The simple model of a dual-processor configuration is suggestive of behavior we can expect from multiprocessor systems that require global communication. We observe that if $f = 0$, execution speed is more than twice that of the uniprocessor illustrated in Fig. 8.1. Just as in the pipeline, doubling the number of processors contributes a factor of two, but additional speed is achieved because each processor addresses a smaller memory.

The model also illustrates the importance of locality in the use each processor makes of its memory. If f is allowed to grow too large, the factor of two contributed by two processors is erased by interference between the processors when accessing the common memory.

Perhaps the most important parameter is d, which is determined by our ability to adapt algorithms to multiprocessor configurations. Some applications seem to decompose nicely for execution on concurrent hardware, and some offer difficulties. In human organizations we have become resigned to *always* attacking large problems in a concurrent way. We will, no doubt, have to do the same with computer programs.

8.2.2 Summary

The schemes we have illustrated that reduce communication costs and try to exploit concurrency can be combined in various ways in computer structures. The table below summarizes the speedup effect that these techniques offer, as derived from our crude models (n denotes the number of processors used):

Technique	Typical speedup factor
Memory hierarchy	10
Pipelining	
instruction overlap	2
special-purpose	n
Multiprocessors	$< n$

The processor-memory structures and algorithms presented in the remainder of this chapter all attempt *to use as many processors as can be kept simultaneously productive and to locate them as close as possible to the data they require*. These are the considerations exhibited by our simple models of memory hierarchies, pipelines and multiprocessors. The examples presented here by no means exhaust the topic of concurrent computation; the interested reader will find literatures on computer architecture,[2,4] parallel processors and processing,[3,5,6,7] performance evaluation,[2] and algorithm design.[8,9,10,11,12]

8.3 ALGORITHMS FOR VLSI PROCESSOR ARRAYS*

8.3.1 Introduction

"And the smooth stream in smoother numbers flows."
Alexander Pope

The developments in microelectronics have revolutionized computer design. Integrated circuit technology has increased the number and complexity of components that can fit on a chip or a printed-circuit board. Component density has been doubling every one-to-two years, and already a multiplier can fit on a very large scale integrated (VLSI) circuit chip. As a result, the new technology makes it feasible to build low-cost, special-purpose, peripheral devices to rapidly solve sophisticated problems. Reflecting the changing technology, this section proposes new multiprocessor structures and parallel algorithms for processing some basic matrix computations.

We are interested in high-performance parallel algorithms that can be implemented directly on low-cost hardware devices. By performance, we are not referring to the traditional operation counts that characterize classical analyses of algorithms, but rather, the throughput obtainable when a special-purpose peripheral device is attached to a general-purpose host computer. This implies that time spent in I/O, control, and data movement as well as arithmetics must all be considered. VLSI offers excellent opportunities for inexpensive implementation of high-performance devices. Thus, in this section the cost of a device will be determined by the expense of a VLSI implementation. "Fit the job to the bargain components" (Blakeslee, p. 4).[13]

VLSI technology has made one thing clear. Simple and regular interconnections lead to cheap implementations and high densities, and high density implies both high performance and low overhead for support components. (Sutherland and Mead[1] contains a discussion of the importance of having simple and regular geometries for data paths.) For these reasons, we are interested in designing parallel algorithms that have simple and regular data flows. We are also interested in using pipelining as a general method for implementing these algorithms in hardware. By pipelining, processing may proceed concurrently with input and output, and consequently overall execution time is minimized. Pipelining plus multiprocessing at each stage of a pipeline should lead to the best-possible performance. In the following, we demonstrate some simple and regular VLSI processor arrays that are capable of pipelining matrix computations with optimal speed-up.

---

*Contributed by H. T. Kung and Charles E. Leiserson, Department of Computer Science, Carnegie-Mellon University. The first version of Section 8.3, including results reported in Sections 8.3.3 thru 8.3.6 of the present version, was written in April 1978 for submission as a paper to the Symposium on Sparse Matrix Computations and Their Applications, which was held in Knoxville, Tennessee, November 2–3, 1978. The paper was presented at the Symposium.

In Section 8.3.2, we describe the basic hardware requirements and interconnection schemes for the proposed VLSI processor arrays and discuss the feasibility of building these networks. Section 8.3.3 deals with the matrix-vector multiplication problem. Multiplication of two matrices is considered in Section 8.3.4. In Section 8.3.5, we show that essentially the same interconnection scheme and algorithm as those used for matrix multiplication in Section 8.3.4 can be applied to find the LU-decomposition of a matrix. Section 8.3.6 is concerned with solving triangular linear systems. We show that this problem can be solved by almost the same network and algorithm for matrix–vector multiplication described in Section 8.3.3. Section 8.3.7 discusses applications and extensions of the results presented in the previous sections. The applications include the computations of finite impulse response filters, convolutions, and discrete Fourier transforms. Some concluding remarks are given in the last section.

The size of each of our processor array networks is dependent only on the band width of the band matrix to be processed and is independent of the length of the band. Thus, a fixed-size processor array can pipeline band matrices with arbitrarily long bands. The pipelining aspect of our algorithms is, of course, most effective for band matrices with long bands. For this reason most of the results in this paper will be presented in terms of their applications to band matrices. It is important to note, however, that all the results apply equally well to dense matrices since a dense matrix can be viewed as a band matrix with the maximum possible band width.

8.3.2 The Basic Components and Array Structures

8.3.2.1 The Inner Product Step Processor

The single operation common to all the algorithms considered in this section is the so-called inner product step, $C \leftarrow C + A \times B$. We postulate a processor that has three registers R_A, R_B, and R_C. Each register has two connections, one for input and one for output. Figure 8.6 shows two types of geometries for this processor. Type (a) geometry will be used for matrix–vector multiplication and solution of triangular linear systems (Sections 8.3.3 and 8.3.6), whereas type (b) geometry will be used for matrix multiplication and LU-decomposition (Sections 8.3.4 and 8.3.5). The processor is capable of performing the inner product step and is called the *inner product step processor*. We shall define a basic time unit in terms of the operation of this processor. In each unit time interval, the processor shifts the data on its input lines denoted by A, B, and C into R_A, R_B, and R_C, respectively; computes $R_C \leftarrow R_C + R_A \times R_B$; and makes the input values for R_A and R_B together with the new value of R_C available as outputs on the output lines denoted by A, B, and C, respectively. All outputs are latched and the logic is clocked so that when one processor is connected to another, the changing output of one during a unit time interval will not interfere with the input to another during this time interval. This is not the only processing element we shall make use of, but it will be the workhorse. A special processor for performing division will be specified later when it is used.

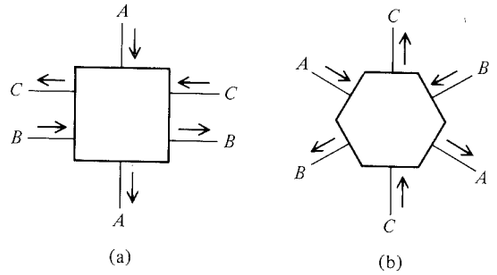


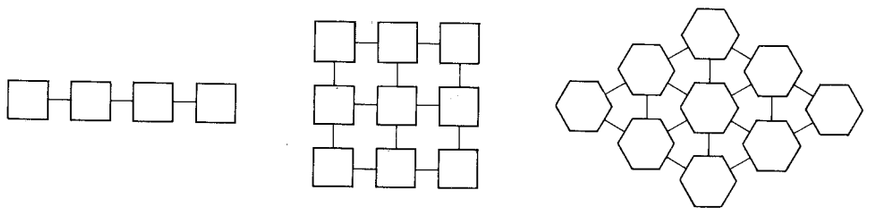
Fig. 8.6 Geometries for the inner product step processor.

8.3.2.2 Processor Arrays

A device is typically composed of many interconnected inner product step processors. The basic network organization we shall adopt for processors is mesh-connected and all connections from a processor are to neighboring processors. (See Fig. 8.7).

The most widely known system based on this organization is the ILLIAC IV.[14] If diagonal connections are added in one direction only, we shall call the resulting scheme *hexagonally mesh-connected*, or *hex-connected* for short. We shall demonstrate that linearly connected and hex-connected processors are natural for matrix problems.

Processors lying on the boundary of the processor array may have external connections to the host memory. Thus, an input/output data path of a boundary processor may sometimes be designated as an external input/output connection for the device. A boundary processor may receive input from the host memory through such an external connection, or it may receive a fixed value such as zero. On the other hand, a boundary processor may send data to the host memory through an external output connection. An output of a boundary processor may sometimes be ignored; this will be designated by omitting the corresponding output line.



(a) Linearly connected (b) Orthogonally connected (c) Hexagonally connected

Fig. 8.7 Mesh-connected processor arrays.

Throughout Section 8.3 we assume that the processors in an array are synchronous as described in Section 8.3.2.1. However, it is possible to view the processors as being asynchronous, each computing its output values when all its inputs are available as in a data flow model. For the purposes of this section we believe the synchronous approach to be more direct and intuitive.

The hardware demands of the VLSI processor arrays described here are readily seen to be modest. The processing elements are uniform, interprocessor connections are simple and regular, and external connections are minimized. It is our belief that construction of these processor arrays will prove to be cost-effective.

8.3.3 Matrix–Vector Multiplication

We consider the problem of multiplying a matrix $A = (a_{ij})$ with a vector $x = (x_1,\ldots,x_n)^T$. The elements in the product $y = (y_1,\ldots,y_n)^T$ can be computed by the following recurrences:

$$\begin{aligned} y_i^{(1)} &= 0, \\ y_i^{(k+1)} &= y_i^{(k)} + a_{ik}x_k, \\ y_i &= y_i^{(n+1)}. \end{aligned}$$

Suppose A is an $n \times n$ band matrix with band width $w = p+q-1$. (See Fig. 8.8 for the case when $p = 2$ and $q = 3$.) Then the above recurrences can be evaluated by pipelining the x_i and y_i through w linearly connected processors. We illustrate the algorithm for the band matrix–vector multiplication problem in Fig. 8.8. For this case the linearly connected network has four processors. See Fig. 8.9.

The general scheme of our pipelining algorithm can be viewed as follows: The y_i, which are initially zero, move to the left while the x_i are moving to the right and the a_{ij} are moving down. All the moves are synchronized. It turns out that each y_i

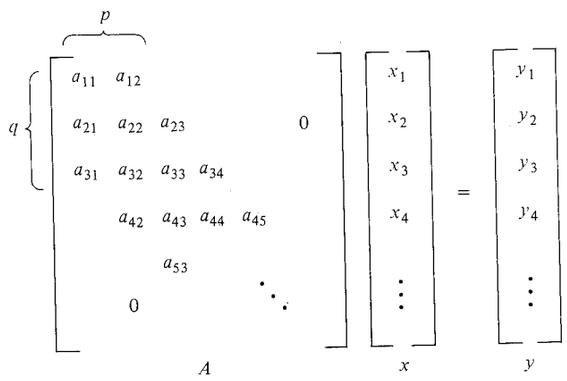


Fig. 8.8 Multiplication of a vector by a band matrix with $p = 2$ and $q = 3$.

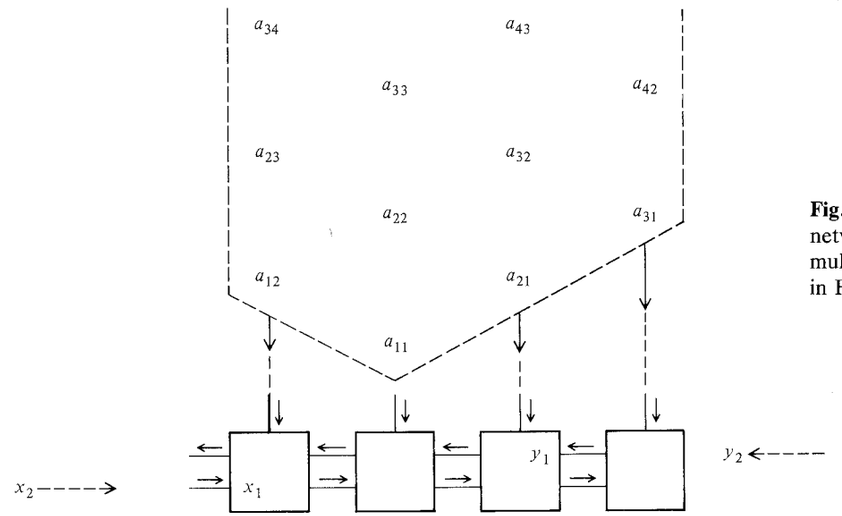


Fig. 8.9 The linearly connected network for the matrix–vector multiplication problem shown in Fig. 8.8.

is able to accumulate all its terms, namely, $a_{i,i-2}x_{i-2}$, $a_{i,i-1}x_{i-1}$, $a_{i,i}x_i$, and $a_{i,i+1}x_{i+1}$, before it leaves the network. Figure 8.10 illustrates the first seven steps of the algorithm.

Note that when y_1 and y_2 are output they have the correct values. Observe also that at any given time alternating processors are idle. Indeed, by coalescing pairs of adjacent processors, it is possible to use $w/2$ processors in the network for a general band matrix with band width w.

We now specify the algorithm more precisely. Assume that the processors are numbered by integers $1, 2, \ldots, w$ from the left-end processor to the right-end processor. Each processor has three registers, R_A, R_x and R_y, which will hold entries in A, x and y, respectively. Initially, all registers contain zeros. Each step of the algorithm consists of the following operations (but for odd-numbered time steps, only odd-numbered processors are activated, and for even-numbered time steps, only even-numbered processors are activated):

1. Shift.
   R_A gets a new element in the band of matrix A.
   R_x gets the contents of register R_x from the left neighboring node.
   (The R_x in processor 1 gets a new component of x.)
   R_y gets the contents of register R_y from the right neighboring node.
   (Processor 1 outputs its R_y contents and the R_y in processor w gets zero.)
2. Multiply and Add.
   $R_y \leftarrow R_y + R_A \times R_x$.

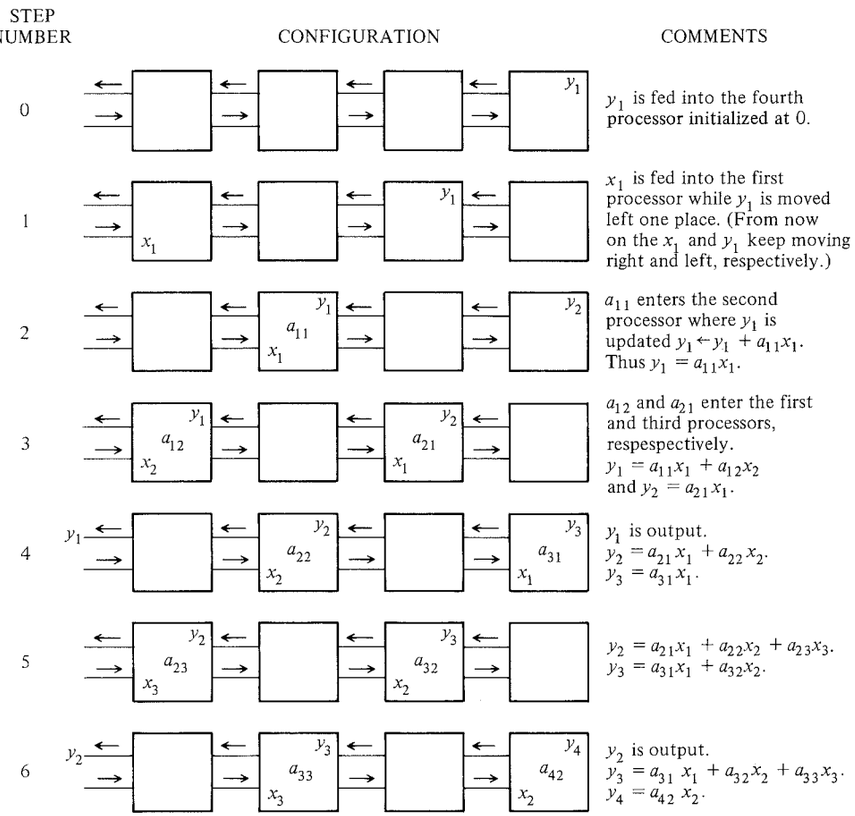


Fig. 8.10 The first seven steps of the matrix–vector multiplication algorithm.

Using the type (a) inner product step processor postulated in Section 8.3.2.1, we note that the three shift operations in step 1 can be done simultaneously, and that each step of the algorithm takes a unit of time. Suppose the bandwidth of A is w. It is readily seen that after w units of time the components of the product $y = Ax$ start shifting out from the left-end processor at the rate of one output every two units of time. Therefore, using our network all the n components of y can be computed in $2n + w$ time units, as compared to the $O(wn)$ time needed for the sequential algorithm on a uniprocessor.

8.3.4 Matrix Multiplication on a Hexagonal Array

This subsection considers the problem of multiplying two $n \times n$ matrices. It is easy to see that the matrix product $C = (c_{ij})$ of $A = (a_{ij})$ and $B = (b_{ij})$ can be

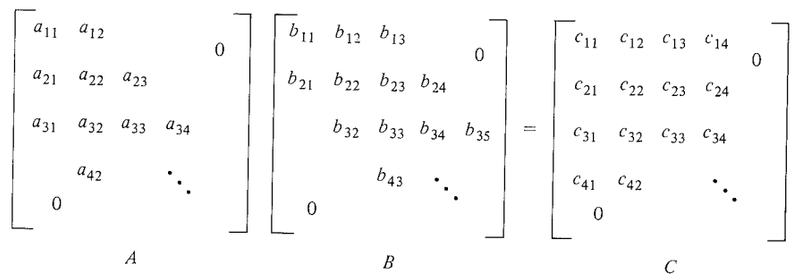


Fig. 8.11 Band matrix multiplication.

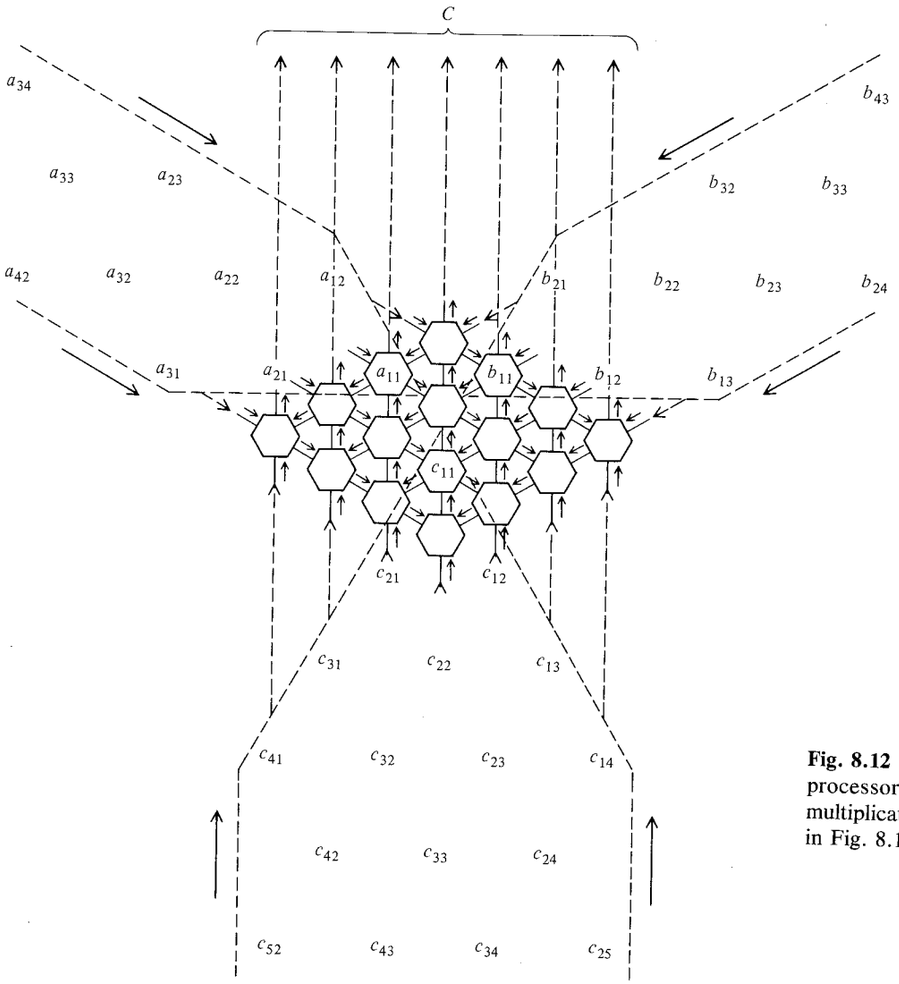
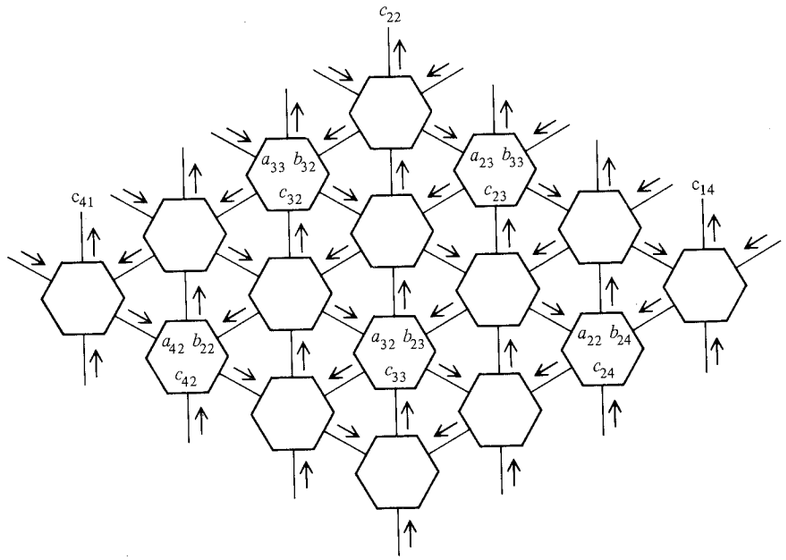
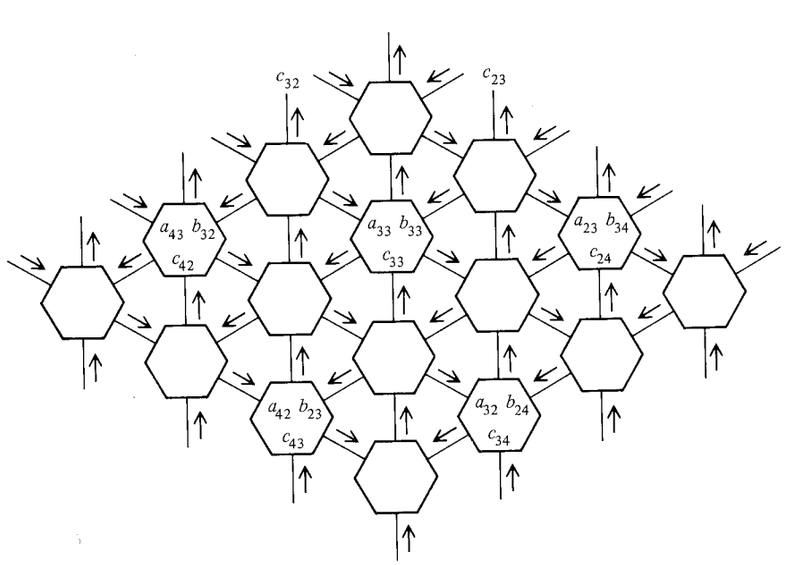


Fig. 8.12 The hex-connected processor array for the matrix multiplication problem shown in Fig. 8.11.

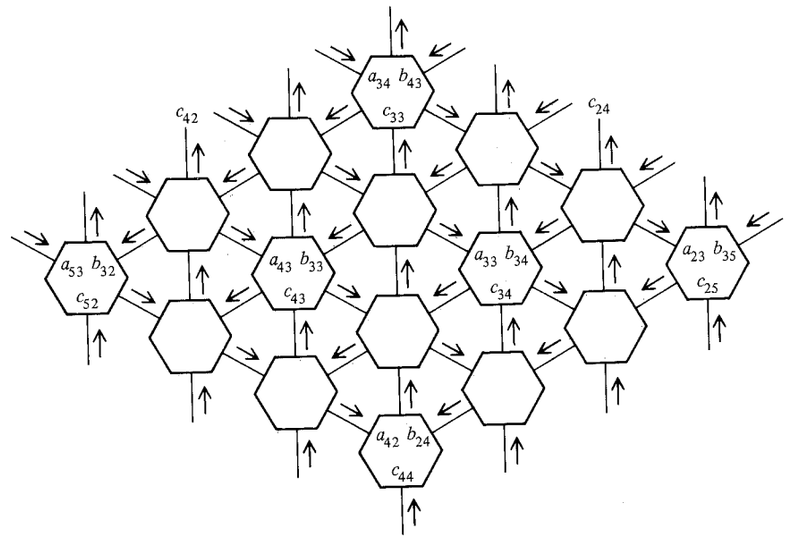


(a)

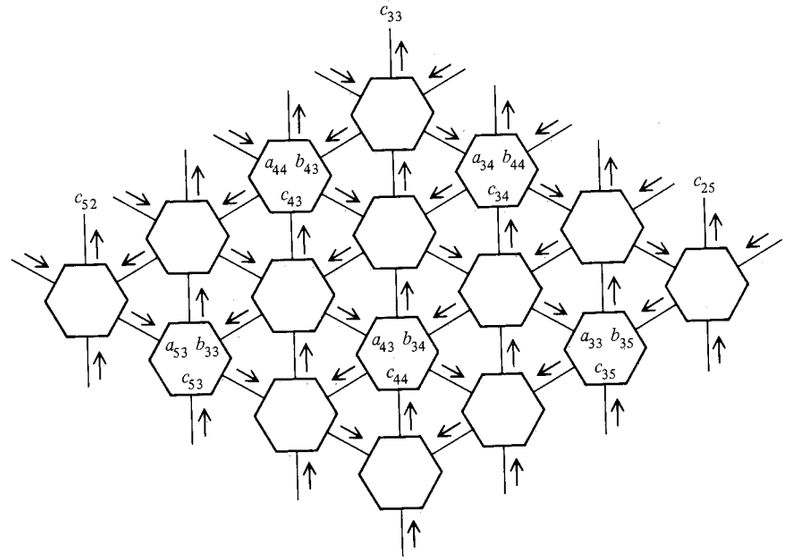


(b)

Fig. 8.13 Four steps during the matrix multiplication shown in Fig. 8.12.



(c)



(d)

computed by the following recurrences:

$$\begin{aligned} c_{ij}^{(1)} &= 0, \\ c_{ij}^{(k+1)} &= c_{ij}^{(k)} + a_{ik}b_{kj}, \\ c_{ij} &= c_{ij}^{(n+1)}. \end{aligned}$$

Let A and B be $n \times n$ band matrices of band width w_1 and w_2, respectively. We show how the recurrences above can be evaluated by pipelining the a_{ij}, b_{ij}, and c_{ij} through an array of w_1w_2 hex-connected processors. The algorithm uses the same principle as the one in Section 8.3.3. We illustrate the general scheme by considering the matrix multiplication problem depicted in Fig. 8.11. The diamond-shaped interconnection network for this case is shown in Fig. 8.12, where processors are hex-connected and data flows are indicated by arrows.

The elements in the bands of A, B, and C move through the network in three directions synchronously. Each c_{ij} is initialized to zero as it enters the network through the bottom boundaries. One can easily see that with the type (b) inner product processors described in Section 8.3.2.1, each c_{ij} is able to accumulate all its terms before it leaves the network through the upper boundaries. Figure 8.13 shows four consecutive steps in the execution of the algorithm. The reader is invited to study the data flow of this problem more closely by making transparencies of the band matrices (shown in the figures), and moving them over the network picture as described in the algorithm.

Let A and B be $n \times n$ band matrices of band width w_1 and w_2, respectively. Then a network of w_1w_2 hex-connected processors can pipeline the matrix multiplication $A \times B$ in $3n + \min(w_1, w_2)$ units of time.

Note that in any row or column of the network, out of every three consecutive processors, only one is active at any given time. It is possible to use about one third of the w_1w_2 processors in the network for multiplying two band matrices with band widths w_1 and w_2.

8.3.5. The LU-Decomposition of a Matrix on a Hexagonal Array

The problem of factoring a matrix A into lower and upper triangular matrices L and U is called LU-decomposition. Figure 8.14 illustrates the LU-decomposition of a band matrix with $p = 4$ and $q = 4$.

Once the L and U factors are known, it is relatively easy to invert A or solve the linear system $Ax = b$. (We deal with the latter problem in Section 8.3.6.) This section describes a parallel LU-decomposition algorithm that has hex-connected data paths.

We assume that matrix A has the property that its LU-decomposition can be done by Gaussian elimination without pivoting. (This is true, for example, when A is a symmetric positive-definite, or an irreducible, diagonally dominant matrix.) The

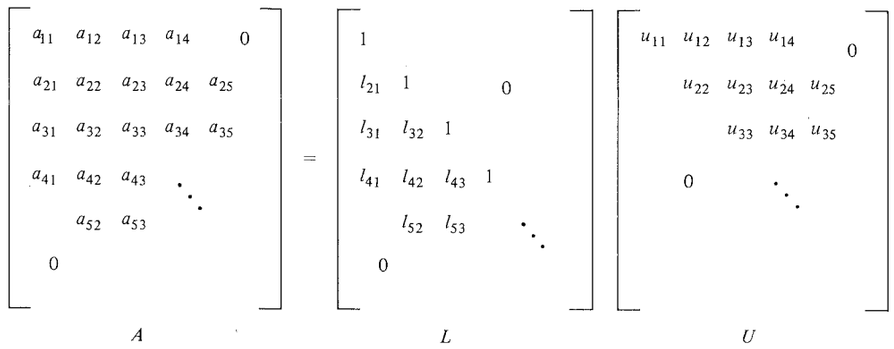


Fig. 8.14 The LU-decomposition of a band matrix.

triangular matrices $L = (l_{ij})$ and $U = (u_{ij})$ are evaluated according to the following recurrences:

$$\begin{aligned} a_{ij}^{(1)} &= a_{ij}, \\ a_{ij}^{(k+1)} &= a_{ij}^{(k)} + l_{ik}(-u_{kj}), \\ l_{ik} &= \begin{cases} 0 & \text{if } i < k, \\ 1 & \text{if } i = k, \\ a_{ik}^{(k)}u_{kk}^{-1} & \text{if } i > k, \end{cases} \\ u_{kj} &= \begin{cases} 0 & \text{if } k > j, \\ a_{kj}^{(k)} & \text{if } k \leq j. \end{cases} \end{aligned}$$

We show that the evaluation of these recurrences can be pipelined on a hex-connected processor array. A global view of this pipelined computation is shown in Fig. 8.15 for the LU-decomposition problem depicted in Fig. 8.14. The processor array in Fig. 8.15 is constructed as follows: The processors below the upper boundaries are the standard type (b) inner product step processors and are hex-connected in exactly the same way as the matrix multiplication network presented in Section 8.3.4. The processor at the top, denoted by a circle, is a special processor. It computes the reciprocal of its input and passes the result southwest and also passes the same input northward unchanged. The other processors on the upper boundaries are again type (b) inner product step processors, but their orientation is changed: the ones on the upper left boundary are rotated 120 degrees clockwise; the ones on the upper right boundary are rotated 120 degrees counterclockwise.

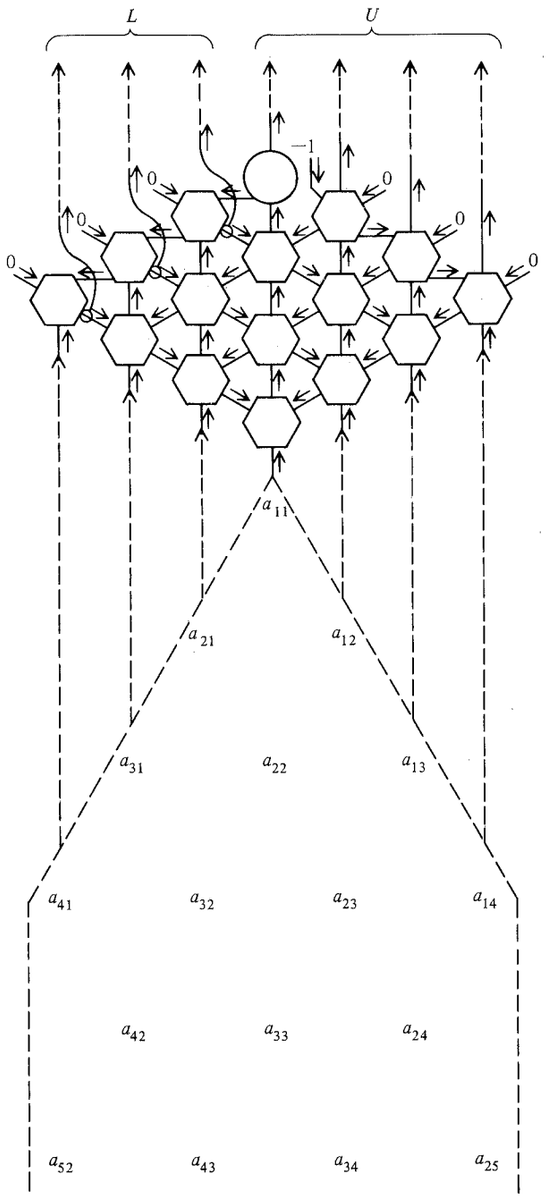
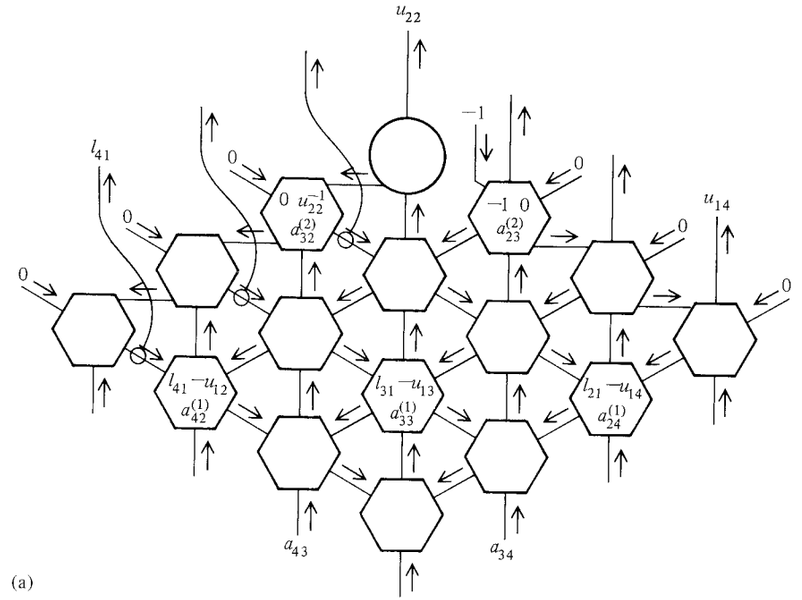
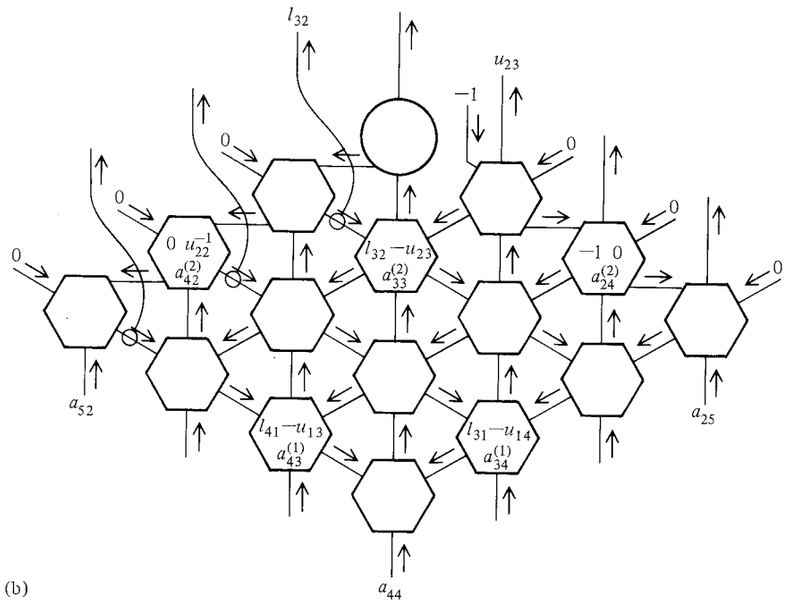


Fig. 8.15 The hex-connected processor array for pipelining the LU-decomposition of the band matrix in Fig. 8.14.



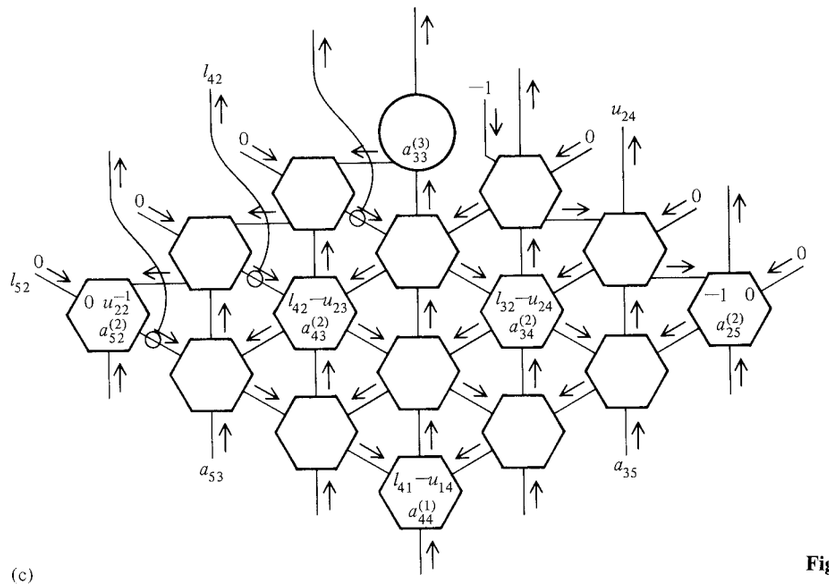
(a)



(b)

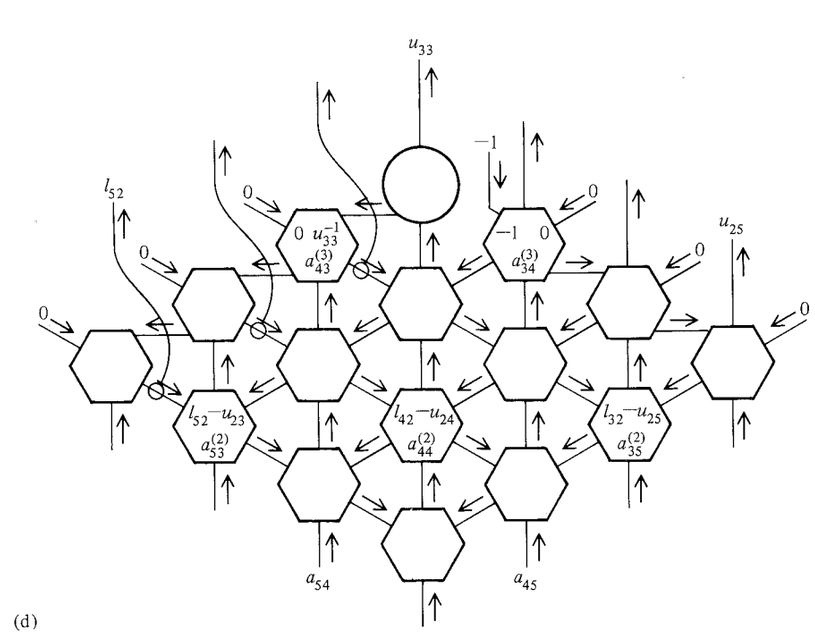
Fig. 8.16 Four steps during the LU-decomposition shown in Fig. 8.15.

(*Continued*)



(c)

Figure 8.16 (*cont.*)



(d)

The flow of data on the network is indicated by arrows in the figure. As in the matrix multiplication algorithm, each processor only operates every third time step. Figure 8.16 illustrates four consecutive steps during the execution of the algorithm. Note that in the figure, because A is a band matrix with $p = 4$ and $q = 4$, we have $a_{i+3,i}^{(k)} = a_{i+3,i}$ and $a_{i,i+3}^{(k)} = a_{i,i+3}$ for $1 \leq k \leq i$ and $i \geq 2$. Thus a_{52}, for example, can be viewed as $a_{52}^{(2)}$ when it enters the network.

There are several equivalent networks that reflect only minor changes to the network presented in this section. For example, the elements of L and U can be retrieved as output in a number of different ways. Also, the "-1" input to the network can be changed to a "$+1$" if the special processor at the top of the network computes minus the reciprocal of its input.

If A is an $n \times n$ band matrix with band width $w = p + q - 1$, a processor array having no more than pq hex-connected processors can compute the LU-decomposition of A in $3n + \min(p,q)$ units of time. If A is an $n \times n$ dense matrix, this means that n^2 hex-connected processors can compute the L and U matrices in $4n$ units of time, which includes I/O time.

The remarkable fact that the matrix multiplication network forms a part of the LU-decomposition network is due to the similarity of their defining recurrences. In any row or column of the LU-decomposition network, only one out of every three consecutive processors is active at a given time. As we observed for matrix multiplication, the number of processors can be reduced to about $pq/3$.

8.3.6 Triangular Linear Systems

Suppose that we want to solve a linear system $Ax = b$. Then after having done with the LU-decomposition of A (e.g., by methods described in Section 8.3.5), we still have to solve two triangular linear systems, $Ly = b$ and $Ux = y$. This section concerns itself with the solution of triangular linear systems. An upper triangular linear system can always be rewritten as a lower triangular linear system. Without loss of generality, this section deals exclusively with lower triangular linear systems.

Let $A = (a_{ij})$ be a nonsingular $n \times n$ band lower triangular matrix. Suppose that A and an n-vector $b = (b_1, \ldots, b_n)^T$ are given. The problem is to compute $x = (x_1, \ldots, x_n)^T$ such that $Ax = b$. The vector x can be computed by the following recurrences:

$$\begin{aligned} y_i^{(1)} &= 0, \\ y_i^{(k+1)} &= y_i^{(k)} + a_{ik}x_k, \\ x_i &= (b_i - y_i^{(i)})/a_{ii}. \end{aligned}$$

Suppose that A is a band matrix with band width $w = q$. (See Fig. 8.17 for the case when $q = 4$.) Then the above recurrences can be evaluated by the algorithm and network almost identical to those used for band matrix–vector multiplication in

Section 8.3.3. (Observe the similarity of the defining recurrences for these two problems.) We illustrate our result by considering the linear system problem in Fig. 8.17. For this case, the network and the general scheme of the algorithm are described in Fig. 8.18.

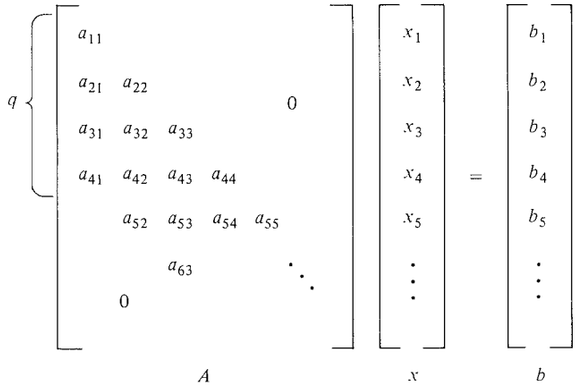


Fig. 8.17 The band (lower) triangular linear system where $q = 4$.

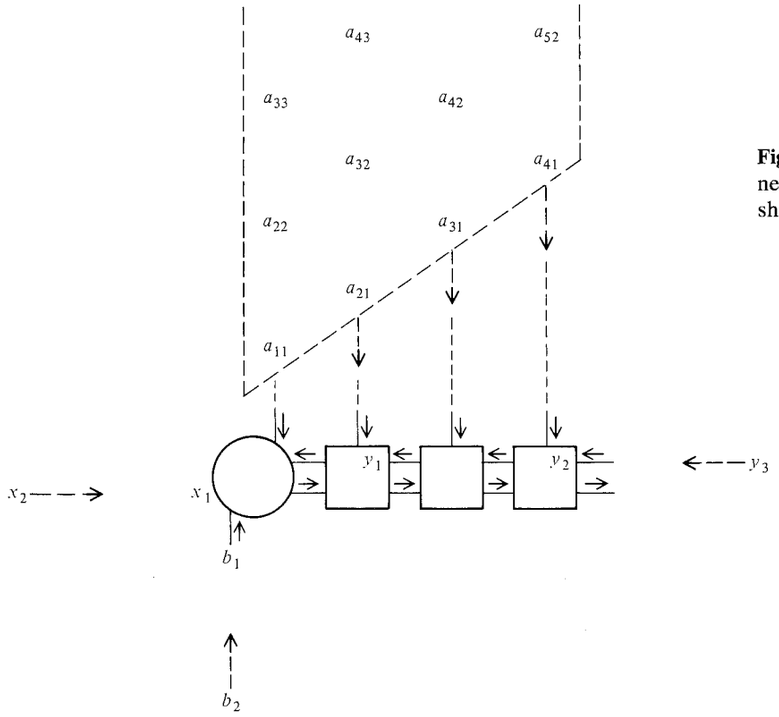


Fig. 8.18 The linearly connected network for solving the linear system shown in Fig. 8.17.

The y_i, which are initially zero, move leftward through the network while the x_i, a_{ij}, and b_i are moving as indicated in Fig. 8.18. The left-end processor is special in that it performs $x_i \leftarrow (b_i - y_i)/a_{ii}$. (In fact, the special processor introduced in Section 8.3.5 to solve the LU-decomposition problem is a special case of this more general processor.) Each y_i accumulates inner product terms in the rest of the processors as it moves to the left. At the time y_i reaches the left-end processor, it has the value $a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{i,i-1}x_{i-1}$, and consequently the x_i computed by $x_i \leftarrow (b_i - y_i)/a_{ii}$ at the processor will have the correct value. Figure 8.19 demon-

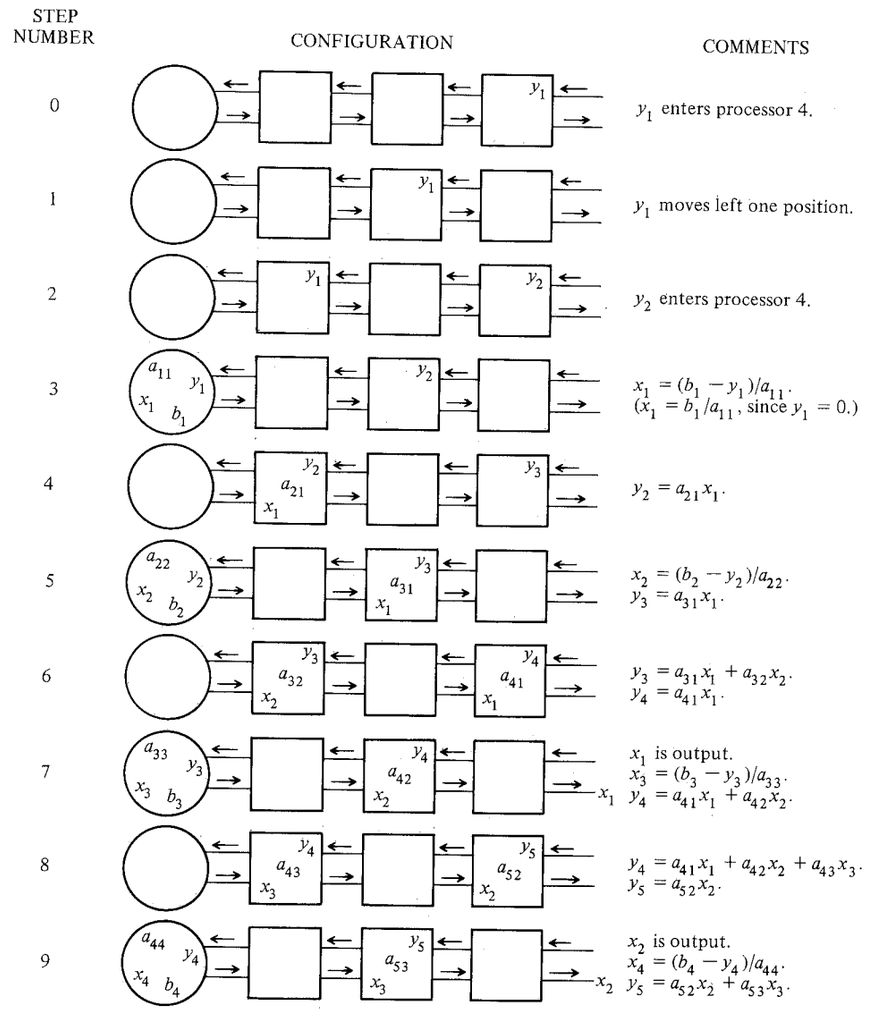


Fig. 8.19 Solving a lower band triangular system ($q = 4$).

strates the first seven steps of the algorithm. From the figure one can check that the final values of x_1, x_2, x_3, and x_4 are all correct. With this network we can solve an $n \times n$ band triangular linear system with band width $w = q$ in $2n + q$ units of time. As we observed for the matrix–vector multiplication problem, the number of processors required by the network can be reduced to $w/2$.

8.3.7 Applications and Comments

8.3.7.1 Variants of the Algorithms and Networks

Variants of the basic algorithms and networks presented above will often be used in actual practice. No attempt is given here for listing all the possible variants; it is important that the reader understand the basic principles used so that he or she can construct appropriate variants for specific problems.

As pointed out in Section 8.3.1, although most of our illustrations are done for band matrices, all the algorithms work for the regular $n \times n$ dense matrix. In this case the band width of the matrix is $w = 2n - 1$. If the band width of a matrix is so large that a corresponding algorithm requires more processors than a given network provides, then one should decompose the matrix and solve each subproblem on the network. For instance, the matrix multiplication of two $n \times n$ matrices or the LU-decomposition of an $n \times n$ matrix can be done in $O(n^3/k^2)$ time on a $k \times k$ array, for $k < n$.

One can often reduce the number of processors required by an algorithm if the matrix is known to be sparse or symmetric. For example, the matrices arising from a set of finite differences or finite elements approximations to differential equations are usually "sparse band matrices." These are band matrices whose nonzero entries appear only in a few of those lines in the band that are parallel to the diagonal. In this case by introducing proper delays to each processor for shifting its data to its neighbors, the number of processors required by the algorithm in Section 8.3.3 can be reduced to the number of those diagonal lines that contain nonzero entries. This variant is useful for performing iterative methods involving sparse band matrices. Another example is concerned with the LU-decomposition problem considered in Section 8.3.5. If matrix A is symmetric positive-definite, then it is possible to use only the left portion of the hex-connected network, since in this case U is simply DL^T, where D is the diagonal matrix $(a_{kk}^{(k)})$.

The optimal choice of the size of the network to solve a particular problem depends upon not only the problem but also the memory bandwidth to the host computer. For achieving high performance, it is desirable to have as many processors as possible in the network, provided they can all be kept busy doing useful computations.

It is possible to use our algorithms and networks to solve some nonnumerical problems when appropriate interpretations are given to the addition (+) and multiplication (×) operations. For example, some pattern-matching problems can be viewed as matrix problems with comparison and Boolean operations. It can be

instructive to view the + and × operations as operations in an abstract algebraic structure, such as a semi-ring, and then to examine how our results hold in these abstract settings.

8.3.7.2 Convolution, Filter, and Discrete Fourier Transform

There are a number of important problems that can be formulated as matrix–vector multiplication problems and thus can be solved rapidly by the algorithm and network in Section 8.3.3. The problems of computing convolutions, finite impulse response (FIR) filters, and discrete Fourier transforms are such examples. If a matrix has the property that the entries on any line parallel to the diagonal are all the same, then the matrix is a Toeplitz matrix. The convolution problem is simply the matrix–vector multiplication where the matrix is a triangular Toeplitz matrix (see Fig. 8.20).

$$\begin{bmatrix} a_1 & & & & & \\ a_2 & a_1 & & & & \\ a_3 & a_2 & a_1 & & & \\ a_4 & a_3 & a_2 & a_1 & & \\ a_5 & a_4 & a_3 & a_2 & a_1 & \\ \cdot & \cdot & \cdot & & & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ \vdots \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ \vdots \end{bmatrix}$$

Fig. 8.20 The convolution of vectors a and x.

A p-tap FIR filter can be viewed as a matrix–vector multiplication where the matrix is a band upper triangular Toeplitz matrix with band width $w = p$. Figure 8.21 represents the computation of a 4-tap filter.

$$\begin{bmatrix} a_1 & a_2 & a_3 & a_4 & & \\ & a_1 & a_2 & a_3 & a_4 & \\ & & a_1 & a_2 & a_3 & a_4 \\ & & & a_1 & a_2 & a_3 \\ & & & & \ddots & \\ 0 & & & & & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ \end{bmatrix}$$

A $\quad$ x $\quad$ y

Fig. 8.21 A 4-tap FIR filter with coefficients a_1, a_2, a_3, and a_4.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & \\ 1 & \omega & \omega^2 & \omega^3 & \\ 1 & \omega^2 & \omega^4 & \omega^6 & \\ 1 & \omega^3 & \omega^6 & \omega^9 & \\ & & & & \ddots \\ & & & & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ \vdots \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ \vdots \end{bmatrix}$$

Fig. 8.22 The discrete Fourier transform of vector x.

On the other hand, an n-point discrete Fourier transform is the matrix–vector multiplication, where the (i, j) entry of the matrix is $\omega^{(i-1)(j-1)}$ and ω is a primitive nth root of unity (see Fig. 8.22).

Therefore, using a linearly connected network of size $O(n)$ both the convolution of two n-vectors and the n-point discrete Fourier transform can be computed in $O(n)$ units of time, rather than $O(n \log n)$ as required by the sequential FFT algorithm. Moreover, note that for the convolution and filter problems each processor has to receive an entry of the matrix only once, and this entry can be shipped to the processor through horizontal connections and can stay in the processor during the rest of the computation. For the discrete Fourier transform problem each processor can in fact generate on-the-fly the powers of ω it requires. As a result, for these three problems it is not necessary for each processor in the network to have the external input connection on the top of the processor, as depicted in Fig. 8.9.

In the following we describe how the powers of ω can be generated on-the-fly during the process of computing an n-point discrete Fourier transform. The requirement is that if a processor is i units apart from the middle processor, then at time $i + 2j$ the processor must have the value of ω^{j^2+ij}, for all i, j. This requirement can be fulfilled by using the algorithm below. We assume that each processor has one additional register R_t. All processors except the middle one perform the following operations in each step, but for odd- (respectively, even-) numbered time steps, only processors that are odd (even) units apart from the middle processor are activated. For all processors except the middle one the contents of both R_A and R_t are initially "0".

1. Shift. If the processor is in the left- (respectively, right-) hand side of the middle processor, then

   R_A gets the contents of register R_A from the right- (respectively, left-) neighboring processor.

   R_t gets the contents of register R_t from the right- (respectively, left-) neighboring processor.

2. Multiply.

   $R_A \leftarrow R_A \times R_t$.

The middle processor is special; it performs the following operations at every even-numbered time step. For this processor the contents of both R_A and R_t are initially "1".

1. $R_A \leftarrow R_A \times R_t^2 \times \omega$.
2. $R_t \leftarrow R_t \times \omega$.

8.3.7.3 *The Common Memory Access Pattern*

Note that all the algorithms given in this section store and retrieve elements of the matrix in the same order. (See Figs. 8.9, 8.12, 8.15, and 8.18.) Therefore, we recommend that matrices always be arranged in memory according to this particular ordering so that they can be accessed efficiently by any of the algorithms.

8.3.7.4 *The Pivoting Problem, and Orthogonal Factorization*

In Section 8.3.5 we assume that the matrix A has the property that there is no need of using pivoting when Gaussian elimination is applied to A. What should one do if A does not have this nice property? (Note that Gaussian elimination becomes very inefficient on mesh-connected processors if pivoting is necessary.) This question motivated us to consider Givens's transformation (see, for example, Hammering[15]) for triangularizing a matrix, which is known to be a numerically stable method. It turns out that, like Gaussian elimination without pivoting, the orthogonal factorization based on Givens's transformation can be implemented naturally on mesh-connected processors, although a pipelined implementation appears to be more complex. (Results on Givens's transformation will be reported elsewhere.) (Sameh and Kuck[16] considered parallel linear system solvers based on Givens's transformation, but they did not give solutions to the processor communication problem considered here.)

8.3.8 Concluding Remarks

Research in interconnection networks and algorithms has been traditionally motivated by large scale parallel array computers such as ILLIAC IV.[6,17,18] The results presented here were, however, motivated by the advance in VLSI, though they are certainly applicable to parallel array processors. We have shown that many basic computations can be done very efficiently by special-purpose multiprocessors, which may be built cheaply using VLSI technology. The important feature common to all of our algorithms is that their data flows are very *simple* and *regular*, and they are *pipeline algorithms*. We have discovered that some data flow patterns are fundamental in matrix computations. For example, the two-way flow on the linearly connected network is common to both matrix–vector multi-

plication and solution of triangular linear systems (Sections 8.3.3 and 8.3.6), and the three-way flow on the hexagonally mesh-connected network is common to both matrix multiplication and LU-decomposition (Sections 8.3.4 and 8.3.5). A practical implication of this fact is that one device may be used for solving many different problems. Moreover, we note that almost all the processors needed in any of these devices are the inner product step processor postulated in Section 8.3.2. A careful design for this processor is desirable since it is the workhorse for all the devices presented.

For the important problem of solving a dense system of n linear equations in $O(n)$ time on $n \times n$ mesh-connected processors, we have improved upon the recent results of Kant and Kimura[19]. The basis of their results is a theorem on determinants that was known to J. Sylvester in 1851. Their algorithm requires that the matrix be "strongly nonsingular" in the sense that *every* square submatrix is nonsingular. It is sufficent for our algorithms that the matrix be symmetric positive-definite or irreducible diagonally dominant.

Hoare[20] and Thurber and Wald[7] describe some matrix multiplication algorithms on an orthogonally connected processor array. Unlike our results, their algorithms require that one or more of the three matrices involved in matrix multiplication have to stay in the array statically during the computation. This means extra I/O time and extra logic in each processing element in the network. Because of the use of hexagonal connection for the array, we are able to pipeline all three matrices through the network.

Inter-processor communications will likely continue to dominate the cost of parallel algorithms and systems. Communication paths inherently take more space and energy than processing elements in many problems of practical interest. We regard the problem of minimizing communication costs as fundamental. We hope the results of this section have demonstrated that the communication problem in parallel algorithms is not only tractable but also interesting. We expect that a large number of algorithms having small communication costs will be discovered in the future.

8.4 HIERARCHICALLY ORGANIZED MACHINES

We know that human organizations use hierarchical structure to extract the greatest possible benefit from the daily activities of tens of thousands of individuals. We know that complex systems can be constructed by subdividing them into less complex systems, which are again subdivided, as many times as necessary, until the resulting systems are simple enough to construct easily. In Section 8.5 we show that the organization of real estate on the silicon surface dictates a hierarchical communication system for any devices that must support global communication. Such hierarchical communication exists in conventional computers only in a limited way. Are there new machine structures that communicate hierarchically,

that support systems consisting of an arbitrary hierarchy of subsystems, and that can coordinate the activities of any number of submachines?

8.4.1 Binary Trees

Consider any number of processors physically arranged as a binary tree. Each processor has two subprocessors that it can control. These subprocessors, in turn, have two sub-subprocessors, and so on. A possible layout of such a binary processor tree is shown in Fig. 8.23. At the lowest level a small array of ordinary memory cells, labeled M_0, is accessed by the lowest level processors, labeled P_0. The combination of one lowest level processor with its associated memory is the element of computing power. These units are grouped together in pairs and accessed by the next level processor, labeled P_1. Two P_1's with their associated lower level units are grouped together and accessed by the next level higher processor, labeled P_2. This arrangement is repeated recursively until an entire silicon chip is covered by the processor-memory hierarchy. The rate at which information can be transferred within a processor is independent of the level of the processor. As the wires within a processor get longer, the drivers must become proportionately larger to drive them. The highest level processor that communicates off the silicon chip to the outside world has large drivers and hence is able to drive off-

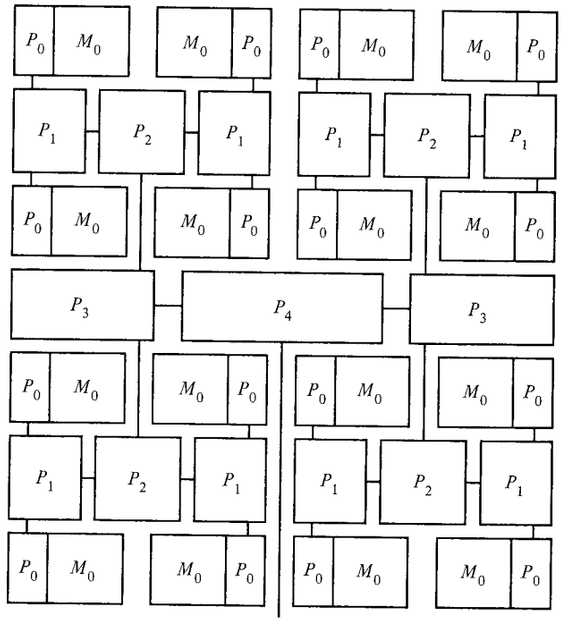


Fig. 8.23 Layout for a binary processor tree.