# Parallel Shear Sort

Manuel Eberl and Peter Lammich

March 17, 2025

**Abstract**

This entry provides a formalisation of *parallel shear sort*, a comparison-based sorting algorithm intended for highly parallel systems. It sorts $n$ elements in $O(\log n)$ steps, each of which involves sorting $\sqrt{n}$ independent lists of $\sqrt{n}$ elements each.

If these smaller sort operations are done in parallel with a conventional $O(n \log n)$ sorting algorithm, this leads to an overall work of $O(n \log^2(n))$ and a span of $O(\sqrt{n} \log^2(n))$ – a considerable improvement over conventional non-parallel sorting.

## Contents

```
theory Parallel_Shear_Sort
  imports Complex_Main "HOL-Library.Multiset" "HOL-Library.FuncSet" "HOL-Library.Log_Nat"
begin
```

### 0.0.1 Facts about sorting

```
lemma sort_map_mono: "mono f ⟹ sort (map f xs) = map f (sort xs)"
  ⟨proof⟩

lemma sorted_boolE:
  assumes "sorted xs" "length xs = w"
  shows   "∃k≤w. xs = replicate k False @ replicate (w - k) True"
⟨proof⟩

lemma rev_sorted_boolE:
  assumes "sorted (rev xs)" "length xs = w"
  shows   "∃k≤w. xs = replicate k True @ replicate (w - k) False"
⟨proof⟩
```

### 0.0.2 Miscellaneous

```
lemma map_nth_shift:
  assumes "length xs = b - a"
  shows   "map (λj. xs ! (j - a)) [a..<b] = xs"
⟨proof⟩
```

## 0.1 Auxiliary definitions

The following predicate states that all elements of a list are equal to one another.

```
definition all_same :: "'a list ⇒ bool"
  where "all_same xs = (∃x. set xs ⊆ {x})"

lemma all_same_replicate [intro]: "all_same (replicate n x)"
  ⟨proof⟩

lemma all_same_altdef: "all_same xs ⟷ xs = replicate (length xs) (hd
xs)"
⟨proof⟩

lemma all_sameE:
  assumes "all_same xs"
  obtains n x where "xs = replicate n x"
  ⟨proof⟩
```

The following predicate states that a list is sorted in ascending or descending order, depending on the boolean flag.

```
definition sorted_asc_desc :: "bool ⇒ 'a :: linorder list ⇒ bool"
```

**where** *"sorted_asc_desc asc xs = (if asc then sorted xs else sorted (rev xs))"*

Analogously, we define a sorting function that takes such a flag.

**definition** *sort_asc_desc :: "bool ⇒ 'a :: linorder list ⇒ 'a list"*
**where** *"sort_asc_desc asc xs = (if asc then sort xs else rev (sort xs))"*

**lemma** *length_sort_asc_desc [simp]: "length (sort_asc_desc asc xs) = length xs"*
⟨*proof*⟩

**lemma** *mset_sort_asc_desc [simp]: "mset (sort_asc_desc asc xs) = mset xs"*
⟨*proof*⟩

**lemma** *sort_asc_desc_map_mono: "mono f ⟹ sort_asc_desc b (map f xs) = map f (sort_asc_desc b xs)"*
⟨*proof*⟩

**lemma** *sort_asc_desc_all_same: "all_same xs ⟹ sort_asc_desc asc xs = xs"*
⟨*proof*⟩

## 0.2 Matrices

We represent matrices as functions mapping index pairs to elements. The first index is the row, the second the column. For convenience, we also fix explicit lower and upper bounds for the indices so that we can easily talk about minors of a matrix (or "submatrices"). The lower bound is inclusive, the upper bound exclusive.

**type_synonym** *'a mat = "nat × nat ⇒ 'a"*

**locale** *shearsort =*
**fixes** *lrow urow lcol ucol :: nat* **and** *dummy :: "'a :: linorder"*
**assumes** *lrow_le_urow: "lrow ≤ urow"*
**assumes** *lcol_le_ucol: "lcol ≤ ucol"*
**begin**

The set of valid indices:

**definition** *idxs :: "(nat × nat) set"* **where** *"idxs = {lrow..<urow} × {lcol..<ucol}"*

The multiset of all entries in the matrix:

**definition** *mset_mat :: "(nat × nat ⇒ 'b) ⇒ 'b multiset"*
**where** *"mset_mat m = image_mset m (mset_set idxs)"*

The $i$-th row and $j$-th column of a matrix:

**definition** *row :: "(nat × nat ⇒ 'b) ⇒ nat ⇒ 'b list"*

```
  where "row m i = map (λj. m (i, j)) [lcol..<ucol]"
definition col :: "(nat × nat ⇒ 'b) ⇒ nat ⇒ 'b list"
  where "col m j = map (λi. m (i, j)) [lrow..<urow]"

lemma length_row [simp]: "length (row m i) = ucol - lcol"
  and length_col [simp]: "length (col m i) = urow - lrow"
  ⟨proof⟩

lemma nth_row [simp]: "j < ucol - lcol ⟹ row m i ! j = m (i, lcol
+ j)"
  ⟨proof⟩

lemma set_row: "set (row m i) = (λj. m (i, j)) ` {lcol..<ucol}"
  ⟨proof⟩

lemma set_col: "set (col m j) = (λi. m (i, j)) ` {lrow..<urow}"
  ⟨proof⟩

lemma mset_row: "mset (row m i) = image_mset (λj. m (i, j)) (mset [lcol..<ucol])"
  ⟨proof⟩

lemma mset_col: "mset (col m j) = image_mset (λi. m (i, j)) (mset [lrow..<urow])"
  ⟨proof⟩

lemma nth_col [simp]: "i < urow - lrow ⟹ col m j ! i = m (lrow + i,
j)"
  ⟨proof⟩
```

The following helps us to restrict a matrix operation to the valid indices. Here, `m` is the original matrix and `m'` the changed matrix that we obtained after applying some operation on it.

```
definition restrict_mat :: "'a mat ⇒ 'a mat ⇒ 'a mat" where
  "restrict_mat m m' = (λij. if ij ∈ idxs then m' ij else m ij)"

lemma row_restrict_mat [simp]:
  "row (restrict_mat m m') i = (if i ∈ {lrow..<urow} then row m' i else
row m i)"
  ⟨proof⟩

lemma col_restrict_mat [simp]:
  "col (restrict_mat m m') j = (if j ∈ {lcol..<ucol} then col m' j else
col m j)"
  ⟨proof⟩
```

The following lemmas allow us to prove that two matrices are equal by showing that their rows (or columns) are the same.

```
lemma matrix_eqI_rows:
  assumes "⋀i. i ∈ {lrow..<urow} ⟹ row m1 i = row m2 i"
  assumes "⋀i j. (i, j) ∉ idxs ⟹ m1 (i, j) = m2 (i, j)"
```

**shows** *"m1 = m2"*
⟨*proof*⟩

**lemma** *matrix_eqI_cols:*
**assumes** *"⋀j. j ∈ {lcol..<ucol} ⟹ col m1 j = col m2 j"*
**assumes** *"⋀i j. (i, j) ∉ idxs ⟹ m1 (i, j) = m2 (i, j)"*
**shows** *"m1 = m2"*
⟨*proof*⟩

The following lemmas express the multiset of elements as a sum of rows (or columns):

**lemma** *mset_mat_conv_sum_rows: "mset_mat m = (∑ i∈{lrow..<urow}. mset (row m i))"*
⟨*proof*⟩

**lemma** *mset_mat_conv_sum_cols: "mset_mat m = (∑ j∈{lcol..<ucol}. mset (col m j))"*
⟨*proof*⟩

Lastly, we define the transposition operation:

**definition** *transpose_mat :: "((nat × nat) ⇒ 'a) ⇒ (nat × nat) ⇒ 'a"*
**where** *"transpose_mat m = (λ(i,j). m (j, i))"*

**lemma** *transpose_mat_apply: "transpose_mat m (j, i) = m (i, j)"*
⟨*proof*⟩

**sublocale** *transpose: shearsort lcol ucol lrow urow*
⟨*proof*⟩

**lemma** *row_transpose [simp]: "transpose.row (transpose_mat m) i = col m i"*
**and** *col_transpose [simp]: "transpose.col (transpose_mat m) i = row m i"*
⟨*proof*⟩

**lemma** *in_transpose_idxs_iff: "(j, i) ∈ transpose.idxs ⟷ (i, j) ∈ idxs"*
⟨*proof*⟩

## 0.3 Snake-wise sortedness

Next, we define snake-wise sortedness. For this, even-numbered rows must be sorted ascendingly, the odd-numbered ones descendingly, etc. We will show a nicer characterisation of this below.

**definition** *snake_sorted :: "'a mat ⇒ bool"* **where**
*"snake_sorted m ⟷*
*(∀ i∈{lrow..<urow}. sorted_asc_desc (even i) (row m i)) ∧*

```
    (∀i i' x y. lrow ≤ i ∧ i < i' ∧ i' < urow ∧ x ∈ set (row m i)
∧ y ∈ set (row m i') ⟶ x ≤ y)"
```

Next, we define the list of elements encountered on the snake-like path through the matrix, i.e. when traversing the matrix top to bottom, even-numbered rows left-to-right and odd-numbered rows right-to-left.

```
context
  fixes m :: "'a mat"
begin

function snake_aux :: "nat ⇒ 'a list" where
  "snake_aux i =
    (if i ≥ urow then [] else (if even i then row m i else rev (row
m i)) @ snake_aux (Suc i))"
  ⟨proof⟩
termination ⟨proof⟩

lemmas [simp del] = snake_aux.simps


definition snake :: "'a list"
  where "snake = snake_aux lrow"

lemma mset_snake_aux: "mset (snake_aux lrow') = (∑i∈{lrow'..<urow}.
mset (row m i))"
  ⟨proof⟩

lemma set_snake_aux: "set (snake_aux lrow') = (⋃i∈{lrow'..<urow}. set
(row m i))"
⟨proof⟩
```

We can now show that snake-wise sortedness is equivalent to saying that `snake` is sorted.

```
lemma sorted_snake_aux_iff:
  "sorted (snake_aux lrow') ⟷
    (∀i∈{lrow'..<urow}. sorted_asc_desc (even i) (row m i)) ∧
    (∀i i' x y. lrow' ≤ i ∧ i < i' ∧ i' < urow ∧ x ∈ set (row m i)
∧ y ∈ set (row m i') ⟶ x ≤ y)"
⟨proof⟩

lemma sorted_snake_iff: "sorted snake ⟷ snake_sorted m"
  ⟨proof⟩

end
```

## 0.4 Definition of the abstract algorithm

We can now define shear sort on matrices. We will also show that the multiset of elements is preserved.

### 0.4.1 Sorting the rows

**definition** *step1 :: "'a mat ⇒ 'a mat"* **where**
  *"step1 m = restrict_mat m (λ(i,j). sort_asc_desc (even i) (row m i) ! (j - lcol))"*

**lemma** *step1_outside [simp]: "z ∉ idxs ⟹ step1 m z = m z"*
  ⟨*proof*⟩

**lemma** *row_step1:*
  *"row (step1 m) i = (if i ∈ {lrow..<urow} then sort_asc_desc (even i) (row m i) else row m i)"*
  ⟨*proof*⟩

**lemma** *mset_mat_step1 [simp]: "mset_mat (step1 m) = mset_mat m"*
  ⟨*proof*⟩

### 0.4.2 Sorting the columns

**definition** *step2 :: "'a mat ⇒ 'a mat"* **where**
  *"step2 m = restrict_mat m (λ(i,j). sort (col m j) ! (i - lrow))"*

**lemma** *step2_outside [simp]: "z ∉ idxs ⟹ step2 m z = m z"*
  ⟨*proof*⟩

**lemma** *col_step2: "col (step2 m) j = (if j ∈ {lcol..<ucol} then sort (col m j) else col m j)"*
  ⟨*proof*⟩

**lemma** *mset_mat_step2 [simp]: "mset_mat (step2 m) = mset_mat m"*
  ⟨*proof*⟩

**lemma** *step2_height_le_1:*
  **assumes** *"urow ≤ lrow + 1"*
  **shows** *"step2 m = m"*
⟨*proof*⟩

We also show the alternative definiton of *step2* involving transposition and sorting rows:

**definition** *step2' :: "'a mat ⇒ 'a mat"* **where**
  *"step2' m = restrict_mat m (λ(i,j). sort (row m i) ! (j - lcol))"*

**lemma** *step2'_outside [simp]: "z ∉ idxs ⟹ step2' m z = m z"*
  ⟨*proof*⟩

**lemma** *row_step2': "row (step2' m) i = (if i ∈ {lrow..<urow} then sort (row m i) else row m i)"*
  ⟨*proof*⟩

**end**

**context** *shearsort*
**begin**

**lemma** *step2_altdef: "step2 m = transpose.transpose_mat (transpose.step2' (transpose_mat m))"*
⟨*proof*⟩

### 0.4.3 Combining the two steps

**definition** *step* **where** *"step = step2 ∘ step1"*

**lemma** *step_outside [simp]: "z ∉ idxs ⟹ step m z = m z"*
⟨*proof*⟩

**lemma** *row_step_outside [simp]: "i ∉ {lrow..<urow} ⟹ row (step m) i = row m i"*
⟨*proof*⟩

**lemma** *mset_mat_step [simp]: "mset_mat (step m) = mset_mat m"*
⟨*proof*⟩

The overall algorithm now simply alternates between steps 1 and 2 sufficiently often for the result to stabilise. We will show below that a logarithmic number of steps suffices.

**definition** *shearsort :: "'a mat ⇒ 'a mat"* **where**
*"shearsort = step ^^ (ceillog2 (urow - lrow) + 1)"*

The preservation of the multiset of elements is very easy to show:

**theorem** *mset_mat_shearsort [simp]: "mset_mat (shearsort m) = mset_mat m"*
⟨*proof*⟩

**end**

## 0.5 Restriction to boolean matrices

To more towards the proof of sortedness, we first take a closer look at shear sort on boolean matrices. Our ultimate goal is to show that shear sort correctly sorts any boolean matrix in $\lceil log_2\ h\rceil + 1$ steps, where *h* is the height of the matrix. By the 0–1 principle, this implies that shear sort works on a matrix of any type.

### 0.5.1 Preliminary definitions

We first define predicates that tell us whether a list is all zeros (i.e. *False*) or all ones (i.e. *True*). The significance of such lists is that we call all-zero rows at the top of the matrix and all-one rows at the bottom "clean", and we will show that even in the worst case, the number of non-clean rows halves in every step.

```
definition all0 :: "bool list ⇒ bool" where "all0 xs = (set xs ⊆ {False})"
definition all1 :: "bool list ⇒ bool" where "all1 xs = (set xs ⊆ {True})"

lemma all0_nth: "all0 xs ⟹ i < length xs ⟹ xs ! i = False"
  and all1_nth: "all1 xs ⟹ i < length xs ⟹ xs ! i = True"
  ⟨proof⟩

lemma all0_imp_all_same [dest]: "all0 xs ⟹ all_same xs"
  and all1_imp_all_same [dest]: "all1 xs ⟹ all_same xs"
  ⟨proof⟩


locale shearsort_bool =
  fixes lrow urow lcol ucol :: nat
  assumes lrow_le_urow: "lrow ≤ urow"
  assumes lcol_le_ucol: "lcol ≤ ucol"
begin

sublocale shearsort lrow urow lcol ucol True
  ⟨proof⟩
```

We say that a matrix *m* of height *h* has a clean decomposition of order *n* if there are at most *n* non-clean rows, i.e. there exists a *k* such that *m* has *k* lines that are all 0 at the top and *h - n - k* lines that are all 1 at the bottom.

```
definition clean_decomp where
  "clean_decomp n m ⟷ (∃k. lrow ≤ k ∧ k + n ≤ urow ∧
     (∀i∈{lrow..<k}. all0 (row m i)) ∧ (∀i∈{k+n..<urow}. all1 (row m
i)))"
```

A matrix of height *h* trivially has a clean decomposition of order *h*.

```
lemma clean_decomp_initial: "clean_decomp (urow - lrow) m"
  ⟨proof⟩

lemma all0_rowI:
  assumes "i ∈ {lrow..<urow}" "⋀j. j ∈ {lcol..<ucol} ⟹ ¬m (i, j)"
  shows   "all0 (row m i)"
  ⟨proof⟩

lemma all1_rowI:
  assumes "i ∈ {lrow..<urow}" "⋀j. j ∈ {lcol..<ucol} ⟹ m (i, j)"
  shows   "all1 (row m i)"
  ⟨proof⟩
```

The *step2* function on boolean matrices has the following nice characterisation: *step2 m* has a *1* at position *(i, j)* iff the number of *0*s in the column *j* is at most *i*.

```
lemma step2_bool:
  assumes "(i, j) ∈ idxs"
  shows   "step2 m (i, j) ⟷ i ≥ lrow + size (count (mset (col m j))
False)"
⟨proof⟩

end
```

### 0.5.2 Shearsort steps ignore clean rows

We now look at a at the matrix minor consisting of the *n* (possibly) non-clean rows in the middle of a matrix with a clean decomposition of order *n*. We call the new upper and lower index bounds for the rows *lrow'* and *urow'*.

```
locale sub_shearsort_bool = shearsort_bool +
  fixes lrow' urow' :: nat and m :: "bool mat"
  assumes subrows: "lrow ≤ lrow'" "lrow' ≤ urow'" "urow' ≤ urow"
  assumes all0_first: "⋀i. i ∈ {lrow..<lrow'} ⟹ all0 (row m i)"
  assumes all1_last: "⋀i. i ∈ {urow'..<urow} ⟹ all1 (row m i)"
begin

sublocale sub: shearsort_bool lrow' urow' lcol ucol
  ⟨proof⟩

lemma idxs_subset: "sub.idxs ⊆ idxs"
  ⟨proof⟩
```

It is easy to see that *step1* does not touch the clean rows at all (i.e. it can be seen as operating entirely on the minor):

```
lemma sub_step1: "sub.step1 m = step1 m"
⟨proof⟩
```

Every column of the matrix has *lrow' - lrow 0*s at the top and *urow - urow' 1*s at the bottom:

```
lemma col_conv_sub_col:
  assumes "j ∈ {lcol..<ucol}"
  shows   "col m j = replicate (lrow' - lrow) False @ sub.col m j @ replicate
(urow - urow') True"
⟨proof⟩
```

mset *step2* preserves the clean rows at the bottom and top.

```
lemma all0_step2:
  assumes "i ∈ {lrow..<lrow'}"
  shows   "all0 (row (step2 m) i)"
⟨proof⟩
```

```
lemma all1_step2:
  assumes "i ∈ {urow'..<urow}"
  shows   "all1 (row (step2 m) i)"
⟨proof⟩
```

Consequently, *step2* can also be seen as operating only on the minor.

```
lemma sub_step2: "sub.step2 m = step2 m"
⟨proof⟩
```

Thus, the same holds for the combined shear sort step.

```
lemma sub_step: "sub.step m = step m"
⟨proof⟩

end
```

### 0.5.3 Correctness of boolean shear sort

We are now ready for the final push. The main work in this section is to show that if we run a single shear sort step on a matrix of height *h*, the number of non-clean rows in the result is no greater than $\lceil h/2 \rceil$.

Together with the fact from above that the step preserves clean rows and can such be thought of as operating solely on the non-clean minor, this means that the number of non-clean rows at least halves in every step, leading to a matrix with at most one non-clean row after $\lceil log_2\ h \rceil$ steps.

```
context shearsort_bool
begin
```

If we look at two rows, one of which is sorted in ascending order and one in descending order, there exists a boolean value *x* such that every column contains an *x* (i.e. for every column index *j*, at least one of the two rows has an *x* at index *j*).

```
lemma clean_decomp_step2_aux:
  fixes m :: "bool mat"
  assumes "i ∈ {lrow..<urow}" "i' ∈ {lrow..<urow}"
  assumes "sorted (row m i)" "sorted (rev (row m i'))"
  shows   "∃x. ∀j∈{lcol..<ucol}. x ∈ {m (i, j), m (i', j)}"
⟨proof⟩
```

*step1* leaves every even-numbered row in the matrix sorted in ascending order and every odd-numbered row in descending order:

```
lemma sorted_asc_desc_row_step1:
  "i ∈ {lrow..<urow} ⟹ sorted_asc_desc (even i) (row (step1 m) i)"
  ⟨proof⟩
```

These two facts imply that applying *step2* to such a matrix indeed leads to at most $\lceil h/2 \rceil$ non-clean rows. The argument is as follows: we go through the matrix top-to-bottom, grouping adjacent rows into pairs of two (ignoring the last row if the matrix has odd height).

The above lemma proves that each such pair of rows either has a *1* in every column or a *0* in every column. Thus, the maximum number $k_0$ such that every column contains at least $k_0$ *0*s plus the maximum number $k_1$ such that every column contains at least $k_1$ 1s is at least $\lfloor h/2 \rfloor$. Thus, after applying *step2*, we have at least $k_0$ all-zero rows at the top and at least $k_1$ all-one rows at the bottom, and therefore at least $\lfloor h/2 \rfloor$ clean lines in total.

```
lemma clean_decomp_step2:
  assumes "⋀i. i ∈ {lrow..<urow} ⟹ sorted_asc_desc (even i) (row m
i)"
  shows   "clean_decomp ((urow - lrow + 1) div 2) (step2 m)"
⟨proof⟩

lemma clean_decomp_step_aux:
  "clean_decomp ((urow - lrow + 1) div 2) (step m)"
  ⟨proof⟩
```

We can now finally show that the number of non-clean rows halves in every step:

```
lemma clean_decomp_step:
  assumes "clean_decomp n m"
  shows   "clean_decomp ((n + 1) div 2) (step m)"
⟨proof⟩
```

Moreover, if we have a matrix that has at most one non-clean row, applying one last step of shear sort leads to a snake-sorted matrix. This is because

1. *step1* leaves the clean rows untouched and sorts the non-clean row (if it exists) in the correct order.
2. *step2* leaves the clean parts of the columns untouched, and since the non-clean part has height at most 1, it also leaves that part untouched.

```
lemma snake_sorted_step_final:
  assumes "clean_decomp n m" and "n ≤ 1"
  shows   "snake_sorted (step m)"
⟨proof⟩
```

It is now easy to show that shear sort is indeed correct for boolean matrices.

```
lemma snake_sorted_shearsort_bool: "snake_sorted (shearsort m)"
⟨proof⟩

end
```

## 0.6 Shearsort commutes with monotone functions

To invoke the 0–1 principle, we must now prove that shear sort commutes with monotone functions. We will only show it for functions that return booleans, since that is all we need, but it could easily be shown the same way for a more general result type as well.

**context** *shearsort*
**begin**

**interpretation** *bool: shearsort_bool lrow urow lcol ucol*
⟨*proof*⟩

**context**
**fixes** *f :: "'a ⇒ bool"*
**begin**

**lemma** *row_commute: "row (f ∘ m) i = map f (row m i)"*
**and** *col_commute: "col (f ∘ m) i = map f (col m i)"*
⟨*proof*⟩

**lemma** *restrict_mat_commute:*
**assumes** *"⋀i j. (i, j) ∈ idxs ⟹ f (m' (i, j)) = fm' (i, j)"*
**shows** *"bool.restrict_mat (f ∘ m) fm' = f ∘ restrict_mat m m'"*
⟨*proof*⟩

**lemma** *step1_mono_commute: "mono f ⟹ bool.step1 (f ∘ m) = f ∘ step1 m"*
⟨*proof*⟩

**lemma** *step2_mono_commute: "mono f ⟹ bool.step2 (f ∘ m) = f ∘ step2 m"*
⟨*proof*⟩

**lemma** *step_mono_commute: "mono f ⟹ bool.step (f ∘ m) = f ∘ step m"*
⟨*proof*⟩

**lemma** *snake_aux_commute: "bool.snake_aux (f ∘ m) lrow' = map f (snake_aux m lrow')"*
⟨*proof*⟩

**lemma** *snake_commute: "bool.snake (f ∘ m) = map f (snake m)"*
⟨*proof*⟩

**lemma** *shearsort_mono_commute:*
**assumes** *"mono f"*
**shows** *"bool.shearsort (f ∘ m) = f ∘ shearsort m"*
⟨*proof*⟩

**end**

## 0.7 Final correctness theorem

All that is left now is a routine application of the 0–1 principle.

**theorem** *snake_sorted_shearsort: "snake_sorted (shearsort m)"*
⟨*proof*⟩

**end**

## 0.8 Refinement to lists

Next, we define a refinement of matrices to lists of lists and show the correctness of the corresponding shear sort implementation. Note that this is not useful as an actual implementation in practice since the fact that we have to transpose the list of lists once in every step negates all the advantage of having a parallel algorithm.

```
primrec step1_list :: "bool ⇒ 'a :: linorder list list ⇒ 'a list list"
where
  "step1_list b [] = []"
| "step1_list b (xs # xss) = sort_asc_desc b xs # step1_list (¬b) xss"

definition step2_list :: "'a :: linorder list list ⇒ 'a list list"
  where "step2_list xss =
          (if xss = [] ∨ hd xss = [] then xss else transpose (map sort
(transpose xss)))"

definition shearsort_list :: "bool ⇒ 'a :: linorder list list ⇒ 'a list
list" where
  "shearsort_list b xss = ((step2_list ∘ step1_list b) ^^ (ceillog2 (length
xss) + 1)) xss"

primrec snake_list :: "bool ⇒ 'a list list ⇒ 'a list" where
  "snake_list asc [] = []"
| "snake_list asc (xs # xss) = (if asc then xs else rev xs) @ snake_list
(¬asc) xss"

lemma mset_snake_list: "mset (snake_list b xss) = mset (concat xss)"
  ⟨proof⟩

definition (in shearsort) mat_of_list :: "'a list list ⇒ 'a mat"
  where "mat_of_list xss = (λ(i,j). xss ! (i - lrow) ! (j - lcol))"
```

The following relator relates a matrix to a list of rows. It ensure that the dimensions and the entries are the same.

```
definition (in shearsort) mat_list_rel :: "'a mat ⇒ 'a list list ⇒ bool"
where
  "mat_list_rel m xss ⟷
     length xss = urow - lrow ∧ (∀xs∈set xss. length xs = ucol - lcol)
∧
```

(∀ *i j. i < urow - lrow* ∧ *j < ucol - lcol* ⟶ *xss ! i ! j = m (lrow + i, lcol + j))"*

**lemma (in** *shearsort) mat_list_rel_transpose [intro]:*
**assumes** *"mat_list_rel m xss" "xss ≠ []"*
**shows** *"transpose.mat_list_rel (transpose_mat m) (transpose xss)"*
⟨*proof*⟩

**lemma (in** *shearsort) mat_list_rel_row [intro]:*
**assumes** *"mat_list_rel m xss" "i ∈ {lrow..<urow}"*
**shows** *"row m i = xss ! (i - lrow)"*
⟨*proof*⟩

**lemma (in** *shearsort) mat_list_rel_mset:*
**assumes** *"mat_list_rel m xss"*
**shows** *"mset_mat m = (∑xs←xss. mset xs)"*
⟨*proof*⟩

**lemma (in** *shearsort) mat_list_rel_of_list:*
**assumes** *"length xss = urow - lrow" "⋀xs. xs ∈ set xss ⟹ length xs = ucol - lcol"*
**shows** *"mat_list_rel (mat_of_list xss) xss"*
⟨*proof*⟩

**lemma (in** *shearsort) mset_mat_of_list:*
**assumes** *"length xss = urow - lrow" "⋀xs. xs ∈ set xss ⟹ length xs = ucol - lcol"*
**shows** *"mset_mat (mat_of_list xss) = (∑xs←xss. mset xs)"*
⟨*proof*⟩

**context** *shearsort*
**begin**

**lemma** *mat_list_rel_col [intro]:*
**assumes** *"mat_list_rel m xss" "j ∈ {lcol..<ucol}" "xss ≠ []"*
**shows** *"col m j = transpose xss ! (j - lcol)"*
⟨*proof*⟩

**lemma** *length_step1_list [simp]: "length (step1_list b xss) = length xss"*
⟨*proof*⟩

**lemma** *nth_step1_list:*
*"i < length xss ⟹ step1_list b xss ! i = sort_asc_desc (b = even i) (xss ! i)"*
⟨*proof*⟩

**lemma** *mat_list_rel_step1:*
**assumes** *"mat_list_rel m xss"*

```
  shows   "mat_list_rel (step1 m) (step1_list (even lrow) xss)"
  ⟨proof⟩

lemma mat_list_rel_step2:
  assumes [intro]: "mat_list_rel m xss"
  shows   "mat_list_rel (step2 m) (step2_list xss)"
⟨proof⟩

lemma mat_list_rel_step:
  "mat_list_rel m xss ⟹ mat_list_rel (step m) (step2_list (step1_list
(even lrow) xss))"
  ⟨proof⟩

lemma mat_list_rel_shearsort:
  assumes "mat_list_rel m xss"
  shows   "mat_list_rel (shearsort m) (shearsort_list (even lrow) xss)"
⟨proof⟩

lemma mat_list_rel_snake_aux:
  assumes "mat_list_rel m xss" "lrow' ∈ {lrow..urow}"
  shows   "snake_aux m lrow' = snake_list (even lrow') (drop (lrow' -
lrow) xss)"
  ⟨proof⟩

lemma mat_list_rel_snake:
  assumes "mat_list_rel m xss"
  shows   "snake m = snake_list (even lrow) xss"
  ⟨proof⟩

end
```

The final correctness theorem for shear sort on lists of lists:

```
theorem shearsort_list_correct:
  assumes "⋀xs. xs ∈ set xss ⟹ length xs = ncols"
  shows   "mset (concat (shearsort_list True xss)) = mset (concat xss)"
    and   "sorted (snake_list True (shearsort_list True xss))"
⟨proof⟩

value "shearsort_list True [[5, 8, 2], [9, 1, 7], [3, 6, 4 :: int]]"

end
```

# References


[1] S. Sen, I. D. Scherson, and A. Shamir. Shear sort: A true two-dimensional sorting techniques for VLSI networks. In *International Conference on Parallel Processing, ICPP'86, University Park, PA,*

*USA, August 1986*, pages 903–908. IEEE Computer Society Press, 1986.