of pivots might also be affected.

3.3.2 Experiment

We define unsorted rate to be the fraction of elements in the array that is out of order. Experiments were conducted on a randomly generated array of 20,000,000 elements with unsorted rate from 0 to 50%. The figure below shows results.



For all number of pivots, we have observed improvements when the unsorted rate is decreasing. But for each unsorted rate, when the number of pivots increase, we see similar tendency that the running time first decrease, and then increase. One interesting fact I observed is that when is data is almost sorted(unsorted rate is small), then single pivot seems to perform better. When the data is more random(unsorted rate is big), the optimal pivots could perform better than single pivot. I think is highly related to the implementation of my partition algorithm.

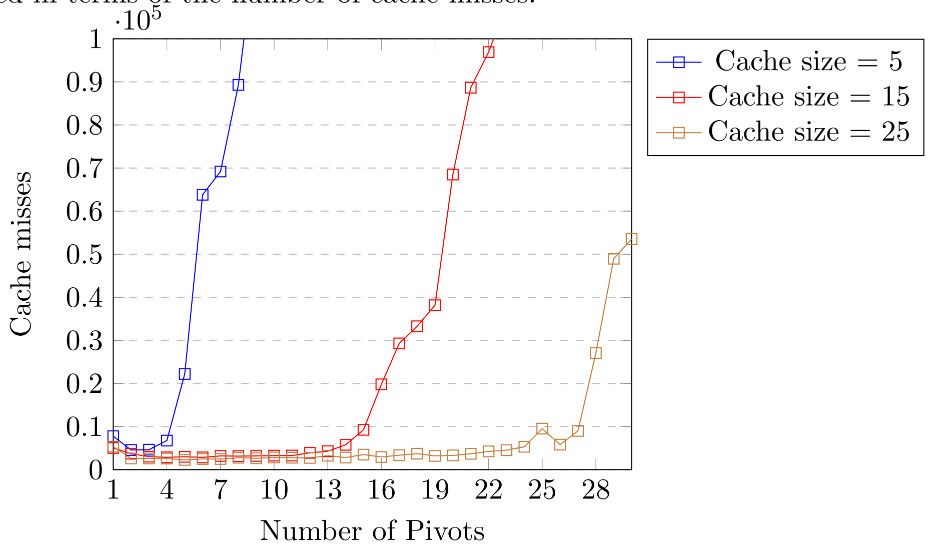
3.4 Question 5: Does the size of cache change the optimal number of pivots?

3.4.1 Hypothesis

In my implementation of partition, more pivots means more random accesses in the array, the partition algorithm is one pass, but for each of the non-pivot element in the array, we have to adjust the position of every pivot. Since the number of pivots is small and is chosen randomly, roughly pivots are from different blocks. Suppose we have k pivots, then for each of the element, accessing k pivots would require accessing k different blocks. It would be okay if we could hold those k blocks in cache, otherwise the cache performance would be very bad.

3.4.2 Experiment

I have implemented a LRU cache simulator for the purpose of this experiment, all array element accesses are substituted by accessing through the simulator interface. Accessing through cache simulator is much slower compared to direct array access, thus this experiment is run with a randomly generated array of 200,000 integer elements, the block size is 100 and cache size varies from 5 to 25 blocks. In this experiment, the performance is measured in terms of the number of cache misses.



As we can see from the figure, the experiment almost verifies my hypothesis, when the cache size is smaller than the number of pivots, that is, the cache cannot hold all the blocks that contains pivots, the cache performance is very bad. For example, when cache is 15 blocks, when the number of pivots is lager than 15, the cache misses increases dramatically.

4 Conclusion

We have seen that multi-pivots quick sort can have better performance compared to single pivot. But the performance does not increase with the number of pivots linearly, when reach certain number of pivots, the performance starts to decrease. One explanation is that multi-pivots partition requires more number of swaps that dominates the decrease in the height of recurrence tree. Another possibility, as we have seen in Question 5, is when the cache cannot hold all the blocks that contains pivots, then the number of cache misses starts to increase rapidly. Apart from that, we have seem improvement when the input data is almost sorted, that is because partition on almost sorted data requires less number of swaps, but surprisingly, when the input data is almost sorted, single pivot seems to perform better than multi-pivots with any number of pivots, the reason remains to be further investigated but must be related to my implementation of partition algorithm.