CS855: Parallel Sorting Algorithms

Marc Moreno Maza

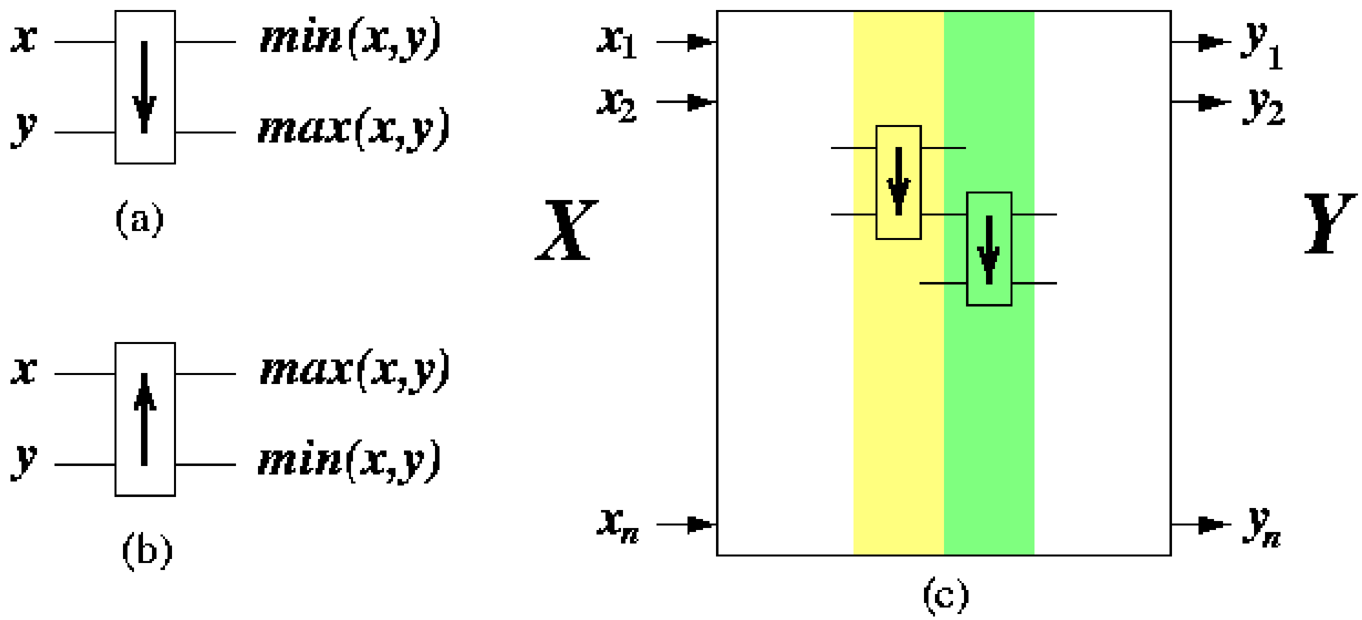
February 19, 2008

We discuss parallel deterministic sorting algorithms based on the compare-and-exchange (CE) operation. The lower bound on the number of CE operations to sort N elements is $\Omega(N \log N)$. Several optimal sequential algorithms are known, such as QuickSort or HeapSort.

A vast number of parallel sorting algorithms have been proposed. Parallel CE sorting algorithms are either PRAM or distributed memory ones.

- Sorting networks.
- Mesh-sorting algorithms.
- Batcher's Merge Sort which runs in $O(\log_2 N)$ for sorting N numbers on N-node hypercubic networks.
- Cole's Merge sort: an EREW PRAM memory sorting N numbers with N processors in time $O(\log N)$. It is a time- and cost-optimal PRAM parallel sorting algorithm.

Sorting networks consist of comparators, which are in fact hardware implementations of the CE operation



- A comparator has two inputs and two outputs.
- A sorting network is an end-to-end network composed of columns of comparators.

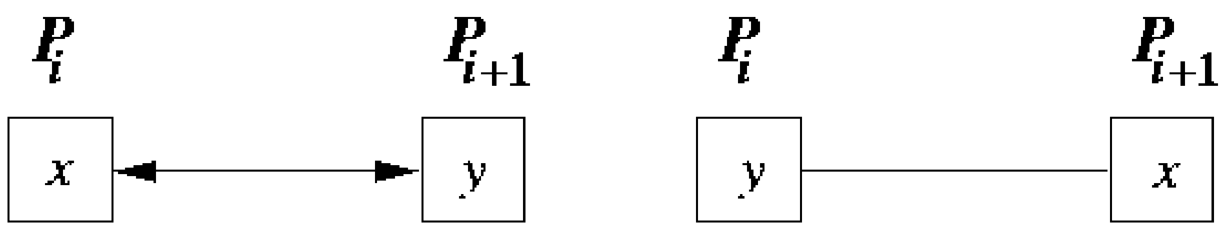
- Unsorted input sequence $X = x_1, \ldots, x_N$ placed on input wires of the leftmost column of comparators passes through the network so that it becomes sorted output sequence $Y = y_1, \ldots, y_N$ on output wires at the rightmost column. The **way** columns of **comparators are interconnected** determines the **sorting algorithm**.
- We will assume static networks with fixed interconnection of comparators. Then, each sorting network is a HW implementation of a sorting algorithm whose behavior is **data-insensitive.** The sorting algorithm induced by the sorting network is called **oblivious.**
- The number of parallel CE steps of an oblivious sorting algorithm is given by the **depth** of its sorting network:
  - Input wires of a network have depth 0.
  - If a comparator in the network has input wires of depth d_1 and d_2, then its output wires have depth $\max(d_1, d_2) + 1$.
  - The depth of a sorting network is the maximum over the depths of output wires.

- If N is greater than the number K of input wires of the sorting network, then
  - we assume that the input data set is split into N/K subsequences, which are sorted;
  - comparators are equipped with **buffers** for N/K numbers and instead of CE operation, perform **Merge-and-Split** (MS) operation:
    * two input sequences are merged and split again to lower and upper half;
    * this takes $O(N/K)$ steps.

Consider a network of $p = N$ processing nodes. We can implement a sorting network as follows.

- First, assume that each processor holds one input number.
- We impose a total ordering on nodes by numbering them $P_1, \ldots, P_N$
- Using **only exchanges** of numbers **between logically** adjacent nodes, we want P_i to hold a smaller number than P_{i+1} for all $i = 1, \ldots, N-1$.

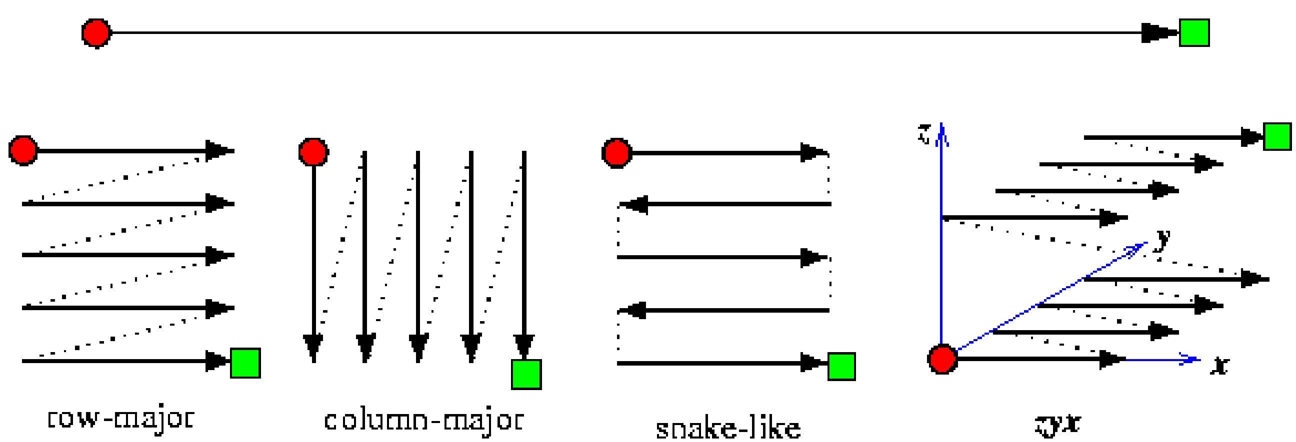
This kind of sorting is called **direct network sorting**.



Designing a direct sorting algorithm leads to address the following 3 issues.

A suitable node indexing $P_1, \ldots, P_N$.

- A node indexing preserving actual adjacency is usually preferable.
- It is trivial if the network graph has a Hamiltonian path
- The choice of indexing scheme has impact on the complexity of sorting for given topology.
- The figure below shows several indexing schemes suitable for meshes.



We must find a sequence of node matchings which will provide the final ordering compliant with the node indexing. Since we consider only oblivious direct sorting algorithms, matchings are given irrespectively to input data values.

Scalability. If $p < N$, then again one node owns N/p numbers and instead of CE operations, the neighbors perform MS operations. More specifically, an oblivious sorting algorithm then consists of two steps:

sort: each node sorts its N/p numbers in $O((N/p)\log(N/p))$ steps

MS: all nodes perform the direct sorting algorithm using MS operations.

The MS operation between 2 nodes, say P_i and P_j, for $i < j$, can be implemented by 2 methods (at least).

- **First Method:**
  1. P_i and P_j exchange their sorted subsequences,
  2. then each of them merges its subsequence with the one received from its partner into a sorted sequence of double length,

3. P_i keeps the first half of it and throws away the second one, whereas P_j does the opposite.

- **Second Method:**
  1. P_i sends its subsequence to P_j,
  2. P_j does the merging and,
  3. returns the first half of the result to P_i.

For a network with duplex links, the first method is clearly faster even if the number of computational steps is double, but asymptotically, both are equivalent.

Since CE and MS sorting differs just in the granularity of the basic step of sorting, we will, for the sake of simplicity, always assume than $\boxed{p = N}$.

In practice, however, we have always $p << N$, so we will always investigate scalability.

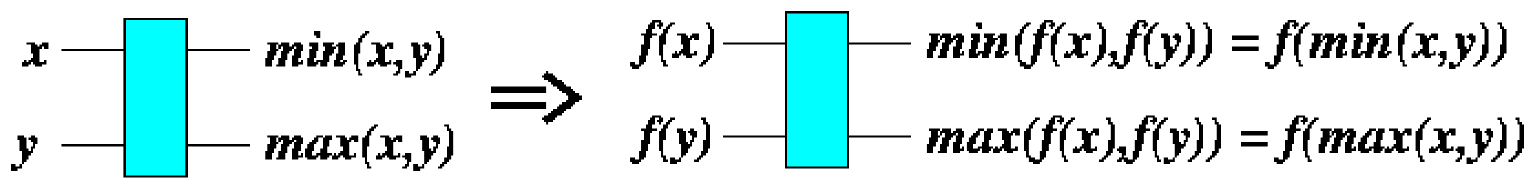
Oblivious sorting algorithms are easy to design and analyze due to the so called 0-1 Sorting Lemma. It says that if a sorting network works correctly for any input sequence of 0's and 1's, then it works correctly on any input of sequence of integers. To prove it, we need an auxiliary result.

Lemma 1 *Let f be a monotonically increasing function on a linearly ordered set S, that is, for all $x, y \in S$ we have*

$$x \leq y \quad \Longleftrightarrow \quad f(y) \leq f(y).$$

If a sorting network transforms an input sequence $X = x_1, \ldots, x_n$ of elements from S into an output sequence $Y = y_1, \ldots, y_n$, then it transforms the input sequence $f(X) = f(x_1), \ldots, f(x_n)$ into the output sequence $f(Y) = f(y_1), \ldots, f(y_n)$.

PROOF. First, we observe that any comparator is oblivious to f, that is, for all $x, y \in S$, a comparator exchanges $f(x)$ and $f(y)$ iff it exchanges x and y. The proof of this claim can be depicted as follows.



Second, we prove the following statement:

> If a wire in the network carries value x_i when the input sequence is X, then it carries value $f(x_i)$ when the input is $f(X)$.

We prove this statement by induction on the depth of the network.

- Basis of induction (trivial): If an input wire at depth 0 carries x_i, then it carries $f(x_i)$ after applying $f(X)$ to the network.
- The induction step:

- Assume that the induction hypothesis holds for all wires of depth $d-1$, for $d \geq 1$.
- Take any wire of depth d. It is an output wire of a comparator C whose input wires are at depth strictly less than d.
- By induction hypothesis, if the input wires of C carry x_i and x_j when the input sequence X is applied, then they must carry $f(x_i)$ and $f(x_j)$ when $f(X)$ is applied.
- Since the comparator C is oblivious to f, its output wires will carry $f(\min(x_i, y_j))$ and $f(\max(x_i, y_j))$.
- And the same output wires carried $\min(x_i, y_j)$ and $\max(x_i, y_j)$ when the input sequence X was applied.
- QED.

$\square$

Lemma 2 (The 0-1 Sorting Lemma) *If a sorting network with N inputs sorts all 2^N possible sequences of 0's and 1's correctly, then it sorts all input sequences of arbitrary numbers correctly.*

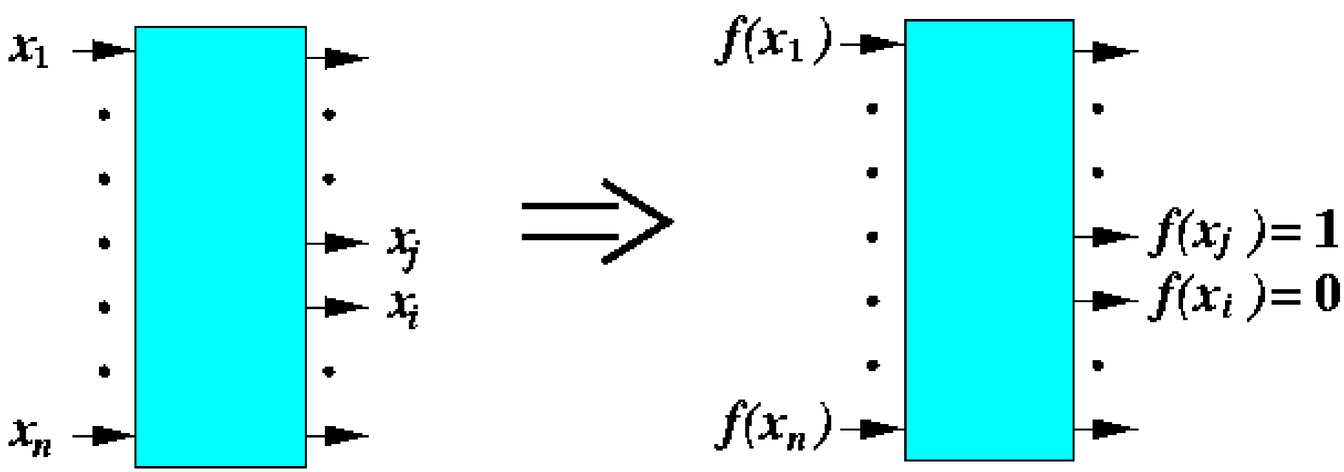
PROOF. By contradiction.

- Assume that the network sorts all zero-one sequences, but there exists a sequence $X = x_1, ..., x_n$ of arbitrary numbers such that the network fails to sort correctly.
- This means that the input sequence contains two input numbers x_i and x_j, with $x_i < x_j$, that the network places in wrong order on the output wires, i.e., x_j is placed before x_i in the output sequence.
- Let us define function f by

$$f(x) = \begin{cases} 0 & \text{if} \quad x \leq x_i, \\ 1 & \text{if} \quad x > x_i. \end{cases}$$

- Then f is clearly monotonically increasing.

- Since the network places x_j before x_i in the output sequence when input X is applied, it follows from Lemma that it places $f(x_j) = 1$ before $f(x_i) = 0$ when the input sequence $f(X)$ is applied.
- But f(X) is a zero-one sequence — a contradiction.



$\square$

- Even-odd transposition sorting on 1-D meshes
- ShearSort on 2-D meshes
- Lower bound on 2-D mesh snakelike sorting
- Lexicographic sorting on 3-D meshes

This algorithm is also called **parallel bubble-sort**. It takes precisely N steps to sort N numbers on a 1-D mesh $M(N)$ with N vertices, which is optimal, due to an obvious **diameter argument**. The idea is obvious: alternate CE operations between odd-even and even-odd adjacent pairs of nodes. The cost of the algorithm is therefore $O(N^2)$.

Algorithm 1 *Let $X = x_1, .., x_N$ be a sequence of numbers to be sorted rightward in ascending order. Then, the following algorithm, called* even-odd transposition*, sorts X on $M(N)$ in N steps.*

> **for** $j = 1...\lfloor N/2 \rfloor$ **do_sequentially**
>   **for** $i = 1, 3, .., 2\lfloor N/2 \rfloor - 1$ **do_in_parallel**
>   **if** $x_i > x_{i+1}$ **then** exchange(x_i, x_{i+1})
>   **for** $i = 2, 4, .., 2\lfloor (N-1)/2 \rfloor$ **do_in_parallel**
>   *if* $x_i > x_{i+1}$ **then** exchange(x_i, x_{i+1})

PROOF. We use the 0-1 Sorting Lemma. Let us assume an input sequence of k 1's and $N-k$ 0's, for any $1 \le k \le N-1$, arbitrarily scrambled. By induction on k, we can show that the algorithm will move the k 1's into positions $N-k+1,\ldots,N$ within N steps.

- The rightmost 1 has only 0's on its right hand, so it starts moving to the right at latest in the second step and keeps moving without interruption until it reaches its final position.
- The second rightmost 1 starts moving at latest in the third step, and proceeds similarly.
- Finally, the k-th rightmost 1 starts moving at latest in $(k+1)$-th step to reach position $N-k+1$.
- Therefore, in the worst case, the total number of steps actually needed is $k+(N-k)=N$.
- Since the algorithm performs N parallel steps in any case, the algorithm is correct.

Observe that the complexity analysis is easy. $\square$

Consider an even-odd transposition sort of $N > p$ numbers on p processors.

Phase 1. Each of the p nodes sorts N/p numbers

- $\Theta((N/p)log(N/p))$ CE operations in each processor and
- communication latency of an exchange of N/p numbers between adjacent nodes is also $\Theta(N/p)$,

Phase 2. The parallel bubble-sort algorithm is run of p nodes.

- During each of the p iterations of the algorithm,
- p nodes perform (sequentially) together p MS operation with N/p numbers.

The total parallel time is

$$T(N,p) = \Theta((N/p)log(N/p)) + \Theta(pN/p) = \Theta((N/p)log(N/p)) + \Theta(N).$$

Hence,

- the parallel time is $\Omega(N)$ (by a diameter argument),
- the parallel time is $O(N)$ only if $p = \Omega(\log N)$ (from the above

formula),

- Recall that we have

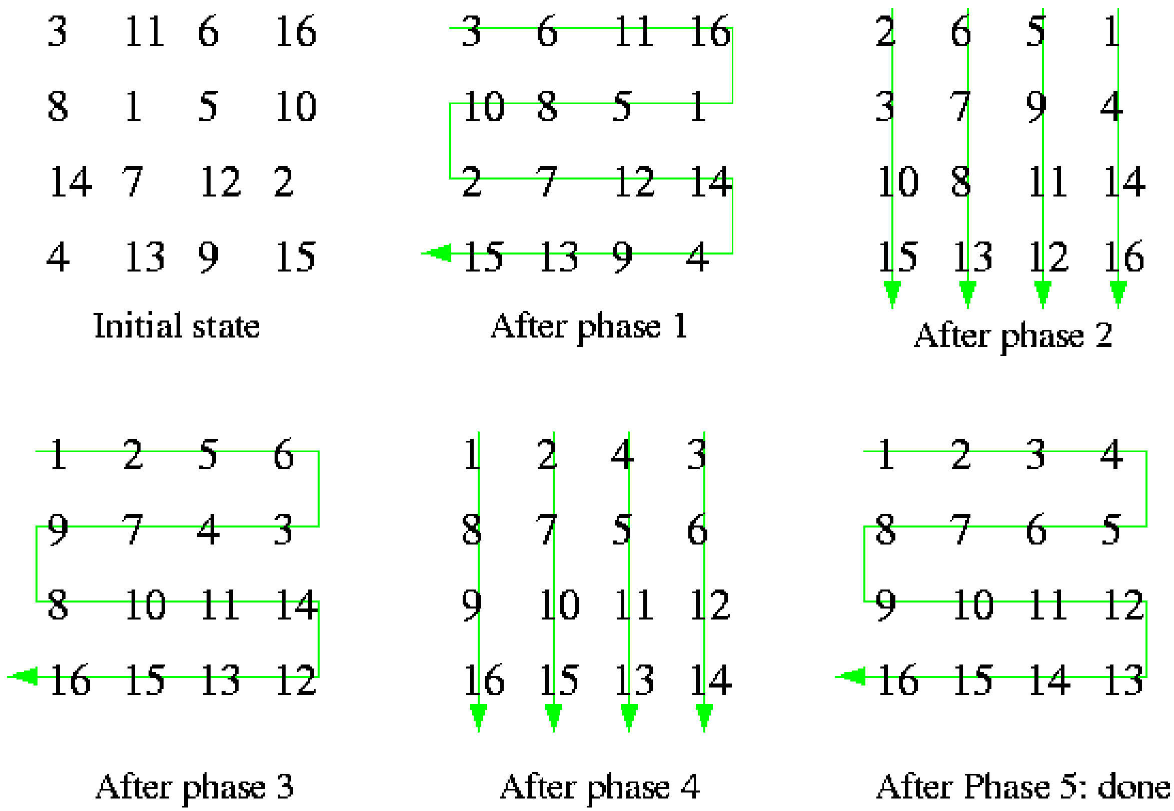
$$E(N,p) = \frac{SU(N)}{p\,T(N,p)} \quad \text{and} \quad SU(N) = \Omega(N \log N).$$

- showing that

$$E(N,p) = O(1) \quad \Longleftrightarrow \quad p = \Omega(logN).$$

Therefore, the scalability of even-odd transposition sort is very poor, since to keep a constant efficiency, input data size must grow exponentially with the number of processors.

The simplest 2-D mesh sorting algorithm is called ShearSort and it is based on the even-odd transposition. It works for any 2-D mesh.



Shearsort on $M(n, m)$ consists of $2logn + 1$ phases:

- In odd-numbered phases, individual rows are sorted alternately rightward and leftward.
- In even-numbered phases, individual columns are sorted downward.
- In both cases, we can use even-odd transposition sorting.

Algorithm 2 *Let* $X = x_1, .., x_N$ *be a sequence of numbers with* $N = nm$. *The* ShearSort *algorithm below needs* $\lceil logn \rceil + 1$ *row phases and* $\lceil logn \rceil$ *column phases to sort* X *on a 2-D mesh* $M(n, m)$ *with* n *rows and* m *columns, in a snakelike order. Moreover, one row (column) phase takes* $\Theta(n)$ *(*$\Theta(m)$*, respectively) computational and communication steps.*

```
for i = 1...2⌊logn⌋ + 1 do_sequentially
    if (i is odd)
        then SORT_ALL_ROWS_SNAKELIKE
        else SORT_ALL_COLUMNS
```

Proof. Consider any zero-one $n \times m$ matrix. It may contain rows of three kinds.

- all-one rows containing only 1's,
- all-zero rows containing only 0's,
- dirty rows containing both 0's and 1's.

Observe that

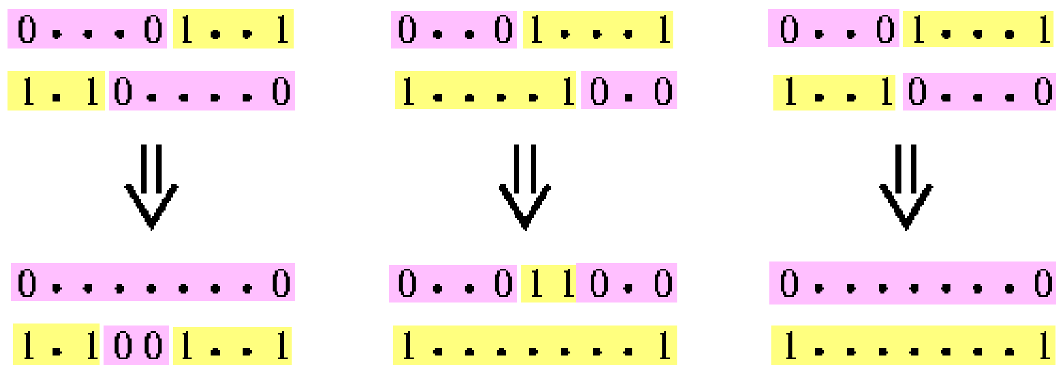
- Initially, the input matrix can contain n dirty rows in the worst case.
- The final matrix sorted snakelike contains at most one dirty row.

The following lemma helps conclude the proof. Indeed, this lemma implies that after $2logn$ phases, at most one dirty row remains and one more *row-sort* completes sorting. □

Lemma 3 *In Algorithm 2, one row phase plus one column phase reduce the number of dirty rows to at least one half.*

PROOF. By a simple case analysis. Consider all dirty rows after one row phase:

- One half of them is sorted rightward, whereas the other half reversely.
- If we consider pairs of leftward and rightward rows, there can exist pairs of at most three different types, as shown on figure below. □



(a) (b) (c)

- It is noteworthy that if the rows were sorted in a row-major order instead of in a snakelike order, then the algorithm will not sort.
- Unfortunately, ShearSort is not optimal. Improvements based on decreasing number of dirty rows help only a little. (Explain!)

Consider sorting N numbers on $p = q^2$ processors.

- Each node sorts N/p numbers in $\Theta((N/p)log(N/p))$ CE operations.
- Algorithm 2 requires $\lceil logq \rceil + 1$ row/column phases
- One row/column phasis takes $\Theta(q)$ MS operations.
- One MS operation is in $\Theta(N/p)$.

Hence, the parallel time is

$$T(N,p) = \Theta((N/q^2)log(N/q^2)) + \Theta((N/q)\log q).$$

So, for $N = p = n^2$, we obtain

$$T(N,N) = \Theta(\sqrt{N}\log N) = \Theta(n\log n).$$

The next lemma shows that it is not possible to do much better with any oblivious snakelike sorting on a 2-D mesh.

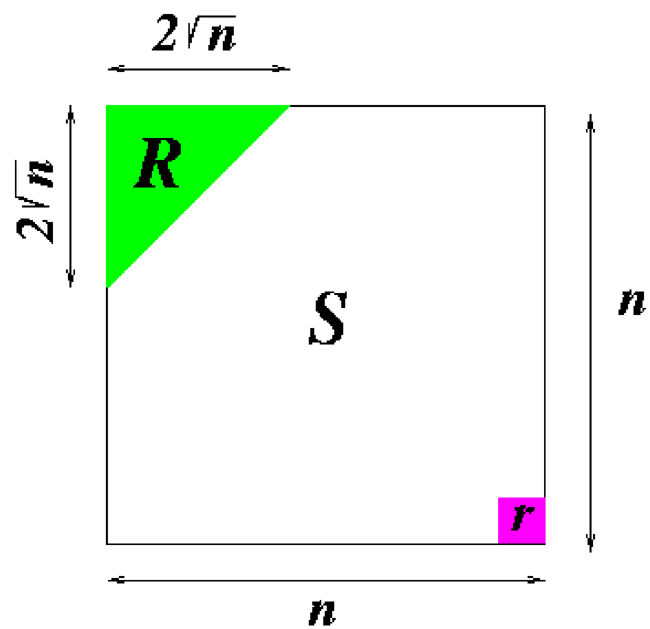
- The trivial lower bound on parallel sorting on a 2-D mesh M(n,n) is 2n-2, the diameter of the mesh.
- If the snakelike order is required, the lower bound on the number of CE steps for oblivious sorting on a 2-D mesh is greater than the trivial diameter-based one.

Lemma 4 *Any oblivious snakelike sorting on a 2-D mesh $M(n, n)$ requires at least $3n - 2\sqrt{n} - 4$ CE steps.*

PROOF. Let $N = n^2$. Assume any snakelike-order sorting algorithm A.

- Partition $M(n, n)$ into the upper left triangular region R with sides $2\sqrt{n} \times 2\sqrt{n}$ (i.e., comprising of $2n$ nodes) and the rest of the mesh, called region S.
- Consider the following input data:
  - each node in region S holds initially an arbitrary number from set $1, \ldots, N - 1$,
  - each node in region R holds either 0 or N.

- The initial configuration of input data on M(n,n) is shown by the figure below.



- Let r be the contents of the lower right corner node in step $2n - 2\sqrt{n} - 3$ of Algorithm A.
- Since the distance between this corner and any node in region R is

$2n - 2\sqrt{n} - 2$, the value of r could not be affected by any number initially located in region R, i.e., r is independent on whatever the contents of R initially was.

- For $0 \leq q \leq 2n$, let $C(q)$ be the **final** column for r if R contains **initially** q 0's and $2n - q$ N's.
- Note that in a row-major snakelike ordering starting in the upper left corner, the head of the snake is either in the left or right lower corner of the mesh, depending on the parity of n.
- As we vary q between 0 and $2n$, the value of $C(q)$ varies between 1 and n, achieving each possible value between 1 and n at least twice.
- Therefore, for n odd or even, we can always find q' such that $C(q') = 1$.
- It follows that by a suitable choice of input values, we can always force algorithm A to perform at least $n - 1$ steps after its step $2n - 2\sqrt{(n)} - 3$.

Consider the problem of sorting n^3 numbers on a 3-D mesh $M(n, n, n)$ in lexicographic $z > y > x$-order. Surprisingly, there is an algorithm which needs just five (!) phases, where one phase sorts numbers within 2-D planes. It works as follows.

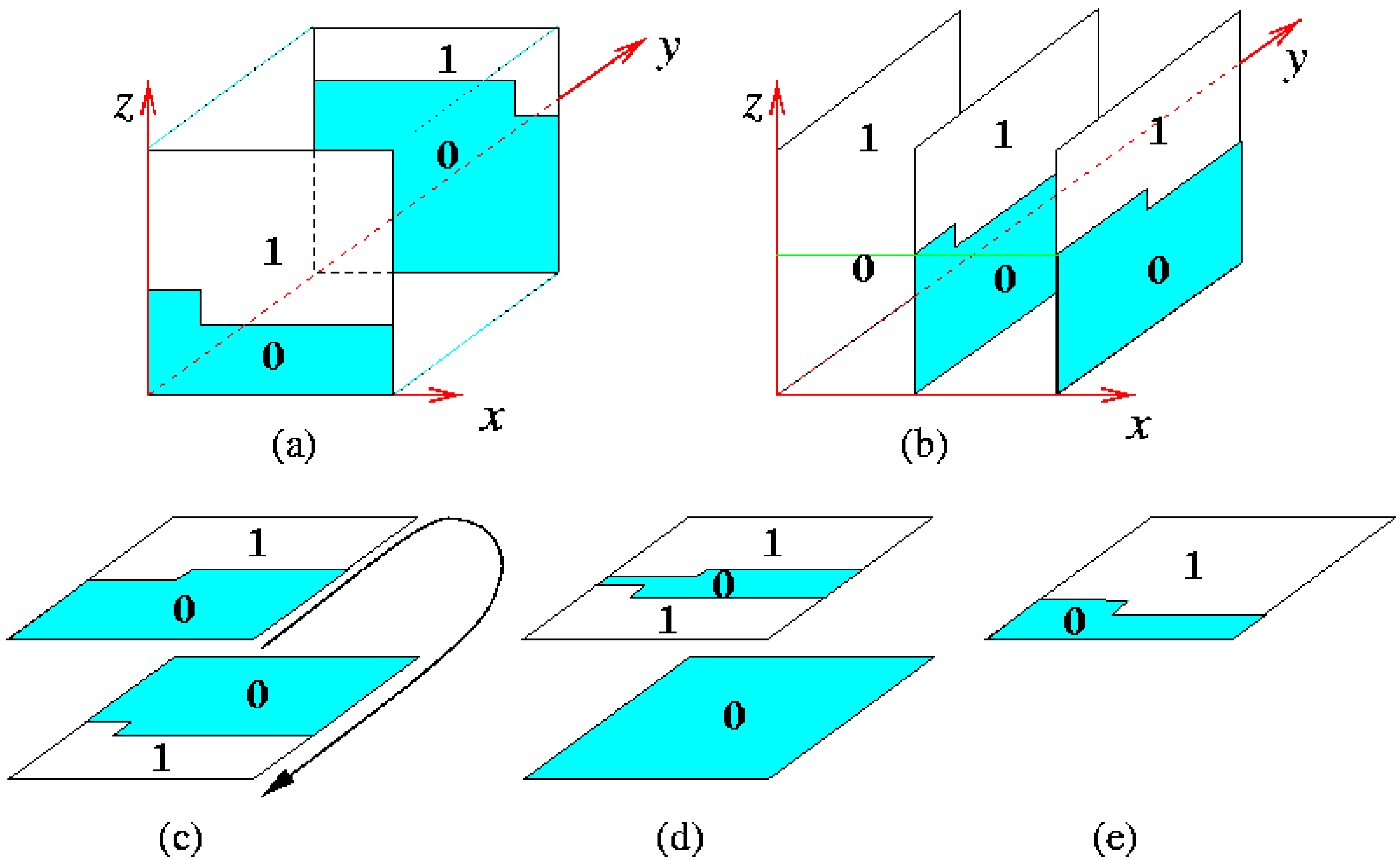
Phase 1. Sort all xz-planes in $z > x$-order;

Phase 2. Sort all yz-planes in $z > y$-order;

Phase 3. Sort all xy-planes in $y > x$-order snakelike (in direction y);

Phase 4. Perform one odd-even and one even-odd transposition within all columns in parallel;

Phase 5. Sort all xy-planes in $y > x$-order.


z
y
x
1
0
1
0
(a)
z
y
x
1
1
1
0
0
0
(b)
1
0
0
1
(c)
1
0
1
0
(d)
1
0
(e)

Proof. We use the 0-1 Sorting Lemma. Therefore, we consider any input sequence of 0's and 1's.

Phase 1. See Figure (a). After Phase 1:

- In every xz-plane, there is at most one dirty row.
- $\boxed{\text{Therefore, any two } yz\text{-planes can differ in at most } n \text{ 0's.}}$
  Indeed, if the planes $x = x_0$ and $x = x_1$ differ at $(n+1)$ 0's, then two of these 0's have the same y-coordinate, say y_0. Then in the plane $y = y_0$, there are two dirty rows.
- Thus, the difference between the numbers of 0's in two yz-planes is at most n. This will remain true after Phase 2.

Phase 2. See Figure (b). After Phase 1, due to the previous remarks:

- $\boxed{\text{all dirty rows of } yz\text{-planes can span at most 2 } xy\text{-plane.}}$

Phases 3 and 4. (See Figures (c) and (d)) If there are 2 dirty xy-planes, Phases 3 and 4 eliminate at least one.

After phase 5. (See Figure (e)) Phase 5 completes the sorting.

Even-Odd MergeSort (EOMS) sorts by recursively merging larger and larger monotonically sorted sequences. Consider an input unsorted sequence of $2N$ numbers, where N is a power of 2. Principle:

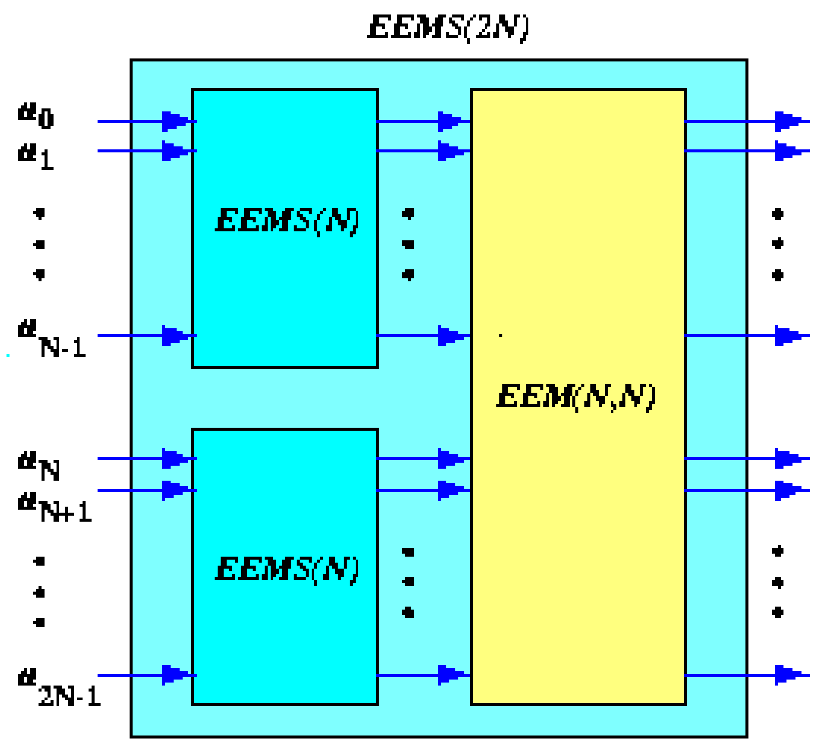
- EOMS treats the input sequence as a sequence of N pairs.
- By passing through a comparator, each pair becomes a sorted sequence of length 2.
- These subsequences are then merged into $N/2$ sorted subsequences of length 4, and so on.
- In the last merge step, two ascending sequences of length N are merged into the final ascending sequence of length $2N$.

Clearly, it takes $\lceil logN \rceil$ parallel merge operations to sort $2N$ numbers and the length of merged sequences doubles each time.

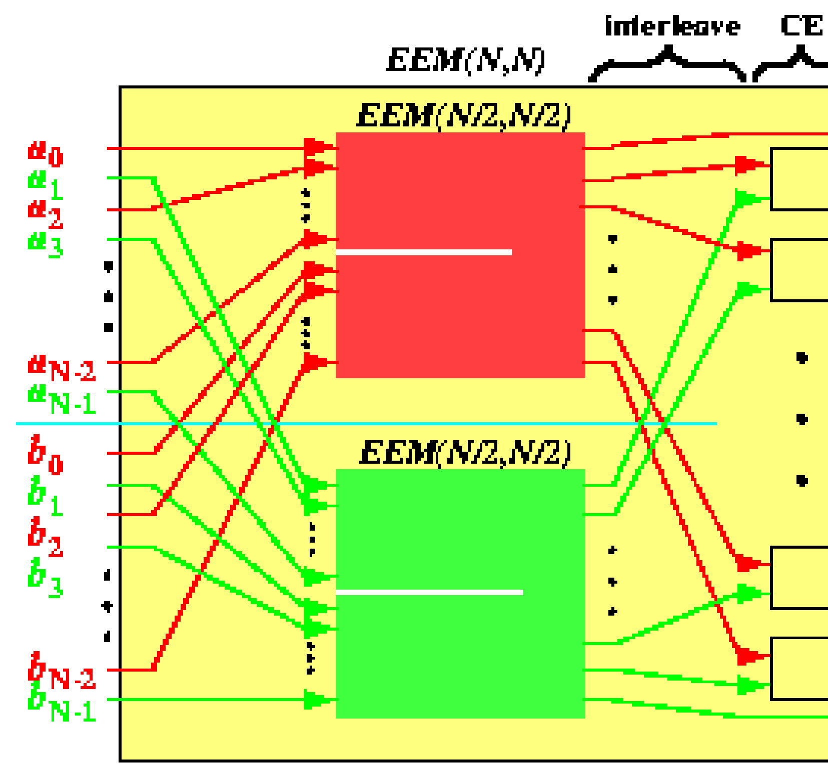
An Even-Odd MergeSort (EOMS) network for sorting $2N$ numbers

- is basically a cascade of $\lceil logN \rceil$ Even-Odd Merge networks.
- has obviously recursive structure as implied by the figure below, where
  - EOMS(k) denotes the EOMS network for sorting input sequences of length k, and
  - EOM(k, k) denotes the network for Even-Odd Merge of two sequences of length k each.

$$\begin{aligned}\text{EOMergeSort}(a_0, \ldots, a_{2N-1}) = \text{EOMerge} \quad & (\text{EOMergeSort}(a_0, \ldots, a_{N-1}), \\ & \text{EOMergeSort}(a_N, \ldots, a_{2N-1})).\end{aligned}$$


EEMS(2N)
EEMS(N)
EEMS(N)
EEM(N,N)
a_0
a_1
a_{N-1}
a_N
a_{N+1}
a_{2N-1}

(a)


interleave
CE
EEM(N,N)
EEM(N/2,N/2)
EEM(N/2,N/2)
a_0
a_1
a_2
a_3
a_{N-2}
a_{N-1}
b_0
b_1
b_2
b_3
b_{N-2}
b_{N-1}

(b)

Clearly, the key trick of the EOMS algorithm is hidden in the merge network. Consider two sorted sequences $A = a_0, a_1, .., a_{N-1}$ and $B = b_0, b_1, .., b_{N-1}$. The sorted sequence $L = \text{EOMerge}(A, B)$ of length $2N$ can be constructed **recursively** as follows:

- Partition A and B into odd and even subsequences:
  - $\text{even}(A) = a_0, a_2, \ldots, a_{N-2}$ and $\text{odd}(A) = a_1, a_3, .., a_{N-1}$.
  - $\text{even}(B) = b_0, b_2, \ldots, b_{N-2}$ and $\text{odd}(B) = b_1, b_3, .., b_{N-1}$.

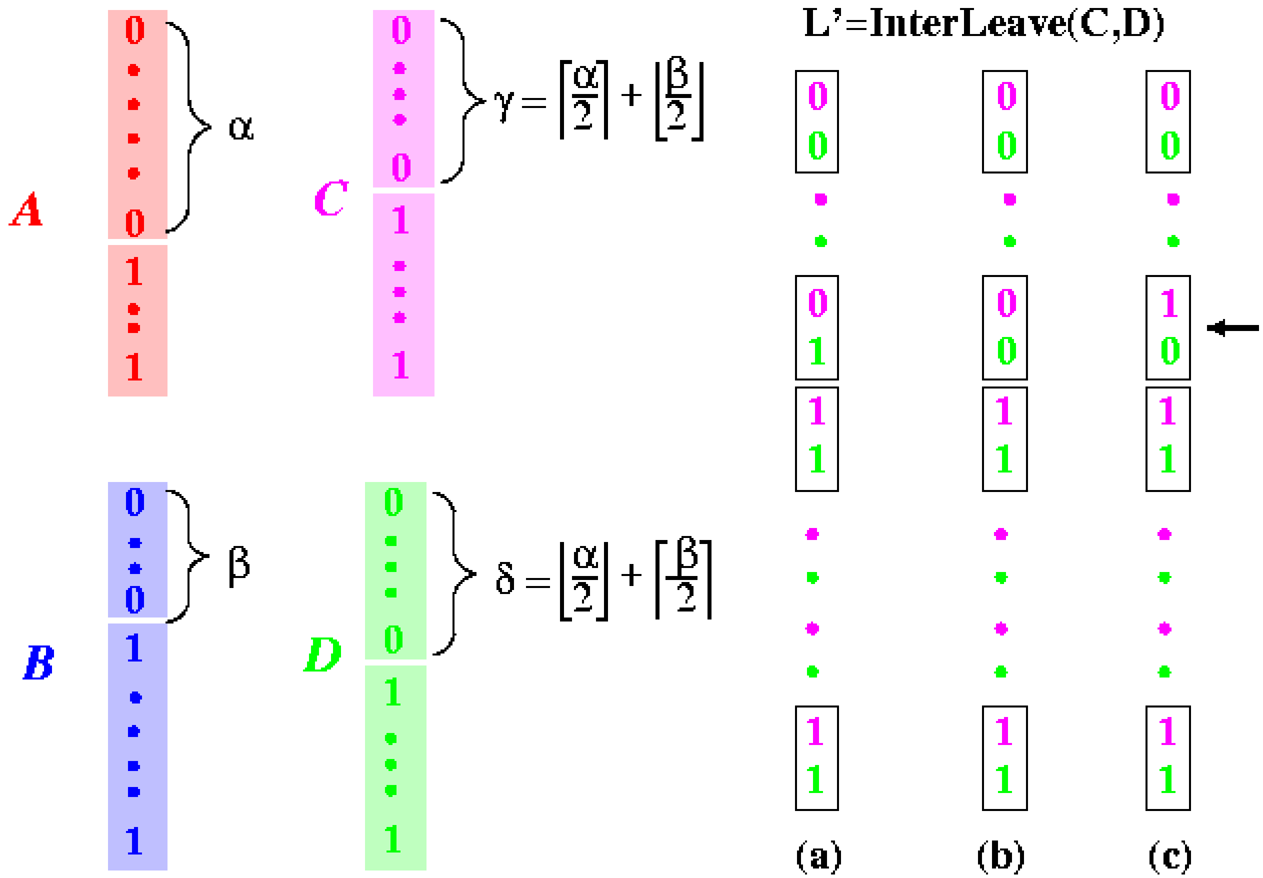
  Since A and B are sorted, so are their odd and even subsequences!
- Recursively merge:
  - $\text{even}(A)$ with $\text{odd}(B)$ into a sorted sequence $C = c_0, c_1, .., c_{N-1}$,
  - $\text{odd}(A)$ with $\text{even}(B)$ into a sorted sequence $D = d_0, d_1, .., d_{N-1}$.
- Interleave C and D to form a sequence $L' = c_0, d_0, c_1, d_1, .., c_{N-1}, d_{N-1}$. See the figure below.
- Apply compare-and-exchange to each pair c_i, d_i.

To summarize:

$$
\begin{array}{|l|}
\hline
\text{EOMerge}(A, B) == \\
\qquad C := \text{EOMerge}(\text{even}(A), \text{odd}(B)) \\
\qquad D : = \text{EOMerge}(\text{odd}(A), \text{even}(B)) \\
\qquad L' : = \text{Interleave}(C, D) \\
\qquad L := Pairwise_C E(L') \\
\\
\hline
\end{array}
$$

The resulting Even-Odd Merge network is depicted by the figure on the next slide. The main trick of this merge algorithm was discovered first by Batcher: to merge A and B, we just need to interleave C and D and we are almost done. Formally, it can be proven as follows.

Lemma 5 *Even-Odd Merge algorithm merges 2 sorted sequences of length N into a sorted sequence of length $2N$ in* $\log N + 1$ *parallel CE steps using* $N(\log N + 1)$ *comparators.*


A
B
C
D
α
β
γ = ⌊α/2⌋ + ⌈β/2⌉
δ = ⌈α/2⌉ + ⌊β/2⌋
L'=InterLeave(C,D)
(a)
(b)
(c)

Proof. We usethe $0-1$ Sorting Lemma. The lemma obviously holds for $N = 1$. Let $N = 2^k$, with $k \geq 1$. Consider two sorted binary sequences A and B of equal length N.

- Assume that A consists of α 0's followed by $N-\alpha$ 1's and
- B consists of β 0's followed by $N-\beta$ 1's.
- Then the sequence $C = \mathrm{EOMerge}(\mathrm{even}(A), \mathrm{odd}(B))$ has $\gamma = \lceil \alpha/2 \rceil + \lfloor \beta/2 \rfloor$ 0's,
- whereas the sequence $D = \mathrm{EOMerge}(\mathrm{odd}(A), \mathrm{even}(B))$ has $\delta = \lfloor \alpha/2 \rfloor + \lceil \beta/2 \rceil$ 0's.
- Since $|\gamma - \delta| \leq 1$, it follows that $\boxed{\text{the number of 0's in C and D can differ by at most one.}}$
- So, if we form

$$L' = \mathrm{Interleave}(C, D) = c_0, d_0, c_1, d_1, .., c_{N-1}, d_{N-1},$$

only three cases can occur:

- $\gamma - \delta = 1$. Then C has one additional 0 compared to D and L' is already sorted.
- $\gamma - \delta = 0$. Then C and D have exactly the same number of 0's and L' is sorted as well.
- $\gamma - \delta = -1$. Then the number of 0's in C is by one less than in D and we get only one incorrectly sorted pair, namely $(c_\gamma, d_\gamma) = (1, 0)$ in L'. By applying the compare-and-exchange operation to this pair, we get a sorted sequence L.

Let us prove the complexity. Let $d_m(2N)$ be the depth of EOM(N, N).

- For $N = 1$, we can merge using one comparator, hence $d_m(2) = 1$.
- Since each level of recursion adds just one column of comparators, we have $d_m(2N) = d_m(N) + 1$ and thus $d_m(2N) = \log N + 1 = \log(2N)$.
- Finally, the number of comparators is $c_m(2N) = N\, d_m(2N) = N \log(2N)$.

$\square$

Lemma 6 *EOMS sorts N numbers in $O(\log^2(N))$ parallel CE steps using $O(N\log^2(N))$ comparators.*

Proof. Let $d_s(N)$ denote the depth of EOMS(N). Then, $d_s(2) = 1$ and $d_s(2N) = d_s(N) + d_m(2N)$. Since $d_m(N) = \log N$, it follows that $d_s(N) = \log N(\log N + 1)/2$. For the number of comparators $c_s(N)$, we have similarly $c_s(2) = 1$ and $c_s(2N) = 2c_s(N) + c_m(2N)$. Since $c_m(2N) = N\, d_m(2N) = N \log(2N)$, we obtain $c_s(N) = O(N\log^2 N)$. $\square$