The serial sample sort algorithm sorts $n$ integers ($X = [x_1, \ldots, x_n]$) with the following steps:

1. A sample of $s$ ($s \ll n$) elements $[x_{i_1}, \ldots, x_{i_s}]$ is selected from the $n$-element vector $X$. We sort the $s$ elements and then choose $p - 1$ “splitter values” $a_1 < \cdots < a_{p-1}$ that divide the sample vector $[x_{i_1}, \ldots, x_{i_s}]$ into $p$ nearly equal sub-vectors.

2. The splitter values define $m$ “buckets” as follows: $B_1 = \{x| x < a_1\}$, $B_i = \{x| a_{i-1} \leq x < a_{i+1}\}$, $2 \leq i \leq p - 1$, $B_p = \{x| a_{p-1} < x\}$. During one pass through the vector $X$ each element $x_j$ is placed in the appropriate bucket according to its value (i.e., is copied into an array $B_j$).

3. Each bucket is sorted independently.

4. The entire sorted vector is obtained by concatenating the contents of the sorted buckets $[B_1 \cdots B_p]$.

Part I. Write a parallel sample sort algorithm for $p$ processors. The algorithm goes as follows:

1. The data (the vector $X$) is initially at $P_0$. The vector is scattered to all processes such that each process $P_i$ gets a sub-vector $X_i$ with $n/p$ elements.

2. Each process selects $k$ sample elements from its sub-vector and sends them to $P_0$. The process $P_0$ now has a sample of $s = kp$ elements which it uses to select the splitter values $a_1 < \cdots < a_{p-1}$. These splitters are broadcast to everybody.

3. Each process $P_i$ goes through the local sub-vector $X_i$ and splits it into $p$ subsequences, such that elements in subsequence $j$ belong to bucket $j$ ($X_{i,j} \in B_j$).

4. Each process $P_i$ sends the sub-sequence $X_{i,j}$ to process $P_j$.

5. Process $P_j$ has all the elements of the original vector $X$ that belong to bucket $B_j$. Each process sorts locally the received sub-sequences (using e.g., quicksort).

6. The sorted subsequences from all processors are gathered to $P_0$ and assembled into the sorted vector $X$.

Part II. Consider the following model for the communication time:

\[ T = t_s + t_w \cdot m \]

where $t_s$ is the startup time, $t_w$ the time needed to transfer one word, and $m$ is the number of words in the message.

Run several sends and receives between different pairs of nodes and transmit messages of different sizes (different $m$’s). Time each of them. From this data estimate the $t_s$ and $t_m$ for the network of workstations (e.g., use a least squares approach).
Part III. Perform a theoretical analysis of the sorting algorithm.

1. Express $T_S$, the serial time, as a function of the data vector size $n$.

2. Express $T_P$, the parallel time, as a function of $n$ and of the number of processors $p$.

3. Express the parallel overhead $T_O$ as a function of $n$ and $p$.

4. Find and expression for the regular speedup, and analyze its behavior when $p \to \infty$.

5. Find and expression for the time-constrained speedup, and analyze its behavior when $p \to \infty$.

6. Find and expression for the memory-constrained speedup, and analyze its behavior when $p \to \infty$.

7. Express the parallel efficiencies in each of the three above cases, and analyze their behavior when $p \to \infty$.

Part IV. Run the parallel sorting algorithm for different data sizes (e.g., $n = 2^{10}$ and up) and on different number of processors (e.g., $p = 1, 2, \ldots$ to 16). Time each of the runs. For each of the three cases (regular, time-constrained, and memory-constrained speedups) plot the experimental speedup AND the theoretical speedup versus the number of processors. Discuss and draw conclusions.

Can you predict the behavior of the sorting algorithm on System X ($p = 2^{11}$)? How large should $n$ be to have a parallel efficiency of at least $E \geq 50\%$?