Chapter 2

Sorting Networks

A sorting network is a more realistic, but less general, model of a parallel computer than a PRAM. A sorting network is built by organizing and connecting comparator modules, or comparators for short, to sort a sequence of numbers. Each comparator has two input and two output “wires,” with the wires holding numerical values. One of the output wires always outputs the smallest input value and the other outputs the largest input value, as depicted in Figure 2.1. The objective is to find a sorting network architecture that only depends on the length of the input sequence and that is independent of the values of the sequence. Therefore, the main difference between sorting networks and traditional comparison-based sorting algorithms is that the sequence of comparisons is set in advance, regardless of the outcome of previous comparisons.

**FIGURE 2.1:** A comparator.

In this chapter, we present two sorting networks. The first network implements a merge sort, just like Cole’s PRAM algorithm presented in Section 1.4. The second network uses an odd-even transposition scheme that can easily be mapped to a one-dimensional (1-D) network of processors.

2.1 Odd-Even Merge Sort

Batcher [15] proposed a merge sorting network constructed recursively from a network for merging two shorter sequences. We first describe the merging network. Throughout this section we assume that the length of the input sequence is a power of two.

Let us define a few notations:

- For an arbitrary sequence \( \langle c_1, c_2, \ldots, c_n \rangle \), \( \text{Sort}(\langle c_1, c_2, \ldots, c_n \rangle) \) denotes the sorted sequence of the \( c_i \)'s.
- Whenever a sequence \( c \) is sorted, i.e., if \( c_1 \leq c_2 \leq \cdots \leq c_n \), we write \( \text{SORTED}(c_1, c_2, \ldots, c_n) \).
- We define \( \text{MERGE}() \), the merging operator of two sorted sequences, as

\[
\text{MERGE}(\langle a_1, a_2, \ldots, a_n \rangle, \langle b_1, b_2, \ldots, b_n \rangle) = \text{Sort}(\langle a_1, b_1, a_2, b_2, \ldots, a_n, b_n \rangle)
\]

Let us build a \( \text{MERGE}_1 \) network that merges two sorted sequences of length \( 2^m \). For \( m = 0 \), we only need a single comparator. For \( m = 1 \), assuming \( \text{SORTED}(\langle a_1, a_2 \rangle) \) and \( \text{SORTED}(\langle b_1, b_2 \rangle) \), we can use three comparators as depicted in Figure 2.2(a). It is not hard to see that this \( \text{MERGE}_1 \) works as expected. We know that \( a_1 \leq a_2 \) and \( b_1 \leq b_2 \). The upper output is \( \min(a_1, b_1) \) and the lower output is \( \max(a_2, b_2) \). An additional comparator is added to sort the two outputs from the middle, i.e., \( \max(a_1, b_1) \) and \( \min(a_2, b_2) \).

Determining that \( \text{MERGE}_2 \) (depicted in Figure 2.2(b)) works is more difficult. So instead let us prove the result in the general case, by induction. The \( \text{MERGE}_m \) network is built with two copies of the \( \text{MERGE}_{m-1} \) network followed with a column of \( 2^m - 1 \) comparators. The first copy of \( \text{MERGE}_{m-1} \) merges odd elements from the input sequences and the second copy merges even elements. Surprisingly, a simple column of comparators suffices to complete the merging of the two input sequences.

**FIGURE 2.2:** Merge networks for sequences of lengths 2 and 4.
2.1. Odd-Even Merge Sort

PROPOSITION 2.1. Consider two sequences \( A = \langle a_1, \ldots, a_{2n} \rangle \) and \( B = \langle b_1, \ldots, b_{2n} \rangle \) such that \( \text{Sorted}(A) \) and \( \text{Sorted}(B) \). Let us denote

\[
\langle d_1, \ldots, d_{2n} \rangle = \text{Merge}(\langle a_1, a_2, \ldots, a_{2n-1} \rangle, \langle b_1, b_2, \ldots, b_{2n-1} \rangle)
\]

\[
\langle e_1, \ldots, e_{2n} \rangle = \text{Merge}(\langle a_2, a_4, \ldots, a_{2n} \rangle, \langle b_2, b_4, \ldots, b_{2n} \rangle)
\]

Then, we have

\[
\text{Sorted}(\langle d_1, \min(d_2, e_1), \max(d_2, e_1), \ldots, \min(d_{2n}, e_{2n-1}), \max(d_{2n}, e_{2n-1}), e_{2n} \rangle).
\]

Proof. Without loss of generality, we can assume that the elements are distinct. In the resulting sequence \( d_i \) is in first position, which is correct as it is the smallest element of the whole sequence. Likewise, \( e_{2n} \)'s position is correct. In the general case, \( d_i \) and \( e_{i-1} \) (for \( i \geq 2 \) and \( i \leq 2n \)) are in position \( 2i - 2 \) or \( 2i - 1 \) in the resulting sequence. We show that their positions are correct by showing that they both dominate \( 2i - 3 \) elements of the complete sequence and are dominated by \( 4n - 2i + 1 \) elements of the complete sequence. Therefore, their position in the whole sequence is necessarily \( 2i - 2 \) or \( 2i - 1 \) and the final comparison finally sets them in their correct positions.

We have four items to prove for \( 2 \leq i \leq 2n \):

(i) \( d_i \) dominates \( 2i - 3 \) elements.

(ii) \( e_{i-1} \) dominates \( 2i - 3 \) elements.

(iii) \( d_i \) is dominated by \( 4n - 2i + 1 \) elements.

(iv) \( e_{i-1} \) is dominated by \( 4n - 2i + 1 \) elements.

Let us start by proving (i): Assume for example that \( d_i \) is in sequence \( A \) (the case where \( d_i \) is in \( B \) is similar). Let \( k \) be the number of elements from \( \langle d_1, d_2, \ldots, d_i \rangle \) that come from sequence \( A \). Then, we have \( d_i = a_{2k-1} \) and \( d_i \) dominates \( 2k - 2 \) elements from \( A \). There are \( i - k \) elements from \( B \) in \( \langle d_1, d_2, \ldots, d_i \rangle \). The largest one is thus \( b_{2i-1-1} \) and \( d_i \) dominates \( 2(i - k) - 1 \) elements from \( B \). Therefore, \( d_i \) dominates \( 2k - 2 + 2(i - k) - 1 = 2i - 3 \) elements. (ii) is proved in a similar way.

Let us now prove (iv): Assume that \( e_{i-1} \) is in sequence \( B \). Let \( k \) be the number of elements from \( \langle e_1, e_2, \ldots, e_{i-1} \rangle \) that come from sequence \( B \). Then, we have \( e_{i-1} = b_{2k} \) and \( e_{i-1} \) is dominated by \( 2n - 2k \) elements from \( B \). There are also \( i - k \) elements from sequence \( A \) in sequence \( \langle e_1, \ldots, e_{i-1}, e_{i-1} \rangle \). Therefore, \( e_{i-1} \) is dominated by \( 2n - 2(i - k) + 1 \) elements from \( A \). Therefore, \( e_{i-1} \) is dominated by \( 2n - 2(i - k) + 1 + (2n - 2k) = 4n - 2i + 1 \) elements. The proof of (iii) is similar and is left as an exercise for the reader.

The positions of \( d_i \) and \( e_{i-1} \) in the resulting sequence are thus correct. \( \square \)

Figure 2.3 shows the \( \text{Merge}_{m} \) network, with its two copies of the \( \text{Merge}_{m-1} \) network whose outputs are connected with \( 2^m - 1 \) comparators.

The processing time \( t_m \) of the \( \text{Merge}_{m} \) network is defined as the maximum number of comparators an input must traverse. Therefore, we have \( t_2 = 2 \) because some data have to go through two comparators (even though some only go through one) and \( t_2 = 3 \). Of course, many comparators can be active at the same time (this book is about parallel computing after all).

LEMMA 2.1. The processing time \( t_m \) and the number of comparators \( p_m \) of \( \text{Merge}_{m} \) satisfy the following recursions:

\[
t_0 = 1 \quad t_2 = 2 \quad t_m = t_{m-1} + 1 \quad (i.e., \ t_m = m + 1)
\]

\[
p_0 = 1 \quad p_1 = 3 \quad p_m = 2p_{m-1} + 2^m - 1 \quad (i.e., \ p_m = 2^m m + 1).
\]

Proof. These recursions follow directly from Proposition 2.1. The expression \( p_m = 2^m m + 1 \) can be proved via a simple induction.

When expressed as a function of the length \( n = 2^m \) of the input sequence, these quantities are, respectively, \( O(\log n) \) and \( O(n \log n) \). The efficiency of the network is rather low: By multiplying the number of comparators with the processing time, one obtains the total work (see Section 1.2): \( W_m = p_m \times t_m = O(n \log n)^2 \). This is far beyond the total work of a sequential merge: Using a single comparator and \( O(n) \) steps, the total work of the sequential merge is \( O(n) \). Note, however, that the overall processing time, \( t_m \), is very short.

The rather poor efficiency of this network can easily be explained: Each comparator is used only once during the merge. The efficiency of the network
could be improved by merging many different pairs of sequences in sequence. Indeed, the same network can start processing a new pair of sequences at each time unit, with multiple sequences traversing the network concurrently in a pipelined usage. After \( d \) time units (where \( d \) is the depth of the network) all comparators are active and a merged sequence is output by the network at each time unit (the **period** of the network is said to be equal to \( 1 \)).

### 2.1.2 Sorting Network

It is now easy to design a **Sort** network recursively to sort \( n = 2^m \) elements. All we need to do is connect the output of two **Sort** networks with the input of a **Merge** network, as shown in Figure 2.4.

![Figure 2.4](image_url)

**FIGURE 2.4:** Recursive design of Batchter’s **Sort** network (\( m = 3 \)).

#### LEMMA 2.2

The processing time \( t_m \) and the number of comparators \( p_m \) of **Sort** satisfy the following recursions:

\[
\begin{align*}
\ell'_1 &= 1 \\
\ell'_m &= \ell'_{m-1} + t_{m-1} \\
(p'_m) &= 2p'_{m-1} + p_{m-1} \\
\ell_m &= (m-1) + 2 + 3 + \cdots + (m-1) + 1 = 2^{m-1}(m-1) = 2^{m-1}(m-1/2) + 2^{m-1} - 1.
\end{align*}
\]

**Proof.** These recursions follow directly from the recursive design of the network. Since \( \ell'_0 = t_m = t'_{m-1} + t_{m-1} = \ell'_m + m \), we get \( \ell_m = O(m^2) \). For the second recursion, \( p'_m = 2p'_{m-1} + p_{m-1} \). The processing time is \( O(m^2) \), and the number of comparators is \( O(2^{m+2} m^2) \).

When expressed as a function of the length \( n = 2^m \) of the input sequence, the processing time is \( O((\log n)^2) \) and the number of comparators is \( O(n(\log n)^2) \). The total work is thus \( O(n(\log n)^3) \). One may think that the number of comparators is prohibitively large given the processing time, at least for practical purposes. But here again, this very fast network can be used in a pipelined fashion.

One may wonder if there exists a sorting network with a \( O(\log n) \) processing time for sequences of length \( n \). Such a network could then be thought of as an equivalent of Cole’s PRAM algorithm (Section 1.4), but for the less flexible sorting network model. Such a network was designed somewhat recently. In 1983, Ajtai, Komlos, and Szemeredi [2] proposed a network for sorting sequences of length \( n \) with \( O(n \log n) \) comparators in \( O(\log n) \) time. Unfortunately, the constants hidden in the \( O(\log \cdot) \) notations are so enormous that this result is of no practical use.

As a conclusion to this section, we recall the main result:

**THEOREM 2.1.** Batchter’s merge sort network sorts a sequence of length \( n \) with \( O(n(\log^2 n)) \) comparators in \( O(\log^2 n) \) time.

### 2.1.3 0–1 Principle

In this section, we present a new technique to prove the correctness of merging and sorting networks. A 0–1 sequence is a sequence whose elements are either 0 or 1.

**PROPOSITION 2.2 (0–1 Principle).** A network is a sorting network for arbitrary sequences if and only if it is a sorting network for 0–1 sequences.

**Proof.** If \( R \) sorts arbitrary sequences correctly, it obviously also sorts 0–1 sequences correctly. Let us now prove that if \( R \) does not sort arbitrary sequences correctly, then it does not sort 0–1 sequences correctly.

Let us first note that for any increasing function \( f \) a comparator has the same behavior when the input is \( \{x_1, x_2\} \) as when the input is \( \{f(x_1), f(x_2)\} \).

Consider a given network \( R \) applied to a given sequence \( \{x_1, \ldots, x_n\} \). The final position of \( x_i \), i.e., the wire where \( x_i \) is output, does not depend on the value \( x_i \) but solely on its relative position in the sequence. Therefore, when applying \( R \) to \( \{f(x_1), \ldots, f(x_n)\} \), where \( f \) is an increasing function, \( f(x_i) \) is output on the same wire as \( x_i \).

Let us assume that \( R \) does not sort correctly. Then, there is a sequence \( x = \{x_1, \ldots, x_n\} \) and a position \( k \) such that \( R(x_k) > R(x_{k+1}) \). Now let us define an increasing function \( f : \{x_1, \ldots, x_n\} \rightarrow \{0, 1\} \) as follows:

\[
f(y) =
\begin{cases}
0 & \text{if } y < R(x_k) \\
1 & \text{if } y \geq R(x_k)
\end{cases}
\]


2.1. Odd-Even Merge Sort

It is easy to see that $R$ does not sort the 0–1 sequence $(f(x_1), \ldots, f(x_n))$ correctly. Indeed, $f(R(x_k)) = 1$ is output at position $k$ and $f(R(x_{k+1})) = 0$ is output at position $k + 1$. Therefore, $R$ does not sort 0–1 sequences correctly, which completes the proof of the proposition.

The 0–1 principle can be used to prove the correctness of $\text{MERGE}_n$. Let us provide a new, less cumbersome proof of Proposition 2.1.

New proof of Proposition 2.1. Using the 0–1 principle we can now restrict the proof of Proposition 2.1 to 0–1 sequences.

Let us denote by $\text{ZEROS}(x)$ the number of zeros in sequence $x = (x_1, \ldots, x_n)$. A sorted 0–1 sequence is always structured as $x = 0^r1^{m-r}$, where $r = \text{ZEROS}(x)$. Let $p = \text{ZEROS}((a_1, \ldots, a_{2n}))$ and $q = \text{ZEROS}((b_1, \ldots, b_{2n}))$. We distinguish four different cases:

- \( p = 2p' \) and \( q = 2q' \). We have
  \[
  \text{ZEROS}((d_1, \ldots, d_{2n})) = \text{ZEROS}((r_1, \ldots, r_{2n})) = p' + q'.
  \]
  \( (d_1, \ldots, d_{2n}) \) and \( (r_1, \ldots, r_{2n}) \) have the same number of 0's. The last column of comparators receives as input \( p' + q' - 1 \) pairs of 0's, followed by a 10 (the first 1 of \( d \) and the last 0 of \( r \)) and \( 2n - p' - q' - 1 \) pairs of 1's. The sorted sequence is thus obtained thanks to the \( (p' + q') \)-th comparator.

- \( p = 2p' - 1 \) and \( q = 2q' - 1 \). We have
  \[
  \text{ZEROS}((d_1, \ldots, d_{2n})) = \text{ZEROS}((r_1, \ldots, r_{2n})) = p' + q' - 1.
  \]

  The last column of comparators receives as input \( p' + q' - 1 \) pairs of 0's, followed by 20–20–20–20 pairs of 1's. None of the \( 2n - 1 \) comparators is really useful as the resulting sequence \( 0^{p'+1}1^{n-p'} \) is obtained with a simple interleaving.

- \( p = 2p' - 1 \) and \( q = 2q' - 1 \). We have
  \[
  \text{ZEROS}((d_1, \ldots, d_{2n})) = \text{ZEROS}((r_1, \ldots, r_{2n})) = p' + q' - 1.
  \]

  The last column of comparators receives as input \( p' + q' - 1 \) pairs of 0's, followed by a 01 and \( 2n - p' - q' \) pairs of 1's. Once again, none of the \( 2n - 1 \) comparators performs truly useful work.

2.2. Sorting on a One-Dimensional Network

2.2.1. Odd-Even Transposition Sort

In this section, we present a very simple network known as an “odd-even transposition sorting network.” To simplify the depiction of the “folding” of this network in the next section, we rotate the comparators as depicted below:

\[
\begin{array}{c}
\text{min}(a,b) \\
\text{max}(a,b)
\end{array}
\]

The odd-even transposition sorting network consists of a sequence of lines of comparators. More precisely, to sort a sequence of \( n = 2p \) elements, we use \( p \) copies of a sub-network that consists of two lines. See Figure 2.5 for a depiction of this sub-network for \( n = 8 \). The first line contains \( p \) comparators whose inputs are the \( p \) pairs of wires \( 2i - 1 \) and \( 2i \), \( 1 \leq i \leq p \) (“odd” step). The second line contains \( p - 1 \) comparators whose inputs are the \( p - 1 \) pairs of wires \( 2i \) and \( 2i + 1 \), \( 1 \leq i \leq p - 1 \) (“even” step). The whole network thus contains a total of \( p(2p-1) = \frac{n(n-1)}{2} \) comparators. The layout is similar when \( n \) is odd (see Figure 2.5 for \( n = 7 \)) and also contains \( \frac{(n+1)(n-1)}{2} \) comparators.

PROPOSITION 2.3. The odd-even transposition sorting network is a sorting network.

Proof. We use the 0–1 principle. Let \( (a_1, \ldots, a_n) \) denote a 0–1 sequence. Let us denote by \( j_0 \) the number of 1’s in this sequence and let \( j_0 \) denote the position of the last 1 (i.e., the rightmost 1). In Figure 2.6, we show an example for \( n = 7 \), \( k = 3 \), and \( j_0 = 4 \). Let us first note that a 1 never moves to the left. The only possible move is when it is compared with a 0 on its right, in which case it moves to the right.

Let us follow the last 1’s moves. If \( j_0 \) is even, then it does not move in the first step but it moves to the right in the second step. If \( j_0 \) is odd, it moves to the right in the first step. In both cases, it moves to the right from step 2 and for all following steps until it is at position \( n \). Before step 2, the last 1 is at least at position 2 and thus it always has enough time to arrive at position \( n \) in \( n - 1 \) steps.

Let us now follow the moves of the next-to-last 1. At step 0 it is at position \( j_1 \) (\( j_1 = 2 \) in Figure 2.6). As the last 1 moves to the right from step 2 on (at least), the next-to-last 1 is never blocked by the last 1 when moving to the right. Therefore, from step 3 on, the next-to-last 1 moves to the right until it reaches position \( n - 1 \). More generally, the \( i \)-th 1 (counting from the right)
moves to the right starting (at least) at step \( i + 1 \) until it reaches position \( n - i + 1 \). It is never blocked by another 1 on its right.

At last, the leftmost 1 (the \( i \)-th, as there are \( n - 1 \)'s in total) moves to the right at step \( k + 1 \) until it reaches its final position \( n - k + 1 \). The \( k - 1 \) remaining 1's are on its right and the sequence is thus sorted.

Another proof of Proposition 2.2.1. Another proof was proposed by Knuth [74] in Exercises 36 and 37, Section 5.3.4. It relies on “primitive sorting networks.” A primitive sorting network is a sorting network such that comparisons are only made between neighbors (the odd-even transposition network is thus a primitive network).

Formally, a sorting network \( \alpha \) can be modeled by a sequence of comparators, i.e., \( \alpha = \langle x_1, y_1 \rangle \circ \ldots \circ \langle x_k, y_k \rangle = \langle x_1, y_1 \rangle \), where \( k \) is the number of comparators.

2.2. Sorting on a One-Dimensional Network

A primitive network is such that for all \( i \): \( y_i = x_i + 1 \). The proof relies on two lemmas:

**Lemma 2.3.** Let \( \alpha \) be an \( n \)-entry primitive network. \( \alpha \) is a sorting network if and only if \( \alpha \) sorts \( n, n-1, \ldots, 2, 1 \).

**Proof.** Proving the implication is trivial. Let us prove the reciprocal by contradiction.

Let \( x \) denote an input sequence such that \( \alpha(x)_j > \alpha(x)_i \), with \( i < j \). Let \( y = (n, n-1, \ldots, 2, 1) \). We prove that \( \alpha(y)_j > \alpha(y)_i \), which establishes the lemma. We prove this by induction on the number of comparators, \( k \). More precisely, our induction hypothesis is

\[
H_{i,j,x,y}(q) : \text{For any primitive sorting network of size } q, \beta(x)_j > \beta(x)_i \Rightarrow \beta(y)_j > \beta(y)_i.
\]

Let us first prove \( H_{i,j,x,y}(0) \): A primitive network \( \beta \) of size 0 does not sort anything, so we necessarily have \( \beta(y)_j = n + 1 - i > n + 1 - j = \beta(y)_j \), hence the result.

Let us now assume that \( H_{i,j,x,y}(q-1) \) is true and consider an arbitrary primitive network \( \gamma \) of size \( q \) such that \( \gamma(x)_j > \gamma(x)_i \). We have \( \gamma = [p, p+1] \circ \beta \), where \( \beta \) is a primitive sorting network of size \( q-1 \).

We need to distinguish between a few cases depending on \( p \):

- **Case 1:** \( p = i \): We have \( \gamma(x)_j = \gamma(x)_i = \min(\beta(x)_j, \beta(x)_{j+1}) > \gamma(x)_i \) and \( \gamma(x)_j \leq \gamma(x)_{j+1} = \max(\beta(x)_j, \beta(x)_{j+1}) \), implying that \( j 
eq p + 1 \). Therefore, we have \( j > p + 1 \) and \( \gamma(x)_j = \beta(x)_j \). Thus, \( \beta(x)_j > \beta(x)_i \) and \( \beta(x)_{j+1} > \beta(x)_i \). Therefore, \( \beta(x)_j > \beta(x)_i \) and using the induction hypothesis \( H_{i,j,x,y}(q-1) \), we obtain \( \beta(y)_j > \beta(y)_i \) and \( \beta(y)_{j+1} > \beta(y)_i \), which gives us \( \gamma(y)_j > \gamma(y)_i \).

- **Case 2:** \( p = i - 1 \): Similar arguments can be used.

- **Other cases:** The result is obvious.

**Lemma 2.4.** A primitive network for sequences of length \( n \) requires at least \( n(n-1)/2 \) comparators to be a sorting network.

**Proof.** Each comparator reduces the number of inversions of the input \( x \). As the permutation \( (n, n-1, \ldots, 2, 1) \) has exactly \( n(n-1)/2 \) inversions, a primitive sorting network has at least \( n(n-1)/2 \) comparators.

Let us now end the proof of the odd-even transposition network. It is made of exactly \( n(n-1)/2 \) comparators. We can observe the effect of the transposition network on \( (n, n-1, \ldots, 2, 1) \) and check that each comparator performs a real inversion. The resulting sequence is thus sorted, proving the result.
The processing time \( t_n \) of the odd-even transposition network is \( t_n = n \) and the number of comparators is \( p_0 = n(n - 1)/2 \). The total work is then \( W_n = O(n^2) \), which is very high. The only nice feature of this network is its simplicity.

### 2.2.2 Odd-Even Sorting on a One-Dimensional Network

In this last section we focus on sorting using a 1-D network of processors. The algorithm we design is directly inspired by the previous odd-even transposition network. The idea is to “fold” the network to get a 1-D network of processors. Each processor communicates alternately with its left neighbor and its right neighbor.

![FIGURE 2.7: Odd-even merge sort on a 1-D network of processors (\( p = 6 \).)](image)

Since general purpose processors are more powerful than comparators, we only use \( p \leq n \) processors to sort sequences of length \( n \). For simplicity, we assume that \( p \) divides \( n \). Each processor is given a sub-sequence of length \( n/p \). These sub-sequences are sorted in parallel, and thus in time \( O(\frac{n}{p} \log \frac{n}{p}) = O(\frac{n}{p} \log n) \). After this initial sorting, \( p \) steps of odd-even transpositions are performed. However, instead of exchanging a single element, processors exchange sub-sequences of length \( n/p \). When two processors exchange two sub-sequences of length \( n/p \) they are merged and the leftmost processor keeps the first half of the resulting sequence, i.e., the \( n/p \) smaller elements, while the rightmost processor keeps the second half. This algorithm is illustrated in Figure 2.7 on an example.

The computation time needed for a transposition is that of the sequential merge, i.e., \( O(n/p) \). The time for all the transpositions is thus \( O(n) \) and the overall sorting time is \( O(\frac{n}{p} \log n + n) \). The total work is \( O(n(p + \log n)) \) and the algorithm is therefore optimal for \( p \leq \log n \). Note that our analysis does not account for the overhead due to communication between the processors (this is left for upcoming chapters).

The proof of correctness comes directly from Proposition 2.3. In the end, we have proved the following result:

**Proposition 2.4.** On a 1-D network of \( p \) general purpose processors, a sequence of length \( n \) can be sorted in time \( O(\frac{n}{p} \log n + n) \). This algorithm is optimal for \( p \leq \log n \).

### Bibliographical Notes

This chapter draws inspiration from the book by Gibbons and Rytter [61] for the Batcher network and from the book by Akl [3] for the section on the 1-D network. For additional information on sorting networks, we refer the curious reader to the seminal book by Knuth [74].