Chapter 8: Highly Concurrent Systems

Sections:

How can the properties of VLSI be exploited to build computational structures? Our discussion to this point has concentrated primarily on principles for structuring circuits and wires on the chip rather than on the application of VLSI to solve interesting computational problems. Although the OM example described in chapters 5 and 6 shows an elegant use of the structuring principles in the design of a conventional processor, we are left with an intriguing question: Does VLSI offer more than inexpensive implementations of conventional computers?

This chapter answers the question with a resounding YES! Because processing elements and memory elements can both be easily implemented in VLSI, we are encouraged to find structures that use a great deal of *concurrency*--a large number of calculations occurring at the same time. Although we can clearly design VLSI structures that have many sites at which processing is performed, how are these structures to be applied? Some applications may require different sorts of concurrent processing than others. Are there any principles or theories that will guide us in the design of highly concurrent systems? (For an excellent introduction to the promises and problems of VLSI and concurrency, see reference [1.]) Unfortunately, we lack experience in designing systems of this sort. As a consequence, this chapter can offer no complete designs which have been applied in real system applications. Instead, we offer several glimpses of the possibilities available with VLSI, and of its limitations.

The chapter is organized into four quite separate sections; although they are designed to be read sequentially, they may also be read concurrently! The first section reviews the problems that conventional computer designs present when implemented in VLSI, and summarizes efforts to achieve concurrency in general-purpose computers. Section 2 takes up a particular sort of concurrent organization--the array of identical processors--and shows its application to matrix arithmetic. Section 3 examines hierarchically-organized machines--in this case, machines structured as a binary tree--and demonstrates how they can be programmed to perform several tasks. Finally, section 4 presents a nascent theory of planar computational structures. It links the topological and electrical properties of VLSI elements to the structure of computations.

1. Communication and Concurrency in Conventional Computers

The architectures of conventional computers suffer from two difficulties that we must avoid when designing VLSI computational structures. First, a processor is separated from its memory by long communication paths such as buses. These buses are long enough to slow substantially the transmission of information between a processor and memory. Second, the "von Neumann machine" provides only a single processor that sequentially fetches and executes instructions--it offers few opportunities for concurrent processing activity. In this section, we survey some of the attempts to reduce communication costs and to use several processors concurrently. Although designs using a great deal of concurrency have been cumbersome to implement in the past, VLSI makes these designs considerably more attractive because of the ease with which memory and processing elements can be placed in close proximity.

Human organizations, like computer organizations, suffer if communication costs are high or if concurrent processing cannot be exploited. In fact, a human brings to an organization what VLSI brings to a circuit: both combine processing and memory effortlessly! Analogies with human structures will help to suggest the kinds of behavior we might achieve in computational structures.

Humans struggle to reduce communication costs, because the cost is often measured in large quantities of time. Consider a student assigned to write a research paper, requiring the use of a large library. Each time he needs to consult a book, he could make a trip to the library, climb into the stacks to retrieve the book, read a few relevant paragraphs, and replace the book. He now heads home to write the sentence that depends on the information he acquired. Libraries and people both recognize the inefficiency of this approach, and allow students to borrow books. The student will take several dozen books home, and store them on a short shelf, handy to his desk. Now the communication cost required to find information is reduced, provided the item lies within the group of books he has selected. If the student finds it difficult to select a small number of books that meet his needs, he may move his work to a carrel in the library, again in order to reduce communication costs with the large library "memory." The human strives to keep his information supply close to his processing task.

Concurrency is widely exhibited in human organizations. Henry Ford introduced the production line as a way to exploit concurrency in a well-understood manufacturing process. This is a particularly simple structure, in which information and goods flow rigidly along the production line. A more prevalent, general-purpose approach to concurrency in organizations is the *hierarchy*: the president of a company supervises several subordinates, each of whom in turn supervises a like number of sub-subordinates, and so forth until we reach the lowest level workers.

Two goals of the hierarchy are to keep everyone about equally busy, and to allow adequate information flow in the organization. A supervisor must generate enough commands to keep several subordinates busy--otherwise it would not be possible to build large organizations at all. In addition, each subordinate requires a certain amount of attention from the supervisor. These requriements limit the number of subordinates who can be assigned to a single supervisor--ten underlings can run the most diligent supervisor ragged. Supervisors gather information to make decisions by querying their subordinates. In a badly organized hierarchy, supervisors may confer frantically with their superiors to find answers needed for crucial decisions. Meanwhile workers stand idle, waiting for directions from above. While it is not possible in general to have all needed information available from one's own subordinates, concurrent systems require this *locality* property to reduce interference from too much communication.

The design of computers and of algorithms has yet to show the ingenuity reflected in human organizations. This failure is not for want of cleverness in designers, but rather because the technologies used to implement computers are much less flexible than the human beings used to implement corporations. VLSI offers more flexibility than earlier technologies because memory and processing structures can be implemented with the same technology, in close proximity.

Communication costs in computers

The archetypal computer consists of a single "processor" (the CPU or "central processing unit"), connected to a large, homogeneous memory (Figure 1). The processor fetches an instruction from memory, decodes it, executes it, and repeats the cycle. Many instructions will cause additional references to memory in order to fetch operands or to store results. The performance of such a computer depends critically on the speed with which memory can be accessed.

A very simple argument can be developed to determine the speed of the memory. If a memory of M bits is implemented on a single chip in a two-dimensional array, wires approximately $M^{1/2}$ long are required to transmit data between a memory cell and the processor. (We are concerned with relative units of length and time, because we intend only to compare different designs, not to determine absolute execution speeds.) The time required for data transmission is proportional to this length: the longer the wire, the greater the distance the signal must propagate and the greater the wire's capacitance, slowing propagation. In addition to slowing the memory, long wires also consume a great deal of chip space and require substantial power to drive. In present implementations of large computers, performance is further decreased by the several levels of packaging required to provide a memory of significant size: chip, printed-circuit board, backplane. The wiring on chips and printed-circuit boards grows as $M^{1/2}$, but backplane wiring grows linearly with memory size.

The organization shown in Figure 1 is also rather wasteful of resources: most of the memory and memory wiring is idle most of the time. For a typical large memory, M might be $32*10^6$, but only a 16- or 32-bit word will be delivered to the processor with each memory reference. If the memory is organized as an array of 10^6 bits for each bit in the word, only 2 of the 2000 wires needed to address the array are used in a given reference (1000 select wires running horizontally, and 1000 select wires running vertically). Vast areas of memory thus lie idle because the amount of information extracted on a single reference is small compared to the size of the entire memory.

The costs of communication are exhorbitant in today's computers. Most of the expense, time, and energy required to compute are consumed by the communication of data over large distances.

Memory Locality

Computer designers have recognized the difficulty of communicating with a very large memory, and have taken steps to utilize the memory more effectively. The result is a *memory hierarchy*, outlined in Figure 2. The processor communicates with a series of memories, whose size increases and speed decreases as they become farther from the processor. The closest memory (M_r) provides high-speed "registers" or "accumulators" that are used very frequently, usually to

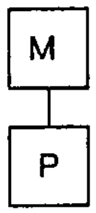


Fig. 1

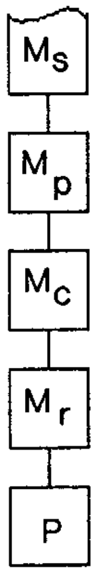


Fig. 2

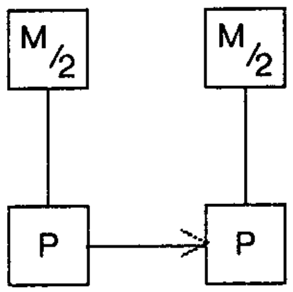


Fig. 3

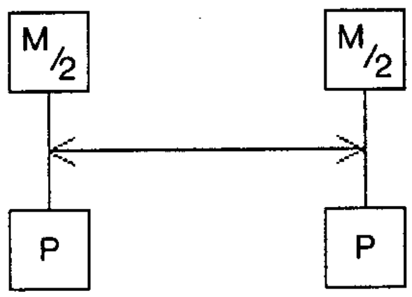


Fig. 4

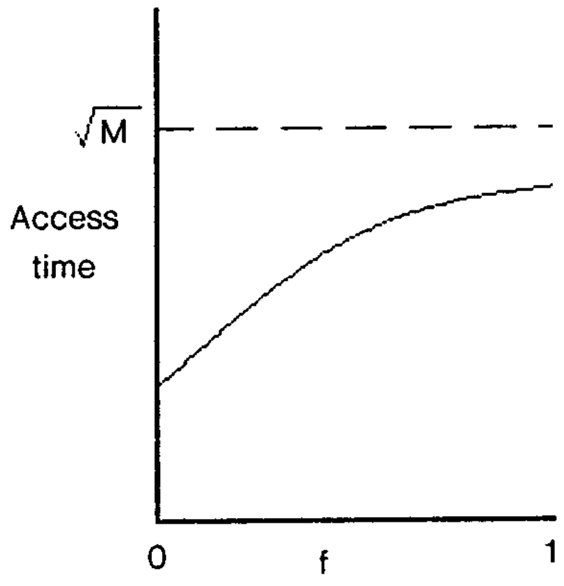


Fig.5

contain intermediate results of arithmetic calculations. Next comes "cache" memory (M_c), designed to hold data and instructions that are referenced frequently. The "primary" memory (M_p) is similar to the large memory of several million bits illustrated in Figure 1. Finally, a "secondary" memory (M_s) of some sort is provided, usually implemented with disks.

The average time required to reference a memory element will depend on which piece of the memory hierarchy holds the desired element. The intent is that fast, small memories be referenced more frequently than the slow, large ones. This desire is reflected in the design of the instruction set of the computer: referencing "registers" is usually encouraged by the structure of the instruction set; referencing primary memory (or cache) is supported by the instruction set, but perhaps in less flexible ways than for register access; finally, accessing a disk is not directly supported by instruction sets at all, but requires complicated "I/O control."

It is instructive to formulate a crude model to estimate the performance of the memory hierarchy. We need to assign representative values to the frequency with which each memory is accessed, and to the size of each memory:

$f_r \sim .6$	Frequency of access to registers (M_r)
$f_c \sim .38$	Frequency of access to cache (M_c)
$f_p \sim .02$	Frequency of access to primary memory (M_p)
$f_s \sim .000005$	Frequency of access to secondary memory (M_s)

$M_r \sim 16$
$M_c \sim 10^3$
$M_p \sim 10^5$
$M_s \sim 10^{10}$

Using our model of memory access time, the time required to access memory on the average is $f_r \# M_r + f_c \# M_c + f_p \# M_p + 100\ f_s \# M_s$, measured in arbitrary units. (The factor of 100 arises because disk access times are substantially worse than our memory wiring model indicates.) It is instructive to note the relative contributions of the separate memories: 2.4, 12, 6, 50, for a total of 70. The cost of access to the slowest memory, the disk, is the most important contribution to the average.

The memory hierarchy is an improvement over the homogeneous memory of Figure 1. The time to reference a single memory of size 10^5 is 320 units. The time to reference a three-level hierarchy of about the same size (M_r, M_c, M_p, with frequencies shown above) is a mere 20 units.

The effectiveness of the memory hierarchy depends on *locality* of the memory references. Cache algorithms copy large chunks (8-32 words) of primary memory into the cache, hoping that additional memory references will occur in the neighborhood of the first reference. A similar hope is attached to transfers from secondary memory. If an application arises in which most of the memory references do *not* go to the fast register memory, the memory hierarchy will perform poorly.

Locality can also be viewed as a function of size. If a program and its data can reside in primary memory for the duration of execution, and do not require secondary memory, the average memory access time will drop from 70 to 20. If the program is small enough to fit in the small cache memory, access time will drop further to 14.

Concurrency in computers

Not content with the increases in speed due to a memory hierarchy, computer designers have also sought to increase the concurrency in computer designs. A number of different approaches have been tried (see reference [3]); we shall illustrate *pipeline* structure and *multiprocessor* structures.

Pipelined processors

Pipelined processors are patterned after the production line found in manufacturing: a portion of the processing is performed by each of several processors, and then handed to the next processor in the line. Starting from Figure 1, the designer reasons that two processors could function concurrently, each assigned to half the original memory (Figure 3); a communication path is provided so that the first processor can transmit results to the second.

The two-processor pipeline more than doubles the processing power available. If we neglect the cost of inter-processor communication, the time required to execute an instruction is (1/2) $(M/2)^{1/2}$, about one third the time required by the uniprocessor in Figure 1. The improvement comes from two effects: doubling the number of processors doubles the speed, but reducing the memory size also increases speed.

A special case of pipelining is illustrated by *instruction-fetch overlap* in computers. One processor is responsible for fetching an instruction from memory; it then passes on to the second processor information required to execute the instruction; the second processor actually performs the execution. In chapter 6, we saw this technique applied in OM: while one microinstruction is being executed, the controller is fetching the next microinstruction. *Execution overlap* allows the execution itself to be pipelined (see reference [6] for more pipelining structures).

Pipelined structures are perhaps most effective in special-purpose applications that can utilize a large number of processors. Signal-processing is a particularly good example: a signal is sampled digitally to generate a *stream* of signal data. This data is pipelined through processors to perform corrections, correlations, frequency analysis, etc. Section 2 of this chapter illustrates the application of pipelines to matrix arithmetic of various sorts.

Unfortunately, it is not always possible to cast problems in a framework suited to execution on pipelined computers. If the workload is not divided evenly among the processors, some will stand idle, reducing the effective speed increase. But it is the rigid communication discipline that most severely restricts the application of pipelines.

Multiprocessors

Another important class of concurrent computers are *multiprocessors.* Unlike the pipeline, these structures provide switching structures that allow each processor to communicate with each other processor. The hope is that those algorithms not suited to pipelines because of their communication requirements can be executed on multiprocessors.

Figure 4 shows a dual-processor configuration, again adapted from Figure 1. Each processor communicates primarily with a memory half the size of the original. In addition, a common "bus" is provided to allow each processor to reference the other's memory.

Two problems with the dual-processor arrangement are immediately apparent. First, if each processor references memories at random, the two will interfere often, and vitiate some of the speed gain. Second, can we assure that the sequential program suited to the uniprocessor architecture of Figure 1 can be adapted to the dual-processor configuration? Putting aside for the moment the problems of programming a multiprocessor, we shall examine its performance.

We shall construct a crude model of the time required to execute an instruction on the dual processor. Assume that each processor references its own memory with probability $(1-f)$, and the other's with probability f. Further, assume that the useful duty cycle of each processor is d. If both processors can be productively employed at all times, d will be 1. However, if the two processors must occasionally wait for each other, i.e., must "synchronize," d may fall below 1. We can identify three cases:

1. P_a references M_a and P_b references M_b; probability is $(1-f)^2$.
2. P_a and P_b both reference M_a (or, equivalently M_b); probabilities sum to $2f(1-f)$.
3. P_a references M_b and P_b references M_a; probability is f^2.

We also need to model the time required to complete each of the three cases. A processor references its own memory, of size M/2, in time $(M/2)^{1/2}$. When a reference is made to a neighbor's memory, we assume the time for communication on the bus and referencing the memory sum to #M, as if it were addressing the entire memory as one array. The costs for the three cases then become:

1. $(M/2)^{1/2}$
2. $(M/2)^{1/2} + M^{1/2}$
3. $M^{1/2} + M^{1/2}$

From these estimates we calculate the expected instruction execution time, remembering that $2d$ processors are available:

$$\text{time} = M^{1/2} \quad (1/d) \quad (2^{1/2}/4 + f - f^2/2)$$

This expression is plotted in Figure 5, assuming $d=1$.

The simple model of a dual-processor configuration is suggestive of behavior we can expect from multi-processor systems that require global communication. We observe that if $f=0$, execution speed is more than twice that of the uniprocessor illustrated in Figure 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 multi-processor 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.

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
Mulitprocessors	$< n$

The processor-memory structures and algorithms presented in the remainder of this chapter all attempt *to have as many processors as can be kept productive simultaneously 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 [2,3], parallel processor and processing [4,5,6,7], performance evaluation [3], and algorithm design [8,9,10,11,14].

2. Algorithms for VLSI Processor Arrays

H. T. Kung and Charles E. Leiserson
Department of Computer Science
Carnegie-Mellon University

2.1. Introduction

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

We are interested in high-performance parallel algorithms that can be implemented directly on low-cost hardware devices. By performance, we are not refering 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. The cost of the device must be measured in how well it can be implemented using LSI technology and must be sensitive to what the technology can do cheaply, and what is expensive.

LSI 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. The two-dimensional array structure consisting of mesh-connected processors enjoys this desirable property. Therefore, we are interested in designing parallel algorithms which have simple and regular data flows so that they can be executed efficiently on such processor arrays. 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 this section, we demonstrate simple and regular multiprocessor networks that are capable of pipelining some important matrix computations with optimal speed-up.

Most of the results reported here are based on a paper by H. T. Kung and C. E. Leiserson, which is to be presented at the Symposium on Sparse Matrix Computations and Their Applications in Knoxville, TN, November 2-3, 1978.

2.2. The Basic Components and Structures

The single operation common to the problems considered in this section is the so-called inner product step, $C \leftarrow C + A \times B$. We postulate a processor which has three registers R_A, R_B, and R_C. Each register has two connections, one for input and one for output. Fig. 2.2.1 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 2.3 and 2.6), whereas type (b) geometry will be used for matrix multiplication and LU-decomposition (Sections 2.4 and 2.5).

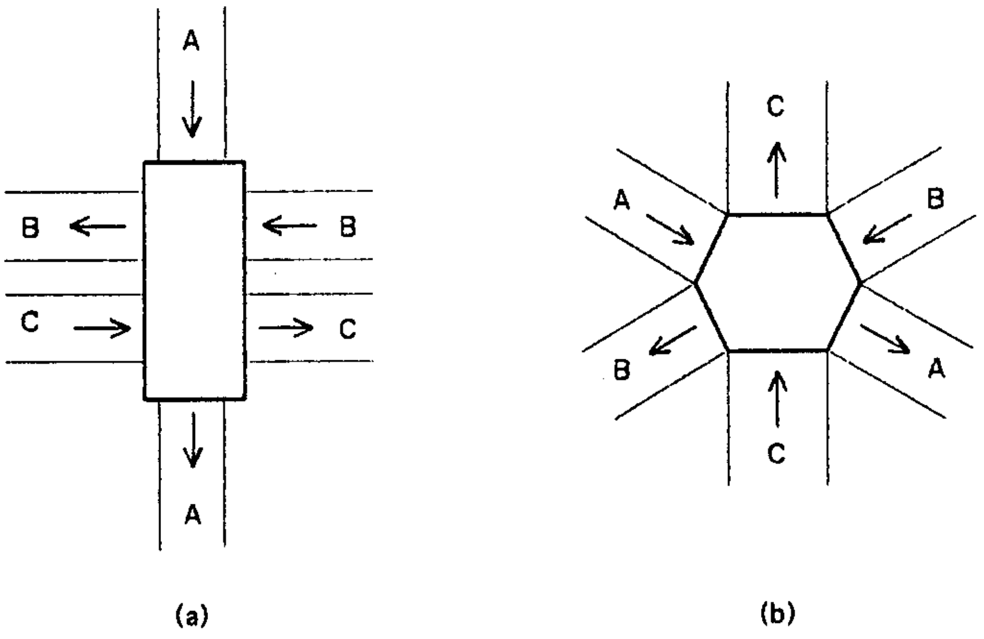


Fig. 2.2.1. Geometries for the inner product step processor.

The processor is capable of performing the inner product step. We shall define a basic time unit in terms of this processor. At time t, the processor shifts its inputs into R_A, R_B, and R_C, and computes $R_C \leftarrow R_C + R_A \times R_B$. At time t+1, the new value of R_C together with the input values for R_A and R_B are available as outputs. All outputs are latched and the logic is clocked so that when one processor is connected to another, the changing output of one during the time interval between t and t+1 will not interfere with the input to the other during this time. This is not the only processing element we shall make use of, but it will be the work horse. These special processors will be specified later when they are used.

The basic network organization we shall adopt for internal communication is the mesh-connected processor scheme. (See Fig. 2.2.2.) All connections from a processor are to neighboring processors. The most widely known system based on this organization is the ILLIAC IV. 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.

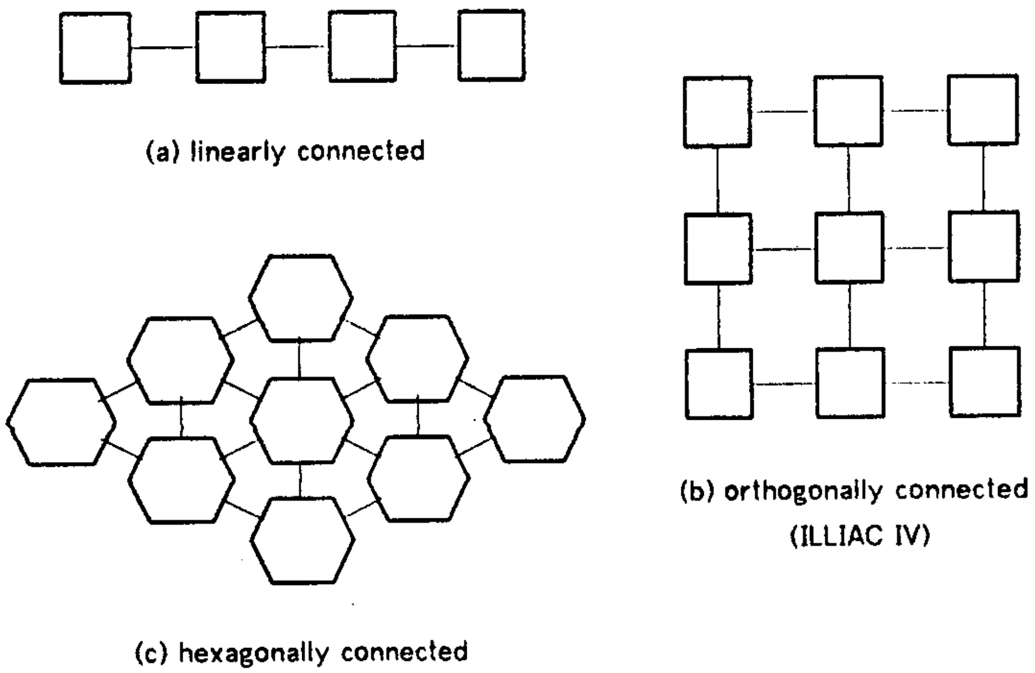
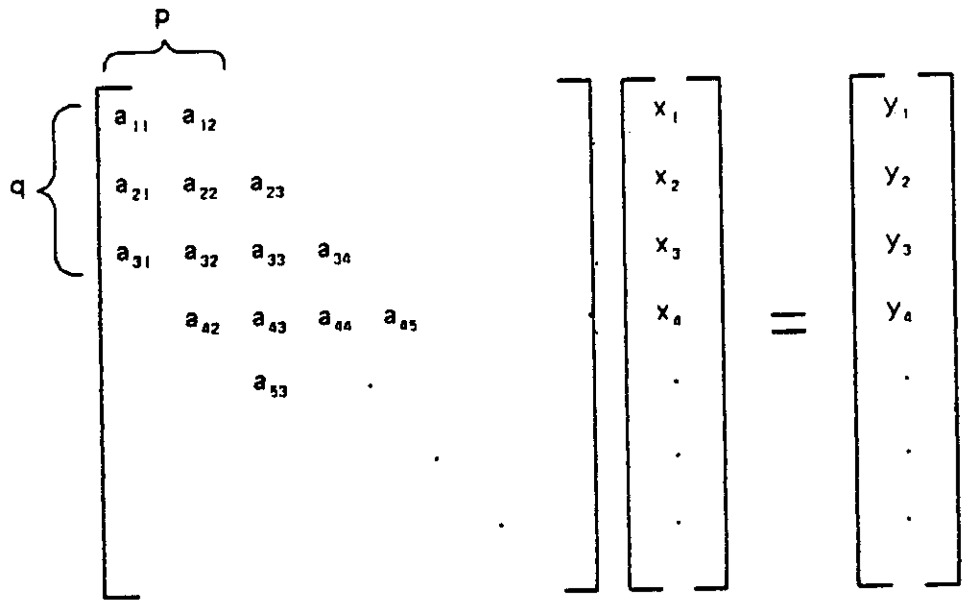

(a) linearly connected
(b) orthogonally connected (ILLIAC IV)
(c) hexagonally connected

Fig. 2.2.2. Examples of mesh-connected processors.

When an input path to a processor lies on an edge of the device, we shall sometimes designate it as an external input connection from the host memory. Alternatively, we may let the input have a fixed value such as zero. An output data path will either go to the host memory or be ignored.

2.3. Matrix-Vector Multiplication

We consider the problem of multiplying a matrix with a vector. Let $A = (a_{ij})$ be an nxn band matrix with band width $w = p+q-1$, and $x = (x_1,...,x_n)^T$, $y = (y_1,...,y_n)^T$ be n-vectors such that $Ax = y$. (See Fig. 2.3.1 for the case when $p = 2$ and $q = 3$.)



p
q
a_{11} a_{12}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33} a_{34}
a_{42} a_{43} a_{44} a_{45}
a_{53}
x_1 x_2 x_3 x_4
=
y_1 y_2 y_3 y_4

Fig. 2.3.1. The matrix-vector multiplication when the matrix is a band matrix with $p = 2$ and $q = 3$.

Suppose A and x are given. The following algorithm computes the product $y = Ax$ by pipelining the computation through w linearly connected processors. Before giving the code for each processor, we illustrate the algorithm for the band matrix-vector multiplication problem in Fig. 2.3.1. For this case the linearly connected network has four processors. See Fig. 2.3.2.

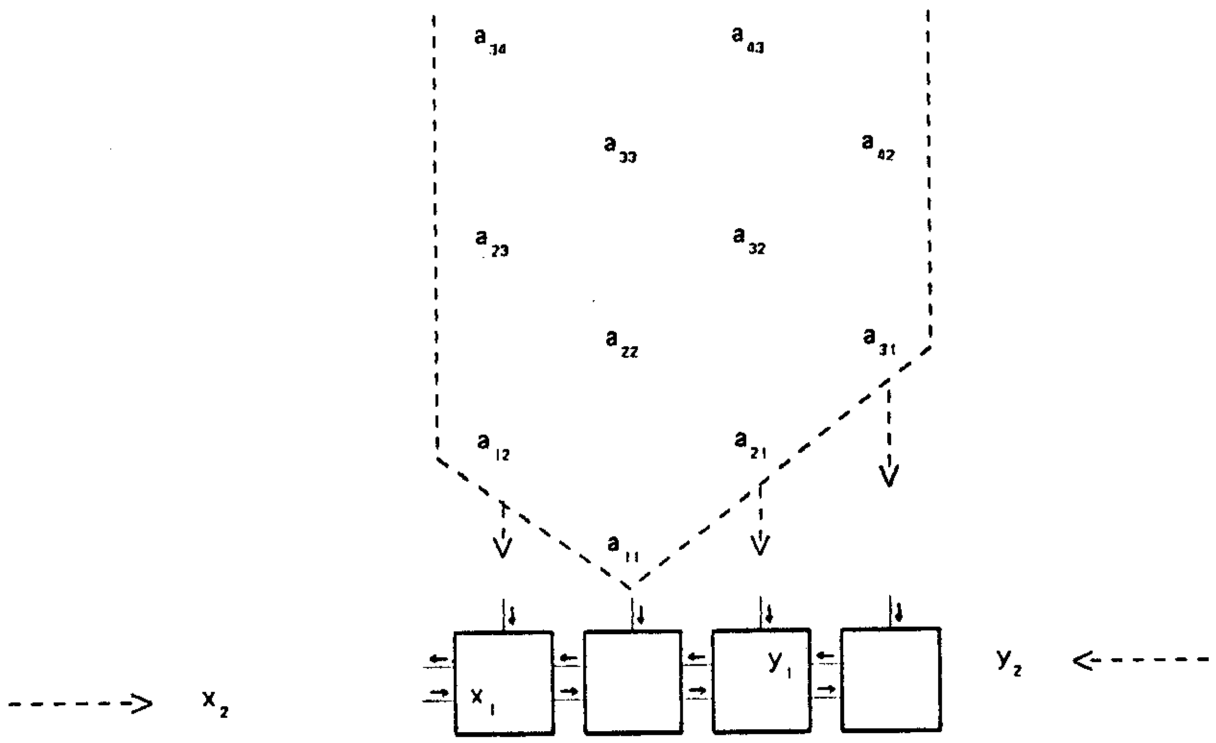


Fig. 2.3.2. The linearly connected network for the matrix-vector multiplication problem shown in Fig. 2.3.1.

The general scheme of our pipelining algorithm can be viewed as follows. The y_i, which are initially zero, keep moving to the left while the x_i are moving to the right and the a_{ij} are moving down. It turns out that each y_i 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. Fig. 2.3.3 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, it is possible to use $w/2$ processors in the network for a general band matrix with band width w. We did not do so for the sake of clarity.)

We now specify the algorithm more precisely. Assume that the processors are numbered by integers 1, 2, . . ., 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.*

   $R_y \leftarrow R_y + R_A \times R_x$.

Using the processor postulated in section 2.2, 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 $w = p+q-1$. 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 single processor.

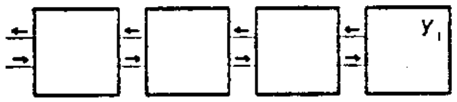
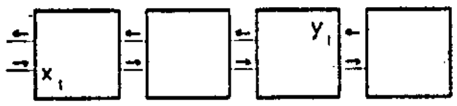
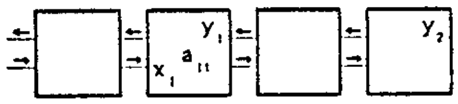
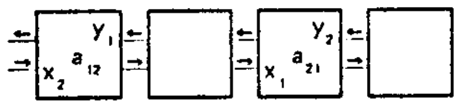
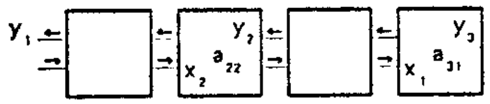
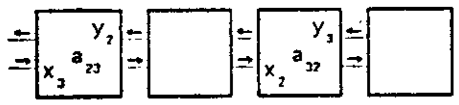
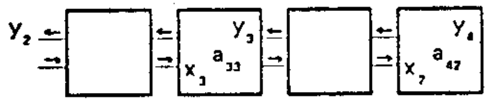
Step Number	Configuration	Comments
0		y_1 is fed into the fourth processor initialized at 0.
1		x_1 is fed into the first processor while y_1 is moved left one place. (From now on the x_i and y_i keep moving right and left, respectively.)
2		a_{11} enters the second processor where y_1 is updated $y_1 \leftarrow y_1 + a_{11}x_1$. Thus $y_1 = a_{11}x_1$.
3		a_{12} and a_{21} enter the first and third processors, respectively. $y_2 = a_{11}x_1 + a_{12}x_2$ and $y_2 = a_{21}x_1$.
4		y_1 is output. $y_2 = a_{21}x_1 + a_{22}x_2$. $y_3 = a_{31}x_1$.
5		$y_2 = a_{21}x_1 + a_{22}x_2 + a_{23}x_3$. $y_3 = a_{31}x_1 + a_{32}x_2$.
6		y_2 is output. $y_3 = a_{31}x_1 + a_{32}x_2 + a_{33}x_3$. $y_4 = a_{42}x_2$.

Fig. 2.3.3. The first seven steps of the matrix-vector multiplication algorithm.

2.4. Matrix Multiplication

This section considers the problem of multiplying two matrices. Let A and B be nxn band matrices of bandwidth w_1 and w_2, respectively. We show that a network of $w_1 w_2$ hex-connected processors can compute the product C = AxB in $3n+\min(w_1,w_2)$ units of time. The algorithm uses the same principle as the one in Section 2.3. We illustrate the general scheme by considering the matrix multiplication problem depicted in Fig. 2.4.1.

The diamond shaped interconnection network for this case is shown in Fig. 2.4.2, where processors are hex-connected and data flows are indicated by arrows. The nonzero elements in A, B and C move through the network in three directions, as indicated in the figure. Initially, the c_{ij} are all zeros. One can easily see that with the type (b) inner product processors described in Section 2.2, each c_{ij} is able to accumulate all its terms before it leaves the network.

Suppose that Fig. 2.4.2 describes the configuration at time 1. Then, for example, c_{11} gets $a_{11}b_{11}$ at time 2 and $a_{12}b_{21}$ at time 3, while c_{21} gets $a_{21}b_{11}$ at time 3 and $a_{22}b_{21}$ at time 4. (Note that approximately only one third of processors in the network are active at a given time. Indeed, it is possible to use about $(w_1 w_2)/3$ processors in the network for multiplying two band matrices with band widths w_1 and w_2.)

$$\begin{bmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & a_{34} & \\ & a_{42} & & \cdot & \\ & & & & \ddots \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & & \\ b_{21} & b_{22} & b_{23} & b_{24} & \\ & b_{32} & b_{33} & b_{34} & b_{35} \\ & & b_{42} & \cdot & \\ & & & & \ddots \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} & \\ c_{21} & c_{22} & c_{23} & c_{24} & \\ c_{31} & c_{32} & c_{33} & c_{34} & \\ c_{41} & c_{42} & & \cdot & \\ & & & & \ddots \end{bmatrix}$$

Fig. 2.4.1. Matrix multiplication.

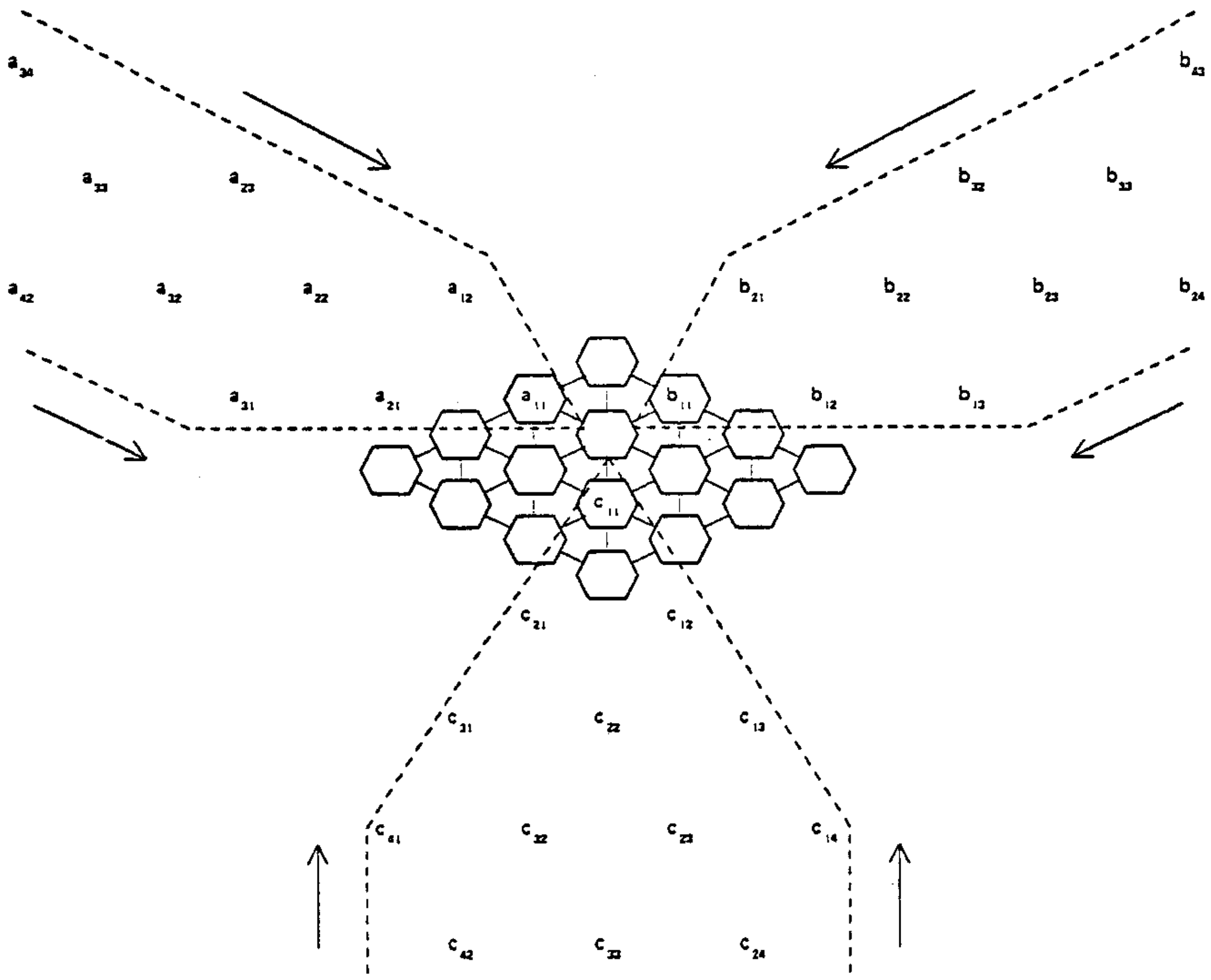


Fig. 2.4.2. The network for the matrix multiplication C = AxB shown in Fig. 2.4.1.

2.5. The LU-Decomposition of a Matrix

The LU-decomposition of a given a matrix A is the problem of computing lower and upper triangular matrices L and U such that A = LU. (Cf. Fig. 2.5.1.)

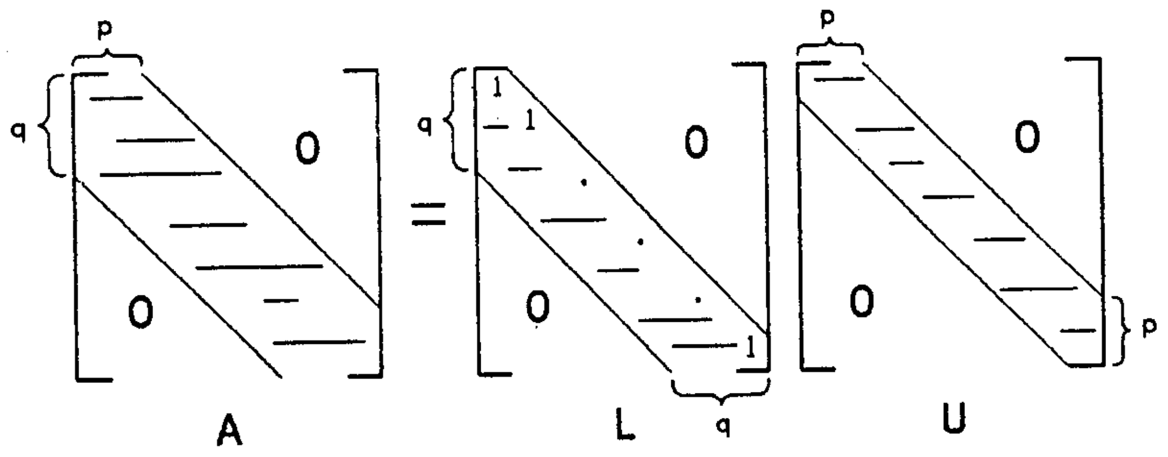


Fig. 2.5.1. The LU-decomposition of a matrix.

Once the L and U factors are known it would be relatively easy to solve a linear system Ax = b or to invert A. In the following we describe a parallel LU-decomposition algorithm using a hex-connected network.

We assume that A is either a symmetric positive-definite or irreducible diagonally dominant matrix. It is well-known that under this assumption the L and U matrices can be obtained by Gaussian elimination without pivoting. We show the rather surprising fact that Gaussian elimination enjoys the same data flow as matrix multiplication and that all the processors except one perform the same inner product step. In fact, the same matrix multiplication network in Section 2.4 can be used to compute L and U matrices, provided that the processor at the top now computes minus the reciprocal of an input and the orientation of the other boundary processors is properly altered. More precisely, at the special processor at the top, the data from the south processor is passed unchanged to the north, minus its reciprocal is computed and sent to the southwest processor, and a numerical value "1" is sent to the southeast processor. The processors on the left hand "upper" side are rotated 120 degrees clockwise and always receive "0" from their northwest external input connections. Similarly, the processors on the right hand "upper" side are rotated 120 degrees counterclockwise and always receive "0" from their northeast external input connections. (Of course, it is not necessary to actually input "0" for these processors; we did so for the sake of uniformity.)

Suppose that $L = (l_{ij})$ and $U = (u_{ij})$. Then Gaussian elimination computes the entries in L and U using the following procedure:

$$l_{ij} = m_{ij} \text{ for } i > j, \ 1 \text{ for } i = j \text{ and } 0 \text{ for } i < j,$$

$$u_{ij} = a_{ij}^{(i)} \text{ for } i \leq j \text{ and } 0 \text{ for } i > j,$$

$$a_{ij}^{(0)} = a_{ij},$$

$$m_{ik} = - a_{ik}^{(k)} / a_{kk}^{(k)},$$

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} + m_{ik} \ a_{kj}^{(k)}$$

To illustrate our results, we consider a band matrix A with $p = 4$ and $q = 4$. When elements in the band of matrix A are fed into the lower edge of the hex-connected network as shown in Fig. 2.5.2, the elements of L and U are output from the upper edge. Fig. 2.5.3 shows an enlargement of the configuration after eight steps of the algorithm have been executed. The flow of data on the network is indicated by arrows in Fig. 2.5.3. The hexagons denote the standard processors which perform the inner product step just like the corresponding processors in the matrix multiplication network (cf. Fig. 2.4.2). The processor at the top denoted by a circle performs the reciprocal and negation operations. As in the matrix multiplication algorithm, each processor only operates once every three time steps. We will not give a formal correctness proof for the algorithm here. But for understanding the algorithm the reader is advised to view the LU-decomposition as the inverse problem of multiplying a lower triangular matrix with 1's on the diagonal to an upper triangular matrix. Then the algorithm of this section can simply be regarded as one which undoes the matrix multiplication algorithm of Section 2.5. Having realized this, one should be able understand also why the two algorithms use the same network and enjoy the same data flow pattern. The idea of using the same network for both the forward and backward problems seems to be general. It will be used again in Section 2.6.

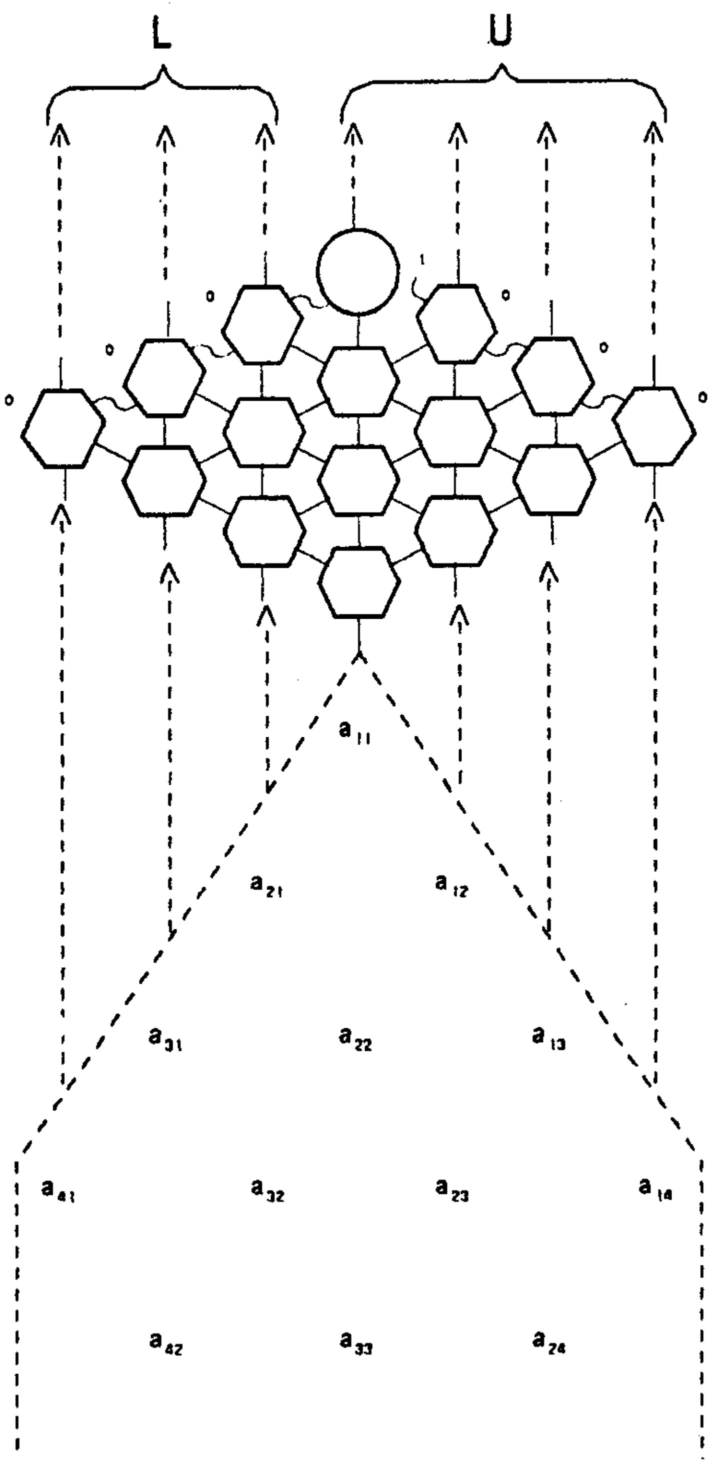


Fig. 2.5.2. The hex-connected network for pipelining the LU-decomposition of a band matrix with p = 4 and q = 4.

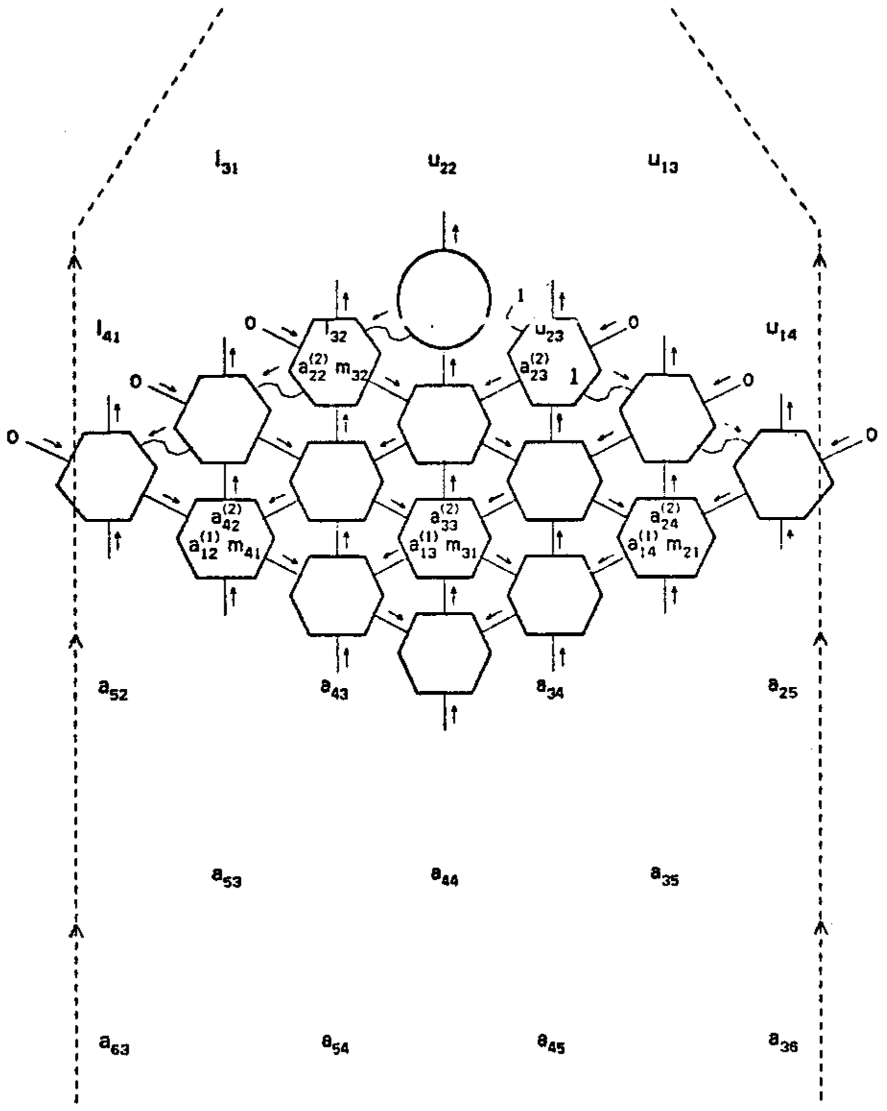


Fig. 2.5.3. LU-decomposition after the first eight steps.

It is readily seen that if matrix A is nxn, then using the network shown in Fig. 2.5.2 the L and U matrices can be computed in 3n+4 units of time. In general, if A is an nxn band matrix with band width w = p+q-1, then with a network of no more than pq hex-connected processors, the LU-decomposition of A can be done in 3n+min(p,q) units of time. (It is possible to reduce the number of required processors to about pq/3.) In particular if A is an nxn dense matrix, then n^2 hex-connected processors can compute the L and U matrices in 4n units of time, including I/O time.

2.6. 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 2.5), 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.

Let A = (a_{ij}) be a nonsingular nxn band lower triangular matrix with band width w=q. 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. (See Fig. 2.6.1 for the case when q=4.)

$$
q\left\{\begin{array}{c}\\ \\ \\ \\ \end{array}\right.
\!\!\begin{bmatrix}
a_{11} & & & & & & \\
a_{21} & a_{22} & & & & & \\
a_{31} & a_{32} & a_{33} & & & & \\
a_{41} & a_{42} & a_{43} & a_{44} & & & \\
 & a_{52} & a_{53} & a_{54} & a_{55} & & \\
 & & a_{63} & & \cdot & & \\
 & & & & & \cdot & \\
 & & & & & & \cdot
\end{bmatrix}
\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ \cdot \\ \cdot \\ \cdot \end{bmatrix}
=
\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ \cdot \\ \cdot \\ \cdot \end{bmatrix}
$$

Fig. 2.6.1. The band (lower) triangular linear system with q=4.

We show that this problem can be solved by the algorithm and network almost identical to those used for band matrix-vector multiplication in Section 2.3. (Note that the linear system problem can be regarded as the inverse of the matrix-vector multiplication problem.) We illustrate our result by considering the linear system problem in Fig. 2.6.1. For this case, the network and the general scheme of the algorithm are described in Fig. 2.6.2.

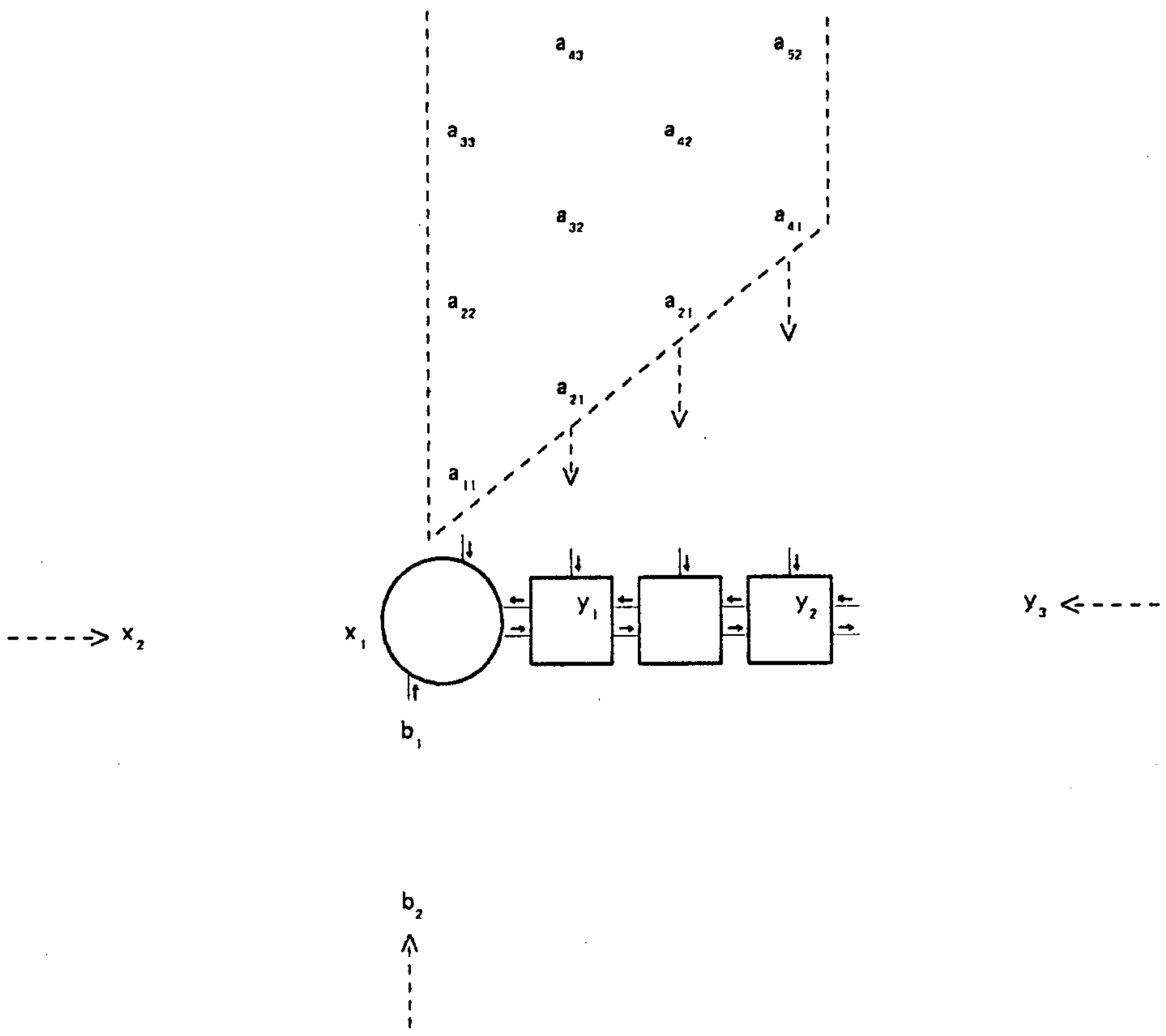
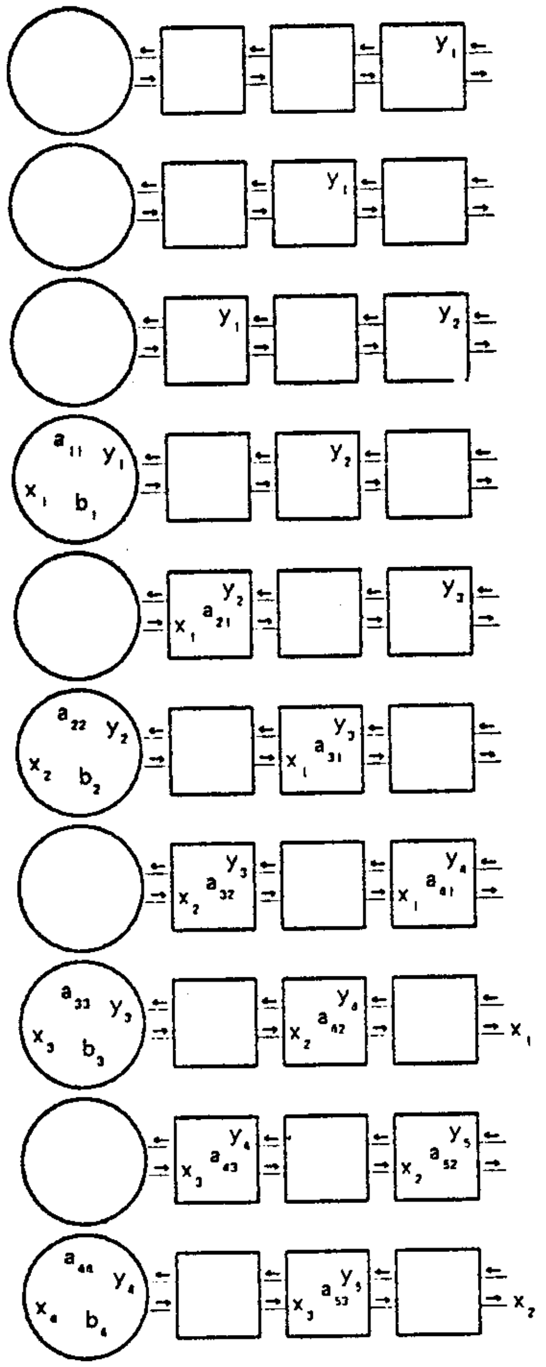


Fig. 2.6.2. The linearly connected network for solving the linear system shown in Fig. 2.6.1.

The y_i, which are initially zero, keep moving to the left while the x_i, a_{ij} and b_i are moving in the network, as indicated in Fig. 2.6.2. The left end processor is special in that it performs $x_i \leftarrow (b_i - y_i)/a_{ii}$. 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 $a_{i1}x_1 + a_{i2}x_2 + ... + a_{i,i-1}x_{i-1}$, and, consequently, the x_i computed by $x_i \leftarrow (b_i - y_i)/a_{ii}$ at the processor will have the correct value. Fig. 2.6.3 demonstrates the the first ten steps of the algorithm.



Step Number	Configuration	Comments
0		y_1 enters processor 4.
1		y_1 moves left one position.
2		y_2 enters processor 4.
3		$x_1 = (b_1 - y_1)/a_{11}$. ($x_1 = b_1/a_{11}$, since $y_1 = 0$.)
4		$y_2 = a_{21}x_1$.
5		$x_2 = (b_2 - y_2)/a_{22}$. $y_3 = a_{31}x_1$.
6		$y_3 = a_{31}x_1 + a_{32}x_2$. $y_4 = a_{41}x_1$.
7		x_1 is output. $x_3 = (b_3 - y_3)/a_{33}$. $y_4 = a_{41}x_1 + a_{42}x_2$.
8		$y_4 = a_{41}x_1 + a_{42}x_2 + a_{43}x_3$. $y_5 = a_{52}x_2$.
9		x_2 is output. $x_4 = (b_4 - y_4)/a_{44}$. $y_5 = a_{52}x_2 + a_{53}x_3$.

Fig. 2.6.3. Solving a lower band triangular system with q = 4.

One can check that the computed x_1, x_2, x_3 and x_4 all have correct values. With this network we can solve an nxn band triangular linear system with band width w=q in 2n+q units of time.

2.7. Applications and Comments

2.7.1 Variants of the Algorithms

Rather than the basic algorithms presented above it is their variants that will be used mostly in practice. No attempt is given here for listing all the possible variants; it is important that the reader understands the basic principles of the algorithms so that he can construct appropriate variants for his specific problems.

We first note that although most of our illustrations are done for band matrices all the algorithms work for the regular nxn 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.

One can often reduce the number of processors required by an algorithm if the matrix is known to be sparse. For example, the matrices derived from differential equations by using finite differences or finite elements approximations are usually "sparse band matrices." These are band matrices whose nonzero entries appear only in a few of those lines in the band which 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 algorithms in Sections 2.3 and 2.6 can be reduced to the number of those diagonal lines which contain nonzero entries. This variant is useful for performing iterative methods involving sparse band matrices.

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

2.7.2. Convolution and Discrete Fourier Transform

There are a number of important problems which can be formulated as matrix-vector multiplication problems and thus can be solved rapidly by the algorithm in Section 2.3. The problems of computing convolutions 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 (cf. Fig. 2.7.1).

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

Fig. 2.7.1. The convolution of vectors a and x

On the other hand the 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 (cf. Fig. 2.7.2).

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

Fig. 2.7.2. The discrete Fourier transform of vector x.

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 problem 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 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 two 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. 2.3.2.

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 $\omega^{j^2 + ij}$ 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 which 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 ever 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. $R_t \leftarrow R_t \times \omega$.

2.7.3. The Common Memory Access Pattern

Note that all the algorithms given in this section retrieve and store elements of the matrix in the some order. (See Fig. 2.3.2, 2.4.2, 2.5.2, and 2.6.2.) Therefore, we recommend that matrices be always arranged in memory according to this particular ordering so that they can be accessed efficiently by any of the algorithms.

2.7.4. The Pivoting Problem

In Section 2.5 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' transformation 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' transformation can be implemented naturally on mesh-connected processors, although a pipelining implementation appears to be more complex.

2.8. Concluding Remarks

Research in interconnection networks and algorithms has been traditionally motivated by large scale array computers such as ILLIAC IV (see, for example, Kuck[5] and Stone [3]). The results presented in this section were, however, motivated by the advance in integrated circuit technology, though they are certainly applicable to parallel array computers. We have shown that many basic matrix computations can be done very efficiently by special purpose multiprocessors, which may be built cheaply using the current technology. The common feature of our algorithms is that their data flows are very *simple* and *regular*, and they are *pipeline algoritms*. We have discovered that some data flow patterns and interconnection schemes are fundamental for matrix computations. For example, the two-way flow on the linearly connected network is common to both matrix-vector multiplication and solution of triangular linear systems (Sections 2.3 and 2.6), and the three-way flow on the hexagonally mesh-connected network is common to both matrix multiplication and LU-decomposition (Sections 2.4 and 2.5). 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 processor postulated in Section 2.2. A careful design for this processor is desirabe since it is the work horse for all the devices presented.

For the important problem of solving a dense system of n linear equations in 0(n) time on nxn mesh-connected processors, we have improved upon the recent results of Kant and Kimura [13]. The basis of their results is an theorem on determinants which 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 sufficient for our algorithms in Section 2.5 that the matrix be symmetric positive-definite or irreducible diagonally dominant.

We end this section by noting that processor communication will likely continue to dominate the cost of parallel algorithms and systems. Communication paths inherently take more space and energy than processing elements. 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.

3. 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. We have seen that the organization of real estate on the silicon surface dictates a hierarchical communication system for any device which must support global communication. Such hierarchical communication exists in conventional computers only in a limited way. Are there new machine structures which communicate hierarchically, which support systems that consist of an arbitrary hierarchy of subsystems, and which can coordinate the activities of any number of submachines?

Binary Trees

Consider any number of processors physically arranged as a binary tree. Each processor has two subprocessors which 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 figure 8. 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 in 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 which communicates off the silicon chip to the outside world has large drivers and hence is able to drive off chip without suffering a severe performance penalty. Such a machine can thus be extended to a large number of individual chips and still maintain the full speed of the individual processors within it.

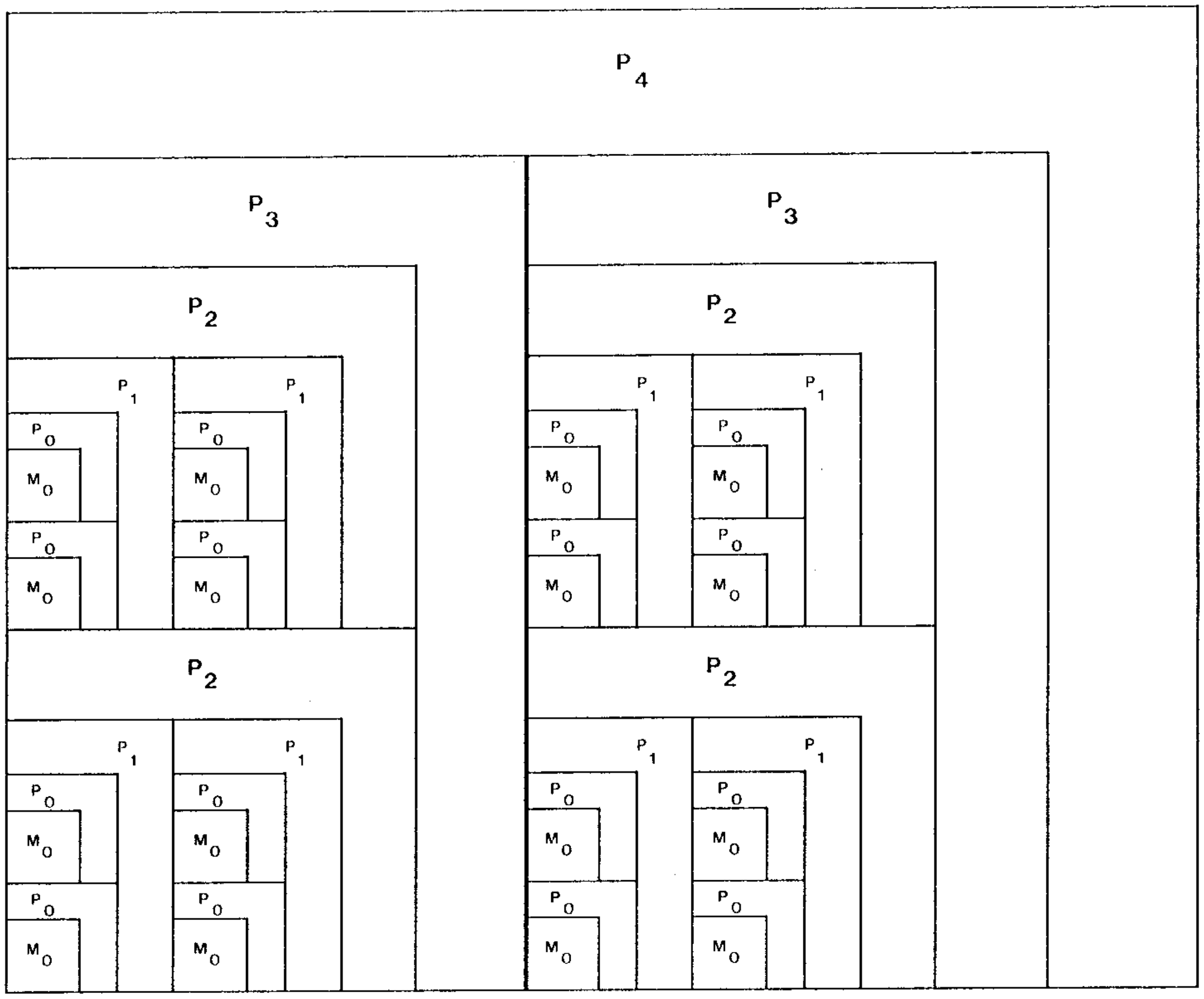
A conventional computer is a special case of this organization, consisting of a memory cell and a bottom-level processor. Also, there is another way to map a conventional computer onto a binary tree of processors. View the highest level processor as a cpu and load all subprocessors with programs that merely decode requests for the memory below them. Loaded with these programs, the structure between the two extreme levels becomes a memory decoder tree between a conventional cpu and its memory.

More importantly, this binary tree structure is a completely general, concurrent processing engine and can be used for problems decomposed in an arbitrary hierarchical way. If a problem requires more than two subprocessors at any level, a subtree of physical processors can be operated as one logical processor, matching the problem's structure. Algorithms for constructing logical processors of any size are given in the next section. The tree has inherent in it the ability for all processors to compute concurrently and hence has a vastly larger potential computing power than a conventional machine using a similar amount of silicon real estate.

Since the number of processors decreases exponentially with the level, the total bandwidth available, whether processing or communication, decreases exponentially with the level. Half of the total bandwidth of the system is concentrated at level 0, one quarter at level 1, one eighth at level 2, etc. A particular computation is well matched to such a processor if its bandwidth requirements are concentrated at the lowest levels. If an algorithm requires more communication at any level than the structure provides, it will not be able to take advantage of all the processing power of the structure. An extreme example of this sort is the von Neumann machine where all computation occurs at the highest level processor and the lower level processors are used only one at a time as an ordinary memory. Such a machine requires equal bandwidth at each level of the hierarchy and is an exponential waste of the resources of the machine.

It is also clear that such a structure is testable if a single processor is testable. Each supervisor merely loads a test program into its two subordinates and exercises them. Once it has established that both work correctly, it loads each with the program it just used to test them. A tree of N levels can thus be tested in N times the time necessary to test one processor.

It is difficult to predict how any radically different machine structure will perform in a real computing environment. Ideally, one should implement a number of complete systems, spanning a large range of user requirements, in order to gain experience with the strengths and weaknesses of any given scheme. Failing that, we can at least map certain algorithms onto our machine in the hope that they will shed light on its capabilities and its problems. Several such mappings are presented in the next section. We plan to develop others and we hope our readers will contribute still more for subsequent editions of this text.



P_4
P_3
P_3
P_2
P_2
P_2
P_2
P_1
P_0
M_0

Algorithms for the Tree Machine

Section contributed by Sally Browning, Caltech

A. A Word About Notation

The notation chosen to describe the processor tree and the algorithms that run in each node of the tree must emphasize that the number of different flavors of processors is small (usually one). That is, a few templates describe them all.

Secondly, we want to emphasize locality. The processor tree is interesting because each node is a powerful computing engine that can work independent from its neighbors. Our notation must be one that encourages self-sufficient modules.

We will use a modified version of the SIMULA syntax. The CLASS concept of SIMULA provides us a means of describing a template that will be instantiated as the nodes of the tree. We can designate individual procedures and data elements as either local to this node or visible to the outside world. And we can use recursion to indicate flow of execution through the tree.

Most importantly, though, SIMULA's CLASS construct is designed for expressing and enforcing locality. SIMULA is an object-oriented language, and, as such, encourages the programmer to think in terms of objects doing operations to themselves. The knowledge of the representation and meaning resides in the class, not in some omnipotent overlord. This is exactly the notation we need to describe the nodes of our tree.

Because we are describing highly concurrent algorithms we need to get around the sequential nature of SIMULA statements. We expand the meaning of the semicolon symbol. In conventional SIMULA, semicolon is used to terminate a statement. We use semicolon to make a statement about the execution as well. Read semicolon as "At this point, all statements in progress must be terminated before advancing to the next statement." Linefeed will be used to indicate syntactic end of the statement. In other words, *linefeeds are used to separate statements: semicolons are used to separate groups of statements which can execute concurrently.*

B. A Word About Branching Ratios

While the physical structure of our tree restricts each processor to two descendants, we can impose a logical structure that allows an arbitrary branching ratio. Each logical processor consists of several physical processors, enough to provide the desired number of offspring. A logical node with N children is built from N-1 physical nodes and is $\lceil \log N \rceil$ levels deep. Figure 1 shows some sample logical processors.

We can describe the process of mapping our logical structure onto the physical tree in SIMULA. We define *two* CLASSes: a node and a processor. A node represents the physical entity. It has exactly two descendants. A processor will refer to the logical entity, with an arbitrary number of children.

In the SIMULA definitions, N represents the number of descendants desired. As we build the logical node, we attempt to keep it balanced. That is, all available physical nodes on a given level of the tree will be used before a new level is added. Nodes on a given level are added to the logical processor from left to right, as in Figure 1. Note that CLASS Processor is a refinement of CLASS Node that knows how to choose one of N descendants.

```
CLASS Node(n); INTEGER n;
BEGIN
    REF(Node)left,right;

    !init code to build logical node;
    If n>2 THEN left:-NEW Node((n+1)//2);
    If n>3 THEN right:-NEW Node(n//2);
END of CLASS Node;


Node CLASS Processor;
BEGIN

    REF(Processor) PROCEDURE Son(s); INTEGER s;
    BEGIN REF(node)p;
        p:-IF s<=(n+1)//2 THEN left ELSE right;
        WHILE NOT (p IN Processor) DO
            p:-IF s<=(p.n+1)//2 THEN p.left ELSE p.right;
        Son:-p;
    END of PROCEDURE Son;

END of CLASS Processor;
```

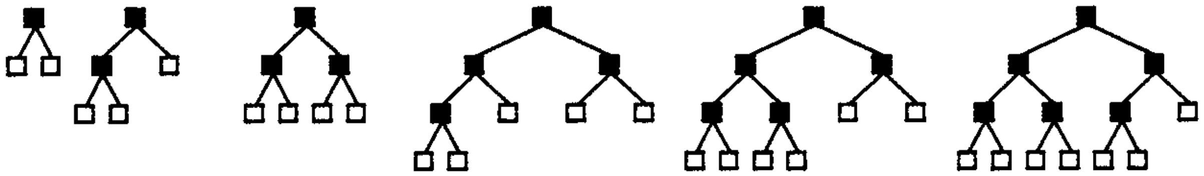


Figure 1. Logical Nodes (solid color) with Two to Seven Descendants

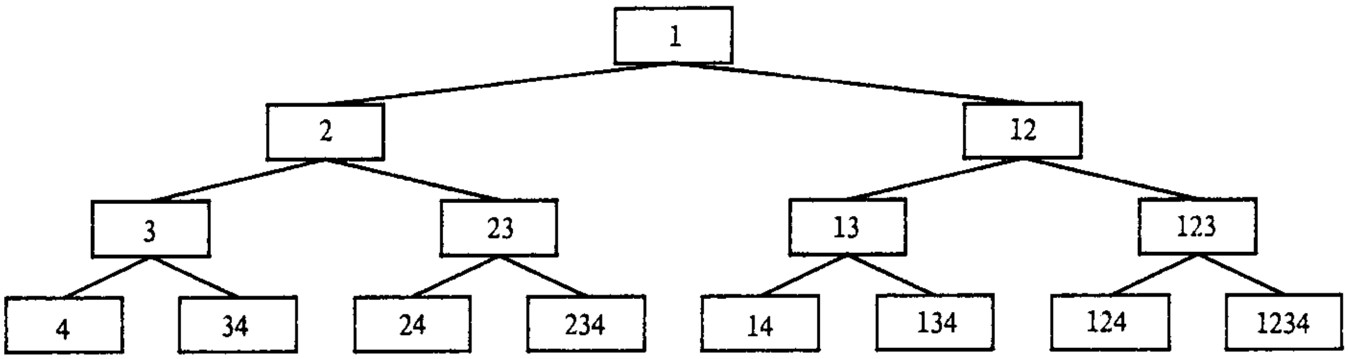


Figure 2. Systematic Generation of Subgraphs in a Graph of 4 Nodes

II. Algorithms with Polynomial Complexity

One of the traditional approaches to solving a problem that is too large or too complex when considered as a whole is to break the problem recursively into pieces that are manageable. The point is to apply as many concurrent processors to the problem as possible in order to reduce execution time. We will look at two algorithms that use this approach, sorting and matrix multiplication. While both of these problems are solved nicely on cellular arrays, it is instructive to map them onto a machine with different communication properties.

A. Sorting in linear time.

We use a binary tree with depth log N to sort N numbers. The sort is accomplished as a byproduct of loading the numbers into memory and then reading them out again. The numbers themselves are never in sorted order internally, but come out of the tree in the desired order.

This algorithm is an implementation of heap sorting, one of the well known techniques used in sequential machines [14]. It is a particularly interresting example because it illustrates a fundamental issue in concurrency. It is well known that sorting on a sequential machine can be done with O(NlogN) comparisons. Heap sorting requires O(N^2) comparisons, and has been considered inferior for that reason. However, it has been shown on very fundamental grounds that if communication is restricted to nearest neighbors, at least N^2 comparisons are required [17]. The apparent advantage of the O(NlogN) algorithms comes as a direct result of longer communication paths. It is also clear that no scheme will be able to produce an ordered set of numbers until all numbers to be sorted are loaded into the machine. For this reason, the best we can expect is to use N processors for O(N) cycles.

The algorithm that runs in each processor has a procedure for loading the tree called *Fillup* and a procedure invoked during the output cycle called *Passup*. *Fillup* keeps the largest number seen to date, and passes the smaller one to the right or left child, keeping the tree balanced by alternating sides. *Passup* returns this processor's current number and refills itself with the larger of the numbers stored in its descendants. This action is pipelined so that the largest number is always available in the root.

Below is a SIMULA description of the algorithm running in each processor. The variable *number* holds the number stored in this processor. The boolean symbol *empty* reflects the validity of that number. The boolean identifier *balanced* is used to keep the tree balanced as it is loaded.

```
CLASS processor;
BEGIN
    INTEGER number;
    BOOLEAN balanced,empty;
    REF(processor)left,right;

    PROCEDURE fillup(candidate); INTEGER candidate;
    BEGIN
        IF empty THEN
        BEGIN
            number: = candidate
            empty: = FALSE;
        END
        ELSE
        BEGIN
            IF candidate > number THEN    !swap;
            BEGIN INTEGER t;
                t: = candidate;
                candidate: = number;
                number: = t;
            END;
            IF balanced
            THEN left.fillup(candidate)
            ELSE right.fillup(candidate);
            balanced: = NOT balanced;
        END;
    END of procedure fillup;
    INTEGER PROCEDURE passupnumber;
    BEGIN
        passupnumber: = number;
        IF left= = NONE AND right= = NONE THEN empty: = TRUE !its a leaf;
        ELSE
        IF left.empty THEN
        BEGIN
            IF right.empty THEN empty: = TRUE    !left & right subtrees empty;
            ELSE number: = right.passupnumber;    !fill from right son;
        END
        ELSE
        IF right.empty THEN number: = left.passupnumber    !fill from left son;
        ELSE number: = IF left.number > right.number
            THEN left.passupnumber ELSE right.passupnumber;
            !take the larger of the two;
    END of procedure passupnumber;
    !init code;
    empty: = FALSE;
    balanced: = TRUE;
    !left and right set;


END of class processor;
```

B. Matrix Multiplication.

Suppose we have two NxN matrices to multiply together. By using the divide and conquer approach, we can break the problem down until we have N^3 problems that multiply two numbers (1x1 matrices) together. We reassemble the matrix on the way back up the tree.

We use the following rule to subdivide the problem:

Let A, B, and C be NxN matrices such that AB=C. We subdivide all three into four N/2xN/2 submatrices, e.g., A_{11}, A_{12}, A_{21}, A_{22}.

Then $C = (A_{i1} B_{1j} + A_{i2}B_{2j})$, i,j=1,2.

We will consider matrices of size $N=2^M$ without loss of generality. A tree to multiply two matrices of size 2^M will have M levels of processors that add two matrices together, M levels that split and assemble the matrix, and one level (the leaf nodes) that multiply two numbers together. Thus the tree is 2M+1 logical levels deep.

Each adder node has two descendants, and each split/assemble node has four descendants. Thus the physical structure will use two levels to simulate the 4-way branching, and the tree will, in fact, be 3M levels deep. That is, the tree is 3log N levels deep and therefore has N^3 leaf nodes. Thus a total of $2N^3$-1 processors are used in the computation.

Let us look at the communication requirements between nodes of the tree. The root node must be prepared to store the entire matrix. The adder nodes in level one (the root is level 0) will each deal with a quarter of the original matrix, as will the split/assemble nodes in level three. The further down the tree you go, the smaller the matrix the node must store and transfer.

However, note that each of the N^2 elements must travel the entire length of the tree and back again during the execution of the algorithm. While communication requirements are low at the leaves, they are extremely high (roughly N^2 numbers to receive, and $(N/2)^2$ to pass down to each descendant) at the root.

In the algorithm given below, the add operation takes N^2 time. By splitting this up among parallel processors by row (N of them) or element (N^2 of them) we can make this operation linear or constant in time. However, the problem is still limited by the split/assemble process that requires each element to travel the *height* of the tree. That is, the best time performance we can achieve with this algorithm is N^2logN.

We now present a SIMULA representation of a matrix and use it in the algorithm that follows. The algorithm uses two kinds of processors, the adders and the split/assemble nodes. Each matrix is divided into submatrices as follows;

$$\begin{array}{cc} 1 & 2 \\ 3 & 4 \end{array}$$

```
CLASS matrix(n); INTEGER n;
BEGIN

    INTEGER ARRAY val[1:n,1:n];

    REF(matrix) PROCEDURE quarter(select); INTEGER select;
    BEGIN REF(matrix)aq;    INTEGER i,j,k,l;
        aq:-NEW matrix(n//2);
        i:=j:=1;
        IF select=2 THEN j:=n//2+1
        ELSE IF select=3 THEN i:=n//2+1
        ELSE IF select=4 THEN i:=j:=n//2+1;
        FOR k:=1 STEP 1 UNTIL aq.n DO
            FOR l:=1 STEP 1 UNTIL aq.n DO
                aq.val[k,l]:=val[i+k-1,j+l-1];
        quarter:-aq;
    END of procedure quarter;

    REF(matrix) PROCEDURE compose(a,b,c,d); REF(matrix)a,b,c,d;
    BEGIN INTEGER i,j;
        FOR i:=1 STEP 1 UNTIL a.n DO
            FOR j:=1 STEP 1 UNTIL a.n DO
                val[i,j]:=a.val[i,j];
        FOR i:=1 STEP 1 UNTIL b.n DO
            FOR j:=1 STEP 1 UNTIL b.n DO
                val[i,j+n//2]:=b.val[i,j];
        FOR i:=1 STEP 1 UNTIL c.n DO
            FOR j:=1 STEP 1 UNTIL c.n DO
                val[i+n//2,j]:=c.val[i,j];
        FOR i:=1 STEP 1 UNTIL d.n DO
            FOR j:=1 STEP 1 UNTIL d.n DO
                val[i+n//2,j+n//2]:=d.val[i,j];
        compose:-THIS matrix;

    END of procedure compose;
END of class matrix;
```

```
CLASS processor(size);
BEGIN

    REF(matrix)mat;
    REF(processor)one,two,three,four;

    REF(matrix) PROCEDURE multiply(a,b); REF(matrix)a,b;
    BEGIN REF(matrix)c;
        c:-NEW matrix(a.n);
        IF c.n=1 THEN
            c.val[1,1]:=a.val[1,1]*b.val[1,1]
        ELSE
    c.compose(one.mult&add(a.quarter(1),b.quarter(1),a.quarter(2),b.quarter(3)),
            two.mult&add(a.quarter(1),b.quarter(2),a.quarter(2),b.quarter(4)),
            three.mult&add(a.quarter(3),b.quarter(1),a.quarter(4),b.quarter(3)),
            four.mult&add(a.quarter(3),b.quarter(2),a.quarter(4),b.quarter(4)));
        multiply:-c;
    END of procedure multiply;

    REF(matrix) PROCEDURE mult&add(a,b,c,d); REF(matrix)a,b,c,d;
    BEGIN REF(matrix)c1,c2; INTEGER i,j;
        c1:-one.multiply(a,b); c2:-two.multiply(c,d);
        FOR i:=1 STEP 1 UNTIL c1.n DO
            FOR j:=1 STEP 1 UNTIL c2.n DO
                c1.val[i,j]:=c1.val[i,j]+c2.val[i,j];
        mult&add:-c1;

    END of procedure mult&add;
END of class processor;
```

III. *Solutions to Nonpolynomial Problems.*

Complexity theory [9,10] has established a context within which it is possible to make certain statements concerning the inherent complexity of computations. These statements are universally couched in the terminology of sequential machines. There is, however, a class of problems for which the possibility of large scale concurrency has been addressed.

Consider a computation in which there are N conceptual steps. At each step, q alternative branches may be taken. Such a computation may be viewed as a tree with q^N possible outcomes. If at each step there is enough information available to decide which branch to take, a sequential machine will be able to complete the computation in KN cycles where K is the average number of cycles spent at each step. The dependence of the number of machine cycles upon the number of conceptual steps is thus linear. The problem is said to be *linear in N* or *of order N*, written O(N).

In many computations, not enough information has been generated by previous steps to determine which branch to take. Later steps will generate this information, but we cannot execute the later steps until after the earlier steps! In such cases, the sequential machine must simply try one branch at random. If it concludes after executing subsequent steps that the particular branch taken was wrong, it must *backtrack* to the original point, and try another route.

In a wild flight of fancy, we might become frustrated with this behavior and wish we had a machine which was so smart that it could tell if it was on the right path, even if there was no possibility of choosing such a path with the information at hand. It would make an arbitrary choice at each branch--and always be right! Such a machine cannot, of course, be built with real logic operating with real programs. However, we can imagine such a machine in much the same way we imagine a space ship traveling faster than the speed of light. Machines of this sort are called *nondeterministic*, since there is no way this behavior can be specified on rational grounds.

Returning to our problem, it is clear that a sequential nondeterministic machine could solve the problem in O(N) cycles. Problems which can be solved by such an imaginary nondeterministic machine in a number of cycles which is bounded by some fixed power of N are said to be *Nondeterministic-Polynomial* abbreviated *NP* [19,9].

It is quite clear that the behavior of a nondeterministic machine can be simulated by a set of concurrent deterministic machines. Each machine can simply follow a separate path through the tree. At the end, there will be q^N processors, representing each possible outcome of the computation. Although different problems will have different branching ratios (q) and different depths (N), all can be mapped onto the tree machine using techniques described earlier.

It has been shown that there is a class of problems of this sort where there are no shortcuts. Working one path through to the end gives no clue concerning the outcome of another path. Such problems are, in some sense, maximally difficult. They are called *NP-complete* problems.

A great deal of lore has developed concerning NP-complete problems. It has been shown that, in some sense they are all "equivalent" [18]. Suppose machine Y can solve a single kind of NP-complete problem. The equivalence property states that there is an algorithm which will run on an ordinary sequential machine in a polynomial number of cycles that transforms a description of any NP-complete problem into a description of a problem solvable by Y. If Y can solve its NP-complete problems in polynomial time, then it can be used to solve *any* NP-complete problem in polynomial time. If Y requires exponential time, any NP-complete problem will also require exponential time.

The methods we use to describe trees of different branching ratios to a binary tree machine are very similar to the methods used to map an NP-complete problem onto a machine that solves another. When a tree with branching ratio greater than 2 is mapped onto a binary tree, the depth of the tree increases. Mapping a tree with branching ratio less than 2 will decrease the depth. In a similar fashion, the algorithm that transforms NP-complete problems may increase the number of alternative branches (q) and decrease the number of conceptual steps (N) or vice-versa. Thus the mappings that establish the equivalence class of NP-complete problems are exactly like the mappings from trees of one branching ratio to another.

The theory that establishes the NP-complete equivalence class offers direct guidance in mapping such problems onto a highly concurrent structure. Because we can solve any one problem in our concurrent tree machine, and because we know a mapping from an arbitrary NP-complete problem into this one, we can solve the arbitrary problem.

The traditional approach to solving the class of problems that grow exponentially has been to recognize space or processing power as a limited resource. The problems have exponential time complexity because the solutions proceed sequentially. As VLSI becomes a reality, however, it is interesting to treat processors as an unlimited resource and look at the time complexity of these problems when they take advantage of concurrency. We emphasize, however, that while the time complexity is significantly reduced, problems require an exponential number of processors. If you solve a problem of reasonable size, you will use an enormous number of processors. In a later section, an example is worked for an NP-complete problem that grows as N^N. The problem uses a graph of 4 nodes, and our concurrent solution requires 95 processors. A graph of 10 nodes could use as many as $2 \cdot 10^{10}$ processors!

We will examine two NP-complete problems. The clique problem has time complexity of $O(2^N)$ when the possible cliques are considered sequentially. The color cost problem is $O(N^N)$. By taking advantage of the parallel consideration of possible solutions, using $O(2^N)$ and $O(N^N)$ processors respectively, we will present solutions to these two problems that take polynomial, in fact $O(N^2)$, time.

A. The Clique Problem.

A clique is complete subgraph. That is, given an undirected graph G, a clique C contained in G is a graph such that for all nodes n,m in C, there is an edge (n,m). Finding the largest clique in an arbitrary graph is an NP-complete problem.

Given a graph G with N nodes, numbered from 1 to N, we will consider each node sequentially and generate potential cliques. Ignoring the edges for a moment, a collection of M nodes leads to 2^M-1 potential cliques. This, interestingly enough, is the number of nodes in a binary tree of depth M. We will use this fact to generate the cliques in our graph incrementally.

Each level in the tree represents the addition of another node to be considered. Each processor at a given level will spawn two descendants. The left child will consider the subgraph consisting of the new node and all but the last node of the parent subgraph. The right child's subgraph will add the new node to the complete parent subgraph. In this manner, we generate all possible subgraphs for a graph of N nodes. Figure 2 is an example for N=4.

If each node stores an edge list, the tree can be pruned of subgraphs that are not cliques. The number of processors required is reduced, but the worst case behavior is identical. At most 2^{N-1} processors are required to solve the problem for a graph of size N. Our solution, which uses pruning, requires $O(N^2)$ time.

Each processor stores the edge list as a boolean matrix called *edge*, an integer *size* that holds the size *n* (number of nodes) of the clique this processor represents. An array called *clique* contains the numbers of the nodes that form the clique.

When a processor is activated, by a call to the procedure *Findclique*, it will already have a clique assigned to it. *Findclique*'s purpose is to generate cliques for its descendent nodes. It does this according to the method described above. That is, if the subgraph that contains the new node and all of the nodes in *clique* except the last one is a clique, it will be assigned to the left child. Likewise, if the addition of *node* to *clique* yields a complete subgraph, the right child will represent it. If either of the subgraphs is not complete, the descendant will not be generated.

The tree of all cliques is generated iteratively by considering each node of the graph in turn. In the main program given below, *p* is a reference to the root processor. Each processor in the tree will pass up the largest clique among its children. Thus the root returns the size of the largest clique known to date.

```
CLASS processor;
BEGIN
    REF(processor)left,right;
    BOOLEAN ARRAY edge[1:n,1:n];
    INTEGER ARRAY clique[1:n];
    INTEGER size;

    BOOLEAN PROCEDURE IsClique;
    BEGIN INTEGER i,j;
        IsClique: = TRUE;
        FOR i: = 1 TO size DO
            FOR j: = 1 TO size DO
                IF NOT edge[i,j] THEN IsClique: = FALSE;
    END of procedure IsClique;

    REF(processor) PROCEDURE FindClique(node);
    BEGIN INTEGER i; REF(processor)l,r;
        l: = r: = THIS processor;
        IF size = 0 THEN                !this is the root node;
        BEGIN
            clique[1]: = node
            size: = 1
            FindClique:-THIS processor;
        END
        ELSE
        BEGIN
            IF left.size = 0 THEN
            BEGIN
                FOR i: = 1 TO size-1 DO left.clique[i]: = clique[i]
                left.clique[size]: = node
                left.size: = size;
                IF NOT left.IsClique THEN left.size: = 0;
            END
            ELSE l:-left.FindClique(node);
            IF right.size = 0 THEN
            BEGIN
                FOR i: = 1 TO size DO right.clique[i]: = clique[i]
                right.clique[size + 1]: = node
                right.size: = size + 1;
                IF NOT right.IsClique THEN right.size: = 0;
            END
            ELSE r:-right.FindClique(node);
            IF l.size > size THEN
            BEGIN
                IF l.size > r.size
                THEN FindClique:-l
                ELSE FindClique:-r;
            END
            ELSE IF r.size > size
            THEN FindClique:-r
            ELSE FindClique:-THIS processor;
        END;
    END of procedure FindClique;

    size: = 0;
    !left and right set up correctly;
    !read in edge list;
END of class processor;
!main program to start it all up;
BEGIN REF(processor)largest; INTEGER i
    FOR i: = 1 TO n DO largest:-p.findclique(node);
END of main;
```

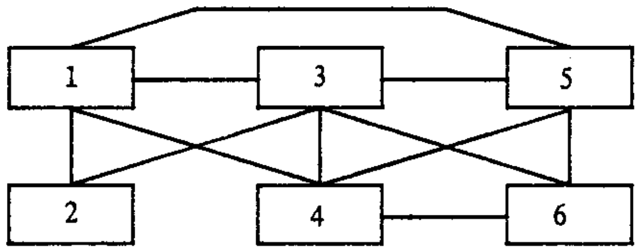


Figure 3. Sample Graph for Clique Problem

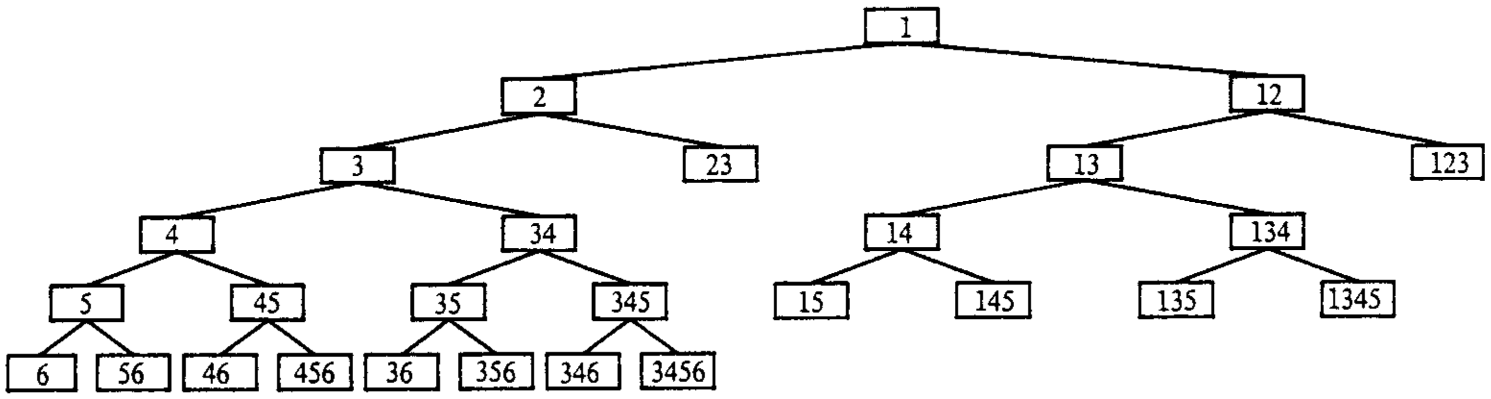


Figure 4. Tree built to find Cliques in Graph of Figure 3

Figure 3 gives a sample graph of six nodes. Figure 4 shows the processor tree that is built and used to find the cliques in the graph. The tree has height 6, and the largest cliques have 4 nodes. Each processor in the tree represents a clique in the graph.

B. The Color Cost Problem.

This NP-complete problem is an adaptation of the K-colorability problem. Given an undirected graph G of N nodes and a set of N colors, each with an associated cost, we want to find a minimum cost coloring of the graph such that no nodes sharing an edge are the same color.

There are N^N possible colorings of the graph. Evaluating them sequentially produces a solution in time $O(N^N)$. We present a parallel algorithm that requires $O(N^2)$ time and $O(N^N)$ processors.

In this problem we will make use of the ability to simulate arbitrary branching ratios on our binary tree. We will discuss the problem in terms of logical nodes with up to N descendants. An earlier part of this section describes the method of mapping logical structures onto the physical one.

As in the clique problem, each level in the processor tree represents the consideration of another node. That is, level one shows possible colors for the first node, level two colors the second node based on the choices made for at level one, and so on. We will describe the generation of the potential colorings.

Each node has an edge list called *edge* and a list of costs indexed by color number called *colorcosts*. There is an array called *coloring* that reflects the color choices for preceding nodes, and a boolean array called *colors* that is used to generate the possible colorings for this node.

The algorithm, given in procedure *color* begins by assuming that all colors yield valid colorings. The array *coloring* is used to eliminate those colors that have been used to color nodes that share an edge with this node. This reduced set of colors, all of which are legal colorings, is used to spawn descendants, one for each coloring of this node.

When the tree is N levels deep all the legal colorings have been generated. The leaf nodes calculate a cost for the coloring they represent, and each parent node takes as its cost the least cost among its children. Thus the minimum cost coloring is stored at the root.

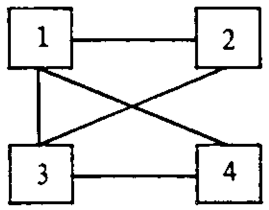
Here is the algorithm that will run in each processor.

```
CLASS processor;
BEGIN
    BOOLEAN ARRAY edge[1:n,1:n],colors[1:n];
    INTEGER ARRAY coloring[1:n],colorcosts[1:n];
    INTEGER cost;

    PROCEDURE color(node); INTEGER node;
    BEGIN INTEGER i;
        IF node > n THEN
        BEGIN
            cost: =0;
            FOR i: =1 TO node-1 DO cost: =cost+colorcost[coloring[i]];
        END
        ELSE
        BEGIN
            FOR i: =1 TO node-1 DO IF edge[i,node] THEN
                colors[coloring[i]]: =FALSE;
            FOR i: =1 TO n DO
                IF colors[i] THEN
                BEGIN
                    son(i).coloring[node]: =i;
                    son(i).color(node+1);
                END
                ELSE son(i):-NONE;
            cost: =maxcost;
            FOR i: =1 TO n DO
                IF (IF son(i) = NONE THEN FALSE ELSE cost > son(i).cost)
                THEN cost: =son(i).cost;
        END;
    END of procedure color;


    REF(Processor) PROCEDURE Son(s); INTEGER s;
    BEGIN REF(node)p;
        p:-IF s< =(n+1)//2 THEN left ELSE right;
        WHILE NOT (p IN Processor) DO
            p:-IF s< =(p.n+1)//2 THEN p.left ELSE p.right;
        Son:-p;
    END of PROCEDURE Son;
END of class processor;
```

Let us work a small example. We will use the graph and color set given in Figure 5. Figure 6 shows the colorings and costs arrived at by the algorithm. Each level of the tree represents a node of the tree. That is, if the root is level 0, the first node is colored in level 1, and level 4 represents potential colorings for the fourth node. Besides representing a part of a coloring, each node also contains the minimum cost coloring found among its decendent colorings.



Color	Cost
blue	0
green	1
red	2
black	3

Figure 5. Sample Graph and Color Table for Color-Cost Problem

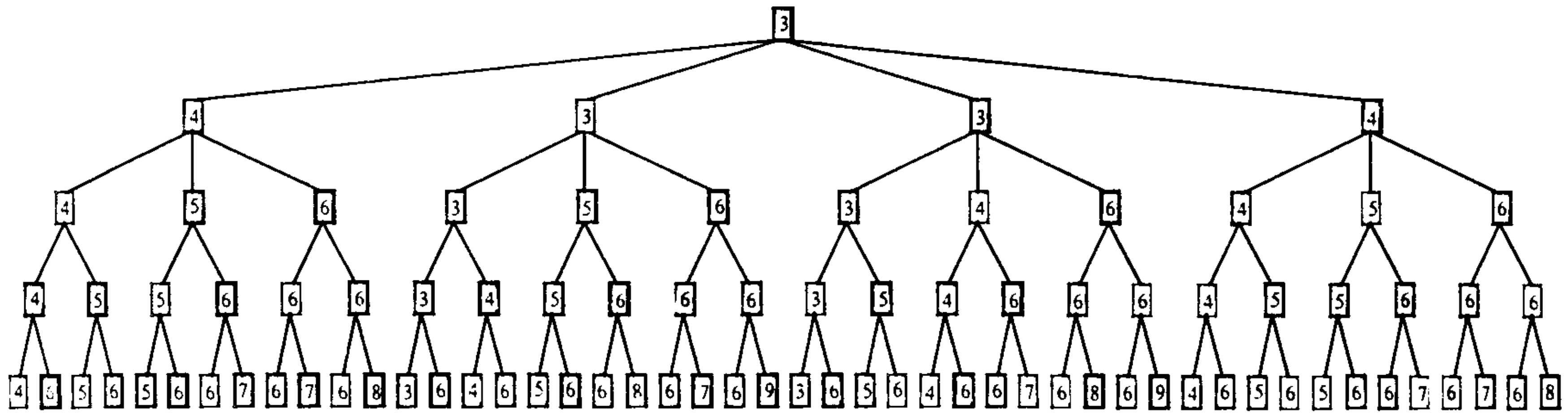


Figure 6. Color-Cost Tree for the Example of Figure 5

We see that there are two equivalent colorings that yield the minimum cost of 3. Coloring nodes (1,2,3,4) either (green,blue,red,blue) or (red,blue,green,blue) gives us a coloring with minimum cost.

V. Conclusions

The tree of processors we have described is a general computing structure. Each node in the tree is a processor with general computing capability. It is not designed with a specific problem or class of problems in mind.

The most dramatic results are achieved when the machine is applied to a problem that can take advantage of the concurrency the tree of processors provide. We have presented solutions to four problems that, in varying degrees, have this characteristic.

The four examples we have presented in this section can be summarized by citing the execution time and the number of processors required. Note that the total chip area of a tree machine is related to the number of processors.

Problem	Time	Processors
Sorting	N	N
Matrix multiplication	$N^2 \log N$	$2N^3$-1
Clique	N^2	2^{N-1}
Color cost	N^2	N^N

If an algorithm exhibits exponential growth, as do the clique and color-cost problems, the lower bound on time complexity is N. A tree with an exponential number of leaves will be O(N) deep. Again, our solutions do not realize this lower bound. The loading of the edge matrix is an $O(N^2)$ operation. Additionally, each node of the graph is considered in turn, and causes the traversal of a tree of depth up to N. This too is of $O(N^2)$ in time. Are there better algorithms that can achieve the lower bound complexity?

Because we are used to designing machines for a sequential environment, we do not yet understand the effect that concurrency will have on the conceptualization of problem solutions. An open question is to characterize those problems that can benefit from the concurrency provided by our tree of processors. Are the communication paths of the tree adequate for this class of problems? Can we design algorithms with the traditional programming notations, or does their sequential nature hide the concurrency? Can NP-complete problems be solved in O(N) time with an unlimited number of processors? What can be said about the concurrency of NP-complete problems in general? These are just a few of the interesting questions that arise from the study of a concurrent environment.

4. Highly Concurrent Structures with Global Communication

(Adapted from a paper by Carver Mead and Martin Rem [16])

This section presents an analysis of the constraints placed by physical laws on a VLSI system in which information must be communicated from any location to any other. The spectacular performance of cellular arrays on algorithms which map into regular structures leads us to ask if there is any possibility that a highly concurrent structure can be built which will act as a general purpose computational engine. Such generality must include the ability to transmit information over distances as large as required by the computation. Before describing such a machine we will analyze in detail the requirements that global communication places on the design of any computing structure.

There has previously been no adequate theoretical basis for optimizing the overall organization of systems implemented in the VLSI technology. Conventional complexity theory is inadequate because its measure of cost is the number of steps taken by a *sequential* machine to complete the computation. No account is taken of the size of the machine (and hence the time required for each step). Possible concurrency is ignored, thereby ruling out the most important potential contribution of the silicon technology. Traditional switching theory is also inadequate. While it provides a beautiful formalism for describing elementary logic functions, its optimization methods concern themselves with logical operations rather than communication requirements. Even in today's integrated circuits, the wires required for communicating information across the chip account for most of the area. Driving these wires accounts for most of the time delay and energy dissipation. In very large scale integrated systems, the situation becomes even more extreme. In this section, we describe a method by which the conceptual organization of a large chip can be analyzed, and a lower bound placed on its size, cycle time, and energy dissipation, before a detailed design is undertaken. The results of this analysis suggest rather general guidelines for the organization of all large integrated systems.

Metrics of Space, Time, and Energy

Physical Properties

Devices used to construct monolithic silicon integrated circuits are universally of the charge-controlled type. A charge Q placed on the control electrode (gate, base, etc.) results in a current I $= Q/\tau$ flowing through the device. The transit time τ is the time required for charge carriers to move through the active region of the device.

All times in an integrated system can be formulated as simple multiples of τ. For one transistor to drive another identical to it, a charge Q must flow through its active region, requiring time τ. If the capacitance C_L of the load driven is K times the gate capacitance C_g of the driving transistor, a time $K\tau = C_L/C_g \ \tau$ is required. Likewise, the elementary energy associated with the signal charge Q on the gate capacitance C_g is $E_0=C_gV^2/2$. A load capacitance KC_g requires an energy KE_0. Since wires have a minimum width, their capacitance is directly proportional to their length. Thus the energy required to transmit a signal from one point on the chip to another is proportional to the distance separating the two points. As the unit of length we employ the minimum spacing of two conducting paths. For the unit of time we choose the time it takes a minimum size transistor to charge a wire of unit length plus another transistor like itself. One unit of time is thus slightly larger than the transit time of a transistor

Advantages of hierarchical structures

We are considering large integrated systems in which it is necessary to communicate information throughout the entire system. As an example, consider a bit of information stored on the gate of a minimum size transistor in a random-access memory which must be communicated to the memory bus of a CPU. Since there are many words of data in the memory, there are many possible sources for each wire in the memory bus. Figure 1 illustrates two possible approaches to organizing such a bus. In the first approach, a transistor associated with each bit drives the bus wire directly. If the bus wire has a capacitance C_w, the time require to drive the bus wire is $t = \tau \ C_w/C_g$. In a typical computer memory C_w is many orders of magnitude larger than C_g, and the delay introduced by such a scheme is very long. Since C_w is proportional to the length of the wire, it is also porportional to S, the number of driver transistors connected to the wire and b_0, the spacing between transistors. Assuming most of the capacitance C_w is due to the wire itself;

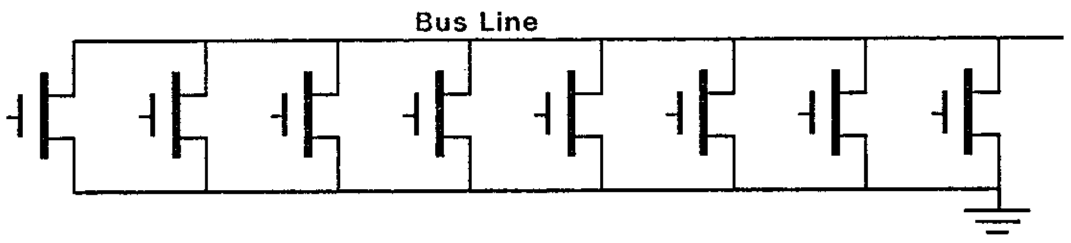


Fig 1a

A bus driven directly by memory cells

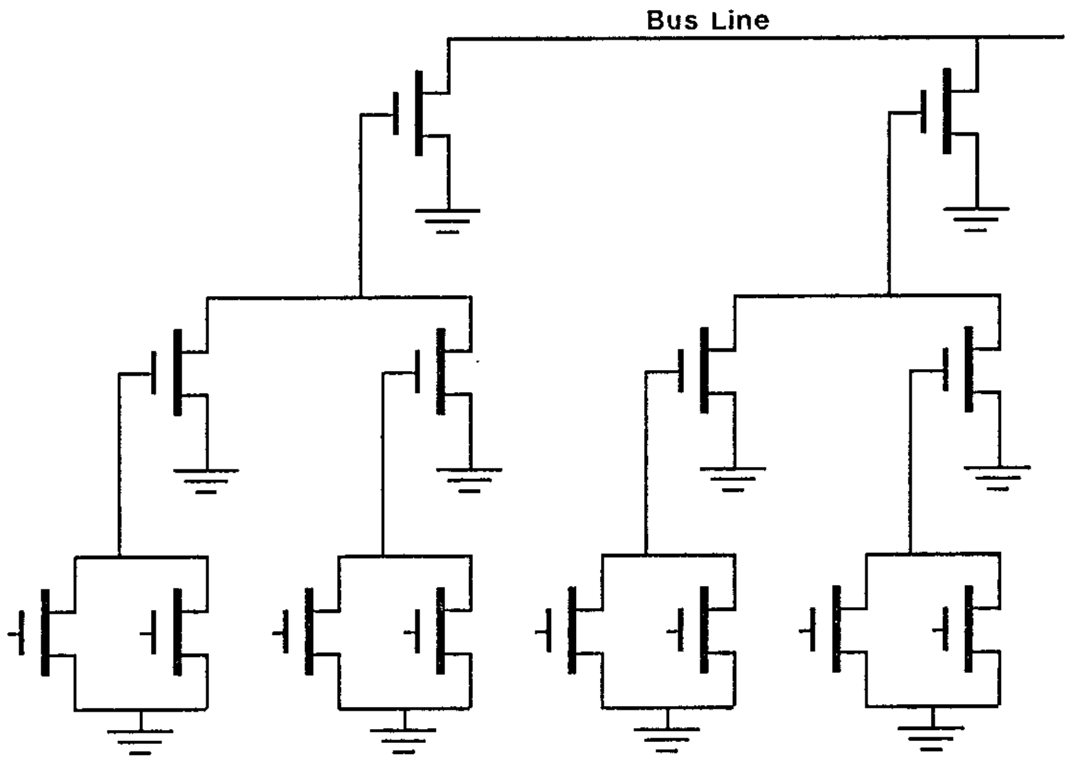


Fig 1b

A bus driver tree

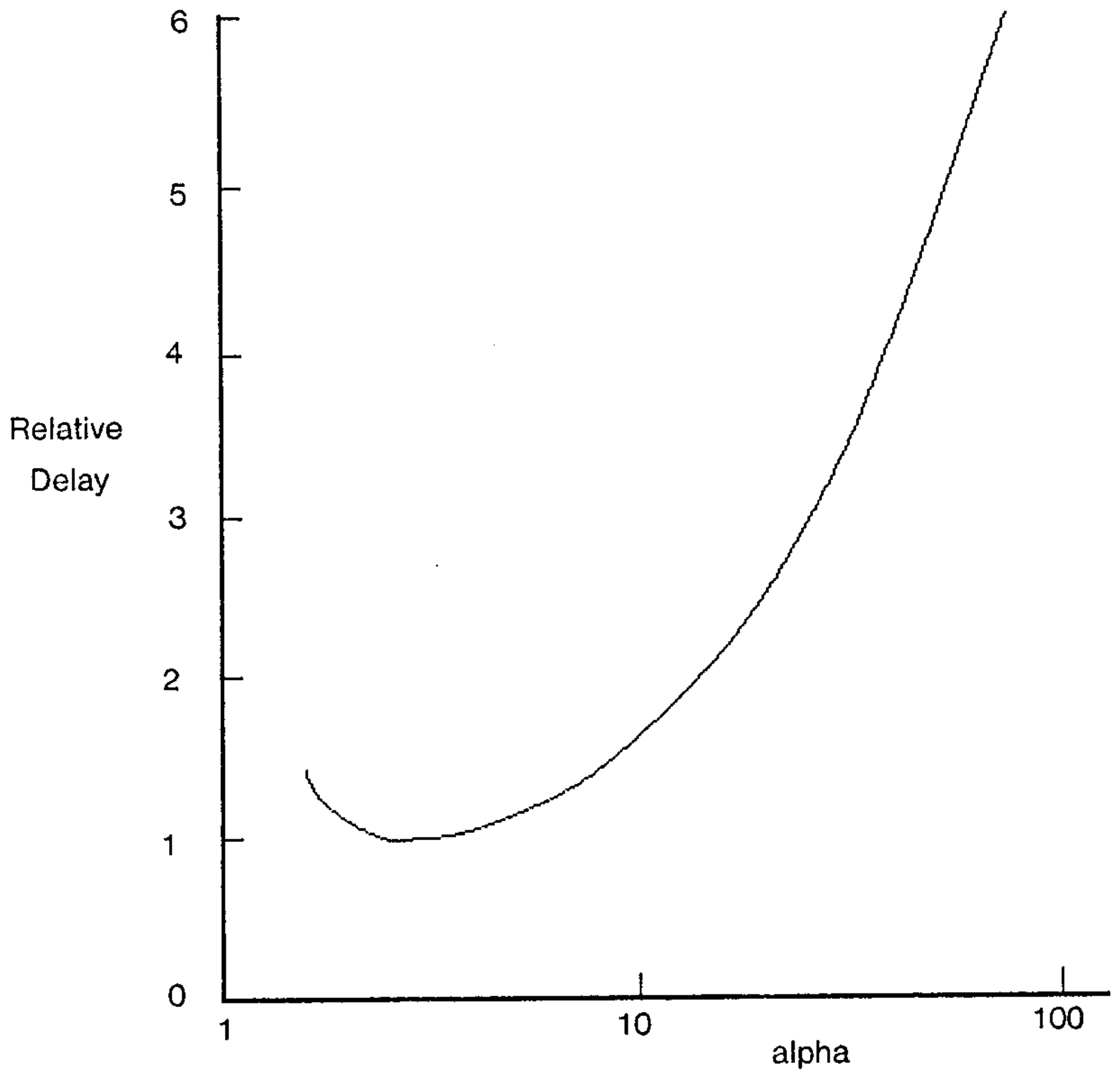


Fig. 2
Delay of a hierarchical structure as a function of alpha

$$t = b_0\tau S \qquad (1)$$

A second scheme is shown in Figure 1b. Here each transistor drives a wire only long enough to reach its neighbor. Each such wire is connected to the gate of a transistor twice as large as the transistor driving it. The arrangement is repeated upward until the top level where all sources have a path to the bus. In this scheme the delay in driving the lowest level wire is approximately $2\tau b_0$. The delay introduced by the wires at each level is the same, since each driver transistor is twice as large as those driving it. Hence the delay in driving the bus line is $2\tau N b_0$ where N is the number of levels in the structure. Since there are $S = 2^N$ transistors at the lowest level, the delay may be written:

$$t = 2\tau b_0 \log_2 S \qquad (2)$$

Comparing (2) and (1), we see that for large S the delay has been made much shorter by using a hierarchical structure.

A Cost Criterion

A hierarchy such as that shown in Figure 1b may use any integral number, α, of transistors driving each wire. We refer α as the *branching ratio* of the driver hierarchy. The driver transistors will in general be α times the size of those driving them. The delay for such a structure is $t = \alpha\tau b_0 \log_\alpha S = b_0\tau\ \alpha/\log_\alpha$, dependent upon the branching ratio of the hierarchy. This delay is plotted in Figure 2, normalized to its minimum value which is attained at $\alpha = e$.

While dramatic improvements in the performance of integrated structures can be achieved by a hierachical organization, a penalty is always paid in the area required for wires. In the simple case shown, a bus requiring one wire when driven directly requires $\log_\alpha S$ wires when organized as a hierarchy. For this reason it is not possible to optimize a design without a cost function involving both area and time. In this paper we will use the area-time product as an example of such a cost function. Other cost functions may be more appropriate under some circumstances. For the above simple example, the cost function is area * time = $b_0\tau(\log S)^2\ \alpha/(\log\alpha)^2$. This cost is minimized for $\alpha = e^2 \approx 7.4$.

Hierarchical computing systems

The analysis given above suggests a very general structure for computing systems. Lowest level cells are grouped together into modules in such a way that α cells drive their outputs onto an output wire. Each output wire is connected to a driver transistor which is α times as large as those driving the wire. Modules are grouped in such a way that α of those modules drivers are connected to an inter-module communication wire. This wire in turn is connected to a driver transistor α^2 times as large as the lowest level transistors. This process is continued until the appropriate size system has been realized. Notice that the area of the driver transistor for each wire in such a structure is proportional to the area of the wire. For this reason, we compute only the area of the wires. The drivers somewhat enlarge the unit of wire area, but do not change the functional form of the solutions.

Random-Access Memory - an example

In this section we discuss the design of a large of a random-access memory (RAM) of S bits. We will apply a rigid structural dicipline to our design, and compute the cost and performance of the resulting memory.

Organization of the RAM

We organize the RAM in a hierarchical fashion. The elements of level 0 are the bits themselves, each bit consisting of two crossing wires: a select wire and a data wire. When the select wire is asserted, it puts its contents on the data wire. We group α^2 bits into an α x α square to form a module of level 1. If the width of an element (a bit) is b_0 the elements have to drive wires of length αb_0. A module on level 1 consists of an array of horizontal select and vertical data wires, constituting the α^2 bits of level 0, and some additional logic and wires at the side. We group again $\alpha^2$of these modules into a square to form a module of level 2, etc. Figure 3 shows three levels of the hierarchy for $\alpha=4$.

To study the memory in more detail we look at a module of level i (Figure 4). We describe how one extracts one of its α^{2i} bits. In order to select one bit of storage $2i\log\alpha$ address wires are required. We run $i\log\alpha$ of them, called the row address wires, vertically along the side of the module and the other $i\log\alpha$, the column address wires, horizontally. Its α^2 submodules are organized into α rows of α submodules each. When the select wire of the module is asserted $\log\alpha$ of the row address wires are used, by the decoder, to select one of the α rows of submodules; the select wire running through that row is asserted. The other $(i-1)\log\alpha$ row address wires are run horizontally into each of the α rows of submodules, where they serve as

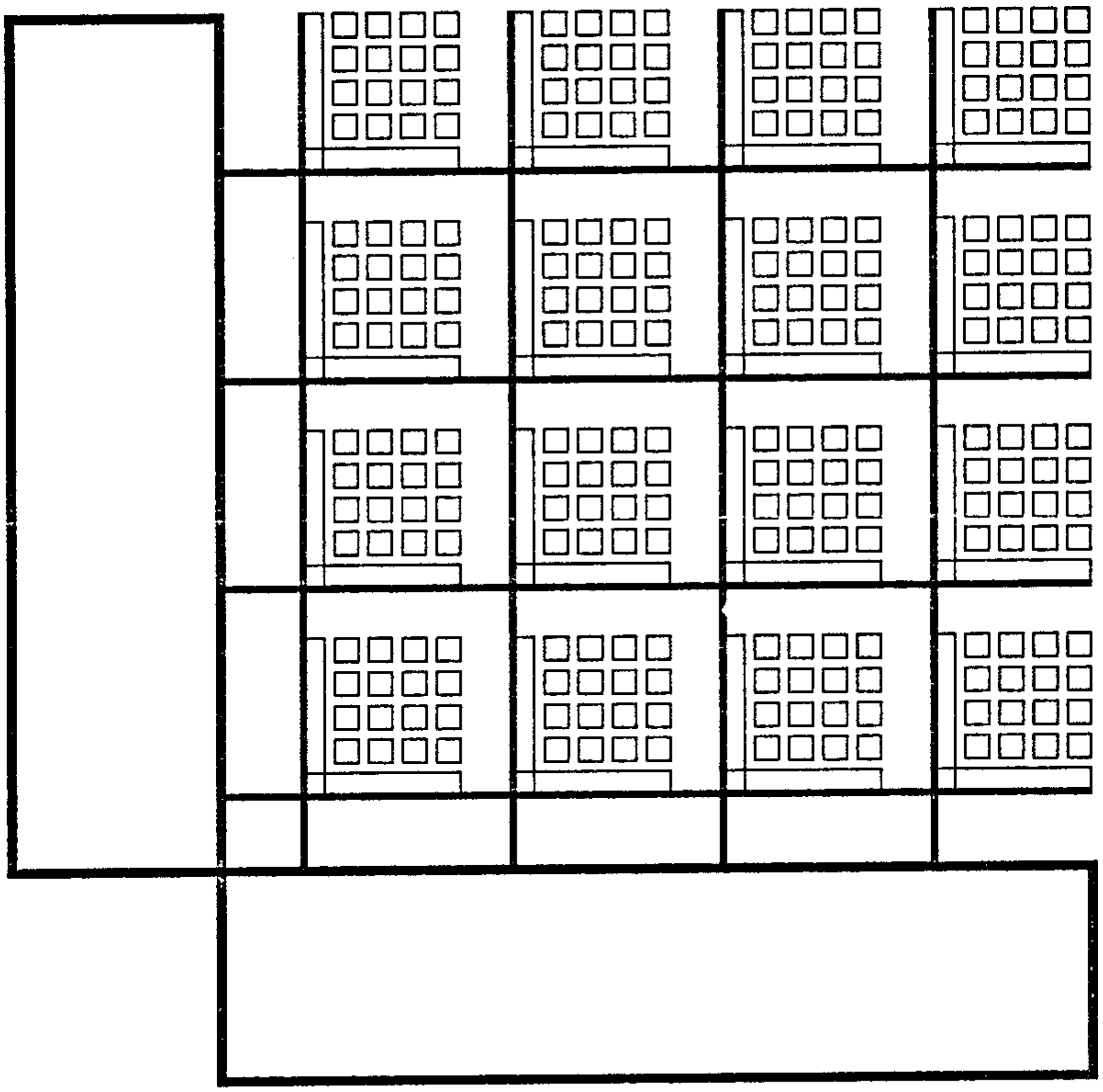


Fig. 3

Three Levels of a Memory Hierarchy with Alpha = 4

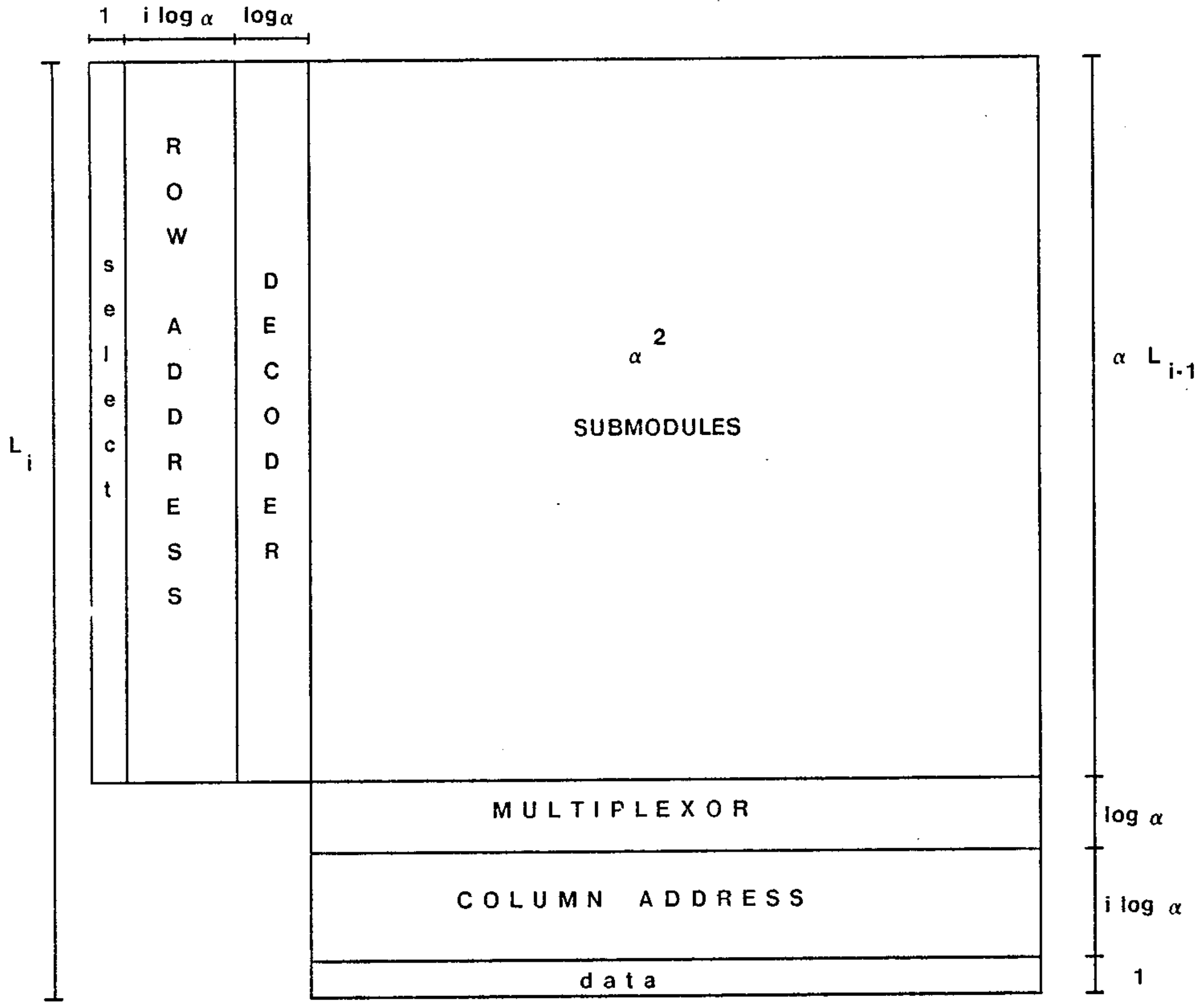


Fig. 4

A RAM module of level i (i > 0)

column address wires for the submodules. Of the $i\log\alpha$ column address wires $(i-1)\log\alpha$ are run vertically into each of the α columns of submodules, where they serve as row addresses. The other $\log\alpha$ address wires are used by the multiplexor to select one of the α data wires coming out of the columns of submodules. The signal on the selected data wire is driven onto the data wire of the module itself.

If we wish to have a memory of S bits with N+1 levels (level 0 through N) we choose $N = \log S/2\log\alpha$, or $S = \alpha^{2N}$. A hierarchical structure results which contains S bits from which we can extract one bit at a time. If we want the word length to be W we employ W of these structures in parallel: to select one word we select one bit in each of the W hierarchies.

Area of the RAM

Figure 4 alllows us to compute the size of a RAM. Let L_i denote the width of a module of level i, then we have the following recurrence relation:

$$L_0 = b_0$$
$$L_i = i\log\alpha + 1 + \log\alpha + \alpha.L_{i-1}$$

The solution to the above relation is

$$L_i = \alpha^i b_0 + (\alpha^i-1)/(\alpha-1) + (2\alpha^{i+1}-\alpha^i-\alpha)/(\alpha-1)^2 - i+1/\alpha-1)\log\alpha .$$

Rather than the width itself we are interested in the width per bit. In one direction, horizontal or vertical, module i has α^i bits; we therefore compute L_i/α^i.

$$L_i/\alpha^i = b_0 + 1/(\alpha-1) + 2\alpha-1/(\alpha-1)^2 \log\alpha - 1/(\alpha-1)\alpha^i [(\alpha/\alpha-1 + 1+i) \log\alpha + 1] \qquad (3)$$

An interesting property of the width per bit, as expressed by (3), is that its limit for $i \rightarrow \infty$ is finite.

$$\lim_{|i\rightarrow\infty} L_i/\alpha^i = b_0 + 1/\alpha-1 + 2\alpha-1/(\alpha-1)^2 \log\alpha \qquad (4)$$

This means that the width per bit L_i/α^i is bounded from above by (4) independent of the number of levels of a RAM. Expression (3) converges in an exponential fashion towards its limit: for small values of i (3) is already very close to (4). We, therefore, use (4) as the width per bit for a RAM; its square is then the area per bit. By dividing the area per bit by the bit area b_0^2

we obtain the total area per bit area for a RAM. Figure 5 shows this quotient as a function of α for four different values of b_0. It gives the overhead factor in the area that is due to the wires. A memory chip will be larger by this factor than the area of its level 0 cells alone. For a memory of 64K bits with $N=2$, α should be 16. Expression (4) is then equal to $b_0 + 0.6$. This shows that in 2-level 64K dynamic MOS memories, for which b_0 lies between 1 and 2, roughly half of the area will be occupied by wires.

One may wonder why we have not discussed the area that is consumed by the wires for power and ground. The reason is that these wires can be thought of as increasing only the width b_0 of each bit; they do this by an amount that is roughly independent of α, as is shown in the following analysis.

For simplicity we assume that the wires for power and ground run in opposite directions, say parallel to the data and select wires. We compute how much one of them contributes to the width of a module i. The width of a power or ground wire is proportional to the number of bits served by it. Let the width of the highest level be u, given S and the design of the lowest level memory cell this parameter is easy to compute. The width of the wire in a module on level i is proportional to the current it must supply and is hence u α^{2i}/α^{2N}. In one direction, horizontal or vertical, there are α^{N}/α^{i} such modules. The total contribution of all modules on level i is thus u α^{i}/α^{N}. Taking the sum of this expression for $i=0,1,...,N$ yields u/α^{N} α^{N+1}-$1/\alpha$-$1 \approx u$ α/α-1. There are u/S bits in one direction, the increase of the bit width, due to power and ground, is therefore

$u/S \ \ \alpha/\alpha\text{-}1,$

which is roughly equal to u/S.

We are interested in the optimal choice of α, but to make that choice we will have to look at the access time, which also depends on α.

Access time of the RAM

Each element of level 0 drives a wire of length αb_0 to reach the periphery of its module on level 1; this takes time αb_0. Each module on level 1 drives in the same amount of time a wire that is α times longer to reach the periphery of its module on level 2, etc. With N being the level of the highest module, the time required to extract one bit of storage adds up to αb_0 N. We use

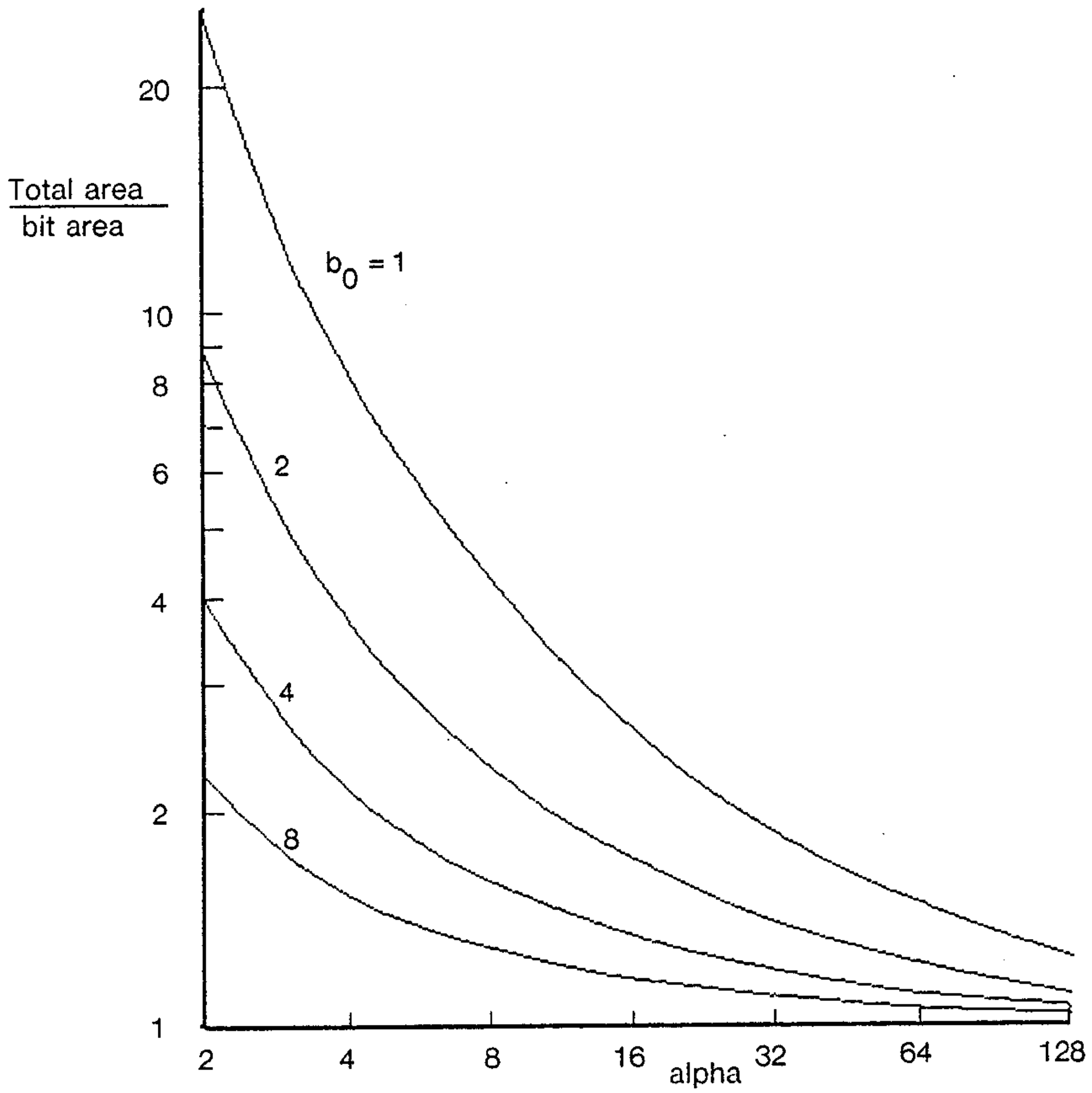


Fig. 5 Total area per bit of a RAM as a function of alpha

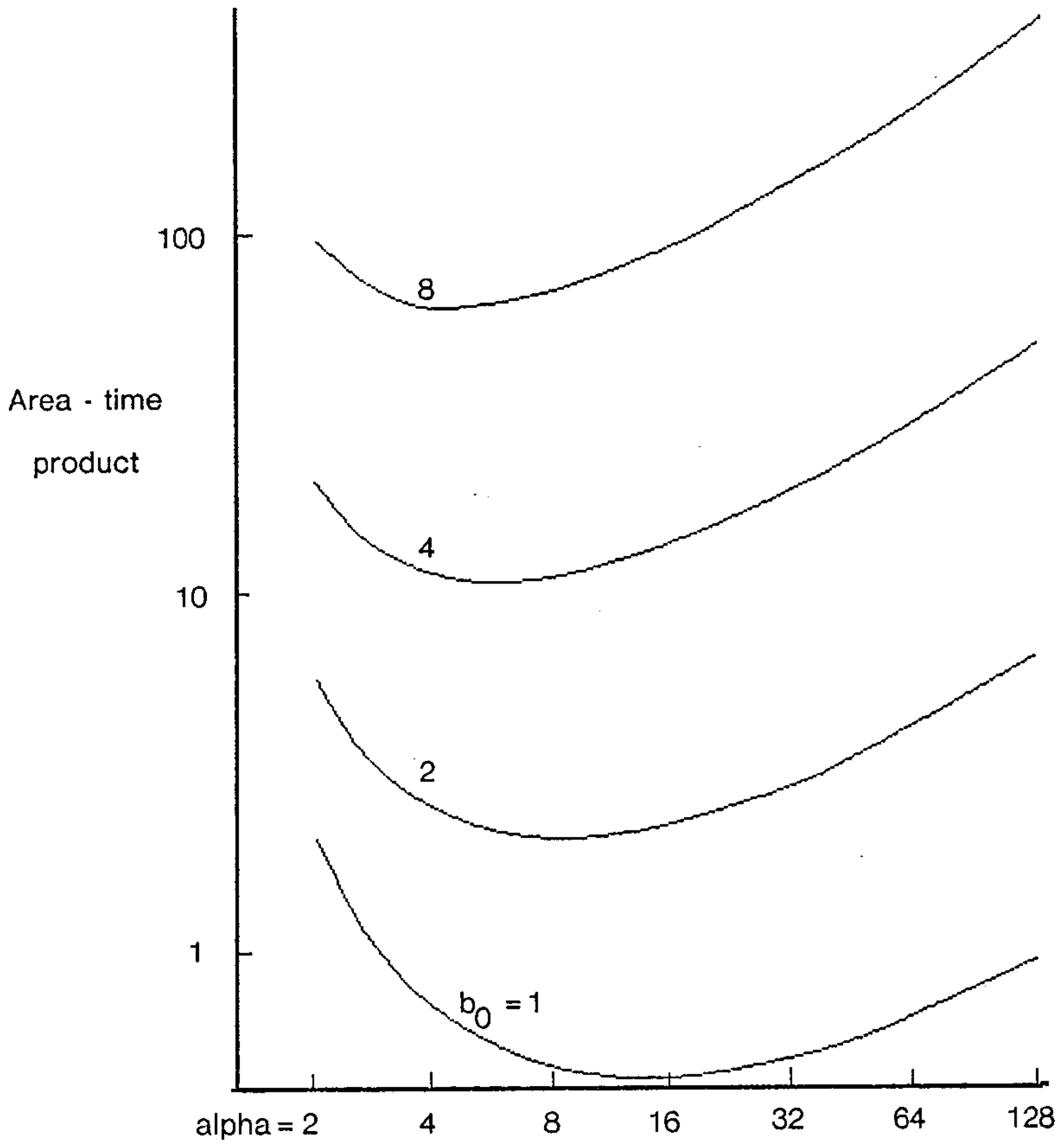


Fig. 6 Area-time product of a RAM as a function of alpha

this figure as the access time. For a RAM of S bits the access time is then αb_0 logS/2logα.

Cost of the RAM

We take the product of the area and the access time as the cost function of the RAM. A RAM of S words of logS bits each has the following area-time product.

$$(b_0 + 1/\alpha\text{-}1 + 2\alpha\text{-}1/(\alpha\text{-}1)^2 \log\alpha)^2 (\alpha b_0/2\log\alpha) \; S\log^2 S \qquad (5)$$

Figure 6 shows (5), normalized with respect to $S\log^2 S$, as a function of α for different values of b_0. One notices that for increasing bit sizes the branching ratio of the hierarchy should decrease. Because of the simplicity of their storage cells, dynamic memories have b_0 between 1 and 2. Static memories require a cross-coupled structure and hence a larger b_0--typically 3 to 4. For optimal designs, static memories should therefore have a smaller α than dynamic ones. For dynamic MOS memories the optimal choice for α lies between 8 and 16, for static MOS memories between 4 and 8. "Smart memories", structures in which part of the processing task is distributed over the memory cells, have quite large level 0 modules containing an entire processor. They should therefore have small branching ratios and hence relatively deep hierarchies. Current commercial memory chips are designed with $\alpha \approx 100$ at the lowest level. This value approximately minimizes the product of the access time and the exponential of the area. Designs of this sort reflect the near exponential dependence of yield on chip area in the early, low-yield phase of a device's production history. However, near its production peak, the area-time product is closer to a realistic cost function. This shift in production economics suggests that redesigns of high-volume devices should be done using smaller values of α than initial designs.

Energy per Access

In real systems, the cost of power, cooling, and electrical bypassing often exceeds the cost of the chips themselves. Hence any discussion of the cost of computation must include the energy cost of individual steps of the computation process. In a RAM, each access costs an energy proportional to the length of the wires which must be charged or discharged during a given cycle. Consider a RAM such as that shown in Figure 4. At the highest level (level N) such a device has $S = \alpha^{2N}$ bits. In each cycle logS address wires of length L_N will in general change state. In addition one horizontal select line, α vertical data lines, and one multiprocessor output line (all of length L_N) will change state. Thus at level N, the energy expended per access will be

$$E_N \sim L_N\,[\log S + \alpha + 2]$$

At level N-1, $2\log\alpha$ fewer address wires will be needed. Since only one select line will be active, only α of the α^2 submodules will be active. Each submodule contains wires approximately $1/\alpha$ as large as those at level N.

Thus the total energy per access is

$$E_T \sim L_N\,[\log S(1 + 1 - 2\log\alpha/\log S + 1 - 4\log\alpha/\log S + . \quad . \quad .) \quad + \alpha+2]$$

This expression evaluates to

$$E_T \sim L_N \log S/\log\alpha\;[\log S/4 + (\alpha+2)/2]$$

Using the b_0 values from (4), the energy per access of any given size RAM may be evaluated. The results of such an evaluation for a 65K bit RAM are shown in Figure 7.

These curves suggest that considerably less power would be required if memory chips, even of current size, were built with smaller submodules and smaller α.

General Method of Analysis

We have presented a general method for analyzing the cost and performance of recursively defined VLSI structures. Parameters of any such structure may be optimized with respect to some combination of access time, area, and energy.

The results of this study indicate that as more processing is available in each module at level zero, b_0 will be larger and the optimal value of α will decrease. A system with $\alpha = 4$ would seem to be appropriate for structures in which substantial processing is commingled with storage.

Very general arguments were used to generate the basic recursive structure. For that reason it appears that a very large fraction of VLSI computing structures will be designed in this way. The way in which the area, time, and energy measures were established should make it clear how to apply these techniques to other recursively defined computing structures.

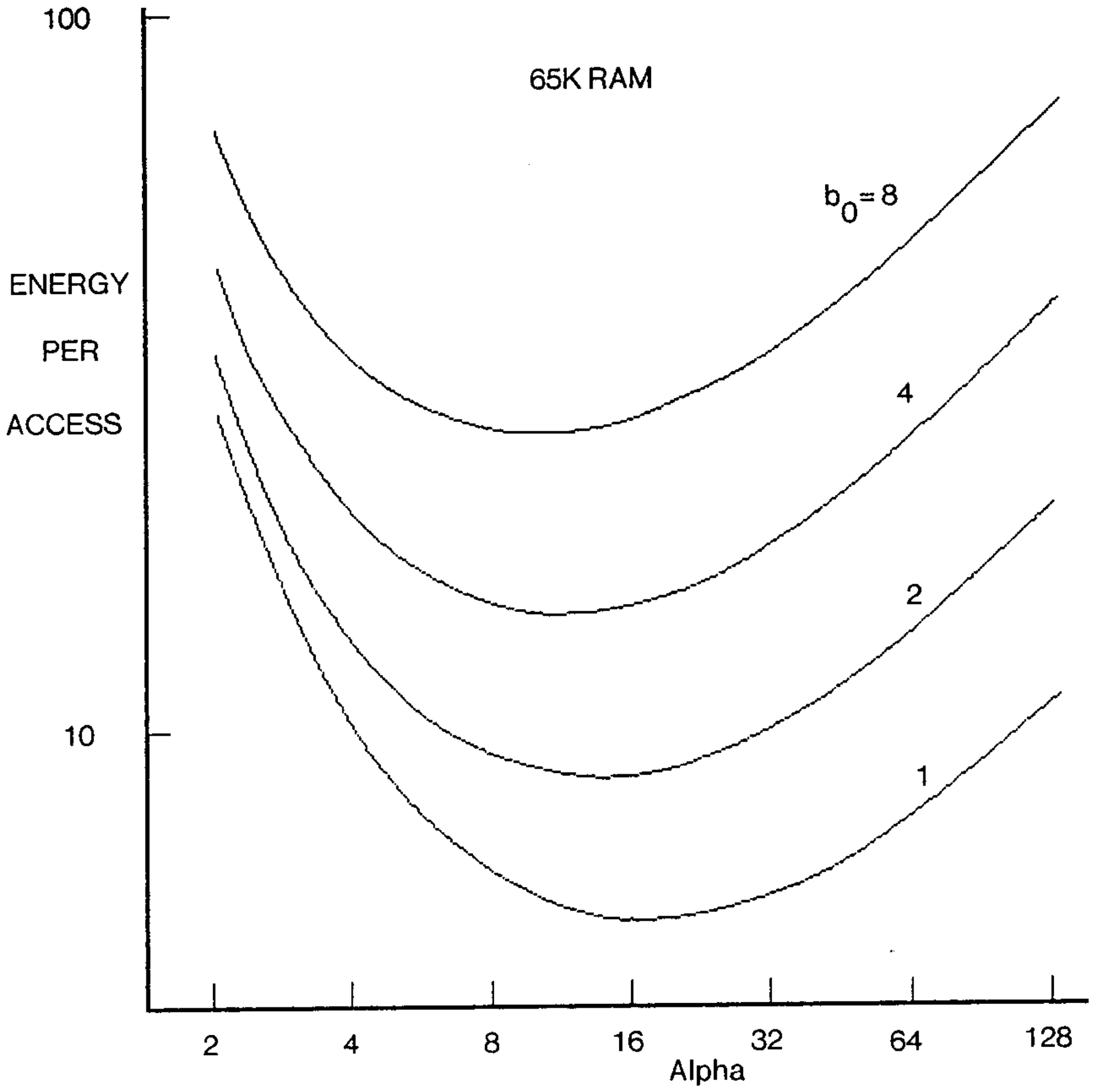


Fig. 7 Energy per Access as a function of Alpha

5. Challenges for the Future

We have seen that it is possible to construct general-purpose computing engines that exploit tremendous concurrency if computations are properly matched to the machine. The vast quantity of concurrency available in such machines can be an enormous help with the computing tasks we face. However, to date we have no formal way of making the possible concurrency in any given calculation apparent or finding if we have come close to the possible concurrency inherent in the computation.

The future of concurrent processing is bounded in part by our ability to escape the strong hold that the conventional sequential machine exerts on our thinking. We must approach problems with concurrency in mind, recognizing that communication is expensive and that computation is not. Progress in these endeavors will surely increase when some VLSI computers of the sort we have illustrated in this chapter begin to appear. When the effort of casting the problem as a structure of concurrent processes is rewarded by a tangible increase in performance, the incentive to design concurrent algorithms will surely increase.

The tools that we use to design and implement concurrent processes are primitive. We are badly in need of notation or language that expresses the power and constraints of highly concurrent machines. Whether such machines are general- or special-purpose, a natural way is needed to map problems onto them. Only in this way will it be possible for applications to find their way into execution in this new computing environment rapidly. In addition we need a method of formally proving the correctness of algorithms mapped onto such machines; it is not possible for human programmers to keep track of the exact relationship of the enormous number of tasks executing on such a machine. An ideal notation would allow expression of only those operations which are free of obvious fatal errors such as deadlock. Only one such notation is known to the authors at this writing, that of the *Associons* by Martin Rem [15].

Perhaps the greatest challenge that VLSI presents to computer science is that of developing a theory of computation that accommodates a more general model of the costs involved in computing. Can we find a way to express computations that is independent of the relative costs of processing and communication, and then use the cost properties of a piece of hardware to derive the proper program or programs? The current VLSI revolution has revealed the weaknesses of a theory too solidly attached to the cost properties of the sequential machine.

References

1. I.E. Sutherland and C.A. Mead, "Microelectronics and Computer Science," *Scientific American*, vol. 237, no. 9, September 1977, pp. 210-228. This article is a readable and inspiring call to abandon conventional computing structures.

2. C.G. Bell and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, New York, 1971. This book gives a general overview of computer structures, describes the PMS and ISP notations for describing the structures, and collects a number of papers concerning alternative computer structures. A second edition is about to emerge that will greatly expand the material.

3. H.S. Stone, ed., *Introduction to Computer Architecture*, Science Research Associates, Chicago, 1975.

4. P.H. Enslow, Jr., "Multiprocessor Organization--A Survey," *Computing Surveys*, 9, 1, March 1977, pp. 103-129. This entire issue of *Computing Surveys* is devoted to surveys of parallel processors and processing.

5. D.J. Kuck, "A Survey of Parallel Machine Organization and Programming," *Computing Surveys*, 9 (1977), pp. 29-59.

6. C.V. Ramamoorthy and H.F. Li, "Pipeline Architecture," *Computing Surveys*, 9, 1, March 1977, pp. 61-102.

7. K.J. Thurber and L.D. Wald, "Associative and Parallel Processors," *Computing Surveys*, 7, 4, December 1975, pp. 215-255.

8. B.A. Trakhtenbrot, *Algorithms and Automatic Computing Machines*, Heath, Boston, 1963. A delightfully readable introduction to algorithms, theory of computation, and unsolvability.

9. A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1974. A survey of algorithms designed using conventional models of computation. Includes discussion of divide-and-conquer, NP-complete problems.

10. R.E. Tarjan, "Complexity of Combinatorial Algorithms," *SIAM Review*, 20, 3, July 1978. A most readable discussion - highly recommended.

11. C.D. Thompson and H.T. Kung, "Sorting on a Mesh-Connected Parallel Computer," *Communications of the Association for Computing Machinery*, 20 (1977), pp. 263-271.

12. G.H. Barnes, R.M. Brown, M. Kato, D.J. Kuck, D.L. Slotnick, and R.E. Stokes, "The ILLIAC IV Computer," *IEEE Transactions on Computers*, C-17 (1968), pp. 746-757. Also appears in (2).

13. R.M. Kant and T. Kimura, "Decentralized Parallel Algorithms for Matrix Computation," Proceedings of the Fifth Annual Symposium on Computer Architecture, Palo Alto, California, April 1978, pp. 96-100.

14. D.E. Knuth, *The Art of Computer Programming*, vol. 3, "Sorting and Searching," Addison-Wesley, Reading, Massachusetts, 1973.

15. M. Rem, "Associons and the Closure Statement". MC tract 76, Mathematical Centre, Amsterdam, 1976.

16. M. Rem and C.A. Mead, Proc. 1978 USA-Japan Computer Conf. (in press)

17. H.B. Demuth, "Electronic Data Sorting" Ph.D. Thesis (Stanford University: October, 1956)

18. S.A. Cook, "The complexity of theorem-proving procedures," *Proc. 3rd Annual ACM Symposium on Theory of Computing*, pp. 151-158, 1971. Shows equivalence of np-complete problems.

19. H.R. Lewis and C.H. Papadimitriou, "The Efficiency of Algorithms," *Scientific American*, vol. 238, no. 1, January 1978, pp. 96-109. General introduction to NP-complete problems.