An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations

HAROLD S. STONE

Stanford University, Stanford, California

ABSTRACT. Tridiagonal linear systems of equations can be solved on conventional serial machines in a time proportional to N, where N is the number of equations. The conventional algorithms do not lend themselves directly to parallel computation on computers of the ILLIAC IV class, in the sense that they appear to be inherently serial. An efficient parallel algorithm is presented in which computation time grows as $\log_2 N$. The algorithm is based on recursive doubling solutions of linear recurrence relations, and can be used to solve recurrence relations of all orders.

KEY WORDS AND PHRASES: parallel computation, ILLIAC IV, linear equations, computer architecture, numerical analysis, LU decomposition, tridiagonal equations

CR CATEGORIES: 5.14, 6.22

1. *Introduction*

The trend in large scale high speed computers today clearly points to the use of internal parallelism to obtain significant increases in speed. For example, the ILLIAC IV computer can perform N simultaneous computations where $N = 64$, 128, 256, or 512. We expect that highly efficient computations performed on a computer of the ILLIAC IV class will be executed N times faster than on a serial computer of the same inherent speed. Actually, inefficiencies due to overhead and constraints on data communication among processors will reduce the speed increase to kN where k lies in the interval $0 \leq k \leq 1$. Efficient algorithms have k near unity.

Unfortunately, many parallel algorithms do not lend themselves to efficient parallel computation. We can exhibit examples of algorithms for which computation time decreases rather slowly as we increase the number of processors, and for some pathological examples the computation time is independent of the number of processors. An efficient parallel algorithm has the property that computation speed on a processor with N-fold parallelism is N times faster than computation on a serial processor.

In this paper we examine the solution of tridiagonal systems of linear equations. It is well known that such systems can be solved using a conventional serial com-

This work was supported by the NASA Ames Research Center, by the Joint Services Electronics Program under contract N-00014-67-A-0112-0044, and by the National Science Foundation under grant GJ-1180.

Author's address: Stanford University, Departments of Electrical Engineering and Computer Science, ERL 416, Stanford, CA 94305.

puter in a time proportional to N where N is the number of equations. We present an algorithm for solving the equations in a time proportional to $\log_2 N$ by using a computer with N-fold parallelism. Hence, for this problem the ratio of the computation speed of a parallel processor to that of a serial processor is proportional to $N/\log_2 N$, since this is the inverse ratio of the computation times. As N grows large, $N/\log_2 N$ grows as $N^{1-\epsilon}$ for any $\epsilon > 0$, and therefore this algorithm asymptotically attains the N-fold speed increase that we require of efficient parallel algorithms. A different parallel algorithm for this problem that exhibits a similar time behavior has been developed by Buneman [1] and Buzbee et al. [2].

In Section 2, we state the problem and indicate conventional serial methods for solution. These methods are inherently serial in that each computation depends on the result of the immediately preceding computation. In Section 3 we show how to perform a forward and backward sweep in $\log_2 N$ steps when given the LU decomposition of the original matrix. In Section IV we show how to obtain the LU decomposition in $\log_2 N$ steps. This particular computation is of general interest because it is an efficient method for evaluating partial fraction expansions and linear difference equations in parallel.

2. *Statement of the Problem*

We wish to solve the tridiagonal system of equations

$$\mathbf{Ax} = \mathbf{b}$$

where

$$\mathbf{A} = \begin{bmatrix} d_1 & f_1 & & & & \\ e_2 & d_2 & f_2 & & & \\ & e_3 & d_3 & f_3 & & \\ & & \cdots & & & \\ & & e_{N-1} & d_{N-1} & f_{N-1} & \\ & & & e_N & d_N & \end{bmatrix}.$$

In the remainder of this paper we assume that N is a power of 2, but this is not an essential assumption.

There are a number of related methods for solving this system serially in a time proportional to N. The parallel algorithm presented here is based upon one such algorithm, the LU decomposition (cf. Forsythe and Moler, [4]). In this algorithm we find two matrices, $\mathbf{L}$ and $\mathbf{U}$, such that (i) $\mathbf{LU} = \mathbf{A}$; (ii) $\mathbf{L}$ is a lower bidiagonal matrix with 1's on its principal diagonal; (iii) $\mathbf{U}$ is an upper bidiagonal matrix.

When $\mathbf{A}$ is nonsingular, its LU decomposition is unique provided that $\mathbf{L}$ and $\mathbf{U}$ satisfy the conditions given. In fact, it is easily shown that

$$\mathbf{U} = \begin{bmatrix} u_1 & f_1 & & & & \\ & u_2 & f_2 & & & \\ & & u_3 & f_3 & & \\ & & & \cdots & & \\ & & & u_{N-1} & f_{N-1} & \\ & & & & u_N & \end{bmatrix}$$

where f_i, $1 \leq i \leq N-1$, is the upper diagonal of A, and

$$u_1 = d_1, \qquad u_i = d_i - (e_i f_{i-1}/u_{i-1}) \quad \text{for} \quad i > 1. \tag{1}$$

The lower bidiagonal matrix, $\mathbf{L}$, is then given by

$$\mathbf{L} = \begin{bmatrix} 1 & & & & \\ m_2 & 1 & & & \\ & m_3 & 1 & & \\ & & \cdots & & \\ & & m_{N-1} & 1 & \\ & & & m_N & 1 \end{bmatrix}$$

where

$$\begin{aligned} m_2 &= e_2/d_1, \\ m_i &= e_i/(d_{i-1} - f_{i-2}m_{i-1}) \qquad \text{for } i > 2, \\ &= e_i/u_{i-1} \qquad\qquad\qquad\quad \text{for } i \geq 2. \end{aligned} \tag{2}$$

After computing L and U, it is relatively straightforward to solve the system of equations. The solution is a two-step process.

Letting $\mathbf{y} = \mathbf{Ux}$, we have

$$\mathbf{Ax} = \mathbf{LUx} = \mathbf{Ly} = \mathbf{b}.$$

The equation $\mathbf{Ly} = \mathbf{b}$ is easily solved for $\mathbf{y}$ since

$$y_1 = b_1, \qquad y_i = b_i - m_i y_{i-1} \quad \text{for } 2 \leq i \leq N. \tag{3}$$

Then we solve $\mathbf{Ux} = \mathbf{y}$ for $\mathbf{x}$. This equation is solved by a backward sweep since

$$x_N = y_N/u_N, \qquad x_i = (y_i - x_{i+1}f_i)/u_i. \tag{4}$$

Note that the recurrence formulas (1), (2), (3), and (4) constitute a complete algorithm for the solution of $\mathbf{Ax} = \mathbf{b}$. Since each computation in this algorithm depends on the results of the previous computation, the algorithm is satisfactory for serial computation but quite unsatisfactory for parallel computation. In the following sections we derive equivalent formulas that are well-suited for parallel computation.

3. *Parallel Evaluation of the Forward and Backward Sweeps*

The model of a parallel processor that lies behind the development of these parallel algorithms is based upon the ILLIAC IV computer. In this computer there are N processors with independent memories, but only one instruction stream. All of the processors operate synchronously, executing the same instruction on N different operand pairs, where N can be 64, 128, 256, or 512. For added flexibility, there is a mask associated with each processor that enables or disables the processor. Hence if a processor's mask is on, the processor executes the current instruction; otherwise the processor remains idle.

Data can be communicated among the processors in one of two ways. One datum can be broadcast to all processors simultaneously, or a vector of N items can be shifted cyclically among the processors. As an example of the latter case, suppose

that the vector $\mathbf{b} = (b_1, b_2, b_3, \cdots, b_N)$ is stored with b_i in the ith processor. Then the vector can be shifted j places cyclically so that b_i is routed to processor $(i + j)$ mod N for all i.

In this section we shall show how to solve (3) by a technique called *recursive doubling*. The idea is to rewrite (3) so that y_{2i} is a function of y_i. Thus in successive iterations we can compute y_1, y_2, y_4, y_8, etc., and y_N can be computed in $\log_2 N$ iterations. Since (4) is of the same form as (3), the backward sweep can be done using the same algorithm, and it also requires $\log_2 N$ iterations.

To begin the derivation, we rewrite (3) in the form

$$y_1 = b_1, \qquad y_i = b_i + (-m_i)y_{i-1}. \tag{3'}$$

This change is necessary because we shall make use of the associativity of addition.

Substituting for y_{i-1} in (3') we find

$$\begin{aligned} y_2 &= b_2 + (-m_2)\cdot b_1, \\ y_3 &= b_3 + (-m_3)\cdot b_2 + (-m_3)\cdot(-m_2)\cdot b_1, \\ y_i &= \sum_{j=1}^{i} b_j \prod_{k=j+1}^{i} (-m_k) \end{aligned} \tag{5}$$

where a vacuous product of m_k's is interpreted as the constant 1.

The last formula in (5) shows the explicit dependence of y_i on each of the coefficients of $\mathbf{m}$ and $\mathbf{b}$. Our goal is to derive a recurrence in which y_{2i} is a function of y_i. To anticipate the answer, momentarily consider what happens when all of the components of $\mathbf{m}$ are equal to -1. In this case y_i is the sum of the first i components of $\mathbf{b}$. Then if $y_i(b_j, b_{j-1}, \cdots, b_{j-i+1})$ is defined to be the sum of b_j through b_{j-i+1}, we have

$$y_{2i}(b_{2i}, b_{2i-1}, \cdots, b_1) = y_i(b_{2i}, b_{2i-1}, \cdots, b_{i+1}) + y_i(b_i, b_{i-1}, \cdots, b_1). \tag{6}$$

Equation (6) holds for all $i \geq 1$. This recurrence has the recursive doubling form that we seek, because it expresses y_{2i} in terms of two functions that are each half as complex as y_{2i}. Moreover, we can evaluate the terms in (6) in parallel because they are computationally identical and differ only in the values of the arguments. For our general solution we compute $Y_1, Y_2, \cdots, Y_N$ where each Y_i is a function of i components of $\mathbf{b}$ and $\mathbf{m}$. We use $Y_i(j)$ as an abbreviation of the more cumbersome notation $Y_i(b_j, b_{j-1}, \cdots, b_{j-i+1}, m_j, m_{j-1}, \cdots, m_{j-i+1})$. That is, $Y_i(j)$ is a function of i consecutive components of $\mathbf{b}$ and $\mathbf{m}$, with the jth component being the highest component.

The following theorem establishes the relation we desire.

THEOREM 1. *Let $Y_i(j)$ satisfy the recurrence relation*

$$Y_{i+1}(j) = Y_i(j) + Y_i(j-1)\cdot(-m_j) \qquad \text{for } i, j \geq 1 \tag{7}$$

with the boundary conditions

$$\begin{aligned} Y_1(j) &= b_j, && \text{for } j \geq 1, \\ Y_i(j) &= 0 && \text{for } j \leq 0, \\ Y_i(j) &= 0 && \text{for } i \leq 0. \end{aligned}$$

Then

(a) *for* $s \geq 1$, $Y_i(j)$ *satisfies the recurrence relation*

$$Y_{i+s}(j) = Y_s(j) + Y_i(j-s) \prod_{k=j-s+1}^{i} (-m_k) \qquad \text{for } i \geq 1,\ j \geq s; \tag{8}$$

(b) $$Y_i(j) = \sum_{k=1}^{j} Y_1(k) \prod_{s=k+1}^{j} (-m_s) \qquad \text{for } i \geq j \geq 1; \tag{9}$$

(c) *for* $i \geq j \geq 1$, $Y_i(j) = y_j$, *where* y_j *is the* j*th component of the unique solution of* (3).

PROOF. To prove part (a), we use induction on s.

Basis step. By hypothesis, (8) holds for $s = 1$.

Induction step. We assume that (8) holds for all s in the interval $1 \leq s \leq n-1$, and we show it holds for $s = n$.

From the induction hypothesis we have

$$\begin{aligned} Y_{i+n}(j) &= Y_{n-1}(j) + Y_{i+1}(j-n+1) \cdot \prod_{k=j-n+2}^{j} (-m_k) \\ &= Y_{n-1}(j) + Y_1(j-n+1) \cdot \prod_{k=j-n+2}^{j} (-m_k) \\ &\quad + Y_i(j-n) \cdot \prod_{k=j-n+1}^{j} (-m_k). \end{aligned}$$

But from the induction hypothesis it follows that

$$Y_n(j) = Y_{n-1}(j) + Y_1(j-n+1) \cdot \prod_{k=j-n+2}^{j} (-m_k).$$

Hence

$$Y_{i+n}(j) = Y_n(j) + Y_i(j-n) \cdot \prod_{k=j-n+1}^{j} (-m_k)$$

which is the same recurrence as (8) with s replaced by n. This proves part (a).

To prove part (b), we use induction on i.

Basis step. By hypothesis, (9) holds for $i = 1$.

Induction step. We assume that (9) holds for all i in the interval $1 \leq i \leq n-1$, and we prove that it holds for $i = n$. Using (8) we have

$$Y_n(j) = Y_1(j) + Y_{n-1}(j-1) \cdot (-m_j).$$

Using the induction hypothesis to substitute for $Y_{n-1}(j-1)$ yields

$$\begin{aligned} Y_n(j) &= Y_1(j) + \left[\sum_{k=1}^{j-1} Y_1(k) \prod_{s=k+1}^{j-1} (-m_s) \right] \cdot (-m_j) \qquad \text{for } 2 \leq j \leq n, \\ &= \sum_{k=1}^{j} Y_1(k) \prod_{s=k+1}^{j} (-m_s) \qquad \text{for } 2 \leq j \leq n. \end{aligned} \tag{10}$$

The interval $2 \leq j \leq n$ for which the equations above are valid arises from the application of the induction hypothesis to $Y_{n-1}(j-1)$ for $1 \leq j-1 \leq n-1$. Since (10) has the same form as (9), it is only necessary to show the validity of (10) for

$j = 1$ to complete the proof. From the theorem hypothesis,

$$Y_n(1) = Y_1(1) + Y_{n-1}(0) = Y_1(1).$$

Since the same result is obtained by setting $j = 1$ in (10), the interval in (10) may be changed to $1 \leq j \leq n$. This proves part (b) of the theorem.

Part (c) is a direct consequence of the fact that with the boundary condition $Y_1(j) = b_j$, (10) is identical to the solution to (3). This completes the proof of the theorem.

COROLLARY.

$$Y_{2i}(j) = Y_i(j) + Y_i(j - i) \cdot \prod_{k=j-i+1}^{j} (-m_k) \qquad \text{for } i, j \geq 1. \tag{11}$$

PROOF. The proof follows directly from part (a) of Theorem 1 by replacing s by i.

The corollary of Theorem 1 provides the recursive doubling algorithm for the solution of (3). The product term in (11) appears to be difficult to evaluate because the number of factors in the product doubles with each iteration. Fortunately, we can also use recursive doubling to compute the product term.

Let $M_i(j)$ be defined to be

$$\begin{aligned} M_i(j) &= \prod_{k=j-i+1}^{j} (-m_k) \qquad \text{for } j \geq i, \\ &= \prod_{k=1}^{j} (-m_k) \qquad \text{for } j < i. \end{aligned} \tag{12}$$

Then (11) can be rewritten as

$$Y_{2i}(j) = Y_i(j) + Y_i(j - i) \cdot M_i(j) \quad \text{for } i, j \geq 1. \tag{13}$$

The recursive doubling computation of $M_i(j)$ is provided by the formula

$$M_{2i}(j) = M_i(j) \cdot M_i(j - i) \qquad \text{for } i, j \geq 1 \tag{14}$$

with the boundary conditions

$$\begin{aligned} M_1(j) &= -m_j \qquad \text{for } j \geq 1, \\ M_i(j) &= 1 \qquad \text{for } j \leq 0, \\ M_i(j) &= 1 \qquad \text{for } i \leq 0. \end{aligned}$$

The parallel algorithm for the solution of (3) is simply the iterative application of (13) and (14). It is given below in an ALGOL-like language. In the program, when an interval of the form $(1 \leq j \leq N)$ appears after a statement, that statement is assumed to be executed simultaneously for all indices in the interval.

```
begin
real array Y[1:N], M[2:N];
real array b[1:N], m[2:N];
comment Y and M are the arrays in which equations (13) and (14) are evaluated Arrays
  b and m are the arrays that give the coefficients of (3) These arrays may utilize the same
  storage space as the arrays Y and M, respectively,
initialize:
  Y[j] := b[j], (1 ≤ j ≤ N);
  M[j] := -m[j], (1 ≤ j ≤ N);
for i := 1 step i until N/2 do
  begin
  Y[j] := Y[j] + Y[j-i] × M[j], (i+1 ≤ j ≤ N);
  M[j] := M[j] × M[j-i], (i+1 ≤ j ≤ N);
  end;
```

At the completion of each iteration, the array Y contains $Y_i(j)$ and M contains $M_i(j)$, $1 \leq j \leq N$. Thus the vectors computed are $Y_2(j)$, $Y_4(j)$, $Y_8(j)$, etc. From Theorem 1, $Y_N(j) = y_j$ for $1 \leq j \leq N$, so that Y_N is the solution to (3). The algorithm exhibits the central property of recursive doubling because the computation of Y_{2i} depends only on Y_i and the complexity of Y_{2i} is approximately double that of Y_i. Since i doubles during each iteration, $\log_2 N$ iterations are required for the computation.

The vector operations indicated in the program are easily carried out in an ILLIAC IV type of computer since masking operations can be used to establish the interval for the index j and cyclic shifting of components of a vector can be used to align $Y[j]$ with $Y[j-i]$. The parallel algorithm is also suitable for efficient operation in vector processors of the pipeline class such as the CDC STAR computer.

For the solution of the backward sweep, eq. (4), the body of the iteration should be modified as indicated below:

begin
$Y[j] := Y[j] + Y[j+i] \times M[j], \quad (1 \leq j \leq N - i);$
$M[j] := M[j] \times M[j+i], \quad (1 \leq j \leq N - i);$
end;

4. *Calculation of the LU Decomposition by Recursive Doubling*

We now focus attention on the efficient calculation of (1) and (2). Again we use recursive doubling to compute the coefficients $u = (u_1, u_2, \cdots, u_N)$ and $m = (m_2, m_3, \cdots, m_N)$. The approach we use is to solve (1) by recursive doubling, then compute $m_i = e_i/u_{i-1}$ simultaneously for $2 \leq i \leq N$ to solve (2).

Since (1) is a continued fraction expansion, it is convenient to cast it into a linear form which is suitable for a recursive doubling algorithm. It is well known (cf. Wall, [10]) that every continued fraction expansion is associated with a linear second-order recurrence relation. In particular, if we define the quantities q_i, $0 \leq i \leq N$, by the recurrence relation

$$q_i = d_i q_{i-1} - e_i f_{i-1} q_{i-2}, \qquad i \geq 2 \tag{15}$$

with the boundary conditions

$$q_0 = 1, \qquad q_1 = d_1$$

then it is easily shown that

$$u_i = q_i/q_{i-1} \qquad \text{for } i \geq 1 \tag{16}$$

or equivalently,

$$q_i = \prod_{j=1}^{i} u_j .$$

To solve (1) efficiently, we have only to solve (15) efficiently, because after calculating $q_i, 0 \leq i \leq N$, we can evaluate (16) in a single operation carried out simultaneously on N processors. Equation (15) is somewhat more difficult to solve than (3) because it is of second order, whereas (3) is of first order. However, we can make use of an artifice to reformulate (15) as a matrix recurrence relation of first order. In particular, it follows from (15) that

$$\begin{bmatrix} q_i \\ q_{i-1} \end{bmatrix} = \begin{bmatrix} d_i & -e_i f_{i-1} \\ 1 & 0 \end{bmatrix} \begin{bmatrix} q_{i-1} \\ q_{i-2} \end{bmatrix} = \mathbf{A}_i \begin{bmatrix} q_{i-1} \\ q_{i-2} \end{bmatrix}.$$

Note that we can substitute $\mathbf{A}_{i-1}(q_{i-2}q_{i-3})^T$ for $(q_{i-1}q_{i-2})^T$ above and can continue this substitution repeatedly until we obtain

$$\begin{bmatrix} q_i \\ q_{i-1} \end{bmatrix} = \mathbf{A}_i \mathbf{A}_{i-1} \cdots \mathbf{A}_2 \begin{bmatrix} q_1 \\ q_0 \end{bmatrix}. \tag{17}$$

This formulation of the problem is ideal for recursive doubling. Since matrix multiplication is associative, we can evaluate the product $\mathbf{A}_i\mathbf{A}_{i-1} \cdots \mathbf{A}_2$ in exactly the same way that we evaluate a product of scalars. In fact, we have encountered this problem before in (12), and the recursive doubling solution is the schema of (14). Then to solve (15) for all q_i simultaneously requires $\log_2 N$ iterations, in which the ith iteration involves the 2^{N-i} simultaneous calculations of the product of two 2×2 matrices.

It is rather interesting to investigate the properties of the functions q_i because it is possible to exploit their characteristics and obtain a parallel algorithm slightly more efficient than the solution to (17) described above. Fortunately, a great deal is known about these functions. One important property is well illustrated by the first few q_i :

$$\begin{aligned} q_0 &= 1 \\ q_1 &= d_1 \\ q_2 &= d_2d_1 - e_2f_1 \\ q_3 &= d_3d_2d_1 - d_3e_2f_1 - e_3f_2d_1 \\ q_4 &= d_4d_3d_2d_1 - d_3d_3e_2f_1 - d_4e_3f_2d_1 - e_4f_3d_2d_1 + e_4f_3e_2f_1 . \end{aligned}$$

Knuth [7] attributes to Euler [3] the observation that q_i contains the term $d_id_{i-1} \cdots d_1$, together with every term that can be constructed by replacing d_jd_{j-1} by $-e_jf_{j-1}$ for all possible combinations of such pairs. This property follows directly from the recurrence relation (15). The first product in (15), d_iq_{i-1}, creates terms in q_i for which adjacent d-pairs are deleted from among only the coefficients $d_1, d_2, \cdots, d_{i-1}$ in all possible ways, and thus produces every possible way there can be terms containing d_i. The second product in (15) replaces d_id_{i-1} by $-e_if_{i-1}$, and combines this with every possible way d-pairs can be eliminated among the coefficients $d_1, d_2, \cdots, d_{i-2}$. This produces every possible term without d_i.

We can obtain factorizations of the q_i functions that correspond to the intermediate results in the evaluation of (17). To arrive at these factorizations, let us define $Q_i(j)$ for $j \geq i$ to be the function q_i with the subscripts of its arguments increased systematically so that the leading subscript is j. For $j < i$, we define $Q_i(j) = Q_j(j)$. Some examples of $Q_i(j)$ should clarify ambiguities in the definition:

$$\begin{aligned} Q_1(1) &= d_1 \\ Q_1(2) &= d_2 \\ Q_3(3) &= d_3d_2d_1 - d_3e_2f_1 - e_3f_2d_1 \\ Q_3(4) &= d_4d_3d_2 - d_4e_3f_2 - e_4f_3d_2 \\ Q_3(2) &= Q_2(2) = d_2d_1 - e_2f_1 . \end{aligned}$$

From this definition it now follows directly that the Q_i functions satisfy the recurrence

$$Q_{i+s}(j) = Q_s(j)Q_i(j-s) - e_{j-s+1}f_{j-s}\,Q_{s-1}(j)Q_{i-1}(j-s-1) \quad \text{for } j \geq s,\ i \geq 1 \tag{18}$$

with the boundary conditions

$$\begin{aligned} Q_1(j) &= d_j \qquad \text{for } j \geq 1, \\ Q_i(j) &= 1 \qquad \text{for } j \geq 0,\ i \leq 0, \\ Q_i(j) &= 1 \qquad \text{for } j \leq 0,\ i \geq 0, \\ e_{j+1}f_j &= 0 \qquad \text{for } j \leq 0. \end{aligned}$$

This recurrence formulation is also well known, with citations in the literature at least as early as 1853 [8, 9].

The validity of (18) can be verified by an intuitive argument. To find all possible ways of eliminating adjacent d-pairs in a sequence of $i + s$ coefficients, combine every possible way of eliminating pairs in the first s coefficients with every possible way of eliminating pairs in the last i coefficients. This accounts for the first term of (18). However, one d pair contains the last coefficient from the set of s coefficients and the first coefficient from the set of i coefficients. The first term in (18) does not account for any of the ways this pair can be eliminated. We see that the second term in (18) accounts for all such ways, because $e_{j-s+1}f_{j-s}$ replaces the pair and this replacement is combined with every possible way of eliminating pairs in the first $s - 1$ coefficients and in the last $i - 1$ coefficients. From (18) we obtain the recursive doubling formulae.

THEOREM 2. *$Q_i(j)$ satisfies the recurrence relations*

$$\begin{aligned} Q_{2i}(j) &= Q_i(j)Q_i(j-i) + (-e_{j-i+1}f_{j-i})Q_{i-1}(j)\,Q_{i-1}(j-i-1), \\ Q_{2i-1}(j) &= Q_i(j)Q_{i-1}(j-i) + (-e_{j-i+1}f_{j-i})Q_{i-1}(j)Q_{i-2}(j-i-1), \\ Q_{2i-2}(j) &= Q_{i-1}(j)Q_{i-1}(j-i+1) + (-e_{j-i+2}f_{j-i+1})Q_{i-2}(j)Q_{i-2}(j-i). \end{aligned} \tag{19}$$

PROOF. These formulas follow directly from (18).

The first of the equations in Theorem 2 is a recursive doubling formula which shows that Q_{2i} depends on both Q_i and Q_{i-1}. Hence, to compute Q_{4i} we need to compute both Q_{2i} and Q_{2i-1}. To compute Q_{4i-1} we have to compute Q_{2i-2}. Since Q_{2i-2} depends on the same quantities as Q_{2i} and Q_{2i-1}, we need only the three equations (19) in a recursive doubling algorithm. Sinse we have to compute Q_{2i-1} and Q_{2i-2} anyway, it is slightly more efficient to compute Q_{2i} by the formula

$$Q_{2i}(j) = d_jQ_{2i-1}(j-1) + (-e_jf_{j-1})Q_{2i-2}(j-2).$$

The complete algorithm to compute q_i, $1 \leq i \leq N$ is given below in an ALGOL-like language. The initial conditions establish the values of Q_0, Q_1, and Q_2. The first iteration computes Q_2, Q_3, and Q_4, the second iteration computes Q_6, Q_7, and Q_8, and the last iteration computes Q_{N-2}, Q_{N-1}, and Q_N.

```
begin
  real array E[2:N], F[1:N-1], D[1:N], EF[1:N],
    TEMP[1 N], QI[1:N], QIM1[0:N], QIM2[-1:N];
  comment the arrays hold the quantities indicated below
    E     the lower diagonal of the tridiagonal matrix A.
    F     the upper diagonal of A.
    D     the major diagonal of A.
    EF    this holds products of the form -e_i f_{i-1}.
    TEMP  a temporary array.
    QI    holds Q_i(j).
    QIM1  holds Q_{i-1}(j).
    QIM2  holds Q_{i-2}(j).
```

```
    The computation begins by initializing EF, QI, QIM1, and QIM2;
  initialize:
    EF[1] := 0;
    EF[i] := −E[i] × F[i−1],  (2 ≤ i ≤ N);
    QIM2[i] := 1,  (−1 ≤ i ≤ N);
    QIM1[0] := 1;
    QIM1[i] := D[i],  (1 ≤ i ≤ N);
    QI[i] := D[i] × D[i−1] + EF[i],  (2 ≤ i ≤ N);
    QI[1] := D[1];
  comment the last three lines initialize the arrays to Q0, Q1 , and Q2 , respectively;
for i := 2 step i until N/2 do
  begin
    TEMP[j] := QIM1[j] × QIM1[j−i+1] + EF[j−i+2] × QIM2[j] × QIM2[j−i],
      (i−1 ≤ j ≤ N);
    comment TEMP contains Q2i−2 . It cannot be written over Qi−2 yet since Qi−2 is needed
      in the next line;
    QIM1[j] := QI[j] × QIM1[j−i] + EF[j−i+1] × QIM1[j] × QIM2[j−i−1],
      (i ≤ j ≤ N);
    QIM2[j] := TEMP[j],  (i−1 ≤ j ≤ N);
    QI[j] := D[j] × QIM1[j−1] + EF[j] × QIM2[j−2],  (i+1 ≤ j ≤ N);
  end;
```

At the termination of the algorithm, $QI[i]$ contains q_i for $1 \leq i \leq N$. We use (16) to compute the diagonal of **U** from the q_i's. This clearly can be done in parallel by dividing the vector QI by a shift of itself. Finally, to compute the subdiagonal of **L**, we note that (2) indicates that this computation can be done by one parallel division.

In executing the algorithm on an ILLIAC IV class of computer, the vector alignment required for the calculation is done by cyclically shifting vectors among the processors. Since the algorithm requires that $QI[j] = QIM1[j] = QIM2[j] = 1$ for $j \leq 0$, we can avoid storing these quantities by changing the cyclic shift of these vectors to an end-off shift in which the integer 1 is shifted into element 1 of each of these vectors. Similarly, $EF[j] = 0$ for $j \leq 1$, so that 0's are always shifted into $EF[2]$ when the EF vector is aligned.

The ranges indicated for each statement in the basic iteration show the positions of the vectors which change when that statement is executed. The algorithm will work correctly when all ranges are replaced by the full range $1 \leq i \leq N$ since values that do not change are recomputed at each step. It is somewhat more efficient to use the full range for a calculation than the ranges given, although redundant recomputation of values may be accompanied by greater round-off error.

The serial solution of a tridiagonal system of equations, when done as outlined in Section 2, requires $3(N - 1)$ of each of the operators division, multiplication, and subtraction. That schema requires the same number of operations on both parallel and serial computers. The parallel computation has three loops, each executed $\log_2 N$ times. The loop that computes the LU decomposition requires eight parallel multiplications and three parallel additions per iteration, whereas the forward and back substitutions each require two parallel multiplications and one parallel addition per iteration. Apart from the computations within loops, there are at least four parallel divisions, two parallel multiplications, and one parallel addition applied to N elements simultaneously.

Hence the operation count for the parallel algorithm (exclusive of overhead com-

putations) is

$$\begin{aligned} &12 \log_2 N + 2 \text{ parallel multiplications,} \\ &5 \log_2 N + 1 \text{ parallel additions,} \\ &4 \qquad\qquad \text{parallel divisions.} \end{aligned}$$

The reduction in the number of divisions is particularly important for computers which take much longer to divide than to multiply. (On the ILLIAC IV computer division is approximately five times longer than multiplication.)

At this writing the stability of the algorithm has not been thoroughly investigated. Clearly, the algorithm is unstable if any q_i vanishes. Since $q_i = \prod_{j=1}^{i} u_j$, q_i vanishes if and only if one of the u_i coefficients vanishes. However, if the **A** matrix is diagonally dominant and nonsingular, every u_i is bounded away from zero [6].

5. *Summary and Conclusions*

The parallel algorithm for the solution of tridiagonal systems of linear equations consists of two different algorithms. One algorithm is the parallel evaluation of first-order difference equations of the form

$$x_i = b_i x_{i-1} + c_i$$

where the b_i and c_i are constants.

The second algorithm solves second-order equations of the form

$$x_i = b_i x_{i-1} + c_i x_{i-2} .$$

Since continued fraction expansions are associated with second-order difference equations, the second algorithm may also be used to compute continued fraction expansions. The form of the solution obviously generalizes to linear recurrence relations of arbitrary mth order, still requiring $\log_2 N$ iterations, where each iteration involves simultaneous multiplications of $m \times m$ matrices.

ACKNOWLEDGMENT. The author expresses his appreciation to William Jones and David Galant of NASA Ames Research Center for their many conversations, comments, and criticisms which materially aided the research. He is also grateful to Donald Knuth of Stanford University for pointing out the early contributions to the factorization of second-order recurrence relations. The recursive doubling algorithm for solving first-order recurrence relations was discovered independently by Harvard Lomax of NASA Ames Research Center and by Robert Downs of Systems Control, Inc. Gene Golub of Stanford University pointed out Buneman's algorithm as an alternative method for solving tridiagonal systems in a time proportional to $\log_2 N$.

REFERENCES

1. BUNEMAN, OSCAR A compact non-iterative Poisson solver. Rep. 294, Inst. for Plasma Res., Stanford U., Stanford, Calif., 1969.
2. BUZBEE, B. L , GOLUB, G. H , AND NIELSON, C. W. On direct methods for solving Poisson's equations. *SIAM J. Numer. Anal 7*, 4 (Dec. 1970), 627–656.
3. EULER, LEONHARD. *Introductio in Analysin Infinitorum*, Lausanne, 1748, Sec. 359.
4. FORSYTHE, G. E., AND MOLER, C. B. *Computer Solution of Linear Algebraic Systems*. Prentice-Hall, Englewood Cliffs, N. J., 1967.

5. GAUTSCHI, WALTER. Computational aspects of three-term recurrence relations, *SIAM Rev. 9*, 1, (Jan. 1967), 24–82.
6. ISAACSON, E., AND KELLER, H. B. *Analysis of Numerical Methods*. Wiley, New York, 1966.
7. KNUTH, D. E. Mathematical analysis of algorithms. Rep. Stan-CS-71-206, Computer Sci. Dep., Stanford U., Stanford, Calif., Mar. 1971.
8. PERRON, O. *Die Lehre von den Kettenbruchen*. Leipzig, 1913.
9. SYLVESTER, J. J. *Philosophical Magazine 6*, (1853), 297–299.
10. WALL, H. S. *Analytic Theory of Continued Fractions*. Van Nostrand, New York, 1948.

RECEIVED DECEMBER 1971; REVISED MARCH 1972