1 Model of Computation

It is natural to ask, if one can perform a computational task considerably faster, by using a different computers. Namely, a different computational model.

The answer to this question is a resounding yes. A cute example of this is the Macaroni sort. We are given a set $S = \{s_1, \ldots, s_n\}$ of $n$ real numbers in the range (say) $[1, 2]$. We get a lot of Macaroni (this are longish and very narrow tubes of pasta), and cut the $i$th piece to be of length $s_i$, for $i = 1, \ldots, n$. Next, take all these pieces of pasta in your hand, made stand up vertically, with their bottom end lying on a horizontal surface. Next, lower your handle till it hit the first (i.e., tallest) piece of pasta. Take it out, measure it height, write down its number, and continue in this fashion till you have extracted all the pieces of pasta. Clearly, this is a sorting algorithm that works in linear time. But we know that sorting takes $\Omega(n \log n)$ time. Thus, this algorithm is much faster than the standard sorting algorithm.

This faster algorithm was achieved by changing the computation model. We allowed new “strange” operations (cutting a piece of pasta into a certain length, picking the longest one in constant time, and measuring the length of a pasta piece in constant time), and using them we could sort in linear time.

If this was all we can do with this approach, that would have only been a curiosity. However, interestingly enough, there are natural computation model which are considerably stronger than the standard model of computation. In particular, consider the task of computing the output of the circuit on the right (here, the input is boolean values on the input wires on the left, and the output is the single output on the right).

Clearly, this can be solved by ordering the gates in the “right” order (this can be done by example by topological sorting), and then computing the value of the gates one by one in this order, in such a way that a gate being computed knows the values coming in on its input wires. In particular, for the example circuit above, this would require 8 units of time, since there are 8 gates.

However, if you consider this circuit more carefully, one realized that we can compute this circuit in 4 time units. By using the fact that several gates are independent of each other, and we can compute them in parallel, as depicted on the right. In fact, circuits are inherently parallel and we should be able to take advantage of this fact.

So, let us consider the classical problem of sorting $n$ numbers. The question is weather we can sort them in sublinear by allowing parallel comparisons. To this end, we need to precisely define our computation model.

---

*This work is licensed under the Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit [http://creativecommons.org/licenses/by-nc/2.5/](http://creativecommons.org/licenses/by-nc/2.5/) or, (b) send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.*
2 Sorting with a circuit – a naive solution

We are going to design a circuit, where the inputs are the numbers and we compare two numbers using a comparator gate. Such a gate has two inputs and two outputs, and it is depicted on the right.

For our drawings, we will draw such a gate as follows as a vertical segment connecting two lines. Thus, our circuits would be depicted by horizontal lines, with vertical segments (i.e., gates) connecting between them. For example, see complete sorting network depicted on the right.

The inputs come on the wires on the left, and are output on the wires on the right. The largest number is output on the bottom line. Somewhat surprisingly, one can generate circuits from a known sorting algorithm.

In fact, consider the sorting circuit on the left. Clearly, this is just the inner loop of the standard insertion sort. In fact, if we repeat this loop, we get the sorting network shown on the right.

An alternative way of drawing this sorting network is depicted in Figure 1(ii). The next natural question, is how much time does it take for this circuit to sort the \( n \) numbers. Observe, that the running time of the algorithm is how many different time ticks we have to wait till the result stabilizes in all the gates. In our example, the alternative drawing immediately tell us how to schedule the computation of the gates. See Figure 1(ii).

In particular, the above discussion implies the following result.

**Lemma 2.1** Using sorting network based on insertion sort, resulting a sorting network that has \( O(n^2) \) gates, and requires \( 2n - 1 \) time units to sort \( n \) numbers.

3 Definitions

**Definition 3.1** A **comparison network** is a DAG (directed acyclic graph), with \( n \) inputs and \( n \) outputs, which each gate has two inputs and two outputs.

**Definition 3.2** The **depth** of a wire is 0 at the input. For a gate with two inputs of depth \( d_1 \) and \( d_2 \) the depth on the output wire is \( 1 + \max(d_1, d_2) \).

The **depth** of a comparison network is the maximum depth of an output wire.

**Definition 3.3** A **sorting network** is a comparison network such that for any input, the output is monotonically sorted. The **size** of a sorting network is the number of gates in the sorting network. The **running time** of a sorting network is just its depth.
4 The Zero-One Principle

The zero-one principle states that if a comparison network sort correctly all binary inputs (i.e., every number is either 0 or 1) then it sorts correctly all inputs. We of course need prove that the zero-one principle is true.

Lemma 4.1 If a comparison network transforms the input sequence \( a = (a_1, a_2, \ldots, a_n) \) into the output sequence \( b = (b_1, b_2, \ldots, b_n) \), then for any monotonically increasing function \( f \), the network transforms the input sequence \( f(a) = (f(a_1), \ldots, f(a_n)) \) into the sequence \( f(b) = (f(b_1), \ldots, f(b_n)) \).

Proof: Consider a single comparator with inputs \( x, y \), and outputs \( x' = \min(x, y) \) and \( y' = \max(x, y) \). If \( f(x) = f(y) \) then the claim trivially holds for this comparator. If \( f(x) < f(y) \) then clearly

\[
\max(f(x), f(y)) = f(\max(x, y)) \quad \text{and} \quad \min(f(x), f(y)) = f(\min(x, y))
\]

Thus, for input \( (x, y) \), for \( x < y \), we have output \( (x, y) \) and output \( (f(x), f(y)) \) for input \( (f(x), f(y)) \) the output is \( (f(x), f(y)) \) and for input \( (x, y) \), for \( x > y \), we have output \( (y, x) \) and output \( (f(y), f(x)) \) for input \( (f(x), f(y)) \) the output is \( (f(y), f(x)) \).

Establishing the claim for one comparator.

This implies that if a wire carry a value \( a_i \) when the network get input \( a_1, \ldots, a_n \) then for the input \( f(a_1), \ldots, f(a_n) \) this wire would carry the value \( f(a_i) \). This follows immediately by using the above claim for a single comparator together with induction on the network structure.

This immediately implies the lemma.

■

Theorem 4.2 If a comparison network with \( n \) inputs sorts all \( 2^n \) binary strings of length \( n \) correctly, then it sorts all sequences correctly.

Proof: Assume for the sake of contradiction, that it sorts incorrectly the sequence \( a_1, \ldots, a_n \). Let \( b_1, \ldots, b_n \) be the output sequence for this input.

Let \( a_i < b_k \) be the two numbers that outputted in incorrect order (i.e. \( a_k \) appears before \( a_i \) in the output).

Let \( f(x) = \begin{cases} 0 & x \leq a_i \\ 1 & x > a_i \end{cases} \). Clearly, by the above lemma, for the input \( f(a_1), \ldots, f(a_n) \) - binary circuit

the circuit would output \( f(b_1), \ldots, f(b_n) \). But then, this sequence looks like \( 000..0??/?f(a_k)??/f(a_i)??1111 \)

but \( f(a_i) = 0 \) and \( f(a_j) = 1 \). Namely, the output is

??/?1???0??/.

Namely, we have a binary input \( f(b_1), \ldots, f(b_n) \) for which the comparison network does not sort it correctly. A contradiction to our assumption.

■

5 A bitonic sorting network

Definition 5.1 A bitonic sequence is a sequence which is first increasing and then decreasing, or can be circularly shifted to become so.

Example 5.2 The following two sequences are bitonic: \( (1, 2, 3, \pi, 4, 5, 4, 3, 2, 1) \) and \( (4, 5, 4, 3, 2, 1, 1, 2, 3) \), while the sequence \( (1, 2, 1, 2) \) is not bitonic.
Figure 2: Depicted are the (i) recursive construction of BitonicSorter[n], (ii) opening up the recursive construction, and (iii) the resulting comparison network.

Observation 5.3 A bitonic sequence over 0, 1 is either of the form $0^i 1^j 0^k$ or of the form $1^i 0^j 1^k$ where $0^i$ denote a sequence of i zeros.

Definition 5.4 A bitonic sorter is a comparison network that sort bitonic sequences.

Definition 5.5 A half-cleaner is a comparison network, connecting line $i$ with line $i + n/2$. In particular, let Half-Cleaner[n] denote the half-cleaner with n inputs. Note, that the depth of a Half-Cleaner[n] is one.

It is beneficial to consider what a half-cleaner do to an input which is a (binary) bitonic sequence. Clearly, in this case we have that the left half size is clean and all equal to 0. And the right size of the output is bitonic.

In fact, it is easy to prove by simple (but tedious) case analysis that the following lemma holds.

Lemma 5.6 If the input to a half-cleaner is a binary bitonic sequence then for the output sequence we have that (i) the elements in the top half are smaller than the elements in bottom half, and (ii) one of the halves is clean, and the other is bitonic.

This suggests a simple recursive construction of BitonicSorter[n], see Figure 2.

Thus, we have the following lemma.

Lemma 5.7 BitonicSorter[n] sorts bitonic sequences of length $n = 2^k$, it uses $(n/2)k = \frac{n}{2} \lg n$ gates, and it is of depth $k = \lg n$.

5.1 Merging sequence

Next, we deal with the following merging question. Given two sorted sequences of length $n/2$, how do we merge them into a single sorted sequence?

The idea here is concatenate the two sequences, where the second sequence is being flipped (i.e., reversed). It is easy to verify that the resulting sequence is bitonic, and as such we can sort it using the BitonicSorter[n].

Specifically, given two sorted sequences $a_1 \leq a_2 \leq \ldots \leq a_n$ and $b_1 \leq b_2 \leq \ldots \leq b_n$, observe that the sequence $a_1, a_2, \ldots, a_n, b_n, b_{n-1}, b_{n-2}, \ldots, b_2, b_1$ is bitonic.
Figure 3: (i) Merger via flipping the lines of bitonic sorter. (ii) A BitonicSorter. (ii) The Merger after we “physically” flip the lines, and (iv) An equivalent drawing of the resulting Merger.

Thus, to merge two sorted sequences of length $n/2$, just flip one of them, and use BitonicSorter[$n$], see Figure 3. This is of course illegal, and as such we take BitonicSorter[$n$] and physically flip the last $n/2$ entries. The process is depicted in Figure 3. The resulting circuit Merger takes two sorted sequences of length $n/2$, and return a sorted sequence of length $n$.

It is somewhat more convenient to describe the Merger using a FlipCleaner component. See Figure 4.

Lemma 5.8 The circuit Merger[$n$] gets as input two sorted sequences of length $n/2 = 2^{k-1}$, it uses $(n/2)k = n/2 \lg n$ gates, and it is of depth $k = \lg n$, and it outputs a sorted sequence.

6 Sorting Network

We are now in the stage, where we can build a sorting network. To this end, we just implement merge sort using the Merger[$n$] component. The resulting component Sorter[$n$] is despited on the right using a recursive construction.

Lemma 6.1 The circuit Sorter[$n$] is a sorting network (i.e., it sorts any $n$ numbers) using $G(n) = O(n \log^2 n)$ gates. It has depth $O(\log^2 n)$. Namely, Sorter[$n$] sorts $n$ numbers in $O(\log^2 n)$ time.

Proof: The number of gates is

$$G(n) = 2G(n/2) + \text{Gates}(\text{Merger}[n]).$$
Which is

\[ G(n) = 2G(n/2) + O(n \log n) = O(n \log^2 n) \]

As for the depth, we have that

\[ D(n) = D(n/2) + \text{Depth}(\text{Merger}[n]) = D(n/2) + O(\log(n)) \]

and thus

\[ D(n) = O(\log^2 n) \]

as claimed.

7 Faster sorting networks

One can build a sorting network of logarithmic depth (see [AKS83]). The construction however is very complicated. A simpler parallel algorithm would be discussed sometime in the next lectures. BTW, the AKS construction [AKS83] mentioned above, is better than bitonic sort for \( n \) larger than \( 2^{8046} \).

References