Uppsala University

High Performance Programming

1TD062

Parallelization of Quicksort using OpenMP

Author:
Philip Ahl

March 22, 2019

Contents

1 Quicksort algorithm

Quicksort is an sorting algorithm with an average performance at O(nlogn). Quicksort is a Divide-and-conquer algorithm meaning that it uses recursive splitting in order to break the problem down to multiple sub-problems of smaller size until the sub-problems are small enough to be solved directly. The solution is then given by the the combination of all sub-problems.
Firstly quicksort picks an arbitrarily element to compare the rest of the array with, called pivot. In this implementation the first element in the array has been chosen as pivot. When comparing each element with the pivot it checks whether it is bigger or smaller, and divides the array into two sub-arrays, with the pivot in the middle and the elements smaller and bigger at each sides of it. Here an equal element is set to be in the array with values smaller than the pivot. This is the recursive split. These above explained steps are applied to the two sub-arrays.

1.1 Problem description

The main question of the problem is whether or not parallelization is applicable to quicksort, and in that case, on which grounds is it optimal? Is there a threshold where parallelization is unnecessary? How many threads are optimal? Are there any other speed-up-factors that could increase the computational speed on the sequential parts?
The parallization of choice is openmp, which is a widely used multi-threading option. Compared with another widely used parallelization method, pthreads, it is rather high-level.

2 Solution method

2.1 OpenMP

Openmp, short for Open specification for Multi Processing, is a parallelization model based on threads, which is an independent stream of instructions which can work parallel with each other, creating a multi-threaded parallelization. Openmp is a relatively high-level model which can automatically parallelize loops over all threads.

2.2 Implementation of OpenMP

The parallellization was implemented to the part which has the highest computational cost, the recursive quicksort-function. In the main method, omp single directive have been used for the first call of the quicksort-function, to tell the program that a single available thread is to be used in this part. In the quicksort-function the directive task was used for the recursive split. Here, the number of threads will be an input parameter to the user.

3 Specs

Intel(R) Core(TM) i7-2630QM CPU @ 2.00GHz
Ubuntu 18.04.1
gcc 7.3.0
Thread(s) per core: 2
Core(s) per socket: 4
Socket(s): 1

L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 6144K

4 Results

The following results documented in the tables below are ran on the computer with an Intel(R) Core(TM) i7-2630QM CPU @ 2.00GHz. All of the below results are for array size = 10^6 and element max size = 500.

The execution time for sequential quicksort is here plotted against a arbitrarily nlogn curve, meaning that its offset is not at focus, but rather the gradient of it. We can clearly see that the quicksort algorithm is steeper than the nlogn plot except for really small array sizes, which is not the case at scale we are interested in.

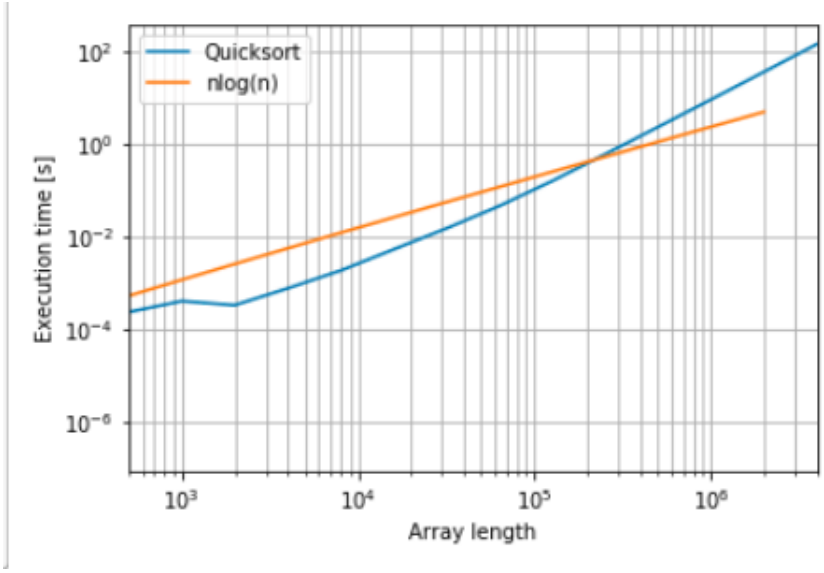

Figure 1: Sequential quicksort in comparison with its average performance nlogn

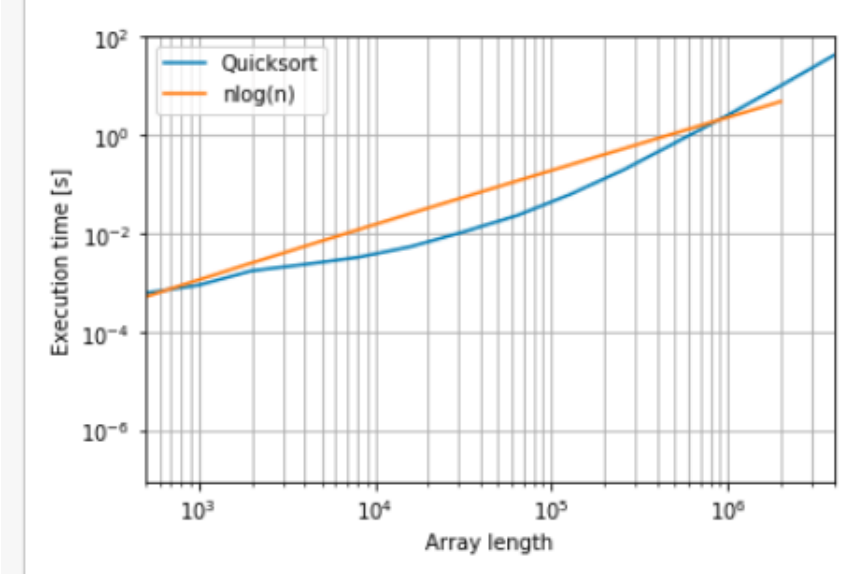

Figure 2: Parallelized quicksort with 4 threads in comparison with its average performance nlogn

The exectuion time for the parallel quicksort is comparably bigger for small array lengths but smaller for bigger arrays. This gives a rather less steep gradient compared to the sequential verison.

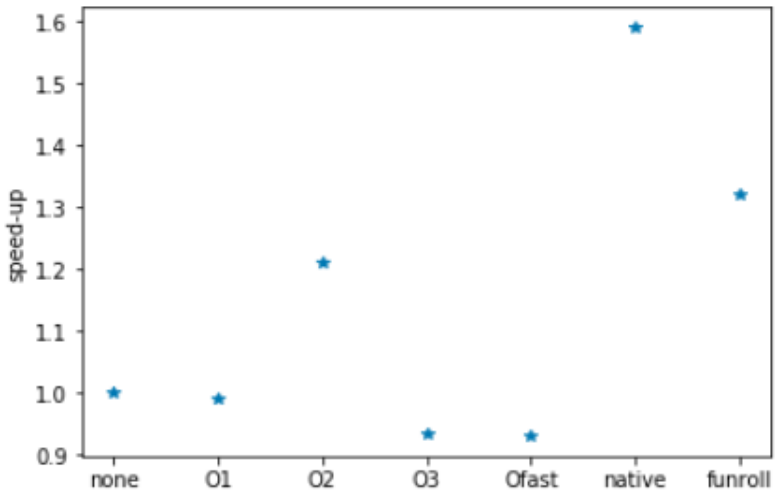

Figure 3: The speed-up ratio of sequential quicksort for different flag optimizations compared to no flag optimization

x = native and x = funroll is a short hand notation for the flag combination -O2 -march=native and -O2 -funroll-all-loops respectively. One can clearly see that -O2 -march=native is gives the highest speed-up.

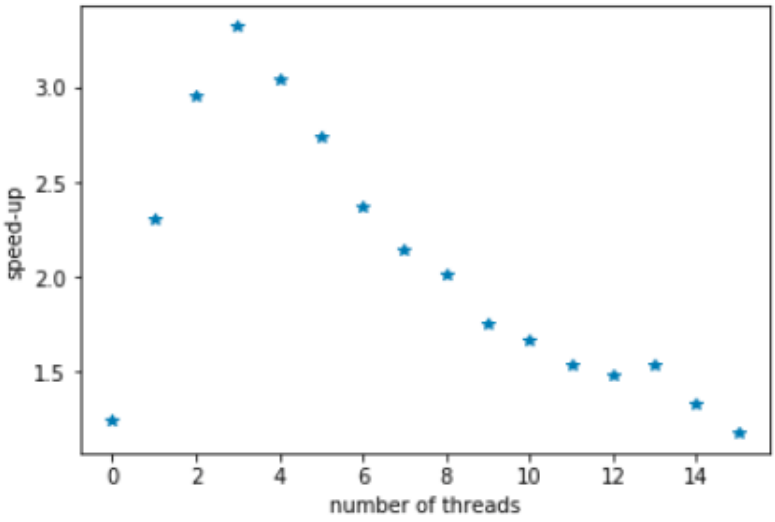

Figure 4: The speed-up ratio of parallelized quicksort for different amounts of threads compared to sequential quicksort.

The speed-up ratio of the parallelized quicksort is at maximum for 4 threads.

5 Discussion and conclusions

In the above results one can see in Fig.1 and Fig.2 that both implementations of quicksort, sequential and parallel, are close to nlogn. Since the algorithm is said to be bounded by O(nlogn) it is reasonable.

In Fig.3 one can see the speed-up ratios for different optimization flags compared to no flag optimization. The highest speed-up was given by -O2-march=native. This is reasonable since -O2 gave the highest speed-up of the single flags and -march=native is supposed to optimize for the given machine at hand.

In Fig.4 one can see the speed-up ratio of parallelized quicksort for different number of threads ranging from sequential (shown as 0 in figure) to 16 threads. There is a clear trend in the figure which shows a distinct peak at 4 threads. This is a reasonable result since the machine at hand have

4 physical cores in the CPU. One hypothesis was that 8 threads would be optimal since the CPU have two threads per core.

Analysing mispredictions with valgrind shows that the sequential quicksort with array size = 10^6 and max element size = 500 have a misprediction rate of 0.2% whereas 98% of that were in the part function. The same analysis in valgrind on the parallelized quicksort with same array size and max element size with four threads gives a misprediction rate of 0.3% whereas 73% of that were in the part function. Hence, a small amount of the predictions were faulty and the difference between sequential and parallel quicksort was unnotably.

Another valgrind analysis showed that the sequential quicksort gave no cache misses at all while parallel quicksort gave no errors but 7 blocks possibly lost and 5 blocks still reachable. None of which are crucual, since according to the valgrind manual, this status means that your program is probably ok, it did not free some memory it could have. Same conclusion can be drawn from this analysis.

Openmp and pthreads is two different parallelization paradigms. Openmp is high-level and more portable across platforms whilst pthreads is a low-level parallelization API which gives absolute control over the thread management. So the choice between them is rather the choice whether the designer wants or needs full control over the threads or if one wants to parallelize with as little work as possible.

As a last conclusion quicksort is much applicable to be parallelized and was optimal at 4 threads with the optimization flags -O2 -march=native which gave an absolute speed-up rate at 4.9. There are a lot of areas that could have been improved and investigated. Pthreads is an option that could give more control of the parallelization but if it would give a significant computational improvement is uncertain.
All the experiments could have been run on an Unix computer at university in order to get a broader view of the improvements.