Chapter 4

Hypercube Network

The *hypercube* is one of the best performing networks, purely based on computation and efficiency capabilities. In fact, hypercube networks have both low diameter and high bisection width, getting the best of both worlds. A hypercube always contains a power of 2 processing units $P = 2^d$, where d are the *dimensions* of the hypercube. The network is defined inductively depending on d. The base network, for $d = 0$, has only one processor, so it has no links. Then, given one of dimensions d, the hypercube of dimensions $d+1$ is obtained from the former by duplicating it and connecting each processing unit to its corresponding copy. Figure 14 shows the construction of hypercubes up to dimensions 4. Each processing unit is given an index in binary form as a sequence of d bits. Thus, in base-ten, the $P = 2^d$ processing units will be indexed from 0 to $2^d - 1$. Observe that each processing unit has as many links as the dimensions d of the hypercube. Indeed, the edges of a hypercube can be classified and divided based on the specific dimension to which they belong (as color-coded in the figure). Then, every processing unit is linked to those whose indices differ exactly on one bit from its index, the one corresponding to the dimension of the edge. For example, in the hypercube of dimensions 3, processor 000 is linked to processors 001, 010 and 100.

The hypercube has some interesting structural properties. Even though a hypercube of dimensions $d \geq 1$ is obtained by linking two copies of a $(d-1)$-dimensional one, looking at its links it actually contains $2d$ different (not necessarily disjoint) hypercubes of dimensions $d-1$, separable in d pairs of disjoint ones. For example, a 3-hypercube (i.e. a cube) has 6 square faces, which are 2-hypercubes. Furthermore, it can be seen that a hypercube of dimensions $2d$ can be partitioned, in $\binom{2d}{d}$ different ways, into 2^d disjoint hypercubes of dimensions d. These properties allow to divide some complex problems into simpler sub-problems without losing the potential of the hypercube network, by solving the sub-problems in parallel on smaller hypercubes. Moreover, the hypercube can also *simulate* most of the other networks (at least all those presented in the course) with no or very small slowdown. This means that the parallel algorithms meant for the other networks can be run in approximately the same time on a hypercube with (at most) the same number of processing units.

A hypercube H_P of dimensions $d = \log P$ with P processing units and bidirectional links has the following properties.

- Diameter $diam(\mathrm{H}_P) = d$, i.e. the distance between any two processing units whose indices have all bits different.

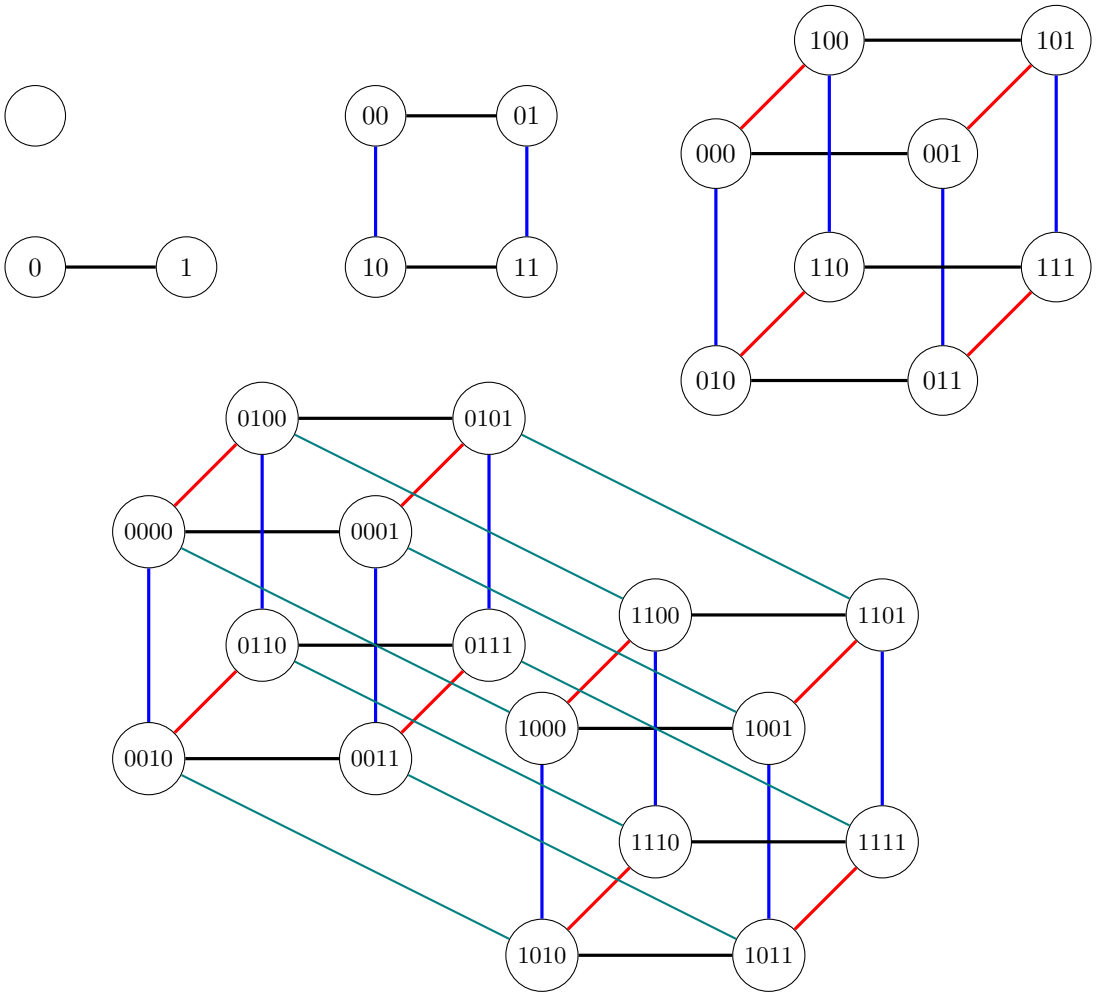

Figure 14: Five hypercubes of growing dimensions $d = 0 \ldots 4$, each with 2^d processing units.

- Bisection bandwidth $b(\mathrm{H}_P) = P/2$, i.e. the number of edges belonging to any one dimension, which connect a pair of disjoint lower $(d-1)$-dimensional hypercubes.

4.1 Associative Operations

Many basic operations that we already saw can be efficiently performed also on the hypercube. For instance, the cumulative application of an associative operation $\otimes$ on an input sequence $\vec{x}$ of size N, power of 2, can be easily computed on the hypercube of dimensions $d = \log N$. Indeed, such a hypercube has $P = 2^d = N$ processing units. Even with constant memory, if initially every input x_i is in processor i, the network can compute the operation $x_0 \otimes x_1 \otimes \ldots \otimes x_{N-1}$ in logarithmic parallel time, putting the result in processor 0 (i.e. $0_{d-1} \ldots 0_1 0_0$ in binary), similarly to what can be done with a binary tree network, but needing just about half of its processors ($P' = 2N - 1$).

The procedure is quite simple. In the first step, every processing unit i whose binary index starts (from the right) with 1, that is, its least significant bit is 1, sends its input value x_i through its link belonging to the first dimension 0. Then, every time a processor receives a value y from a link belonging to dimension k it will simply update its stored value

$x = x \otimes y$ and send that result through the link belonging to the next dimension $k+1$, if k was not the last dimension $d-1$ yet. Eventually the values will travel through all dimensions, from lowest to highest, at which point the receiving processors will just update their values and the computation will terminate. Therefore, the procedure takes d steps, one for each dimension. Observe that we are first moving and applying the associative operation to pairs of values originally in processing units whose indices differ only on the least significant bit, and after, one by one, the more significant ones. Therefore, the values in each pair involved in the first step will be adjacent in the original input sequence. Then, each of those pairs will be used together with another adjacent pair, and so on. In this way, the inputs of every binary operation are preserved, working even without commutativity. In the last phase, processor 0, currently containing the partial result $x_0 \otimes x_1 \otimes \ldots \otimes x_{N/2-1}$, will receive from processor 2^{d-1} the other half $x_{N/2} \otimes x_{N/2+1} \otimes \ldots \otimes x_{N-1}$, and will compute the final output. The computation of the cumulative sum of the values $1, 1, 0, 1, 3, 1, 0, 2$ on a 3-dimensional hypercube is shown in Figure 15.

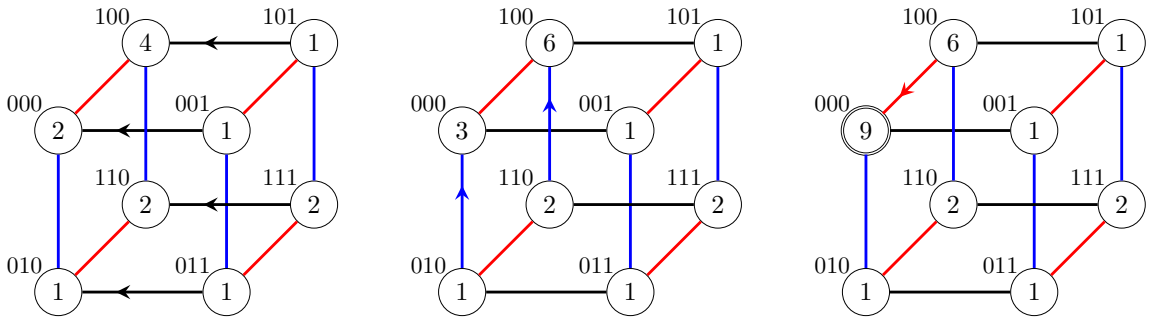

Figure 15: Addition of 8 values on a hypercube of dimensions 3.

Performances

- The parallel execution time is $\mathcal{T}_N(N) = \Theta(\log N)$ since the procedure takes $d = \log N$ constant steps, as long as the operation itself is constant, 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 $\mathcal{E} = \Theta\left(\frac{N/\log N}{N}\right) = \Theta(1/\log N)$.

4.2 Communications in the Hypercube

Before continuing with more algorithms, we will explore a few techniques to move data between processing units in a hypercube. As we will see, these communications take at most logarithmic time w.r.t. the number of processing units P. Since any interesting computation will usually take at least logarithmic time already, such communications will typically not impact the asymptotic parallel time complexity of most algorithms, unless they are employed sequentially multiple times proportionally to the input size.

4.2.1 Message Routing

In this section we describe how a message, that is some data, can be efficiently delivered from any processing unit i to another j, in a hypercube of dimensions d. Whichever are the source and the destination, this process will take at most d constant steps, so logarithmic in the number of processing units P. We assume that the source processor $i = i_{d-1}, \ldots, i_1, i_0$ in binary, knows the message x to be delivered and the index of the destination processing unit $j = j_{d-1}, \ldots, j_1, j_0$. First, processor i computes the bitwise XOR operation $r = i \oplus j$, the result r determines the (shortest) route that the message should travel. So, it sends the message x through the link belonging to dimension k such that r_k is the least significant (rightmost) bit of r equal to 1. Along with x, the processing unit will send also $r_{d-1}, \ldots, r_{k+1}, 0, \ldots, 0$, i.e. r where r_k has been changed into 0. Then, when a processor h will receive the pair x, r, and r is not yet only 0s, it will just forward, through the link belonging to the dimension identified by the least significant 1 in r, the pair x, r, after zeroing that bit in r as before. Eventually r will contain only 0s, and the processor receiving the pair at that point will be exactly the destination j. In fact, recall that neighbour processors linked by an edge belonging to dimension k have indices that differ only in bit k. Therefore, starting from processing unit i and traversing all and only the edges corresponding to non-zero bits in the XOR with the destination j, the message must have reached a processor whose index differs from i only on such bits, which is exactly the destination j, by definition of the bitwise XOR operation. Since at most d bits (all) can be different between i and j, the maximum number of (constant) steps required to deliver the message to its final destination is d.

4.2.2 Circular Shift

Another basic communication procedure is the so-called circular shift. A *circular k-shift* in a d-dimensional hypercube, for $0 \leq k \leq 2^d - 1$, is the cyclic (modular) redistribution of some data already contained in each processing unit i to another k positions after, whose index is $i+k \bmod 2^d$. So, a circular k-shift rotates data by k positions along all processing units of the hypercube. The procedure works similarly to the previous routing method, that is, every single message travels and is delivered through the edges, from lowest to highest dimension, as specified by the difference in bits (XOR) between the source i and the destination $i+k \bmod 2^d$. However, this time there are as many messages as many processing units, and we want all of them to move simultaneously in parallel. Moreover, as usual, we assume that processors have only constant memory, so that they cannot hold more than one message at a time. Interestingly, the same procedure used before for routing a single message has the property that, for the destinations defined above, even if all processing units simultaneously send their message, two different messages will never reach the same processor at the same time. Thus, congestion is always avoided and all communications can actually happen in parallel, so that the computation takes as many constant steps as the routing of a single message, i.e. d steps. Note that this is achieved by sending, at each step, pair of messages through the same (bidirectional) link in opposite directions, making the corresponding pair of processors exchange such messages. So, the circular k-shift is performed by, first, having every processing unit i compute its own $r = i \oplus (i+k \bmod 2^d)$, and then routing all messages as described in the previous section. The difference is that, this time, possibly all processing units will be active in parallel, each one forwarding a different message. The circular 3-shift in a 3-dimensional hypercube is shown in Figure 16,

where initially processing unit i contained data i, for all $0 \leq i \leq 2^3-1 = 7$. For example, the message 2, initially contained in processor $2 = 010_2$, has to be delivered to $2+3 \bmod 8 = 5 = 101_2$. Indeed, the XOR operation $010 \oplus 101 = 111$ specifies that the message has to travel through all dimensions: $010 \to 011 \to 001 \to 101$.

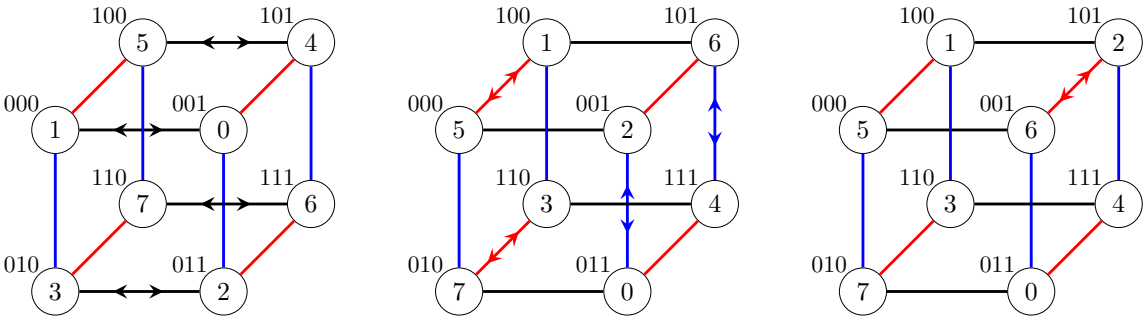

Figure 16: Circular 3-shift in a hypercube of dimensions 3.

The reason why every message reaches, in at most d steps, its destination is the same explained in the previous section for the routing of a single message. On the other hand, we still need to prove that the chosen communication pattern never suffers from congestion. This can be deduced from a technical fact about the results of the bitwise XOR operations $r = u \oplus (u+k \bmod 2^d)$ and $r' = v \oplus (v+k \bmod 2^d)$ initially performed by every pair of neighbour processing units u and v. Since they are neighbours, their indices must differ only in exactly one bit, say bit h, i.e., we must have that $u_h \neq v_h$ and $u_i = v_i$ for all other $i \neq h$. Without loss of generality, we can assume that $u_h = 1$ and $v_h = 0$, otherwise we could just swap them. Then, $u+k = (1+k_h)2^h + \sum_{i \neq h} (u_i+k_i) \cdot 2^i$ and $v+k = k_h 2^h + \sum_{i \neq h} (u_i+k_i) \cdot 2^i$, which means that $u + k = v + k + 2^h$. Let $u' = u + k$ and $v' = v + k$, hence $u' = v' + 2^h$. Clearly, the addition of 2^h cannot change the first least significant h bits of v', but will always change the bit h, so we must have that $u'_i = v'_i$ for all $0 \leq i \leq h-1$ and $u'_h \neq v'_h$. Also note that the $\bmod 2^d$ part in the computation of r and r' does not change the first least significant d bits, in fact it is equivalent to truncating the binary versions of u' and v' to those first d bits. From these facts we can conclude that the first least significant $h+1$ bits of r and r' must be the same, i.e., $r_i = r'_i$ for all $i \leq h$. Indeed, for $0 \leq i \leq h-1$ we have that $u_i = v_i$ and $u'_i = v'_i$, while $u_h \neq v_h$ and $u'_h \neq v'_h$. Thus, for $0 \leq i \leq h$, both XOR operations $r_i = u_i \oplus u'_i$ and $r'_i = v_i \oplus v'_i$ will give the same result. This tells us that messages starting from neighbour processing units, directly linked along dimension h, will travel along the same dimensions, from lowest to highest, up to dimension h included. In turn, this implies that, up to dimension $h-1$, if they move they will do so in parallel through pairs of neighbour processors always linked along dimension h. So, whenever a message, that was initially in processor u, needs to be sent through a link belonging to dimension i, actually it will be exchanged through such a link with the message that initially started from processor v neighbour of u along that dimension i, since their routes shared the same travel directions up to dimension i included. Thus, congestion is not possible, every time a processor receives a message through a link, it just sent its own message through that same link.

4.2.3 Broadcast

The last communication type that we will see is the *broadcast*, that is, the communication of some data from one processing unit to all the other in the hypercube. Given a hypercube of dimensions d with $P = 2^d$ processing units with constant memory, let s be the index of the source processor, who currently contains the data x to be broadcasted. Initially s sends x to all its neighbours. We assume that all the other $P - 1$ processing units do not know who s is, and can only forward the message x through links decided by a fixed rule. So, we need a forwarding rule such that every processing unit, other than s, receives the message x exactly one time. There are many different rules that achieve this. A simple one is to make processors forward any message, received from dimension k, to all and only those neighbours which are connected through an higher dimension $h > k$. The procedure is exemplified in Figure 17 below, where processing unit $2 = 010_2$ broadcasts some data x in a hypercube of dimensions 3. We now show that the rule we gave satisfies the desired

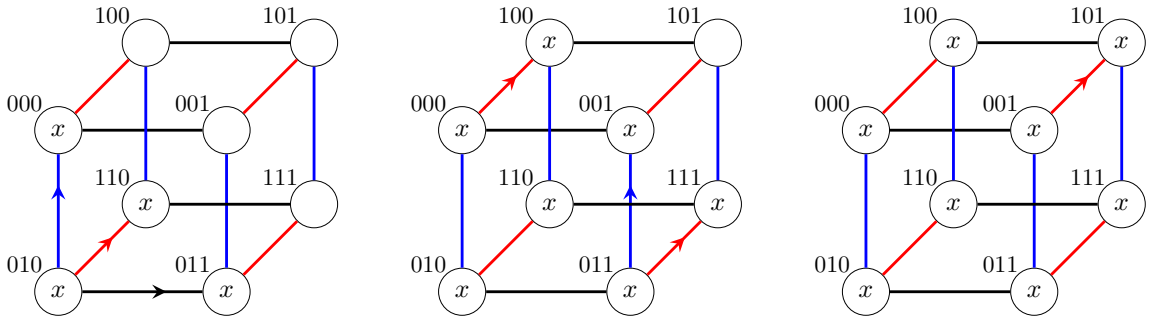

Figure 17: Broadcast of a value x from processing unit 010 in a 3-dimensional hypercube.

property. First, recall that, since all messages start from processor s, to reach processor t they must at least travel through the dimensions corresponding to bits equal 1 in the bitwise XOR $s \oplus t$. Moreover, since processors only forward messages through strictly higher dimensions, every route travelled by every message traverses dimensions from lowest to highest, at most one time each. Therefore, when a message reaches processing unit t it must have travelled exactly through every dimension k such that $(s \oplus t)_k = 1$, lower dimensions first. Since there is a unique path going through all and only such dimensions from lowest to highest, and only one message is ever sent through each different path, it is not possible that a processing unit would ever receive two messages. Furthermore, every processing unit $t \neq s$ will eventually receive a message because a path described as before always exists, and every processing unit along such a path, starting with s, will forward the message through, in particular, the next dimension specified by that path. Since every path traverses dimensions in strictly increasing order, at most all of them without repetitions, the maximum number of steps required to broadcast some value to all other processing units is the number $d = \log P$ of dimensions of the hypercube.

4.3 Longest Common Substring

A classic pattern matching problem on strings is that of finding the *longest common substring* of two strings over some alphabet. Typically, one string t, called the *text*, is longer than the other s, referred to as the *pattern*. The longest common substring problem asks to

find the longest string x that is a substring of both t and s, answering with the length of x and the index where such substring starts in s. We will write $|w|$ for the length of a string w. Note that the simpler problem of checking whether the pattern s is itself a substring of the text t is a special case of the longest common substring problem. Indeed, if the output of the latter is the whole length of s (with starting index 0), then this means that the whole pattern is a substring of the text. To find the longest common substring one can check all possible alignments of the two strings, then comparing them symbol by symbol. There are as many possible alignments as the length of the string t (assuming $|t| \geq |s|$), since symbol s_0 can be aligned to every symbol t_i of the text. After fixing an alignment, one can just search for the longest common substring w.r.t. to that alignment, only comparing aligned symbols. For example, consider the following strings and alignments

$t =$	$a\ a\ a\ b\ b$	$a\ a\ a\ b\ b$
$s =$	$a\ a$	$a\ a$
longest common aligned substring:	a	$a\ a$

The first alignment, on the left, produces a longest common substring of length 1, while the second alignment, on the right, produces one of length 2, which is also (one of) the longest common substring in general. Sequentially, this procedure would take $|t| \cdot |s|$ steps, since there are $|t|$ possible alignments and $|s|$ comparisons are performed for each of them. If $|s| = |t|$, the complexity becomes quadratic in the length of the strings.

Let the two input strings t and s be such that s is at most as long as t. We assume that both strings finish each one with a different special symbol, e.g. '#' for t and '$' for s, also different from all the others symbols in the strings. If they are not like this already, one can just append such symbols to the end of the corresponding string. Furthermore, if s is shorter than t, we suppose to append more instances of the special symbol '$' to the end of s until it has the same length of t. Now, letting N be the length of t (and s), we will assume that N is a power of 2 and let $d = \log N$. The parallel algorithm for solving the problem operates on a hypercube of dimensions $2d = 2 \log N$, which has $P = N^2$ processing units with constant memory. As we already observed, such a hypercube can be partitioned, in multiple ways, into $2^d = N$ disjoint hypercubes of dimensions $d = \log N$, hence with N processing units each. Via this kind of partitioning, the whole hypercube can be conceptualised as a $N \times N$ mesh where each row is a d-dimensional hypercube, whose processing unit are still ordered as they are in the hypercube. Note that, since the rows are hypercubes (instead of linear arrays), the processors in them will not be linked in the usual way (linearly along the row). Nevertheless, we can now index processors either by their original binary index in the whole $2d$-dimensional hypercube, or by their position in the imaginary mesh. So, processor (i, j) will be the one in row i column j of the mesh, for $0 \leq i, j \leq N-1$, and will have binary index $i_{d-1}, \ldots, i_0, j_{d-1}, \ldots, j_0$ in the hypercube, where we concatenated the binary versions of i and j. Indeed, processors with the same d most (resp. least) significant bits in their binary indices correspond to a single row (resp. column) of the mesh. Moreover, such processors always form a d-dimensional hypercube, as wanted. For example, a 4-dimensional hypercube can be seen as a 4×4 mesh where each row is actually a 2-dimensional hypercube, as depicted in Figure 18, where processors have been coloured based on the row of the mesh they belong to.

We can now describe the initial state of the network and the three phases of the algorithm. Initially each processing unit (i, j) will receive as input a copy of symbols t_j and s_j, independently of i, which means that all processors in each column j of the mesh will get the same pair of input symbols t_j and s_j. From there the computation begins.

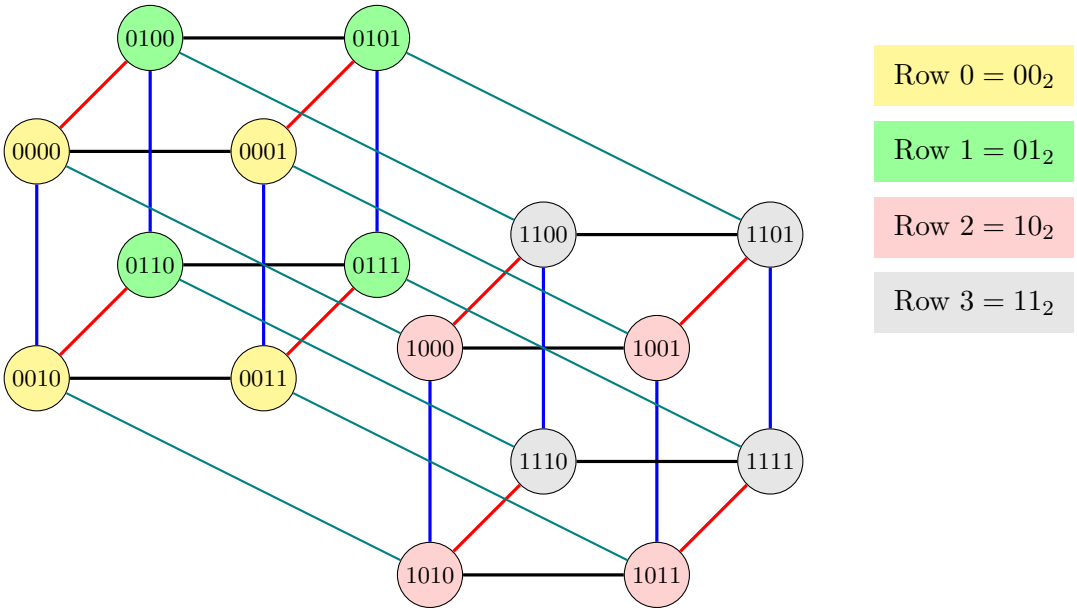

Figure 18: A 4-dimensional hypercube conceptualised as a 4×4 mesh.

1. In the first phase the symbols of the string t in each row i of the imaginary mesh rotate by i positions to the right. To do this, for $0 \leq i \leq N-1$, the d-dimensional hypercube corresponding to row i performs a circular i-shift, using the procedure explained in the previous section, which takes at most d constant steps. After all those d-dimensional hypercubes, in parallel, have completed their corresponding circular shift, each processing unit (i, j) will now contain symbols $t_{j-i \bmod N}$ and s_j. So, all N possible rotations of t have been generated, one unique rotation per row of the mesh.

2. The second phase is the core of the algorithm. For each row of the mesh, so for each different alignment of t and s attained by the previous phase, the current phase has the purpose of finding the longest common substring considering only those which are aligned in that row. The computation is performed in parallel by each d-dimensional hypercube corresponding to each row of the mesh. First of all, every processor (i, j) checks, in constant $O(1)$ time, whether its two stored symbols are the same or not, recording the result. Then, hypercube/row i performs the cumulative application of an elaborate but still constant time associative operation. As we know, this will take d steps. The operation calculates the starting index and length of three common substrings of a given part of the two strings t and s (initially just the pair of symbols in each processor): the longest common aligned substring starting at the beginning of that part, the longest one finishing at the end, and the longest one in general. It may seem unnecessary to compute the starting index and the length of all those substrings, since we are actually only interested in the longest in general, but the other two are still needed to correctly update and identify the general longest one when two different parts of the strings are conceptually joined together, to obtain the values for larger parts up to the whole t and s. To ease understanding, consider the following example showing the hinted process for finding the longest common aligned substring of the strings *aabaaaba* and *aaaaaaaa*, distributed among 8 processors.

	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7
step 0:	a	a	b	a	a	a	b	a
	a	a	a	a	a	a	a	a
step 1:	a	a	b	a	a	a	b	a
	a	a	a	a	a	a	a	a
step 2:	a	a	b	a	a	a	b	a
	a	a	a	a	a	a	a	a
step 3:	a	a	b	a	a	a	b	a
	a	a	a	a	a	a	a	a

The symbols highlighted in red are those belonging to the overall longest common substring in the corresponding part of the strings, while those in blue belongs to the trailing longest common substrings (if they are not the same of the former). Note that there is no need to record the actual three common substrings nor the parts of t and s they refer to, indeed it is enough to record their starting index and length, so constant time and memory will suffice to perform the operation even for growing parts of t and s. Moreover, observe that, although not fully described, the operation is in fact associative but not commutative, because the parts of t and s that are conceptually joined at each application are required to be consecutive. At the end of this phase, the first processor $(i, 0)$ of each row i will contain the starting index and the length of the longest common aligned substring of t and s, for the alignment corresponding to that row.

3. The final phase of the algorithm is quite simple. We just need to find the longest substring among those given by the results of the previous phase. We know that each of those results currently resides in the first processor of each row, that is, in the first column of the imaginary mesh. For example, in Figure 18, the first column would comprise processors: 0000, 0100, 1000, 1100. Observing that, as it was for the rows, also the columns are d-dimensional hypercubes, although different ones, it is enough to compute the maximum of the previous results in the hypercube corresponding to the first column. This can again be performed by the cumulative application of another associative operation: the maximum based on length. So, after another d constant steps, the output will be located in the first processor of the first column, that is $(0, 0)$, which is also the first processor 0 of the whole $2d$-dimensional hypercube.

Performances

- The parallel execution time is $\mathcal{T}_{N^2}(N) = \Theta(\log N)$ since each one of the three phases described above takes $d = \log N$ constant steps, then the work is $\mathcal{W} = \Theta(N^2 \log N)$.
- The speedup is $\mathcal{S} = \Theta(N^2)/\Theta(\log N) = \Theta(N^2/\log N)$ w.r.t. a standard sequential algorithm considering all possible alignments of the two strings and comparing them symbol by symbol, then the efficiency is $\mathcal{E} = \Theta\left(\frac{N^2/\log N}{N^2}\right) = \Theta(1/\log N)$.