Uppsala University

HIGH PERFORMANCE PROGRAMMING

Parallel Quicksort in C

Individual Project

Isak Voltaire Edoh

March 12, 2021

1 Introduction

Sorting is an extremely important task in computer science for two main reasons; it makes datasets more readable to humans and more importantly, it optimizes the performances of other algorithms, namely search and merge algorithms. For instance, using a sorted dataset is critical when implementing a search algorithm for fetching an item in a large dataset. If the item to be retrieved is the last element and the searching starts with the first element, it would take a very long time to find it in a large unsorted dataset. Since sorting is such a common and well-understood task there exists multiple high-performing sorting algorithms with different properties.

This brief report covers the implementation, optimization and evaluation of a parallel Quicksort algorithm in the C language.

2 Problem description

The task at hand consists of implementing a parallel Quicksort to sort arrays of various lengths N, containing randomized integers from 0 to 999. The performance of the parallelized Quicksort will be evaluated by measuring execution times and by comparison to the non-parallelized Quicksort (single-threaded).

3 Solution Method

Quicksort is a divide-and-conquer algorithm, meaning that it recursively breaks down a large unsorted dataset into smaller unsorted subdatasets which are easier to sort. These subdatasets are then combined to create the solution of the original sorting problem. By multithreading (with `pthreads`), the subdatasets are sorted in parallel, which makes for faster executions times

3.1 Implementation and Parallelization

The first step of implementing Quicksort is to pick an element called the *pivot* on which to partition the array. In this assignment, the pivot is computed as a median `pivot = min + (min+max)/2`, where `min` and `max` are the first and last element of the array respectively.

The next step is recursively partitioning the array. The array is reordered in a way so that all elements smaller than the pivot are placed in the so called left subarray, while the larger elements are placed in the right subarray. After the first partition is complete, a new thread is created for the left subarray which is then partitioned and sorted recursively again. Meanwhile in the original thread the right subarray is partitioned and sorted recursively.

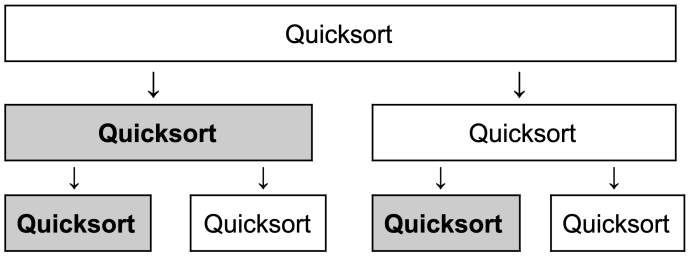


Figure 1: Parallelized Quicksort. The grey boxes represent threads being created.

As illustrated in figure 1, the process of selecting a pivot, partitioning and threading is repeated until the whole array is sorted. Lastly `pthread_join` is called to wait for the left subarray thread to finish and clean up its resources.

Calling the Quicksort function recursively causes many threads to be created which motivates the reason for setting a thread-creation threshold. Without a threshold the parallelization becomes too fine-grained and performance is negatively affected. Hence a threshold has been set to 10 000 such that arrays with lengths below 10 000 are sorted without multithreading.

3.2 Optimizing

In order to achieve an efficient and optimized Quicksort algorithm the code has been written to be as clear and readable for the GCC-compiler as possible. Since jumps and branches are expensive operations they have been limited when possible, this includes avoiding certain function calls and longer chains of if-statements. To make for faster memory access the use of local variables are also limited.

The memory for the array to be sorted was first allocated on the stack since heap allocation is more expensive. However when sorting lists over 2 million integers the stack overflows, causing segmentation fault. It was thus decided to use heap allocation instead.

Several GCC optimization options were attempted in order to reduce the execution time of the Quicksort algorithm. The wall time was `10.014` seconds when sorting an array of 3 million integers without any optimization flags. Table 1 lists the wall times for various flags, each measurement was done with the 3 million-integer array.

GCC Optimization Flags	**Execution Time (s)**	**Compile Time (s)**
`-O1 -pthread`	4.234	0.065
`-O2 -pthread`	1.690	0.079
`-O3 -pthread`	1.698	0.081
`-Ofast -pthread`	1.702	0.083
`-O2 -pthread -march=native`	1.605	0.086
`-O2 -pthread -march=native -ffast-math`	1.602	0.093

Table 1: Wall and compile timings for different GCC optimization options.

As evident in table 1 the fastest wall time is `1.602` seconds when using the `-O2` optimization level combined with the `-pthread -march=native -ffast-math` flags. Since the increase in compile time and code size are so small they are neglected. The program is only optimized for the sole purpose of reducing execution time.

4 Experiments

The evaluation of the parallelized Quicksort algorithm was done on a Macbook Pro 2017 with a 2,3 GHz Intel Core i5 processor (4 logical cores, 2 physical cores). The Quicksort algorithm was tested by inputting several large arrays with lengths N, filled with randomized integers from 0 to 999. To make sure that each test is random `srand(time(NULL))` is used which generates a random seed based on the current time. Each test is illustrated in table 2 below, including tests for the non-parallelized sequential Quicksort algorithm.

	Execution Time (s)	
N	Parallel Quicksort	Sequential Quicksort
100 000	0.004	0.004
500 000	0.053	0.083
1 000 000	0.192	0.317
2 000 000	0.729	1.370
3 000 000	1.606	2.755
4 000 000	2.836	4.860
5 000 000	4.406	7.586
6 000 000	6.367	10.778
7 000 000	8.673	14.657

Table 2: Wall times (s) for various unsorted arrays with sizes N.

With relative small arrays (N = 100 000) there is no speed-up by multithreading, suggesting that it does not make sense to use parallelized sorting algorithms for smaller arrays. But as N increases the parallelized Quicksort becomes 40% faster, which is a huge improvement. It would have been interesting to perform tests with even larger arrays but the program crashes when N $>$ 7 000 000 due to `Bus error: 10`.

The performances of both the parallelized and sequential Quicksort algorithms are also displayed in figure 2. The slope in the graph suggests that the time-complexity is $\mathcal{O}$(N logN).

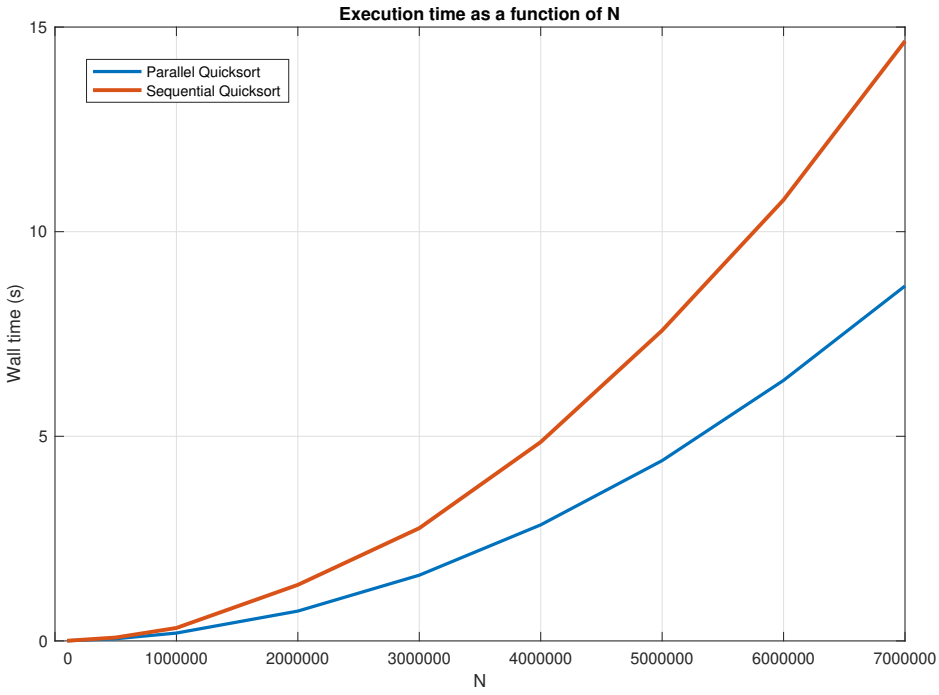
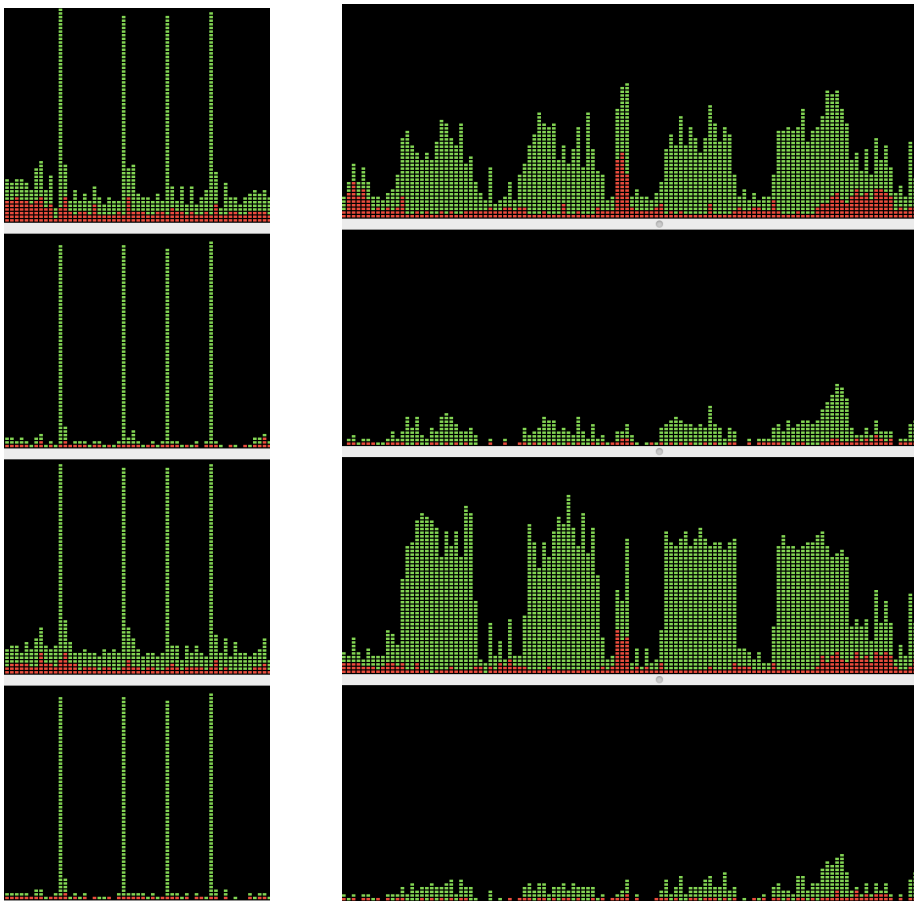


Figure 2: Wall times in (s) as a function of the input array length N.



(a) Parallelized (b) Non-parallelized

Figure 3: CPU graphs from the Activity Monitor application, the four windows represents the four logical cores. CPU usage is the y-axis and time is the x-axis.

An interesting observation can be made from the CPU graphs in figure 3 where each window represents a logical core. Both the parallelized and non-parallelized Quicksort are executed four times (N = 3000000), corresponding to the four spikes in CPU usage in both 3(a) and 3(b). The parallelized CPU graph 3(a) confirms that the algorithm really is running concurrently due to the maximum activity on all four cores. Calculating the CPU usage with `CPU usage = (CPU time / number of processors / wall time)` yields 98%. Meanwhile the CPU usage for the non-parallelized algorithm is only 25% which is also clearly visible in figure 3(b), two out of the four logical cores seem to be almost idle.

5 Conclusions

There is no doubt that there is an advantage of designing programs to make use parallelism for boosting performance. Parallelizing the Quicksort algorithm speeds up the sorting process by reducing the execution time with 40% compared to the non-parallelized algorithm. Although one have to be careful because too fine-grained parallelism causes performance issues since threads are expensive to create. This explains why a threshold was implemented to turn off the parallelism when the arrays become small enough. For an improved parallelized Quicksort one could test different values for the threshold to see which is the optimal. One could also test the algorithm on other machines with more than two physical cores to study the scalability.

To conclude, there is a 40 % speed-up when sorting larger arrays using parallelism, however there is no benefit in using parallelism for sorting smaller arrays.

6 References

[1] C. A. R. Hoare, Quicksort, The Computer Journal, Volume 5, Issue 1, 1962, Pages 10–16, `https://doi.org/10.1093/comjnl/5.1.10`

[2] GCC Documentation, `https://gcc.gnu.org/onlinedocs/`