UniTartuCS Poster

Rustam Abdumalikov, Aral Acikalin[1]

[1]Institute of Computer Science, University of Tartu

UNIVERSITY OF TARTU 1632

An Important Problem

Sorting is one of the most important problem in Computer Science. And quicksort is quite interesting in this regard. It was developed by British computer scientist Tony Hoare in 1959 and for many years computer scientists considered that single-pivot implementation is the most optimal. Although Sedgewick and Bentley [1] in 2002 discovered potential for multi-pivot but failed to prove it through implementation. Since 2002 no one tried to do anything about quicksort, until the year 2009 when Vladimir Yaroslavskiy [3] empirically showed that quicksort with dual-pivot is faster than single-pivot quicksort. This discovery disrupted the Computer Science community, which lead to the idea on how much we can increase the number of pivots and get better performance.

Model

In this research project, we tried to answer the following questions, "As we increase the number of pivots beyond three would it lead to better performance? And why?". In order to answer those questions, it would require implementing quicksort with general partitioning, where we could easily choose a number of pivots to use. The reason for general partitioning is that we want to test all the extreme cases, even 1000 pivots and so on, and we can't possibly write 1000 separate implementations. On top of that, we would need to analyze low-level CPU statistics.

Algorithm

The algorithm for quicksort with general partitioning turned out to be quite tricky to implement. But the most challenging part of all was to make it efficient. Nonetheless, here primary steps of the algorithm:

1. Randomly choose n-pivots
2. Sort n-pivots and put them at the beginning of the array
3. iterate over all entries of array in reverse order
4. n-pivots divides array into $n+1$ sectors. We need to identify the sector where the current element belongs.
5. After the correct sector is found, in order to put the current entry into found sector we need to do serious of swaps.
6. Repeat steps 1-5 recursively for all sub-partitions until array is sorted.

Results

The goal of the present paper was to answer the following questions "As we increase the number of pivots beyond three, would it lead to better performance? And why?" As it turns out during testing, we discovered that as we increase the number of pivots, the performance starts to improve until we reach a certain bound in a number of pivots, after which it starts to decrease.

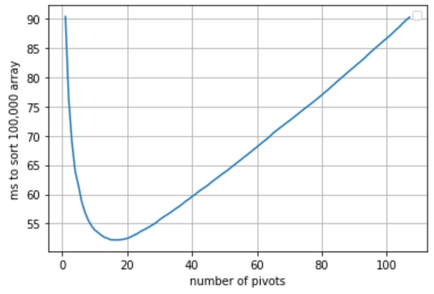

Fig. 1: Various Pivot Performance

From those finding two additional questions arise, "Why is the performance getting better? And why does the performance starts to worsen after some point?". To answer those questions, we need to dive deeper into low-level CPU statistics(such as **perf** and **valgrind**) to understand what might be the reason behind this behavior. This what we discovered:

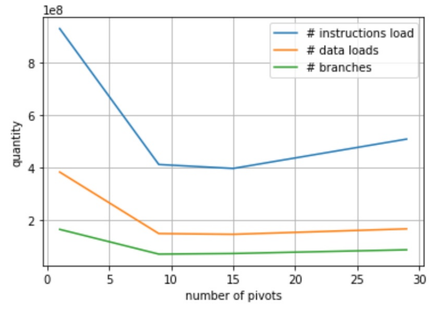

Fig. 2: Low level statistics

In the graph above, we can see that there is a correlation between the performance(i.e., number of pivots) and the number of instructions, data loads, and number of branches. However, all three quantities are quite related because the number of instructions affects the number of data loads and branches. Thus we could only consider the number of instructions. As we increase the number of pivots starting from one, the number of instructions starts to decline until we reach a point of 15 pivots, after which the number of instructions starts to grow gradually, which leads to worse performance. This tells us the same thing which was discovered here [2] "as we increase the number of pivots, we reduce the number of cache misses and thus obtain better performance." It might not be obvious, but our tests give identical results since the number of data loads and cache misses correlate. Thus as we increase the number of pivots, the number of data loads decreases and thus decreases the number of cache misses.

But it is still not clear why exactly for some pivots the number of instructions decreases, and for some, it starts to increase. We have one assumption. We also counted the number of swaps and comparisons for our algorithms those quantities increase as we increase the number of pivots and they are more or less the same. But we didn't see anything interesting in this statistic. So our assumption is that after a certain pivot the number of swaps and comparisons reaches a point where it becomes quite inefficient to process.

Future work

There are still things we don't understand like even though for all pivots the number of comparisons and swaps increases, nonetheless for some pivots the number of instructions decreases, and for some, it increases. There should be something about it because somehow it affects the performance. Another thing is to try different implementations for general partitioning because right now it isn't that efficient. We are doing at most n comparisons and $2n$ swaps. Also, I was able to outperform **C** quicksort with single pivots. To reach that I switched from dynamic arrays to static for that I used template meta-programming technique. But interestingly if I increase the number of pivots the performance starts to worsen right away. To conclude, this topic is quite broad there are many things to try in order to get answers.

References

[1] Robert Sedgewick and Jon Bentley. "Quicksort is optimal". In: *http://www.cs.princeton.edu/ rs/talks/ QuicksortIsOptimal.pdf* (2002).

[2] Alejandro L´opez-Ortiz Shrinu Kushagra and J. Ian Munro. "Multi-Pivot Quicksort: Theory and Experiments". In: *https://iopscience.iop.org/article/10.1088/1757-899X/180/1/012051* (2013).

[3] Vladimir Yaroslavskiy. "Dual-pivot quicksort". In: *http://iaroslavski.narod.ru/quicksort/Dua* (2009).