Chapter 3

Binary Tree Network

Binary trees are one of the most common structures in computer science. It is not surprising that they are also used as network architectures. Processing units in the *binary tree* interconnection network have up to three connections each. It is useful to distinguish the units occurring on the leaves of the network from those on its inner nodes, which can be further separated from the root. These distinctions can be locally ascertained by checking the links available to the processor. Leaf processors only have one link above. The root processor has just two links, one to the left and one to the right. Other inner nodes have all three such links. These three kinds of node positioning are usually sufficient for describing algorithms that run on a binary tree, so indices will mostly be avoided. We will always consider *complete* binary tree networks, where all levels of the tree are filled, so that all the leaves are distant the same from the root. Such a distance is the *height* h of the tree. A binary tree with P processing units has height $h = \log(P+1) - 1$, or, vice versa, has $P = 2^{h+1} - 1$ processing units, of which 2^h are leaves. An example of binary tree is depicted in Figure 10.

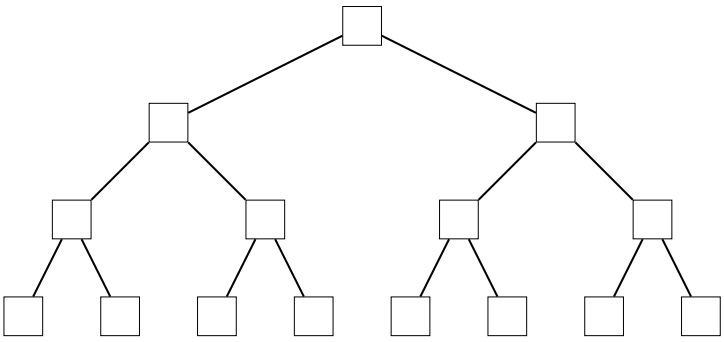

Figure 10: Binary tree of height 3 with $2^{3+1} - 1 = 15$ processing units.

A binary tree B_P of height $h = \log(P+1) - 1$ with P processing units and bidirectional links has the following properties.

- Diameter $diam(\mathrm{B}_P) = 2h$, i.e. the distance between any two leaves on opposite sides w.r.t. the root.
- Bisection bandwidth $b(\mathrm{B}_P) = 1$, independent of the number P of processors.

3.1 Associative Operations

The binary tree is especially suited for the computation of associative operations on some elements. Formally, this applies to any *semigroup* $(S, \otimes)$, where S is a set and $\otimes : (S \times S) \to S$ is an *associative* binary operation, i.e. such that $(x \otimes y) \otimes z = x \otimes (y \otimes z)$ for all $x, y, z \in S$. Examples of such operations are addition, multiplication, maximum and minimum, logical operators $\wedge$ and $\vee$, and many others. The "conquer" phases of some divide-and-conquer procedures can also be expressed by associative operations.

A binary tree network of height $h = \log N$ can apply an associative operation to N elements in just $h = \log N$ steps, which are constant as long as the operation itself is constant. The algorithm works for any associative operation $\otimes$ as follows. Initially each leaf processor receives one of the N inputs. Since the binary tree has height $\log N$, it will have $2^h = N$ leaves and $P = 2N - 1$ processing units in total. At each step of the computation the values simply move upwards, from child node to parent. Every time an inner node receives two values x and y, one from each child, it computes $x \otimes y$ and then sends the result to its parent. The last node to receive some values is the root of the binary tree, which, after applying the operation on such values, it will store/output the result, terminating the computation. *Attention: the fact that the operation is associative does not imply that it is also commutative, and $\otimes$ is not required to be so!* However, in this case, the algorithm computes the sequence of operations $x_0 \otimes x_1 \otimes \ldots \otimes x_{N-1}$ by applying the binary operation to pairs of sub-results obtained from adjacent sub-sequences. E.g. the first time the operation is applied to each pair of adjacent elements. Therefore, the entire order in which the operation is applied is preserved. Each processing unit only needs constant memory to execute the algorithm (as long as every element of the semigroup S requires constant/bounded memory space). In Figure 11 is shown the computation of the addition of 4 values.

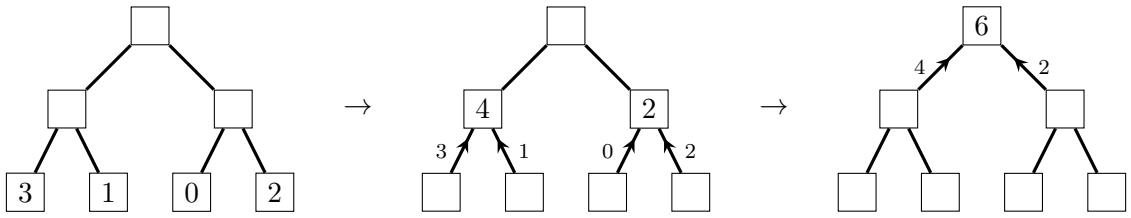

Figure 11: Addition of 4 values on a binary tree of height 2.

Performances

- The parallel execution time is $\mathcal{T}_{2N-1}(N) = \Theta(\log N)$ since the procedure takes $h = \log N$ constant steps, then the work is $\mathcal{W} = \Theta(N \log N)$.
- The speedup is $\mathcal{S} = \Theta(N)/\Theta(\log N) = \Theta(N/\log N)$ w.r.t. the sequential iteration of the $N - 1$ binary operations, then the efficiency is $\mathcal{E} = \Theta\left(\frac{N/\log N}{2N-1}\right) = \Theta(1/\log N)$.

3.2 Prefix Computation

Another common problem related to associative operations is the computation of the prefix operation of a sequence of elements. Given an associative operation $\otimes$ and a sequence $x_0, x_1, \ldots, x_{N-1}$ of $N-1$ elements, we want to apply $\otimes$ to every prefix of the sequence to obtain another sequence of the same length

$$\begin{aligned} y_0 &= x_0 \\ y_1 &= x_0 \otimes x_1 \\ y_2 &= x_0 \otimes x_1 \otimes x_2 \\ &\vdots \\ y_{N-1} &= x_0 \otimes x_1 \otimes \ldots \otimes x_{N-1} \end{aligned}$$

We will again use a binary tree with $2N-1$ processing units with constant memory, of which N are leaves where the input will initially be placed, one element in each leaf from left to right. The computation of the N prefixes is performed in two phases, at the end of which each leaf of the binary tree will contain the corresponding prefix, i.e., the result of the operation $\otimes$ applied to the sequence of elements initially contained in that leaf and those to its left. The first phase is ascending, similar to the procedure described in the previous section, where all data is sent upwards and inner nodes forward the results $x_L \otimes x_R$ of the binary operation applied to the values x_L and x_R received from their children. However, in this case, inner nodes also store in their memory the value x_L that they received from their left child, only the left one. Eventually the root of the binary tree will receive from its left child the result of the operation $\otimes$ applied to the values initially contained in the leaves of the left subtree. The same will happen for the right subtree from the right child, although this value is actually useless to the root. At this point, the second phase begins, which, instead, is a descending one. First, every inner node will send its stored value only to its right child. Then, such values will be forwarded downwards to both children by every following node. Meanwhile, whenever a leaf receives a value x from above, it updates $x_i = x \otimes x_i$, where x_i initially was its given input. The evolution of the state of the network through the whole computation with 4 input values is graphically represented in Figure 12, where $\otimes$ is the classical addition.

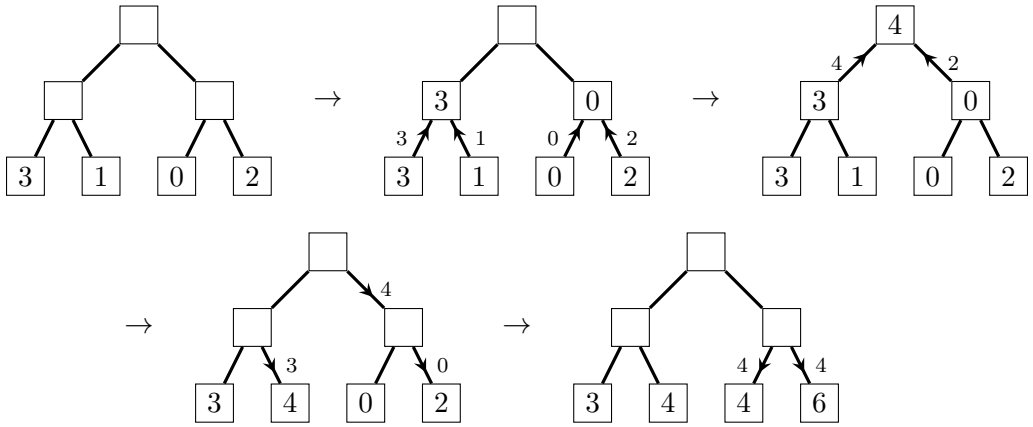

Figure 12: Prefix sum of 4 values on a binary tree of height 2.

The two phases, described above as happening separately one after the other, actually can and should overlap as much as possible. This also makes sense w.r.t. the instructions assigned to the processing units of the network. Indeed, instead of globally separating the two phases, the algorithm/program would simply prescribe to act as follows. First, every leaf sends its input value x_i upwards. Then, every processing unit just acts in response to receiving a value from some of its links.

- The first time (and only one in fact) a processor receives values x_L and x_R from its left and right child, respectively, it forwards x_L to its right child, computes $x_L \otimes x_R$ and forwards the result upwards to its parent (the root will ignore the latter operation).
- Every time a processor receives a value x from above, if it is a leaf it updates its $x_i = x \otimes x_i$, otherwise it forwards such a value x to both its left and right children.

Since, at most, values will need to flow from the leaves up to the root and back, after $2h = 2\log N$ steps every leaf will have received all necessary values to compute the corresponding prefix.

The correctness of the results can be proved by induction on the height h of the binary tree. In the base case, when $h = 1$, the root, which is the only inner node, will just receive x_0 from the left leaf (ignoring the value coming from the right) and forward x_0 to the right leaf, which then will compute $x_0 \otimes x_1$, as desired. Now, to prove the correctness for a tree of height $h = k + 1$, assume that the algorithm produces the correct results on the left and right subtrees of height k, which would be

$$\begin{aligned} y_0 &= x_0 \\ y_1 &= x_0 \otimes x_1 \\ &\vdots \\ y_{N/2-1} &= x_0 \otimes x_1 \otimes \ldots \otimes x_{N/2-1} \end{aligned} \qquad \text{and} \qquad \begin{aligned} z_0 &= x_{N/2} \\ z_1 &= x_{N/2} \otimes x_{N/2+1} \\ &\vdots \\ z_{N/2-1} &= x_{N/2} \otimes x_{N/2+1} \otimes \ldots \otimes x_{N-1} \end{aligned}$$

respectively, since $N = 2^{k+1}$. Furthermore, it is easy to see that the root will receive value $y_{N/2-1}$ from its left child, that is the root of the left subtree, since every inner node forwards upwards the result of the operation applied to the sequence of inputs to leaves in the subtree rooted in such a node. Then, the root of the binary tree will forward the received value $y_{N/2-1}$ only to its right child. Such a value will be forwarded downwards to all descendants, and so all the leaves, in the right subtree. We conclude by observing that, after receiving such a value, every leaf in the right subtree will compute the corresponding $y_{N/2-1} \otimes z_i = x_0 \otimes \ldots \otimes x_{N/2-1} \otimes x_{N/2} \otimes \ldots \otimes x_{N/2+i} = y_{N/2+i}$, for $0 \leq i \leq N/2 - 1$. And so, each leaf in general will finally contain the corresponding prefix $y_j = x_0 \otimes x_1 \otimes \ldots \otimes x_j$, for all $0 \leq j \leq N-1$ starting from the leftmost leaf.

Performances

- The parallel execution time is $\mathcal{T}_{2N-1}(N) = \Theta(\log N)$ since the procedure takes $2h = 2\log N$ constant steps, then the work is $\mathcal{W} = \Theta(N \log N)$.
- The speedup is $\mathcal{S} = \Theta(N)/\Theta(\log N) = \Theta(N/\log N)$ w.r.t. the sequential iteration of the $N - 1$ binary operations (recording every partial result corresponding to every prefix), then the efficiency is $\mathcal{E} = \Theta(1/\log N)$.

3.3 Selection

The possibility to compute in logarithmic time cumulative operations on sets of values can be exploited to efficiently solve many interesting, and much more complex, problems. For example, the *selection* of the k-th smallest element in a (multi-)set, also known as the k-th *order statistic* of the set. This problem generalises those of finding the maximum, the minimum, and, especially, the *median* of a set. A general idea to solve this problem is to find some pivot element such that we are able to partition the set in two parts, i.e. the elements below and those above the pivot, of size as similar as possible. Then, depending on the index k and the sizes of the two parts, the procedure is repeated on only one of those parts, until so few elements remain that the desired one can be selected among them in constant time.

The algorithm presented here solves any selection problem of size N (the size of the set) such that all values are represented in binary by B bits, for some fixed B, which determines the total number of iterations, similar to those described above. In the following we enumerate the B bits of input values so that the 1st bit is the most significant and the B-th is the least significant. The procedure uses a binary tree with N leaves, one for each input value as usual, so with a total of $2N - 1$ processing units with constant memory. At the end of each iteration some leaf processors will turn off, intuitively those that contain inputs that have been excluded from the next iteration. Initially, each leaf is on and contains one of the N inputs, while the root of the binary tree is given the input size N itself and the index k of the order statistic to be selected. Every iteration is divided in two phases, one ascending and one descending, both taking a logarithmic (of N) number of steps. During the first phase of the first iteration, all leaves send upwards the most significant bit of their input. The inner nodes receive those bits and compute their sum, which they forward upwards to their parents. At the end of the ascending phase the root will receive the sum of all the most significant bits coming from the leaves of the left and the right subtree, respectively. The sum of those two numbers will correspond to the exact number S of inputs whose most significant bit equals 1. This means that S inputs must be above or equal to 2^{B-1} (the pivot), while the other $N - S$ inputs must be strictly below the latter. So, if $k \leq N - S$ we can deduce that the requested element must have the most significant bit equal 0, otherwise its most significant bit must be 1. This check is performed by the root of the binary tree, the only processor who knows S, N and k, then, depending on the outcome, will communicate to all the leaves whether such a bit should be 0 or 1, by sending it downwards to its children, which will forward it. This is the descending phase mentioned above. Then, when a leaf receives the bit decided by the root, it checks whether the most significant bit of its input is the same, and, if not, it turns off for the rest of the computation. Thus, at the end of the iteration, all the inputs in the leaves that are still on will agree on their most significant bit. During the descending phase the root processor also updates its stored values k and N depending on the selected partition of the input set. If bit 0 was selected, because $k \leq N - S$, then k remains the same and $N = N - S$, otherwise $k = k - (N - S)$ and $N = S$. The first iteration for the selection of the 2nd smallest value among 4, expressed as 2 bits binary numbers, is depicted in Figure 13. The grey leaves are those that have been turned off as a result of the first iteration.

In general, the i-th iteration, for $1 \leq i \leq B$, works as follows. Assume that N' leaves are still on and that their inputs have all the same bits from the most significant one up to the $(i-1)$-th bit (this is certainly true initially and after the first iteration, as we saw). Then, let N' and k' be the values currently stored in the root. During the ascending phase,

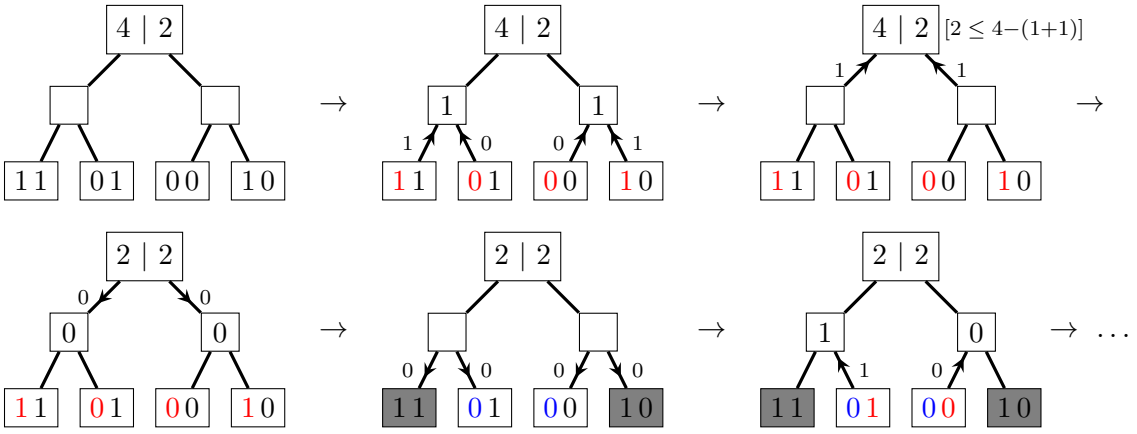

Figure 13: First part of the selection of the 2nd smallest of 4 binary values with 2 bits.

the N' leaves that are still on send upwards the i-th bit of their input. The inner nodes of the binary tree perform the cumulative sum of those bits, which will end up in the root. The total cumulative sum S will be the exact number of remaining inputs that have their i-th bit equal 1. This means that there remain $N' - S$ inputs with i-th bit equal 0 and S inputs with i-th bit equal 1. Furthermore, since they all have the same $i-1$ leftmost bits $b_1\, b_2 \ldots b_{i-1}$, they can be separated by the pivot $b_1 2^{B-1} + b_2 2^{B-2} + \ldots + b_{i-1} 2^{B-i+1} + 2^{B-i}$. Then, the descending phase begins with the root communicating to its children that the i-th bit of the requested element must be either 0 or 1, depending on whether $k' \leq N' - S$ or not. The root also updates its stored values correspondingly, as already explained above. At the end of the descending phase the N' leaves still on will receive the bit information and decide whether to turn off or stay on depending on the i-th bit of their input. So, after the i-th iteration the only leaves still on will be those containing inputs whose i leftmost bits are exactly those of the requested element, communicated by the root during the previous i iterations. Therefore, after all B iterations have been completed, the only remaining leaves will all contain the same value, which must be the requested element. So, a final ascending phase is performed, where they send such an element upwards and inner nodes forward it up to the root (it does not matter whether it comes from the left or right child since it is always the same), after which the root will contain the requested k-th smallest element of the input set. (Otherwise, the remaining leaves could directly output their value, taking any one as the final output since they are all the same.)

Performances

- The parallel execution time is $\mathcal{T}_{2N-1}(N) = \Theta(B \log N)$ since the algorithm performs B iterations, each taking $\Theta(\log N)$ constant steps, and a final ascending phase for another $\log N$ steps, then the work is $\mathcal{W} = \Theta(B\, N \log N)$.
- The speedup is $\mathcal{S} = O(N)/\Theta(B \log N) = O\left(\dfrac{N}{B \log N}\right)$ w.r.t. a general sequential selection algorithm, such as the *median-of-medians* one, then the efficiency is $\mathcal{E} = O\left(\dfrac{1}{B \log N}\right)$.