A Parallel Method for Tridiagonal Equations

H. H. WANG
IBM Scientific Center

A new (partition) method for solving a tridiagonal system of linear equations is presented in this paper The method is suitable for both parallel and vector computers. Although the partition method has a slightly higher vector operation count than those of the two competing methods (the recursive doubling method and the cyclic reduction method), it has a scalar count much smaller than that of the recursive doubling. The scalar counts between the partition method and the cyclic reduction method are so close as to make a timing evaluation inconclusive without considering the data management problem, especially when large systems are solved. Various situations under which the partition method can be preferable are described.

Key Words and Phrases: tridiagonal equations, parallel computers, vector computers, partition method, cyclic reduction method, recursive doubling method
CR Categories: 5 14, 6.22

1. INTRODUCTION

The solution of a tridiagonal system of linear equations lies at the heart of many programs for scientific computation. With the recent development and availability of various parallel and vector computers, new algorithms have appeared for solving tridiagonal systems of equations suitable for these machines. Notable among these methods are the recursive doubling method (Stone [8]) and the cyclic reduction method (Lambiotte and Voigt [5]). The recursive doubling method is designed for a parallel computer such as the Illiac IV. Such a computer typically consists of an aggregate of M identical processors, each capable of executing the same instruction at the same time. The gain in speed on a parallel computer is proportional to the number of processors being gainfully utilized in the computation. Although the cyclic reduction method can be efficiently applied on a parallel computer, it is most effective on a vector computer, such as the Control Data Corporation STAR-100 or CRAY I. The gain in speed on a vector computer is made possible by streaming vectors of operands through a pipelined arithmetic unit.

In this paper we present a new parallel method of solving the tridiagonal equations suitable for both parallel and vector computers. (In the following sections, we refer to the new method as the "partition method.") Although the partition method has a slightly higher vector operation count than the recursive doubling method, it has a smaller scalar operation count. Hence there should be

Author's address. IBM Scientific Center, 1530 Page Mill Road, Palo Alto, CA 94304.

situations for which the partition method ought to be considered. One such situation is the solution of a tridiagonal system on a parallel machine with M processors, where M is much less than the order of the system of equations. The partition method may also be preferred on a vector machine over the cyclic reduction method because of its versatility and its simple data management requirement. We expand on these points later. First, however, in Section 2 we discuss briefly the chracteristics of a parallel and a vector computer. The partition method is then presented in subsequent sections.

2. CHARACTERISTICS OF A PARALLEL/VECTOR COMPUTER

The parallel and vector computers with which we are concerned belong to a class of computers called the single instruction stream-multiple data stream (SIMD) machines. The more general multiple instruction stream-multiple data stream (MIMD) machines capable of executing different instructions simultaneously are not considered here. SIMD machines are best suited for algorithms requiring the same operations on large arrays of independent data. An algorithm designed for a vector machine is often applicable to a parallel machine, and vice versa, except for some algorithms designed for parallel machines with an unlimited number of processors. However, there are important differences between the two types of machines. Notably the timing considerations for a parallel computer are much different than those for a vector computer. Also the characteristics of each individual machine, such as the size of available central memory, the data accessibility, the instruction set, and so on, can greatly influence the applicability of a particular algorithm.

Consider a parallel computation involving L elements. The timing T_p on a parallel computer is given by

$$T_p = T_{op}\,[L/M], \tag{1}$$

where M is the number of processors available, T_{op} is the time required per parallel operation, and $[L/M]$ is the smallest integer equal to or greater than L/M. Thus, when $L = 33$ and $M = 32$, then $[L/M] = 2$ and two parallel computations must be performed. Two important observations can be made immediately from the form of eq. (1):

(1) The parallel machine is computing at maximum speed if parallel operations are operating on a length $L = M$, or if L is an integer multiple of M.
(2) For L greater than M, T_p also reflects the total number of scalar operations performed.

As noted before, the main approach to enhanced speed of operation in a vector machine is pipelining. A pipelined arithmetic unit is functionally divided into subunits. Each subunit performs a specific task that is a part of the overall arithmetic operation. Concurrent operations are achieved by streaming vectors of operands through the pipeline (arithmetic unit). Assuming each subunit takes time T_c to perform its task, then, after the first result has emerged from the pipeline, subsequent results can be completed at the rate of one per every T_c. Hence the time T_v for a vector operation can be expressed by

$$T_v = S + (L - 1)T_c \tag{2}$$

where S, called the vector start-up time, is the time elapsed after the vector instruction is first issued and just before the first result emerges from the pipeline. To enhance the effective data flow rate to match the execution speed of the arithmetic unit, interleaving of the memory is often employed in high-performance computers. (See, for instance, Lorin [6].) For an n-way interleaved memory bank, an n-fold increase in data rate can be achieved if no bank conflicts are encountered.

3. A NEW PARALLEL ALGORITHM

Consider the tridiagonal system of linear equations

$$Ax \equiv \begin{bmatrix} a_1 & b_1 & & & & & \\ c_2 & a_2 & b_2 & & & & \\ & c_3 & a_3 & b_3 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & & & b_{n-1} & \\ & & & & c_n & a_n & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ \vdots \\ r_n \end{bmatrix} \equiv r. \tag{3}$$

A unique solution x exists for given right-hand side r and nonsingular coefficient matrix A. The new method, which we call the "partition" method, is based on the notion of "divide and conquer." The system (matrix A) is first partitioned into subsystems, after which elimination can proceed simultaneously on all subsystems by elementary row transformations until finally A is diagonalized. We first illustrate the elimination pattern via an example prior to formally presenting the algorithm. Consider a tridiagonal system of 16 equations.

(a) Partition the matrix into, say, a 4×4 block tridiagonal form as shown in Figure 1.

(b) Eliminate c_2, c_6, c_{10}, c_{14} simultaneously, then eliminate c_3, c_7, c_{11}, c_{15} and c_4, c_8, c_{12}, c_{16} simultaneously. The matrix is now triangular except for the fourth, the eighth, and the twelfth columns. The f's, shown in Figure 2, are nonzero elements called "fill-ins" created during the elimination.

(c) Next, eliminate b_2, b_6, b_{10}, b_{14} simultaneously, then b_1, b_5, b_9, b_{13} and b_4, b_8, b_{12}. This leaves us with a diagonal matrix, except for the fourth, the eighth, the twelfth, and the sixteenth columns, as shown in Figure 2. The g's are fill-ins created during this step.

(d) The matrix can then be triangularized by the elimination of c_5, f_6, f_7, f_8, followed by c_9, f_{10}, f_{11}, f_{12} and c_{13}, f_{14}, f_{15}, f_{16}.

(e) The matrix is diagonalized by eliminating b_{15}, g_{14}, g_{13}, g_{12}; b_{11}, g_{10}, g_9, g_8; b_7, g_6, g_5, g_4; and b_3, g_2, g_1.

No fill-ins are created during the last two steps.

In general, for a system of n equations and for an arbitrary positive integer $p > 0$, the partition algorithm can be described as follows (assuming that n is divisible by p and $k \equiv n/p \geq 2$):

(1) Partition A into $p \times p$ block tridiagonal form with each diagonal block a $k \times k$ tridiagonal matrix and each subdiagonal (superdiagonal) block a $k \times k$

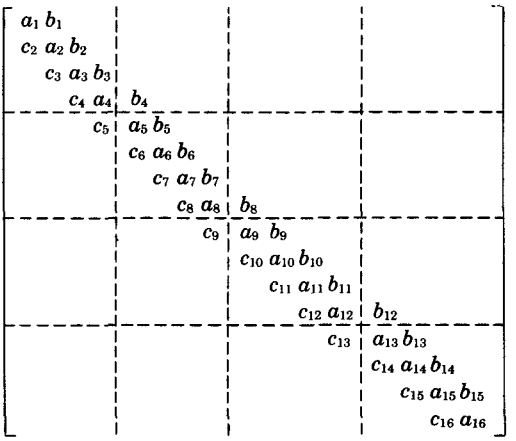

Figure 1

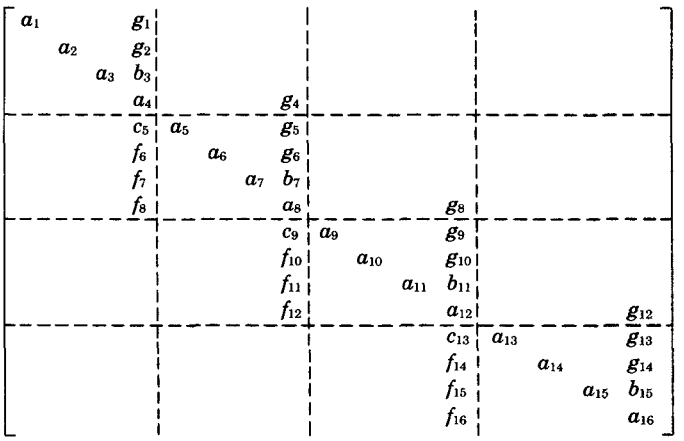

Figure 2

null matrix, except for one single nonzero element on its upper right (lower left) corner. (See Figure 1.)

(2) Apply elementary row transformations on all the p diagonal blocks simultaneously such that each block is transformed into an upper triangular matrix. This process creates fill-ins (f's in Figure 2) in the subdiagonal blocks. That is, the rightmost column of each subdiagonal block is now completely filled.

The processing involved is given by

$$f_{ik+1} \leftarrow c_{ik+1}, \qquad i = 1, \ldots, p-1; \tag{4}$$

for $j = 2, \ldots, k$, do (5)-(8):

$$c_{ik+j} \leftarrow \frac{c_{ik+j}}{a_{ik+j-1}} \tag{5}$$

$$a_{ik+j} \leftarrow a_{ik+j} - c_{ik+j} * b_{ik+j-1} \qquad i = 0, 1, \ldots, p-1 \tag{6}$$

$$r_{ik+j} \leftarrow r_{ik+j} - c_{ik+j} * r_{ik+j-1} \tag{7}$$

$$f_{ik+j} \leftarrow -c_{ik+j} * f_{ik+j-1}, \qquad i = 1, \ldots, p-1. \tag{8}$$

(3) Elimination continues on the superdiagonals of the diagonal blocks, and finally the nonzero elements of the superdiagonal blocks are also eliminated. This process again creates fill-ins, g's (Figure 2). Now the entire matrix is diagonal, except in the ikth columns for $i = 1, \ldots, p$, as shown in Figure 2.

The processing involved is given by

$$g_{ik-1} \leftarrow b_{ik-1}, \qquad i = 1, \ldots, p; \tag{9}$$

for $j = k - 1, k - 2, \ldots, 2$, do (10)-(13):

$$\left.\begin{aligned} b_{ik+j-1} &\leftarrow \frac{b_{ik+j-1}}{a_{ik+j}} && (10)\\ g_{ik+j-1} &\leftarrow -b_{ik+j-1} * g_{ik+j} && (11)\\ r_{ik+j-1} &\leftarrow r_{ik+j-1} - b_{ik+j-1} * r_{ik+j} && (12) \end{aligned}\right\} \quad i = 0, 1, \ldots, p-1$$

$$f_{ik+j-1} \leftarrow f_{ik+j-1} - b_{ik+j-1} * f_{ik+j}, \qquad i = 1, \ldots, p-1; \tag{13}$$

and also do (14)-(17):

$$\left.\begin{aligned} b_{ik} &\leftarrow \frac{b_{ik}}{a_{ik+1}} && (14)\\ a_{ik} &\leftarrow a_{ik} - b_{ik} * f_{ik+1} && (15)\\ g_{ik} &\leftarrow -b_{ik} * g_{ik+1} && (16)\\ r_{ik} &\leftarrow r_{ik} - b_{ik} * r_{ik+1} && (17) \end{aligned}\right\} \quad i = 1, \ldots, p-1.$$

We point out here that steps (10)-(13) are skipped if the choice $k = 2$ is made.

(4) Elimination can continue below the main diagonal on the ikth column for $i = 1, \ldots, p - 1$. This process creates no fill-ins. At the end of this step, the matrix is in triangular form.

The processing involved in this step is given by

For $i = 1, \ldots, p - 1$ do (18)-(23):

$$t_j \leftarrow g_{ik+j-1}, \qquad j = 1, \ldots, k \tag{18}$$

$$t_{k+1} \leftarrow a_{(i+1)k} \tag{19}$$

$$\left.\begin{aligned} f_{ik+j} &\leftarrow \frac{f_{ik+j}}{a_{ik}} && (20)\\ t_{j+1} &\leftarrow t_{j+1} - f_{ik+j} * t_1 && (21)\\ r_{ik+j} &\leftarrow r_{ik+j} - f_{ik+j} * r_{ik} && (22) \end{aligned}\right\} \quad j = 1, \ldots, k$$

$$a_{(i+1)k} \leftarrow t_{k+1} \tag{23}$$

where t is a temporary array of $(k + 1)$ elements.

(5) The matrix can be diagonalized by eliminating the elements on the ikth column ($i = p, \ldots, 1$) above the main diagonal. Again, no fill-ins are generated during this step.

The processing involved is given by the following:

For $i = p - 1, \ldots, 1$ do (24)-(25):

$$\left.\begin{aligned} g_{ik+j-1} &\leftarrow \frac{g_{ik+j-1}}{a_{(i+1)k}} \qquad (24)\\ r_{ik+j-1} &\leftarrow r_{ik+j-1} - g_{ik+j-1} * r_{(i+1)k} \qquad (25)\end{aligned}\right\} \quad j = 1, \ldots, k.$$

Also do (26) and (27):

$$\left.\begin{aligned} g_j &\leftarrow \frac{g_j}{a_k} \qquad (26)\\ r_j &\leftarrow r_j - g_j * r_k \qquad (27)\end{aligned}\right\} \quad j = 1, \ldots, k-1.$$

(6) The solution can now be computed by

$$x_i = \frac{r_i}{a_i}, \qquad i = 1, \ldots, n. \tag{28}$$

This completes the algorithm description. Note that we have included the right-hand-side computation along with the elimination in the above algorithm. In addition, operations defined by (18), (19), and (23) can be avoided if we store the g's in a slightly bigger array; for instance, in the 16-equation example above, let G be a 5×4 array such that

$$G = \begin{bmatrix} 0 & g_2 & g_8 & g_{12} \\ g_1 & g_5 & g_9 & g_{13} \\ g_2 & g_6 & g_{10} & g_{14} \\ b_3 & b_7 & b_{11} & b_{15} \\ a_4 & a_8 & a_{12} & a_{16} \end{bmatrix}. \tag{29}$$

Instead of (18), (19), and (23), we need only one vector operation $G_{k+1,j} \leftarrow a_{jk}$, $j = 1, \ldots, p$ at the beginning of step (4) and an inverse operation $a_{jk} \leftarrow G_{k+1,j}$, $j = 1, \ldots, p$ at the end of step (4). In this case, of course, we must replace t_{j+1} by $G_{j+1,i+1}$ and t_j by $G_{1,i+1}$, respectively, in (21). In addition, all the g's must be replaced by the corresponding G's in the algorithm.

4. OPERATION COUNTS

The partition algorithm given in the last section is suitable for both the parallel and the vector computer. We now give arithmetic counts for both the vector and the scalar operations. We also discuss the data management problem, which becomes particularly important when choosing an algorithm for parallel/vector processing. Table I lists the vector operation counts, while Table II lists the scalar operation count for each step of the algorithm. Operation counts for right-hand-side computation are listed separately for easy referencing.

To implement the algorithm on a parallel computer with an unlimited number of processors [or at least equal to max (p, k)], the only numbers to be concerned with are the vector operation counts. However, on machines with M processors

Table I. Vector Operation Counts
($n = kp$)

	Diagonalization			Right-side computation		
	Divide	Multiply	Add	Divide	Multiply	Add
Step (2)						
Length p	$k-1$	$k-1$	$k-1$		$k-1$	$k-1$
Length $p-1$		$k-1$				
Step (3)						
Length p	$k-2$	$k-2$			$k-2$	$k-2$
Length $p-1$	1	k	$k-1$		1	1
Step (4)						
Length k	$p-1$	$p-1$	$p-1$		$p-1$	$p-1$
Step (5)						
Length k	$p-1$				$p-1$	$p-1$
Length $k-1$	1				1	1
Step (6)						
Length n				1		
Total for steps (2)-(6)	$2k+2p-3$	$4k+p-5$	$2k+p-3$	1	$2k+2p-3$	$2k+2p-3$
Total number of vector OPS		$8k+4p-11$			$4k+4p-5$	

less than either p or k (or both), the length L of each vector operation (hence the scalar operation counts) must also be taken into account since each operation now requires $[L/M]$ parallel operations. On a vector machine, both the vector and the scalar counts are important. The vector counts become less significant when the vector lengths (p and k) become large.

Several observations pertinent to the following discussion can be made:

(1) The vector implementation of the partition algorithm is consistent (in the sense of Lambiotte and Voigt [5]), since its scalar mathematical operations are of order $O(n)$, which is the same order required by the usual serial algorithm (Gaussian elimination).

(2) The total number of vector operations for the algorithm is (Table I) $12k + 8p - 16$, which is at a minimum when $k \cong 0.8\sqrt{n}$ and $p \cong 1.25\sqrt{n}$.

(3) The minimum total scalar operation count occurs at $p = n/2$ and is equal to $15n$.

(4) Assume that data are stored according to their natural ordering. That is, a_1, $a_2, \ldots, a_n$; b_1, $b_2, \ldots$ are stored in consecutive memory locations. Then all the elements of the operand vectors in steps (2) and (3) are spaced k-words apart in memory, while in steps (4)-(6) the elements are in consecutive memory locations.

(5) For implementation on a parallel computer with M processors with $M \ll n$, it would be easy to choose p (hence k) such that both p and k are divisible by M. In this case the parallel processors will be operating at near optimal efficiency. For instance, if $n = 2048$, $M = 16$, a good choice would be $p = 64$ (hence $k = 32$).

(6) The total fill-ins (sum of f's and g's) created by the algorithm is given by $2n$

Table II. Scalar Operation Counts
($n = kp$)

	Diagonalization			Right-side computation		
	Divide	Multiply	Add	Divide	Multiply	Add
Step (2)	$n - p$	$2n - 2p - k + 1$	$n - p$		$n - p$	$n - p$
Step (3)	$n - p - 1$	$2n - 2p - k$	$n - p - k + 1$		$n - p - 1$	$n - p - 1$
Step (4)	$n - k$	$n - k$	$n - k$		$n - k$	$n - k$
Step (5)	$n - 1$				$n - 1$	$n - 1$
Step (6)				n		
Total for steps (2)-(6)	$4n - 2p - k - 2$	$5n - 4p - 3k + 1$	$3n - 2p - 2k + 1$	n	$4n - 2p - k - 2$	$4n - 2p - k - 2$
Total numbers of scalar OPS		$12n - 8p - 6k$			$9n - 4p - 2k - 4$	

Table III. Sample Operation Counts

n	p	k	Total vector count	Total scalar count (in thousands)
	8	8	144	1.2
64	16	4	160	1.1
	32	2	256	.9
	16	16	304	5.1
256	32	8	336	4.9
	128	2	1032	3.8
	32	32	624	20 9
1024	128	8	1104	19 9
	512	2	4096	15.3

$- 2p - k$, which is slightly less than $2n - 2$, the total number of b's and c's. If solution x is needed for only one right-hand side r, then no temporary store is needed [except perhaps the small array t in step (4)] for implementation. The fill-ins f's and g's can be temporarily stored in c's and b's, respectively.

(7) The number of fill-ins is at the minimum when $p = n/2$. This coincides with the minimum condition for scalar operation count. However, this condition also makes the vector operation counts large in steps (4) and (5) of the algorithm.

(8) When solutions with successively generated right-hand sides but with fixed coefficient matrix are required, the diagonalization part needs to be done only once. The information on all the transformations can be recalled by storing a, b, c, f, and g. The solution of each right-hand side can be obtained by the "right-side computation" only. (See Tables I and II and the algorithm description.) If division is expensive, we can trade the n divisions (Table II) for n multiplications by storing the reciprocal elements of the vector a.

(9) When implementing the algorithm in a paged environment with the storage scheme outlined in (5) above, large page demands may result. To reduce these demands, the following scheme may be used. Typically, store the array a as

$$(a_1, a_{k+1}, \ldots, a_{(p-1)k+1}, a_2, a_{k+2}, \ldots, a_{(p-1)k+2}, \ldots, a_{pk}).$$

At the end of step (3), a transpose operation is applied to the data to rearrange the elements in their natural ordering. This puts the elements of all the operand vectors in consecutive memory locations. This storage scheme can also be used on computers whose vector instructions are architected to operate only with vector operands whose elements are stored contiguously in memory. We point out here that very efficient algorithms exist for transposing matrices. See [2] and [7] for details. These transposition algorithms apply immediately to both parallel and vector computers (see Wang [10]).

In Table III, we list operation counts for some typical sizes of n. We conclude that it is advisable to choose a value of p that is about the same value as k. To choose otherwise would result in a large increase in vector count and only a small decrease in scalar count.

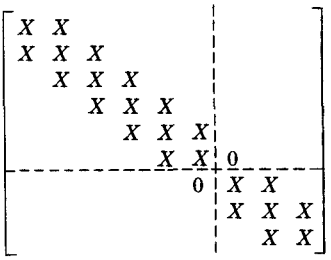

Fig. 3. Structure of a linked parallel system.

5. SOLUTION OF MULTIPLE INDEPENDENT SYSTEMS

Often in practice, $m > 1$ independent systems of tridiagonal linear equations need to be solved. For example, when the ADI method (see Wachspress [9]) is used for the numerical solution of the elliptic partial differential equation. The partition algorithm can be used in two ways in this situation.

(1) The most obvious way is to apply the partition algorithm successively to m systems.

(2) We can take advantage of the independence of the systems and apply the partition algorithm to a single large tridiagonal system. For instance, Figure 3 shows the structure of a single system that consists of two independent systems, each with six and three equations, respectively. Lambiotte [4] called this approach the "linked parallel vectorization." Note that we can take advantage of the zeros on the upper and the lower diagonals of the linked matrix by choosing a p such that fewer fill-ins are created, and hence less work needs to be performed in steps (4) and (5) of the algorithm. In the example of Figure 3, for instance, we can choose $p = 3$. However, the percentage of saving is small for large linked systems, except for the case when m systems of n equations each are linked and a choice of $p = m$ is made. In that case, the linked approach reduces to the Gaussian elimination method (simultaneously applied to m systems). Therefore, no fill-ins are generated, which results in a minimum scalar operation count.

It is easy to calculate the operation counts for the two approaches discussed in this section by using Tables I and II for the solution of m independent systems of n equations each. In Table IV we list operation counts for sample values of n and m. We have also included operation counts for the Gaussian elimination method. The vector and scalar counts for the Gaussian elimination method applied to the same m systems simultaneously are $8n$ and $8nm$, respectively.

We note that, on a vector computer, the linked approach should be preferred over the successive approach because of its smaller vector counts. However, the linked partition method cannot compete with the Gaussian elimination method, except, perhaps, for the case when both n/m and S/T_c in (2) are large ($\gg 1$). These observations are also true on a parallel computer with a number of processors $M \geq \max(p, k)$.

Table IV. Sample Operation Counts

Size						
n	m	Method	p	k	Vector count (in hundreds)	Scalar count (in thousands)
100	100	Successive	10	10	154	190
		Linked[a]	125	80	19	208
		Gaussian			8	80
1200	10	Successive	40	30	54	245
		Linked[a]	120	100	18	245
		Gaussian			96	96

[a] The actual counts for the linked method should be slightly smaller if the savings from zeros in b and c of the linked matrix are accounted for

6. COMPARISON WITH EXISTING PARALLEL METHODS

In this section we compare the partition method with the recursive doubling method and the cyclic reduction method for solving tridiagonal systems of linear equations. Table V gives the operation counts for the three methods.

As noted before, the vector count for the partition method is near minimum when $p = \sqrt{n}$. Hence, the best we can say is that the partition method has a vector count proportional to $\sqrt{n}$. The vector counts for both the recursive doubling method and the cyclic reduction method are proportional to $\log_2 n$. Since $\sqrt{n}$ increases more rapidly than $\log_2 n$ as n increases, the vector count of the partition method will always be the highest. Therefore, we cannot recommend the new method on an M-processor parallel computer with $M \geq n$. However, on a parallel machine with $M < n$ processors, the situation is not that clearcut. The length of each vector operation (and hence the number of scalar operations) now determines how many parallel operations must be performed (see eq. (1)). Most probably we can rule out the recursive doubling method since it has a scalar count with order higher than the other two methods. Stone [8] devised a modified recursive doubling procedure with a lower scalar count that is consistent. But the count is still about twice as big as either of the two competing methods.

The partition method is related to the cyclic reduction method in the following situation. If one executes steps (1)–(3) with a partitioning of $p = n/2$, one has eliminated the odd variables from the even equations, and the resulting system is a tridiagonal set of equations involving the even variables. This is equivalent to the first step of cyclic reduction. The two algorithms differ at this point since cyclic reduction repeats the procedure on the smaller system, whereas the partition method proceeds to diagonalize the matrix.

The partition method also resembles the one-way dissection method due to George [3]. The last variable in each subsystem corresponds to a separator in the graph of the tridiagonal matrix. But the positions in which fill-ins are created are different in the two approaches.

If we do not distinguish among divisions, multiplications, and additions, then the total scalar arithmetic counts for the partition method and the cyclic reduction method are $21n$ and $20n$, respectively. When solutions for additional right-hand sides are required, the counts of the two methods are identical. Since the partition algorithm has a larger vector count than the cyclic reduction algorithm, the latter

Table V. Operation Counts for Tridiagonal Solvers

	Partition	Recursive doubling[a]	Cyclic reduction[a]
Vector count	$2(k+p-1)$ D $+(6k+3p-8)$ M $+(4k+3p-6)$ A	D $+(12q-3)$ M $+(5q-1)$ A	qD $+(12q+1)$ M $+6q$A
Scalar count	$(5n-2p-k)$ D $+(9n-6p-4k)$ M $+(7n-4p-3k)$ A	nD $+(5n+12qn_a)$ M $+(n+5qn_a)$ A	n(D + 13M + 6A)
Scalar count for additional solution after first solution	$(5n-2p-k)$ M $+(4n-2p-k)$ A	qn_a(4M + 2A)	n(5M + 4A)

Note: $q = [\log_2 n]$, $kp = n$, $n_a \cong n$ (an exact formula for n_a is given by Lambiotte [4, eq. (4.9)]).
D: division, M. multiplication; A: addition.
[a] Data extracted from [4, Tables IV and V].

should be preferred, everything else being equal. We present below three situations under which the partition algorithm can be preferable.

(1) On an M-processor parallel computer with $M \ll n$, then the processors are most likely to operate at maximum efficiency if the partition algorithm is used. (See observation (6) in Section 3.) Hence, the partition algorithm will require the least parallel operations, particularly if repeated solutions are needed for additional right-hand sides.

(2) Both the recursive doubling and the cyclic reduction methods are most efficient when solving systems with $n = 2^q$ equations. If n is slightly larger than 2^q, then the scalar counts will be doubled for these two methods. In addition, the storage requirement will also be doubled. Under these conditions, the partition algorithm might be the least costly method.

(3) In Section 2 we have mentioned that in modern large-scale-vector computers the data rate is enhanced to match the computation speed of the arithmetic unit through the use of interleaved memory banks. If a vector operation calls for an operand vector whose elements are located h words apart in the memory, then the data rate might be reduced due to bank conflicts and thus result in a longer vector operation time. When using the cyclic reduction algorithm to solve a tridiagonal system of order n, there are $\log_2 n$ steps. The ith step requires vector operands that are 2^i words apart in storage. This causes a potential delay in the vector operations. On the other hand, when the partition method is used, we can always choose a value for p such that no bank conflicts will occur. For instance, choose a p that makes k odd with m-way interleaved memory banks with m even.

7. SUMMARY REMARKS

We have presented in this paper the partition method for solving a tridiagonal system of linear equations. The method is suitable for both parallel and vector computers. Although the partition method has a slightly higher vector operation count than those of the two competing methods, it has a scaler count much

smaller than that of the recursive doubling. The scalar counts between the partition method and the cyclic reduction method are so close as to make a timing evaluation inconclusive without considering the data management problem, especially when large systems are solved. We have described various situations in which the partition method can be superior.

Finally, we want to say a word about the relative stability of the algorithms. None of the algorithms considered here includes pivoting; therefore, for a general tridiagonal system, any method might fail. For a symmetric and diagonally dominant system, Dubois and Rodrigue [1] showed that the recursive doubling algorithm when applied to the forward and backward substitutions is numerically stable. They also gave conditions for stability of the recursive doubling algorithm as applied to the LU decomposition of a tridiagonal system. The cyclic reduction algorithm is stable if the system is diagonally dominant or if the system is symmetric and positive definite (Lambiotte and Voigt [5]). The partition method is also stable for diagonally dominant systems. An argument for this is included in the paper by Wilkinson [11, sec. 8]. In particular, for diagonally dominant systems with constant diagonals, the fill-ins (f's and g's) become progressively smaller, which leads to an abbreviated calculation of the algorithm.

It is possible to include partial pivoting in step (2) of the new algorithm. The result is the probable creation of additional fill-ins along the diagonal next to the superdiagonal. The operation can be performed entirely in parallel. Unfortunately, there is no convenient way to incorporate pivoting in the rest of the elimination. With this limited pivoting, the only thing we can predict is the gain in stability during the elimination of the subdiagonal in each subsystem.

ACKNOWLEDGMENT

The author is sincerely grateful to the referees whose comments and suggestions were most helpful.

REFERENCES

1. DUBOIS, P., AND RODRIGUE, G. An analysis of the recursive doubling algorithm. In *High Speed Computer and Algorithm Organization*, D. J Kuck, D. H Lawrie, and A. H Sameh, Eds., Academic Press, New York, 1977.
2. EKLUNDH, J. O. A fast computer method for matrix transposing *IEEE Trans. Comput. C-21* (1972), 801-803.
3. GEORGE, A An efficient band-oriented scheme for solving nbyn grid problems. In *Proc Fl Jt Computer Conf.*, AFIPS Press, Arlington, Va., 1972, pp. 1317-1320.
4. LAMBIOTTE, J. J. The solution of linear systems of equations on a vector computer. Ph.D. Thesis, Univ. of Virginia, Charlottesville, 1975.
5. LAMBIOTTE, JR., J. J., AND VOIGT, R. G. The solution of tridiagonal linear systems on the CDC STAR-100 computer. *ACM Trans. Math. Softw. 1*, 4 (Dec. 1975), 308-329.
6. LORIN, H. *Parallelism in Hardware and Software: Real and Apparent Concurrency.* Prentice-Hall, Englewood Cliffs, N.J., 1972.
7. SCHUMANN, U. Comments on "A fast computer method for matrix transposing" and application to the solution of Poisson's equation. *IEEE Trans. Comput. C-22* (1973), 542-543.
8. STONE, H. S. Parallel tridiagonal equation solvers. *ACM Trans. Math. Softw. 1*, 4 (Dec. 1975), 289-307
9. WACHSPRESS, E. L. *Interactive Solution of Elliptic Systems.* Prentice-Hall, Englewood Cliffs, N.J., 1966.

10. WANG, H. H. Transposing matrices on a vector computer. Rep. G320-3389, IBM Palo Alto Scientific Center, Palo Alto, Calif., 1979.
11. WILKINSON, J. H. Error analysis of direct methods of matrix inversion. *J. ACM 8*, 3 (July 1961), 281-330.

Received November 1979; revised August 1980; accepted October 1980.