Parallel In-Place Partitioning, Quickselect and Quicksort

Algorithm Engineering 2022 Project Paper

Konstantin Roppel
Friedrich Schiller University Jena
Germany
konstantin.roppel@uni-jena.de

ABSTRACT

Partitioning, Quickselect and Quicksort are algorithms that play an important role in a lot of use-cases, starting at the information extraction from scientific data sets all the way to presenting the sorted product items of an online shop to the customer in the way he desires. But as the amount of data in the world increases drastically, it becomes more and more important that these algorithms perform as fast as the underlying hardware allows them to. Because working on data sets that hold millions of elements can take a lot of time if your algorithm works serially. In this paper, I use fetch-and-add instructions as described in [1] to parallelize the algorithms Partitioning, Quickselect and Quicksort. To avoid a lot of incurred synchronization overhead, I process the items in a block-wise manner. For Partitioning and Quickselect, these parallelized versions can be used on hardware providing a lot of CPU cores to speed up the execution significantly, as underlined by the performed benchmarks. For Quicksort, I was not able to implement a performant, parallelized version.

KEYWORDS

Partitioning, Quickselect, Quicksort, Atomics (fetch-and-add), header-only library, templates

1 INTRODUCTION

Partitioning, Nth-Element-Selection and Sorting are algorithms that have been well-studied in the past, especially from an algorithmic point of view. There is a wide variety of sorting algorithms with different time and space complexity characteristics. You only have to choose which of them is the most fitting for your application. But there are use-cases in which it just is not enough to perform these algorithms serially, no matter how efficient the algorithm works. When working on sets that contain millions of elements, it is mandatory to utilize all hardware resources available to keep the execution time at a minimum.

1.1 Background

I will use the following definitions when talking about the algorithms mentioned above and their functionality:

- Partitioning reorders the elements in an array given pivot so that the pivot element resides at its position in the array as if it was sorted. Furthermore, all elements at positions with an index smaller than the pivot's index are smaller or equal to the pivot element itself, and all elements at positions with an index greater the pivot's index are greater than the pivot element itself.
- Nth-Element-Selection retrieves for a given N the element that would reside inside the array at the position with index N if it was sorted.
- Sorting reorders the elements in an array so that for each element e in the array it holds true that all elements at positions with an index smaller than the index of e are smaller or equal to the e itself, and all elements at positions with an index greater than the index of e are greater than e itself.

To synchronize accesses of multiple threads when working on an array, I use fetch-and-add instructions. A fetch-and-add instruction atomically fetches the value of an atomic variable that it currently holds, and then adds a given argument to the atomic variable. The atomic variable can i.e. be used to hold index information, and multiple threads can fetch and increment the value in a thread-safe manner. Lastly, I want to talk about pivot selection. Fast Quickselect and Quicksort rely on a good pivot selection, as that selection can not only influence the recursion depth of the respective function, but also the load distribution. Good pivots divide the set of elements into two partitions of nearly equal size. For my implementation, I chose the median-of-3 method. This method chooses three elements out of the vector as possible pivots, and selects the median of the three elements as the final pivot. For simplicity, my implementation chooses the first, the middle and the last vector element as possible pivots.

1.2 Related Work

Heidelberger, Norton and Robinson (1990) discuss an algorithm for multi-threaded Partitioning using fetch-and-add instructions to synchronize the threads. This is accomplished by holding global indices for the array access, which are fetched and incremented by each thread atomically. That way safe and sequential array accesses are ensured. They also discuss an approach to process the vector elements in batches to reduce the number of fetch-and-add instructions significantly. What they did not provide though is a proof-of-concept implementation and benchmarks for the respective algorithms. Only assumptions about the theoretical speedup are stated. [1]

1.3 My Contributions

This work is providing an implementation for in-place parallel Partitioning that is based on the serial algorithm discussed by Heidelberger, Norton and Robinson (1990). Using this Partitioning function, I additionally provide two implementations for in-place parallel Quickselect and Quicksort. The main problem of the serial algorithm presented by Heidelberger, Norton and Robinson (1990) is that it performs poorly on modern architectures. This is

caused by the poor cache locality as each thread grabs one single data element after another to process. Another problem is the high synchronization overhead that is caused by all the fetch-and-add operations that are needed to synchronize the access on the global array indices. To improve performance and aid these issues, this paper discusses the approach of processing the elements block-wise, similar to the follow-up approach discussed in [1]. This highly reduces synchronization overhead, as threads only need to execute a fetch-and-add each time they fetch a block of elements. It also increases cache locality because threads process blocks of consecutive vector elements.

1.4 Outline

The paper is organized as follows. Section 2 discusses the implementations of the algorithms that were introduced in section 1.1. Section 3 then presents the benchmark setup and its results. Finally, section 4 discusses the conclusions that can be drawn from the benchmark results and the limitations that I encountered for my specific implementations.

2 THE ALGORITHMS

2.1 Partitioning

Partitioning is the center piece of both Quickselect and Quicksort, as both of these algorithms apply it recursively to tackle their problem. Performant Partitioning is therefore a key requirement for fast Quickselect and Quicksort.

Input to the partitioning algorithm is a vector of length N as well as the block size and number of threads to be used. The pivot element used for partitioning is extracted as the last vector element. Given these inputs, the algorithm works as follows. All threads grab one block of elements from the left and one the right end of the vector. Three atomic index variables i, j and k are held to synchronize the vector accesses. When a thread fetches a block, it first performs a fetch-and-add on i to see if there are still blocks to be processed. If the value fetched from i is lower than the number of total blocks, then there are still blocks to process, otherwise not. It then performs a fetch-and-add on j if it needs to fetch a left block, or on k if it needs a right one. The return values of the fetch-and-add instructions on j and k indicate the block index that was fetched. With this block index, the thread can identify the elements on which the thread now has exclusive read/write access. Every block that is grabbed from the right is seen as a block that should only contain elements greater than the pivot, while every block that is grabbed from the left should only contain elements smaller or equal. Each thread then identifies elements to swap from their respective fetched blocks and swaps them. When a block is processed completely, the thread fetches the next block from the same side of the vector and continues the processing. This phase of the algorithm is finished if there are no more blocks of elements to be fetched. Now the clean-up phase begins. First, all blocks need to be identified that were not processed completely. For that, each thread checks if a block it fetched was not completely processed, and then adds the block to a list of blocks that still need further processing. There are two of those lists, one that is containing remaining blocks from the left side and one containing blocks from the right side of the vector. Now, left and right blocks can be processed just like in the processing phase before, by swapping elements that do not fulfill the partition criterion. When this is done, one of the lists must be processed completely, while the other one might still contain blocks that need to be processed further. To finish the partitioning operation, we also have to handle the remainder of the vector subdivided into blocks, as there is no guarantee that the vector length is a multiple of the block size. That means the part of the vector that still needs to be partitioned correctly is defined by the borders of that remainder and the outermost of the blocks that have not been processed completely yet. So all that needs to be done is performing partitioning on the partial vector defined by the just mentioned boundaries.

2.2 Quickselect

For Quickselect I chose a very simple design that uses the parallel Partitioning Implementation to retrieve the Kth element out of the input vector. The vector gets repeatedly partitioned and if the result index of the Partitioning is not equal to K, then that result index is the new upper or lower boundary for the new partial vector to be partitioned in the next iteration. If it is equal to K, then the pivot itself is the Kth element. To provide a well-suited pivot element, I implemented the median-of-3 and median-of-5 approaches, but found that median-of-3 was the more suitable one. As my Partitioning implementation chooses the last vector element as the pivot, my median-of-3 and median-of-5 functions respectively swap the pivot element that they found with the element at the last vector position, so it will be chosen as the Partitioning pivot.

2.3 Quicksort

My Quicksort implementation shows different behaviour based on the input vector size. For vectors with more than 1.000.000 elements it performs recursive parallel partitioning in combination with median-of-3 pivot selection making use of all available threads. When called on a vector containing less than 1.000.000 elements, but more than 100.000 elements, I create tasks that perform parallel Partitioning with a share of the available threads that is proportional to the workload of the task. Vectors with less than 100.000 elements are just sorted sequentially.

3 BENCHMARKS

3.1 Hardware and Setup

The benchmarks were performed on an A64 ARM compute node of the Future Technologies Partition of the National HPC Center, which is part of the KIT (Karlsruher Institut für Technologie). The machine features a Fujitsu A64FX processor that consists of 48 cores that each clock at 1.8 GHz and works with the ARMv8.2-A+SVE instruction set. The compute node contains 32 GB of main memory.

For the benchmarks, vectors with different lengths were created by generating uniformly distributed integer values between zero and vector.size(). To make up for variations in the results, e.g. due to varying vector contents and their influence on the algorithms' performance, each benchmark was performed three times for random input and the median value of the three runs was chosen as the benchmark result. For every run, all benchmarked implementations worked on identical copies of the same vector, so that the results are comparable. The benchmarks were performed for my Partitioning,

Quickselect and Quicksort implementations with the number of threads set to 48 and the block size set to vector.size()*0.00005. As reference measurements the benchmarks were also performed for the single-threaded std::partition, std::nth_element and std::sort functions as well as their parallel counterparts. The parallel versions of these algorithms can be explicitly obtained by referencing them through the namespace __gnu_parallel.

3.2 Results

Figures 1 to 3 show the result of the benchmark. It can be seen that the implementations for Partitioning and Quickselect perform well on the provided hardware for a workload that is large enough. For smaller vector sizes (below 1.000.000 elements) they perform slightly worse than the reference implementations, but as the vector size increases they show their advantage over serial execution. They can utilize the power of the 48 core processor well and even slightly outperform the parallel version of GNU's standard sort implementation. For Quicksort on the other hand, my implementation shows almost no performance gain over the serial std::sort implementation.

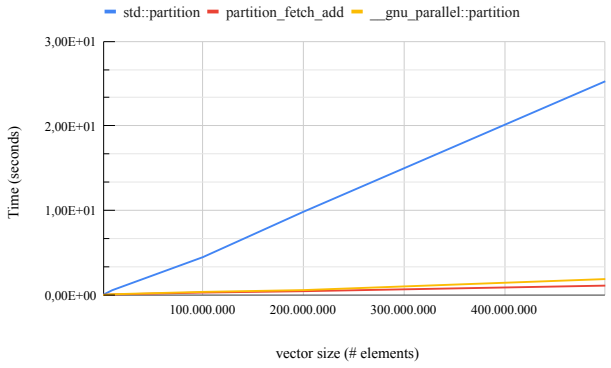

Figure 1: Benchmark results for the three Partitioning implementations.

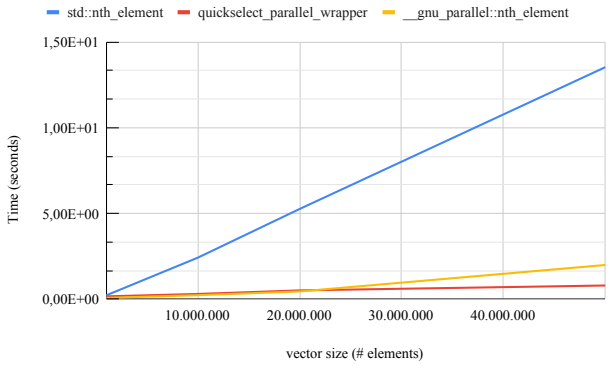

Figure 2: Benchmark results for the three Quickselect implementations.n

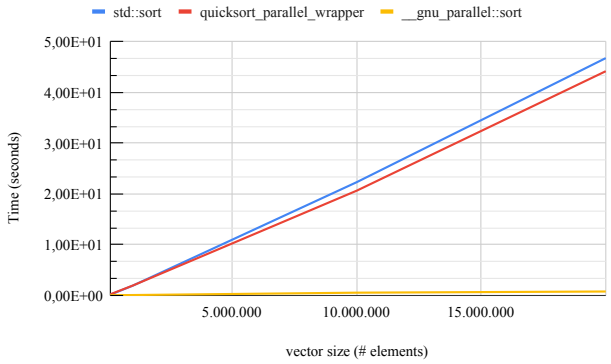

Figure 3: Benchmark results for the three Quicksort implementations.

4 CONCLUSIONS

For the operations Partitioning and Quickselect, I managed to implement algorithms that can make effective use of lots of processing units to accelerate the processing of large vectors. For Quicksort, on the other hand, my implementation provides nearly no improvement over serial execution. One problem that I see is that my parallel Partitioning implementation does not perform well for small vectors that have less than 1.000.000 elements. Also the median-of-3 pivot selection can become more of a performance bottleneck than an enhancement for small vectors. Both of these factors cause my Quicksort implementation to be very inefficient, as there are a lot of those Partitioning operations on small vectors at the bottom of the recursion levels, while there are only few Partitioning operations on large vectors at the top level of the recursion.

ACKNOWLEDGEMENTS

This work was performed on the NHR@KIT Future Technologies Partition testbed funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research.

REFERENCES

[1] Philip Heidelberger, Alan Norton, and John T. Robinson. 2016. Parallel Quicksort Using Fetch-and-Add. In *IEEE TRANSACTIONS ON COMPUTERS, VOL. 39, NO. 1, JANUARY 1990*. IEEE.