Implementation of a partitioner according to Tsigras and Zhang

Algorithm Engineering LAB 2021 Project Paper

Brian Zahoransky
Friedrich Schiller University Jena
Germany
brian.zahoransky@uni-jena.de

ABSTRACT

Many applications depend on well-performing sorting algorithms. Therefore, Philipas Tsigas and Yi Zhang have developed a parallel quicksort for cache-coherent shared address space multiprocessors. [4] For this paper, the parallel partitioner was extracted and implemented first. The C++ header-only library uses lock-free parallel regions realized with openMP and atomic data types. After that, the parallel quicksort was completed, and a parallel quickselect was implemented, too. The partitioner was applied in both algorithms. All three algorithms achieved a significant speedup compared to the single-threaded algorithms of the std library. However, this implementation still needs improvement to reach the same level as the gnu-parallel library performing between 20 and 40 per cent better in the executed tests.

KEYWORDS

partitioning, sorting, parallel, C++, openMP

1 INTRODUCTION

As shown in the paper of Philipas Tsigas and Yi Zhang, sorting is a relevant topic in today's application of computers.[4] The authors also discussed the advantages of quicksort: It is fast, in-place, simple to implement, and exhibits good cache performance. Then, they specified the system class for which the invented algorithm is suitable: cache-coherent shared address space multiprocessors. In their paper, the scientists provided a detailed description how a multi-threaded quicksort could be implemented. The algorithm is characterized by three major techniques:

- *"Cache-efficient: Each processor tries to use all keys when sequentially passing through the keys of a cached-block from the key array."*[4]
- *"Communication Overlapping Fine-grain Parallelism: It is a fine-grain parallel algorithm. Although fine-grain parallelism has been thought to be inefficient for computations like sorting due to the synchronization overheads, we achieved efficiency by incorporating non-blocking techniques for sharing data and computation tasks. No mutual locks or semaphores are used in our implementation."*[4]
- *"Parallel Partition of Data: A parallel technique for partitioning the data similar to the one presented in [2] is used. We rediscovered this technique when parallelizing quicksort."*[4]

In section 2, the algorithm is discussed in detail. However, in the experimental results chapter, they explained that sample sort was supposed to be one of the best performing sorting algorithms on multiprocessor machines. Despite that, the algorithm of Tsigas and Zhang outperformed sample sort. The algorithm was tested on a system with up to 32 cores; especially when all these 32 cores were used, quicksort showed a remarkably higher speed up than sample sort. As an explanation for their success, the scientists emphasized, again, the three applied programming techniques.

The goal of this project is to implement a partitioner, quicksort, and quickselect based on the findings of Philipas Tsigas and Yi Zhang. The algorithms will be written in C++. Further information about implementing portable sorting algorithms was found on stackoverflow.com [5]. The quickselect [7] and the insertion sort [6] is based on the pseudo-code from Wikipedia. To create a parallel, lock-free library, openMP and atomic data types were applied. In the library are two algorithms not discussed in this paper: an iterative quickselect [3] and a dual-pivot [1] quicksort. These algorithms have performed worse than the algorithms considered here. A serial version of both is provided on geeksforgeeks.com.

2 THE ALGORITHM

As mentioned, the algorithm is based on the paper of Philippas Tsigas and Yi Zhang.[4] The following explains how the algorithm was implemented in C++, also using openMP and atomic data types. First, the algorithm of Tsigas and Zhang was split into an independent partitioning algorithm and a quicksort algorithm invoking the partitioner. That allowed applying the partitioner by itself as well as within the quickselect and quicksort algorithm. The algorithm works blockwise. Optionally, the block size should be chosen that exactly two blocks fit into L1-cache.

2.1 The partitioner

The original partitioner has two phases, a parallel and a sequential one. For the implementation, the sequential phase was split into a sequential neutralization, a sequential swapping, and a sequential partitioning. Within the header-only library, the "ppartition" function combines all these parts to the final parallel partitioner. Now, have a closer look at these phases.

The **parallel or parallel partition phase** (as named in the original paper) partitions the input array blockwise. It is shown in figure 1. A task picks the most left and the most right block available. Available blocks are those that were not finished and not taken by some other task. Then, the algorithm invokes the neutralization function. This function neutralizes at least one of the two blocks. In other words, the procedure goes through both blocks. It searches for elements not fulfilling the predicate in the left block and for those fulfilling the predicate in the right block. When the algorithm has found such two elements, it exchanges them. In the end, it returns whether the left, the right or both blocks were neutralized. Neutralized blocks do not contain elements that

would be exchanged by the neutralizing function. After this step, the task picks a new left block and/or right block corresponding to the neutralized blocks and starts again. When no more blocks remain, each task writes the first index of its last remaining block in the intended array. After finishing, this phase hands over an array with the start indices of remaining blocks, the number of remaining blocks, the number of partitioned left-side (LN) and right-side elements (RN). Further details are in the original paper. It also includes a pseudo-code description of this phase differing marginally from the implemented version.

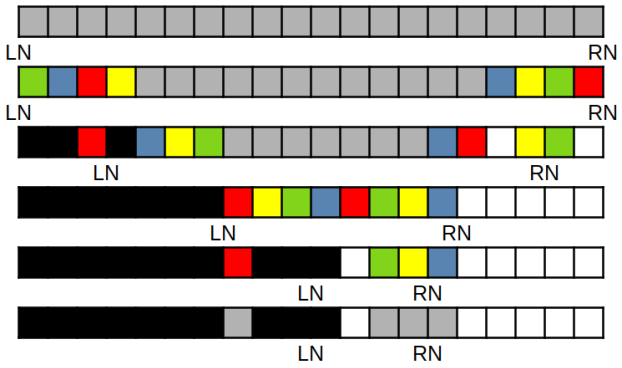


Figure 1: The parallel phase (an example). The remaining blocks are coloured grey, left-side blocks black, and right-side blocks are white. The other colours show that a processor works on it at the moment. Each colour represents one task. LN and RN are the left-side and right-side element counters.

The paper algorithm uses core dependent threads instead of tasks. In previous versions, the implemented algorithm realized it in the same way. However, the implemented version uses the partitioner as own module, which is also invoked by the quicksort algorithm. Since the quicksort launches already a parallel region, it causes nested parallel regions. These nested parallel regions led to compiler compatibility issues. Due to this, the parallel-phase-run function was introduced. Thus, a new parallel region within the partitioner will launch only if no parallel region has started until now. The algorithm ensures that the number of threads does not exceed the number of logical cores. (Actually, the "OMP_NUM_THREADS" environment variable limits the threads to be created. To achieve the best performance, the variable should be set correspondingly.)

The **sequential neutralization** phase, shown in figure 2, partitions the remaining blocks single-threaded. Therefore, it first sorts the array of remaining blocks in ascendance order. Now, it proceeds in the same manner as the parallel phase. It picks the first and last remaining block, neutralizes at least one of them, and picks either a new block from the left side and/or one from the right side. When this phase ends, there will remain only one block unless the algorithm finishes perfect, having neutralized all blocks. Note that maybe not all blocks will be in the right place. Left-side blocks neutralized out of range of the left-side element counter (LN) do not increase it. The same rule is applied for neutralized right blocks out of the range of the right-side element counter (RN). These elements

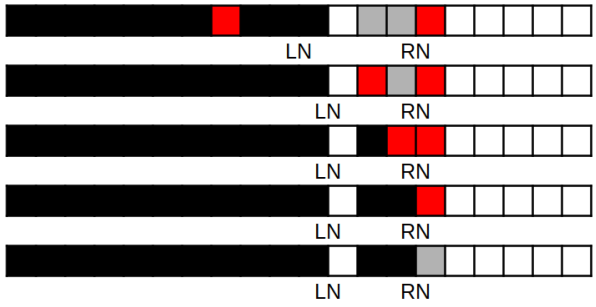


Figure 2: The sequential neutralization (an example). Input is the last bar of the parallel phase (figure 1). Note, the two classified black (left-side) blocks in the middle are also remaining blocks for the sequential swapping (figure 3).

will be held in the remaining blocks array. In summary, this phase ends with all blocks except maximal one have been classified. The array of remaining blocks stores all blocks being not in the right place, and two additional right/left index memorizer show which of the previous remaining blocks were transformed into left-side and right-side blocks.

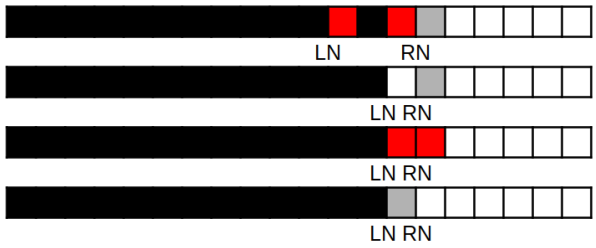


Figure 3: The sequential swapping (example). Input is the last bar of the sequential neutralization (figure 2). Note that the grey and the black (left-side) blocks are the remaining blocks. The right-side element counter (RN) is in its final position. The black (left-side) blocks will be swapped in their final position beginning with the rightmost one.

The **sequential swapping** phase, presented in figure 3, brings the remaining blocks into the right place. The last phase has also led to either the left-side (LN) or the right-side element counter (RN) being in its final position. However, in the range of this first counter may be blocks from the second side. The blocks in the range of the second counter are all in the right position. Additionally, the block next to the range of the second counter belongs to the domain of the first counter. This state is restored after each block-swap. Due to this, before every block-swap is clear, which blocks will be exchanged next. To minimize the number of swaps, the right-side blocks will be put into position beginning with the leftmost one, and the left-side blocks will begin with the rightmost one. After swapping all remaining left-side and right-side blocks in their final position, the left-side (LN) and right-side element counter (RN) are in their final position. Finally, the last unpartitioned block, if existing, is swapped into the right position.

Figure 4: The sequential partitioning. The last remaining block is partitioned sequentially.

The **sequential partitioning** phase, visualized in figure 4, partitions the last remaining block if existing. In the paper, a sequential quicksort was used for this task. To optimize the runtime of the partitioner, a sequential partitioner was applied in this implementation. The sequential partitioner is described in the following. First, a for-loop runs through the remaining block and counts all entries fulfilling the predicate. Now, the block can be split. The left subblock ends after the number of elements registered by the counter. There starts the second subblock. These two parts are the input for the neutralize function partitioning the block. Thus, the phase can return a pointer to the first element of the second subblock. This element also partitions the whole input array and can be returned by the parallel partitioner.

2.2 Quicksort

The quicksort function is per default single-threaded but can be executed parallel using the "pquicksort" function. The "pquicksort" function opens a parallel region and starts the quicksort function while applying the omp single directive. First, the quicksort algorithm measures the distance between the first and last element of the input array. If the distance is beneath 32, the insertion sort algorithm is applied to be faster for small arrays. Otherwise, the quicksort algorithm proceeds. Second, the median is chosen. Therefore, the median of the first, middle and last element is determined. Third, the array is partitioned with the partitioning algorithm described above. To avoid that quicksort gets stuck, the second half is partitioned again, sorting all elements equal to the pivot at the beginning. Fourth, when enough processors available, they are split corresponding to the remaining work. Fifth, two quicksort tasks are launched if the subarrays have a size of over 10,000 elements. Thus, starting small tasks having too much overhead will be avoided.

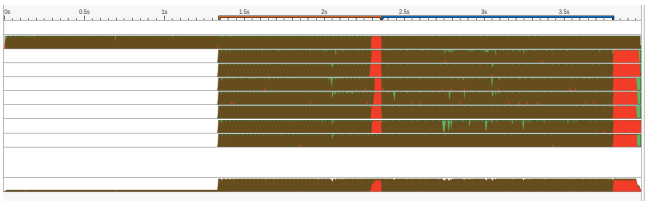
2.3 Quickselect

The "pquicksort" function works similarly to the quicksort function with minor differences. It needs no starter function since there is only one recursive call at the end. First, it checks if the first and last element pointers are pointing to the same address. When that is the case, the function ends. Elsewhere, the quick select algorithm proceeds. Second, the median is chosen. Therefore, the median of the first, middle and last element is determined. Third, the array is partitioned in parallel with the partitioning algorithm described above. To avoid that quickselect gets stuck, the second half is partitioned again, sorting all elements equal to the pivot at the beginning. Fourth, the function invokes itself: either on the first or on the second part of the array depending on the index of the nth element.

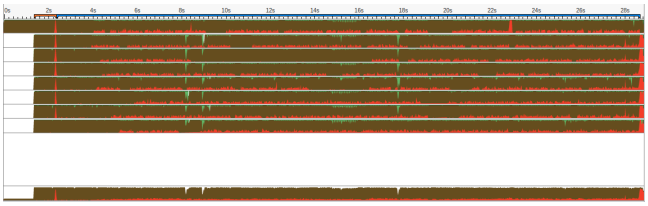
3 EXPERIMENTS

The experiments were executed on a laptop running ubuntu 20.04. with 16 GB of RAM and an Intel Core i7-4800MQ CPU. The processor has four physical cores and supports Intel hyper-threading, which leads to eight logical cores. It has a 16 KB L1-cache. That implies an optimal block size of 4096 elements if only the physical cores are used. However, the first tests showed that the program runs best with eight threads. Thus, a block size of 2048 was chosen for the following experiments.

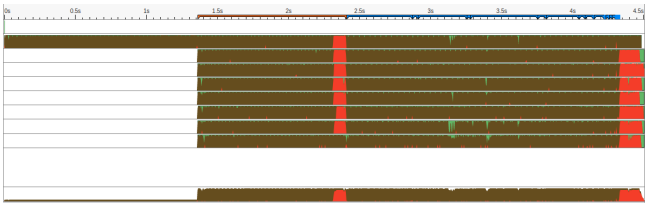
3.1 Parallelism



(a) The parallel partitioner



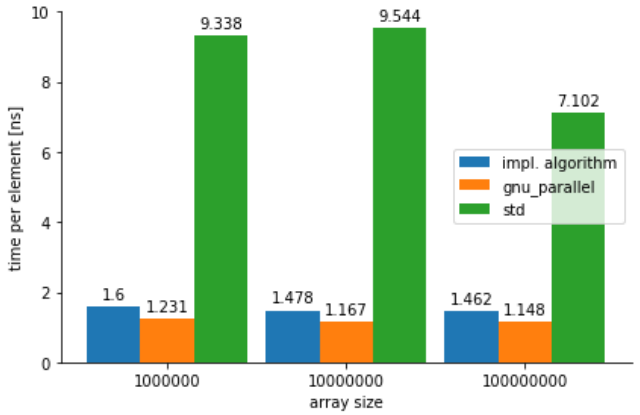
(b) The parallel quicksort



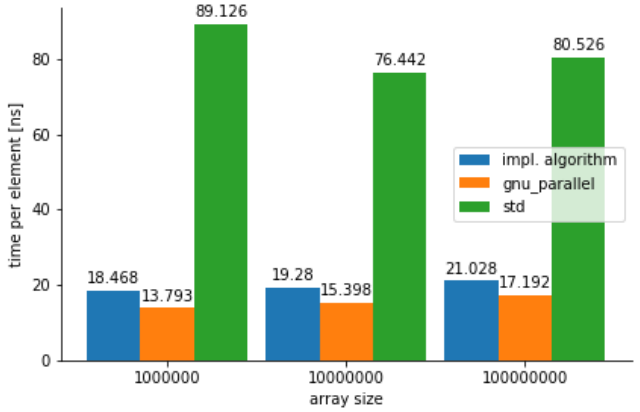
(c) The parallel quickselect

Figure 5: Intel Vtune threading analyses of the parallel algorithms.

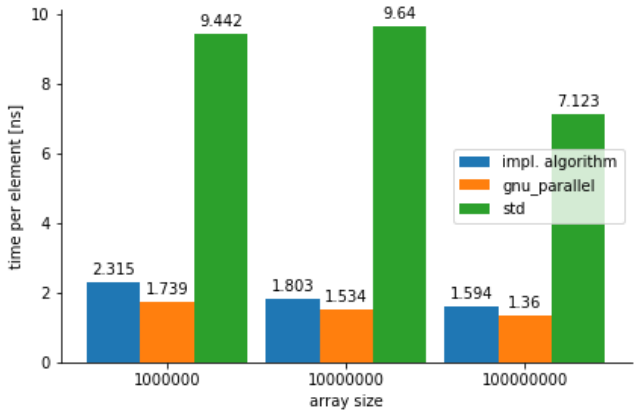
In Figure 5, the parallel runtime was analyzed with Intel Vtune. The eight bars at the top represent the eight threads. The bar below shows the overall CPU use over time. There are three colours: A thread is green when started, brown means actual computing time, and red stands for overhead. Every graph starts with a single-threaded initialization phase evaluating the input parameters. Then, the test vector is generated in parallel, causing the start of the other threads. Since it is computing time, the graph becomes mainly brown. The following red area can be explained with the joining phase of openMP. After, the parallel algorithms are invoked. Every chart is mostly brown, meaning the program runs in parallel. All algorithms end with overhead from the openMP joining routine.



(a) Comparison of the partitioners



(b) Comparison of quicksort, std::sort, and __gnu_parallel::sort



(c) Comparison of quickselect and the corresponding nth_element algorithms

Figure 6: Comparison between the implemented algorithms, the C++ std counterparts and the corresponding __gnu_parallel realizations. Every bar of each graph is a mean value per iteration after running the algorithms 100 times.

3.2 Benchmarks

Figure 6 shows the meantime each element was worked on until the wanted result was achieved. Each algorithm ran for 100 iterations. The elements were uniform distributed between the maximal and minimal 32-bit integer values.

When considering the partitioners, it can be seen that the parallel algorithms slightly prefer larger arrays. The std-partitioner increases its speed noticeably when changing the array size from 10,000,000 to 100,000,000 elements. The implemented version is between 25 and 30 per cent slower than the gnu-parallel algorithm. Despite that, it is remarkably faster than the single-threaded version. The speedup for 10'000'000 elements is around 6.5 and for 100'000'000 elements about 4.5.

Next, the selecting algorithms are discussed since there can be observed similar effects. After that, follows the discussion about the sorting algorithms. Here, similar effects can be observed. Compared to the partitioners, the parallel selecting algorithms can decrease their needed time per element even more when the array size grows. Furthermore, the implemented algorithm was able to reduce the difference to the gnu-parallel version. For 1,000,000 elements, it needed 35 per cent more time per element, whereas, for 100,000,000, it was only 20 per cent slower. The speedup of the parallel quickselect in comparison to std-select is nearly the same as the speedup for the partitioner.

Finally, sorting algorithms are discussed. For the parallel version can be seen that the needed time per element grows together with the array size. However, the std-sort improves its results for 10,000,000 elements and falls a bit back for 100,000,000 elements. While 40 the parallel quicksort needs 40 per cent more time per element than the gnu-parallel-sort for 1,000,000 elements, it is only 25 per cent slower for 100,000,000 elements. The speedup is between 4 and 5.

4 CONCLUSIONS

It has been shown that the applied approach can lead to a significant speed-up compared to the single-threaded std library. On the other hand, the gnu-parallel library performed better in the test cases. The quoted paper, however, provides further advises to speed up the algorithm.[4] For example, another version of quicksort was applied while each processor works on its own.

REFERENCES

[1] Shlomi Elhaiani. 2021. Dual pivot Quicksort. geeksforgeeks.org. https://www.geeksforgeeks.org/dual-pivot-quicksort/
[2] Heidelberger, Norton, and Robinson. 1990. Parallel quicksort using Fetch-and-Add. In *2016 IEEE Transactions on Computers*. IEEE.
[3] kartik. 2021. QuickSelect (A Simple Iterative Implementation). geeksforgeeks.org. https://www.geeksforgeeks.org/quickselect-a-simple-iterative-implementation/
[4] Philipas Tsigas and Yi Zhang. 2003. A Simple, Fast Parallel Implementation of Quicksort and its Performance. IEEE.
[5] unkown. 2021. How to implement classic sorting algorithms in modern C++? stackoverflow.com. https://stackoverflow.com/questions/24650626/how-to-implement-classic-sorting-algorithms-in-modern-c
[6] unkown. 2021. Insertion sort. Wikipedia.org. https://en.wikipedia.org/wiki/Insertion_sort#Algorithm
[7] unkown. 2021. Quickselect. Wikipedia.org. https://en.wikipedia.org/wiki/Quickselect#Algorithm