SIAM J. SCI. STAT. COMPUT.
Vol. 11, No. 3, pp. 425-449, May 1990

002

ADAPTATION OF A TWO-POINT BOUNDARY VALUE PROBLEM SOLVER TO A VECTOR-MULTIPROCESSOR ENVIRONMENT*

S. J. WRIGHT† AND V. PEREYRA‡

Abstract. Systems of linear equations arising from finite-difference discretization of two-point boundary value problems have coefficient matrices that are sparse, with most or all of the nonzeros clustered in blocks near the main diagonal. Some efficiently vectorizable algorithms for factorizing these types of matrices and solving the corresponding linear systems are described. The relative effectiveness of the different algorithms varies according to the distribution of initial, final, and coupled end conditions. The techniques described can be extended to handle linear systems arising from other methods for two-point boundary value problems, such as multiple shooting and collocation. An application to seismic ray tracing is discussed.

Key words. two-point boundary value problems, finite-difference methods, vector processors, seismic ray tracing

AMS(MOS) subject classifications. 34B10, 65F05, 65L10, 86-08

1. Introduction. In this paper we describe new algorithms for solving a sparse linear system of equations that arises in global discretization schemes for solving two-point boundary value problems of the general form

$$\begin{aligned} &z'(t) = f(t, z), \qquad a \leqq t \leqq b, \\ &g(z(a), z(b)) = 0. \end{aligned} \tag{1.1}$$

Here $f, z, g \in R^m$, $t \in R$. These discretization schemes lead to a system of nonlinear equations whose solution is a discrete approximation to the true solution of (1.1). If Newton's method (or some variant) is used to solve this nonlinear system, it is known that for the "obvious" orderings of the equations and unknowns, the Jacobian will have most of its nonzero elements clustered about the main diagonal (see [2], [7], [8]–[10]). To find the Newton correction it is necessary to solve a linear system for which the Jacobian is the coefficient matrix. This operation is carried out repeatedly and is often the most time-consuming part of the solution process.

The particular linear systems that the algorithms of this paper are intended to solve are those arising from the PASVAR series of codes (see [9], [10] and appropriate sections of the IMSL, NAG and Harwell computer software libraries), but they also have other applications. The basis of the PASVAR algorithms is a trapezoidal-rule discretization of (1.1), with deferred corrections used to enhance the solution accuracy. Similar systems arise in algorithms based on the one-step formulae described in Cash [2]. Although we do not deal with linear systems arising from multistep, multiple shooting, and collocation methods here, we note that linear equation solvers analogous to those discussed in § 3 could also be devised for these methods.

* Received by the editors November 9, 1987; accepted for publication (in revised form) May 1, 1989. This research was performed under National Science Foundation Small Business Innovation Research award #8619607. Any opinions, findings, conclusions, and recommendations are those of the authors and do not necessarily reflect the views of the National Science Foundation.

† Mathematics Department, North Carolina State University, Raleigh, North Carolina 27695. The research of this author was partially supported by National Science Foundation grants ASC-8714009, DMS-8619903, and DMS-880033P and Air Force Office of Scientific Research grant AFOSR-ISSA-870092.

‡ Weidlinger Associates, 4410 El Camino Real, Suite 110, Los Angeles, California 94022.

For the trapezoidal-rule discretization of (1.1), a mesh $\{t_1, t_2, \cdots, t_n\}$ is chosen such that

$$a = t_1 < t_2 < \cdots < t_n = b,$$

and the differential equation is replaced by a system of algebraic equations

$$\text{(1.2a)} \quad s_{i-1}(z_{i-1}, z_i) = z_i - z_{i-1} - \frac{h_i}{2}[f(t_i, z_i) + f(t_{i+1}, z_{i+1})] = 0, \qquad i = 2, \cdots, n$$

where $h_i = t_i - t_{i-1}$ and $z_i \approx z(t_i)$. Often, the boundary conditions in (1.1) are well structured and can be separated into initial, coupled, and final conditions as follows:

$$\text{(1.2b)} \quad \begin{aligned} g_1(z(a)) &= 0 \qquad (g_1 \in R^p), \\ g_2(z(a), z(b)) &= 0 \qquad (g_2 \in R^r), \\ g_3(z(b)) &= 0 \qquad (g_3 \in R^q). \end{aligned}$$

Here $p + q + r = m$. To complete the algebraic equations for the discrete solution $z_1, \cdots, z_n$, we replace $z(a)$ by z_1 and $z(b)$ by z_n in (1.2b). A modified Newton's method is used to solve these equations.

Depending on the ordering of the equations in (1.2) and the unknowns $z_1, \cdots, z_n$, the Jacobian is usually structured in one of the following ways:

(1) Block tridiagonal, where the off-diagonal blocks have some zero rows, and the lower left block is partially filled if $r > 0$ (see Fig. 1). The ordering of equations here is

$$(g_1, s_1, s_2, \cdots, s_{n-1}, g_2, g_3),$$

and the variables are ordered as

$$(z_1, \cdots, z_n).$$

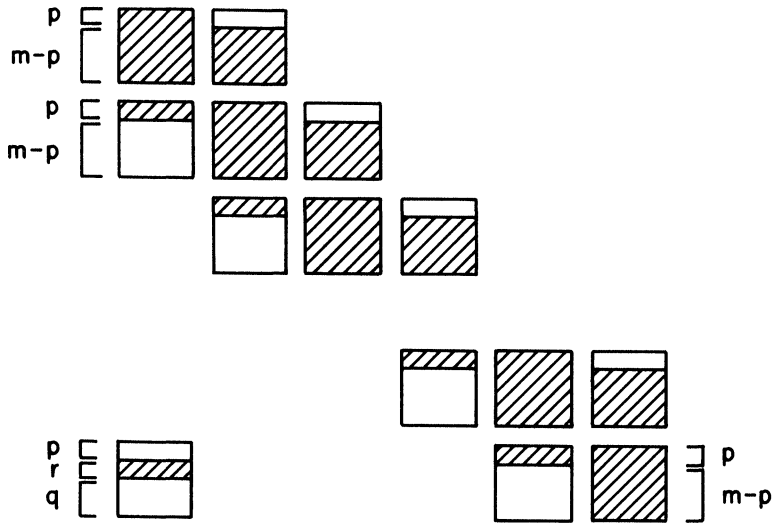

FIG. 1. *"Block-tridiagonal" structure used by method of § 2. Shaded areas indicate possible nonzeros.*

(2) Block upper-bidiagonal, with a partially filled block in the lower left corner (see Fig. 2). The ordering of the unknowns here is the same as in (1), but the equations are ordered as follows:

$$(s_1, \cdots, s_{n-1}, g_1, g_2, g_3).$$

The total dimension of the matrix is (mn). In this paper we are interested in the commonly occurring situation in which n is substantially larger than m, corresponding to a fine mesh with a system of differential equations of relatively small dimension.

Existing codes that handle the case of separated end conditions ($r = 0$) [3], [7], [10] use the ordering of Fig. 1. Partial pivoting, involving both row and column swaps at different stages, is used during the factorization. No fill-in occurs, and it is shown in [7] that this pivot strategy ensures a valid factorization is produced whenever the matrix is nonsingular. The algorithm of [10] (to be discussed in § 2) can also handle the case of coupled end conditions. Fill-in of approximately rn elements in the last row of blocks in the L factor takes place. Elements in these blocks are not considered as possible pivots until the final few steps of the elimination. This limitation on the pivoting means that there are some nonsingular matrices for which this algorithm fails to produce a valid factorization (see the example in the Appendix). However, the analysis of Keller [8] can be applied to show that, when the matrix is a Jacobian that arises from a one-step discretization scheme, the algorithm will work when the mesh is sufficiently fine. This is shown in the Appendix.

Some disadvantages become clear when this method is implemented in a vector-processing environment. Since it handles only a small number of blocks at a time (proceeding sequentially down the diagonal in Fig. 1), vectorizable loops in the code involve m or fewer elements.

The algorithms to be discussed here, which use the second ordering, start by factorizing the first $(n-1)$ diagonal blocks concurrently. This allows vectorizable loops of length $(n-1)$ to appear in the code. Similar loops arise in the factorization of the above-diagonal blocks. Since we are assuming $n > m$ here, there is clearly greater potential for speedup in the new algorithms. However, there is a trade-off: the new methods tend to produce more fill-in and have higher operation counts, and hence are unsuitable for use in a scalar environment. A blocked ordering similar to that of Fig. 2 was used originally in the first version of PASVAR [9], although of course it was not then solved by a vectorized method.

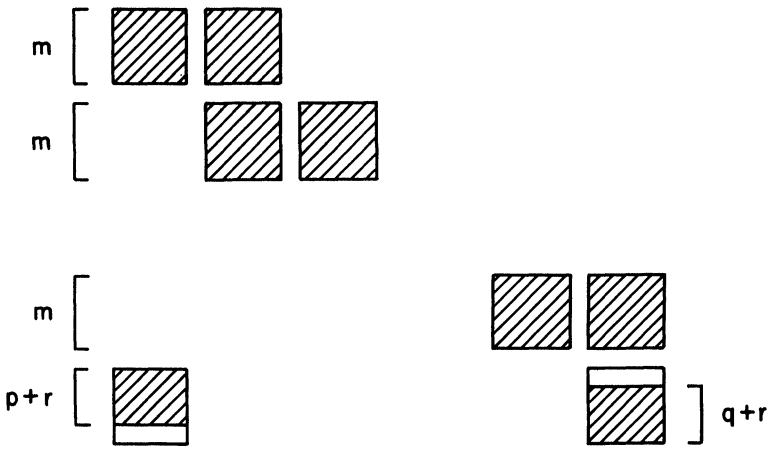

FIG. 2. *"Nearly block-bidiagonal" structure used by methods of* § 3.

Since these algorithms essentially perform a block factorization, with row pivoting allowed within blocks but not between blocks, they cannot factorize every nonsingular matrix of the form of Fig. 2, except in the case $p+r=0$. The analysis of the Appendix can be applied again, however, to show that they will succeed if the mesh is sufficiently fine. Numerical experience suggests that in low precision environments (e.g., 4-byte floating point wordlength), the new algorithms may not be as stable as the algorithm in [10]. This is not surprising, since the $p+r$ rows at the bottom of the matrix are always excluded from consideration as pivots, rather than just r rows in [10]. It may therefore be wise to retain the existing codes as a "backup" in case the new methods fail.

The new algorithms are discussed in § 3. In § 4, we present some timing comparisons between the new methods and the existing codes on the Alliant FX/8 vector multiprocessor, the CRAY X-MP, and the CRAY-2. Application of the resulting vectorized code to the problem of seismic ray tracing is discussed in the final section, and some results from the CRAY-2 are given.

In [5], Goldmann considers vectorization of the multiple shooting method. He notes that in many applications it is possible to vectorize the process of evaluating derivatives of f at different points. That is, the (j, k) elements of the $m \times m$ blocks $\partial f/\partial z(t_i, z_i)$ can be evaluated concurrently for $i = 1, \cdots, n$. Since this aspect of the overall method is problem-dependent, we do not discuss it further. Goldmann also considers a partially vectorizable method for solution of a linear system similar to that of Fig. 2, but it depends strongly on the fact that the above-diagonal blocks are multiples of the $m \times m$ identity matrix, and hence cannot be applied here.

2. The Routines DECOMP and SOLVE. The routine DECOMP in the PASVAR codes takes the Jacobian matrix A (as ordered in Fig. 1) and produces two factors L and U such that

$$LU = PAQ^T, \tag{2.1}$$

where P and Q are permutation matrices. The matrix U is only block upper triangular and so (2.1) is not a true LU factorization; however, linear systems involving L and U can be easily solved. The row and column interchanges (which are accounted for in P and Q, respectively), are chosen so that there is no fill-in near the main diagonal. If $r > 0$, then r dense rows will appear at the bottom of the L factor (see Fig. 3).

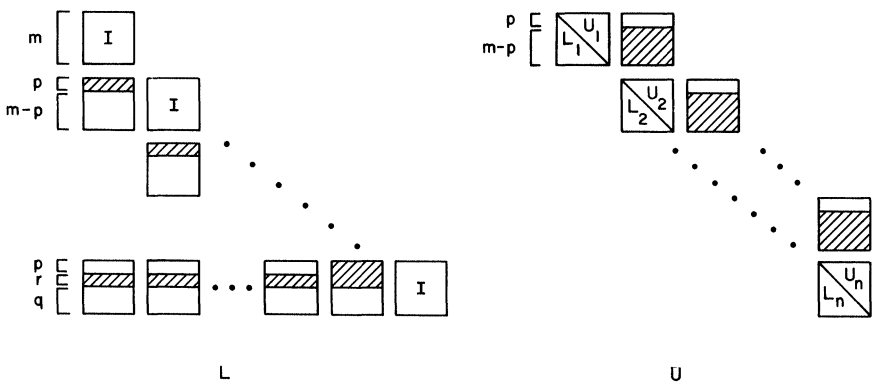

FIG. 3. *Structure of the factors produced by* DECOMP. (L_i *and* U_i *are the L and U factors of the* (i, i) *block.*)

The partial pivoting strategy needs some explanation. For the factorization of the first p rows of the matrix, column interchanges are used to ensure that the pivot element has the largest magnitude of any element in its row. Clearly, only columns 1 through m are candidates for the pivot column, since all other columns have zeros in rows 1 to p, and no fill-in occurs. For the factorization of the remaining rows in the $(1, 1)$ block, row interchanges involving rows $p+1$ through $m+p$ of the overall matrix are allowed. Next, the nonzero rows in the $(2, 1)$ and $(n, 1)$ blocks are eliminated, and a set of r dense rows is introduced into the $(n, 2)$ block as a result. In addition, the first p rows of the $(2, 2)$ block need to be updated. The elements in the $(1, 2)$ block are not altered, except perhaps for some row interchanges.

Factorization of the $(2, 2)$ through (n, n) blocks now proceeds similarly. At the $(n-1)$st stage, it is necessary to eliminate $(p+r)$ dense rows in the $(n, n-1)$ block: p rows from the original matrix, plus r rows introduced by the elimination at the previous stage. For the (n, n) block, only column pivoting is used. The overall strategy remains one of *partial* rather than complete pivoting, since at each step we choose a pivot element from *either* the row *or* the column corresponding to the current pivot position, rather than from some submatrix of the unfactored part of A. The choice between a row and a column search is made according to which will avoid fill-in, as we describe above. Further details on this type of pivoting strategy, and its extension to more general matrices, can be found in [3] and [14].

Ignoring low-order terms, the algorithm takes

$$n[\tfrac{1}{3}m^3 + \tfrac{5}{2}m^2 p - mp^2 + mr(2m-p)] \tag{2.2}$$

flops (where each flop consists of an addition and a multiplication). For $p = r = 0$ this reduces to $\frac{1}{3}m^3 n$; for $r = 0, p = m$ it is $(11/6)m^3 n$. The small operation count in the former case arises from the fact that there are no subdiagonal blocks to be eliminated. In the case of fully coupled boundary conditions $(p = 0, r = m)$ the flop count is $(7/3)m^3 n$, which reflects the work involved in "chasing" elements from the $(n, 1)$ block across the bottom of the matrix during factorization.

The SOLVE routine is relatively simple. Given a right-hand side y, the following equations are solved in succession:

$$Lv = Py \quad \text{for } v, \tag{2.3a}$$

$$Uw = v \quad \text{for } w, \tag{2.3b}$$

$$u = Qw \quad \text{for the final answer } u. \tag{2.3c}$$

The forward substitution (2.3a) requires about $nm(p+r)$ flops, while (2.3b) requires about $nm(2m-p)$, making a total of

$$nm(2m+r) \tag{2.4}$$

operations for the whole process. For the case $r = 0$ this reduces to $2nm^2$, and hence requires between $12/11m$ and $6/m$ of the execution time of DECOMP (depending on the value of p). For small values of m, therefore, the SOLVE phase is a significant part of the overall computation, particularly if p is also small. This effect becomes even more important when it is necessary to call SOLVE more than once for each call to DECOMP. This happens in PASVA4 for two reasons. First, a modified Newton's method is used to solve (1.2), in which the Jacobian is not necessarily updated at each iteration, obviating a refactorization. Second, PASVA4 allows the differential system (1.1) to be augmented by some algebraic equations and, correspondingly, some extra parameters. The linear system therefore has a "border" of extra rows and columns,

and a Schur-complement approach involving one extra call to SOLVE for each extra parameter, is used to solve it. This occurs in the application to be discussed in § 5, in which SOLVE is called up to 10 times more often than DECOMP.

3. The new algorithms. In this section we report on four new algorithms for the factorization of the Jacobian, all based on the equation and variable ordering of Fig. 2. Three new methods that use the resulting L and U factors are also needed for the solution phase. The common feature of the four factorization algorithms is that the LU factorizations of blocks $(1, 1)$ through $(n-1, n-1)$ are formed concurrently. Row partial pivoting is used within each block. The algorithms differ in how the fill-in elements in the last row of blocks are computed, and also in whether, and how, the superdiagonal blocks are altered by the factorization (as they are not, in the original DECOMP).

In all the algorithms, the most significant parts of the computation take place in loop constructs of one of the following forms:

(3.1) General triad:
$$\begin{aligned}&\underline{\text{for}}\ i = 1, 2, \cdots\\ &\quad \underline{\text{for}}\ j = 1\ \underline{\text{to}}\ t\\ &\quad\quad a_i(j) \leftarrow a_i(j) + b(j) * c_i(j)\end{aligned}$$

(3.2) Inner product:
$$\begin{aligned}&\underline{\text{for}}\ i = 1, 2, \cdots\\ &\quad \underline{\text{for}}\ j = 1\ \underline{\text{to}}\ t\\ &\quad\quad \alpha_i \leftarrow \alpha_i + a_i(j) * b(j)\end{aligned}$$

(3.3) Saxpy:
$$\begin{aligned}&\underline{\text{for}}\ i = 1, 2, \cdots\\ &\quad \underline{\text{for}}\ j = 1\ \underline{\text{to}}\ t\\ &\quad\quad a(j) \leftarrow a(j) + \beta_i * c_i(j)\end{aligned}$$

(3.4) Saxpy′:
$$\begin{aligned}&\underline{\text{for}}\ i = 1\ \underline{\text{to}}\ t\\ &\quad \underline{\text{for}}\ j = 1\ \underline{\text{to}}\ i\\ &\quad\quad a(j) \leftarrow a(j) + \beta_i * c_i(j)\end{aligned}$$

Here a_i, b, c_i are vectors whose jth components are $a_i(j)$, $b(j)$, $c_i(j)$.

On a vector processor, the time required for the inner loop in each of the above structures is comprised of

(a) The time to fetch the argument vectors from main (or cache) memory.

(b) The time required to initialize the pipeline.

(c) Processing time for each component of the result (typically, after the start-up time, one component of the result appears at the end of the pipe at each clock cycle).

(d) The time required to store the result.

On register-based machines (including the Alliant and all CRAY machines), this overhead may be incurred repeatedly as the vectors are processed in "strips" of 32 or 64 vector components. For the general triad (3.1), it is necessary to load two vectors for each new value of i (except for three vectors when $i = 1$), and store one result. For the inner product in (3.2), it is usually only necessary to load one vector and store a scalar result for each value of i. In (3.3) and (3.4), there is usually one vector load for each i, and no store until the end.

The operations (a)-(d) above are implemented quite differently on different architectures. For example, on the CRAY X-MP and Y-MP machines, two vectors can be loaded almost simultaneously, while on the CRAY 1 and CRAY 2, the second load cannot start until the first is complete. Preparation of the arithmetic processing pipe takes a different number of clock cycles on different machines. On the Alliant, the flop at each inner iteration of the loops in (3.1)-(3.4) is performed by a single "add-multiply" instruction; on the CRAYs, the add and multiply are done separately (although they are "chained" to give the effect of a single command).

We use T_t, R_t, S_t, and S'_t to denote the average time required for each iteration of the inner loop in (3.1), (3.2), (3.3), and (3.4), respectively. Each of these quantities could be expressed in terms of more fundamental parameters, such as clock-cycle time, main/cache memory fetch time, vector-register length, etc., but the resulting expressions would be nonlinear (and rather complex) functions of t. Although each of these quantities is generally monotonically decreasing with t, there may be sharp increases at certain values of t (e.g., multiples of the vector register length). The relation $S_t < S'_t$ should hold in general.

For the purpose of the algorithm descriptions below, we introduce the following notation for the various blocks:

$$A_i = \text{the } (i, i) \text{ block},$$

$$B_i = \text{the } (i, i+1) \text{ block},$$

$$D = \text{the } (n, 1) \text{ block}.$$

Using this notation, Fig. 2 can be rewritten as

$$A = \begin{pmatrix} A_1 & B_1 & & & \\ & A_2 & B_2 & & \\ & & \ddots & \ddots & \\ & & & & B_{n-1} \\ D & & \cdots & & A_n \end{pmatrix}. \tag{3.5}$$

The first two algorithms produce a true LU factorization, that is,

$$LU = PA, \tag{3.6a}$$

where the L and U factors and the permutation matrix have the form

$$L = \begin{pmatrix} L_1 & & & \\ & L_2 & & \\ & & \ddots & \\ E_1 & \cdots & E_{n-1} & L_n \end{pmatrix}, \qquad U = \begin{pmatrix} U_1 & V_1 & & & \\ & U_2 & V_2 & & \\ & & \ddots & \ddots & \\ & & & & V_{n-1} \\ & & & & U_n \end{pmatrix},$$

$$P = \begin{pmatrix} P_1 & & & \\ & P_2 & & \\ & & \ddots & \\ & & & P_n \end{pmatrix}. \tag{3.6b}$$

Here the L_i are unit lower triangular, U_i are upper triangular, and P_i are $m \times m$ permutation matrices. The matrices $E_1, \cdots, E_{n-1}$ each have $p+r$ dense rows, the positions of which depend on the permutation matrix P_n. Relating (3.5) and (3.6), we

find that equations for the submatrices are:

$$L_i U_i = P_i A_i, \qquad i = 1, \cdots, n-1, \tag{3.7}$$

$$L_i V_i = P_i B_i, \qquad i = 1, \cdots, n-1, \tag{3.8}$$

$$E_1 U_1 = P_n D, \tag{3.9a}$$

$$E_i V_i + E_{i+1} U_{i+1} = 0, \qquad i = 1, \cdots, n-2, \tag{3.9b}$$

$$E_{n-1} V_{n-1} + L_n U_n = P_n A_n. \tag{3.9c}$$

If the matrix A in (3.5) is partitioned as

$$A = \begin{pmatrix} & & & & \vdots & 0 \\ & C & & & \vdots & \vdots \\ & & & & \vdots & B_{n-1} \\ & & & & \vdots & \\ \cdots & \cdots & \cdots & \cdots & & \cdots \\ D & 0 & \cdots & 0 & \vdots & A_n \end{pmatrix}$$

it can be seen that the factorization (3.6) is obtained by finding an LU factorization of C with (unrestricted) partial pivoting, then computing

$$[D \quad 0 \cdots 0] C^{-1} = [E_1 \quad E_2 \quad \cdots \quad E_{n-1}],$$

and finally computing an LU factorization of the Schur complement of C in A, namely,

$$A_n - \{[D \quad 0 \cdots 0] C^{-1}\} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ B_{n-1} \end{pmatrix}.$$

All our algorithms are based on the calculation of

$$[D \quad 0 \cdots 0] C^{-1}$$

rather than

$$C^{-1} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ B_{n-1} \end{pmatrix}.$$

To obtain the latter, the solution of m linear systems with coefficient matrix C are required, while for the former, only $p+r$ linear systems with coefficient matrix C^T need to be solved.

The first algorithm follows directly from (3.7)–(3.9).

ALGORITHM D1.

(1) for $i = 1$ to $n-1$
    factor $L_i U_i = P_i A_i$
(2) for $i = 1$ to $n-1$
    solve $L_i V_i = P_i B_i$ for V_i
(3) solve $\bar{E}_1 U_1 = D$ for $\bar{E}_1$

(4) for $i=1$ to $n-2$
    solve $\bar{E}_i V_i + \bar{E}_{i+1} U_{i+1} = 0$ for $\bar{E}_{i+1}$
(5) factor $L_n U_n = P_n(A_n - \bar{E}_{n-1} V_{n-1})$
(6) set $E_i = P_n \bar{E}_i$, $i = 1, \cdots, n-1$.

The computationally significant steps in this algorithm are steps (1), (2), and (4). Step (1) requires about $\frac{1}{3}nm^3$ flops, step (2), about $\frac{1}{2}nm^3$, and step (4) about $\frac{3}{2}nm^2(p+r)$ flops. However, the factorizations in step (1) and backsolves in step (2) can be carried out concurrently, that is, they can be coded in the form of (3.1), where the innermost loop has $n-1$ iterations. The iterations in step (4), on the other hand, must be carried out in sequence. The only possibilities for vectorization in this step involve vectors of length m or less. The formation of $\bar{E}_i V_i$ can be coded in the form (3.3). The back substitution for $\bar{E}_{i+1}$ can be done in two ways: it can use either the structure (3.3), with $t = p+r$, or the structure (3.4), with $t = m$. Which of these is faster obviously depends on the value of $p+r$. Since the "average" length of the loop in (3.4) is $t/2$ (i.e., $m/2$ in our case), a simple switch in the code causes it to use (3.3) when $p+r > m/2$, and (3.4) otherwise. The approximate time required for Algorithm D1 will thus be

$$\frac{5}{6}nm^3 T_{n-1} + nm^2(p+r)S_m + \frac{1}{2}nm^2(p+r)\min(S'_m, S_{p+r}). \tag{3.10}$$

We note in passing that in the decoupled case ($r=0$), we can make the assumption that $p \leqq m/2$. If this is not true, we can simply reverse the order of the unknowns and rearrange the equations appropriately so that p and q are interchanged. This corresponds to reversing the sign of the independent variable t in (1.1).

The second algorithm is a variant of Algorithm D1, in which step (4) is partially vectorized at the expense of an increased operation count. From (3.7)–(3.9) and step (6) of Algorithm D1 we have by a simple inductive argument that

$$\bar{E}_i V_i = (-1)^{i+1} D \prod_{j=1}^{i} W_j, \quad \text{where } W_j = U_j^{-1} V_j. \tag{3.11}$$

Given that $(\bar{E}_i V_i)$ is known for $i = 1, \cdots, n-2$, it is possible to use forward substitution to calculate the matrices $\bar{E}_{i+1}$ concurrently. The algorithm can be summarized as follows.

ALGORITHM D2.
(1) for $i=1$ to $n-1$
    factor $L_i U_i = P_i A_i$
(2) for $i=1$ to $n-1$
    solve $L_i V_i = P_i B_i$ for V_i
(3) for $i=1$ to $n-1$
    solve $U_i W_i = V_i$ for W_i
(4) set $\bar{Z}_1 = D$
    for $i=1$ to $n-1$
        set $\bar{Z}_{i+1} = -\bar{Z}_i W_i$
(5) for $i=1$ to $n-1$
    solve $\bar{E}_i U_i = \bar{Z}_i$ for $\bar{E}_i$
(6) factor $L_n U_n = P_n(A_n + \bar{Z}_n)$
(7) set $E_i = P_n \bar{E}_i$, $i = 1, \cdots, n-1$.

The new steps in Algorithm D2 are (3), (4), and (5). Steps (3) and (5) are "vectorizable" (i.e., contain operations involving vectors of length at least $n-1$), with flop counts of about $\frac{1}{2}nm^3$ and $\frac{1}{2}nm^2(p+r)$, respectively. Step (4) is "sequential," but noting that each Z_i has only $(p+r)$ dense rows, we count about $nm^2(p+r)$ flops. The

approximate execution time is thus

$$[\tfrac{4}{3}nm^3+\tfrac{1}{2}nm^2(p+r)]T_{n-1}+nm^2(p+r)S_m, \qquad p+r>0, \tag{3.12a}$$

$$\tfrac{5}{6}nm^3T_{n-1}, \qquad p+r=0. \tag{3.12b}$$

(In the case (3.12b) steps (3), (4), and (5) are not necessary.) There is a trade-off in Algorithm D2: the total operation count is increased, but the number of "sequential" operations is decreased.

The solution phase corresponding to Algorithms D1 and D2 is the same, and referring to (3.6), it can be stated as follows.

ALGORITHM S1. Given a right-hand side $y^T=[y_1^T,\cdots,y_n^T]$:
(1) for $i=1$ to $n-1$
  solve $L_iv_i=P_iy_i$
(2) replace y_n by $(P_ny_n-\sum_{i=1}^{n-1}E_iv_i)$
(3) solve $L_nv_n=y_n$
  solve $U_nu_n=v_n$
(4) for $i=n-1$ to 1
  replace v_i by $(v_i-V_iu_{i+1})$
  solve $U_iu_i=v_i$.

Here, steps (1) and (2) are vectorizable: step (1) requires about $\frac{1}{2}nm^2$ flops, while step (2) requires about $nm(p+r)$ operations. In fact, step (2) consists of inner products involving vectors of length $(n-1)m$ (if the matrices E_i are stored appropriately). The other computationally significant step is step (4), which is sequential in nature and requires about $\frac{3}{2}nm^2$ operations. The time required for Algorithm S1 is thus

$$nm(p+r)R_{(n-1)m}+\tfrac{1}{2}nm^2T_{n-1}+nm^2S_m+\tfrac{1}{2}nm^2S'_m. \tag{3.13}$$

Again we note that a comparison of (3.13) with (3.10) and (3.12), shows that the solution phase is not insignificant in relation to the factorization phase unless m is relatively large.

In the third algorithm, a block-LU factorization is performed. Matrices $\tilde{L}$, $\tilde{U}$, and P are obtained such that $\tilde{L}\tilde{U}=PA$, where

$$\tilde{L}=\begin{pmatrix} I & & & \\ & & I & \\ D_1 & \cdots & D_{n-1} & I \end{pmatrix}, \qquad \tilde{U}=\begin{pmatrix} \tilde{A}_1 & \tilde{B}_1 & & & \\ & \tilde{A}_2 & \tilde{B}_2 & & \\ & & & \tilde{A}_{n-1} & \tilde{B}_{n-1} \\ & & & & \tilde{A}_n \end{pmatrix} \tag{3.14}$$

and P is as in (3.6b). In (3.14), the matrices $\tilde{A}_1,\cdots,A_n$ are overwritten in storage by their L and U factors. Comparing (3.14) and (3.5), we obtain the formulae

$$\begin{aligned} &\tilde{A}_i=P_iA_i, && i=1,\cdots,n-1,\\ &\tilde{B}_i=P_iB_i, && i=1,\cdots,n-1,\\ &D_1\tilde{A}_1=P_nD,\\ &D_i\tilde{B}_i+D_{i+1}\tilde{A}_{i+1}=0, && i=1,\cdots,n-2,\\ &D_{n-1}\tilde{B}_{n-1}+\tilde{A}_n=P_nA_n. \end{aligned}$$

These suggest the following algorithm.

ALGORITHM D3.
(1) for $i=1$ to $n-1$
    factor $L_iU_i = P_iA_i = \tilde{A}_i$
(2) solve $\bar{D}_1\tilde{A}_1 = D$
(3) for $i=1$ to $n-2$
    solve $\bar{D}_i\tilde{B}_i + \bar{D}_{i+1}\tilde{A}_{i+1} = 0$ for $\bar{D}_{i+1}$
(4) factor $L_nU_n = P_n(A_n - \bar{D}_{n-1}\tilde{B}_{n-1})$
(5) set $D_i = P_n\bar{D}_i$, $i=1,\cdots,n-1$.

The only step that can be efficiently vectorized here is step (1), which requires about $\frac{1}{3}nm^3$ operations. Step (3) is strictly sequential, requiring a matrix multiplication $\bar{D}_i\tilde{B}_{i+1}$ and two triangular solves using the L and U factors of A_{i+1}. (The L and U factors of each diagonal block are the same as those computed in Algorithm D1, and Algorithm D2, provided the same pivot strategy is used within each block.) The workload for step (3) is $2nm^2(p+r)$ operations, and the approximate total time for Algorithm D3 will thus be

$$\frac{1}{3}nm^3T_{n-1} + nm^2(p+r)S_m + nm^2(p+r)\min(S'_m, S_{p+r}). \tag{3.15}$$

The solution phase of course differs from Algorithm S1, and can be stated as follows.

ALGORITHM S3. Given a right-hand side $y^T = [y_1^T, \cdots, y_n^T]$:
(1) for $i=1$ to n
    replace y_i by P_iy_i
(2) replace y_n by $(P_ny_n - \sum_{i=1}^{n-1} D_iy_i)$
(3) solve $L_nU_nu_n = y_n$ for u_n
(4) for $i=1$ to $n-1$
    solve $L_iU_iu_i = y_i - \tilde{B}_iu_{i+1}$ for u_i.

Only the second step (inner products with vectors of length $(m(n-1))$) can be efficiently vectorized here. The approximate time required is therefore

$$nm(p+r)R_{(n-1)m} + nm^2S_m + nm^2S'_m. \tag{3.16}$$

A fourth combination of routines, suggested by a referee, is also based on a block factorization. Here we find matrices $\hat{L}$, $\hat{U}$, and P such that $\hat{L}\hat{U} = PA$, where P is as in (3.6) and $\hat{L}$ and $\hat{U}$ have the form

$$\hat{L} = \begin{pmatrix} \tilde{A}_1 & & & & \\ & \tilde{A}_2 & & & \\ & & \ddots & & \\ & & & \tilde{A}_{n-1} & \\ Z_1 & \cdots & & Z_{n-1} & \tilde{A}_n \end{pmatrix}, \qquad \hat{U} = \begin{pmatrix} I & W_1 & & & \\ & I & W_2 & & \\ & & \ddots & \ddots & \\ & & & & W_{n-1} \\ & & & & I \end{pmatrix}.$$

Again, a comparison with (3.5) yields the formulae

$$\begin{aligned} &\tilde{A}_i = P_iA_i, \qquad i=1,\cdots,n-1, \\ &\tilde{A}_iW_i = P_iB_i, \qquad i=1,\cdots,n-1, \\ &Z_1 = P_nD, \\ &Z_{i+1} = -Z_iW_i, \qquad i=1,\cdots,n-2, \\ &Z_{n-1}W_{n-1} + \tilde{A}_n = P_nA_n. \end{aligned}$$

Note that the W_i and $\tilde{A}_i$ in the formulae above are the same as those in Algorithm D2, while the Z_i above are identical to $P_n\bar{Z}_i$ from Algorithm D2. The resulting algorithm is likewise very similar to Algorithm D2, the main different being omission of step (5):

ALGORITHM D4.
(1) for $i=1$ to $n-1$
    factor $L_iU_i = P_iA_i$
(2) for $i=1$ to $n-1$
    solve $L_iV_i = P_iB_i$ for V_i
(3) for $i=1$ to $n-1$
    solve $U_iW_i = V_i$ for W_i
(4) set $\bar{Z}_1 = D$
    for $i=1$ to $n-1$
    set $\bar{Z}_{i+1} = -\bar{Z}_iW_i$
(5) factor $L_nU_n = P_n(A_n + \bar{Z}_n)$
(6) set $Z_i = P_n\bar{Z}_i$, $i=1,\cdots,n-1$.

The approximate run time is

$$\tfrac{4}{3}nm^3T_{n-1} + nm^2(p+r)S_m. \tag{3.17}$$

Only steps (4) and (6) can be omitted when $p+r=0$. The corresponding solution phase follows.

ALGORITHM S4.
(1) for $i=1$ to $n-1$
    solve $L_iv_i = P_iy_i$
(2) for $i=1$ to $n-1$
    solve $U_iw_i = v_i$
(3) replace y_n by $(P_ny_n - \sum_{i=1}^{n-1} Z_iw_i)$
(4) solve $L_nU_nu_n = y_n$
(5) for $i=n-1$ to 1
    set $u_i = w_i - W_iu_{i+1}$.

Here steps (1), (2), and (3) are all vectorizable. The approximate execution time is thus

$$nm(p+r)R_{(n-1)m} + nm^2T_{n-1} + nm^2S_m. \tag{3.18}$$

When $T_{n-1} < S_m \leqq S_{m'}$, we see that none of the Algorithms D1–D4 is superior to any one of the others in all situations. However when $p+r=0$, Algorithm D3 is clearly fastest, followed by Algorithms D1 and D2 (which are identical in this case) and finally Algorithm D4. For $p+r>0$, it is clear by comparing (3.12a) and (3.17) that Algorithm D4 will be faster than Algorithm D2, while for $p+r=m$, Algorithm D4 will be the fastest, and Algorithm D1 will be faster than Algorithm D3. All other relationships between run times of the factorization algorithms will depend on the relative values of p, r, m and the execution times S'_m, S_{p+r}, and T_{n-1}.

For the solve algorithms, making the same assumption that $T_{n-1} < S_m \leqq S_{m'}$ it is clear that Algorithm S4 is always fastest, followed by Algorithm S1 and then Algorithm S3.

4. Computational comparisons. Here we report on computational experience with the algorithms of the two preceding sections, on three computers with vectorization capabilities. Matrices of the form (3.5) were generated in order to test Algorithms D1–D4, and these same matrices were rearranged into the form of Fig. 1 to test the

original DECOMP. In accordance with the purpose of the exercise, the form of these matrices corresponds roughly to what would be obtained from a finite-difference discretization of (1.1), that is,

$$A_i = I + h\bar{A}_i, \quad B_i = -I + h\bar{B}_i, \quad i = 1, \cdots, n-1$$

where the elements of $\bar{A}_i$ and $\bar{B}_i$ are uniformly distributed in $[-1, 1]$, and $h = .1$. The nonzero elements of A_n and D are uniformly distributed in $[-h, h]$.

The new factorization algorithms contain various checks for ill-conditioning. The largest and smallest elements of the U_i blocks are monitored throughout the factorization. In Algorithms D2 and D4, a comparison of the Frobenius norm of the matrices A_n and Z_n is made, to detect possible blowup as a result of the matrix multiplications in step (4).

The three computers used were an Alliant FX/8 vector multiprocessor (at the Advanced Computing Research Facility, Argonne National Laboratory), a CRAY X-MP/48 (at the Pittsburgh Supercomputer Center), and a CRAY-2 (at Cray Research, Mendota Heights, Minnesota).

The Alliant is a machine with eight processors (known as computational elements or CEs), which share a main memory and a 512 Kbyte fast-access cache memory. Each computational element can perform operations on vector registers that hold up to 32 double-precision (8-byte) words. The peak performance of each CE in single precision is 11.7 Mflops. When a data element is referenced by a program, it is automatically read into the cache, so that subsequent accesses to that element take substantially less time. However if the data references in the program are not localized, the element may be overwritten in cache, and will thus have to be fetched from main memory again when next referenced. This occurrence is referred to as a "cache fault." Careful management of the cache memory can lead to sharply reduced runtimes (see, for example, [4]).

Only one processor of each CRAY is used. The Pittsburgh CRAY X-MP has a clock-cycle time of 8.5 ns, and a common memory of 8 Mwords, arranged in 32 banks. To load a word of data from main memory, 14 clock cycles are required; to store a word takes somewhat less time. A set of 64-word vector registers is available, together with a corresponding set of vector instructions. To illustrate the processing capabilities of the X-MP, consider its performance in computing the general triad $d(j) = a(j) + b(j)c(j)$, $j = 1, \cdots, t$ (the same as (3.1), except that three vector loads and one store are required). For $t = 10$ the throughput is 25.2 Mflops, for $t = 100$ it is 73.7 Mflops, and for $t = 1000$, 91.1 Mflops (see Schönauer [13]). By comparison, the peak rates for multiplication and addition in scalar mode are 15 Mflops and 17.5 Mflops, respectively. On the CRAY-2, the cycle time is 4.1 ns, and the main memory consists of 256 Mwords organized into 128 banks. To retrieve a word from main memory, 57 clock cycles are required. However, data can be retrieved from each of the banks simultaneously, and so components of a long vector are typically available for processing at the rate of one per clock cycle after this initial delay (although problems of "section conflict" can arise; see [13]). The long fetch time is also partially offset by the presence of 16 Kwords of local (cache) memory for each processor. Access time for this storage is much shorter—four clock cycles. A set of eight 64-word vector registers can be used by each processor.

Although the expressions for approximate execution times were derived in the previous section with vectorization in mind, the algorithms were also run on the Alliant in "scalar" and "parallel-vector" mode. In scalar mode with global optimization (i.e., the compiler optimizations `-Og -DAS` were used), the program runs on a single CE

without using any vectorization capabilities. In parallel-vector mode (i.e., using the compiler options -O -DAS -alt), the compiler attempts to vectorize the innermost loop of any nested loop structure, and to schedule different iterations of some outer loop to execute concurrently on different CEs. This arrangement may be disturbed if later iterations of a loop depend on the results of earlier iterations (i.e., data dependencies), or if user-inserted compiler directives are present in the code. For details see [1]. In some of the vectorizable steps of the algorithms in § 3 (e.g., step (2) of Algorithm D1, steps (2) and (3) of Algorithm D2) it is possible to arrange for the "parallelized" loop to have length m, so that the step can take full advantage of the parallelism described above.

Results are given for three problem sets. In the first, the values $m=7$, $r=0$, $n=20$ are used; for the second, $m=7$, $r=0$, $n=100$. (These choices were motivated by the application to be discussed in § 5.) For the final problem set we use $m=32$, $r=0$, $n=100$. In all cases, the values of p and q were varied between 0 and m and timings are given for some of these values. Times were measured using the utility functions second (on the CRAYs) and etime on the Alliant. Readings taken at different times showed that they appeared to be relatively independent of the system load, and they are almost certainly accurate to within 10 percent.

The codes for all machines were identical, except for the use of different timing routines, and the insertion of different compiler directives. Single precision (24 bits of accuracy on the Alliant, 53 bits of accuracy on the CRAYs) was used throughout. On the CRAY machines, the cft77 compiler was used, with options -esaq -dp for the vectorized versions, and -esaq -dp -o novector for the nonvectorized versions.

Tables 1-6 give timings obtained for the first data set. For the vectorized versions of the codes on the Alliant (Tables 1 and 2), the results correspond closely to what we might expect from the timing expressions of the previous section. When $p=0$ and three or fewer solves are needed for each factorization, D3/S3 is the preferred

TABLE 1

Alliant. Scalar/vectorized/vector-concurrent times (in ms*) for decomposition routines,* $m=7$, $n=20$.

p	DECOMP	D1	D2	D3	D4
0	28./24./15.	34./22./14.	34./22./13.	20./16./11	52./28./14.
1	47./41./27.	40./25./22.	72./33./18.	30./24./24.	54./28./17.
4	80./75./47.	70./50./24.	97./43./18.	68./58./26.	73./39./18.
7	89./85./51.	97./62./26.	122./55./20.	99./68./29.	91./50./20.

TABLE 2

Alliant. Scalar/vectorized/vector-concurrent times (in ms*) for solution routines,* $m=7$, $n=20$.

p	SOLVE	S1	S3	S4
0	10./7.7/5.6	11./6.9/5.2	9.9/8.2/6.7	12./5.4/3.4
1	10./7.8/6.2	11./7.2/5.5	10./8.3/6.7	12./5.6/3.5
4	9.9/7.9/6.3	12./7.4/5.5	12./8.6/6.9	13./5.9/3.6
7	10./7.8/5.7	13./7.7/5.6	13./8.9/7.1	15./6.1/3.6

TABLE 3
CRAY X-MP. *Scalar/vectorized times (in* ms) *for decomposition routines,* $m = 7$, $n = 20$.

p	DECOMP	D1	D2	D3	D4
0	2.4/1.2	3.1/0.9	3.0/1.0	1.7/0.8	5.0/1.3
1	4.1/3.2	4.1/1.8	6.3/1.8	3.1/2.1	5.3/1.6
4	7.0/8.1	6.5/2.9	8.7/2.7	6.2/3.3	6.8/2.6
7	8.3/9.6	8.8/4.0	11./3.6	8.9/4.5	8.4/3.6

TABLE 4
CRAY X-MP. *Scalar/vectorized times (in* ms) *for solution routines,* $m = 7$, $n = 20$.

p	SOLVE	S1	S3	S4
0	1.0/1.5	1.0/0.5	1.1/0.6	0.9/0.3
1	1.0/1.5	1.1/0.5	1.2/0.6	0.9/0.3
4	1.0/1.5	1.1/0.5	1.2/0.6	1.0/0.3
7	1.0/1.5	1.2/0.5	1.3/0.6	1.1/0.3

TABLE 5
CRAY-2. *Scalar/vectorized times (in* ms) *for decomposition routines,* $m = 7$, $n = 20$.

p	DECOMP	D1	D2	D3	D4
0	2.8/2.0	2.8/1.1	2.8/1.3	1.6/0.9	4.1/1.5
1	5.2/4.3	3.9/2.1	6.0/2.3	3.1/2.5	5.0/1.9
4	9.1/9.2	6.3/3.6	8.4/3.5	6.2/4.0	6.7/3.1
7	10./11.	8.6/4.8	11./4.7	9.1/5.3	8.5/4.2

TABLE 6
CRAY-2. *Scalar/vectorized times (in* ms) *for solution routines,* $m = 7$, $n = 20$.

p	SOLVE	S1	S3	S4
0	1.3/1.4	1.3/0.7	1.5/0.9	1.1/0.4
1	1.3/1.4	1.3/0.7	1.5/0.9	1.1/0.5
4	1.3/1.4	1.4/0.8	1.6/0.9	1.2/0.5
7	1.3/1.4	1.5/0.8	1.6/0.9	1.3/0.5

combination. In almost all other cases, D4/S4 will be fastest. The combination D4/S4 is always fastest in parallel-vector mode. Note that the speedups (defined as the ratio of runtime for the vectorized implementation to runtime for the parallel-vector implementation) are not impressive—nowhere greater than three on this 8-processor machine. This is due to the fine-grained nature of the computation for this value of n, and the fact that in step (1) of each factorization algorithm, the number of vector operations that can be scheduled to run in parallel decreases from six to one during the computation. The original combination of DECOMP and SOLVE only remains competitive in scalar mode.

On the CRAY X-MP (Tables 3 and 4), the improvement due to vectorization is seen to best advantage in Algorithms D2 and D4. In some cases, the vectorized implementation of DECOMP and SOLVE is actually slower than the scalar implementation, because the overhead associated with the vector instructions is not worthwhile for vectors with seven elements. Among scalar implementations, there is little to choose between the various possible combinations. For the vectorized implementations, the combination D4/S4 is superior, except possibly when $p = 0$.

Times for the scalar CRAY-2 codes (Tables 5 and 6), are similar to those for the scalar CRAY X-MP codes, but the speedups due to vectorization are a little smaller. Possibly this is because, on the CRAY-2, the longer memory cycle time means that there is less difference between $R_{m(n-1)}$, T_{n-1}, and S_m. For scalar implementations, there is little to choose between D1/S2, D4/S4 and D3/S3 except for small p, when the latter is best. For the vectorized codes, D4/S4 is superior.

Results for the second data set appear in Tables 7–12. Most of the comments for the first data set apply again here, although for some algorithms (particularly the most highly vectorized codes D2, D4, and S4) the improvement due to vectorization is more pronounced. That is, T_{99} is substantially smaller than T_{19}. The combination D4/S4 is now best in almost all vectorized and parallel-vector implementations. Among scalar implementations there is not much difference between the various possibilities. On the

TABLE 7

Alliant. Scalar/vectorized/vector-concurrent times (in s) for decomposition routines, $m = 7$, $n = 100$.

p	DECOMP	D1	D2	D3	D4
0	.14/.11/.065	.14/.063/.052	.14/.064/.051	.072/.048/.071	.23/.085/.055
1	.24/.20/.13	.19/.11/.098	.34/.12/.072	.15/.11/.11	.25/.10/.073
4	.41/.39/.24	.35/.23/.11	.47/.17/.073	.34/.28/.12	.35/.16/.077
7	.46/.43/.26	.48/.29/.11	.59/.22/.076	.49/.33/.12	.44/.21/.080

TABLE 8

Alliant. Scalar/vectorized/vector-concurrent times (in ms*) for solution routines, $m = 7$, $n = 100$.*

p	SOLVE	S1	S3	S4
0	50/38/28	53/33/25	51/41/33	59/23/14
1	51/39/31	55/34/26	53/41/33	61/23/14
4	50/39/31	61/35/26	59/43/33	67/24/14
7	50/39/28	67/36/26	64/44/34	73/25/14

TABLE 9
CRAY X-MP. *Scalar/vectorized times (in* ms) *for decomposition routines,* $m = 7$, $n = 100$.

p	DECOMP	D1	D2	D3	D4
0	14./6.0	15./3.7	15./4.1	8.1/3.3	23./4.9
1	21./17.	21./8.3	30./6.9	16./11.	26./6.5
4	36./43.	33./14.	42./12.	31./16.	34./11.
7	43./50.	45./19.	54./16.	46./22.	42./16.

TABLE 10
CRAY X-MP. *Scalar/vectorized times (in* ms) *for solution routines,* $m = 7$, $n = 100$.

p	SOLVE	S1	S3	S4
0	5.0/7.5	5.3/2.4	5.8/2.8	4.4/1.1
1	5.1/7.6	5.3/2.4	5.9/2.8	4.5/1.1
4	5.1/7.6	5.6/2.5	6.2/2.9	4.9/1.1
7	5.2/7.6	5.9/2.5	6.4/2.9	5.2/1.2

TABLE 11
CRAY-2. *Scalar/vectorized times (in* ms) *for decomposition routines,* $m = 7$, $n = 100$.

p	DECOMP	D1	D2	D3	D4
0	14./10.	13./3.8	13./4.5	7.3/3.6	21./5.3
1	27./22.	19./9.0	28./8.3	15./12.	24./6.8
4	47./48.	32./16.	40./14.	31./20.	33./13.
7	53./56.	43./22.	52./20.	46./26.	42./19.

TABLE 12
CRAY-2. *Scalar/vectorized times (in* ms) *for solution routines,* $m = 7$, $n = 100$.

p	SOLVE	S1	S3	S4
0	6.6/7.2	6.4/3.4	7.4/4.4	5.1/1.9
1	6.7/7.2	6.6/3.5	7.5/4.4	5.3/1.9
4	6.8/7.3	6.8/3.5	7.8/4.5	5.6/1.9
7	6.7/7.1	7.1/3.6	8.1/4.6	5.9/2.0

Alliant and X-MP, DECOMP/SOLVE is usually best, except when p is small, in which case D3/S3 is competitive.

The results for the third data set (Tables 13–18) have a quite different character. The value of m is now large enough that almost all operations in all the algorithms vectorize efficiently. T_{n-1}, $R_{m(n-1)}$, S_m, and S'_m are now much closer together than for previous data sets, and so the algorithms with the lower operation counts (most notably D1) are more competitive. Overall speedups due to vectorization are quite impressive. Among scalar implementations, the original combination DECOMP/SOLVE is clearly best on all machines. Among vectorized implementations on the CRAY machines, there is little difference between the new algorithms, either for factorization or solve, although the combination D3/S3 has an advantage when $p = 0$.

The results on the Alliant (Tables 13 and 14) are distorted by the hierarchical memory structure of that machine. When $m = 32$ and $n = 100$, the data structures that store the A_i's, the B_i's, and the E_i's occupy about 400 Kbytes each—substantially more than the 256 Kbyte cache can handle. Since data references in the new algorithms tend not to be localized, it is likely that many items of data will cause repeated cache faults during execution of the program. Algorithm D2 (Table 13) is most seriously affected by this phenomenon. For example, the data structure that contains the A_i's (and, after step (1), the L_i and U_i factors), is continually being overwritten in cache, and hence needs to be fetched repeatedly. This occurs at the start of step (3), where the data structure containing the W_i's is initialized to the V_i's. (This is not a problem in Algorithm D4, since the V_i's are not needed in the subsequent solution phase and hence can be overwritten by the W_i's.) It occurs again during step (4), and so the U_i blocks need to be fetched again for step (5). Finally, note that the value $m = 32$ allows greater scope for parallelism, and so the speedups in going from vector mode to parallel-vector mode are better for this data set.

We conclude that the new algorithms, in particular Algorithms D4 and S4, can under most circumstances execute significantly faster than DECOMP and SOLVE if implemented in an appropriate way on machines with vectorization capabilities. The

TABLE 13

Alliant. Scalar/vectorized/vector-concurrent times (in s) for decomposition routines, $m = 32$, $n = 100$.

p	DECOMP	D1	D2	D3	D4
0	5.0/2.2/.60	12./3.4/.98	12./3.6/1.0	4.8/1.7/.62	19./5.5/1.3
2	7.5/3.6/1.3	24./3.9/1.3	27./17./3.2	6.8/2.4/1.0	20./5.8/1.4
16	18./8.2/3.8	24./7.3/1.7	37./24./3.8	20./6.7/1.6	27./7.5/1.7
32	23./12./6.0	36./10./2.2	49./32./4.9	34./10./2.2	34./9.6/2.0

TABLE 14

Alliant. Scalar/vectorized/vector-concurrent times (in s) for solution routines, $m = 32$, $n = 100$.

p	SOLVE	S1	S3	S4
0	.67/.41/.15	.84/.28/.20	.72/.37/.25	1.1/.28/.15
2	.67/.44/.16	.86/.29/.21	.75/.38/.25	1.1/.29/.15
16	.67/.44/.16	.99/.32/.21	.88/.41/.26	1.2/.32/.16
32	.67/.41/.16	1.1/.36/.22	1.0/.44/.27	1.4/.36/.16

TABLE 15
CRAY X-MP. *Scalar/vectorized times (in* s) *for decomposition routines,* $m = 32$, $n = 100$.

p	DECOMP	D1	D2	D3	D4
0	.59/.089	1.2/.12	1.2/.12	.49/.073	2.0/.18
2	.77/.18	1.4/.15	2.2/.19	.70/.17	2.1/.19
16	1.6/.76	2.3/.28	3.3/.30	1.9/.31	2.7/.29
32	2.1/1.1	3.4/.41	4.6/.43	3.3/.47	3.3/.40

TABLE 16
CRAY X-MP. *Scalar/vectorized times (in* ms) *for solution routines,* $m = 32$, $n = 100$.

p	SOLVE	S1	S3	S4
0	55/36	82/13	79/12	80/11
2	55/35	80/13	79/12	81/12
16	55/36	86/14	86/13	87/12
32	55/36	93/15	93/14	94/13

TABLE 17
CRAY-2. *Scalar/vectorized times (in* s) *for decomposition routines,* $m = 32$, $n = 100$.

p	DECOMP	D1	D2	D3	D4
0	.54/.15	1.1/.14	1.2/.17	.47/.081	1.8/.21
2	.82/.27	1.2/.19	2.1/.27	.67/.21	1.9/.24
16	2.0/.88	2.2/.38	3.1/.43	1.9/.40	2.5/.38
32	2.7/1.2	3.2/.55	4.3/.59	3.2/.59	3.2/.52

TABLE 18
CRAY-2. *Scalar/vectorized times (in* ms) *for solution.*

p	SOLVE	S1	S3	S4
0	83/39	101/24	99/20	91/27
2	83/39	102/25	100/21	92/27
16	83/38	108/25	106/21	97/28
32	83/38	115/27	113/22	104/29

relative efficiency of the various codes varies with the values of m, n, the distribution of the initial, coupled, and final boundary conditions, and the characteristics of the computers on which they are implemented. However, as seen above, the timing expressions derived in § 3 provide a useful way of predicting the performance.

5. Application to seismic ray tracing. The two-point boundary value problem code PASVA4 has been utilized as part of a program for tracing seismic rays through heterogeneous three-dimensional media by Pereyra (see [11], [12]). The program takes a seismic "event" (earthquake, explosion) at a given location, and traces rays from this source to a network of receivers (e.g., an array of geophones on the earth's surface). The wave propagation properties of the earth in the vicinity of the source and receivers are modeled by a set of blocks limited by surface patches. Parameters that define the interfaces between the blocks, and the properties *within* each block (assumed to be smoothly varying) are given. Initial estimates of the ray paths are obtained by shooting rays from the source in a given set of directions. These paths are refined using PASVA4, which attempts to trace all rays between a given source-receiver pair with a given "signature." The signature is a list of interfaces, which the ray is assumed to contact in a specified order.

PASVA4 is able to handle discontinuities in the material properties due to layer interfaces, and to efficiently consider any additional algebraic parameters present in the problem. Discontinuities are handled by forcing the interface intersections to be repeated meshpoints. Extra conditions (continuity of the ray, laws of refraction) are introduced to determine the additional unknowns. We mentioned in § 2 that the resulting differential-algebraic system can be solved by calling SOLVE more often than DECOMP on each iteration. This is discussed in more detail in [10].

In order to test the effect of the new linear solvers on the overall speed of the ray tracing program, the code was modified so that Algorithm D4 took the place of DECOMP and Algorithm S4 took the place of SOLVE. The original DECOMP was retained as a backup, because of its possibly better stability properties. An "interface" routine was introduced to rearrange the rows of the Jacobian and produce the structure of Fig. 2. This introduces a small overhead, which could be avoided by making appropriate changes to the routines that fill in this matrix.

Comparative results for four problems are given in Table 19. In all problems, rays were traced from a point on the surface to a rectangular network of receivers, also on the surface. Rays with a number of different signatures were obtained. The simplest rays were those which traveled from the source to the first interface, and then were reflected back to a receiver. Other rays penetrated as far as eight interfaces below the surface. Many ray paths were quite complex, because the three-dimensional geometries modeled by the four examples were nontrivial. For example, the problem "fold" contains an interface that literally folds back on itself, and so rays can pass through

TABLE 19
Runtimes in seconds on CRAY-2 *for raytracing codes on four problems.*

	Original code	Modified code
fold	269	175
reverse fault	208	140
norf	154	105
mushroom	1124	849

the fold before or after reflecting from the interface. Similar ray paths can be found in "mushroom," whose main feature is a subsurface salt dome. The original and modified codes produced identical results.

Almost all the runtime for both codes is spent in either executing DECOMP and SOLVE, or in evaluating functions and derivatives associated with the discretized problem. The latter part of the program is not helped much by vectorization. It consists of millions of calls to small routines that perform such tasks as evaluating the splines that define the layer interfaces, or evaluating derivatives with respect to the spline parameters of the point at which a ray contacts the interface. On the other hand, since the values $m = 7$, $p = 4$ are always applicable for ray-tracing problems, and since $n \geqq 18$, we would expect from Tables 1-12 that substantial improvements are possible. This is indeed the case, as we see from the run profiles in Tables 20 and 21. Here, the fold and reverse fault problems are profiled using the flowtrace utility on the CRAY-2. Runtimes are given in absolute terms, and also as a proportion of the runtime for their program. (A slight discrepancy can be detected between these times and those in the relevant lines of Table 19; this is because of the overhead introduced by the flowtrace utility.) In absolute terms, the time needed by DECOMP/SOLVE is reduced by a factor of three, and as a proportion of the total runtime it drops from 45.8 percent to 20.9 percent in the case of the fold problem and 44.7 percent to 22.0 percent in the case of the reverse fault. Note that the number of DECOMP and SOLVE calls is the same for both codes—no problems of stability arose in the use of D4.

Appendix. Validity of the new factorization algorithms. Suppose the matrix considered in § 3 is the Jacobian of a system of nonlinear equations, which arises from a one-step global discretization scheme applied to the problem (1.1)-(1.2). Here we show that if this Jacobian is evaluated in the vicinity of an isolated solution of the nonlinear system, then under mild conditions, the factorization schemes described in § 3 will work, for a sufficiently fine mesh. This is done by referring to some of the results of Keller [8], in particular, those results that prove that the factorization scheme of § 2 is (asymptotically) valid.

TABLE 20
Run profile for fold problem.

	Original code	Modified code
calls to decomp	3776	3776
time in decomp	60.2 secs (21.4%)	18.5 secs (9.3%)
calls to solve	29681	29681
time in solve	68.7 secs (24.4%)	23.1 secs (11.6%)
time in interface	—	1.5 secs (0.8%)

TABLE 21
Run profile for reverse fault problem.

	Original code	Modified code
calls to decomp	3559	3559
time in decomp	44.2 secs (19.2%)	14.3 secs (9.0%)
calls to solve	31772	31772
time in solve	58.7 secs (25.5%)	20.6 secs (13.0%)
time in interface	—	1.4 secs (0.9%)

For simplicity we start by considering a linear version of (1.1):

$$\text{(A1)} \qquad \begin{aligned} &y'(t)-A(t)y(t)=f(t), \qquad a\leqq t\leqq b,\\ &B_a y(a)+B_b y(b)=\beta, \end{aligned}$$

where $f, y, \beta \in R^m$, $B_a, B_b, A \in R^{m\times m}$. In one-step discretization schemes for (A1), the submatrices A_i and B_i in (3.5) have the form

$$\text{(A2)} \qquad \begin{aligned} &A_i(h)=-h_{i+1}^{-1}I+\tilde{A}_i(h),\\ &B_i(h)=h_{i+1}^{-1}I+\tilde{B}_i(h), \qquad i=1,\cdots,n-1, \end{aligned}$$

where the mesh

$$a=t_1<t_2<\cdots<t_n=b$$

is used, with

$$h_i=t_i-t_{i-1}, \qquad h=\max_i h_i.$$

Here $\|\tilde{A}_i(h)\|$ and $\|\tilde{B}_i(h)\|$ are assumed to be uniformly bounded as functions of h. The boundary condition matrices in (A1) are used to define the last row of blocks in (3.5):

$$\text{(A3)} \qquad D=B_a, \qquad A_n=B_b.$$

The approximate discretized solution of (A1) is then found by solving the following system of linear equations:

$$\text{(A4)} \qquad A^h u^h=F^h,$$

where A^h is the matrix from (3.5) and F^h has the form

$$\text{(A5)} \qquad F^h=\begin{pmatrix} F_1(f,h)\\ \vdots\\ F_{n-1}(f,h)\\ \beta \end{pmatrix}.$$

Each F_i is the right-hand side of the ith equation in the one-step discretization scheme, and

$$u^h=\begin{pmatrix} u_1\\ \vdots\\ u_n \end{pmatrix}$$

contains the approximate discrete solution of (A1) at each meshpoint.

The following result is a direct consequence of Keller [8, Cor. 2.11] and Isaacson and Keller [6, p. 396].

LEMMA A1. *If the one-step scheme* (A2)-(A5) *is consistent for the initial value problem obtained by setting* $B_a=I$ *and* $B_b=0$ *in* (A1), *then it is stable and consistent for all problems* (A1) *that have a unique solution.*

By considering the special case of a result of Keller [8, Thm. 2.20], it can be shown that a factorization similar to that produced by Algorithm D3 is valid.

THEOREM A2. *Let* (A1) *have a unique solution, and let the one-step scheme* (A2)-(A3) *be consistent for the initial value problem form of* (A1) (*with* $B_a=I$, $B_b=0$).

Then there is a small positive number $\bar{h}$ such that for all $h \leqq \bar{h}$, the factorization

$$A^h = \begin{pmatrix} I & & & \\ & \ddots & & \\ & & I & \\ G_1 & \cdots & G_{n-1} & I \end{pmatrix} \begin{pmatrix} A_1 & B_1 & & \\ & \ddots & \ddots & \\ & & A_{n-1} & B_{n-1} \\ & & & \hat{A}_n \end{pmatrix} \tag{A6}$$

is valid, with no need for pivoting.

Proof. Set $p = 0$ in Theorem 2.20 of [8], and note that in this case the partial pivoting strategy defined in Theorem 2.17 of [8] (which is equivalent to the row partial pivoting in DECOMP) is not needed.

The factorization (A6) is simply related to (3.14) by appropriate introduction of permutation matrices; in particular, for the bottom row of blocks,

$$D_i = P_n G_i P_i^T, \qquad 1, \cdots, n-1,$$
$$\tilde{A}_n = P_n \hat{A}_n.$$

The validity of the other factorizations (of Algorithms D1, D2, and D4) now follows directly from the validity of (3.14), since the pivot sequence is the same in both cases. Using the submatrices L_i from (3.6) we can define

$$\hat{L} = \begin{pmatrix} L_1 & & & \\ & L_2 & & \\ & & \ddots & \\ & & & L_n \end{pmatrix}.$$

Then (3.6) and (3.14) can be related by

$$PA^h = LU = (L\hat{L}^{-1})(\hat{L}U) = \tilde{L}\tilde{U}.$$

It remains to extend these results for the linear system (A1) to the nonlinear case (1.1). This is done in [8, § 2.3]. The matrices $\tilde{A}_i(h)$, $\tilde{B}_i(h)$, B_a, and B_b now arise from derivatives of the functions f and g in (1.1) and (1.2). The partly separated nature of the boundary conditions in (1.2) means that the first p rows of B_b and the last q rows of B_a are always zero. The system (A4) is now solved iteratively, with the sequence of vectors u^h now (hopefully) converging to the discrete approximation of the true solution. Basically, the analysis of [8] can be used to show that if the solution of (1.1)-(1.2) is isolated, and the current iterate u^h is sufficiently close to the discrete solution, then the factorization schemes of § 3 are valid.

Simple examples of discretizations for which the various factorization schemes described here may fail on a coarse mesh are provided by the following problem:

$$y'(t) = -2y(t), \qquad t \in [0, T]. \tag{A7}$$

Here $T > 1$ is an integer. The trapezoidal rule discretization (1.2a) on the usual mesh yields the equations

$$y_{i+1}(1 + h_{i+1}) + y_i(-1 + h_{i+1}) = 0, \qquad i = 1, \cdots, n-1.$$

Applying the boundary condition $y(0) = A$ (for which the exact solution is $y(t) = A e^{-2t}$), setting $h_i \equiv 1$, and using the ordering of Fig. 1, we obtain the following

coefficient matrix:

$$\begin{pmatrix} 1 & & & & \\ & 2 & & & \\ & & 2 & & \\ & & & \ddots & \\ & & & & 2 \end{pmatrix}.$$

This trivial matrix could clearly be factorized by any "reasonable" algorithm. However, the ordering of Fig. 2 produces

$$\begin{pmatrix} 0 & 2 & & & \\ & 0 & 2 & & \\ & & \ddots & \ddots & \\ & & & 0 & 2 \\ 1 & 0 & \cdots & 0 & 0 \end{pmatrix},$$

and so D1–D4 would fail, since in the notation (3.5), $A_i \equiv 0$. If instead we use the coupled boundary condition

$$y(0) = y(T) = A$$

(for which the exact solution is $y(t) = A\,e^{-2t}/(1+e^{-2T})$), Fig. 1 and Fig. 2 orderings are identical:

$$\begin{pmatrix} 0 & 2 & & & \\ & 0 & 2 & & \\ & & \ddots & \ddots & \\ & & & 0 & 2 \\ 1 & 0 & \cdots & 0 & 1 \end{pmatrix}.$$

All the factorization algorithms (including DECOMP) will fail on this matrix.

Acknowledgments. We thank the Advanced Computing Research Facility at Argonne National Laboratory and the Computer Science Department at Stanford University for the use of their Alliant FX/8 computers. We also thank the Office of Advanced Scientific Computing at the NSF for giving us access to the CRAY X-MP/48 at the San Diego Supercomputer Center, and Cray Research for the use of a CRAY-2 at their Mendota Heights, Minnesota location. Jack Dongarra provided helpful comments regarding the CRAY and Alliant architectures, and we also acknowledge with gratitude the important contributions of the editor and referees of earlier versions of this paper, in particular the referee who suggested the algorithms D4 and S4.

REFERENCES

[1] *Alliant Users Guide*, Alliant Computer Corporation, 1985.

[2] J. R. CASH, *Numerical integration of nonlinear two-point boundary value problems using iterated deferred corrections. I: A survey and comparison of some one-step formulae*, Comput. Maths. Appl., 12A (1986), pp. 1029–1048.

[3] J. C. DIAZ, A. FAIRWEATHER, AND P. KEAST, FORTRAN *packages for solving certain almost block diagonal linear systems by modified alternate row and column elimination*, ACM Trans. Math. Software, 9 (1983), pp. 358–375.

[4] K. GALLIVAN, W. JALBY, U. MEIER, AND A. SAMEH, *The impact of hierarchical memory systems on linear algebra algorithm design*, CSRD Tech. Report, University of Illinois, Urbana, Illinois, October 1986.

[5] M. GOLDMANN, *Vectorization of the multiple shooting method for the nonlinear boundary value problem in ordinary differential equations*, Parallel Comput., 7 (1988), pp. 97–110.

[6] E. ISAACSON AND H. B. KELLER, *Analysis of Numerical Methods*, John Wiley, New York, 1966.

[7] H. B. KELLER, *Accurate difference methods for non-linear two-point boundary value problems*, SIAM J. Numer. Anal., 11 (1974), pp. 305–320.

[8] ———, *Numerical Solution of Two-Point Boundary Value Problems*, CBMS–NSF Regional Conference Series in Applied Mathematics 24, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1976.

[9] M. LENTINI AND V. PEREYRA, *An adaptive finite difference solver for nonlinear two-point boundary value problems with mild boundary layers*, SIAM J. Numer. Anal., 14 (1977), pp. 91–111.

[10] ———, PASVA4: *An* ODE *boundary solver for problems with discontinuous interfaces and algebraic parameters*, Mat. Apl. Comput., (1983), pp. 103–118.

[11] V. PEREYRA, *Improved automatic two-point ray tracing in inhomogeneous three-dimensional media*, Report 87-01, Weidlinger Associates Inversion Project, Weidlinger Associates, Los Altos, CA, 1987.

[12] ———, *Numerical methods for inverse problems in three-dimensional geophysical modeling*, Appl. Numer. Math., 4 (1988), pp. 97–139.

[13] W. SCHÖNAUER, *Scientific Computing on Vector Computers*, North-Holland, Amsterdam, 1987.

[14] J. M. VARAH, *Alternate row and column elimination for solving certain linear systems*, SIAM J. Numer. Anal., 13 (1976), pp. 71–75.