QUICKSORT

SEQUENTIAL QUICKSORT:

(Sequential) Quicksort is divide-and-conquer sorting algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot (based on some total order relation among the elements to sort) and the sub-arrays are then sorted recursively by recalling the quicksort algorithm on each of them.
The time complexity of this algorithm is, on average O(n log n) but in the worst case it makes O(n^2) comparisons.
Based on this general idea, there are different implementations of Quicksort based on the way to choose the pivot in each call of Quicksort's instance and on how to partition the array in two subarray according to the given pivot.
The implementation used in my work follows the pseudo-code idea of Tony Hoare in the so called **Hoare partition scheme.**
In such implementation the pivot is the median element of the input array and the partition is based on two pointers (indices into the range) that start at both ends of the array of the array being partitioned and then move toward each other. This implementation ends-up in good average performances.

Pseudo-code:

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[ floor((hi + lo) / 2) ]
    i := lo - 1
    j := hi + 1
    loop forever
        do
            i := i + 1
        while A[i] < pivot
        do
            j := j - 1
        while A[j] > pivot
        if i ≥ j then
            return j
        swap A[i] with A[j]
```

PARALLEL QUICKSORT:

The Parallel Quicksort implemented in my work can be think as an extension of the presented implementation of the Sequential Quicksort.
Basically, given *P* processors, the original array is evenly partitioned among all the *P* processors, according to the assumption that the size of the array is a multiple of *P* (this can be easily obtained by adjusting *P* and the size of the array through some padding). Then the Sequential Quicksort is run in parallel in each of the *P* processors obtaining an ordered subarray in each of them.
Finally, all the processes send their subarrays to the root processor (processor of rank = 0) and there, the subarrays are sequentially merged according to the increasing order of the elements, obtaining at the end the whole array sorted.
To achieve this, initially the original array is stored into the root processor and then is portioned through the function:

*int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,*
*void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)*

Then, in order that each processor send its sorted subarray to the root processor, it is used the function:

*int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,*
*void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)*

According to the time complexity of the Sequential Quicksort and considering *P* processors and an input array of size n, the cost of the sorting the subarray (of size $\frac{n}{P}$ each) is O($\frac{n}{P}$ *log* $\frac{n}{P}$) while the cost of the final merging is O($n\ P$) so the time complexity of the whole algorithm is O($\frac{n}{P}$ *log* $\frac{n}{P}$ + $n\ P$).
So, we can see that the number of processors *P* determines what of the two terms dominates, and by choosing:

1 < P < log n

the resulting time complexity results to be better than the one of the sequential case.

EXPERIMENTAL SETUP AND RESULTS:

The performance metrics considered in the work are *Speedup*, which measures the performance gain as the number of processors increases:

$$S_{up}(p) = \frac{T(1)}{T(p)} \quad 1 \leq p \leq P$$

Where $T(p)$ is the execution time with p processors and $T(1)$ is the execution time of the sequential algorithm.

And the *Scalability* $Scal(n)$ which is the execution time as input size increases.

All the tests are run under the optimization -O2, which it turned out to be the better.
Below are listed the processing times (in seconds) for the different numbers of processor and sizes of the initial array:

N. process. / Size array	2^{18}	2^{20}
$T(1)$	0.017881	0.051640
$T(4)$	0.016878	0.047622
$T(8)$	0.018802	0.098880

- **$S_{up}(p)$ measured under an array of size n =2^{18}:**

  $S_{up}(4)$ = 1.059

  $S_{up}(8)$ = 0.951

- **$S_{up}(p)$ measured under an array of size n =2^{20}:**

  $S_{up}(4)$ = 1.084

  $S_{up}(8)$ = 0.522

Even if with 4 processors we can see a positive speedup of the algorithm, already with 8 we can see a worsening of the performance. This, so early aggravation, even if not completely expected, it turns out to be understandable not only for some inter - processors communication slowness but is even due to the bottleneck given by the merge step performed in the root processor. Furthermore, the speed-up increases with the increasing of the size of the array, if the number of processors is less than the number for which the bottleneck given by the merge step takes place.

CONCLUSIONS:

To conclude, it is possible to see an improvement as far as the number of processors is not too large with respect to the size of the input, due to the merge step in which a sequential number of merging operations (in the size of the processors) must be done in sorted subarrays. Although there are many parallel quicksort implementations that are better, this implementation was thought of by me, so I feel satisfied with my work.

BIBLIOGRAPHY:

- My GitHub code
- https://www.open-mpi.org
- https://en.wikipedia.org/wiki/Quicksort
- https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/
- https://gitlab.com/skimmy/capri/-/tree/master