The **Simple Parallel Quicksort** is the first attempted method to parallelize the quicksort algorithm in a distributed memory environment. Given a number P of processes, the algorithms first requires to split the initial array $\mathbf{X}$ into P equally sized[1] chunks, $\mathbf{X}^i$, of size n/P and then distribute these sub-arrays among the processes.
One of the processes then selects a pivot value τ and broadcasts it to all the others. This can be easily achieved in MPI [?] thanks to the use of the `MPI_Bcast` function. Each process can then partition its chunk according to this value forming in this way two local partitions: the *low partition* $\mathbf{X}^i_{<\tau}$, with elements smaller than the pivot, and the *high partition* $\mathbf{X}^i_{\geq\tau}$, with elements greater or equal to the pivot.
Subsequently, any process having rank $i < P/2$ exchanges its high partition with the low partition of the process with rank $i + P/2$ in a send and receive operation. Such operation has indeed been implemented in MPI with the `MPI_Sendrecv` function. This results in processes with rank $i < P/2$ having only elements smaller than the pivot and processes with rank $i \geq P/2$ having elements greater or equal than the pivot.
The algorithm then proceeds recursively as in the serial version with the lower rank group of processes working on the *low* part of the array on one hand, while higher rank processes keep sorting the *high* section on the other. In the MPI implemented code this separation is acheved through the use of the `MPI_Comm_split` function, which allows to define custom communicators for each subsequent recursive calls.
The following pseudocode briefly summarizes the core steps in the MPI implementation:

MPI Simple Parallel Quicksort

```
function MPI_Parallel_qsort($\mathbf{X}^i$, P, comm):
    if P > 1:
        if rank == 0:
            $\tau \leftarrow$ choose_pivot($\mathbf{X}$)
            MPI_Bcast($\tau$, comm)

        $\mathbf{X}^i_{<\tau}, \mathbf{X}^i_{\geq\tau} \leftarrow$ partition($\mathbf{X}^i, \tau$)

        if rank < P/2:
            MPI_Sendrecv($\mathbf{X}^i_{\geq\tau}$, $\mathbf{X}^{i+P/2}_{<\tau}$, comm)
        else:
            MPI_Sendrecv($\mathbf{X}^i_{<\tau}$, $\mathbf{X}^{i-P/2}_{\geq\tau}$, comm)

        MPI_Comm_split(comm, rank < P/2, low_comm)
        MPI_Comm_split(comm, rank >= P/2, high_comm)

        MPI_Parallel_qsort($\mathbf{X}^i$, P/2, low_comm)
        MPI_Parallel_qsort($\mathbf{X}^i$, P/2, high_comm)
    else:
        $\mathbf{X}_i$ $\leftarrow$ sort($\mathbf{X}_i$)
```

Code 1: Pseudocode for the MPI implemetation of the Simple Parallel Quicksort algorithm.

From pseudocode 1 it's possible to deduce two observations. First of all, the processes P must be a power of 2 for the algorithm to work properly, since at each recursive iteration the number of processes is halved. Secondly, the recursion stops when only one process is entering each recursvie call, at which point the local array is sorted using either the serial quicksort ot the **Task Parallel Quicksort** showed in code **??**. After $\log_2(P)$ recursions, in fact, the largest value in the chunk of the i^{th} process will be smaller than the smallest value in the chunk of the $(i+1)^{th}$ process. The final sorted array can therefore be obtained by simply letting each process sort its partition locally.
The average time complexity of this algorithm is $\tilde{\mathcal{O}}(n/P \log n/P)$, but its actual execution is strictly dependent on the choice of the pivot at each recursive call. A bad choice of the

[1]Plus any eventual remainder distributed in a *round-robin* fashion among the processes.

pivot might lead to the processes having to work on very differently sized chunks, resulting in a very unbalanced workload.
An attempt has also been made to implement a shared memory version of this algorithm using OpenMP [?]. This implementation is not as straightforward as its distributed memory counterpart and requires the more complex procedure here described.
At the beginning of the recursive call, each thread is assigned an independent portion of the array to be sorted, which, in this case, is a shared variable. One among the threads then picks a pivot and writes its value in a shared variable so that each thread can parallely partition its portion in place. Since in OpenMP there's no direct way to perform message passing, nor to exchange the partitions between threads, the next step assumes the form of a global rearrangement of the elements in the array according to a shared `indexes` array. Such operation requires to know in advance where the elements of each thread's partition will be placed in the final array. In order to find these positions, each thread first writes the count of elements in its low partition on a reserved location of a shared array **L**. The same happens for the high partitions in an array **H**. The prefix sum is then performed on these arrays so that

- the j^{th} element in the low partition of the i^{th} thread will end up in position $L[i] + j$ from the start of the initial array[2];
- the j^{th} element of the high partition of the i^{th} thread will end up in position $L[P] + H[i] + j$ from the start of original array[3].

Therefore, each thread can independently compute the final positions for the elements in its partitions and write these in the corresponding locations of the shared `indexes` array. This whole procedure is schematically depicted in figure 1.
Before entering the next recursive call, one among the threads takes the job of serially rearranging the elements of the array according to the so computed `indexes` array. Unfortunately, this last step requires a sequential execution in order to avoid possible data races or other types of undefined behaviors. This indeed constitutes the major bottleneck of this implementation.

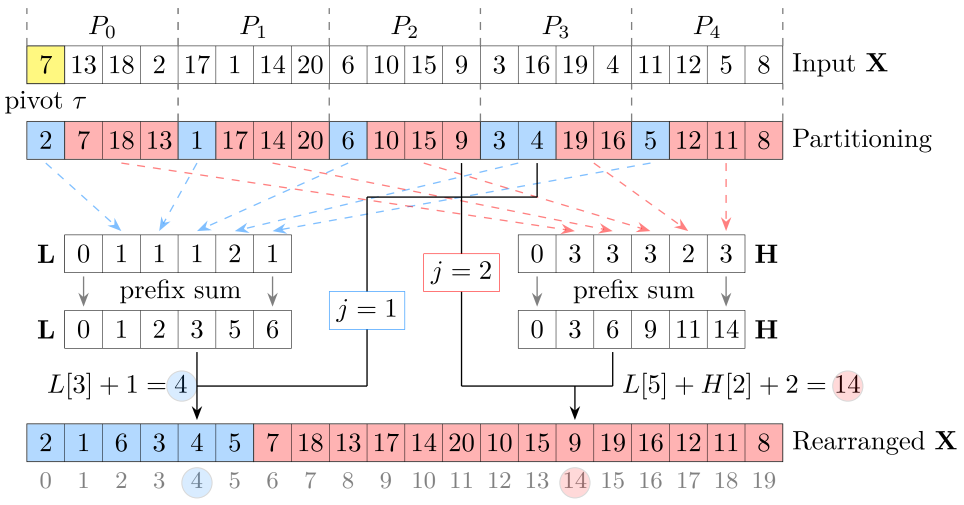

Figure 1: Rearrangement of partitions in shared memory using the prefix sum and $P = 5$ threads [?]. The pivot value is highligted in yellow, low partitions $\mathbf{X}^i_<$ are in blue and high partitions $\mathbf{X}^i_>$ are in red. Note that thread numbering is zero-based so the first thread will be identified by $i = 0$.

[2]The first element of both the **L** and **H** arrays is set to 0 to account for the final position of the first elements in each partition.
[3]Where P here is also the total number of threads.