Chapter 11

Distributed Sorting

> "Indeed, I believe that virtually *every* important aspect of programming arises somewhere in the context of sorting [and searching]!"
>
> – Donald E. Knuth, The Art of Computer Programming

In this chapter we study a classic problem in computer science—sorting—from a distributed computing perspective. In contrast to an orthodox single-processor sorting algorithm, no node has access to all data, instead the to-be-sorted values are *distributed*. Distributed sorting then boils down to:

Definition 11.1 (Sorting)**.** *We are given a graph with n nodes $v_1, \ldots, v_n$. Initially each node stores a value. After applying a sorting algorithm, node v_k stores the k^{th} smallest value.*

Remarks:

- What if we route all values to the same central node v, let v sort the values locally, and then route them to the correct destinations?! According to the message passing model studied in the first few chapters this is perfectly legal. With a star topology sorting finishes in $O(1)$ time!

- Indeed, if we allow the All-to-All model of Chapter 10 we can even sort n values in a single round! So we need to make sure that we restrict our model appropriately:

Definition 11.2 (Node Contention)**.** *At any time, a node can only store $O(1)$ values. Alternatively, not quite as strong but sufficient: In each step of a synchronous algorithm each node can only send and receive $O(1)$ messages containing $O(1)$ values, no matter how many neighbors it has.*

Remarks:

- Using Definition 11.2 sorting on a star graph takes linear time.

11.1 Array & Mesh

To get a better intuitive understanding of distributed sorting, we start with two simple topologies, the array and the mesh. Let us begin with the array:

Algorithm 44 Odd/Even Sort

```
Given an array of $n$ nodes $(v_1, \dots, v_n)$, each storing a value (not sorted).
repeat
    Compare and exchange the values at nodes $i$ and $i+1$, $i$ odd
    Compare and exchange the values at nodes $i$ and $i+1$, $i$ even
until done
```

Remarks:

- The compare and exchange primitive in Algorithm 44 is defined as follows: Let the value stored at node i be v_i. After the compare and exchange node i stores value $\min(v_i, v_{i+1})$ and node $i+1$ stores value $\max(v_i, v_{i+1})$.
- How fast is the algorithm, and how can we prove correctness/efficiency?
- The most interesting proof uses the so-called 0-1 Sorting Lemma. It allows us to restrict our attention to an input of 0's and 1's only, and works for any "oblivious comparison-exchange" algorithm. (Oblivious means: Whether you exchange two values must only depend on the relative order of the two values, and not on anything else.)

Lemma 11.3 (0-1 Sorting Lemma)**.** *If an oblivious comparison-exchange algorithm sorts all inputs of 0's and 1's, then it sorts any input.*

Proof. We prove the opposite direction (does not sort any input ⇒ does not sort 0's and 1's). Assume that there is an input $x = x_1, \dots, x_n$ that is not sorted correctly. Then there is a smallest value k such that the value at node v_k after running the sorting algorithm is larger than the k^{th} smallest value $x(k)$. Define an input $x_i^* = 0 \Leftrightarrow x_i \leq x(k)$, $x_i^* = 1$ else. Since $x_i \geq x_j \Rightarrow x_i^* \geq x_j^*$ all the compare-exchange operation are the same with x^* as with the original input x. The output with only 0's and 1's will also not be correct. □

Theorem 11.4. *Algorithm 44 sorts correctly in n steps.*

Proof. Thanks to Lemma 11.3 we only need to consider an array with 0's and 1's. Let j_1 be the node with the rightmost (highest index) 1. If j_1 is odd (even) it will move in the first (second) step. In any case it will move right in every following step until it reaches the rightmost node v_n. Let j_k be the node with the k^{th} rightmost 1. We show by induction that j_k is not "blocked" anymore (constantly moves until it reaches destination!) after step k. We have already anchored the induction at $k = 1$. Since j_{k-1} moves after step $k-1$, j_k gets a right 0-neighbor for each step after step k. (For simplicity we omitted a couple of simple details.) □

Algorithm 45 Shearsort

```
We are given a mesh with $m$ rows and $m$ columns, $m$ even, $n = m^2$.
The sorting algorithm operates in phases, and uses the odd/even sort algorithm on rows or columns.
repeat
    In the odd phases $1, 3, \ldots$ we sort all the rows, in the even phases $2, 4, \ldots$ we sort all the columns, such that:
    Columns are sorted such that the small values move up.
    Odd rows $(1, 3, \ldots, m-1)$ are sorted such that small values move left.
    Even rows $(2, 4, \ldots, m)$ are sorted such that small values move right.
until done
```

Remarks:

- Linear time is not very exciting, maybe we can do better by using a different topology? Let's try a mesh (a.k.a. grid) topology first.

Theorem 11.5. *Algorithm 45 sorts n values in $\sqrt{n}(\log n+1)$ time in snake-like order.*

Proof. Since the algorithm is oblivious, we can use Lemma 11.3. We show that after a row and a column phase, half of the previously unsorted rows will be sorted. More formally, let us call a row with only 0's (or only 1's) *clean*, a row with 0's *and* 1's is *dirty*. At any stage, the rows of the mesh can be divided into three regions. In the north we have a region of all-0 rows, in the south all-1 rows, in the middle a region of dirty rows. Initially all rows can be dirty. Since neither row nor column sort will touch already clean rows, we can concentrate on the dirty rows.

First we run an odd phase. Then, in the even phase, we run a peculiar column sorter: We group two consecutive dirty rows into pairs. Since odd and even rows are sorted in opposite directions, two consecutive dirty rows look as follows:

00000 . . . 11111

11111 . . . 00000

Such a pair can be in one of three states. Either we have more 0's than 1's, or more 1's than 0's, or an equal number of 0's and 1's. Column-sorting each pair will give us at least one clean row (and two clean rows if "$|0| = |1|$"). Then move the cleaned rows north/south and we will be left with half the dirty rows.

At first glance it appears that we need such a peculiar column sorter. However, any column sorter sorts the columns in exactly the same way (we are very grateful to have Lemma 11.3!).

All in all we need $2\log m = \log n$ phases to remain only with 1 dirty row in the middle which will be sorted (not cleaned) with the last row-sort. □

Remarks:

- There are algorithms that sort in $3m + o(m)$ time on an m by m mesh (by diving the mesh into smaller blocks). This is asymptotically optimal, since a value might need to move $2m$ times.
- Such a $\sqrt{n}$-sorter is cute, but we are more ambitious. There are non-distributed sorting algorithms such as quicksort, heapsort, or mergesort that sort n values in (expected) $O(n \log n)$ time. Using our n-fold parallelism effectively we might therefore hope for a distributed sorting algorithm that sorts in time $O(\log n)$!

11.2 Sorting Networks

In this section we construct a graph topology which is carefully manufactured for sorting. This is a deviation to previous chapters where we always had to work with the topology that was given to us. In many application areas (e.g. peer-to-peer networks, communication switches, systolic hardware) it is indeed possible (in fact, crucial!) that an engineer can build the topology best suited for her application.

Definition 11.6 (Sorting Networks)**.** *A* comparator *is a device with two inputs x, y and two outputs x', y' such that $x' = min(x, y)$ and $y' = max(x, y)$. We construct so-called* comparison networks *that consist of wires that connect comparators (the output port of a comparator is sent to an input port of another comparator). Some wires are not connected to output comparators, and some are not connected to input comparators. The first are called input wires of the comparison network, the second output wires. Given n values on the input wires, a* sorting network *ensures that the values are sorted on the output wires.*

Remarks:

- The odd/even sorter explained in Algorithm 44 can be described as a sorting network.
- Often we will draw all the wires on n horizontal lines (n being the "width" of the network). Comparators are then vertically connecting two of these lines.
- Note that a sorting network is an oblivious comparison-exchange network. Consequently we can apply Lemma 11.3 throughout this section. An example sorting network is depicted in Figure 11.1.

Definition 11.7 (Depth)**.** *The depth of an input wire is* 0*. The depth of a comparator is the maximum depth of its input wires plus one. The depth of an output wire of a comparator is the depth of the comparator. The depth of a comparison network is the maximum depth (of an output wire).*

Definition 11.8 (Bitonic Sequence)**.** *A* bitonic sequence *is a sequence of numbers that first monotonically increases, and then monotonically decreases, or vice versa.*

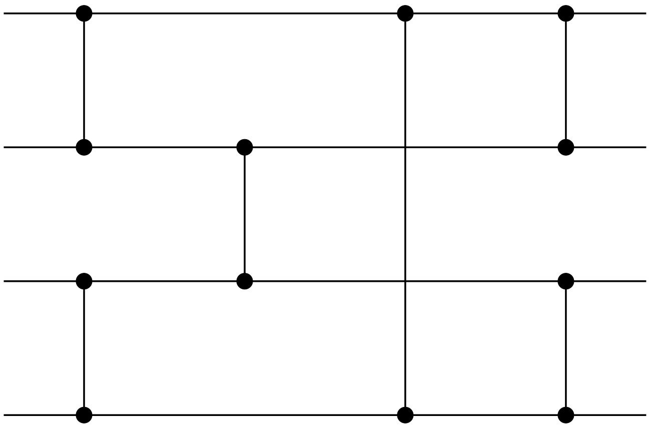

Figure 11.1: A sorting network.

Remarks:

- $< 1, 4, 6, 8, 3, 2 >$ or $< 5, 3, 2, 1, 4, 8 >$ are bitonic sequences.
- $< 9, 6, 2, 3, 5, 4 >$ or $< 7, 4, 2, 5, 9, 8 >$ are not bitonic.
- Since we restrict ourselves to 0's and 1's (Lemma 11.3), bitonic sequences have the form $0^i1^j0^k$ or $1^i0^j1^k$ for $i, j, k \geq 0$.

Algorithm 46 Half Cleaner

```
1: A half cleaner is a comparison network of depth 1, where we compare wire
   $i$ with wire $i + n/2$ for $i = 1, \ldots, n/2$ (we assume $n$ to be even).
```

Lemma 11.9. *Feeding a bitonic sequence into a half cleaner (Algorithm 46), the half cleaner cleans (makes all-*0 *or all-*1*) either the upper or the lower half of the n wires. The other half is bitonic.*

Proof. Assume that the input is of the form $0^i1^j0^k$ for $i, j, k \geq 0$. If the midpoint falls into the 0's, the input is already clean/bitonic and will stay so. If the midpoint falls into the 1's the half cleaner acts as Shearsort with two adjacent rows, exactly as in the proof of Theorem 11.5. The case $1^i0^j1^k$ is symmetric. □

Algorithm 47 Bitonic Sequence Sorter

```
1: A bitonic sequence sorter of width $n$ ($n$ being a power of 2) consists of a
   half cleaner of width $n$, and then two bitonic sequence sorters of width $n/2$
   each.
2: A bitonic sequence sorter of width 1 is empty.
```

Lemma 11.10. *A bitonic sequence sorter (Algorithm 47) of width n sorts bitonic sequences. It has depth* $\log n$*.*

Proof. The proof follows directly from the Algorithm 47 and Lemma 11.9. □

Remarks:

- Clearly we want to sort arbitrary and not only bitonic sequences! To do this we need one more concept, merging networks.

Algorithm 48 Merging Network

```
1: A merging network of width $n$ is a merger followed by two bitonic sequence sorters of width $n/2$. A merger is a depth-one network where we compare wire $i$ with wire $n - i + 1$, for $i = 1, \dots, n/2$.
```

Remarks:

- Note that a merging network is a bitonic sequence sorter where we replace the (first) half-cleaner by a merger.

Lemma 11.11. *A merging network (Algorithm 48) merges two sorted input sequences into one.*

Proof. We have two sorted input sequences. Essentially, a merger does to two sorted sequences what a half cleaner does to a bitonic sequence, since the lower part of the input is reversed. In other words, we can use same argument as in Theorem 11.5 and Lemma 11.9: Again, after the merger step either the upper or the lower half is clean, the other is bitonic. The bitonic sequence sorters complete sorting. □

Remarks:

- How do you sort n values when you are able to merge two sorted sequences of size $n/2$? Piece of cake, just apply the merger recursively.

Algorithm 49 Batcher's "Bitonic" Sorting Network

```
1: A batcher sorting network of width $n$ consists of two batcher sorting networks of width $n/2$ followed by a merging network of width $n$. (See Figure 11.2.)
2: A batcher sorting network of width 1 is empty.
```

Theorem 11.12. *A sorting network (Algorithm 49)sorts an arbitrary sequence of n values. It has depth $O(\log^2 n)$.*

Proof. Correctness is immediate: at recursive stage k ($k = 2, 4, 8, \dots, n$) we merge $n/(2k)$ sorted sequences into n/k sorted sequences. The depth $d(n)$ of the sorter of level n is the depth of a sorter of level $n/2$ plus the depth $m(n)$ of a merger with width n. The depth of a sorter of level 1 is 0 since the sorter is empty. Since a merger of width n has the same depth as a bitonic sequence sorter of width n, we know by Lemma 11.10 that $m(n) = \log n$. This gives a recursive formula for $d(n)$ which solves to $d(n) = \frac{1}{2}\log^2 n + \frac{1}{2}\log n$. □

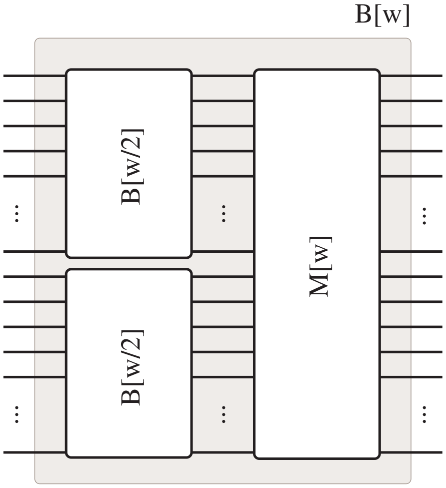

Figure 11.2: A batcher sorting network

Remarks:

- Simulating Batcher's sorting network on an ordinary sequential computer takes time $O(n \log^2 n)$. As said, there are sequential sorting algorithms that sort in asymptotically optimal time $O(n \log n)$. So a natural question is whether there is a sorting network with depth $O(\log n)$. Such a network would have some remarkable advantages over sequential asymptotically optimal sorting algorithms such as heapsort. Apart from being highly parallel, it would be completely oblivious, and as such perfectly suited for a fast hardware solution. In 1983, Ajtai, Komlos, and Szemeredi presented a celebrated $O(\log n)$ depth sorting network. (Unlike Batcher's sorting network the constant hidden in the big-O of the "AKS" sorting network is too large to be practical, however.)

- It can be shown that Batcher's sorting network and similarly others can be simulated by a Butterfly network and other hypercubic networks, see next Chapter.

- What if a sorting network is asynchronous?!? Clearly, using a synchronizer we can still sort, but it is also possible to use it for something else. Check out the next section!

11.3 Counting Networks

In this section we address distributed counting, a distributed service which can for instance be used for load balancing.

Definition 11.13 (Distributed Counting)**.** *A* distributed counter *is a variable that is common to all processors in a system and that supports an atomic* test-and-increment *operation. The operation delivers the system's counter value to the requesting processor and increments it.*

Remarks:

- A naive distributed counter stores the system's counter value with a distinguished central node. When other nodes initiate the test-and-increment operation, they send a request message to the central node and in turn receive a reply message with the current counter value. However, with a large number of nodes operating on the distributed counter, the central processor will become a bottleneck. There will be a congestion of request messages at the central processor, in other words, the system will not scale.

- Is a scalable implementation (without any kind of bottleneck) of such a distributed counter possible, or is distributed counting a problem which is inherently centralized?!?

- Distributed counting could for instance be used to implement a load balancing infrastructure, i.e. by sending the job with counter value i (modulo n) to server i (out of n possible servers).

Definition 11.14 (Balancer)**.** *A balancer is an asynchronous flip-flop which forwards messages that arrive on the left side to the wires on the right, the first to the upper, the second to the lower, the third to the upper, and so on.*

Algorithm 50 Bitonic Counting Network.

```
Take Batcher's bitonic sorting network of width $w$ and replace all the comparators with balancers.
When a node wants to count, it sends a message to an arbitrary input wire.
The message is then routed through the network, following the rules of the asynchronous balancers.
Each output wire is completed with a "mini-counter."
The mini-counter of wire $k$ replies the value "$k + i \cdot w$" to the initiator of the $i^{th}$ message it receives.
```

Definition 11.15 (Step Property)**.** *A sequence $y_0, y_1, \ldots, y_{w-1}$ is said to have the* step property, *if $0 \leq y_i - y_j \leq 1$, for any $i < j$.*

Remarks:

- If the output wires have the step property, then with r requests, exactly the values $1, \ldots, r$ will be assigned by the mini-counters. All we need to show is that the counting network has the step property. For that we need some additional facts...

Facts 11.16. *For a balancer, we denote the number of consumed messages on the i^{th} input wire with x_i, $i = 0, 1$. Similarly, we denote the number of sent messages on the i^{th} output wire with y_i, $i = 0, 1$. A balancer has these properties:*

(1) A balancer does not generate output-messages; that is, $x_0 + x_1 \geq y_0 + y_1$ in any state.

(2) *Every incoming message is eventually forwarded. In other words, if we are in a quiescent state (no message in transit), then* $x_0 + x_1 = y_0 + y_1$*.*

(3) *The number of messages sent to the upper output wire is at most one higher than the number of messages sent to the lower output wire: in any state* $y_0 = \lceil (y_0 + y_1)/2 \rceil$ *(thus* $y_1 = \lfloor (y_0 + y_1)/2 \rfloor$*).*

Facts 11.17. *If a sequence* $y_0, y_1, \ldots, y_{w-1}$ *has the step property,*

(1) *then all its subsequences have the step property.*

(2) *then its even and odd subsequences satisfy*

$$\sum_{i=0}^{w/2-1} y_{2i} = \left\lceil \frac{1}{2} \sum_{i=0}^{w-1} y_i \right\rceil \text{ and } \sum_{i=0}^{w/2-1} y_{2i+1} = \left\lfloor \frac{1}{2} \sum_{i=0}^{w-1} y_i \right\rfloor.$$

Facts 11.18. *If two sequences* $x_0, x_1, \ldots, x_{w-1}$ *and* $y_0, y_1, \ldots, y_{w-1}$ *have the step property,*

(1) *and* $\sum_{i=0}^{w-1} x_i = \sum_{i=0}^{w-1} y_i$*, then* $x_i = y_i$ *for* $i = 0, \ldots, w-1$*.*

(2) *and* $\sum_{i=0}^{w-1} x_i = \sum_{i=0}^{w-1} y_i + 1$*, then there exists a unique* j *(*$j = 0, 1, \ldots, w-1$*) such that* $x_j = y_j + 1$*, and* $x_i = y_i$ *for* $i = 0, \ldots, w-1$*,* $i \neq j$*.*

Remarks:

- That's enough to prove that a Merger preserves the step property.

Lemma 11.19. *Let* $M[w]$ *be a Merger of width* w*. In a quiescent state (no message in transit), if the inputs* $x_0, x_1, \ldots, x_{w/2-1}$ *resp.* $x_{w/2}, x_{w/2+1}, \ldots, x_{w-1}$ *have the step property, then the output* $y_0, y_1, \ldots, y_{w-1}$ *has the step property.*

Proof. By induction on the width w.

For $w = 2$: $M[2]$ is a balancer and a balancer's output has the step property (Fact 11.16.3).

For $w > 2$: Let $z_0, \ldots, z_{w/2-1}$ resp. $z'_0, \ldots, z'_{w/2-1}$ be the output of the upper respectively lower $M[w/2]$ subnetwork. Since $x_0, x_1, \ldots, x_{w/2-1}$ and $x_{w/2}, x_{w/2+1}, \ldots, x_{w-1}$ both have the step property by assumption, their even and odd subsequences also have the step property (Fact 11.17.1). By induction hypothesis, the output of both $M[w/2]$ subnetworks have the step property. Let $Z := \sum_{i=0}^{w/2-1} z_i$ and $Z' := \sum_{i=0}^{w/2-1} z'_i$. From Fact 11.17.2 we conclude that $Z = \lceil \frac{1}{2} \sum_{i=0}^{w/2-1} x_i \rceil + \lfloor \frac{1}{2} \sum_{i=w/2}^{w-1} x_i \rfloor$ and $Z' = \lfloor \frac{1}{2} \sum_{i=0}^{w/2-1} x_i \rfloor + \lceil \frac{1}{2} \sum_{i=w/2}^{w-1} x_i \rceil$. Since $\lceil a \rceil + \lfloor b \rfloor$ and $\lfloor a \rfloor + \lceil b \rceil$ differ by at most 1 we know that Z and Z' differ by at most 1.

If $Z = Z'$, Fact 11.18.1 implies that $z_i = z'_i$ for $i = 0, \ldots, w/2-1$. Therefore, the output of $M[w]$ is $y_i = z_{\lfloor i/2 \rfloor}$ for $i = 0, \ldots, w-1$. Since $z_0, \ldots, z_{w/2-1}$ has the step property, so does the output of $M[w]$ and the Lemma follows.

If Z and Z' differ by 1, Fact 11.18.2 implies that $z_i = z'_i$ for $i = 0, \ldots, w/2-1$, except a unique j such that z_j and z'_j differ by only 1, for $j = 0, \ldots, w/2-1$. Let $l := min(z_j, z'_j)$. Then, the output y_i (with $i < 2j$) is $l + 1$. The output y_i (with $i > 2j + 1$) is l. The output y_{2j} and y_{2j+1} are balanced by the final balancer resulting in $y_{2j} = l + 1$ and $y_{2j+1} = l$. Therefore $M[w]$ preserves the step property. □

A bitonic counting network is constructed to fulfill Lemma 11.19, i.e., the final output comes from a Merger whose upper and lower inputs are recursively merged. Therefore, the following Theorem follows immediately.

Theorem 11.20 (Correctness)**.** *In a quiescent state, the w output wires of a bitonic counting network of width w have the step property.*

Remarks:

- Is every sorting networks also a counting network? No. But surprisingly, the other direction is true!

Theorem 11.21 (Counting vs. Sorting)**.** *If a network is a counting network then it is also a sorting network, but not vice versa.*

Proof. There are sorting networks that are not counting networks (e.g. odd/even sort, or insertion sort). For the other direction, let C be a counting network and $I(C)$ be the isomorphic network, where every balancer is replaced by a comparator. Let $I(C)$ have an arbitrary input of 0's and 1's; that is, some of the input wires have a 0, all others have a 1. There is a message at C's i^{th} input wire if and only if $I(C)$'s i input wire is 0. Since C is a counting network, all messages are routed to the upper output wires. $I(C)$ is isomorphic to C, therefore a comparator in $I(C)$ will receive a 0 on its upper (lower) wire if and only if the corresponding balancer receives a message on its upper (lower) wire. Using an inductive argument, the 0's and 1's will be routed through $I(C)$ such that all 0's exit the network on the upper wires whereas all 1's exit the network on the lower wires. Applying Lemma 11.3 shows that $I(C)$ is a sorting network. □

Remarks:

- We claimed that the counting network is correct. However, it is only correct in a quiescent state.

Definition 11.22 (Linearizable)**.** *A system is* linearizable *if the order of the values assigned reflects the real-time order in which they were requested. More formally, if there is a pair of operations o_1, o_2, where operation o_1 terminates before operation o_2 starts, and the logical order is "o_2 before o_1", then a distributed system is not linearizable.*

Lemma 11.23 (Linearizability)**.** *The bitonic counting network is not linearizable.*

Proof. Consider the bitonic counting network with width 4 in Figure 11.3: Assume that two *inc* operations were initiated and the corresponding messages entered the network on wire 0 and 2 (both in light gray color). After having passed the second resp. the first balancer, these traversing messages "fall asleep"; In other words, both messages take unusually long time before they are received by the next balancer. Since we are in an asynchronous setting, this may be the case.

In the meantime, another *inc* operation (medium gray) is initiated and enters the network on the bottom wire. The message leaves the network on wire 2, and the *inc* operation is completed.

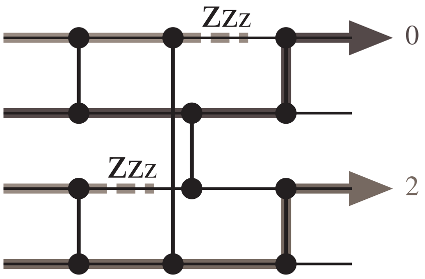

Figure 11.3: Linearizability Counter Example.

Strictly afterwards, another *inc* operation (dark gray) is initiated and enters the network on wire 1. After having passed all balancers, the message will leave the network wire 0. Finally (and not depicted in figure 11.3), the two light gray messages reach the next balancer and will eventually leave the network on wires 1 resp. 3. Because the dark gray and the medium gray operation do conflict with Definition 11.22, the bitonic counting network is not linearizable. □

Remarks:

- Note that the example in Figure 11.3 behaves correctly in the quiescent state: Finally, exactly the values $0, 1, 2, 3$ are allotted.

- It was shown that linearizability comes at a high price (the depth grows linearly with the width).