SIAM J. SCI. STAT. COMPUT.
Vol. 5, No. 1, March, 1984

© 1984 Society for Industrial and Applied Mathematics
0196-5204/84/0501-0007 $01.25/0

NUMERICALLY STABLE SOLUTION OF DENSE SYSTEMS OF LINEAR EQUATIONS USING MESH-CONNECTED PROCESSORS*

A. BOJANCZYK†, R. P. BRENT‡ AND H. T. KUNG§

Abstract. We propose a multiprocessor structure for solving a dense system of n linear equations. The solution is obtained in two stages. First, the matrix of coefficients is reduced to upper triangular form via Givens rotations. Second, a back substitution process is applied to the triangular system. A two-dimensional array of $\theta(n^2)$ processors is employed to implement the first step, and (using a previously known scheme) a one-dimensional array of $\theta(n)$ processors is employed to implement the second step. These processor arrays allow both stages to be carried out in time $O(n)$, and they are well suited for VLSI implementation as identical processors with a simple and regular interconnection pattern are required.

Key words. Givens method, least squares, linear systems, numerical stability, orthogonal factorization, parallel algorithms, *QR* method, special-purpose hardware, systolic arrays, VLSI

1. Introduction. Recently, several algorithms have been proposed for solving a system of linear equations on a parallel computer. The algorithm of Csanky [1] solves a dense system of size n in $\theta(\log^2 n)$ time steps with $\theta(n^4)$ processors. This is the best known upper bound on the time complexity of the problem. Since $\Omega(\log n)$ is a lower bound we have a gap of order $\log n$. Unfortunately, Csanky's algorithm is numerically unstable [14] and uses too many processors to be useful in practice. Gaussian elimination without pivoting can trivially be carried out in parallel in $\theta(n)$ steps using n^2 processors [5]. If the matrix of the system is not special (e.g., diagonally dominant or symmetric positive definite) then pivoting is generally necessary to guarantee numerical stability. With pivoting we need $\theta(n \log n)$ steps and n^2 processors. To avoid the pivoting problem, Sameh and Kuck [12] (and also Kowalik et al. [7], [8], [11]) proposed the use of Givens transformations to triangularize the matrix of coefficients. The orthogonal factorization requires $\theta(n)$ steps with $\theta(n^2)$ processors. The factorized linear system can then be solved in $\theta(n)$ steps using $\theta(n)$ processors. Hence, the algorithm for solving dense system of linear equations requires $\theta(n)$ time steps and $\theta(n^2)$ processors, yielding a speed-up of order n^2 over the usual sequential algorithms, which require $\theta(n^2)$ time steps.

However, traditional operation counts do not adequately measure the cost of a parallel computation. There are many other factors which must be considered when evaluating the performance of parallel algorithms. One of the most important is the cost of data transmission. In many papers dealing with parallel algorithms, there is an explicit or implicit assumption that the time required to obtain a single datum is negligible. This is not true in practice as every data transfer between processors takes time. Interprocessor communication must be realized by a network that interconnects the processors. Any algorithm can be supported by different networks but, in general, the number of data transfers depends on the topology of the network. With different networks one can have different execution times for the same algorithm. Thus, one should decide what kind of network is to be employed and only then proceed to evaluate the performance of the algorithm. Bearing this in mind, Kant and Kimura

* Received by the editors June 10, 1981, and in revised form July 15, 1982.
† Institute for Informatics, University of Warsaw, PKiN, p. 850, 00-901 Warsaw, Poland.
‡ Department of Computer Science, Australian National University, Canberra, ACT 2600, Australia.
§ Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213. The research of this author was supported in part by the National Science Foundation under grant MCS78-236-76.

[6] showed that the solution of a dense system of linear equations can be obtained in $\theta(n)$ steps using n^2 mesh connected processors, but their algorithm requires that the matrix of the system be "strongly nonsingular". The assumption of strong nonsingularity is a severe one as it excludes many interesting nonsingular matrices (e.g. the identity matrix) and it appears to be no easier to verify the assumption than to solve the corresponding linear system. Thus, the result of Kant and Kimura [6] is mainly of theoretical interest.

Kung and Leiserson [10] introduced a new model of parallel computation. The model takes into account such issues as cost of I/O, control and data transfers. A point of their work is that one should fit a network to an algorithm in order to obtain good overall performance. Using a simple and regular network, called a "systolic array", of $\theta(n^2)$ hexagonally connected processors, Kung and Leiserson [10] improved the result of Kant and Kimura [6] by requiring only that the linear system be solvable by Gaussian elimination without pivoting. For example, the matrix of the linear system could be symmetric positive definite or irreducible and diagonally dominant. See Kung [9] for a general discussion of systolic architectures for various special-purpose computational devices.

Combining the ideas of Sameh and Kuck [12] and Kung and Leiserson [10], we introduce a systolic array of $\theta(n^2)$ processors which is capable of transforming any nonsingular matrix to triangular form in $\theta(n)$ units of time in a numerically stable manner. The resulting triangular system can be solved in $\theta(n)$ steps on an array of n linearly connected processors. Both processor arrays enjoy regular geometries, and all processors are similar. As a consequence, cost-effective special purpose hardware devices based on our scheme could conceivably be built using VLSI technology. For many applications each processor needs to perform floating-point computations on words of at least 32 bits. To achieve a throughput of one floating-point operation every microsecond, present technology would allow only one (or a small number) of processors per chip, but advances in technology should soon make it possible to put many processors on a chip.

2. Givens rotations. Our algorithm is based on the orthogonal factorization of a real nonsingular n by n matrix $A = (a_{ij})$,

$$QA = R,$$

where Q is an orthogonal matrix formed as the product of plane rotations, and R is upper triangular.

A plane rotation is defined by a matrix

$$P_{i+1,j} = \begin{array}{c} \text{col. } i \\ \downarrow \end{array} \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & c_i & s_i & \cdots & \\ & & -s_i & c_i & & \\ & & & & \ddots & \\ & & & & & 1 \end{bmatrix} \leftarrow \text{row } i.$$

The matrix $P_{i+1,j}$ applied on the left rotates rows i and $(i+1)$ of A so as to annihilate the off-diagonal element $a_{i+1,j}$. The parameters of $P_{i+j,i}$ are defined (except in degener-

ate cases, which are dealt with in § 4) by

$$\bar{a}_{i,j} = (a_{i,j}^2 + a_{i+1,j}^2)^{1/2},$$

$$c_i = a_{i,j}/\bar{a}_{i,j},$$

$$s_i = a_{i+1,j}/\bar{a}_{i,j}.$$

Rows i and $i+1$ of the product $\bar{A} = P_{i+1,j}A$ are given by

$$\bar{a}_{i,p} = c_i a_{i,p} + s_i a_{i+1,p},$$

$$\bar{a}_{i+1,j} = 0,$$

$$\bar{a}_{i+1,p} = -s_i a_{i,p} + c_i a_{i+1,p} \quad \text{for } p \neq j.$$

The orthogonal matrix Q is formed as the product of plane rotations $P_{i,j}$ such that the elements of A below the main diagonal are annihilated.

3. Parallel Givens rotations. As some of the rotations $P_{i,j}$ are independent it is possible to apply more than one at a time. There are many possibilities. We propose a scheme that requires $N = 3n - 5$ sweeps. Each sweep Q_k, $k = 1, 2, \cdots, N$, is a direct sum of plane rotations $P_{i,j}$, where $(i, j) \in L_k$ and sets of indices L_k, $k = 1, 2, \cdots, N$, are defined by

$$(3.1) \qquad (i, j) \in L_k \quad \text{iff} \quad 3(j-1) + n - (i-1) = k, \qquad 1 \leqq j < i \leqq n.$$

Note that Q_k is a product of commuting orthogonal matrices $P_{i,j}$. The orthogonal matrix Q is the product of sweeps Q_k,

$$Q = Q_N Q_{N-1} \cdots Q_1.$$

From (3.1) it follows that if $(i, j) \in L_k$ then $(i+3, j+1)$ and $(i-3, j-1)$ also belong to L_k provided they are in $\{(i, j) | 1 \leqq j < i \leqq n\}$. The rule of thumb is as follows. Starting from any element $a_{i,j}$, $i > j$, and moving like a "long" chess knight on the chessboard, one square left and three squares up or one square right and three squares down within the lower triangular part of the matrix A, we reach all elements which are annihilated at the same time as the element $a_{i,j}$. This is illustrated for $n = 8$ in Fig. 1, where all elements annihilated in the kth sweep are denoted by $\boxed{k}$.

x							
7	x						
6	9	x					
5	8	11	x				
4	7	10	13	x			
3	6	9	12	15	x		
2	5	8	11	14	17	x	
1	4	7	10	13	16	19	x

FIG 1. *Ordering of rotations* ($n = 8$).

4. The basic processing element. Two kinds of operations are required for the transformations $P_{i,j}$: determination of the rotation parameters c and s, and application of the rotation, which is equivalent to

$$\begin{bmatrix} x \\ y \end{bmatrix} := \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}. \tag{4.1}$$

The operation (4.1) will be referred to as a rotation step.

Thus, we need a processor which is able to determine rotation parameters and execute rotation steps. In addition to its computing capabilities the processor should have four connections (two inputs and two outputs) and four internal registers (R_x, R_y, R_c and R_s). To perform the first operation, the processor shifts data x and y on its input lines (denoted by X and Y) into registers R_x and R_y (see Fig. 2). Then it

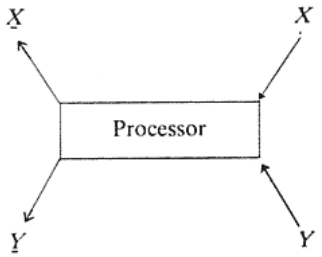


FIG. 2. *The processing element.*

computes parameters c and s by the following algorithm:

```
if R_x = 0 then
    begin
        c := 0;
        s := 1
    end
else if abs (R_x) > abs (R_y) then
    begin
        x := abs (R_x) × sqrt (1 + (R_y/R_x)^2);
        c := R_x/x;
        s := R_y/x
    end
else
    begin
        x := abs (R_y) × sqrt (1 + (R_x/R_y)^2);
        c := R_x/x;
        s := R_y/x
    end;
```

The computed values c and s are stored in registers R_c and R_s, and the new value x is made available as output on the output line X. (The new value y on the output line Y is not calculated since c and s are chosen in such a way that y is known to be zero.) The processor determines the parameters c and s only once, so the contents of the R_c and R_s registers are not subsequently changed. Every subsequent operation performed by the processor is a rotation step. More precisely, the processor shifts data on its input lines X and Y into registers R_x and R_y, then executes the rotation

step, i.e.,

$$\underline{x} := R_x R_c + R_y R_s, \qquad \underline{y} := -R_x R_s + R_y R_c$$

and makes new values $\underline{x}$ and $\underline{y}$ available as outputs on the output lines $\underline{X}$, $\underline{Y}$. There is a simple finite-state machine which controls switching the processor from one kind of operation to the other.

Knowing what operations the processor must perform, we define a *time unit* to be the maximal time that is necessary for a processor to determine parameters c and s *or* to perform a rotation step together with loading and unloading its registers.

It is possible that a rotation will take more or less time than determination of the rotation parameters. The rotation parameters are determined only once, so the processors may occasionally be idle. This is a price we pay to guarantee that the whole system works correctly while keeping the system control relatively simple.

We assume that there is a synchronization mechanism which latches input and output lines. When processors are connected together, the changing output of one processor during a time unit should not interfere with the input to another processor. Sometimes we shall refer to the operations executed by a processor within one time unit as a pulsation (see Kung and Leiserson [10]).

5. Network organization. The systolic array proposed here is made up of a network of $n(n-1)/2$ processors, where n is the problem dimension. The position of a processor in the network is fully determined by integers i and k, $1 \leqq k < i \leqq n$, so every processor will be specified by a pair (i, k). The processor (i, k) is assigned to perform the transformation $P_{i,k}$.

The network organization has the property that all connections from a processor are to at most four neighboring processors. More precisely, output line $\underline{X}$ of processor (i, k) coincides with input line Y of processor $(i-1, k)$, and output line $\underline{Y}$ of processor (i, k) coincides with input line X of processor $(i+1, k+1)$ (see Fig. 3). All connections form a rectangular grid on a triangle, as illustrated for $n = 6$ in Fig. 4.

There are special "gray" processors or shift registers along the bottom of the network. A gray processor does not perform any arithmetic. It simply delays data

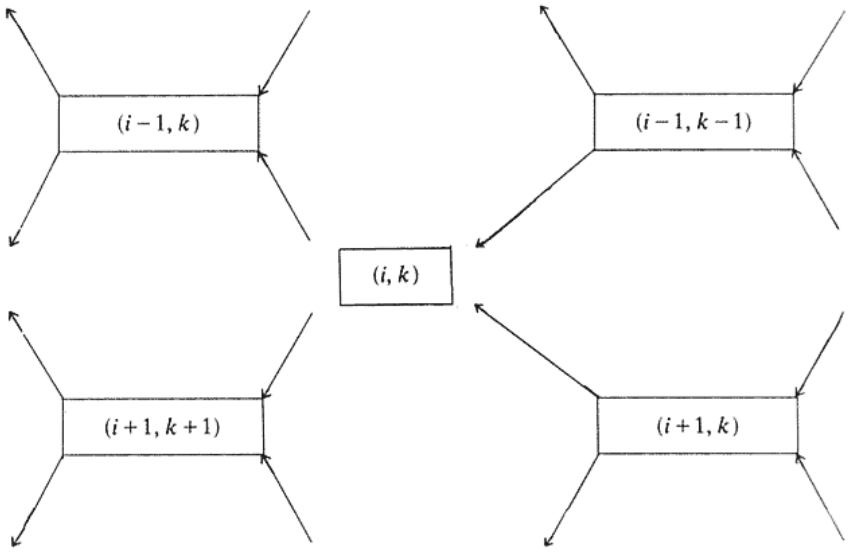


FIG 3. *Inter-processor communication.*

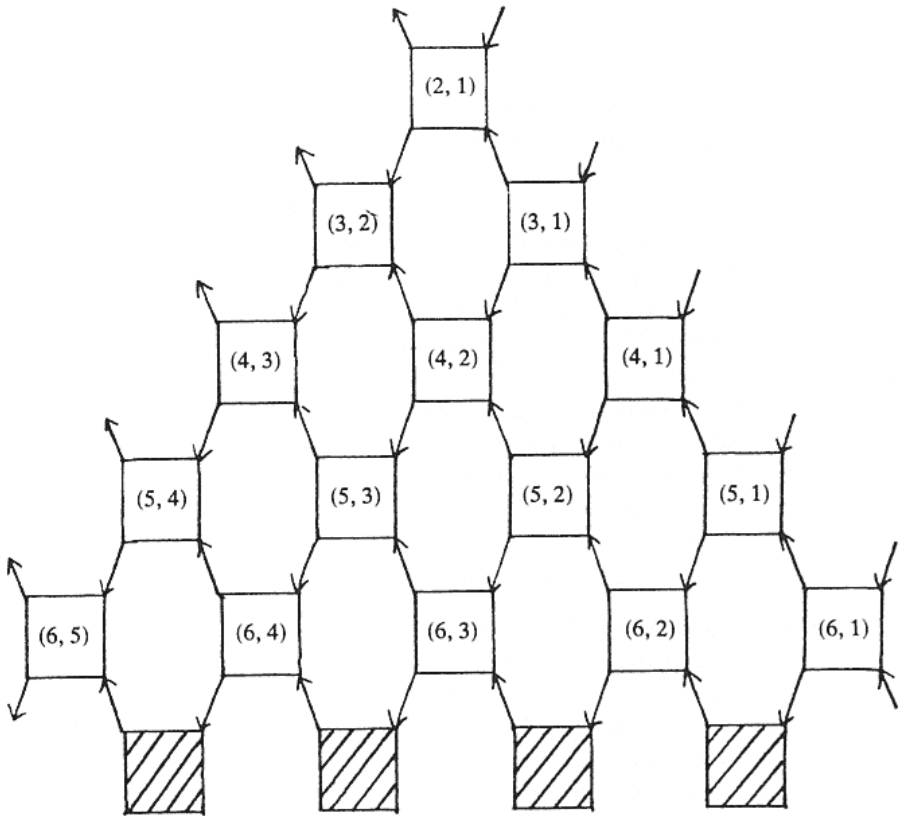


FIG. 4. *Layout of the processors* ($n = 6$).

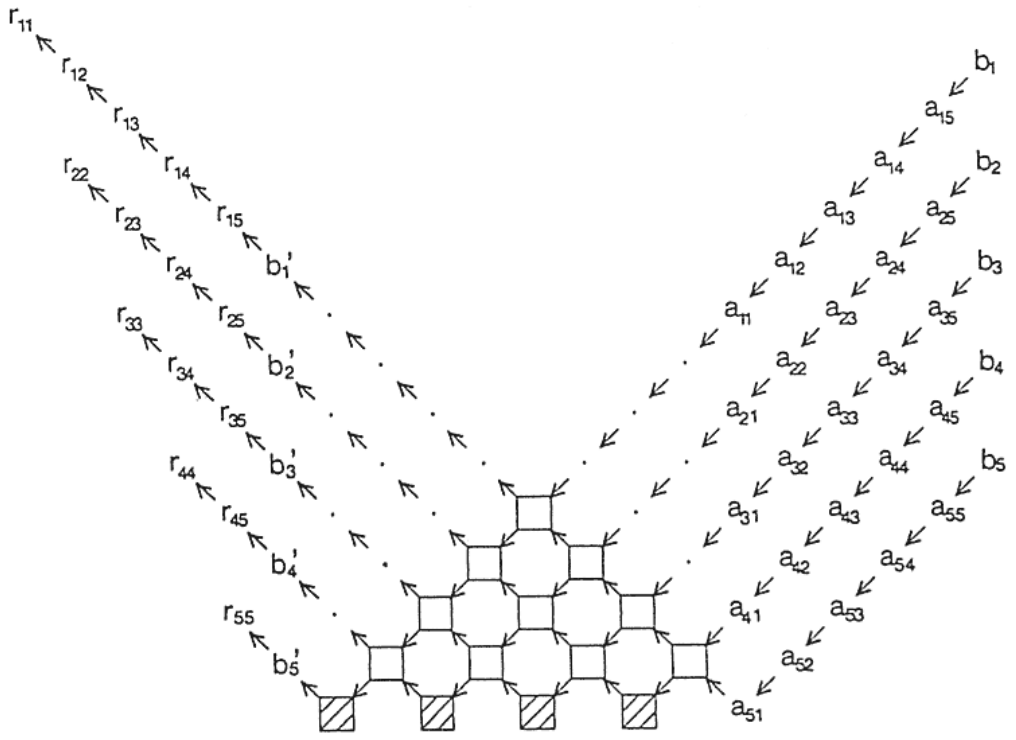


FIG. 5. *Data flow into the systolic array* ($n = 5$).

transmission by one time unit. This is necessary because only every third sweep introduces a zero in the last row.

Data enter the network through the right boundary, i.e. through the processors $(i, 1)$, $i = n, n-1, \cdots, 2$, and leave the network through the left boundary, i.e. through the processors $(i, i-1)$, $i = 2, 3, \cdots, n$ (see Figs. 4 and 5).

6. Operation of the systolic array. Our computational scheme is applied to the $n \times (n+1)$ matrix $\underset{\sim}{A} = [A, b]$, i.e., to the matrix A augmented by the vector b.

In the following description the superscript of a matrix coefficient will usually indicate the number of plane rotations $P_{i,k}$ in which this coefficient was involved. By convention, the coefficients of the original augmented matrix have superscript 0, with the exception of the coefficients of the last row, which for notational convenience have superscript 1.

The computation is initiated at time $\tau = 0$, when $a_{n-1,1}^{(0)}$ and $a_{n,1}^{(1)}$ enter the systolic array through the right bottom processor, i.e., processor $(n, 1)$. As its first operation, this processor determines rotation parameters c and s corresponding to transformation $P_{n,1}$ as well as computing $a_{n-1,1}^{(1)}$. At subsequent time steps processor $(n, 1)$ performs rotation steps. At time $\tau = 1$, processor $(n-1, 1)$ starts to work. Subsequently processors $(n-2, 1), (n-3, 1), \cdots, (2, 1)$ are activated at times $\tau = 2, 3, \cdots, n-2$. Figure 5 depicts how elements of the matrix are fed into the systolic array. In Fig. 6 we show four consecutive pulsations of the network.

We now specify the operation of the network precisely by giving the schedule of processor (i, k), $1 \leqq k < i \leqq n$. The processor (i, k) (assigned to perform plane rotation $P_{i,k}$) begins its activity at time $\tau = 3(k-1)+(n-i)$. Its first task is to annihilate the k-th element of row i. (The first $k-1$ elements of rows $i-1$ and i will already have been annihilated.) At time τ the processor determines rotation parameters c and s based on data $a_{i-1,k}^{(2k-2)}$ and $a_{i,k}^{(2k-1)}$, and computes the new value of the kth element of row $i-1$, i.e. element $a_{i-1,k}^{(2k-1)}$. Then $a_{i-1,k}^{(2k-1)}$ is made available as an output on the output line $\underset{\sim}{X}$. Every subsequent operation by the processor is a rotation step. More precisely, for $k < j \leqq n$, at time $3(k-1)+(n-i)+(j-k)$ processor (i, k) executes operations

$$a_{i-1,j}^{(2k-1)} := c \times a_{i-1,j}^{(2k-2)} + s \times a_{i,j}^{(2k-1)},$$
$$a_{i,j}^{(2k)} := -s \times a_{i-1,j}^{(2k-2)} + c \times a_{i,j}^{(2k-1)}$$

and makes $a_{i-1,j}^{(2k-1)}$ and $a_{i,j}^{(2k)}$ available as output on its output lines $\underset{\sim}{X}$ and $\underset{\sim}{Y}$ respectively. It is easy to check by induction that every processor gets its data at the right time.

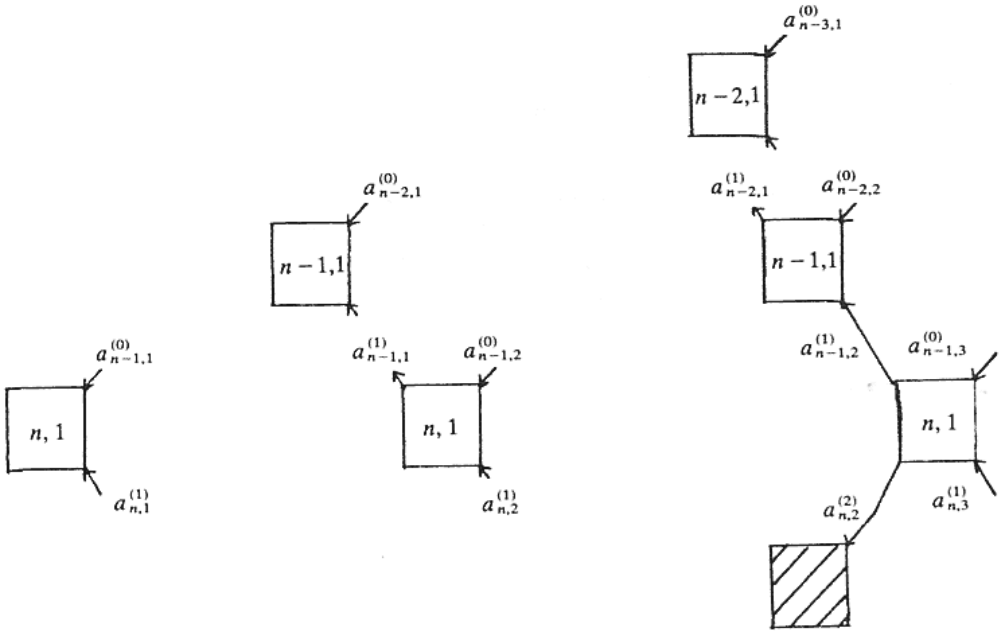
It follows from the schedule of the output processors, i.e. processors $(i, i-1)$, $i = 2, 3, \cdots, n$, that at time $n-1$ the coefficient $a_{1,1}^{(1)}$ leaves the network. The whole upper triangular matrix $R = QA$ and transformed right-hand side vector Qb are known at time $3n-3$. Thus we have:

THEOREM 6.1. *A dense nonsingular system of n linear equations can be orthogonally transformed to a triangular system in $3n-3$ time units using a systolic array consisting of $n(n-1)/2$ mesh-connected processors.*

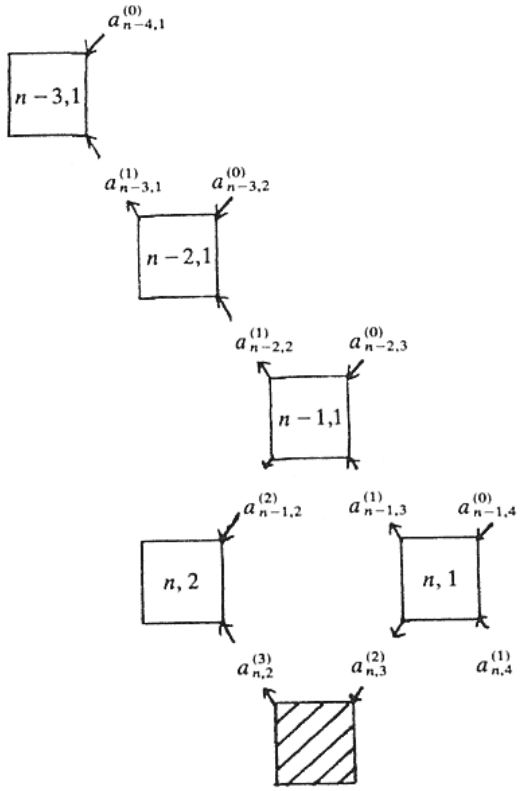
We still have to solve a triangular linear system. This can be done in $3n$ time units, using a systolic array first introduced by Kung and Leiserson [10]. Thus we have:

THEOREM 6.2. *If A is an $n \times n$ nonsingular matrix, then a linear system of equations $Ax = b$ can be solved in $6n + O(1)$ time units using systolic arrays of $n(n-1)/2$ and n processors.*

Remarks. Note that several systems with the same matrix A and different right-hand side vectors $b_1, b_2, \cdots, b_m$ can be processed almost as easily as one. The



(a) time $\tau = 0$ (b) time $\tau = 1$ (c) time $\tau = 2$



(d) time $\tau = 3$

FIG. 6. *The first four steps.*

computational scheme is applied to the matrix $\underline{A} = [A, b_1, \cdots, b_m]$ rather than to $[A, b]$, and $Q\underline{A}$ is obtained in the time $3n-4+m$. Similarly, an obvious extension of our scheme may be used to solve linear least squares problems. Recently Gentleman and Kung [4] have proposed a different systolic scheme which has advantages over ours for solving linear least squares problems. In addition, for handling banded matrices they have proposed a scheme in which the number of processors needed in the systolic array depends on the band width of the matrix rather than its order.

The error analyses of our Givens process and back substitution process are as described in Gentleman [3] and Wilkinson [14] for the classical sequential processes. Thus, we have speeded up the process of solving systems of linear equations and maintained the numerical quality of the well-behaved sequential algorithm at the same time. (Singular or nearly singular A can be detected once the Givens triangularization of A has been computed, as in the sequential case.) A multiprocessor array structure equivalent to ours was independently proposed by Gannon [2].

It is worth noting that matrices which are too large for a given systolic array can be triangularized by first splitting them into blocks. The triangularization time is $\theta(n^3/p^2)$ where n is the matrix dimension and p^2 is the number of processor used, $p \leqq n$.

Sameh and Kuck [12] and Kowalik and Kamgnia [7] present schemes that require only $N = 2n-3$ "sweeps" using the "short" chess knight move elimination order. However, the time taken by each of these sweeps is that required to generate the parameters in a rotation matrix *and* to perform a rotation. In addition, they do not consider the cost of data transfers. Suppose that generating a rotation matrix and performing a rotation each take a unit time, as assumed by the timing analysis of this paper. Then one can easily see from data dependency relations that our $N = 3n-5$ sweeps with the "long" chess knight move elimination is the best one can do.

7. Application to the *QR* algorithm. One iteration of the QR algorithm can be expressed in the form

$$\text{factorization phase: } QA = R,$$

$$\text{multiplication phase: } \tilde{A} = RQ^T,$$

where Q is orthogonal and R is upper triangular. See, for example, Stewart [13].

Our systolic array is capable of performing the factorization phase. While the matrix A passes through the network, the orthogonal matrix Q is formed as a product of plane rotations $P_{i,j}$. Parameters defining the transformations $P_{i,j}$ are stored among the processors of the network. If we do not switch our network to process another factorization phase, the previously computed orthogonal matrix Q (in multiplicative form) is not destroyed and remains intact in the network. Now, by passing any other matrix B through the network, we obtain the product QB.

In the multiplication phase we have to know how to form a product $\tilde{A} = RQ^T$. By applying our systolic device to the matrix R^T we can easily get $\tilde{A}^T = QR^T$ instead. Now, to complete the multiplication phase it is enough to transpose the matrix $\tilde{A}^T$. Thus we need a fast method for matrix transposition. One way to achieve this is to use a buffer that supports fast two-dimensional addressing.

When we have a systolic array for matrix triangularization and a buffer to support matrix transposition, one iteration of the QR algorithm is easy to execute. First we produce the matrix R, then transpose it, form $\tilde{A}^T = QR^T$, and transpose the matrix $\tilde{A}^T$ to obtain $\tilde{A}$. The cost of one iteration of the QR algorithm performed in this way is Kn time units. We shall not specify the constant K as it depends on how fast we

compute matrix transposition. Our treatment of the *QR* algorithm here is preliminary; future research is needed to study issues such as shift selection, convergence testing, etc.

REFERENCES

[1] L. CSANKY, *Fast parallel matrix inversion algorithms.* SIAM J. Comput, 5 (1976), pp. 618–623.
[2] D. GANNON, *A note on pipelining a mesh connected multiprocessor for finite element problems by nested dissection,* in *Proc. 1980 International Conference on Parallel Processing,* IEEE Computer Society, August, 1980, pp. 197–216.
[3] W. M. GENTLEMAN, *Error analysis of QR decompositions by Givens transformations,* Linear Algebra and Appl, 10 (1975), pp. 189–197.
[4] W. M. GENTLEMAN AND H. T. KUNG, *Matrix triangularization by systolic arrays,* in Proc. of SPIE Symposium, Vol. 298, Real-Time Signal Processing IV, The Society of Photo-optical Instrumentation Engineers, August, 1981.
[5] D. HELLER, *A survey of parallel algorithms in numerical linear algebra,* SIAM Rev., 20 (1978), pp. 740–777.
[6] R. M. KANT AND T. KIMURA, *Decentralized parallel algorithms for matrix computation,* in Proc.Fifth Annual ACM Symposium of Computer Architecture, Palo Alto, CA, 1978, pp. 96–100.
[7] J. S. KOWALIK AND E. R. KAMGNIA, *Parallel Givens transformations applied to matrix factorization and systems of linear equations,* Techn Rep. CS-79-050, Washington State University, Pullman, WA, 1979.
[8] J. S. KOWALIK, S. P. KUMAR AND E. R. KAMGNIA, *An implementation of fast Givens transformations on a* MIMD *computer,* Techn. Rep. Washington State University, Pullman, WA, November, 1980.
[9] H. T. KUNG, *Why systolic architectures?,* IEEE Computer Magazine, 15(1): (Jan 1982), pp. 37–46.
[10] H. T. KUNG AND C. E. LEISERSON, *Systolic arrays (for* VLSI), in Sparse Matrix Proceedings 1978, I. S. Duff and G. W. Stewart, eds., Society for Industrial and Applied Mathematics, Philadelphia, 1979, pp. 256–282. A slightly different version appears in Introduction to VLSI Systems by C. A. Mead and L. A. Conway, Addison-Wesley, Reading, MA, 1980, 8.3.
[11] R. E. LORD, J. S. KOWALIK AND S. P. KUMAR, *Solving linear algebraic equations on a* MIMD *computer,* Techn. Rep., Washington State University, Pullman, WA, August, 1980.
[12] A. H. SAMEH AND D. J. KUCK, *On stable parallel linear system solvers,* J. Assoc. Comput. Mach., 25 (1978), pp. 81–91.
[13] G. W. STEWART. *Introduction to Matrix Computations,* Academic Press, New York, 1973.
[14] J. H. WILKINSON, *The Algebraic Eigenvalue Problem.*, Oxford Univ. Press, 1965.