Parallelization of Quicksort algorithm on CPU

1st Aya Ali Al Zayat
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
aalialzaya1@etf.unsa.ba

2nd Asim Ćeman
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
aceman1@etf.unsa.ba

3rd Adna Halilović
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
ahalilovi15@etf.unsa.ba

4th Vahidin Hasić
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
vhasic1@etf.unsa.ba

***Abstract*—For some time sorting has had a large part in computing and has been the subject of various research, as it is widely used to reduce the complexity of algorithms and procedures. This paper presents a program solution that implements the Quicksort algorithm using parallelization techniques and SIMD vectorization using AVX2 instructions. The performance of this implementation is compared to the qsort function from the C++ algorithm library, the sequential implementation of Quicksort with pivot chosen by the "median-of-three" rule, the sequential implementation using AVX2 instructions, as well as versions of parallel implementations using sections and the "median-of-three" rule. The results show that our implementation has a better performance for larger array sizes and equal performance for smaller array sizes.**

***Index Terms*—Quicksort, SIMD, AVX2, vectorization, sort, OpenMP**

I. Introduction

Sorting elements of an array is a fundamental problem in computing and everyday life. Not obvious examples where sorting is important are combinatorial optimization, astrophysics, molecular dynamics, linguistics, genomics and weather forecasting. Sorting is also used in databases, in statistical software to estimate distributions. Sorting allows binary search, simplifies problems such as checking unique elements of the array and finding the nearest pair of point in the plane so basically sorting optimization optimizes these problems as well.

The Quicksort algorithm was developed by the British scientist Charles Antony Richard Hoare in 1959 [5] and it is one of the most intensively studied problems in computing. The Quicksort algorithm is most often used for sorting today because it is one of the fastest sorting algorithms, and therefore strives for its additional optimization, which is the aim of this paper. Of course, for different needs and different data to be sorted, different algorithms are chosen. In the case of the Quicksort algorithm, in most cases it is faster than the others "Divide and conquer" sorting algorithms unless the elements are almost pre-sorted or fully sorted or when the elements are quite similar.

The rest of the paper is organized as follows: An overview of the Quicksort algorithm is given in Section II. Section III presents the related works. In Section IV the qsort version, sequential versions of the algorithm, and the parallel implementation versions of the Quicksort algorithm are presented. Section V contains a comparative analysis and results from all the algorithms presented in the previous section and the final Section VI contains the conclusion.

II. Quicksort algorithm

The Quicksort algorithm uses the "divide and conquer" strategy. The algorithm divides the input array into two parts by reorganizing that array so that all the elements in the first part are smaller than all the elements in the second part of the array. After the reorganization, those two parts are unsorted and separated by an element called a pivot.

Since the pivot element is extremely important when sorting, finding the pivot represents one of the key problems of this algorithm. The choice of pivot element is possible on different ways. For example, the pivot can be chosen by taking the first element of an array, although there are more efficient ways to select it. Ideally, the pivot element should divide the array into two subsets with the same number of elements. Because of the way the input array is reorganized, it can be concluded that the pivot element is positioned to its final position, and will no longer move in the further procedure. Furthermore, it can be concluded that sorting can be continued by applying the same algorithm recursively to the obtained two parts of the input sequence. The recursive process continues until the resulting parts have only one element. Completion of the recursive procedure yields a sorted array.

Mathematical analyzes of Quicksort show that on average the algorithm has complexity $\mathcal{O}(n \log n)$, while in the worst case it has complexity $\mathcal{O}(n^2)$. The Quicksort algorithm achieves the best results when the initial elements are uniformly random deployed. In this way, the array is divided into subsets of approximately equal number of elements.

If the pivot is not well chosen, there is a danger of the worst case for Quicksort. It is a situation where the initial array is

already sorted (or almost sorted), or when the elements are the same (or there is a lot of equal elements). Then it happens that the subarrays can not be "nicely" divided and the array constantly has a part that contains one element and a part that contains all the other elements. In this situation, this algorithm can be very slow.

For arranging elements in relation to the pivot, the Hoare method [5] is most often used to partition a given array. Hoare uses two indices that start at the ends of the array being partitioned, then move toward each other until they detect an inversion: a pair of elements, one greater than the pivot, one smaller, in the wrong order relative to each other. The inverted elements are then swapped. When the indices meet, the algorithm stops and returns the final index.

III. Related works

In 2015, Shay Gueron and Vlad Krasnov [4] present a version of the Quicksort algorithm which uses special shuffle masks generated by a *lookup* table to accelerate the sort operation. After finding the pivot element, the algorithm compares each element with the pivot element, and these comparisons are independent of each other. Thanks to that, SIMD instructions can be used for the purpose of comparing elements with the pivot element because SIMD instructions perform, simultaneously, the same operation on several elements that are stored in the instructions operands (registers or memory locations) [6]. On the other hand, dividing an array into two parts requires branching for each element. The problem was solved by using the aforementioned masks, and they serve to separate the elements into two parts. The difference in our algorithm is is that we use only one mask.

In 2017, Berenger Bramas [2] presents a Quicksort algorithm based on the use of AVX-512 instructions [7]. The AVX-512 is an instruction set that appeared in 2016 for Intel KNL processor units. It offers operations that do not have their equivalents in previous editions of the x86 instruction set. For the implementation of the Quicksort algorithm using AVX-512 instructions, very important functions used in the paper are: *load/set/store* and *store some* instructions. As in the instruction sets before the AVX-512, the *load/set/store* instruction loads a block of values from memory, populates the SIMD vector with that array, and writes the SIMD vector back into memory. As for the *store some* operation, it is a new operation, characteristic for this set of instructions. It allows only certain SIMD vector values to be stored in memory. This is a big improvement because without this function, more operations were required for such actions, which greatly slows down the operation of the algorithm itself.

The greatest improvement is achieved by using the *store some* function, especially in the part of determining the pivot element. This operation allows SIMD vector values that are less/larger than the pivot to be stored directly inside the memory. The main idea of partitioning in this algorithm is to load values from the array and store them directly in memory using two iterators, to the right and left of the pivot element. For this reason, the implementation is run in one place and only 3 SIMD vectors are needed for execution.

Another improvement is achieved by using the so-called Bitonic-Based Sort that uses a sorting network. Bitonic sort is a comparison-based sorting algorithm that can be run in parallel. It focuses on converting a random sequence of numbers into a bitonic sequence, one that monotonically increases, then decreases. The initial unsorted sequence enters through input pipes, where a series of comparators switch two entries to be in either increasing or decreasing order [1]. The basic idea of using graphs when creating a sorting network is the possibility of better selection of values to be compared as opposed to comparing neighboring values. This significantly reduces the number of instructions required to execute the algorithm.

In 2017, Tinku Singh, Durgesh Kumar Srivastava, and Alok Aggarwal [8] use multi-core processors to implement the Quicksort algorithm. The reason for using multi-core processors is that the overall performance of a multi-core processor is much better compared to a single-core processor, although the actual speed of each core of a multi-core processor is slower than a single-core processor. In order to take full advantage of the core of a multi-core processor, multi-threaded or parallel processing techniques are required. The sorting is performed serially and in parallel for a large number of data elements In the parallel implementation of the algorithm, it is assumed that the system has distributed memory. An unsorted list is distributed using some thread distribution approach. The first element is taken as the pivot element. The process of splitting into two subsets is divided into 2 subprocesses (threads).

IV. Implementation

The library function *qsort* in subsection IV-B and two other sequential implementations are used for a performance comparison to our parallel implementations.

A. Identified parts for optimization

A major improvement of the algorithm compared to the standard implementation can be achieved by taking a random element from the sub array for the pivot element instead of the first element. This can reduce the time required to execute the worst case, which is $\mathcal{O}(n^2)$, to the time required to execute the average case, which is $\mathcal{O}(n \log n)$, specifically $\mathcal{O}(1.386\ n \log n)$. Another strategy that improves the time complexity to $\mathcal{O}(1.188\ n \log n)$ is using the "median-of-three" rule which comprises of choosing the median of the first, middle and last element of the partition for the pivot.

Part of the algorithm for selecting the pivot element is suitable for the use of SIMD operations, because each element can be compared with the pivot element, independently of other elements. This allows parallel comparison on multiple ALUs per core, which speeds up the movement of elements in the partition function.

Since Quicksort is a recursive algorithm because it calls recursively over the top and bottom of the array, and since these recursive calls are completely independent of each other, this part of the algorithm is suitable for parallel execution on multiple threads.

B. qsort library function

The qsort function is defined in C++ algorithm library. This function sorts the given array in ascending order. The array contains count elements of size bytes. Function pointed to by comp is used for object comparison. The qsort function is an optimized version of the Quicksort algorithm that reduces the worst case execution time to the average case execution time.

The complexity for worst case and average case of qsort function is $\mathcal{O}(n \log n)$.

C. Sequential implementations

There are two sequential implementations of Quicksort algorithm chosen and implemented for comparison in this paper, and those include:

- Implementation which uses the "median-of-three" rule for finding the pivot
- Implementation using AVX2 instructions

1) Sequential implementation with using "median-of-three": This sequential implementation of Quicksort algorithm uses the "median-of-three" rule for finding the pivot element. This strategy comprises of choosing the median of the first, middle and last element of the partition for the pivot and has the time complexity of $\mathcal{O}(1.188\ n \log n)$. The code for choosing the pivot using this rule is presented below.

```
template<typename Tip>
int partition_medianOfThreePivot(Tip
*array,int low, int high) {
    int mid = (low + high) / 2;
    if (array[mid] < array[low]) {
        std::swap(array[low], array[mid]);
    }
    if (array[high] < array[low]) {
        std::swap(array[low], array[high]);
    }
    if (array[mid] < array[high]) {
        std::swap(array[mid], array[high]);
    }
    return standardPartition(array, low, high);
}
```

The core part of the algorithm remains the same for this version as well as all the other implemented versions with recursive calls to both partitions formed in regards to the pivot.

2) Sequential implementation with AVX2: This implementation uses AVX2 instructions to further optimize the most time consuming part of Quicksort algorithm - choosing the pivot element.

The underlying idea of this implementation is to vectorize the Quicksort algorithm using the AVX2 instruction set. AVX2 256-bit integer masks are used to compare and exchange units following the *hit-or-miss* strategy. The *hit-or-miss* strategy is used when choosing the right mask to select elements that need to be moved according to their relation to the pivot element. One static (hard-coded) mask is used for this purpose. This mask sets the indices of elements lesser than the pivot to -1, so that they can be shuffled and sorted in the ascending order.

Because only one mask is used to select the array elements lesser than the pivot and shuffle them afterwards, the elements may not be sorted in the correct order. These kinds of mistakes are called *misses*, as the masking and shuffling doesn't sort the elements in the correct order. Opposed to missed, there are *hits*. Hits happen when the masking and shuffling sorts the 256-bit of elements in the correct order. To overcome this issue, the algorithm continues to iterate through the 256-bit section of the array elements and repeats the procedure on the shuffled section.

D. Parallel implementations

After implementing optimizations for pivot choosing in the sequential implementations, the next step in optimizing the Quicksort algorithm is to use parallelization. Three parallel versions of this algorithm are implemented by upgrading the above-mentioned sequential versions, as well as a version utilizing sections. If the size of the array being sorted is less than the set limit, then the sequential version of the algorithm is run to save resources and execution time.

1) Parallel implementation with using "median-of-three": This implementation uses the same strategy known as the "median-of-three" for finding the pivot element.

For the parallelization part, the OpenMP [3] #pragma omp task directive is used because it is useful for parallelizing irregular algorithms such as recursive algorithms.

```
template<typename Tip>
void quickSortTasks(Tip *array, int first, int last,
     int sequentialLimit) {
    if (first < last) {
                        // sequential sorting
        if (last - first < sequentialLimit) {
            return
    sequentialQuickSortMedianOfThreePivot(array,
    first, last);
        } else {
                         // parallel sorting
            int j = partition_medianOfThreePivot(
    array, first, last);

#pragma omp task default(none) firstprivate(array,
    first, j, sequentialLimit)
            {
                quickSortTasks(array, first, j - 1,
    sequentialLimit);
            }
#pragma omp task default(none) firstprivate(array,
    last, j, sequentialLimit)
            {
                quickSortTasks(array, j + 1, last,
    sequentialLimit);
            }
        }
    }
}
```

2) Parallel implementation with AVX2: This implementation uses the same strategy as its sequential counterpart for choosing the pivot, with added parallelization used for the recursive calls of the algorithm. The same OpenMP directive # pragma omp task is used for this implementation.

3) Parallel implementation with sections: The strategy used for choosing the pivot is the "median-of-three" rule. In this version is used the OpenMP # pragma omp parallel sections directive. The difference between tasks and sections is in the time frame in which the code will execute. Sections are

enclosed within the sections construct and threads will not leave it until all sections have been executed.

The code containing the use of this directive is shown below.

```
#pragma omp parallel sections default(none)
    firstprivate(array, first, last, j,
    sequentialLimit) num_threads(numThreads)
            {
#pragma omp section
                {
                    quickSortSections(array, first,
    j - 1, sequentialLimit);
                }
#pragma omp section
                {
                    quickSortSections(array, j + 1,
    last, sequentialLimit);
                }
            }
```

V. Comparative Analysis and Results

This section provides a detailed evaluation of the Quicksort sequential and parallel algorithm implementations discussed in section IV.

The implemented algorithms are tested on devices whose characteristics are shown in Table I.

TABLE I
DEVICE FEATURES

Device features			
Type	Memory	Frequency	No. of cores
Intel Core i5-7200U	8GB	2.7 GHz	2
Intel Core i7-2620M	4GB	2.7 GHz	2
Intel Core i7-8565U	16GB	1.8 GHz	4
Intel Core i7-11800H	16GB	2.3 GHz	8

Figure 1 shows the average execution times of all the versions in regards to the array size.

Based on the results shown in Figure 1, it can be observed that for array sizes smaller than 1 million, the execution times are the same for parallel and sequential versions which is to expected because of the limit set in the parallel versions of the algorithm. This value dictates if the sequential version of the Quicksort algorithm will be used in favor of the parallel version which will most likely use up too much time on the overhead. For sizes between 1 million and 50 million the parallel and sequential versions have very similar execution times. However, for array sizes above 50 million the parallel versions are faster than their sequential counterparts, with the Parallel task AVX2 being the fastest.

The results in figures 2, 3 and 4 show that the best execution time for all parallel versions is achieved with 16 threads, while the Parallel task AVX2 version for the worst case executed on 512 threads has the overall best execution time.

In Figure 5, it can be seen that the parallel tasks AVX2 version is the best for both average and worst case scenarios respectively. Another thing to note is that strategies used, such as the ”median-of-three” rule and the partition with AVX2, transform the worst case into best case, but only improve the average case to a better average case. The reason for this is that the ”median-of-three” rule divides an array exactly in half

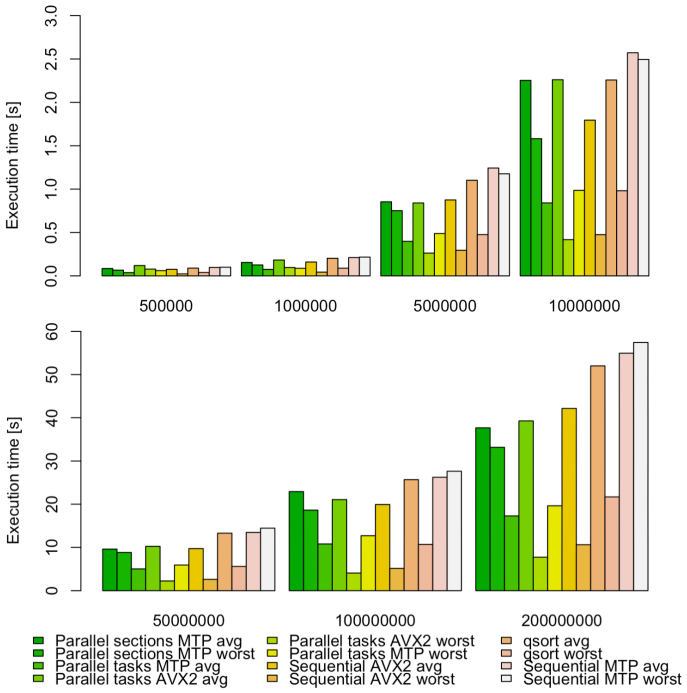

Fig. 1. Comparison of average execution times of all implemented versions in regards to array size

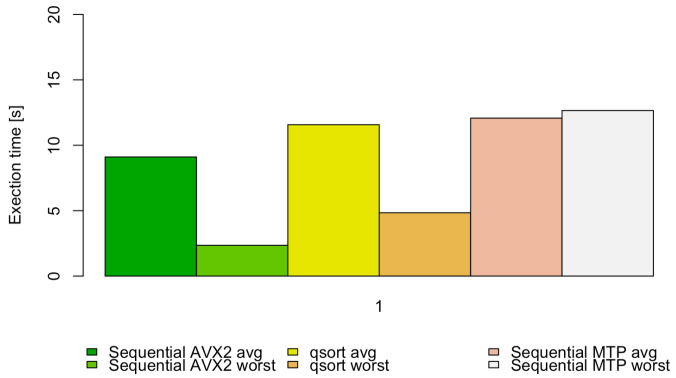

Fig. 2. Comparison of execution times of all sequential implemented versions in regards to thread count

in the worst case (the case where an array is already sorted). According to that, the work is evenly distributed among the threads.

Based on the results shown in Figure 6, it can be observed that there is no much difference in execution time for smaller array sizes. However, with the increasing number of array sizes, the differences are more noticeable.

The execution time comparison of the sequential and parallel AVX2 versions and the ratio of the sequential and parallel execution time (speedup) is shown in Table II.

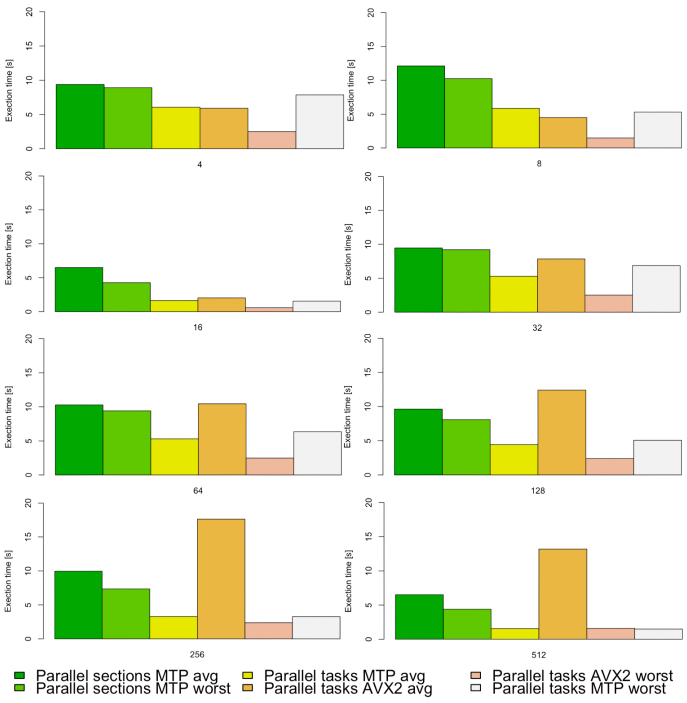

Fig. 3. Comparison of execution times of all parallel implemented versions in regards to thread count

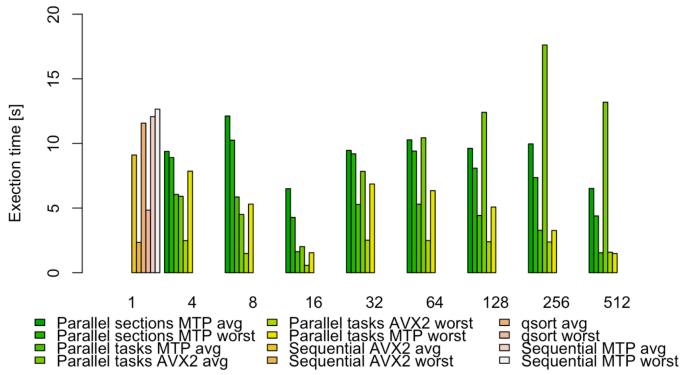

Fig. 4. Comparison of average execution times of all implemented versions in regards to thread count

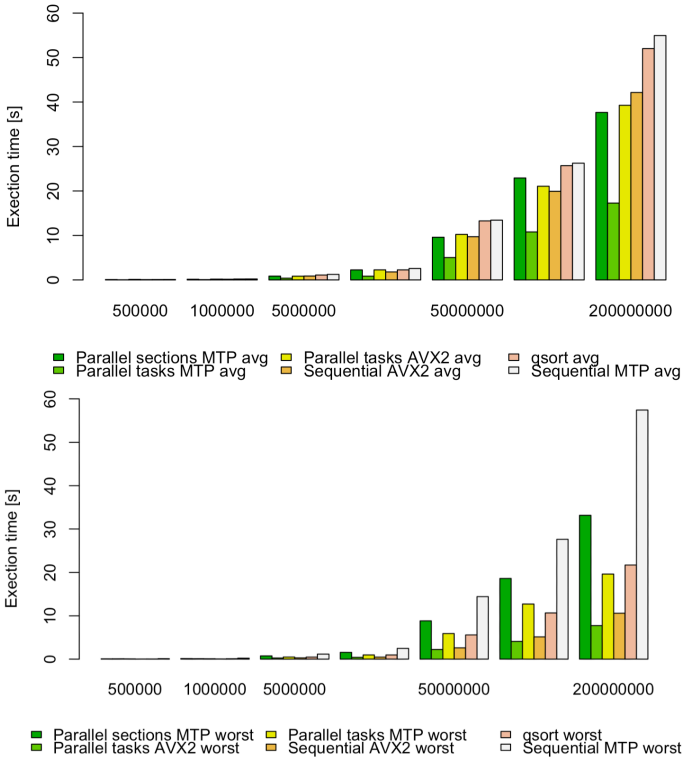

Fig. 5. Comparison of average execution times of implemented versions for the average and worst cases

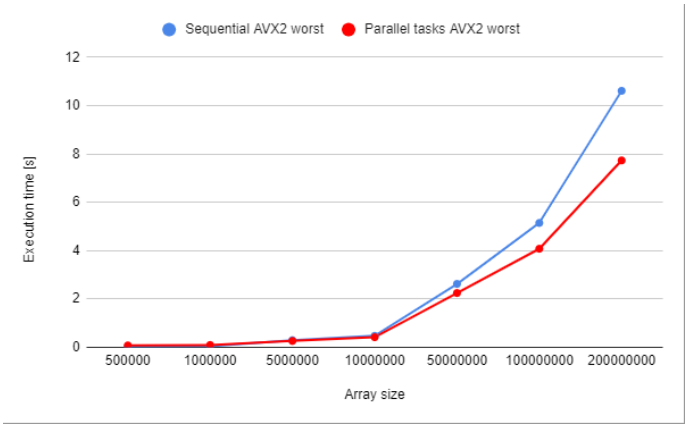

Fig. 6. Comparison of average execution times of the fastest sequential and parallel implementation

TABLE II
SPEEDUP

	Execution time [s]		
Array size	Sequential "median-of-three" (average)	Parallel tasks AVX2 (average on 16 threads)	*Speedup (average)*
500 000	0.097	0.029	3.345x
1 000 000	0.2115	0.057	3.710x
5 000 000	1.24325	0.233	5.336x
10 000 000	2.57175	0.407	6.319x
50 000 000	13.44375	1.903	7.064x
100 000 000	26.254	3.798	6.913x
200 000 000	54.954	7.720	7.118x

VI. CONCLUSION

This paper presents multiple solutions that gradually increase the performance of the Quicksort algorithm. Starting by implementing the "median-of-three" rule for choosing the pivot, then upgrading that to the use of AVX2 instructions for pivot choosing, and finally using OpenMP directives for parallel execution of the recursive parts of the algorithm. Final version containing AVX2 instructions for pivot choosing and parallel tasks achieved a speedup of about 6.5x for array sizes above 1 million in comparison to first version - the sequential

implementation with ”median-of-three” pivot choosing.

REFERENCES

[1] Kenneth Edward Batcher. “Sorting networks and their applications”. In: *Proc. AFIPS Spring Joint Computer Conf.* 32 (1968), pp. 307–314.

[2] Berenger Bramas. “Fast sorting algorithms using AVX-512 on Intel Knights Landing”. In: *arXiv preprint arXiv:1704.08579* 305 (2017), p. 315.

[3] Leonardo Dagum and Ramesh Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *Computational Science & Engineering, IEEE* 5.1 (1998), pp. 46–55.

[4] Shay Gueron and Vlad Krasnov. “Fast quicksort implementation using AVX instructions”. In: *The Computer Journal* 59.1 (2016), pp. 83–90.

[5] Charles AR Hoare. “Quicksort”. In: *The Computer Journal* 5.1 (1962), pp. 10–16.

[6] *Intel Advanced Vector Extensions*. URL: http://software.intel.com/en-us/intel-isa-extensions (visited on 11/30/2021).

[7] James Reinders. *AVX-512 instructions*. 2013. URL: http://software.intel.com/en-us/blogs/2013/avx-512-instructions (visited on 11/30/2021).

[8] Tinku Singh, Durgesh Kumar Srivastava, and Alok Aggarwal. “A novel approach for CPU utilization on a multicore paradigm using parallel quicksort”. In: *2017 3rd International Conference on Computational Intelligence & Communication Technology (CICT)*. IEEE. 2017, pp. 1–6.