The **Parallel Sorting by Regular Sampling (PSRS)** prioritizes the pivot selection phase to improve load balancing among processes. Contarily to the prevoius algorithms, this one avoids the tipical quicksort recursive calls, hence the number of processes involved doesn't need to be a power of 2 and the all algorithm can be summarized in 4 stages.
In the first stage, each process P_i sorts its own chunk locally and chooses P samples at regular intervals from positions:

$$\frac{i \cdot n}{P^2} \quad \text{for} \quad i = 0, 1, \ldots, P-1$$

One process then gathers all the samples from the other processes, sorts the so obtained array of $P \times P$ samples and chooses $P-1$ pivots again at regular intervals. These pivots are then broadcasted to all the other processes. Each process then partitions its chunk in P sub-chunks using the received pivots and then participates in a *all-to-all* total echange in which, $\forall i \neq j$, each process P_i keeps its i^{th} partition and sends to process P_j the j^{th} partition. Finally, each process merges the P sub-chunks it received and sorts the resulting chunk. Pseudocode 1 describes the MPI implementation of this algorithm.
The overall complexity of the algorithm is $\tilde{\mathcal{O}}(n/P \log P)$, to which we have to account an overall communication cost of $\tilde{\mathcal{O}}(n/P)$ due to the *all-to-all* exchange.
The main advantange of the PSRS algorithm it that it prioritizes the pivot selection phase in order to reach optimal load balancing among processes. Repeated swappings of the same values are also avoided and communications among processes is minimized, making this algorithm the best candidate for large scale parallelization.

As for the previous algorithms, a shared memory implementation of this procedure has

MPI_PSRS

```
1   function MPI_PSRS(X^i):
2       X^i ← sort(X^i)
3       local_samples ← sample(X^i, P)
4
5       MPI_Gather(local_samples, samples, P×P, 0, MPI_COMM_WORLD)
6
7       if rank == 0:
8           samples ← sort(samples)
9           pivots ← pick_pivots(samples, P)
10          MPI_Bcast(pivots, MPI_COMM_WORLD)
11
12      (X^i_0, ... X^i_P) ← partition(X^i, pivots)
13
14      MPI_Alltoall(X^i, P, X^i, P, MPI_COMM_WORLD)
15
16      X^i ← sort(X^i)
```

Code 1: Pseudocode for the MPI implementation of the PSRS algorithm.

been attempted in OpenMP. Similarly to the distributed memory algorithm, the local chunk sorting, samples and pivot selection as well as the local partitioning phases are all done in parallel by each thread, with the only difference being that the *gather* and *broadcast* operations are simulated using shared variables.
The problem arises again in this algorithm when it comes to exchanging the different partitions among the threads. As in the shared memory versions of the **Simple Parallel Quicksort** and the **Hyperquicksort**, the *all-to-all* exchange assumes the form of a global rearrangement of the elements in the array according to a shared `indexes` array. The difference in the case of the PSRS algorithm is that the number of partitions to be exchanged is not fixed, but depends on the number P of threads. For this reason, the computation of the elements' final positions cannot be done with only two *prefix-sum* arrays, but requires

an entire $(P+1) \times (P+1)$ *prefix* matrix[1] $\mathbf{M}$. This matrix is filled by all the threads and for all partitions in the following way. Initially, the i^{th} thread writes the count of elements in its k^{th} partition on the $(k+1)^{th}$ row and $(i+1)^{th}$ column of the matrix. After all the threads have finished writing, the i^{th} thread proceeds by computing the prefix sum of the $(i+1)^{th}$ row of the matrix. Once all the threads complete this operation, the prefix sum of the last column of the matrix is also computed and stored in an array $\mathbf{S}$. Having these quantities computed, finding the final positions of the elements in each thread's partition is straightforward: the j^{th} element in the k^{th} partition of the i^{th} thread will end up in position

$$\mathbf{S}[k] + \mathbf{M}[k+1][i] + j$$

since $\mathbf{S}[k]$ represents the total number of elements in the partitions preeceding the k^{th} one, while $\mathbf{M}[k+1][i]$ is the total number of elements in the same partition of the j^{th} element that the threads preceeding the i^{th} one have to place.
Once the `indexes` array has been filled with the so computed final positions, these are used by a single thread to serially reorder the elements in the shared input array. However, this algorithm presents a small advantage when compared to the other two shared memory versions. In fact, once the previous serial step is completed, no recursive call is needed and the final sorting of the array can actually happen in parallel by having thread i^{th} sorting the elements from position $\mathbf{S}[i]$ to position $\mathbf{S}[i] + \mathbf{M}[i+1][P]$. This possibility helps minimizing the penality of the serial portion of the algorithm and the difference when compared to the other two shared memory version is noticeable as reported in the results section.

[1]The additional dimension is needed since the 1^{st} row and the 1^{st} column of the matrix are zeroes for correct computation of the final positions.