Challenge III: Shearsort

Description

For this challenge we look at shearsort, which is a parallelisable algorithm for sorting an $n \times n$ integer matrix in a snake-like order. With snake-like we mean that, after termination of shearsort, the rows of the given input matrix have been sorted in alternating direction.

The next page shows a pseudo-code implementation of shearsort. It takes an integer matrix $M$ as input, which is assumed to be of size $n \times n$ (with $n$ a positive integer). Then the following two steps are repeated $\lceil \log_2(n) \rceil + 1$ times:

1. Sort all rows of $M$ in an alternating manner.
2. Sort all columns of $M$ in ascending order.

Below an example application of shearsort is given, on a $3 \times 3$ matrix:

\[
\begin{array}{ccc}
5 & 8 & 2 \\
9 & 1 & 7 \\
3 & 6 & 4 \\
\end{array}
\]

Input matrix

\[
\begin{array}{ccc}
2 & \leftarrow 5 \rightarrow 8 \\
9 & \leftarrow 7 \rightarrow 1 \\
3 & \leftarrow 4 \rightarrow 6 \\
\end{array}
\]

Round 1: sorting rows

\[
\begin{array}{ccc}
1 & \leftarrow 2 \rightarrow 4 \\
6 & \leftarrow 5 \rightarrow 3 \\
7 & \leftarrow 8 \rightarrow 9 \\
\end{array}
\]

Round 2: sorting rows

\[
\begin{array}{ccc}
1 & 2 & 3 \\
6 & 5 & 4 \\
7 & 8 & 9 \\
\end{array}
\]

Round 2: sorting columns

\[
\begin{array}{ccc}
1 & 2 & 3 \\
6 & 5 & 4 \\
7 & 8 & 9 \\
\end{array}
\]

Round 3: sorting rows

\[
\begin{array}{ccc}
2 & 4 & 1 \\
3 & 5 & 6 \\
9 & 7 & 8 \\
\end{array}
\]

Round 1: sorting columns

\[
\begin{array}{ccc}
\end{array}
\]

Output matrix

The row and column sorts in every round can be performed in parallel, as they operate on disjoint memory. The implementations of sort-row and sort-column are left abstract, but could be chosen to be any sorting function.
Furthermore, an alternative implementation of `shearsort` is given, that uses a matrix transpose operation instead of `sort-column`. This version should be equivalent to `shearsort`, although less efficient. Feel free to perform the verification tasks using `alternative-shearsort` instead, if that is more convenient.

Moreover, in case your verifier does not support reasoning about concurrency, feel free to turn all parallel for-loops into sequential ones.

**Verification tasks**

The verification tasks for `shearsort` are:

1. Verify that `shearsort` terminates, and is memory safe.
2. Verify that `shearsort` permutes the input matrix.
3. Verify that `shearsort` sorts the matrix in a snake-like manner.
4. Verify that (parallel) `shearsort` satisfies the same specification as sequential `shearsort`, in which all parallel for-loops are replaced by sequential ones.
5. Verify that `shearsort` and `alternative-shearsort` satisfy the same specification.
6. Extra: give implementations to `sort-row`, `sort-column` and `transpose`, and verify these as well.
// Sorts the row-th row of M in ascending order if ascending is true,
// or in descending order if ascending is false.
3 void sort-row(int[][] M, int row, bool ascending) {
4   ...

// Sorts the column-th column of M in ascending order.
7 void sort-column(int[][] M, int column) {
8   ...

// Sorts M in snake-like order, assuming that M is an n × n matrix.
11 void shearsort(int n, int[][] M) {
12   repeat ⌈log₂(n)⌉ + 1 times {
13      for int tid = 0...n do in parallel {
14         sort-row(M, tid, tid % 2 = 0);
15      }
16      for int tid = 0...n do in parallel {
17         sort-column(M, tid);
18      }
19   }
20
// An alternative version of shearsort, that only uses sort-row.
22 void alternative-shearsort(int n, int[][] M) {
23   repeat ⌈log₂(n)⌉ + 1 times {
24      for int tid = 0...n do in parallel {
25         sort-row(M, tid, tid % 2 = 0);
26      }
27      transpose(M);
28      for int tid = 0...n do in parallel {
29         sort-row(M, tid, true);
30      }
31      transpose(M);
32   }