Chapter 1

Linear Array and Ring Networks

The *linear array* and the *ring* are interconnection networks of processors organised in the shape of a "line". Each processor is physically linked only to each of its neighbours, which in the linear array can either be two or just one for the outmost processors, instead in the ring they are always two, since processors are connected in a circle. Examples, with $P = 6$ processors, of both kinds of networks are depicted below. In the example of linear array (Figure 1) links are *bidirectional*, indicated by the straight lines. Instead, in the example of ring (Figure 2) links are *unidirectional*, in which case they are represented using arrows, to specify the direction in which data can flow. The number of processors and the kind of links adopted will be specified for each network depending on the use case.

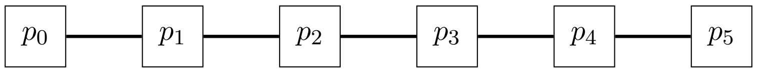

Figure 1: Linear array with 6 processors and bidirectional links.

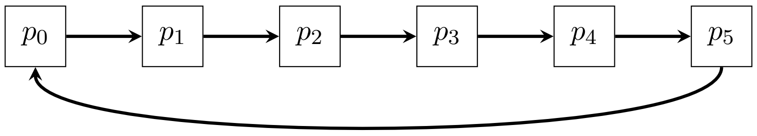

Figure 2: Ring with 6 processors and unidirectional links.

A linear array L_P with P processors and bidirectional links has the following properties.

- Diameter $diam(\mathrm{L}_P) = P - 1$, i.e. the distance between the two opposite outmost processors.
- Bisection bandwidth $b(\mathrm{L}_P) = 1$, independent of the number P of processors.

A ring R_P with P processors and bidirectional links has the following properties.

- Diameter $diam(\mathrm{R}_P) = \lfloor P/2 \rfloor$, i.e. the distance between any two opposite processors.
- Bisection bandwidth $b(\mathrm{R}_P) = 2$, independent of the number P of processors.

1.1 Odd-Even Transposition Sort

In this section we show a sorting algorithm, called *odd-even transposition sort*, for sorting N values, each in input to a different processing unit of a linear array with $P = N$, constant memory $M = 1$ and bidirectional links. Given that the diameter of the linear array is $P - 1$, the task will take at least $N - 1$ steps, since in the worst case it requires to move the value initially in the leftmost processor to the rightmost one, or the opposite. Despite being very simple, odd-even transposition sort is actually very efficient, because it always takes exactly N steps, so just one more than the previous lower bound.

The algorithm works as follows. At each step $t = 0 \dots N{-}1$, every processor i such that i and t share the same parity, compares and possibly exchanges its value with that of its right neighbour $i{+}1$ (if it exists, i.e. if $i{+}1 < N$). Processors decide what to do at each step based on their index in the linear array, which is assumed to be known, locally stored somewhere. After exactly N steps all values are guaranteed to be ordered (following the order of the processors). The procedure for an input of size $N = 4$ is shown in Figure 3 below.

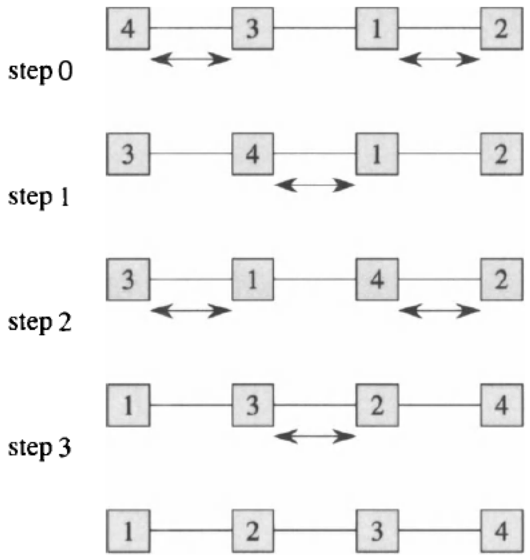

Figure 3: Odd-even transposition sort computation on a linear array with $P = N = 4$.

Odd-even transposition sort belongs to a class of sorting algorithms called *oblivious comparison-exchange algorithms.* The comparison-exchange part means simply that the algorithms operates only by comparing pairs of values and possibly exchanging their positions. On the other hand, oblivious means that the behaviour of the algorithm does not depend on the input values, that is, the algorithm always performs the same sequence of comparison-exchange operations, on predetermined pairs, independently on the values to be sorted. To prove the correctness of this kind of sorting algorithms we can use the following lemma.

Lemma (0–1 sorting). *If an oblivious comparison-exchange algorithm sorts all input sets consisting solely of* 0*s and* 1*s, then it sorts all input sets with arbitrary values.*

Proof. Assume by contradiction that the algorithm fails to sort some sequence of arbitrary values $x_0, x_1, \ldots, x_n$. Let π be a permutation such that $x_{\pi(0)}, x_{\pi(1)}, \ldots, x_{\pi(n)}$ is the output of the algorithm. Since the algorithm fails to sort the sequence, there must exist a pair of indices h and k such that $h < k$ but $x_{\pi(h)} > x_{\pi(k)}$. Now, for every $0 \leq i \leq n$ define

$$y_i = \begin{cases} 0 & \text{if } x_i \leq x_{\pi(k)} \\ 1 & \text{if } x_i > x_{\pi(k)} \end{cases}$$

Observe that, for every i and j, if $x_i \leq x_j$ then also $y_i \leq y_j$, because otherwise we would have $x_i > x_{\pi(k)} \geq x_j \geq x_i$. Therefore, the algorithm, being oblivious and comparison-based, would perform exactly the same exchanges when executed on either sequence, resulting in the same permutation π of before and output $y_{\pi(0)}, y_{\pi(1)}, \ldots, y_{\pi(n)}$. Since we know that $x_{\pi(h)} > x_{\pi(k)}$, then $y_{\pi(h)} = 1$ and clearly $y_{\pi(k)} = 0$. But since $h < k$, this contradicts the hypothesis that the algorithm correctly sorts all inputs with only 0s and 1s. □

Using the previous lemma, in order to prove correctness of an oblivious comparison-exchange algorithm, it is enough to show that it can sort all inputs consisting of solely 0s and 1s. For instance, in the case of odd-even transposition sort, let the input be any sequence $\vec{x}$ of k 1s and $N-k$ 0s, in some order. We need to show that, within N steps, the 1s have all been moved to the k rightmost processors, that is, those indexed by $N-k, N-k+1, \ldots, N-1$. Observe that after any comparison performed by the algorithm on such an input sequence, a 1 can only move to the right, or not move at all. Let h be the rightmost processor to initially have a 1. Then, all processors to its right will necessarily have 0s initially. So, depending on the parity of h, during either the first or the second step of the computation the rightmost 1 will start moving to the right, and it will continue doing so at every subsequent step until it reaches the last processor $N-1$. This also means that surely after the second step, so at least from the third, the second rightmost 1 will also start moving to the right, and will only stop once it reaches processor $N-2$. This reasoning extends to all the 1s, one step later each. So, in general, the i-th rightmost 1 will not move for at most i steps, after which it will start moving to the right and continue doing so until it reaches processor $N-i$. Note that the i-th rightmost 1 initially can at most be located in processor $k-i$, which is distant $N-k$ from its final destination, that is, processor $N-i$. So, all 1s need to move at most $N-k$ times. Furthermore, since each of them will stay still for at most k steps, and will move at every step after that, this means that within $k+N-k = N$ steps all 1s will have reached their destination, and the sequence will, in fact, be sorted.

Performances

- The parallel execution time is $\mathcal{T}_N(N) = \Theta(N)$, since the procedure takes exactly N constant steps, then the work is $\mathcal{W} = \Theta(N^2)$.
- The speedup is $\mathcal{S} = \Theta(N \log N)/\Theta(N) = \Theta(\log N)$ w.r.t. the fastest sequential sorting algorithm, hence the efficiency is $\mathcal{E} = \Theta\left(\frac{\log N}{N}\right)$.

1.2 Rank Enumeration Sort

A different kind of sorting algorithms, counts for each value to be sorted how many precede it in the order. Clearly, these are not comparison-exchange algorithms. A simple algorithm of this kind is *rank enumeration sort*, meant to execute on a ring with $P = N$ processing units with constant memory $M = 2$, working as follows. The values, initially assigned each one to a different processing unit, are circulated around the ring, together with the index of their initial processor. Every time a processing unit i receives a pair (x_j, j), it adds 1 to its own counter r_i only if $x_j < x_i$ or if $x_j = x_i$ and $j < i$. Indeed, the order which is actually used is the so-called lexicographic order on the pairs (value, index). Since every index is unique, there can be no duplicates among the pairs. After every processor i had the opportunity to see every other value, i.e. after $N - 1$ steps, its counter r_i will be the *rank* of the value x_i that it originally received. So, the rank r_i is the number of values that precede x_i once the sequence is sorted, which is also the index of the processor where x_i should be at the end. Then, the pairs (x_i, r_i) stored in each processor i are circulated again as before, hence for another $N - 1$ steps. This time, when processor r_i receives the pair (x_i, r_i), matching its own index, it will simply hold onto and store the value x_i, while continuing passing along all the others. Given what we said before about the lexicographic order of pairs, it should be immediate to see that after $2N - 2$ total steps all values are sorted. Note also that unidirectional links are enough to execute this algorithm.

Performances

- The parallel execution time is $\mathcal{T}_N(N) = \Theta(N)$ since the procedure takes exactly $2N-2$ constant steps, then the work is $\mathcal{W} = \Theta(N^2)$.
- The speedup is $\mathcal{S} = \Theta(N \log N)/\Theta(N) = \Theta(\log N)$ w.r.t. the fastest sequential sorting algorithm, hence the efficiency is $\mathcal{E} = \Theta\left(\dfrac{\log N}{N}\right)$.

1.3 Discrete Convolution

The *discrete convolution* of two N-vectors $\vec{a} = (a_0, \ldots, a_{N-1})$ and $\vec{b} = (b_0, \ldots, b_{N-1})$ is the $(2N-1)$-vector $\vec{c} = (c_0, \ldots, c_{2N-2})$ where c_k, for every $0 \le k \le 2N-2$, is defined by $c_k = \sum_{0 \le i,j < N}^{i+j=k} a_i \, b_j$, i.e. the sum of products of elements with indices that sum to k. Convolutions arise in a variety of applications, including signal processing and polynomial multiplication. In order to compute every c_k on a systolic linear array with $P = 2N$ and $M = 1$, we need to find a flow of the data such that, for every i and j, the values a_i and b_j meet, simultaneously, during the same step in the processing unit computing c_{i+j}. This ensures that each component of each sum is accounted for in the right place, and that the constant memory space is not exceeded in the process. One way to do this is to input data one value at a time from both ends of the linear array, $\vec{a}$ from the right and $\vec{b}$ from the left, and make them flow towards the opposite direction. Furthermore, to guarantee that all pairs correctly align with the corresponding processing units, the values are input in alternating steps, first one value of $\vec{a}$ then one of $\vec{b}$ and so on, and $\vec{b}$ is input in reverse order $b_{N-1}, \ldots, b_0$, while $\vec{a}$ in the regular one. The flow of data for the computation of the discrete convolution of two vectors of size $N = 3$ is shown in Figure 4. The computation of each c_k is done entirely by processing unit k, which updates its $c_k = c_k + a_i b_j$ every time it receives values a_i, b_j from both sides during the same step. Initially, $c_k = 0$ in every processor k.

As it can be easily seen in the figure, each value a_i enters the linear array from the right at step $2i + 1$, and then moves one to the left at every subsequent step. Therefore value a_i

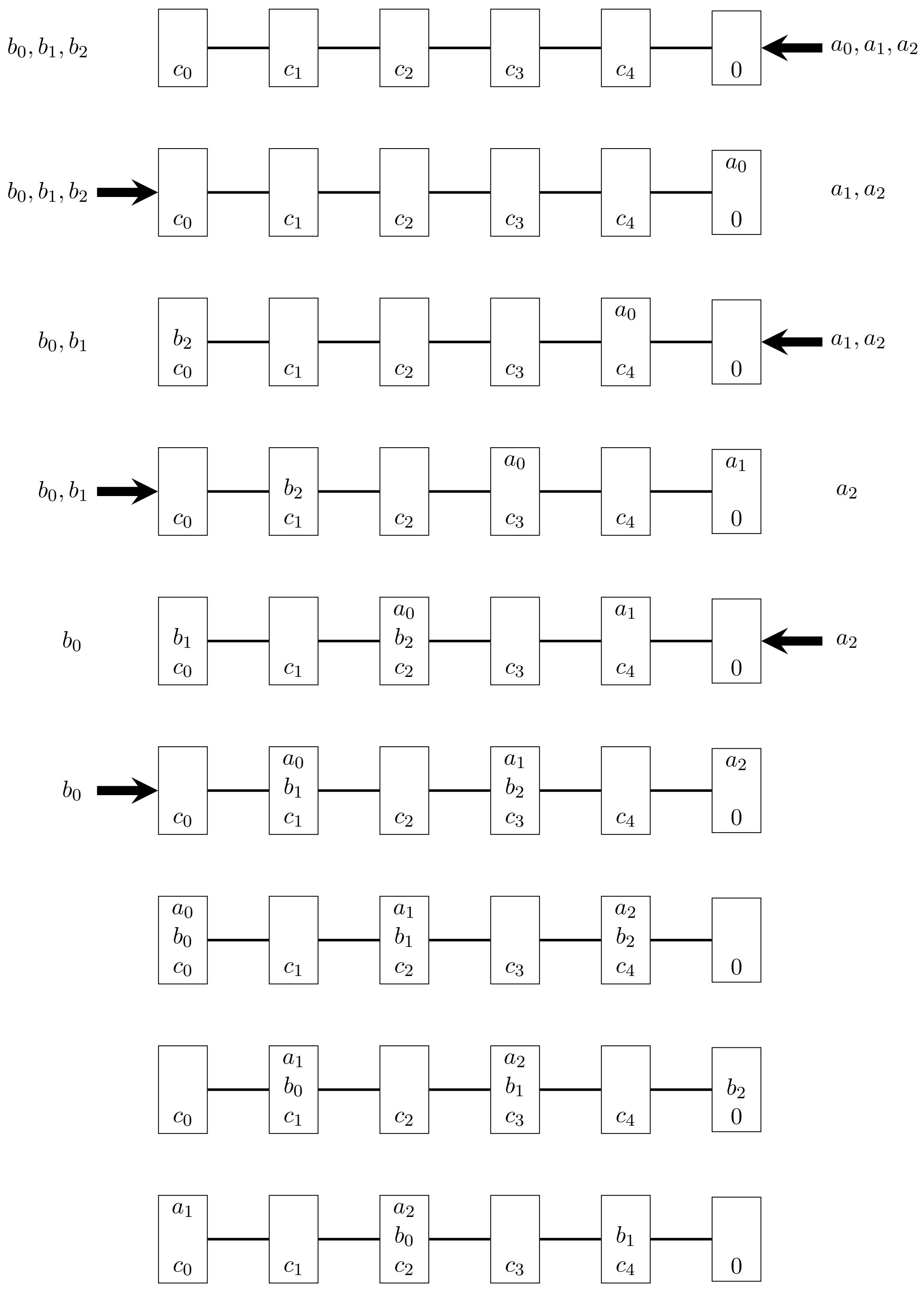

Figure 4: Data flow for the discrete convolution of two vectors of size 3 on a linear array.

will be located in processor k at step $2i+2N-k$. On the other hand, each value b_j enters the linear array from the left at step $2N-2j$, and then moves one to the right at every subsequent step. Therefore value b_j will be located in processor k at step $2N-2j+k$. Then, we deduce that two values a_i and b_j will meet in the same processor k when $2i+2N-k = 2N-2j+k$, that is, when $i+j = k$, as required. Moreover, the last values to enter the linear array, which are a_{N-1} and b_0, will meet at step $2(N-1)+2N-(N-1) = 2N-2\cdot 0+(N-1) = 3N-1$. Since they are the last to enter, this must be the last time any two values will meet. And so the algorithm completes its execution at step $3N-1$, after which every processing unit will contain the corresponding value of $\vec{c}$ (the rightmost processor, which could be ignored since $\vec{c}$ has size $2N-1$, will still contain its initial value 0 unchanged, since no pair of values has ever met there, which can be deemed correct anyway).

Performances

- The parallel execution time is $\mathcal{T}_{2N}(N) = \Theta(N)$ since the procedure takes exactly $3N-1$ constant steps, then the work is $\mathcal{W} = \Theta(N^2)$.
- The speedup is $\mathcal{S} = \Theta(N^2)/\Theta(N) = \Theta(N)$ w.r.t. naive sequential algorithms which would compute the product of each pair a_i, b_j, in such a case the efficiency is $\mathcal{E} = \Theta\left(\frac{N}{2N}\right) = \Theta(1)$, while the latter is not that informative it highlights that it is actually bounded by a constant, independent of N.

1.4 Discrete Fourier Transform

The *Discrete Fourier Transform* (DFT) of a vector $\vec{x}$ of size N is the vector of the same size defined by $\hat{x}_k = \sum\limits_{0\le i<N} x_i\,\omega^{ik}$ for every $0 \le k \le N-1$, where ω is a primitive N-th root of unity. The DFT has many uses in many different fields. Among them it can also be used, along with its inverse operation (IDFT), to compute the discrete convolution described in the previous section. Also in this case, we employ a parallel algorithm on a systolic linear array with $P = N$ and $M = 3$, aiming at computing each $\hat{x}_k$ in a different processing unit k. The three cells of the local memory of each processor k will be used, respectively, to store ω, to accumulate the sum $\hat{x}_k$, and to store a value v_k corresponding to a power of ω depending on the current step of the sum (*not to be confused with the step of the execution*). We can assume that initially each processor has already ω stored somewhere. Otherwise, it would be simple to prepend ω to the input and have each processing unit simply store the first input it receives as the value of ω. Similarly, we can assume that $v_k = 1/\omega = \omega^{-1}$ and $\hat{x}_k = 0$ can be initialised so in every processing unit k. The vector $\vec{x}$ is input to the leftmost processing unit one value at a time in the regular order. As in a standard systolic computation, values will flow towards the right from one processor to the next at each step, after going through some computation. Each time a processor k receives a value $\tilde{x}_i$ it updates its $\hat{x}_k = \hat{x}_k + \tilde{x}_i$ and $v_k = v_k \cdot \omega$, and then it sends the product $\tilde{x}_i \cdot v_k$ to the next processor to the right.

To prove that the algorithm works correctly, we have to show that each $\tilde{x}_i$ received by processor k and added to its $\hat{x}_k$ is the actual $x_i\,\omega^{ik}$ component of the sum corresponding to $\hat{x}_k$. To do this, it is enough to observe two facts.

1. After processing unit k has received the i-th value $\tilde{x}_{i-1}$ and has updated its stored values, its $v_k = \omega^{i-1}$, hence it will relay the product $\tilde{x}_{i-1}\cdot\omega^{i-1}$. In fact, every processor

initially had $v_k = \omega^{-1}$, and then it multiplied v_k by ω every time it received a new value.

2. Each processing unit k will eventually receive every $\tilde{x}_i$, for $0 \leq i \leq N-1$, but this happens only after that value went through k other units, that is, those whose indices are $0, \ldots, k-1$. Moreover, the leftmost processor always receives $\tilde{x}_i = x_i$, since it is the first to receive values.

The first fact implies that each value x_i, initially input as is to the leftmost processor 0, is multiplied by ω^i every time it moves one to the right. Then, by the second fact above, when it reaches processor k the value $\tilde{x}_i = x_i \cdot \omega^{ik}$, which is the correct component to be added to $\hat{x}_k$, as immediately done by the processing unit after receiving it. The computation terminates when all processors have received all the N values, that is, after $2N - 1$ steps (since the last value is input to the leftmost processor after N steps, and then it takes another $N - 1$ steps to reach the rightmost processor), at which point each processing unit k will have computed the corresponding $\hat{x}_k$.

Performances

- The parallel execution time is $\mathcal{T}_N(N) = \Theta(N)$ since the procedure takes $2N - 1$ constant steps, then the work is $\mathcal{W} = \Theta(N^2)$.
- The speedup is $\mathcal{S} = \Theta(N \log N)/\Theta(N) = \Theta(\log N)$ w.r.t. a classic Fast Fourier Transform (FFT) algorithm, hence the efficiency is $\mathcal{E} = \Theta\left(\frac{\log N}{N}\right)$.