Direct VLSI Implementation of Combinatorial Algorithms

L.J. Guibas, H.T. Kung, and C.D. Thompson
Carnegie-Mellon University
Pittsburgh, Pennsylvania

Abstract

We present new algorithms for dynamic programming and transtive closure which are appropriate for very large-scale integration implementation.

CALTECH CONFERENCE ON VLSI, January 1979

0. The VLSI Model of Computation

The purpose of this paper is to give two examples of algorithmic design suitable for direct implementation in VLSI (very large-scale integration). We show new algorithms for two important combinatorial problems, *dynamic programming* and *transitive closure.* In our design we attempt to meet the challenge offered by the new VLSI technology by taking account of its true costs and capabilities. The algorithms of this paper meet the goals of modularity and ease of layout, simplicity of communication and control, and extensibility. These goals are of paramount importance in all VLSI designs.

Modularity and ease of layout: The design of significant system components using large-scale integration is notoriously expensive. Two major factors of the design cost are the difficulty of designing each chip, and the number of different chips that must be designed. A modular design decreases both cost factors, as well as facilitating chip and system layout. An ideal structure for VLSI has a large number of identical modules organized in a simple, regular fashion. The resulting ease of layout dramatically reduces design costs, accounting for the successful use of VLSI in memory and PLA chips.

We have taken a somewhat extreme approach to system modularity by proposing a single simple module, called a *cell.* Cells are laid out in a planar array, with connections only to nearest neighbors. The layout of a chip is thus trivial, as is the layout of chips on a board. Our cells combine memory and processing in a finer grain than has been customary. A device built from such cells can perform a substantial computational task, even though it has a topology much more like that of a passive memory than a von Neumann microprocessor.

We restricted our attention to cells that could be implemented with at most a few thousand active elements (gates, transistors). Many modules may thus be laid out on a single VLSI chip, giving structure to the chip design problem. Furthermore, our algorithms can make efficient use of ten to ten thousand or more cells, so that many identical chips can be used in one installation.

Communication and control: For a VLSI design to be truly practical, it must not sidestep any communication or control issues. A good design minimizes both the complexity of each module as well as the number of connecting wires between modules. The latter consideration becomes more important as VLSI technology improves. An increase in the number of active elements on a chip is of little benefit when the chip's I/O bandwidth is limited by its pinout.

We considered designs with eight connections to each cell: power, ground, clock, reset, and four data lines to neighboring cells in the planar array (left, right, up, down). We then attempted to minimize the solution time for a dynamic programming (or transitive closure) problem, assuming a data rate of one bit per clock cycle.

A VLSI chip has enough pins to implement many of our cells, if the cells form a square region of the planar array. For example, 9 cells in a 3×3 array need only 16 external connections: 12 data lines for the cells on the periphery, and 4 common wires for power, ground, clock and reset.

Similarly, a 5×5 matrix will fit on a 24 pin chip, an 8×8 on a 36 pin chip, and a 9×9 on a 40 pin chip. This is the limit of present technology, though 100 pin chips are conceivable in the future. With a few thousand active elements per cell, our designs are of appropriate complexity for VLSI.

Another consideration is the balancing of on-chip processing with I/O. Input and output are fundamentally slower than communication on the chip. This suggests that the construction of custom devices will only be economical for the implementation of "super-linear algorithms", where on-chip processing is of sufficient complexity to balance the time required to read in or write out the data. In other words, our aim is to take an algorithm of more than linear complexity in the classical serial model (say $O(n^3)$) and speed it up by the use of parallelism and pipelining, so that the resulting device can process the data at roughly the same rate that data can be input or output.

Extensibility: This paper deals with the design of special purpose hardware to solve two specific classes of problems. The utility of such designs is limited by their specificity. We seek extensibility in two ways, through size and problem independence.

The hardware should be able to solve arbitrarily sized instances of the problem for which it is intended. Here we suppose that our designs are used to build special purpose devices controlled by a more conventional processor. In this light, it is important that the devices can be used efficiently for the solution of problems that exceed their (one-pass) capacity. This issue will be further explored in Section 3, under the rubric of *decomposability*.

Problem independence is even more important: special purpose hardware should solve as many different problems as possible. Modules could conceivably be microprogrammed as logic density increases. This paper indicates the utility of the mesh-style interconnection pattern for modules, as well as demonstrating two (perhaps generally useful) patterns for data flow in such systems.

Systolic algorithms: There is a newly coined word for our style of algorithm design for VLSI: "systolic", in the sense of [KL2]. The term is meant to denote arrays of identical cells that circulate data in a regular fashion. Kung and Leiserson's cells [KL,KL2] perform but one simple operation; we have relaxed this restriction somewhat so that a small amount of control information circulates with the data.

Data movement is synchronous and bit-serial, to reduce pinout requirements. It is a difficult but hardly insurmountable problem to design chips with ten to one hundred identical modules synchronized with a single clock line.

Time is measured in terms of data transmission. One "word" may be sent along a wire in one unit of time. This convention avoids extraneous detail in the discussion of our algorithms: the choice of word length is left to the implementor. It unfortunately obscures one important issue, namely, that one bit of control information must be sent with each word of data in the transitive closure and dynamic programming algorithms.

Traditional models and goals: Algorithmic design is dominated by the traditional goals that arise naturally from classical machine architectures and technologies. A good serial algorithm minimizes

operation counts, while a good parallel algorithm maximizes concurrent processing. Both viewpoints are somewhat inappropriate to the evaluation of VLSI designs. The theory of cellular automata is more helpful.

In a typical serial model, $o(n^3)$ boolean operations are sufficient to compute a transitive closure [AHU, Chap. 6]. However, the recursive algorithm employed does not seem suitable for direct VLSI implementation, since too much information is passed across recursive calls.

A parallel algorithm has optimal speedup if it cuts computation time by a factor proportional to the number of processors used. But computation time is measured in most parallel models by counting elementary operations, with little consideration of the time necessary to transmit intermediate results.

The theory of cellular automata [vN] does offer some insight into VLSI design. Its generality obscures some important issues, for example, the cost of building the cells (especially if there is more than one type), and the amount of information received by a cell from its neighbors in one unit of time. It lacks the notion of the I/O bottleneck between chips due to pinout restrictions. And it does not address the problem of decomposability.

Other work: Models of computation similar to ours have been previously considered in the literature. This paper was inspired by Kung and Leiserson's [KL] solutions to several matrix problems, and by the vision of VLSI expressed in Mead and Conway's book [MC]. Levitt and Kautz [LK] explored the hardware implementation of Warshall's [Ws] algorithm for transitive closure. However, their designs are not readily decomposable, and they use many more connections per cell.

Organization of the paper: Algorithms for dynamic programming and transitive closure are developed separately in sections 1 and 2. Section 3 concludes the body of the paper with a discussion of decomposablity and further topics.

1. Dynamic Programming

Many problems in computer science can be solved by the use of dynamic programming techniques. These include shortest path, optimal parenthesization, partition, and matching problems, and many others. For a fuller discussion of this spectrum see the review article by K. Brown [Br] and the references mentioned therein. In this paper we will confine our attention to optimal parenthesization problems. This will allow us to explain the ideas without excess generality, while at the same time covering a vast range of significant problems, including the construction of optimal search trees, file merging, context free language recognition, computation of order statistics, and various string matching problems.

The optimal parenthesization problems can all be put in the form:

> Given a string of n items, find an *optimal* (in a certain sense) parenthesization of the string.

As an example, there are five distinct parenthesizations of the string (1 2 3 4):

$$(((1\ 2)\ 3)\ 4) \qquad (1.1)$$

$$(((1\ (2\ 3))\ 4) \tag{1.2}$$

$$((1\ 2)\ (3\ 4)) \tag{1.3}$$

$$(1\ ((2\ 3)\ 4)) \tag{1.4}$$

$$(1\ (2\ (3\ 4))) \tag{1.5}$$

Note that a parenthesization of n items has $n-1$ pairs of parentheses. Also, each parenthesis pair encloses two elements, each of which is either an item or another parenthesis pair. These five parenthesizations correspond to the five ways of performing the addition $1+2+3+4$ without rearranging the terms.

The optimality of a parenthesization is defined with respect to a cost for each parenthesis pair; the total cost is some function (usually the sum) of the individual costs. The optimal parenthesization is the one with minimum total cost.

In the previous example, if the cost of a parenthesis pair is defined as the sum of the enclosed items, then the optimal parenthesization is the first one, with total cost 19.

The obvious way of solving an optimal parenthesization problem involves examining all possible parenthesizations of the string, then picking the one with the smallest cost. This algorithm is clearly exponential in n, as the number of distinct parenthesizations is itself exponential. A dynamic programming strategy for this problem is derived from the following observations. If we have an optimal parenthesization of the whole string, then we also have an optimal parenthesization of each of its substrings. (If we could improve a substring, then we could improve the whole). This suggests that we calculate the optimal parenthesizations of successively larger substrings of the original string, starting from the singleton items. Further, we note that the solution for a given substring will arise as a subproblem for several larger problems, and thus it will pay to remember the optimal solution in order to avoid recomputation.

To simplify matters suppose that we are only interested in the cost of the minimum cost parenthesization, not its structure. Let the original string items be numbered by the integers 1 through n from left to right, and let $c(i,j)$ denote the minimum cost of parenthesizing the substring consisting of items i through $j-1$ (here we assume $1 \leq i < j \leq n+1$). Then, according to the above discussion, the $c(i,j)$ can be computed using a recurrence of the form

$$c(i,j) \quad = \quad \min_{i<k<j} F_{ikj}(c(i,k), c(k,j))\,. \tag{1.6}$$

Here, the $c(i,k)$ and $c(k,j)$ are the optimal costs for parenthesizing two substrings. The range of the minimization variable, k, ensures consideration of all pairs of substrings that can be formed from items i through $j-1$. The function F_{ikj} computes the total cost of parenthesizing the items

i through $j-1$, and it is normally of the form

$$F_{ikj}(c(i,k), c(k,j)) \quad = \quad c(i,k) + c(k,j) + f(i,k,j). \tag{1.7}$$

The total cost of parenthesizing items i through $j-1$ is in this case the sum of the optimal parenthesization of a pair of substrings, plus some additional cost for the outermost pair of parentheses.

In our toy example delineated in the text above,

$$c(1,2) = c(2,3) = c(3,4) = c(4,5) = 0, \tag{1.8}$$

and

$$F_{ikj}(c(i,k), c(k,j)) \quad = \quad c(i,k) + c(k,j) + \sum_{i \le h < j} h. \tag{1.9}$$

The desired value is $c(1,5)$, the cost of parenthesizing items 1 through 4. The dynamic programming approach to evaluating $c(1,5)$ is to solve all "subproblems" first. Thus,

$$c(1,3) = c(1,2) + c(2,3) + (1+2) = 3, \tag{1.10}$$

$$c(2,4) = 5, \tag{1.11}$$

$$c(3,5) = 7. \tag{1.12}$$

With these values known, the following may be computed:

$$c(1,4) = \min(3+6, 5+6) = 9, \tag{1.13}$$

$$c(2,5) = 14. \tag{1.14}$$

Finally,

$$c(1,5) = \min(9+10, 14+10) = 19, \tag{1.15}$$

as asserted previously. The optimal parenthesization is that of (1.1), as identified by the optimal values of k found in applications of (1.6).

In general, one may compute the $c(i,j)$ in order of increasing value of $j-i$, ending with the computation of $c(1, n+1)$.

Note that since there are $\Theta(n^2)$ substrings of the original, and for each substring we are minimizing over $\Theta(n)$ values of k on the average, the computation takes a total of $\Theta(n^3)$ steps. Thus dynamic programming has given us a low order polynomial algorithm for an apparently exponential problem.

We visualize a more general computation by using the triangle depicted below for the case $n = 6$.

[Figure 1.1]

Denote by (ij) the solution to the parenthesization problem for the substring from i to $j-1$. We start out with (12), (23), ..., (67), the singleton substrings, whose solution we assume is given.

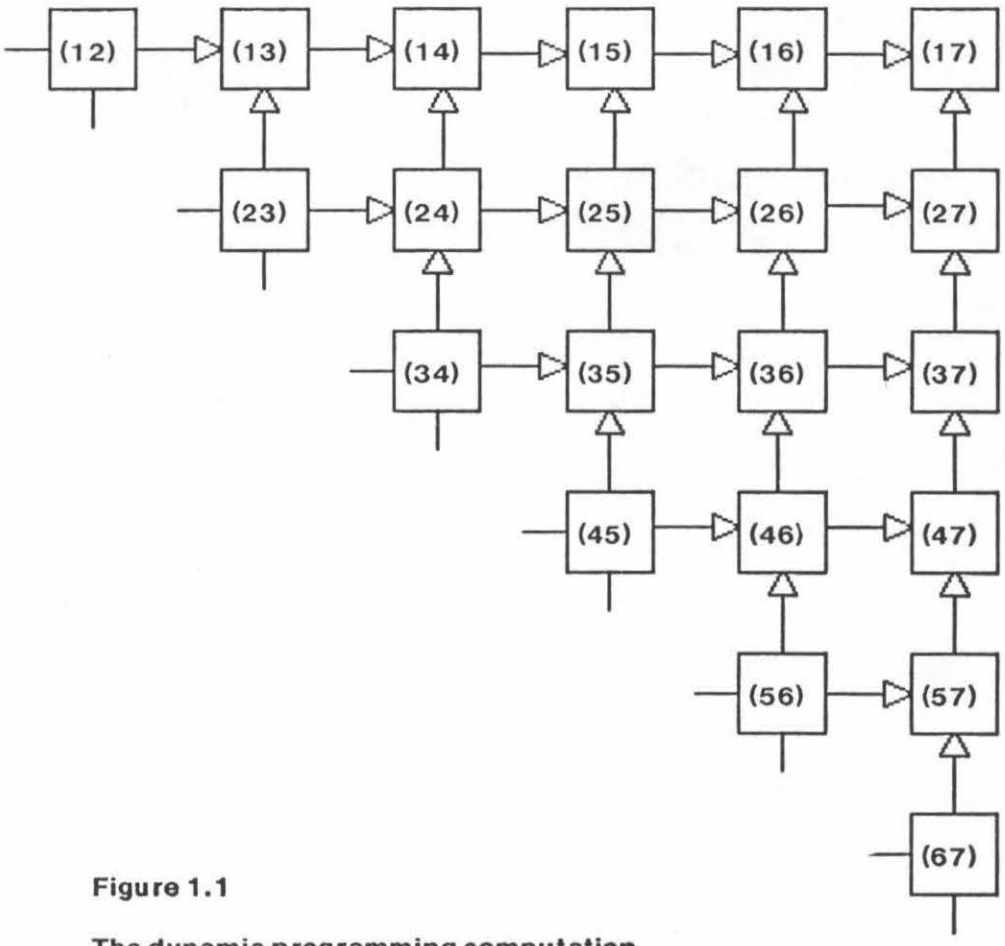


Figure 1.1

The dynamic programming computation.

Using these we can then compute the solutions for substrings of length 2, namely (13), (24), ..., (57). Next we can get (14), (25), ..., (47), then (15), (26), (37), then (16), (27), and finally the desired result (17).

The above serial algorithm is amenable to certain obvious parallelism. If all (ij) with $j - i < s$ are available, then the (ij) with $j - i = s$ can be computed in parallel. Thus if we had $\Theta(n)$ cells, and ignored the cost of data movement, we could finish the computation in $\Theta(n^2)$ steps. This decomposition is, of course, much closer to classical parallel models than to the VLSI model we are advocating. Note that each cell is working in isolation on a complete sub-problem. Previous results must be made available to several cells and thus, unless we design the data movement carefully, contention problems may arise. By contrast, we are seeking an algorithm with simple and regular data flow which offers maximal pipelining. For us cells are inexpensive, as long as they are of a simple and uniform kind. We are happy to provide $\Theta(n^2)$ cells, if we can then solve the problem in $\Theta(n)$ steps.

We now drop the toy example for a more realistic problem, for the remainder of this section. The problem we wish to tackle is that of the construction of an optimum binary search tree [Kn, p.433], for which the above recurrence becomes

$$c(i,j) \quad = \quad w(i,j) + \min_{i<k<j} (c(i,k) + c(k,j)), \tag{1.16}$$

where $w(i,j)$ is the sum of the probabilities that each of the items i through j will be accessed. We will try to solve this problem on a network of cells suggested by Figure 1.1, that is, a triangle of $n(n+1)/2$ cells connected along a rectangular mesh. Cell (ij) will compute one number, the value of $c(i,j)$. Then, it will send its result to the cells that need it to compute their own value.

Note that this structure satisfies many of the *a priori* requirements for efficient VLSI implementation. We have a simple and regular interconnection pattern that corresponds well to the geometrical layout. Furthermore only the cells along the diagonal, and the cell at the upper right hand corner need communicate data to or from the outside world, thus guaranteeing a reasonable pin count. However, much remains to be worked out with respect to data flow. According to our recurrence, for example, cell (17) will need to combine the result of (16) (one of its neighbors) with the result of (67) (a cell far away). The art in the design of this algorithm lies in arranging for the right data to arrive at the right time at each cell, without overloading the communication paths.

As noted in the introduction, time in our systems is defined by data transfers. One unit of time is sufficient for the communication of the value of a $c(i,j)$ between neighboring cells. (Eventually, it will be seen that *two* $c(i,j)$ values and one bit of control information must pass in unit time over the single wire connecting neighboring cells. If additional pinout is available, this bit-serial transmission may be parallelized in the normal fashion.)

We now explain our algorithm. For simplicity of exposition we assume that cell (ij) has been preloaded with $w(i,j)$. (In fact this loading operation can be merged with the computation described

below). Let us say that cell (ij) is at distance $j - i$ *away from the boundary* (e.g., the diagonal (11), ..., (77)). An informal description of the algorithm can then be given as follows:

> If a cell is at distance t away from the boundary, then its result is ready at time $2t$. At that moment the cell starts transmitting its result upwards and to the right. The result travels along both directions by moving by one cell per time unit, for t additional time units. From that moment until eternity the result slows its movement by a factor of 2. (That is, it now moves to the next cell every *two* time units).

Before we descend into the details of how to implement this data flow pattern, let us see that it causes all the right data to arrive at a cell at the right time. A proof of this can be given formally, but is best illustrated by an example: cell (17). The first pair that this cell can hope to combine is (14) and (47) (every other pair has some member that will be generated later than these two). Both (14) and (47) will be ready at time $2*3 = 6$, as they are a distance of 3 away from the boundary. They will travel at full speed for 3 more time units along the paths towards (17), arriving there at time 9. Now we claim the at time 10 cell (17) will be able to combine the two additional pairs (15) with (57), and (13) with (37). We may check just the first one, as the other is clearly symmetric. The result of (15) is available (by our assumption) at time 8, and thus will arrive at (17) at time 10. But (57) is more interesting. It will be ready at time 4, will travel towards (17) at full speed for 2 more time units, arriving at (37) at time 6. But now it will slow down by a factor of 2, and thus it will need 4 more time units to get to (17), arriving there at time 10! Similarly, at time 11, our cell will be able to combine (16) with (67), and (12) with (27). This is all for the good, because thus at time 12 cell (17) is ready to start transmitting its result, exactly as our scheme would require, since it is a cell at distance 6 away from the boundary. For a network of size n, $2n$ time units will be required before the final result is available.

After this overview of the algorithm, we must now examine the implementation at greater depth and check that the available communication paths are adequate to carry the data flow required.

It is simplest to divide the capacity of the wire connecting adjacent cells into three channels. We call these the *fast belt*, the *slow belt*, and the *control line*. The first two channels carry one $c(i, j)$ value per unit time, whereas the *control line* transmits one bit per unit time. (We defer discussion of the control line until the action of the cells has been completely described.)

The cells make use of their communication channels in the following manner. Each cell has five registers: the *accumulator* where the current minimum is maintained, the *horizontal fast* and *horizontal slow* registers, and similarly the *vertical fast* and *vertical slow* registers. On its horizontal *fast belt* connection, a cell normally receives the contents of its left neighbor's horizontal fast register (storing it into its own horizontal fast register), while passing the old contents of that register to its right neighbor. The horizontal slow belts behave in exactly the same way, except that the horizontal slow registers consist of two stages. The data coming in enters the first stage, moves to the next stage at the next time unit, and finally exits the cell at the following time unit. The nomenclature should

now be clear: data in the *fast belt* moves one cell to the right every time unit, while data in the *slow belt* only moves by half to the right every time unit. Completely analogous comments apply to the movement of data upwards in columns of vertical registers.

The operation of a cell is then the following. During every unit of time a cell partakes in the belt motion and also updates its accumulator. The new value is the minimum of the old accumulator contents, the sum of the new contents of its horizontal fast and vertical slow registers, and the sum of its horizontal slow and vertical fast registers. In addition, if this cell is at distance t away from the boundary, then at time $2t$ it will copy the contents of its accumulator into its fast horizontal and vertical registers. And finally, if it is at an even distance $t = 2s$ from the boundary, then at time $3t/2$ it will load the first stage of its horizontal (and vertical) slow register from the horizontal (resp. vertical) fast belt, ignoring its slow belts entirely.

We see that our algorithm uses only a bounded number of registers per cell, thus meeting another of the desired attributes of a solution. In order to prove that this works we must show that for every belt no data gets overwritten which still needs to be used. Figure 1.2 below illustrates the contents of the fast and slow horizontal belts for the first row of cells in the example discussed earlier.

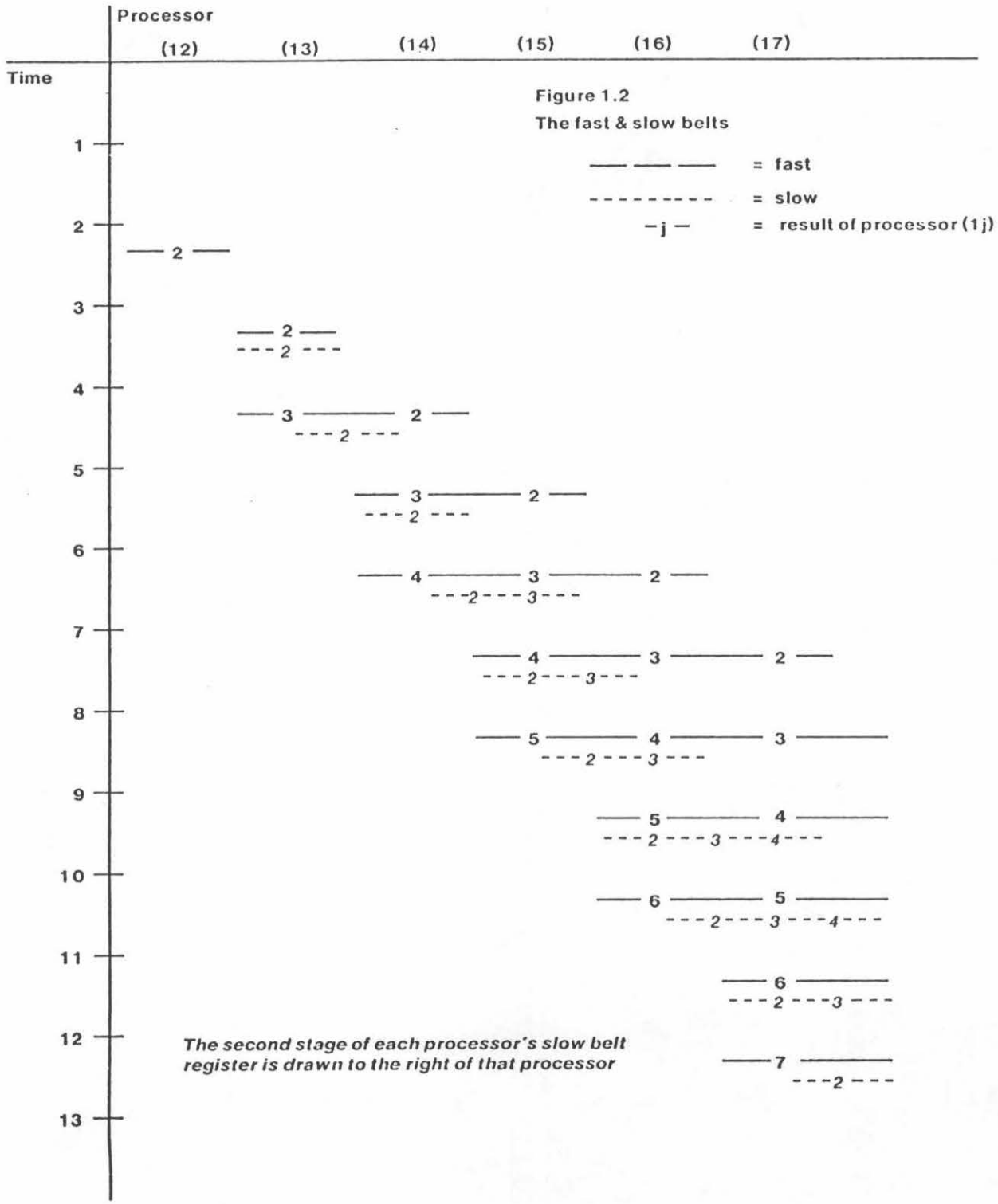
[Figure 1.2]

Notice that when a new value is stored on a belt, it is never on top of a previously written value. In addition the fast belt, in a sense, "reverses" in space the results the cells write on it. In other words, the results occur on this belt in the opposite direction from the direction that the corresponding cells are laid out in space. In contrast, the slow belt maintains the ordering of the results the same as that of the cells. This sheds some light as to why the two speed scheme succeeds in allowing a cell to combine results generated "close by" with results generated "far away".

At last we need to address the timing issue: does a cell need to know its distance from the boundary and count accordingly? The answer is no, the signals on the *control lines* are sufficient to determine the action of each cell in a uniform fashion. These lines have a capacity of one bit per time unit. During each time unit, a cell receives one control bit from the left and one from below, and transmits one bit to the right and one bit to its upward-adjacent neighbor.

The control signals "flow" through the system, much as the data does, although at a different rate. The accumulator to fast belt transfer that occurs in cell (ij) at time $j - i$ is controlled by a rightward moving signal that moves at a rate of one cell every two time units. The fast to slow belt transfer that occurs at time $3(j - i)/2$ is controlled by an upward moving signal that moves at a rate of two cells every three time units.

We end this section by summarizing again the important attributes of our cell. First of all, there is only one kind. It has a small number of registers and small number of data and control paths connecting it to its immediate geometric neighbors. And finally, both the architecture of the cell and the algorithm it executes are independent of the network size.


Processor
(12)
(13)
(14)
(15)
(16)
(17)
Time
1
2
3
4
5
6
7
8
9
10
11
12
13
Figure 1.2
The fast & slow belts
= fast
= slow
= result of processor (1j)
The second stage of each processor's slow belt register is drawn to the right of that processor

2. Transitive Closure

The transitive closure problem is the following. Consider an $n \times n$ matrix A of 0's and 1's. We interpret A as defining a directed graph on the n vertices 1, 2, ..., n. The (ij) entry of A is 1, if and only if there is a directed arc in the graph from vertex i to vertex j. The transitive closure of A, to be denoted by A^*, is also an $n \times n$ 0-1 matrix, whose (ij) entry is a 1 iff there is a directed *path* from vertex i to vertex j in the graph. Formally speaking, there is a directed path from i to j iff 1) there is a directed arc from i to j, or 2) there is a vertex k for which there are directed paths from i to k and from k to j, or 3) if $i = j$.

The transitive closure problem arises in many contexts in computer science. In implementing process synchronization, when a resource becomes available, we must trace down chains of processes each suspended on a resource held by another in order to discover which may run next. In updating a computer display containing objects partially obscuring other objects, we again must compute the closure of the "in front of" relation. In the data-flow analysis of computer programs we often need the closure of the "call" relation. Tree or graph traversal (such as garbage collection) can also be viewed as transitive closure problems. Dijkstra [D] has argued that transitive closure should be one of the fundamental building blocks in any programming system. He pointed out several other problems in the solution of which the computation of a transitive closure naturally arises. Furthermore, as became clear in the last section, what really defines our algorithms is data movement and not data operations. This implies that any network we construct for transitive closure is likely to be also applicable to any other problem with the same data flow. A large class of such problems, called shortest path problems, is discussed in [AHU, Sect. 5.6-5.9].

There is a well-known serial algorithm for the transitive closure problem due to Warshall [Ws]. Subsequently Warren [Wn] published an interesting "row-oriented" algorithm. Both of these algorithms compute the transitive closure of an $n \times n$ matrix in time $\Theta(n^3)$. It is also well known that the serial complexity of transitive closure and matrix multiplication are the same. Thus, at the expense of much more complicated code, the asymptotic complexity of the problem can be further reduced, using the techniques of Strassen or Pan. In this section we will seek a simple $O(n)$ algorithm implementable on a square mesh of n^2 cells. The transitive closure problem has in fact been previously considered by cellular automata theorists and two solutions are known to the authors, one by Christopher [Ch], and one by van Scoy [vS]. Both of these operate in $O(n)$ time. However, they lack certain essential ingredients of an algorithm appropriate for VLSI implementation. First, the complexity of the program executed by a cell is high. In both papers the code expressed in pseudo-ALGOL is over four pages long. Second, and more important, these algorithms have bi-directional data flow along both the horizontal and vertical connections. As we will see in the next section, this makes it quite difficult to *decompose* the algorithm, that is to implement it when only a $k \times k$ array of cells is available, with k less than n.

A useful device for the correctness proof for many of the above algorithms is the notion of "versions". This is especially appropriate when we imagine that we are updating each entry a_{ij} of A in place.

We introduce versions 1, 2, ..., n for each element a_{ij}, where version 1 is the original a_{ij}, and version $n+1$ the (ij) entry of the transitive closure. In terms of the graph model, we interpret the t-th version of a_{ij}, written as $a_{ij}^{(t)}$, to denote the existence of a path from vertex i to vertex j which, aside from its starting and ending vertices, only visits vertices in the set $\{1, 2, \ldots, t-1\}$. This interpretation makes clear that the $a_{ij}^{(t)}$ can be computed from the recurrence

$$a_{ij}^{(t+1)} = a_{ij}^{(t)} + a_{it}^{(t)} a_{tj}^{(t)}, \tag{2.1}$$

where we use "+" to denote logical *or* and product to denote logical *and*. The above recurrence indicates a partial order according to which versions of different elements must be computed. Both Warshall's and Warren's algorithms can be viewed as certain natural "topological sorts" of this partial order. The same machinery will be useful for justifying the algorithm suggested below. As a final point, note that the values of $a_{ij}^{(t)}$ are monotonic increasing in t, and thus wherever version t is required in the right-hand side of the above recurrence, it is always safe to use version s, if $s \geq t$.

We now describe our solution. We use an $n \times n$ array of cells with the rectangular mesh connections. End-around (toroidal) connections are useful but not essential. Each cell has an accumulator initialized to 0 (false). We also use some external memory that can hold a copy of A. We visualize two copies of the array A flowing past this processor array, one copy horizontally and the other copy vertically, as suggested in Figure 2.1.

[Figure 2.1]

Note that the horizontal copy is a vertical mirror image of A and that it is tilted backwards by 45 degrees. Analogous comments apply to the vertical copy. The tilting of the copies is used so that element a_{i1} from the horizontal copy and element a_{1j} from the vertical copy arrive at the cell in location (ij) at the same time. The algorithm now proceeds as follows (for simplicity we assume here the existence of the end-around connections):

> During each time unit, the horizontal and vertical copies advance by one to the right and down respectively. Each cell *and*s the data coming to it from the left and above and *or*s it into its accumulator. Normally a cell passes the data coming from the left to its right, and that from above, downwards. However, when an element of the horizontal copy passes its *home location* in the cell array, it updates itself to the value of the accumulator at that location. Thus when element (32) of the horizontal copy enters cell (32) on the left, the contents of that cell's accumulator exit on the right. As the horizontal copy starts coming out at the right end of the cell array, it is immediately fed back in at the left using the end-around connections. Entirely analogous comments apply to the vertical copy. After the two copies have cycled thus *three* times through, the accumulators of the cell array contain the transitive closure A^* of A (stored in the standard row-major order). The result can now be unloaded by a fourth pass like the above, or by a separate mechanism.

The correctness of the algorithm can be proven by using the idea of versions discussed earlier. Observe that over each cell, during every time unit, the column index of the element currently there

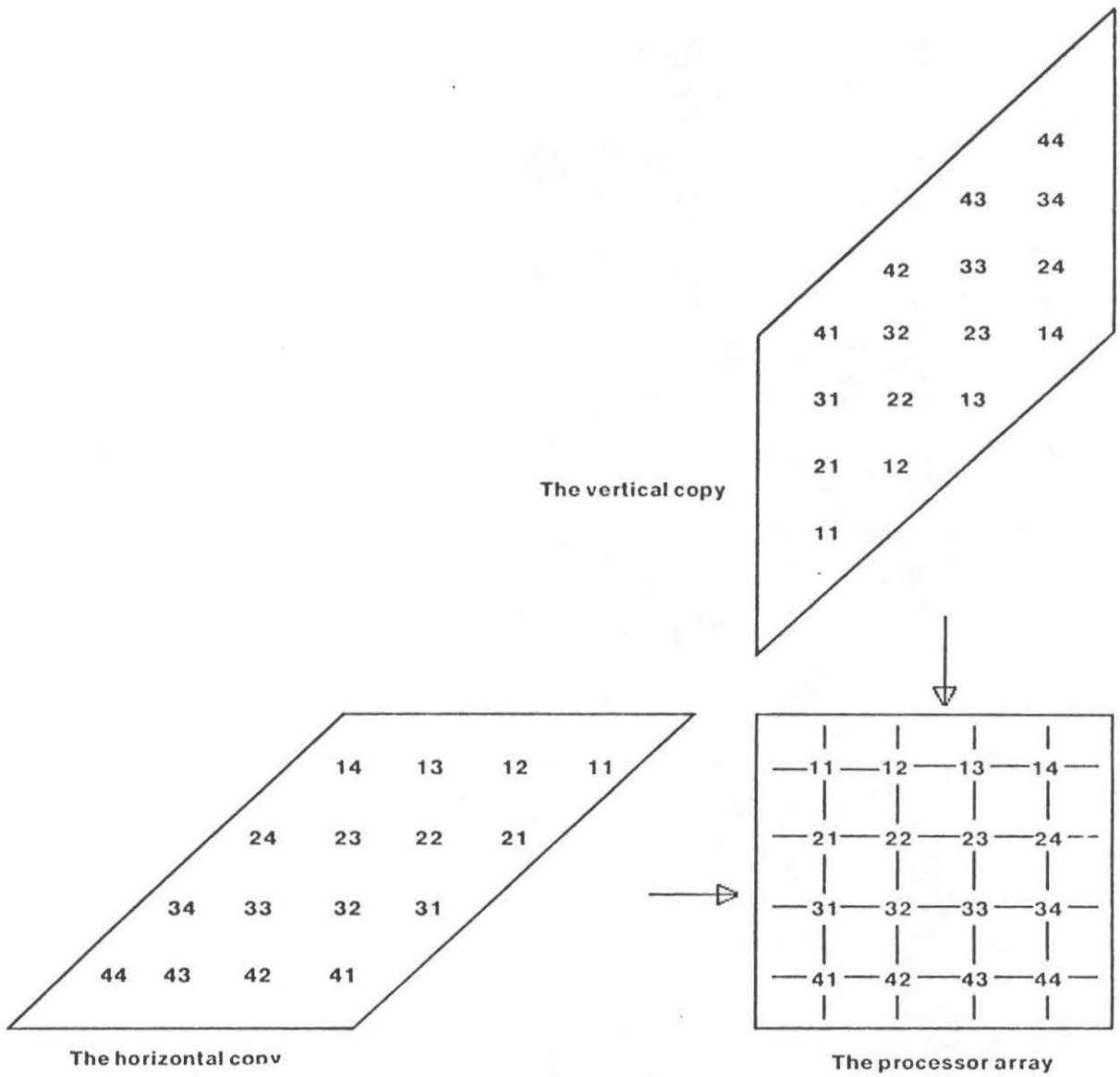


Figure 2.1
Transitive closure

from the horizontal copy equals the row index of the element from the vertical copy. Inductively this implies that the contents of accumulator (ij) are always majorized by transitive closure entry a^*_{ij} (they cannot ever become 1 if the corresponding transitive closure entry is 0). So it is necessary to show only that the accumulator of cell (ij) is brought up to version $n+1$ of element a_{ij}.

We claim that, after the first pass of the two copies over the cell array, element a_{ij} is brought up to version $p = \min(i, j)$ in both copies. To see this note that equation (2.1) implies that

$$a^{(p)}_{ij} = a^{(1)}_{i1}a^{(1)}_{1j} + a^{(2)}_{i2}a^{(2)}_{2j} + \ldots + a^{(p-1)}_{i(p-1)}a^{(p-1)}_{(p-1)j}. \tag{2.2}$$

If we inductively assume that our claim is true for elements in either copy of the form $a_{ij'}$ with $j' < j$, or elements of the form $a_{i'j}$ with $i' < i$, then we can conclude that cell (ij) will perform the inner product in (2.2) above, *before* the (ij) entry of either copy has passed over it. Thus by induction and the monotonicity of the *or* operation our claim is proved. A slight modification of the same argument proves that, after the second pass, element a_{ij} is brought up to version j in the horizontal copy and version i in the vertical one. Finally, if we use (2.2) with $p = n+1$ then we can conclude that, after the third pass, the accumulator of cell (ij) will contain the ij-th entry of the transitive closure.

As in the case of dynamic programming, it remains to discuss the implementation of timing. How does a cell know when one of its "mates" is passing over it? Once again, this problem can be solved by including one bit of control information with each datum transmitted in the array of cells.

Let's confine our attention to the horizontal mates, as the situation for vertical mates is entirely dual. Note that the horizontal mate arrives at cell (ij) exactly when the diagonal element (jj) of that column in the vertical copy arrives there also. This suggests an extremely simple timing implementation. We start enabling signals at the top edge of the cell array which propagate downwards. The signals move by one cell during each unit of time. Furthermore, we start the signal at cell $(1j)$ at time $2*(j-1)$. It is easy to check that these signals coincide with the diagonal elements of the vertical copy. Thus here also our cells execute code independent of the network size.

The overall time required to complete the computation (including unloading of the cell array) is easily seen to be $5*(n-1)$, the same as the best previously known solution for cellular automata [Ch]. However, a direct implementation of that solution in VLSI would be inferior to our algorithm, as the units of time are different: more control information flows between cells during each time unit in Christopher's solution.

3. Algorithm Decomposition and Further Topics

We now take up some additional issues. First is the problem of algorithm decomposition. We explore here this issue in the context of the transitive closure problem. Similar comments apply to dynamic programming. If we are given a $k \times k$ array of cells and want to compute the transitive closure of an $n \times n$ matrix, with $k < n$, how do we do it? For simplicity we suppose that k divides

n. Thus we can evenly divide our $n \times n$ matrix into $k \times k$ blocks. To process a block we will cycle through it a horizontal and a vertical section of the array, using the algorithm of the previous section. From the horizontal copy of the full array we extract the $k \times n$ slice corresponding to the block rows and feed that into the device horizontally. Similarly, from the vertical copy we extract the $n \times k$ vertical slice correspoding to the columns of the block. As the slices flow out of the device, they update the memory in which the corresponding array copies are stored. The $k \times k$ blocks can then be processed in this fashion in any order consistent with both the left-right and the top-down ordering of the blocks (the Young tableau order). Many variations on this basic idea are possible, including interleaving the processing of the blocks with the three passes, regenerating one of the slices on the fly so it need not be stored, and others. Recent work of Mead and Rem [MR] on LSI implementations of arrays so they can be accessed either by row or by column has applications here.

The correctness of the decomposition can be proved by using the "monotonicity of versions" remark in the previous section. The case $k = 1$ gives an interesting serial algorithm, which can be viewed as the next logical step in the sequence whose first two terms are Warshall's and Warren's algorithms (in this order). Note that the computation time is now $O(n^3/k^2)$, and thus we still have optimal speed-up to within a constant factor. Finally we remark that a decomposition such as the above is possible precisely because we have signals flowing only downwards and to the right. This leads to an acyclic dependency graph, since there is an order in which to process the blocks such that each computation depends only on previous computations. If we had bi-directional signals along some dimension, so that there exist two blocks along that dimension each depending on signals from the other, then we could not complete the processing of either block without starting the other. Although it is still possible to run the two blocks as coroutines, the complications of saving state and loading and unloading the device would make such a solution prohibitively expensive.

We now conclude with some more general remarks. There is substantial similarity between the dynamic programming and transitive closure cell. Even stronger is the similarity between the transitive closure cell and that used in Kung and Leiserson's work on matrix algorithms. Both are "inner product step" cells. The possibility of mapping all these algorithms onto one type of module needs further exploration.

From a mathematical point of view, perhaps the most interesting question is to ask for a characterization of the computations which can be carried out in this style within certain performance bounds. If we start from a recurrence describing a serial algorithm for the solution to a problem, is there a theory to help us in designing a network like those described here, which would execute exactly the same computation steps, only in a highly parallel and pipelined fashion? Can we describe what processor topologies can be used for what kind of recurrences? The number of open questions is vast.

4. References

[AHU] Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974

[Br] Brown, K.Q., *Dynamic programming in computer science*, CMU tech. rep., June 1978

[D] Dijkstra, E.W., *Determinism and recursion versus non-determinism and the transitive closure*, EWD456, October 1974, Burroughs Corp.

[Ch] Christopher, T.W., *An implementation of Warshall's algorithm for transitive closure on a cellular computer*, rep. no. 36, Inst. for Comp. Res., Univ. of Chicago, February 1973

[KL] Kung, H.T., and Leiserson, C.E., *Algorithms for VLSI processor arrays*, Symp. on Sparse Matrix Computations, Knoxville, Tn., November 1978

[KL2] Kung, H.T., and Leiserson, C.E., *Systolic Arrays for VLSI*, Cal Tech Conference on VLSI, Pasadena, Ca., January 1979

[Kn] Kunth, D.E., *The Art of Computer Programming*, vol. 3, Sorting and Searching, Addison-Wesley, 1973

[LK] Levitt, K.N., and Kautz, W.H., *Cellular arrays for the solution of graph problems*, Comm. ACM, Vol. 15, No. 9, (1972), pp. 789-801

[MC] Mead, C.A., and Conway, L.A., *Introduction to VLSI Systems*, textbook in preparation

[MR] Mead, C.A., and Rem, M., *Cost and performance of VLSI computing structures*, Calif. Inst. of Tech. tech. rep., June 1978

[vS] van Scoy, F.L., *A parallel transitive closure algorithm requiring linear time*, unpublished manuscript, May 1977

[vN] von Neumann, J., *The theory of self-reproducing automata*, (Burks, A.W., editor), Univ. of Illinois, 1966

[Wn] Warren, S.W., Jr., *A modification of Warhsall's algorithm for the transitive closure of binary relations*, Comm. ACM, vol. 18, no. 4, April 1975, pp.218-220

[Ws] Warshall, S., *A theorem on boolean matrices*, Journ. ACM, vol. 9, no. 1, Jan. 1972, pp.11-12