Part II

Interconnection Network Architectures

Chapter 2

Linear arrays

2.1 Connecting Processors

As we mentioned, in the mid-eighties low production cost of processor chips enabled computer manufacturers attempt building machines that could contain hundreds, even thousands, of processors. However, obtaining cheap processors is only half of the manufacturing problem: The other half is, how do you connect them so they can communicate fast. For now, we leave for later the problem of how to program such a machine once you have built it.

There are several ways researchers and manufacturers came up in building parallel machines. For example, given 9 processing elements, any graph on 9 nodes is a potential interconnection network. Figure 1.5 shows some of the simpler ones: a linear array L_9, a circular ring C_9, a star S_9, a binary tree T_9, a 2-dimensional array D_9, or maybe a random network R_9. Which of them is the best way of connecting the processing elements?

Maybe be the way to go is to connect every pair of PEs, creating one direct connection between them. This is known as the complete graph K_9. (Figure 2.1). This approach seems to incur a large cost, however, since so many wires need to be implemented. As a matter of fact, there are 36 connections in K_9 and, in general, $n(n-1)/2$ connections in K_n. Can

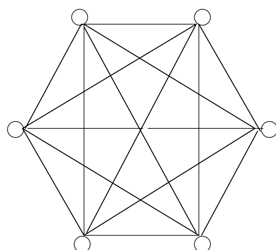

Figure 2.1: The complete graph K_6.

we do that? Moreover, even if we actually can do that, is the trouble of implementing a complete graph worth the cost?

Clearly, to be able to answer such questions, we need some way of measuring the "goodness" of each candidate.

2.2 Network Characteristics

We would like to compare all networks containing n nodes. Apparently, the number of edges in a network is one of the parameters we need to consider. In addition, we are also interested in its *diameter*, its *bisection width*, its *I/O bandwidth* and *vertex degree*, since they are proven to be crucial factors in determining the network "goodness". We also care about the existence of *symmetry* and *recursive structure* in them. In this section we will describe each one of them in some detail.

2.2.1 Number of Edges

We already got a tase of why the number of wires is an important factor in the speed of a network. Simply put, more wires mean faster communication between processors. For example, the complete graph guarantees one-step communication between any pair of processors. However, every complete graph with more than 4 vertices is not planar. This means that one cannot create a one-level chip that contains more than 4 processors. To connect

5 or more processors on a single chip, one has to create multiple layers on the chip which is a difficult engineering task. So, clearly, we would like the network not to have too many wires. Looking at the examples in figure 1.5 above, we can deduce that all the networks but the complete graph are easy to build.

2.2.2 Diameter

In a network, information needs to travel from one processor to another. If two processors are distance d apart, i.e., if d wires need to be traversed to get from one processor to the other, then d steps will be necessary for two processors to communicate every time they do. Naturally we are interested in the maximum distance between any pair of processors. We call this maximum distance the *network diameter*. In figure 1.5 above, K_9 has diameter 1, while R_9 has diameter 5.

Networks with small diameters are preferable. Note that if the diameter of a network is d, then this network may run some algorithm d times slower than one that has diameter 1. In fact, the diameter, sometimes (but not always), sets the lower bound for the running time of an algorithm performed on the network.

Bisection width

A third characteristic of network performance is its *bisection width*. To *bisect* a network is to disconnect it into two pieces, each one containing roughly the same number of processors. (We say "roughly' since in a network with odd number of processors, one piece will have one more processor than the other after the bisection.) The *bisection width* is defined as the minimum number of wires removed in order to bisect a network.

To see why the bisection width is an important characteristic, consider the following example: Sometimes, results calculated by one half of the

network might be needed by the other half. If the bisection width of the network is b, which is much smaller than n, the network will spend n/b steps just shipping values around. A larger bisection width enables faster information exchange, and is, therefore, preferable.

K_9 has a bisection width of 20. In general K_n has a bisection width of $(n/2)^2$, if n is even, and $(\lfloor n/2 \rfloor) \cdot (\lceil n/2 \rceil)$ if n is odd. To see that, consider pulling half of the vertices of K_n away from the other half. The edges that will be stretched are those connecting the two halfs: each edge in one half is connected to each edge of the second half.

I/O bandwidth

By *I/O bandwidth* we mean the number of input lines in the network. This number apparently affects the computational speed of the network since the time to load the processors may dominate the time to solve a particular problem. In general, the larger the bandwidth, the better; but also the more expensive to build the network.

Vertex degree

Each chip in a parallel machine may contain one or more interconnected processors. Theoretically, a network may have a number of connections that is a function on n, the total number of processors in the system, but in real life a chip has a *fixed* pre-specified number of connections. The lower this number, the easier to build the network.

Symmetry

If the network seems the same from any processor, we say that it has symmetry. Symmetric networks are desirable, since they simplify the resource management and the algorithm design.

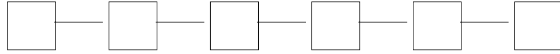

Figure 2.2: A 6-element linear array

Recursive Structure

A network has recursive structure if every instance of it can be created by connecting smaller instances of the same network. For example, a network with n nodes created by connecting in some fixed way four networks of $n/4$ nodes each has recursive structure. This property often makes scalable computers, an important feature of successful machines.

2.2.3 Arrays

2.3 Description and properties

In this chapter we will investigate linear arrays, the simplest parallel network, by studying various sorting algorithms performed on them (Figure 2.2. The following figures illustrate the basic architecture of these 1-D arrays. Before we start, however, we will introduce some terminology and factors that will be used to measure the power of the networks.

So, keep these factors in mind and try to calculate them every time we introduce a new network.

The linear array of n processors has:

- diameter $n-1$. Many times this is responsible or lower bounds.
- bisection width 1 (Figure 2.3).
- I/O bandwidth anywhere between 1 and n. When it is 1 or 2, it is often referred to as a "systolic array".

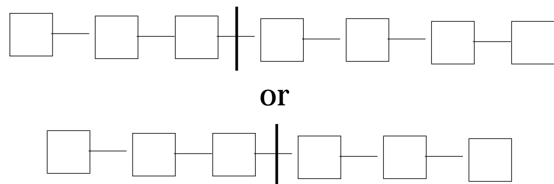

Figure 2.3: The bisection widths of linear arrays

- vertex degree 1 or 2.
- it is not symmetric, but the closely related *ring* is.

2.4 Algorithms

2.4.1 A first sorting algorithm — Zero time sorting

One of the first sorting algorithms developed on systolic arrays is described next.

We keep two variables per processor initialized to hold a special value greater than any number, which we call $+\infty$. Data enter from the left end and of the array. In each processing element of the following step is executed:
Input phase step: Receive an element from your left and compare it with the one you are currently holding; keep the minimum and pass the maximum to the right.

After all numbers have been inputted into the array the first processor holds the minimum number; also, when some number reaches the last processor, all elements are sorted. So, at that point we may output the sequence in sorted order. However, we may save some time by outputting numbers *as soon as* the last number has been inputted to the array [1]. Figure 2.4 has an example. The special marker (#) is used to sign the end of the input and to switch the processors to execute the following step:

[1] This is the reason that this algorithm is called zero-time sorting.

Output phase step: Receive an element from your right and pass it to your left.

As you can see in the figure, we need two variables per processor that will hold the numbers to be compared. *Local* will hold the minimum of the two numbers while *temp* will hold the maximum that will be passed to the right.

The code that each processor has to execute is as follows:

```
Procedure  zero-time-sorting
    let  local := +∞
    let  temp := +∞
    repeat        /* input phase */
          temp := receive from left port
          if (temp = #) then
                send temp to right port
                exit repeat
          else
                send max(local, temp) to right port
                local := min(local, temp)
          end if
    end repeat
    send local to left port
    repeat        /* output phase */
          local := receive from the right port
          send local to left port
          if local = # then exit repeat
    end repeat
```

The time for this procedure is $3n$ using n processors, therefore the total

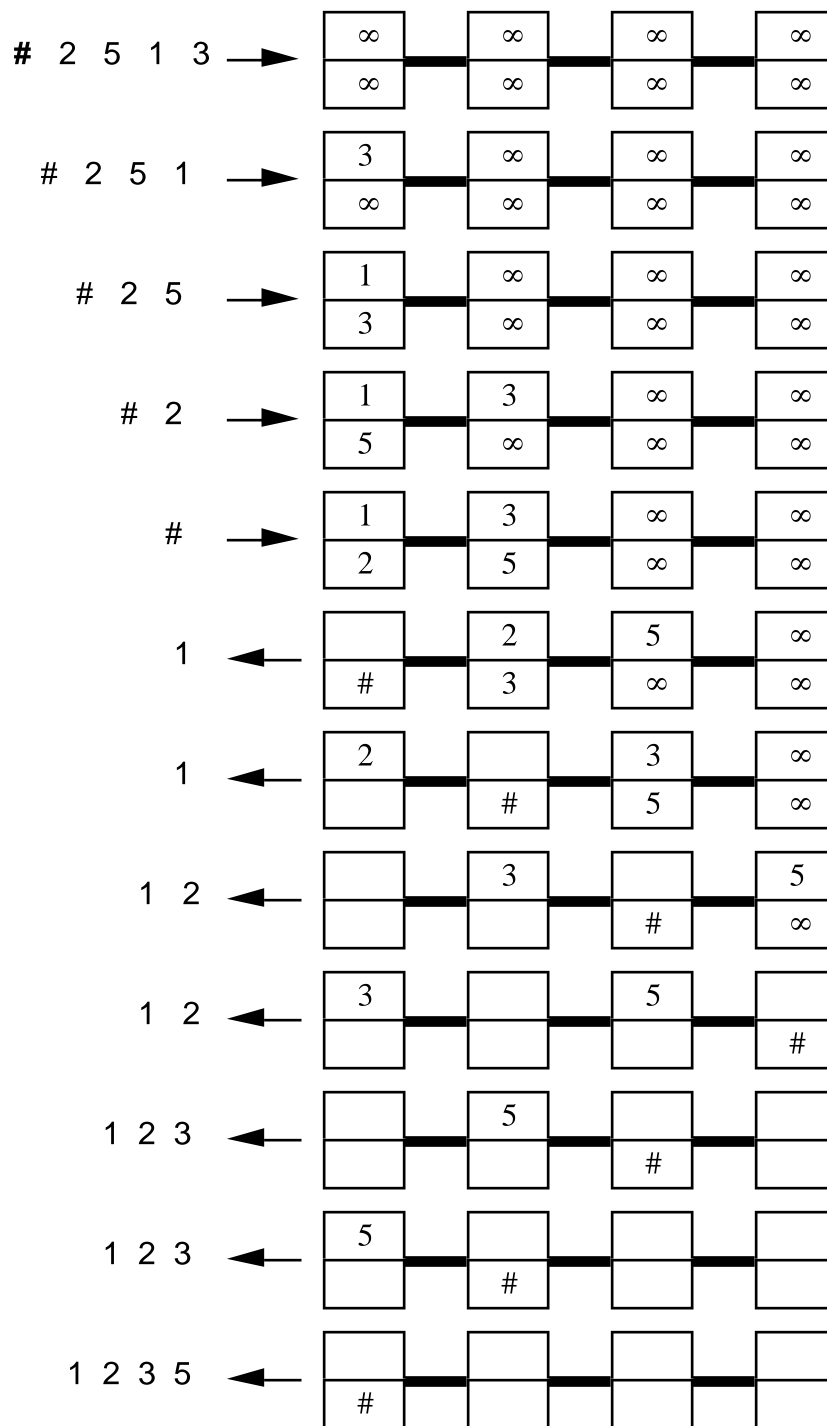

Figure 2.4: An example for zero-time sorting. Observe that immediately after the last number entering the array (step 5), the first sorted number comes out (step 6).

work is $3n^2$.

2.4.2 The "0-1" Sorting Lemma

An algorithm is *oblivious* if the action taken at each step is independent of any previous steps. For example, bubble sort is oblivious, but counting sort is not. For oblivious comparison-exchange sorting algorithms, i.e., algorithms that sort by performing a sequence of predefined comparison and exchange steps between elements, the "0-1" Sorting Lemma is the most important technique to show the correctness and compute the running times of the algorithms.

Lemma 1 *If an oblivious comparison-exchange algorithm sorts all inputs of 0's and 1's correctly, then it sorts all inputs with arbitrary values correctly.*

Proof: For the purpose of contradiction, let's assume an oblivious comparison-exchange algorithm sorts all inputs of 0's and 1's correctly, but fails to sort the input $a_1, a_2, \ldots, a_n$. Suppose the correct ordering is $a_{\sigma(1)} \leq a_{\sigma(2)} \leq \ldots \leq a_{\sigma(n)}$, where σ is a permutation of the set $1, 2, \ldots, n$. Since the algorithm fails to sort $a_1, a_2, \ldots, a_n$, we can assume $a_{\sigma(1)}, a_{\sigma(2)}, \ldots, a_{\sigma(n)}$ does not appear at any step in sorting, and the output is $a_{\phi(1)}, a_{\phi(2)}, \ldots, a_{\phi(n)}$, where ϕ is a different permutation from σ. Then there exists the smallest index k such that $a_{\phi(k)} \neq a_{\sigma(k)}$. In fact, since $a_{\phi(i)} = a_{\sigma(i)}$ for $1 \leq i < k$, we have $a_{\phi(k)} > a_{\sigma(k)}$, and so $a_{\phi(l)} = a_{\sigma(k)}$ for some $l > k$.

Now let's define a new sequence $b_i = \begin{cases} 0 & a_i \leq a_{\sigma(k)} \\ 1 & a_i > a_{\sigma(k)} \end{cases}$. Notice that if $a_i \geq a_j$ then $b_i \geq b_j$. Since the comparison-exchange algorithm is oblivious, the action taken on two sequences $\{a_i\}$ and $\{b_i\}$ is identical at each step. So the output of sorting the sequence $\{b_i\}$ is $b_{\phi(1)}, b_{\phi(2)}, \ldots, b_{\phi(n)} = 0, \ldots, 0, 1, \ldots, 1, 0, \ldots, 0$ where $b_{\phi(k)} = 1$, since $a_{\phi(k)} > a_{\sigma(k)}$, and $b_{\phi(l)} = 0$, since $a_{\phi(l)} = a_{\sigma(k)}$. But we also know $l > k$, so $b_{\phi(1)}, b_{\phi(2)}, \ldots, b_{\phi(n)}$ is not

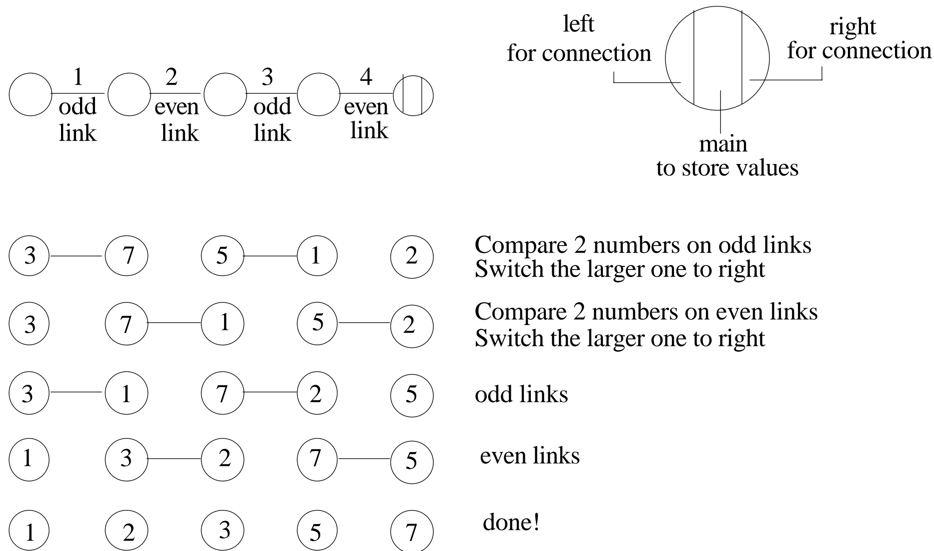

Figure 2.5: Odd-even transposition.

sorted. This contradicts the assumption that the algorithm sorts 0's and 1's correctly. □

The "0-1" Sorting Lemma has the amazing power of reducing the number of input strings to be considered by the sorting algorithm from n^n to 2^n. By the definition of oblivious comparison-exchange algorithm, if it sorts all inputs of 0's and 1's in N steps, it can sort all inputs of arbitrary numbers in N steps. In many cases, this fact greatly simplifies the analysis of running time, since we only need to consider inputs of 0's and 1's.

2.4.3 Odd-even transposition

The odd-even transposition algorithm sorts N numbers on a linear array of N processors. The algorithm compares and exchanges (if necessary) the numbers linked by odd links on odd steps, and the numbers linked by even links on even steps. Figure 2.5 has a simple example.

Stack "Odd-Even Transposition

The sequential running time for bubble sort is $O(n^2)$; the parallel time with n processors is exactly n. Note that each number is at most $n-1$ positions away from its right ordering.

The pseudo code is as follows:

for $i := 1$ **to** n **in parallel do**
  if (odd i) **then**
    if $list[2k].val < list[2k-1].val$ **then** swap them
  else
    if $list[2k].val > list[2k-1].val$ **then** swap them
  end if

An interesting observation is that the largest and the smallest number start moving no later than step 2. They move toward their final positions at each step and stop moving once they are there. This simple fact is useful for odd-even transposition sort. It can be proved (see [**?**]) that it takes N steps to sort N numbers with odd-even transposition.

Lemma 2 *Any linear array sorting algorithm that only uses disjoint comparison-exchange operations as in odd-even transposition needs at least N steps to sort N numbers in the worst case if $N > 2$.*

Proof: Without loss of generality, let's assume we are sorting numbers $1, 2, \ldots, N$. Consider the initial configuration with N in processor 1 and $N-1$ in processor 2 (Figure 2.6).

In order to move number N to the Nth processor in less than N steps, N has to move right in every step. So the configuration after step 1 looks as in figure 2.7.

Since N will move right at step 2, $N-1$ will be stuck in processor 1 for another round. Since after step 2 it will take at least $N-2$ steps to move

Figure 2.6: Initial configuration in an odd-even transposition.

Figure 2.7: Configuration after step 1 in an odd-even transposition

number $N-1$ to processor $N-1$, the total number of steps needed for such an algorithm is at least N. □

2.4.4 Matrix-vector multiplication

We will now see how we can use a systolic array to perform matrix-vector multiplication. Again, timing plays the most important role here: we have to make sure that the data arrive at each processor at the right time for processing.

Stack "Matrix-Vector Mult"

Figure 2.8 describes an example of matrix -vector multiplication. This idea will be essential in the next section when we consider multiplication between matrices.

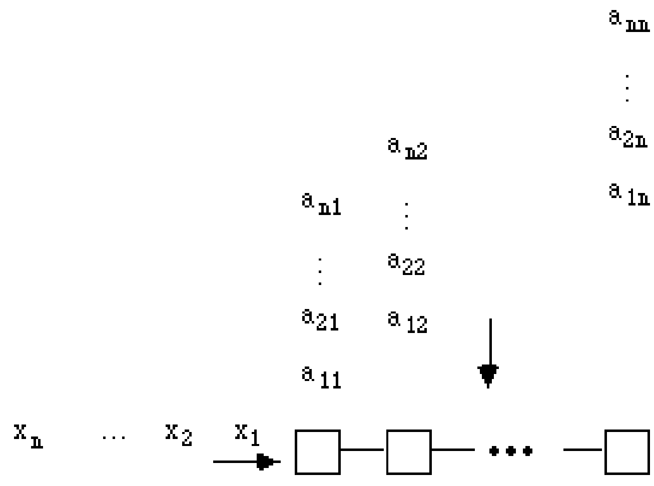

Figure 2.8: Matrix-vector Multiplication

Chapter 3

Meshes

3.1 Description and properties

A straightforward generalization of the linear (1-D) array is the mesh (2-D) array (Figure 3.1).

We can observe that the square mesh has:

- n^2 processors
- I/O bandwidth usually $2n$ (sometimes n^2)
- diameter $2n - 2$
- vertex degree 2 or 4
- almost symmetric

If we also require that there are wraparound edges that connect processors of the first column (resp. row) to those at the last column (resp. row), then it is called a *torus*. Note that the torus *is* a symmetric network. One of the networks that MasPar efficiently simulates is a torus.

The diameter of a mesh is smaller than the diameter of a linear array with the same number of processors, but it is still high. There is a network

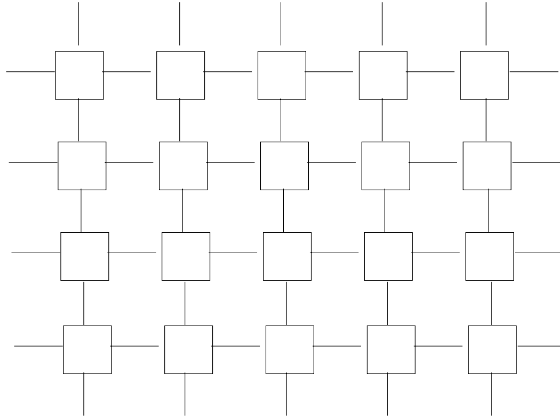

Figure 3.1: A 4×5 mesh

that can reduce it further to $\log n$, the mesh-of-trees, which we will describe in the next chapter.

3.2 The r-dimensional array

A generalization of the mesh is the r-dimensional array $N_1 \times N_2 \times \ldots \times N_r$, defined as follows: each processor is numbered $n_1 n_2 \ldots n_r$, where $1 \leq n_i \leq N_i$ for $1 \leq i \leq r$. There is a wire between processor $u = u_1 u_2 \ldots u_r$ and $v = v_1 v_2 \ldots v_r$ if and only if there is some i, $1 \leq i \leq r$ such that $u_i = v_i$ for $i \neq k$, and $|u_k - v_k| = 1$. The wire between u and v is said to have dimension k. If $N_1 = N_2 = \ldots = N_r = 2$, it is called a *hypercube* of dimension r, a powerful network we will study later.

3.2.1 Diameter

A 2-D array of size $m \times n$ has diameter $m + n - 2$. In general, an r-dimensional array $N_1 \times N_2 \times \ldots \times N_r$ has diameter $N_1 + N_2 + \ldots + N_r - r$. (The diameter is found by considering the distance between processor $11 \ldots 1$ and $N_1 N_2 \ldots N_r$.)

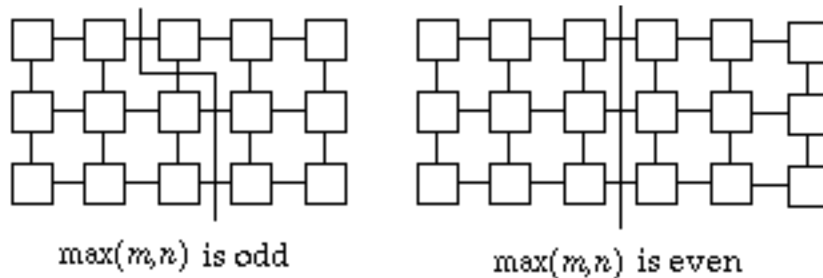

Figure 3.2: The bisection widths of 2-D arrays

3.2.2 Bisection width

Intuitively the bisection width of an $m \times n$ 2-D array is $\min(m, n)$, if $\max(m, n)$ is even, or $\min(m, n) + 1$, if $\max(m, n)$ is odd. (Figure 3.2.)

To prove the claim of bisection width rigorously is not as simple, since all partitions of the array needs are to be considered.

The following theorem from mathematics will be useful in proving Lemma 3. We quote it here without a proof.

Theorem 1 *The area enclosed by a loop with fixed perimeter m can be at most $(m/2\pi)^2 \times \pi = m^2/4\pi$. This maximum value is reached exactly when the loop is a circle.*

□

Lemma 3 *The bisection width of an $m \times n$ mesh is at least* $\min(m, n)$.

Proof: Let's look at the cut going through the wires to be removed in order to bisect. First notice that the cut has to be in one connected piece, otherwise the cut divides the network into more than two pieces. There are two cases to consider: either the cut forms a closed loop, in which case the inside and outside of the loop are two disconnected pieces; or the cut intersects the edges of the 2-D array exactly twice. Let's define the *size* of

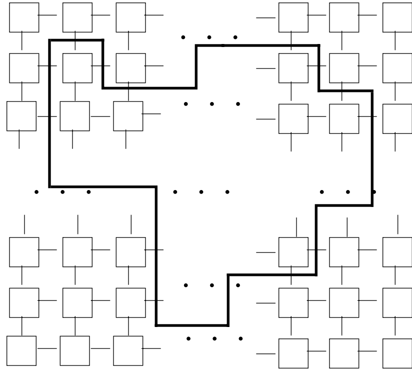

Figure 3.3: The bisection cut forms a loop

a cut to be the number of wires it goes through. Without loss of generality, we can assume $m \leq n$. Since the minimum size of a cut that bisects is the bisection width, it is the same as proving that a bisecting cut has size at least m.

Case 1: If the cut forms a loop, then the number of processor enclosed by the cut is the same as the area enclosed.

By Theorem 1, the area, i.e., the number of processors, enclosed by the loop can be at most $m^2/4\pi$, less than $mn/2$. So a loop of size m can not bisect an $m \times n$ array.

Case 2: Let A, B be the two intersections of the cut and the edges of the array. If A and B are on the same edge, one half of the network is enclosed by the cut and line segment AB. To reach B from A on a cut, at least the horizontal distance $|AB|$ has to be traveled. So $|AB| \leq m$.

If A and B are on the adjacent edges, one half of the network is enclosed by the cut, line segment AC and BC. (where C is a corner of the array. To reach B from A on a cut, at least the horizontal distance AC and the vertical distance BC have to be traveled, so $|AC| + |BC| \leq m$.

In either way, one half of the network is enclosed by a loop with perimeter no larger than $2m$. By Theorem 1 we can conclude that the area, the number of processors, can be at most $(2m/2\pi)^2\pi = m^2/\pi$, which is again less than

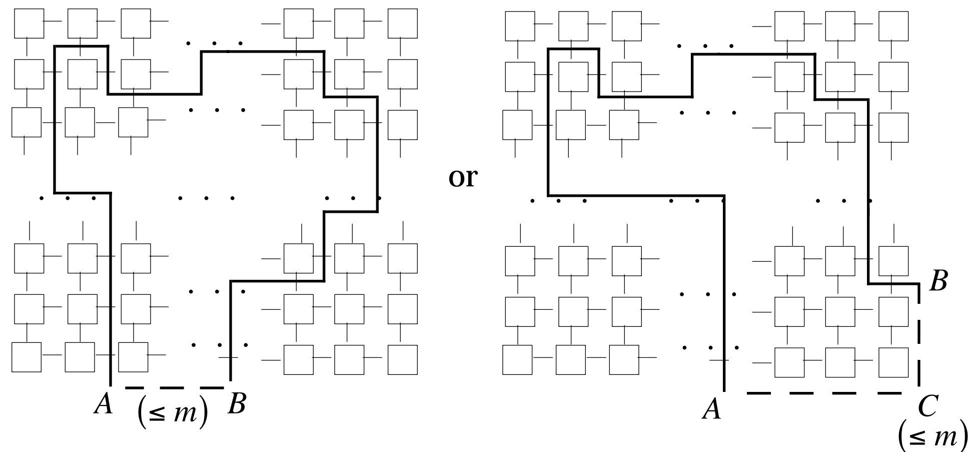

Figure 3.4: The bisection cut intersects the array at A and B.

$mn/2$. Notice that Case 2 also includes some special cases, for example, when A and B coincide. So A and B have to be on the cross edges, and it follows immediately that the size of the cut is at least m. □

3.3 Algorithms for meshes

3.3.1 Discussion

Recall that the best (and, indeed, optimal) sequential sorting algorithm, works in $O(N \log N)$ steps. A square mesh with $N = n^2$ nodes contaning one number per node, cannot be sorted in $O(\log N)$ time, because it may take $O(\sqrt{N})$ steps for the smallest element to move from, say, the lower right location to the upper left location. So, sorting on the mesh cannot be done efficiently since there is apparently a $2\sqrt{N} - 2$ lower bound on sorting. Two immediate questions that arize are:

1. Is there a better (higher) lower bound than $2\sqrt{N} - 2$; and
2. How close can we get to the lower bound.

We will try to answer these questions in the following sections.

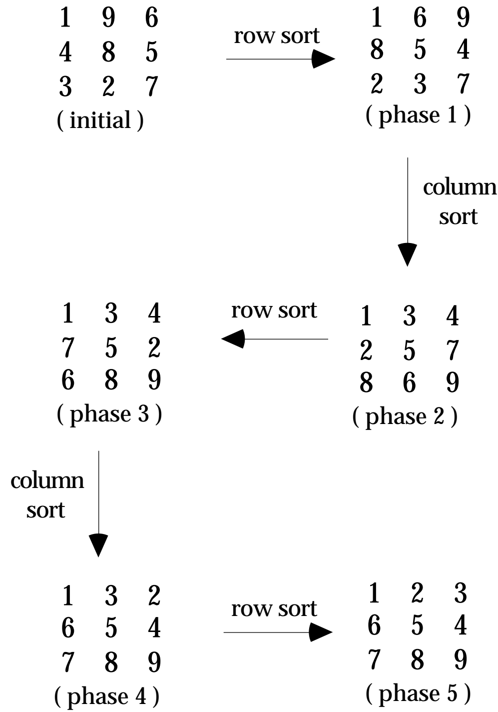

Figure 3.5: Shearsort on a 3×3 array

3.3.2 Shearsort

Employing $n \times n$ processors, with $n = \sqrt{N}$ on a square mesh network, we can try to sort an $n \times n$ array using odd-even transposition sort in $O(n)$ time by repeating the following two steps until fixpoint (i.e., until no change occurs): sort the rows, in $O(n)$ time, and then the columns in another $O(n)$ time, independently. The resulting array at the end of each step is shown is in Figure 3.6. However, this algorithm presents one problem: depending on the order of application of the row sort and column sort, the resulting array may be ordered differently. We use an example to illustrate this issue.

Given 2-D array $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, if we apply row sort first, then column sort, the resulting array is $\begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$; if we apply column sort first, then row sort, the resulting array is $\begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$.

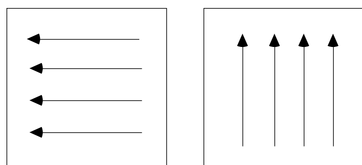

Figure 3.6: The "natural" order of row sorting and column sorting.

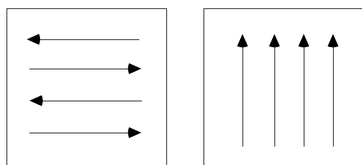

Figure 3.7: Shearsort sorts rows in an ox-turning fashin (left) while maintaining the "natural" order when sorting columns.

A solution to this problem is to first sort rows in ox-turning order (figure 3.7) and then sort the columns in natural order. (Or, to first sort columns in ox-turning order and the rows in natural order.)

This 2-D sorting algorithm is called Shearsort (figure 3.5), which interleaves ox-turning row sort with column sort and runs in $O(n \log n)$ time employing the same number of processors.

0-1 lemma here?

When sorting 0's and 1's, Shearsort has the useful property that the number of *dirty rows* (rows that are not all 0's or 1's) is halved at the end of each column sort. So there are at most $2 \log \sqrt{N}$ phases after the row sort of phase 1. Since each phase performs odd-even transposition which takes $\sqrt{N}$ steps, the total running time is $T(N) = \sqrt{N}(\log N + 1)$.

However, when sorting an arbitrary set of numbers, the dirty rows are not halved at each step. In the example above none of the three rows is sorted until the last phase. This fact does not contradict the "0-1" Sorting Lemma and the analysis of the running time is still correct because the "0-1"

Sorting Lemma guarantees any arbitrary set of numbers to be sorted in the same number of steps as any set of 0's and 1's, so $T(N) = \sqrt{N}(\log N + 1)$ is correct. The number of dirty rows in Shearsort is not a feature captured by the Lemma — a property owned by sorting 0's and 1's does not have to be valid for sorting arbitrary numbers.

*** Improving the running time**

There are many ways to improve the running time of Shearsort. If analyzing carefully, we can take advantage of the fact that fewer and fewer steps are used for successive column-sorting phases. Suppose there are d dirty rows, so it takes d steps using odd-even transposition to sort the columns. If this is taken into consideration the running time is

$$\begin{aligned} T &= T(\text{rowsort}) + T(\text{columnsort}) \\ &= \sqrt{N}(\log\sqrt{N} + 1) + \sum_{d=0}^{\log\sqrt{N}} \sqrt{N}/2^d \\ &= \sqrt{N}(\log\sqrt{N} + 3) - 1. \end{aligned}$$

Compare with the running time before, we have

$$\frac{\sqrt{N}(\log\sqrt{N}+3)-1}{\sqrt{N}(\log\sqrt{N}+1)} = \frac{\frac{1}{2}\sqrt{N}(\log N+1)+\frac{5}{2}\sqrt{N}-1}{\sqrt{N}(\log\sqrt{N}+1)} = \frac{1}{2} + o(1/\log N).$$

This is close to an improvement of a factor of one half.

*** A lower bound**

Theorem 2 *Any comparison-exchange sorting algorithm which sorts inputs into snake-like order, has lower bound* $\Omega(\sqrt{N})$ *(more precisely* $3\sqrt{N} - o(\sqrt{N})$*) if performed on* $\sqrt{N} \times \sqrt{N}$ *arrays.*

Proof: Consider the initial configuration in Figure 3.8. The upper-left (shaded) triangle consists of 0's and N's. The number of 0's or N's are to be

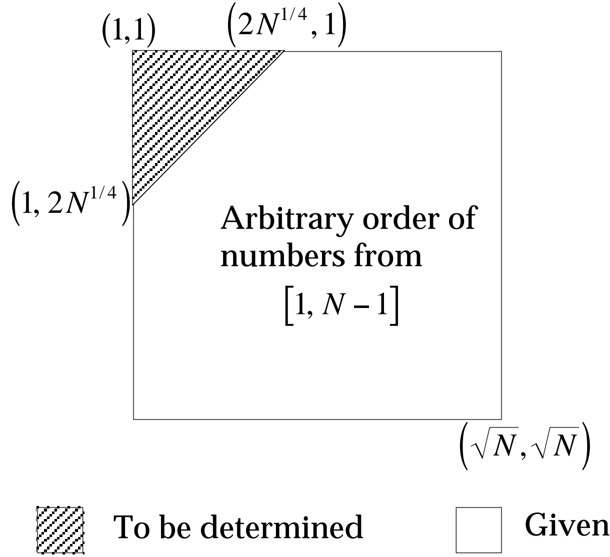

Figure 3.8: The initial configuration

determined, and we will see that the order of the 0's and N's does not matter. Since the algorithm only allows comparisons and exchanges on links, it takes at least $2\sqrt{N} - N^{1/4} - 2$ steps for any information from the shaded triangle to influence cell $(\sqrt{N}, \sqrt{N})$. This is the same as saying at step $2\sqrt{N} - N^{1/4} - 3$, cell $(\sqrt{N}, \sqrt{N})$ does not contain 0 or N. Moreover, the number in $(\sqrt{N}, \sqrt{N})$ is independent of the configuration of the shaded triangle. (This is a little hard to accept at first, but think this way: the influence of the triangle moves like a wave from the upper-left to the lower-right corner. Each step pushes the wave down by one unit in the direction of the main diagonal. The area not yet influenced obviously contains the configuration independent of the triangle. Some simple examples are convincing.)

Let's assume cell $(\sqrt{N}, \sqrt{N})$ contains $x \in [1, N-1]$. Let c be the column number of x after the input is sorted, and m be the number of 0's in the triangle. (Notice that x is independent of m, but c is dependent on m.) Given the ordering of numbers ordering of numbers from 1 to $[1, N-1]$ in the initial configuration, if we vary m between 0 and $\sqrt{N}$, c will reach every number between 1 and $\sqrt{N}$ at least once. This claim can be justified as

follows. If $\sqrt{N}$ is odd, then the last row will be sorted from left to right. Suppose when $m = 0$, the final position for x is $c_0 = k(1 \le k \le \sqrt{N})$. Then for $m = 1$ the final position for x is $c_1 = |k - 1|$. In general for $m = l$, we have $c_l = |k - l|$. When l varies from 0 to $\sqrt{N}$, c_l reaches every value between 1 and $\sqrt{N}$ at least twice. [1] The same argument holds for even $\sqrt{N}$. So we are able to set m to the value which forces c to be 1. Now we can see after step $2\sqrt{N} - N^{1/4} - 3$, at least $\sqrt{N} - 1$ steps will be needed to move x from column $\sqrt{N}$ to column 1. And so the algorithm takes at least $3\sqrt{N} - 2N^{1/4} - 4$ steps. The proof is complete. □

3.3.3 Matrix multiplication

Given two matrices A, B, both of size $\sqrt{N} \times \sqrt{N}$, they can be multiplied in $O(\sqrt{N})$ time using $\sqrt{N} \times \sqrt{N}$ processors. This matrix multiplication algorithm involves preprocessing of the matrices. First, rotate matrix A by 180 degrees about the vertical axis and shear to the right, producing A'. Second, rotate matrix B by 90 degrees, counterclockwise and shear upward, producing B'. We then pass A' in the horizontal direction to the right, and B' in the vertical direction downward, as indicated in Figure 3.9.

Stack "Matrix Multiplication"

The pseudo code for performing the multiplicationis listed below.

Procedure Matrix_ multiplication

for each processor p_{ij} **in parallel do**

$c := 0$

end for

repeat $2\sqrt{N} - 1$ times

[1] We may ponder on the validity of this statement. Why cant we have $\sqrt{2}N^{1/4}$ instead of $2N^{1/4}$ as the size of the triangle on the upper-left corner? It is conceivable on the first thought that $\sqrt{2}N^{1/4}$ would do, since we can follow the same reasoning and c_l would reach every value between 1 and $\sqrt{N}$ at least once. Not quite! The mistake here is subtle: $c_l = k, k-1, \ldots, 2, 1, 2, \ldots, \sqrt{N} - k$. So $2N^{1/4}$ for the triangle is necessary.

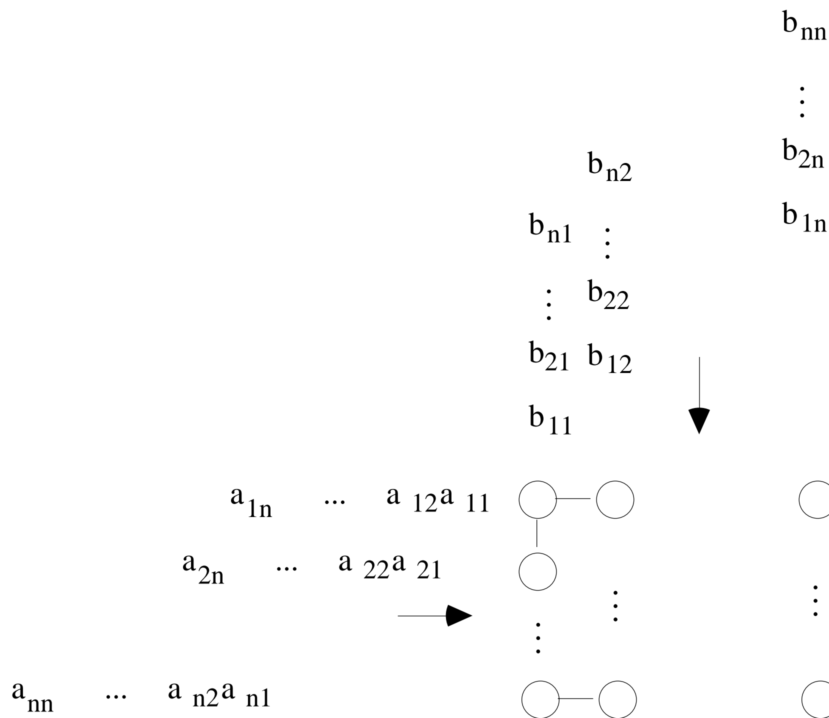

Figure 3.9: Multiplication of matrices A and B.

> **for** each processor p_{ij} **in parallel do**
>> receive a from the left
>>
>> receive b from the top
>>
>> **let** $c := c + a \cdot b$
>>
>> send a to the right
>>
>> send b down
>
> **end for**

end repeat

At the end of the $(2\sqrt{N} - 1)$-st iteration, the values in each of the p_{ij} processors are the elements of the resulting matrix $C = A \times B$, in the exact order.

The total work done by the algorithm is $O((\sqrt{N})^3)$ which matches the *simple* sequential algorithm, therefore it is optimal (yet, not in NC). We

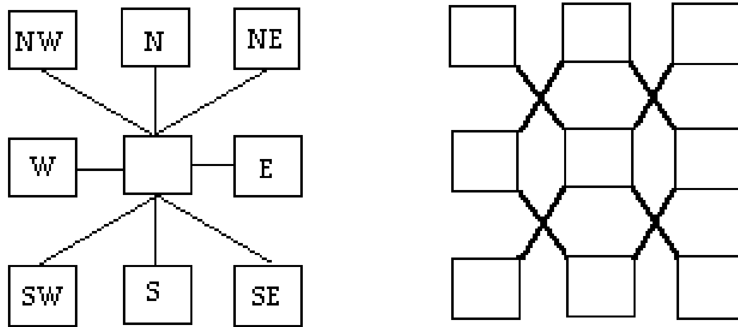

Figure 3.10: The MasPar X-net. On the left is the logical view (what the programmer sees) and on the right the hardware view (the actual implementation).

emphasize the simple here, because there is a very complicated sequential algorithm that can sort in $O((\sqrt{N})^{2.3})$.

3.4 Related Networks

3.4.1 The X-net

The MasPar computer grid does not have the typical mesh connections North, East, West, South (known as NEWS on the CM-2), yet can efficiently simulate them on its so-called X-net (Figure 3.10). In this network, four neighboring processors are connected with two X-like wires, which have a switch in the middle. This way it is very easy to also have the NE, NW, SE and SW connections at no extra cost.

Chapter 4

Trees

4.1 Description and properties

Another inexpensive way to connect n processors is by connecting them in a binary tree structure as shown in Figure 4.1. A full binary tree with n nodes has $n - 1$ edges. Each node has a connection to its parent node, plus two connections to its left and right children, with the exception of the root (which has no parent), and the leaves (which have no children). Its characteristics are:

- I/O bandwidth $\lceil \frac{n}{2} \rceil + 1$ — typically I/O connections are held at the root and the leaves.
- diameter $2\lceil \log n \rceil$.
- bisection width 1,
- vertex degree 1, 2, or 3, and
- a recursive structure.

So it is comparable to the linear array with which it shares several characteristics (namely small bisection width and low, fixed vertex degree), but

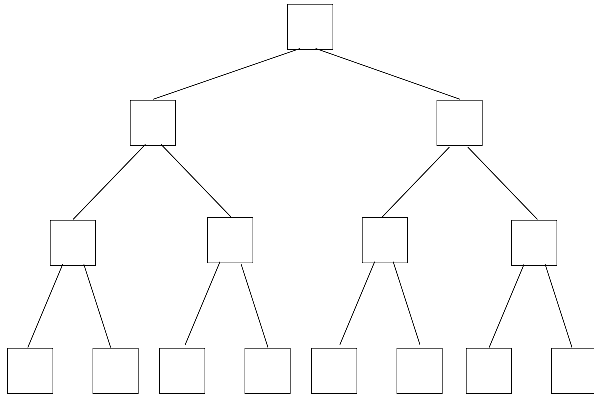

Figure 4.1: A regular binary tree.

it has an advantage in terms of the small diameter. So, algorithms, in general, are expected to run faster on a tree than on an array. Many times the small bisection width may be responsible for lower bounds of algorithms. This is not always the case, however, and one has to be careful.

4.2 Algorithms

4.2.1 Prefix sum

Computing the prefix sum is a straightforward algorithm that can be easily adapted from the PRAM algorithm. The algorithm executes in two phases:

- an upwards phase, in which a node receives values from its children, computes the sum and sends the result to its parent, and

- a downwards phase in which a node sends to the left child the value that the upward phase sent to its parent (i.e., the sum of the values received from its children; and to the right child the sum of the value received from its parent plus the value of its left child

Figure 4.2 shows the two phases of the *prescan* algorithm.

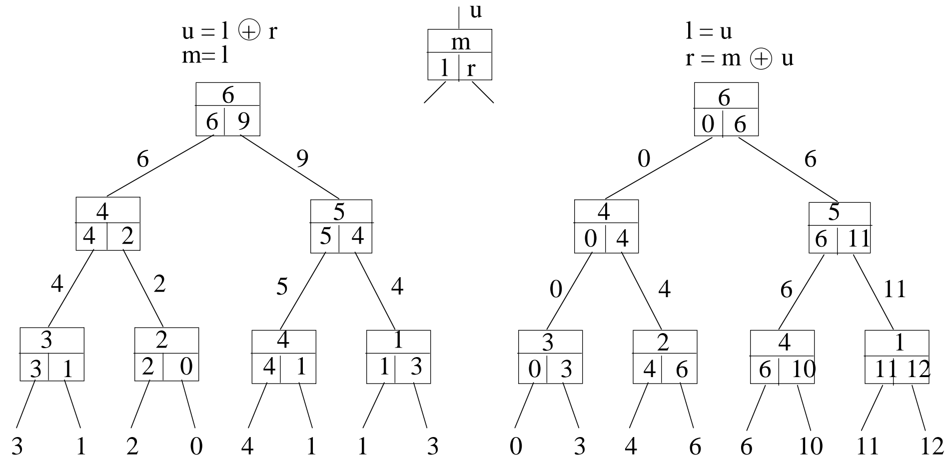

Figure 4.2: Upwards phase (left) and downwards phase (right). The variables needed are shown in the middle of the figure.

Pseudocode

We will describe the two phases of the prescan algorithm, the upward phase and the downward phase.

Upward phase.

for $i = 1$ **to** $\lceil \lg(n+1) \rceil$ **do**
    for each node at height i **in parallel do**
        receive value form left child into l
        receive value form right child into r
        let $m \leftarrow l$
        let $u \leftarrow l \oplus r$
        if current node is not the root **then**
            send u to parent
        end if
    end for
end for

Downward phase.

```
for i = ⌈lg n⌉ downto 1 do
    for each node at height i in parallel do
        if current node is not the root then
            receive from parent into u
        end if
        let l ← u
        let r ← m ⊕ u
        send l to left child
        send r to right child
    end for
end for
```

How do you declare a tree on the MasPar? One way would be to use the router. However, for regular algorithms, one can use the whole machine as a 1-D array.

4.3 Related networks

4.3.1 Fat trees

The CM-5 has two networks; a data network and a control network. The control network is a tree, like the ones we just examined. The data network is a *fat tree*, which is like a tree of meshes (Figure 4.3). (This is not exactly right, it is actually a tree of butterflies, which we will examine in a later chapter.) The characteristic of the fat tree is that its branches are getting fatter as you get closer to the root, so the bisection width increases allowing for more communication. This, along with the fact that are usually drawn with the root at the bottom of the tree make them look closer to real-life trees.

The advantages of the fat trees are:

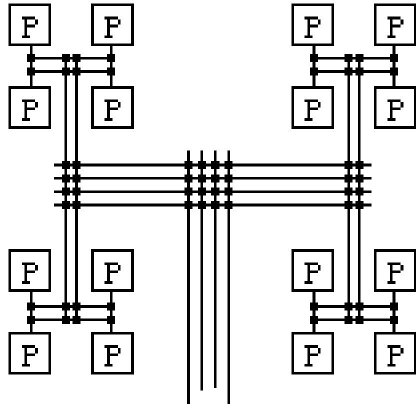

Figure 4.3: A tree of meshes. The processors are at the leaves, and communicate via meshes.

1. Like a mesh, it occupies little area: n nodes occupy area $O(n)$.
2. Like a mesh, it has lots of wires: n nodes have $O(\sqrt{n})$ wires.
3. Unlike a mesh, it has a small diameter: $O(\log n)$.
4. Unlike a mesh, it is *area-universal*, i.e., it can efficiently simulate any network that occupies $O(n)$ area.

Given all these nice features, it is not suprising that it was chosen as the network for the expensive CM-5 parallel computer.

Leiserson's talk

Chapter 5

Mesh of Trees (MOTs)

5.1 Description and properties

In previous chapters we examined the mesh, a simple network with desirable properties, i.e., large bisection width and low vertex degrees, but also with a drawback: large diameter. we also examined the full binary trees, that have nice recursive structure, low vertex degree and the desired low diameter, but they also have a small bisection width.

So, the idea is to combine these two networks hoping that we can get the best of both worlds. In fact, the mesh of trees accomplishes that. Imagine of "planting" trees on each column and row of a mesh. This would have the following effects: (Figure 5.1)

- $2n(n-1) + n^2 = 3n^2 - 2n$ processors
- $2n$ bandwidth (i.e., the roots of the trees)
- $4 \log n$ diameter
- n bisection width
- node degree 2 or 3 and

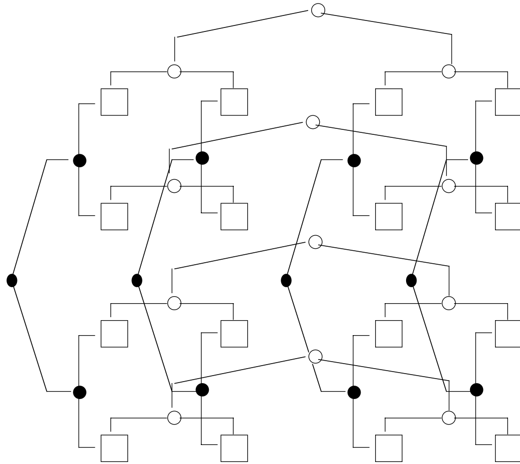

Figure 5.1: A mesh of trees: $M_{2,4}$.

- $2n(2n-2)$ wires, because each tree connects $2n-1$ nodes with $(2n-1)-1$ wires.
- a recursive structure: removing column and row roots you get 4 copies of MOTs, each one-fourth in size of the originals.

These characteristics make it comparable to meshes, but with a much smaller diameter, which makes it more powerful. In fact, MOTs are so nice that is rather surprising that no manufacturer has already built one.

5.2 Algorithms

5.2.1 The Ideal Computer

To see the power of MOTs, we will examine how they can be used the problem of creating what we called an "ideal computer". So, let's assume that we have n processors and n memories connected. We would like to be able to send n messages with minimal delay.

As we mentioned before, the best thing would be to connect them in a complete bipartite graph $K_{n,n}$ as in Figure 5.2, but this is rather expensive.

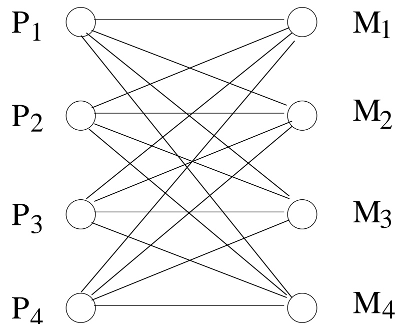

Figure 5.2: A bipartite graph: $K_{4,4}$.

It turn out that MOTs can simulate $K_{n,n}$ with a delay of only $2 \log n$. We have to view them, however, as in Figure 5.3.

As you can see, the delay is due to traversing the tree nodes. Since $K_{n,n}$ behaves as a PRAM machine, we have the following:

Theorem 3 *Any CRCW PRAM algorithm can be simulated on a MOT with a slowdown factor of* $2 \log n$.

□

5.2.2 Sorting

Let us apply the last theorem to simulate the CRCW PRAM sorting algorithm that uses $\log n$ time and n^2 processors. Recall that the algorithm is based on comparing every pair of the input numbers, counting how many other numbers are, say, larger from each number, and then placing each number in the array location that reflects the result of the count. On a MOT, this algorithm is implemented as follows:

1. Input n distinct numbers at the n row- and n column-roots of the MOT (takes constant time).

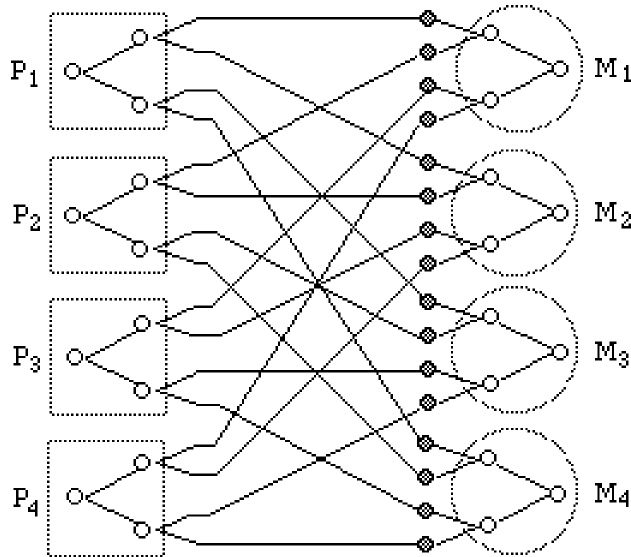

Figure 5.3: $M_{2,4}$ mesh of trees simulating $K_{4,4}$. The dark gray are the mesh processors.

2. Broadcast the numbers from the roots to the mesh processors (takes logarithmic time).

3. Each mesh processor compares the pair of numbers it received. It stores 1 in some variable *wins* if the number that came from the row root was larger than the number that came form the column root; it stores 0 otherwise (takes constant time).

4. *wins* are being sent to the row roots. They are being counted on their way up. Each row root processor receives the total *wins*, say i; therefore its input number is the $(i+1)$-st larger. This also takes $O(\log n)$ steps.

5. Finally, each row root processor sends the input number to the $(i+1)$-st column root processor in $O(2\log n)$ steps. After that, the column roots contain the sorted array.

The whole algorithm takes $4 \log n + O(1)$ time using $3n^2 - 2n$ processors. Given the logarithmic running time, we see that it is a fast (i.e., in NC) algorithm, but not an efficient one, because it performs $O(n^2 \log n)$ work versus $O(n \log n)$ of the best sequential algorithm.

Chapter 6

Hypercubic networks

6.1 The r-dimensional hypercube

6.1.1 Description and Properties

The hypercubic graphs are very versatile networks and have been used extensively to interconnect the processors of several parallel computers. The reason that manufacturers have done that is that the hypercube is *universal*, i.e., it can easily emulate all the networks we have discussed so far. We will see later that linear arrays, trees, meshes, n-dimensional arrays, can all be embedded on the hypercube, therefore the algorithms that have been developed for these networks can work on a hypercube in time $O(\log n)$. Also, any PRAM algorithm could be adapted to run on a hypercube with a delay of $O(\log N)$.

A hypercube H_r of *dimension* r has $n = 2^r$ nodes, each of which has $r = \log n$ edges. We can number each node with a binary string of length r. Two nodes are linked by an edge if and only if they differ by one bit. This edge has dimension k if two nodes differ on the kth bit.

In general, it is easier to describe a hypercube than to draw it. Figure 6.1 shows hypercubes with $r = 0, 1, 2, 3, 4$. From this construction, we observe that hypercubes have a simple recursive structure, i.e., the $r+1$-dimensional

Stack "Networks"

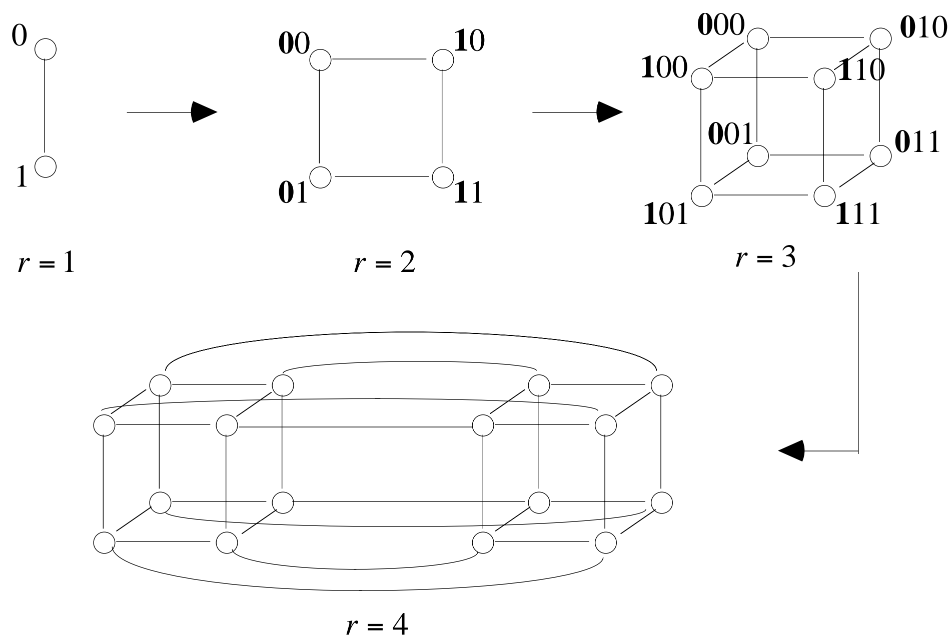

Figure 6.1: Hypercubes of dimension 1, 2, 3 and 4. The bold bits show the numbering at each stage.

hypercube can be constructed from two r-dimensional hypercubes by connecting the corresponding nodes. Similarly, by ignoring the first bit of the nodes of H_r, you get two H_{r-1} hypercubes. Because of the recursive structure property, inductive proofs are useful (and common) for hypercubes. We will see some of them later on.

As we mentioned before (page 37), an r-dimensional hypercube is an r-dimensional 2-sided array. So, the bisection width is 2^{r-1}. Consider the distance from node 00...00 to 11...11 you see that the diameter is r.

To recap, the hypercube H_r has the following characteristics:

- diameter $r = \log n$,
- vertex degree $r = \log n$ — small but not fixed.
- bisection width $2^{r-1} = \frac{n}{2}$,
- symmetry,

- a nice recursive structure.

6.1.2 Routing on the Hypercube

The numbering of the hypercube nodes may seem as random, but that is not so. An important fact about it is the following:

Lemma 4 *Every pair of neighboring nodes differ exactly in one bit.*

Proof. (sketch) We can prove it by induction on the number of nodes. We know it holds in the base case. Now, by construction, when we connect two hypercubes of dimension $r-1$ to get one of dimension r, we connect the nodes with identical names and we add a 0 in front of the id of the one and a 1 in front of the id of the other (the id's are in binary). All the previous connection remain the same. □

This provides a great routing scheme: To send data from node x to node y, we compare the binary representation of the x and y id's. In each routing step, we move so that we "fix" one of the bits; at most $\log n$ moves are required.

For example, to send data from node 1011 to node 0110, a possible path is

$$1011 \rightarrow 0011 \rightarrow 0111 \rightarrow 0110$$

It is interesting that there are many ways to send the data, in fact, if the two nodes differ in k bits, there are $k!$ different paths from x to y.

6.2 Embeddings

Hypercubes are powerful because they contain many simple networks as subgraphs. So, algorithms designed for the simple nets can be implemented on hypercubes without much modification. In particular, arrays of any size and dimension can be embedded in hypercubes.

6.2.1 Linear arrays and the Gray code

A Hamiltonian cycle of a graph G is a tour that visits every node of G exactly once. An r-bit *Gray code* is an ordering of all r-bit numbers in such a way that consecutive numbers differ in exactly one position.

Example 1 *The 3-bit Gray code is: 000, 001, 011, 010, 110, 111, 101, 100.*

The Gray code of a hypercube is the sequence of nodes traversed by a Hamiltonian cycle of a hypercube. We can inductively prove the following:

Lemma 5 *The r-bit Gray code exists for every r-dimensional hypercube.*

Proof For $r = 1$ the Gray code is $v_1 = 0, v_2 = 1$. Suppose for $r = k$, the Gray node is $v_1, v_2, \ldots, v_N$, where $N = 2^k$ and v_i is a binary string of length k for $1 \leq i \leq N$. Then the Gray node for the $k+1$-dimensional hypercube is $1|v_1, 1|v_2, \ldots, 1|v_N, 0|v_N, 0|v_{N-1}, ..., 0|v_1$, where "|" means concatenation. This is because $1|v_N$ and $0|v_N$, $1|v_i$ and $1|v_{i+1}$, $0|v_i$ and $0|v_{i+1}$ differ exactly by 1 bit, since v_i and v_{i+1} differ exactly by 1 bit for $1 \leq i \leq N-1$. □

The above observation of the Gray code proves that all the N-node linear arrays $a[1..N]$ can be embedded in the $2^{\lceil \log N \rceil}$-dimensional hypercube by mapping $a[i]$ to the Gray code v_i.

6.2.2 * Multidimensional arrays

It can be shown that

Theorem 4 *An $N_1 \times N_2 \times \ldots \times N_k$ array can be contained in an N-node hypercube, where $N = 2^{\lceil \log N_1 \rceil + \lceil \log N_2 \rceil + \ldots + \lceil \log N_k \rceil}$.*

Theorem 4 is a direct consequence of the containment of linear arrays and Lemma 6 of graph theory:

Lemma 6 *If $G = G_1 \otimes G_2 \otimes \ldots \otimes G_k$ and $G' = G'_1 \otimes G'_2 \otimes \ldots \otimes G'_k$, and each G_i is a subgraph of G'_i, then G is a subgraph of G'.*

$G(V, E) = G_1 \otimes G_2 \otimes \ldots \otimes G_k$ stands for the *cross-product* of graphs $G_1(V, E)$, $G_1(V, E)$, $\ldots G_k(V, E)$, where $V(V_i)$ and $E(E_i)$ are vertex sets and edge sets, respectively, and the following conditions are satisfied:

$$V = \{(v_1, v_2, \ldots v_k) \text{ where } v_i \in V_i \text{ for } 1 \leq i \leq k\}$$

$$E = \{\{(u_1, u_2, \ldots u_k), (v_1, v_2, \ldots v_k)\} \text{ where}$$
$$\text{there exists a } j \text{ such that } (u_j, v_j) \in E_j \text{ and } u_i = v_i \forall i \neq j\}.$$

Example 2 *The cross-product of linear arrays $N_1, N_2, \ldots, N_k$ is the k- dimensional array $N_1 \times N_2 \times \ldots \times N_k$, and the cross-product of the hypercubes $H_1, H_2, \ldots H_k$ of dimensions $r_1, r_2, \ldots r_k$ respectively is the $(r_1+r_2+\ldots+r_k)$-dimensional hypercube H.*

Since linear array N_i is contained in the $2^{\lceil \log N_i \rceil}$ -dimensional hypercube, $N_1 \times N_2 \times \ldots \times N_k$ is contained in the $(\lceil \log N_1 \rceil + \lceil \log N_2 \rceil + \ldots + \lceil \log N_k \rceil)$-dimensional hypercube.

6.3 Related networks

Although the hypercube is powerful in computing, it has weaknesses in practice. One of them is that the node degree is not fixed, but grows with the size, so machines are not scalable; you can not add wires on an existing chip on demand. To overcome this disadvantage, networks such as butterfly and cube-connected cycles have been designed. We will study the basic architecture of these structures.

6.3.1 The butterfly

The r-dimensional butterfly has $2^r(r+1)$ nodes, each of which named $< w, l >$, where w is the row number (where $0 \leq w \leq 2^r - 1$, expressed in binary form) and l is the level number (where $0 \leq l \leq r$). Nodes $< w, l >$

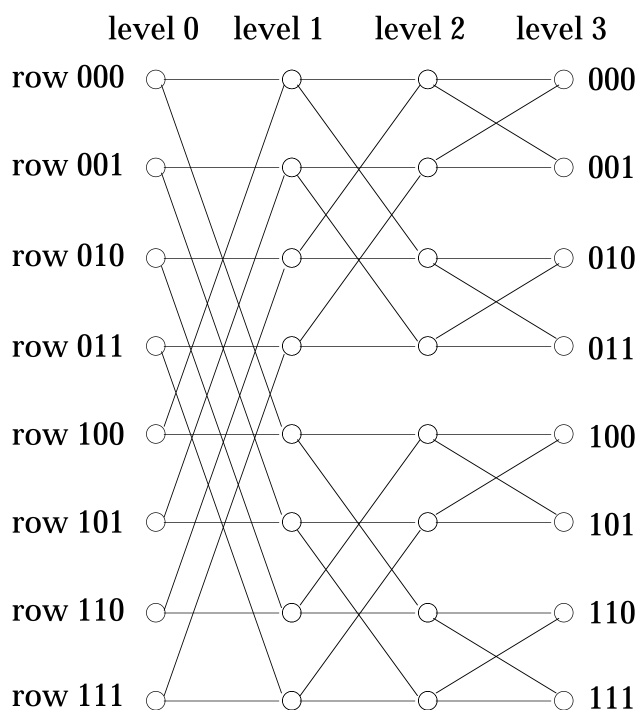

Figure 6.2: The 3-dimensional butterfly

and $< w', l' >$ are linked either by a straight edge (if $l' = l+1, w' = w$), or by a cross edge (if $l' = l+1$, and w' and w differ on the l'-th most significant bit). Therefore, the r-dimensional butterfly has $r \cdot 2^{r+1}$ edges. Figure 6.2 shows the 3-dimensional butterfly.

Routing on the butterfly is similiar to routing on the hypercube, i.e. it can be done by "bit fixing": To route from node x to node y, we look at the binary representation of their id's. Starting from the most significant bit, we choose the straight edge if the two bits are the same, and the cross edge if the two bits differ. For example, to go from node 101 to node 011, we follow two cross edges, then the straight one.

Leighton's talk

Hypercubes and butterflies are similar in structure even though they look different. In fact, for many years it was believed that they were two completely different networks, but recently it was proved that they are essentially the same network, part of a large family called the *Cosmic Cube*.

The characteristics of the burrterfly are:

- diameter $2 \log n = 2r$
- node degree 2 or 4
- bisection width $2^r = \Theta(\frac{n}{\log n})$
- recursive structure

Let us examine the recursive structure of the butterfly. Obviously, if the nodes of level 0 from the r-dimensional butterfly are removed, the first 2^{r-1} rows and the last 2^{r-1} rows each form an $(r-1)$ -dimensional butterfly. Deleting the first digit of the row numbers, we obtain the new row numbers for the $(r-1)$-dimensional butterflies. What is not so obvious is that the removal of the level r nodes also yields two $(r-1)$-dimensional butterflies. In this case the odd number rows and even number rows each form an $(r-1)$-dimensional butterfly. Deleting the last digit of the row numbers we obtain the new row numbers for the $(r-1)$-dimensional butterflies.

We can obtain the r-dimensional hypercube from the r-dimensional butterfly by removing straight edges, merging all the nodes in one row and then deleting the extra copy of the cross edges. This allows us to prove that 2^r edges must be removed from the r-dimensional butterfly in order to partition the rows into two equal-size sets, because partitioning rows in butterflies is the same as bisecting hypercubes. We showed before that the bisection width of the r-dimensional hypercube is 2^{r-1}. Since each edge in the hypercube corresponds to two cross edges in the butterfly, $2 \times 2^{r-1} = 2^r$ edges in the butterfly must be removed (otherwise we would have a lower bisection width for the hypercube).

*** Properties of the Butterfly**

We will now investigate the two interesting properties of every network, the diameter and the bisection width.

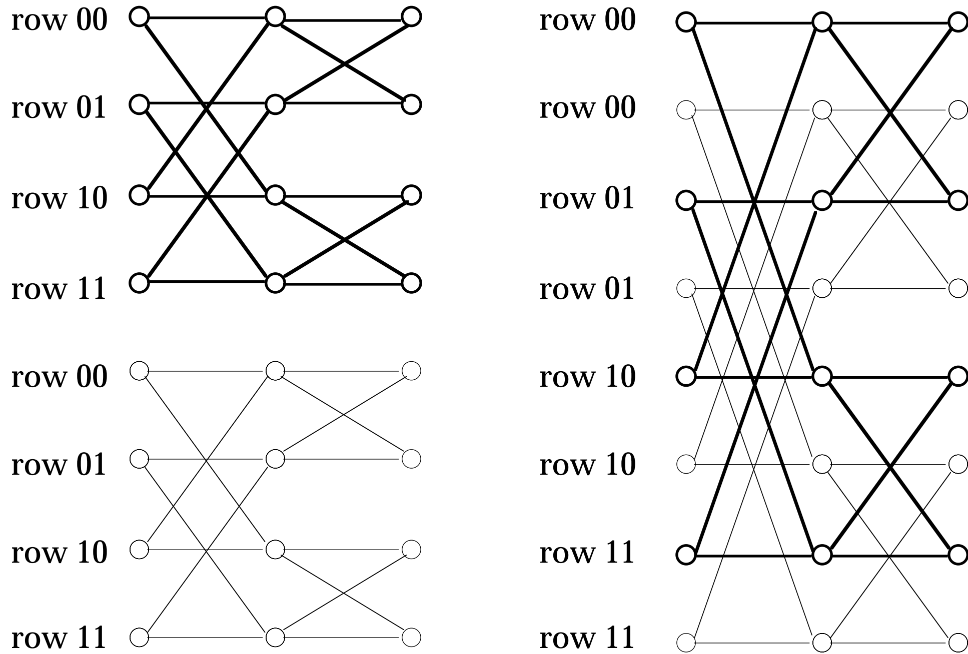

Figure 6.3: The recursive structure of the 3-dimensional butterfly

Lemma 7 *The diameter of the r-dimensional butterfly is $\Theta(r)$.*

Proof: There is a unique path from node $< w_1, w_2, \ldots, w_r, 0 >$ to $< w_1', w_2', \ldots, w_r', r >$ defined as $< w_1, w_2, \ldots, w_r, 0 > \rightarrow < w_1', w_2, \ldots, w_r, 1 > \rightarrow < w_1', w_2', \ldots, w_r, 2 > \rightarrow \ldots \rightarrow < w_1', w_2', \ldots, w_r', r >$ where the transition $< w_1', w_2', \ldots, w_i', w_{i+1}, \ldots w_r, r > \rightarrow < w_1', w_2', \ldots, < w_i', w_{i+1}', \ldots w_r, r >$ follows a straight edge if $w_{i+1} = w_{i+1}'$, and follows a cross edges otherwise. This shows that the diameter of the r-dimensional butterfly is at least r. In general, the path from an arbitrary node $< w_1, w_2, \ldots, w_r, l >$ to $< w_1', w_2', \ldots, w_r', l' >$ ($< w_1, w_2, \ldots, w_r, l >$ is smaller than $< w_1', w_2', \ldots, w_r', l' >$ by the canonical order) is: $< w_1, w_2, \ldots, w_r, l > \rightarrow < w_1, w_2, \ldots, w_r, 0 > \rightarrow < w_1', w_2', \ldots, w_r', r > \rightarrow < w_1', w_2', \ldots, w_r', l' >$. The moves $< w_1, w_2, \ldots, w_r, l > \rightarrow < w_1, w_2, \ldots, w_r, 0 >$ and $< w_1', w_2', \ldots, w_r', r > \rightarrow < w_1', w_2', \ldots, w_r', l' >$ are achieved by following the straight edges. If $< w_1, w_2, \ldots, w_r, l >$ is larger than $< w_1', w_2', \ldots, w_r', l' >$, the path is $< w_1, w_2, \ldots, w_r, l > \rightarrow$

$< w_1, w_2, \ldots, w_r, r > \rightarrow < w_1', w_2', \ldots, w_r', 0 > \rightarrow < w_1', w_2', \ldots, w_r', l' >$. This shows that the diameter is at most $2r$. Thus, the diameter of the r-dimensional butterfly is $\Theta(r)$. □

Lemma 8 *The bisection width of the N-node r-dimensional butterfly is $\Theta(N/\log N)$.*

Proof: We will use the similar embedding technique as before to prove this lemma. We embed a complete directed graph G in the butterfly B by embedding the edge from $< w_1, w_2, \ldots, w_r, l >$ to $< w_1', w_2', \ldots, w_r', l' >$ of G by the path in B described in the previous paragraph.

We first show that a cross edge $(< w, l >, < w', l+1 >)$ of B is contained in at most $\Theta(r^2 \cdot 2^r)$ paths in G. A path going through edge $(< w, l >, < w', l+1 >))$ looks like $< w_0, l_0 > \rightarrow < w_0, 0 > \rightarrow < w, l > \rightarrow < w', l+1 > \rightarrow < w_1, r > \rightarrow < w_1, l_1 >$ where $l_0 \leq l_1$. We have 2^l choices for w_0, 2^{r-l-1} choices for w_1, and $\sum_{i=1}^{r+1}(i) = \frac{1}{2}(r+1)(r+2)$ choices for l_0 and l_1. Since there are 2 directions, there are at most $(r+1)(r+2)2^{r-1} = \Theta(r^2 \cdot 2^r)$ paths going through the cross edge connecting node $< w, l >$ and $< w', l+1 >$ of B. Similarly there are $\Theta(r^2 \cdot 2^r)$ paths going through a straight edge.

To divide G into 2 halves each with $(r+1) \cdot 2^{r-1}$ nodes, $2 \cdot [(r+1) \cdot 2^{r-1}]^2 = \Theta(r^2 \cdot 2^r)$ paths have to be removed. Since each edge of B is contained in $\Theta(r^2 \cdot 2^r)$ paths of G, at least $\frac{\Theta(r \cdot 2^{2r})}{\Theta(r \cdot 2^{2r})} = \Theta(2^r)$ edges of B have to be removed. Thus, the bisection width of the r-dimensional butterfly is $\Omega(2^e)$.

It is easy to check that if we remove the cross edges connecting level 0 and level 1 of the butterfly, we bisect the network. Thus we have proved that the bisection width of the N-node r-dimensional (where $N = 2^r \cdot (r+1)$) butterfly is $\Theta(N/\log N)$. □

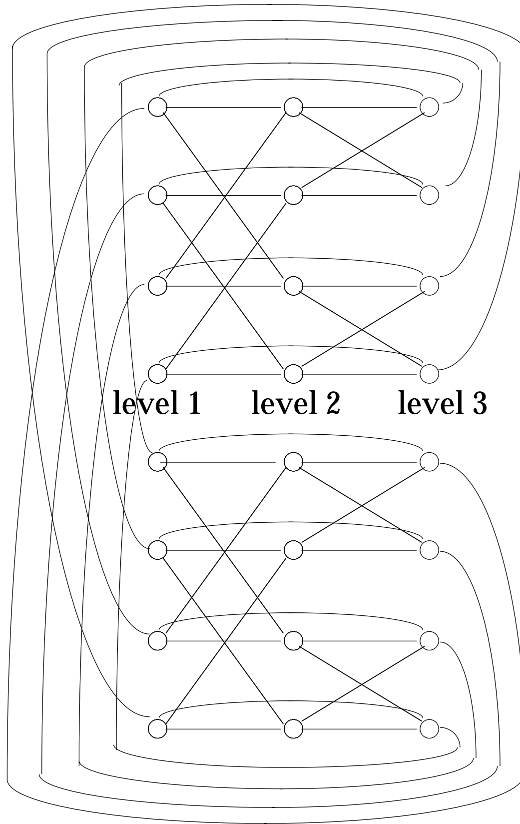

Figure 6.4: The 3-dimensional wrapped butterfly

6.3.2 The wrapped butterfly and cube-connected cycles

The r-dimensional wrapped butterfly (Figure 6.4 is obtained by merging the level 0 and level r nodes of the r-dimensional ordinary butterfly. Thus r-dimensional wrapped butterfly consists of $2^r \cdot r$ nodes, each of which has degree 4. Formally, the link between node $< w, l >$ and $< w^{'}, l^{'} >$ is:

1. a straight edge if $l = l^{'} + 1 \bmod r$ and $w = w^{'}$, or
2. a cross edge if $l = l^{'} + 1 \bmod r$ and $w, w^{'}$ differ in the $l^{'}$-bit.

The r-dimensional cube-connected cycle (CCC) is obtained from the r-dimensional hypercube by replacing each node of the hypercube with a cycle of r nodes (Figure 6.5). So the r-dimensional CCC has $r \cdot 2^r$ nodes each with node degree 3. Formally, the link between node $< w, l >$ and $< w^{'}, l^{'} >$ is:

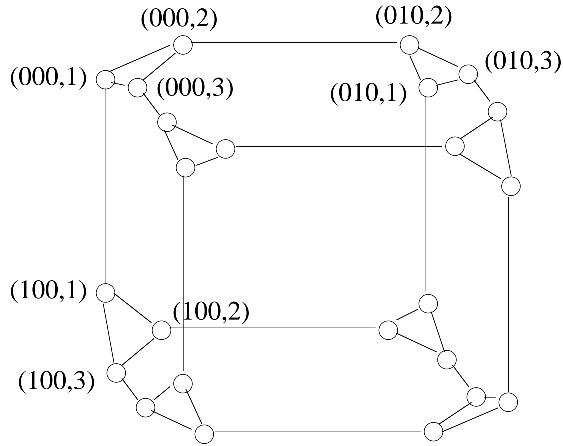

Figure 6.5: The 3-dimensional cube-connected cycles

1. a cycle edge if $l = l' \pm 1 \bmod r$ and $w = w'$, or
2. a hypercube edge if $l = l'$ and w, w' differ in the l-bit.

The CCC has the following characteristics:

- nodes $N = r \cdot 2^r = n \log n$
- diameter $2r = 2 \log n$
- node degree 3
- bisection width $n/2$
- recursive structure
- can simulate any hypercube algorithm with a slowdown of $\log n$.

It can be shown that the CCC and wraparound butterfly can be embedded one-to-one in each other with dilation 2. The wraparound butterfly can also be mapped onto the butterfly with dilation 2.

So we can treat the butterfly, wraparound butterfly and CCC identically. It is true that

Theorem 5 *The butterfly, wraparound butterfly and cube-connected cycles can simulate most simple networks without slowdown.*

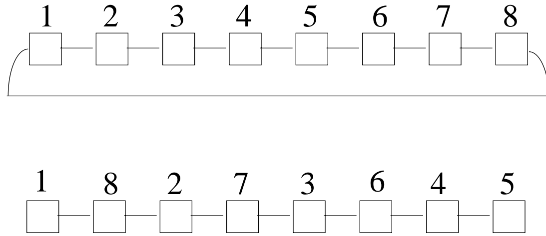

Figure 6.6: Embed the 8-element linear array one-to-one to the 8-element ring with dilation 2

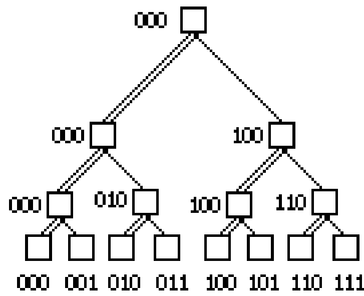

Figure 6.7: Processor assignment for canonical tree computation. Double edges denote work by the same processor.

This theorem is important as it proves the existence of structures with bounded node degrees and comparable computational power to hypercubes.

6.3.3 The Shuffle-Exchange Network

The last two networks succeeded in simulating the hypercube while containing only nodes with a small fixed degree. However, the number of processors had to increase by a factor of $\log n$ in order to do so. Could we relax this requirement? It turns out that most of the algorithms that one wants to run on these networks are *normal* — algorithms that work in phases such that in each phase separate sets of processors are active.

The prefix sum is an example of a normal algorithm. One does not really need to have 1 processor per node in the tree, because in each phase of the algorithm only one level of processors needs to work: in the beginning only

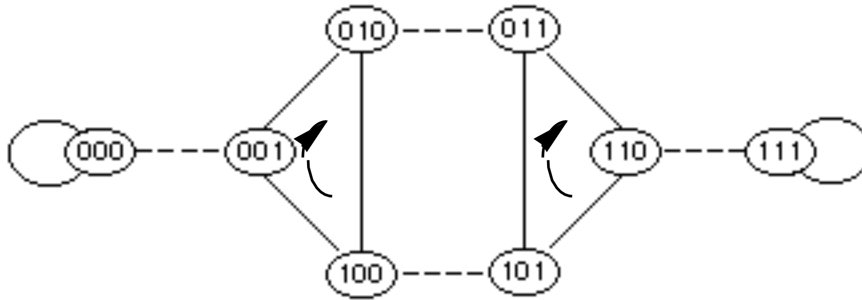

Figure 6.8: The shuffle-exchange network.

the leaves, in the second phase the parents of the leaves, and so on. So, we can "recycle" processors by assigning the processor in each leaf to play the role of all the processors in the left branches of the tree. Figure 6.7 shows an example of processor assignment for this problem.

The question that naturally arises is: can we simulate normal algorithm with a "cheaper" network than a hypercube? The answer is yes; the shuffle-exchange network we describe next can achieve that.

Before we show a picture of it, let us describe how it is created from the hypercube. Let's assume we have a hypercube (Figure 6.9) which only has the wires of dimension i, for some i: $1 \leq i \leq \log n$. If you rotate this hypercube about two opposite points, it will appear as if only wires of dimension $i+1$ (or dimension $i-1$, depending on the rotation direction) exist.

Following this construction, a shuffle-exchange network is a hypercube that contains edges of only one dimension, along with edges that simulate that rotation just described. (Figure 6.8).

A normal algorithm can now be simulated on a shuffle-exchange net with slowdown of 2 as follows:

1. Do the calculation; then, send data along the exchange i-wires.

2. Do the calculation; then, send data along shuffle wires, so dimensions

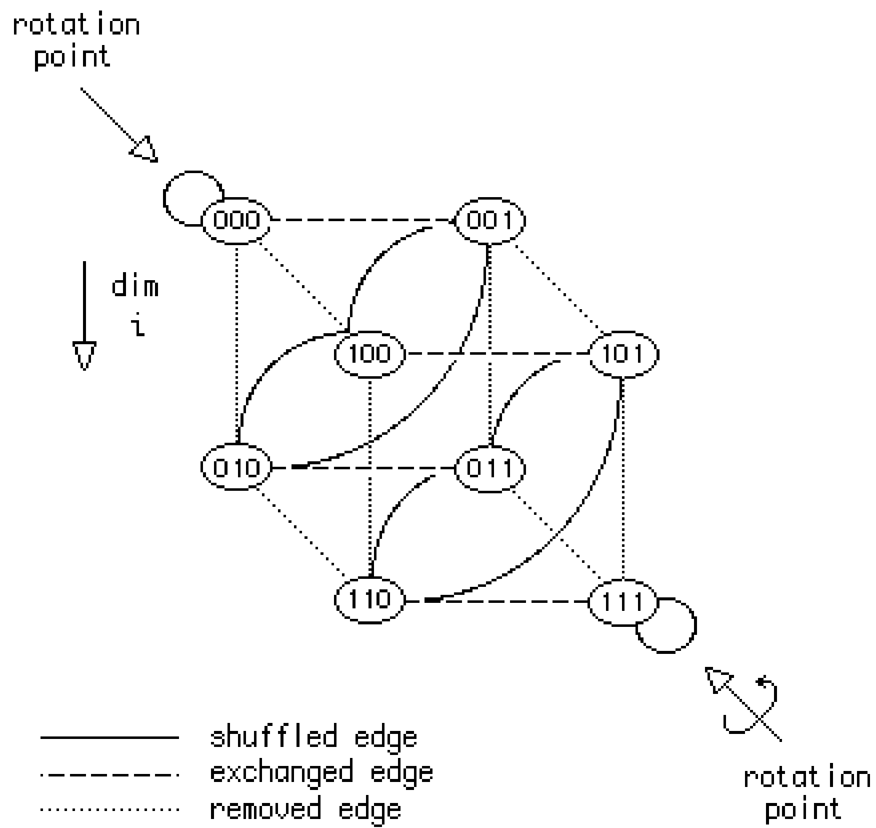

Figure 6.9: The way you get the shuffle-exchange network from the hypercube by rotation. Compare it with its canonical drawing in the previous figure.

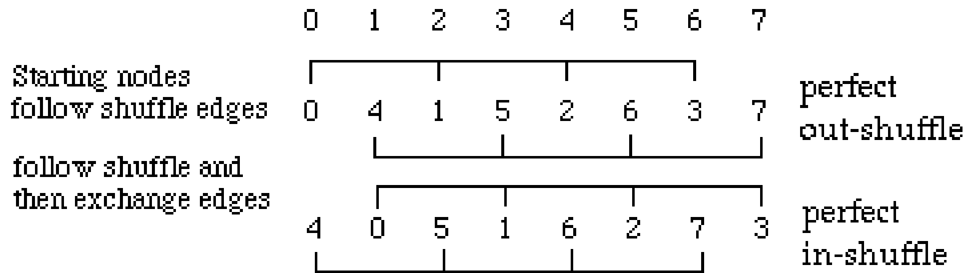

Figure 6.10: In-shuffle and out-shuffle.

$i+1$ or $i-1$ can be used in the next phase.

The term *shuffle* comes from the fact that sending data along this wire simulates a perfect in-shuffle of a deck of n cards. The term *exchange* comes from the fact that crossing such an edge gets you to a node that has the name of the starting node whose last bit is swapped.

As a matter of fact, on the shuffle exchange network you can simulate both an in-shuffle and an out-shuffle as Figure 6.10 shows. This fact explains a nice card trick: with $\lg n$ shuffles you can bring any card to any position of a deck of n cards.

So, the characteristic of the shuffle-exchange network are:

- diameter $2 \log n$,
- vertex degree 3,
- $n = 2^r$ nodes and $3 \cdot 2^{r-1} = 1.5n$ edges.

6.3.4 The de Bruijn network

The r-dimension de Bruijn graph is derived from the $r+1$-dimension shuffle-exchange graph by the following algorithm:

1. Collapse nodes connected by exchange edges into a new node; name this node with the common first $r-1$ bits of the collapsed nodes.

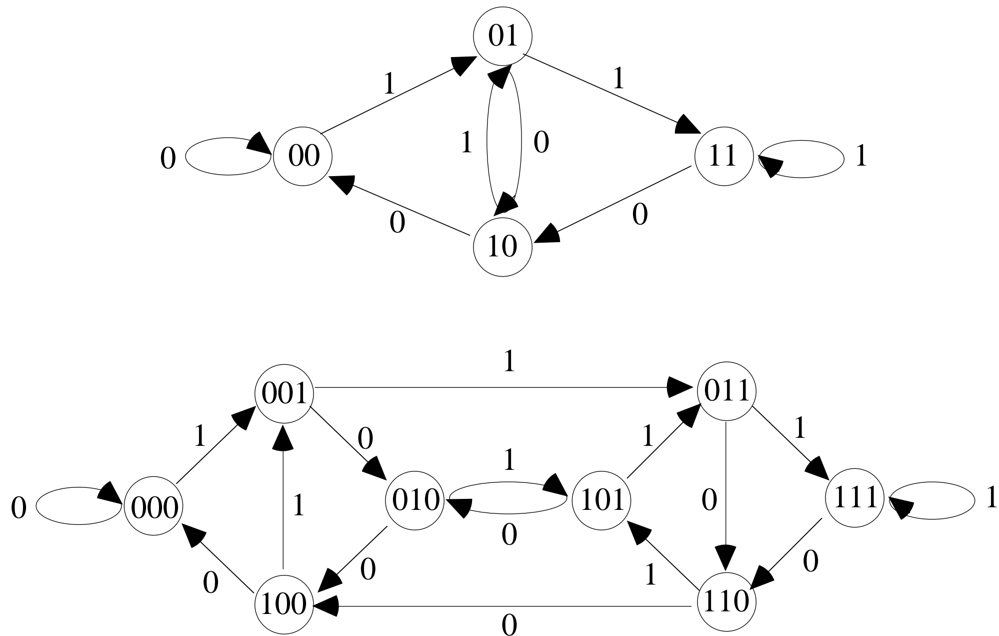

Figure 6.11: The 2-dimensional and 3-dimensional de Bruijn graphs

2. Label the shuffle edges (v, w) by the last bit of the target vertex w.

The characteristics of the de Bruijn graph are

- node in-degree 2 and out-degree 2
- diameter $\log n$
- 2^r nodes and 2^{r+1} directed edges
- bisection width $\Theta(N/\log N)$
- recursive structure

The de Bruijn graph has several interesting properties as described by the following lemmas:

Lemma 9 *The de Bruijn graph has an Eulerian tour which can be found following the edges labeled by the bits in de Bruijn sequence.*

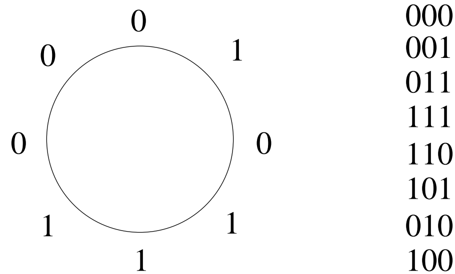

Figure 6.12: The de Bruijn sequence for $N = 8$.

A *de Bruijn sequence* is an ordering of N bits, where every $\log N$ sequence appears exactly once in the cyclic ordering.

Example 3 *The sequence 00011101 is a de Bruijn sequence, since* $N = 8$*,* $\log N = 3$ *and every 3 consecutive digits in the circular ordering appears exactly once. (Figure 6.12)*

Note that we can start from any point of the 2-dimensional de Bruijn graph and make an Euler tour by following the de Bruijn sequence: start at some random point of the de Bruijn sequence, and follow the edges whose labels appear on the sequence. Because of the in-degree and out-degree property, there is an Euler tour. Because of the way the edges are labeled and the property of the de Bruijn sequence, the sequence follows the tour. □

Lemma 10 *The de Bruijn graph has a Hamiltonian circuit. The concatenation of the edge labels visited by the Hamiltonian circuit produces de Bruijn sequence.*

□

Finally, the recursive structure of the de Bruihn graph can be seen by the following construction.

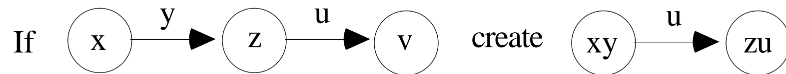

Figure 6.13: Create edges for a de Bruijn graph.

- For each node labeled x and each edge out of x labeled y create a new vertex called xy (by concatenating labels).
- Add edges as in Figure 6.13.

Example 4 *See how the 3-dimensional graph can be derived from the 2-dimensional graph.*

A neat card trick based on the uniqueness of $\log N$ bits of de Bruijn sequence can be done where one can guess $\log N$ cards given to people by just asking them to raise hands if they hold a red card.

6.4 Bibliographic Notes

Part of the materials in this Chapter are from Leighton's book [**?**] and its exercises, and from Y.L. Zhang's honors thesis.