Massively Parallel Algorithms

Parallel Sorting

G. Zachmann

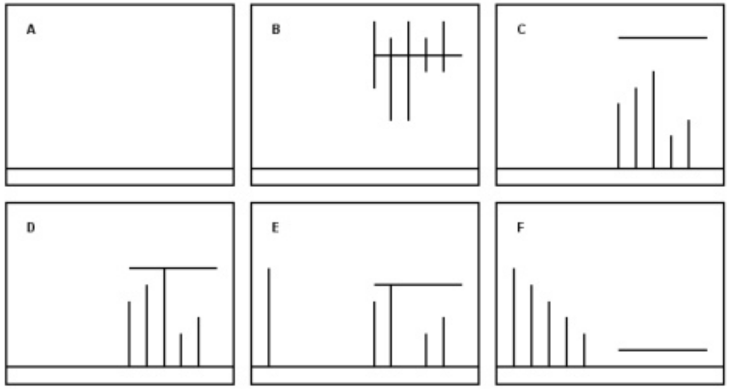
University of Bremen, Germany

cgvr.cs.uni-bremen.de

Sorting using Spaghetti in *O*(1) (?)

- Is *O*(*n*) really the lower bound for sorting?
- Consider the following thought experiment:
  - B. For each number *x* in the list, cut a spaghetto to length *x* → list = bundle of spaghetti & unary repr.
  - C. Hold the spaghetti loosely in your hand and tap them on the kitchen table → takes O(1) !
  - D. Lower your other hand from above until it meets with a spaghetto — this one is clearly the longest
  - E. Remove this spaghetto and insert it into the front of the output list
  - F. Repeat

- If we could use this *mechanical* computer, then sorting would be O(1)



Difficulties With Parallel Implementation of Standard Sequential Algorithms

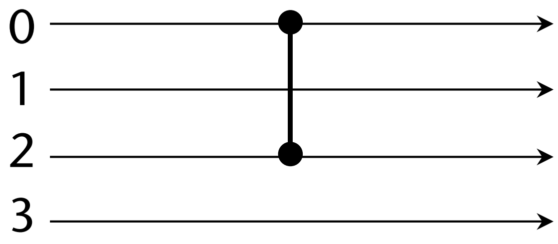
- Insertion sort:
  - Considers only one element at a time
- Quicksort:
  - Yes, some parallelism at lower levels of the recursion tree
  - But, would need *median* as a pivot element ⟶ hard to find
  - Otherwise, random pivot element causes varying sub-array sizes
- Heapsort:
  - Only one element at a time
  - Heap (= recursive data structure) is difficult on mass.-parallel architecture
- Radix sort:
  - Yes, we've seen that already, works well
  - But, can handle only fixed-length numbers

Assumptions

- In this chapter, we will always assume that $n = 2^k$
- Elements can have any type, for which there is a comparison operator

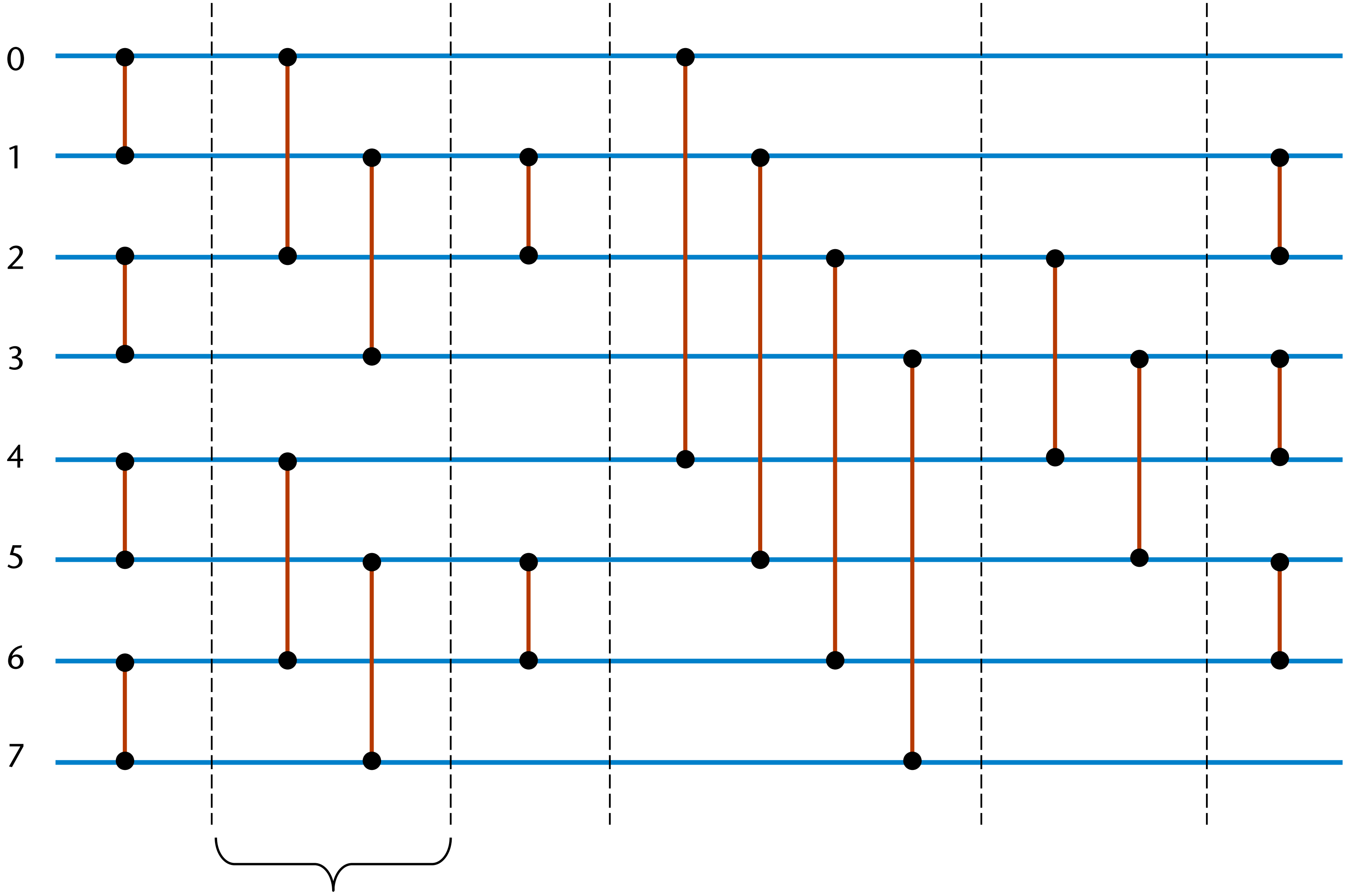
Sorting Networks

- Informal definition of comparator networks:
  - Consist of a bundle of "wires"
  - Each wire *i* carries a data element D_i (e.g., float) from left to right
  - Two wires can be connected vertically by a comparator
  - If $D_i > D_j \wedge i < j$ (i.e., wrong order), then D_i and D_j are swapped by the comparator before they move on along the wires



- Observation: every *comparator network* is *data independent*, i.e., the arrangement of comparators and the running time are always the same!
- Goal: find a "*small*" comparator network that performs sorting for *any* input → sorting network

Example



One stage / step

The 0-1 Principle

- Definition (*monotone function*):

  Let A, B be two sets with a total ordering relation,

  and let $f : A \rightarrow B$ be a mapping.

  f is called monotone iff

$$\forall a_1, a_2 \in A : \; a_1 \leq a_2 \Rightarrow f(a_1) \leq f(a_2)$$

- Lemma:

  Let $f : A \rightarrow B$ be monotone. Then, f and *min* commute, i.e.

$$\forall a_1, a_2 \in A : \; f(\min(a_1, a_2)) = \min(f(a_1), f(a_2))$$

  Analogously for the *max*.

- Proof:

  Case 1: $a_1 \leq a_2 \Rightarrow f(a_1) \leq f(a_2)$

$$\min(a_1, a_2) = a_1 \;, \; \min(f(a_1), f(a_2)) = f(a_1)$$

$$f(\min(a_1, a_2)) = f(a_1) = \min(f(a_1), f(a_2))$$

  Case 2: $a_2 < a_1 \rightarrow$ analog

- Extension of $f : A \rightarrow B$ to sequences over A and B, resp.:

$$f\,(a_0, \ldots, a_n) = f(a_0), \ldots, f(a_n)$$

- Lemma:

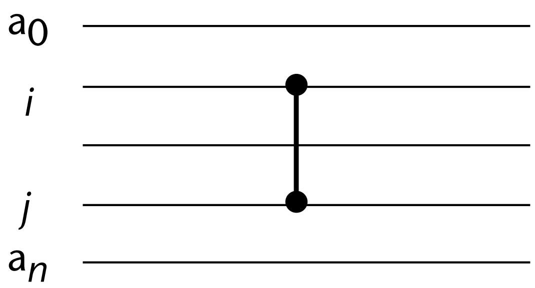
  Let f be a monotone mapping and $\mathcal{N}$ a comparator network.

  Then $\mathcal{N}$ and f commute, i.e.

$$\forall n\, \forall a_0, \ldots, a_n : \ \mathcal{N}\big(f(a)\big) = f\big(\mathcal{N}(a)\big)$$

- Proof:
  - Let $a = (a_0, \ldots, a_n)$ be a sequence
  - Notation: we write a comparator connecting wire i and j like so:

$$[i:j](a)$$



  - Now the following is true:

$$\begin{aligned}
[i:j]\big(f(a)\big) &= [i:j]\big(f(a_0), \ldots, f(a_n)\big) \\
&= \big(f(a_0), \ldots, \underbrace{\min(\,f(a_i), f(a_j)\,)}_{i}, \ldots, \underbrace{\max(\,f(a_i), f(a_j)\,)}_{j}, \ldots, f(a_n)\big) \\
&= \big(f(a_0), \ldots, f(\,\min(a_i, a_j)\,), \ldots, f(\,\max(a_i, a_j)\,), \ldots, f(a_n)\big) \\
&= f\big(a_0, \ldots, \min(a_i, a_j), \ldots, \max(a_i, a_j), \ldots, a_n\big) \\
&= f\big([i:j](a)\big)
\end{aligned}$$

- Theorem (the 0-1 principle):
  Let $\mathcal{N}$ be a comparator network.
  Now, if $\mathcal{N}$ sorts *every* sequence of 0's and 1's, then it also sorts *every* sequence of arbitrary elements!

- Proof (by contradiction):
  - Assumption: $\mathcal{N}$ sorts all 0-1 sequences, but does not sort sequence a
  - Then $\mathcal{N}(a) = b$ is not sorted correctly, i.e. $\exists k : b_k > b_{k+1}$
  - Define $f : A \rightarrow \{0,1\}$ as follows:

$$f(c) = \begin{cases} 0, & c < b_k \\ 1, & c \geq b_k \end{cases}$$

  - Now, the following holds:

$$f(b) = f\big(\mathcal{N}(a)\big) \underset{\substack{\uparrow \\ f \text{ monotone}}}{=} \mathcal{N}\big(f(a)\big) = \mathcal{N}(a')$$

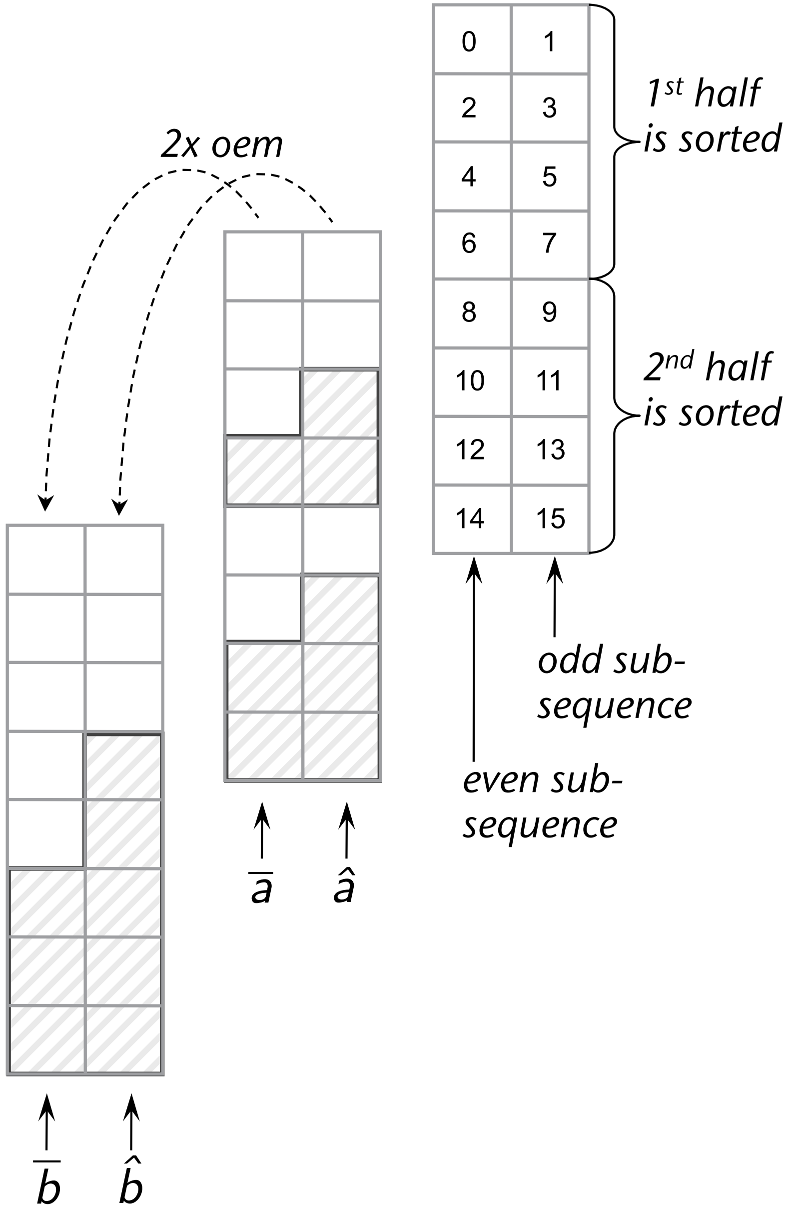
    where a' is a 0-1 sequence.
  - But: $f(b)$ is not sorted, because $f(b_k) = 1$ and $f(b_{k+1}) = 0$
  - Therefore, $\mathcal{N}(a')$ is not sorted as well, in other words, we have constructed a 0-1 sequence that is not sorted correctly by $\mathcal{N}$.

Batcher's Odd-Even-Mergesort [1968]

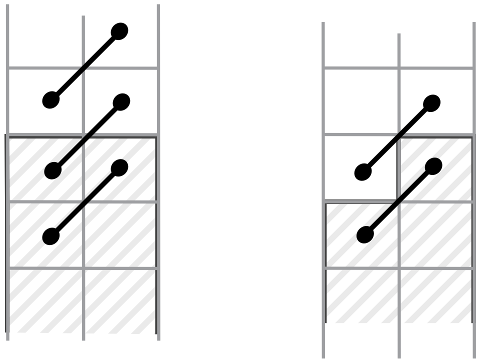
- In the following, we'll always assume that the length n of a sequence $a_0,\dots,a_{n-1}$ is a power of 2, i.e., $n = 2^k$
- First of all, we define the sub-routine "odd-even merge":

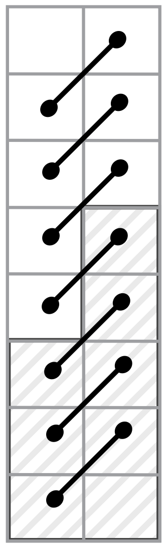
```
oem( $a_0,\dots,a_{n-1}$ ):
precondition:  $a_0,\dots,a_{n/2-1}$ and $a_{n/2},\dots,a_{n-1}$ are both sorted
postcondition: $a_0,\dots,a_{n-1}$ is sorted
if $n = 2$:
    compare $[a_0 : a_1]$                                        (1)
if $n > 2$:
    $\bar{a} \leftarrow a_0,a_2,\dots,a_{n-2}$          // = even sub-sequence
    $\hat{a} \leftarrow a_1,a_3,\dots,a_{n-1}$          // = odd sub-sequence
    $\bar{b} \leftarrow$ oem( $\bar{a}$ )
    $\hat{b} \leftarrow$ oem( $\hat{a}$ )                                      (2)
    copy $\bar{b} \rightarrow a_0,a_2,\dots,a_{n-2}$
    copy $\hat{b} \rightarrow a_1,a_3,\dots,a_{n-1}$
    for i $\in \{1,3,5,\dots,n-3\}$                                (3)
        compare $[a_i : a_{i+1}]$
```

- Proof of correctness:
  - By induction and the 0-1-principle
  - Base case: $n = 2$
  - Induction step: $n = 2^k$, $k > 1$
  - Consider a 0-1-sequence $a_0,\ldots,a_{n-1}$
  - Write it in two columns
  - Visualize 0 = white, 1 = grey
  - Obviously: both $\bar{a}$ and $\hat{a}$ consist of two sorted halves → preconditon of *oem* is met
  - After line (2) we have this situation (the odd sub-sequence can have at most two 1's more than the even sub-sequence)

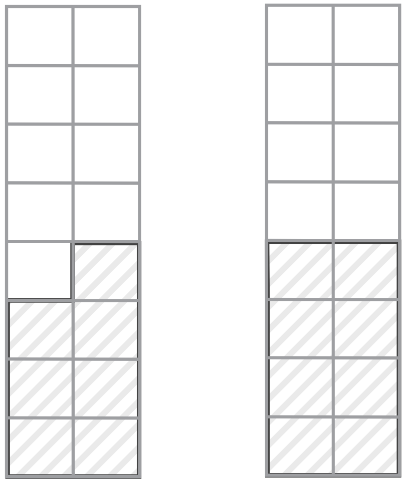


- In loop (3), these comparisons are made, and there can be only 3 cases:





- Afterwards, one of these two situations has been established:



- Result: the output sequence is sorted
- Conclusion:

  every 0-1-sequence (meeting the preconditions) is sorted correctly
- Running time (sequ.) : $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{2} - 1 \in O\left(n \log n\right)$

- The complete general sorting-algorithm:

```
oemSort($a_0,\dots,a_{n-1}$):
if n = 1:
    return
$a_0,\dots,a_{n/2-1}$ ← oemSort($a_0,\dots,a_{n/2-1}$)
$a_{n/2},\dots,a_{n-1}$ ← oemSort($a_{n/2},\dots,a_{n-1}$)
oem($a_0,\dots,a_{n-1}$)
```

- Running time (sequ.): $T(n) \in O\left(n \log^2 n\right)$

Optional

Mapping the Recursion on a Massively-Parallel Architecture

- Load data onto the GPU (global memory)
- The CPU executes the following controlling program:

```
oemSort(n):
if n = 1 → return
oemSort( n/2 )
oem( n, 1 )
```

```
oem( n, stride ):
if n = 2:
    launch oemBaseCaseKernel(stride)
   // launches n parallel threads
else:
    oem( n/2, stride*2 )
    launch oemRecursionKernel(stride)
```

- With the stride parameter, we can achieve sorting "in situ"

Optional

- The kernel for line (3) of the original function *oem*():

```
oemRecursionKernel( stride ):
if tid < stride || tid ≥ n-stride:
    output SortData[tid]
else:
    a_i ← SortData[tid]
    a_j ← SortData[ tid+stride ]
    if tid/stride is even:
        output max( a_i, a_j )
    else:
        output min( a_i, a_j )
```

  - As usual, *tid* = thread ID = 0, …, *n*-1

Optional

- Kernel for line (1) of the function *oem*():

```
oemBaseCaseKernel ( stride ):
i = tid                           // tid = thread ID
if tid/stride is even:           // are we on even/odd side?
    j = i + stride
else:
    j = i - stride
a0 ← SortData[i]                  // SortData = global array
a1 ← SortData[j]
if on even side:
    SortData[i] = min(a0,a1)      // write output back
else:
    SortData[i] = max(a0,a1)
```

- Reminder: this kernel is executed in parallel for each index *tid* = 0, …, *n*-1 in a stream

- Depth complexity:

$$\frac{1}{2}\log^2 n + \frac{1}{2}\log n$$

- E.g., for 2^{20} elements this are 210 passes

Bitonic Sorting

- Definition "bitonic sequence":

  A sequence of numbers $a_0, \ldots, a_{n\text{-}1}$ is bitonic ⇔

  there is an index i such that

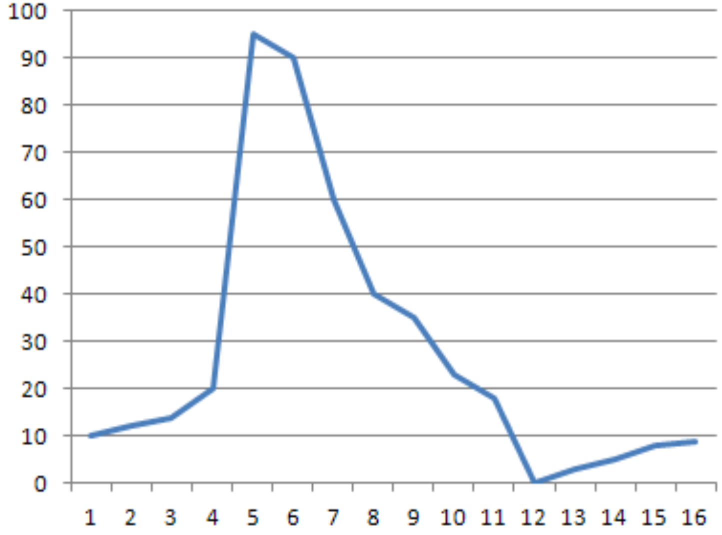
  - $a_0, \ldots, a_i$ is monotonically increasing, *and*
  - $a_{i+1}, \ldots, a_{n\text{-}1}$ is monotonically decreasing;

  OR

  if there is a cyclic shift of this sequence such that this is the case.

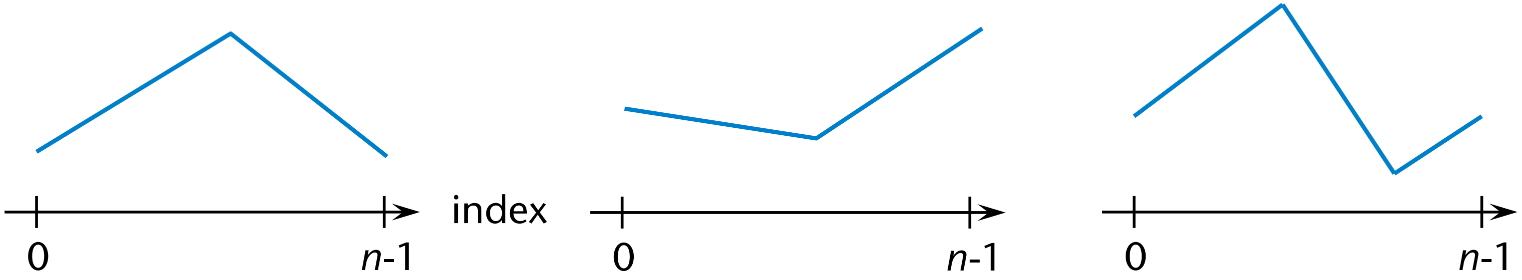
- Because of the latter "OR", we understand *all index arithmetic* in the following modulo n, and/or we assume in the following that the sequence(s) have been cyclically shifted as described above

- Examples of bitonic sequences:
  - 0 2 4 8 10 9 7 5 3 ; 2 4 8 10 9 7 5 3 0 ; 4 8 10 9 7 5 3 0 2 ; …
  - 10 12 14 20 95 90 60 40 35 23 18 0 3 5 8 9
  - 1 2 3 4 5
  - [ ]
  - 000001111110000 ; 11111000000111111 ; 1111100000 ; 000011111

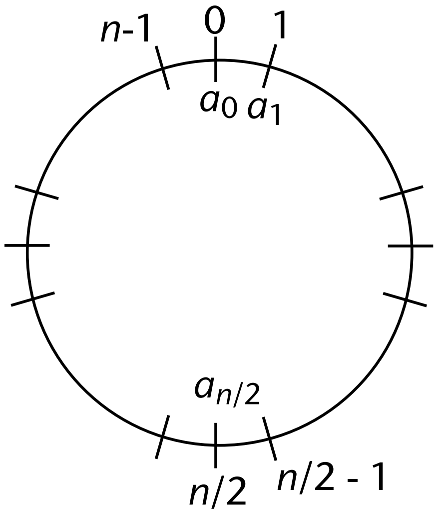


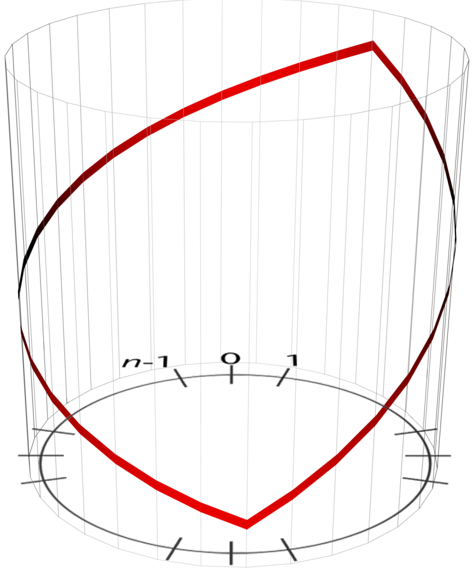
- These sequences are NOT bitonic sequences:
  - 1 2 3 1 2 3
  - 1 2 3 0 1 2

- Visual representation of bitonic sequences:



- Because of the "modulo" index arithmetic, we can also visualize them on a circle or cylinder:
  - Clearly, bitonic sequences have exactly two inflection points



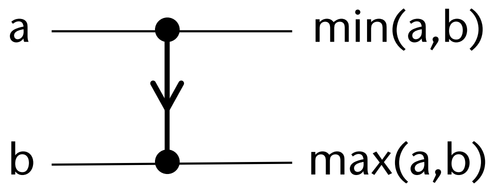


Properties of Bitonic Sequences

- Any sub-sequence of a bitonic sequence is a bitonic sequence
  - More precisely, assume $a_0, \ldots, a_{n-1}$ is bitonic and we have indices $0 \leq i_1 \leq i_2 \leq \ldots \leq i_m < n$
  - Then, $a_{i_0}, a_{i_1}, \ldots, a_{i_m}$ is bitonic, too
- If $a_0, \ldots, a_{n-1}$ is bitonic, then $a_{n-1}, \ldots, a_0$ is bitonic, too
- (If we mirror a bitonic sequence "upside down", then the new sequence is bitonic, too)
- A bitonic sequence has exactly *one* local(!) minimum and *one* local maximum

Some Notions and Definitions

- More precise graphical notation of a comparator:



- Definition rotation operator:

  Let $\mathbf{a} = (a_0, \ldots, a_{n-1})$, and $j \in [1, n\text{-}1]$.

  We define the rotation operator R_j acting on **a** as

$$R_j \mathbf{a} = (a_j, a_{j+1}, \ldots, a_{j+n-1})$$

- Definition L / U operator:

$$L\mathbf{a} = (\,\min(a_0, a_{\frac{n}{2}}), \ldots, \min(a_{\frac{n}{2}-1}, a_{n-1})\,)$$
$$U\mathbf{a} = (\,\max(a_0, a_{\frac{n}{2}}), \ldots, \max(a_{\frac{n}{2}-1}, a_{n-1})\,)$$

- Lemma:

  The L/U operators are *rotation invariant*, i.e.

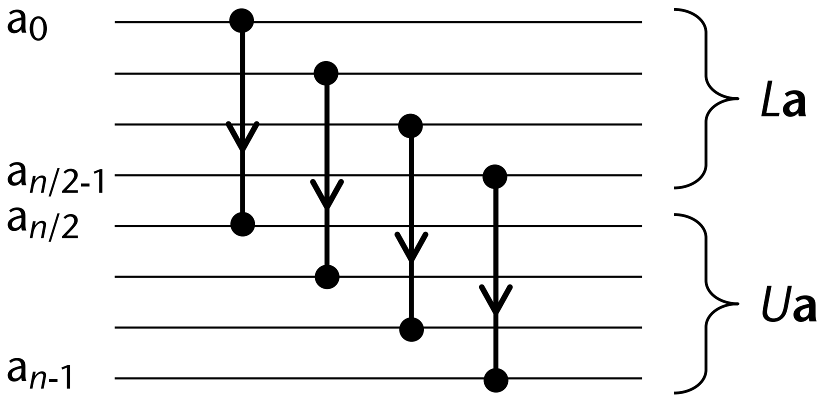
$$L\mathbf{a} = R_{-j}LR_j\mathbf{a}, \quad \text{and} \quad U\mathbf{a} = R_{-j}UR_j\mathbf{a}.$$

  (Remember that indices are always meant mod n)

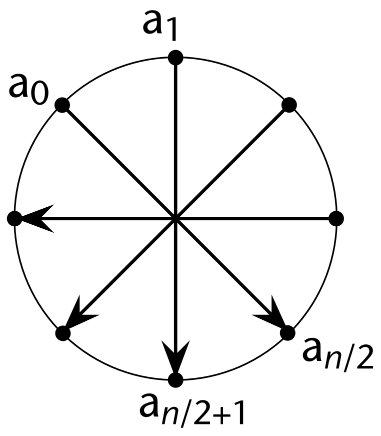
- Proof :
  - We need to show that $R_jL\mathbf{a} = LR_j\mathbf{a}$
  - This is trivially the case:

$$LR_j\mathbf{a} = \left(\min(a_j, a_{j+\frac{n}{2}}), \ldots, \min(a_{\frac{n}{2}-1}, a_{n-1}), \ldots, \min(a_{j-1}, a_{j-1+\frac{n}{2}})\right) = \ldots$$

- Definition half-cleaner:
  A network that takes **a** as input and outputs (*L***a** , *U***a**)
  is called a half-cleaner.
- The network that
  realizes a *half-cleaner*:



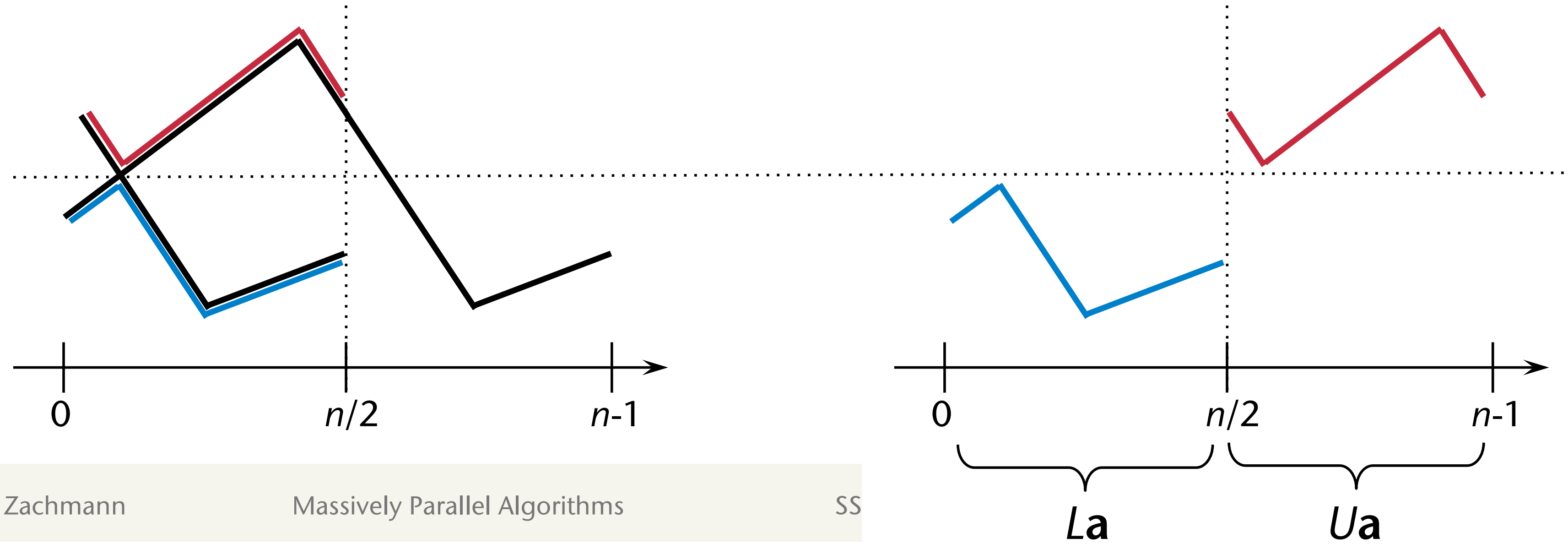
- Because of the rotation invariance, we can depict a half-cleaner on a circle:
  - It always produces *L***a** and *U***a**,
    no matter how **a** is rotated around the circle!



- Theorem 1:
  Given a bitonic input sequence **a**, the output of a half-cleaner has the following properties:
  1. $L\mathbf{a}$ and $U\mathbf{a}$ are bitonic, too;
  2. $\max\{L\mathbf{a}\} \leq \min\{U\mathbf{a}\}$

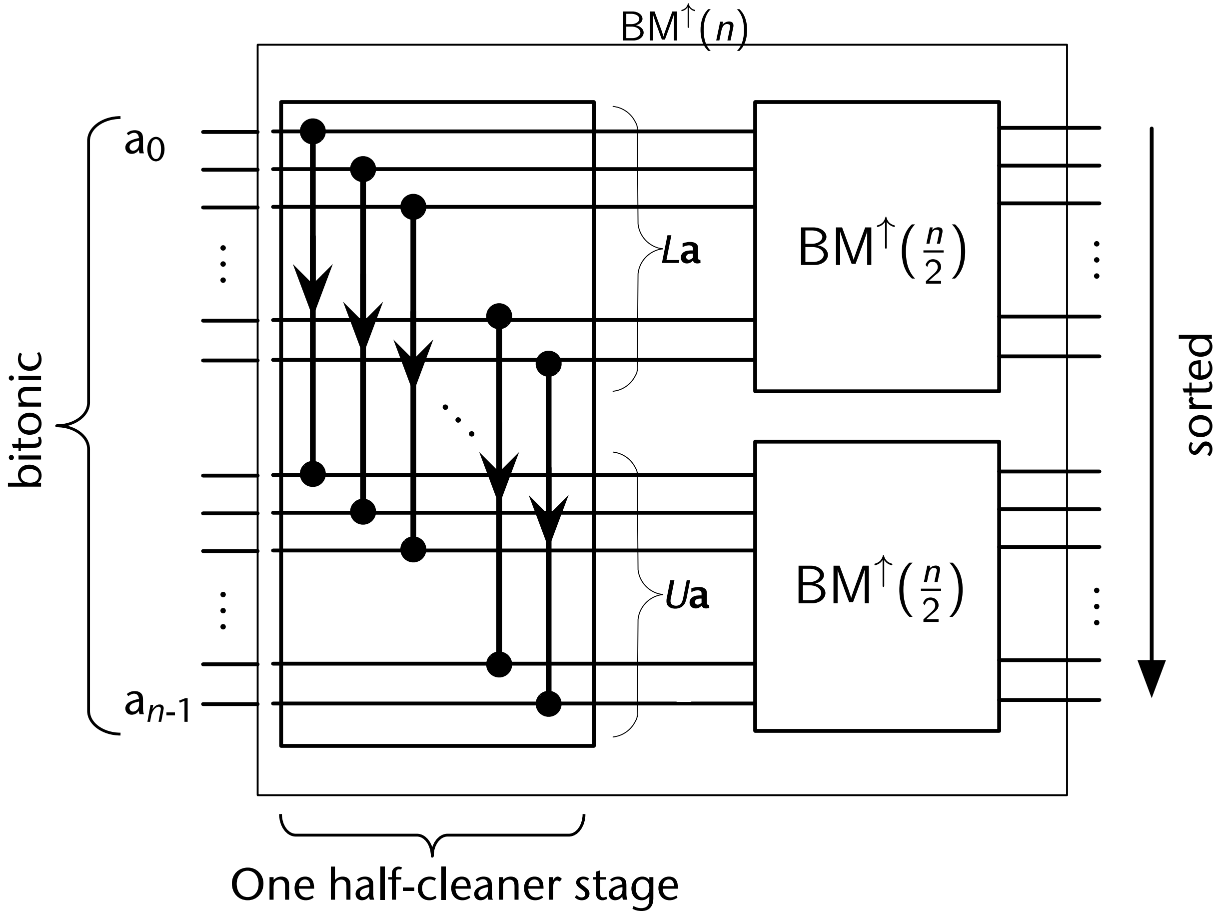
Proof

- The half-cleaner does the following:
  1. Shift (only conceptually) the right half of **a** over to the left
  2. Take the point-wise min/max ⟶ *L***a** , *U***a**
  3. Shift *U***a** back to the right
- Because **a** is bitonic, there can be only one *cross-over point*
- By construction, both *L***a** and *U***a** must have length *n*/2
- Property 1 follows from the sub-sequence property



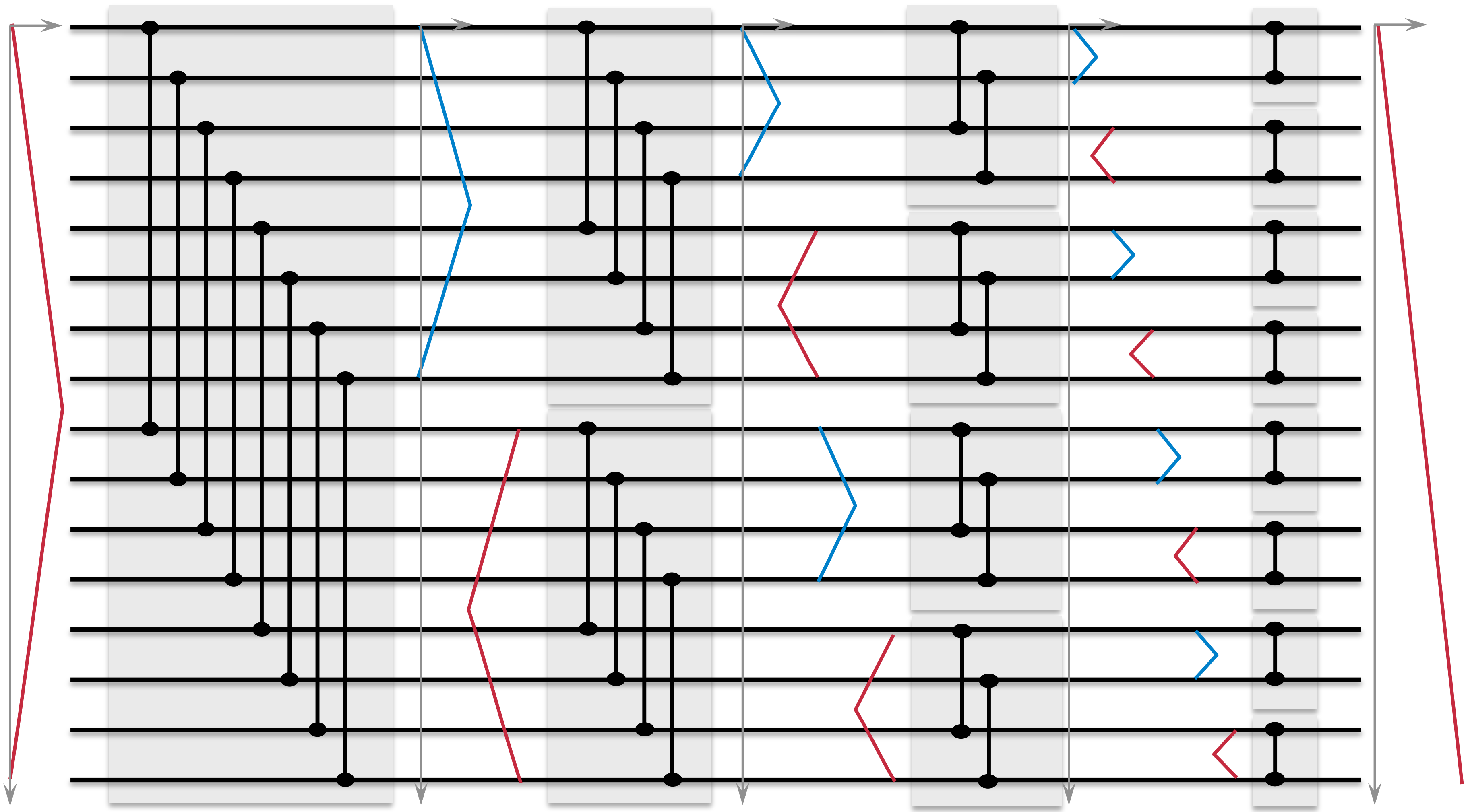
The Bitonic Merger

- The half-cleaner is the basic (and only) building block for the bitonic sorting network!
- The recursive definition of a bitonic merger $\mathrm{BM}^{\uparrow}(n)$:
  - Input: bitonic sequence of length *n*
  - Output: sorted sequence in ascending order



- Analogously, we can define $\mathrm{BM}^{\downarrow}(n)$

Visualization of the Workings of a Bitonic Merger



Mapping to Massively Parallel Architecture

- We have $n = 2^k$ many "lanes" = threads
- At each step, each thread needs to figure out its partner for compare/ exchange
- This can be done by considering the ID of each process (in binary):
  - At step j, $j = 1, \ldots, k$: partner ID = ID obtained by reversing bit (k-j) of own ID
- Example:

```
000  001   010  011   100  101  110  111
|      ^    ^         ^
|______|    |         |
| k-3       |         |
|___________|         |
|    k-2              |
|_____________________|
           k-1
```

The Bitonic Sorter

- The recursive definition of a bitonic sorter $\mathrm{BS}^{\uparrow}(n)$: