.

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

LAPACK Working Note #2

Block Reduction of Matrices to Condensed Forms for Eigenvalue Computations

Jack J. Dongarra, Sven J. Hammarling, and Danny C. Sorensen

Mathematics and Computer Science Division

Technical Memorandum No. 99

September 1987

.

ABSTRACT

In this paper we describe block algorithms for the reduction of a real symmetric matrix to tridiagonal form and for the reduction of a general real matrix to either bidiagonal or Hessenberg form using Householder transformations. The approach is to aggregate the transformations and to apply them in a blocked fashion, thus achieving algorithms that are rich in matrix-matrix operations. These reductions to condensed form typically comprise a preliminary step in the computation of eigenvalues or singular values. With this in mind, we also demonstrate how the initial reduction to tridiagonal or bidiagonal form may be pipelined with the divide and conquer technique for computing the eigensystem of a symmetric matrix or singular value decomposition of a general matrix to achieve algorithms which are load balanced and rich in matrix-matrix operations.

Block Reduction of Matrices to Condensed Forms for Eigenvalue Computations

*Jack J. Dongarra**, Sven J. Hammarling, and Danny C. Sorensen**

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439

and

Numerical Algorithms Group Ltd.
NAG Central Office, Mayfield House
256 Banbury Road, Oxford OX2 7DE

Abstract — In this paper we describe block algorithms for the reduction of a real symmetric matrix to tridiagonal form and for the reduction of a general real matrix to either bidiagonal or Hessenberg form using Householder transformations. The approach is to aggregate the transformations and to apply them in a blocked fashion, thus achieving algorithms that are rich in matrix-matrix operations. These reductions to condensed form typically comprise a preliminary step in the computation of eigenvalues or singular values. With this in mind, we also demonstrate how the initial reduction to tridiagonal or bidiagonal form may be pipelined with the divide and conquer technique for computing the eigensystem of a symmetric matrix or singular value decomposition of a general matrix to achieve algorithms which are load balanced and rich in matrix-matrix operations.

1. Introduction

The key to using a high-performance computer effectively is to avoid unnecessary memory references. In most computers, data flows from memory into and out of registers and from registers into and out of functional units, which perform the given instructions on the data. Algorithm performance can be dominated by the amount of memory traffic rather than by the number of floating-point operations involved. The movement of data between memory and registers can be as costly as arithmetic operations on the data. This provides considerable motivation to restructure existing algorithms and to devise new algorithms that minimize data movement.

Along these lines there has been much activity in the past few years involving redesign of some of the basic routines in linear algebra[10, 7, 9]. A number of researchers have demonstrated the effectiveness of block algorithms on a variety of modern computer architectures with vector-processing or parallel-processing capabilities, on which potentially high performance can easily be degraded by excessive transfer of data between different levels of memory (vector registers, cache, local memory, main

---

* Work supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U. S. Department of Energy, under Contract W-31-109-Eng-38.

** Work supported in part by the National Science Foundation.

memory, or solid-state disks)[8, 10, 1, 13, 2, 3, 4]. This redesign has led to the development of algorithms that are based on matrix-vector and matrix-matrix techniques[10, 2].

This approach to software construction is well suited to computers with a hierarchy of memory and true parallel-processing computers. A survey that provides a description of many advanced-computer architectures may be found in [6]. For those architectures it is often preferable to partition the matrix or matrices into blocks and to perform the computation by matrix-matrix operations on the blocks. By organizing the computation in this fashion we provide for full reuse of data while the block is held in cache or local memory. This approach avoids excessive movement of data to and from memory and gives a *surface-to-volume* effect for the ratio of operations to data movement, i.e. $O(n^3)$ operations to $O(n^2)$ data movement. In addition, on architectures that provide for parallel processing, parallelism can be exploited in two ways: (1) operations on distinct blocks may be performed in parallel; and (2) within the operations on each block, scalar or vector operations may be performed in parallel. For a description of blocked implementation for Cholesky factorization, LU decomposition, and matrix multiply and the specifications for a set of building blocks to aid the development of block algorithms see [5].

Many of the most successful algorithms for computing eigenvalues or singular values of matrices require an initial reduction to condensed form. Typically, this condensed form is well suited to the implementation of an underlying iterative process used to compute the eigenvalues or singular values. We present block algorithms suitable for computing three different condensed forms. These are the reduction of a symmetric matrix to tridiagonal form, and the reduction of a (real) general matrix to either upper Hessenberg form or bidiagonal form. The reduction of a symmetric matrix to tridiagonal form dominates the computation of eigenvalues if no eigenvectors are required and represents about half the work if both eigenvalues and eigenvectors are sought. A similar remark is appropriate for the reduction of a general matrix to bidiagonal form in preparation for the computation of singular values. When the full eigensystem or singular value decomposition is desired then divide and conquer techniques are appropriate for both of these computations and we shall discuss how to pipeline the reduction to condensed form with a divide and conquer scheme.

2. The Algorithm - Reduction to Tridiagonal Form

We usually think of applying a sequence of Householder transformations to reduce the original symmetric matrix to symmetric tridiagonal form. We apply the transformations as similarity transformations to preserve the eigenvalues of the original matrix. The process can be described as follows:

$$P_i = I - u_i u_i^T, \quad u_i^T u_i = 2$$

$$T = P_{n-2} \cdots P_2 P_1 A P_1 P_2 \cdots P_{n-2}.$$

Each transformation P_i is designed to introduce zeros in the i^{th} column (and row) of the matrix below the subdiagonal (and above the superdiagonal) so as to leave the upper part of the matrix in tridiagonal form and the lower part full and symmetric. At the i^{th} step of the process, the matrix is of the form

$$\left[\begin{array}{cccccccc}
x & x & & & & & & \\
x & x & x & & & & & \\
 & x & . & . & & & & \\
 & & . & . & x & & & \\
 & & & x & x & x & x & x \\
 & & & & x & x & x & x \\
 & & & & x & x & x & x \\
 & & & & x & x & x & x
\end{array}\right].$$

To describe the algorithmic details of this reduction, we use the notation $A^{(i:n,j:n)}$ to denote the $(n-i)\times(n-j)$ submatrix of A beginning at the (i,j) location of A; we denote the subvector of a vector a beginning at the $i-th$ position by $a^{(i:n)}$; and the $i-th$ component of a vector a by $a^{(i)}$. The vector u_i is constructed from the i^{th} column of the reduced matrix so that

$$\alpha_i = -sign(a_i^{(i+1)})\,||a_i^{(i+1:n)}||_2$$

$$u_i^{(i+1)} = sqrt(1 - a_i^{(i+1)}/\alpha_i)$$

$$u_i^{(i+2:n)} = -a_i^{(i+2:n)}/\alpha_i$$

$$u_i^{(1:i)} = 0\ .$$

In practice u_i is constructed and applied to the matrix as follows:

$$y_i = A_i u_i$$

$$v_i = y_i - \tfrac{1}{2}(y_i^T u_i)u_i$$

$$A_{i+1} = A_i - u_i v_i^T - v_i u_i^T\ . \tag{2.1}$$

In the process A is repeatedly modified by a symmetric rank-two update. This requires updates of half the $n-i\times n-i$ elements of the symmetric matrix at each stage of the process.

To achieve better memory utilization we can consider aggregating a sequence of transformations, say p of them, so that the matrix will be updated by a rank $2p$ symmetric matrix. Such an implementation would be as follows: instead of explicitly updating the matrix with the rank two change, we form only the second column (row) of A_2, say a_2. We then update a_2 by applying (2.1) in the following way:

$$a_2 = a_2 - v_1^{(2)}u_1 - u_1^{(2)}v_1\ ,$$

where $v_1^{(2)}$ and $u_1^{(2)}$ are the second elements of the vectors v_1 and u_1. From this we can compute u_2; and y_2 would be formed as, $y_2 = A_2 u_2$. However we have not explicitly formed A_2. We can construct y_2 as follows:

$$\begin{aligned} y_2 &= A_2 u_2 \\ &= (A_1 - u_1 v_1^T - v_1 u_1^T)u_2 \\ &= A_1 u_2 - (v_1^T u_2)u_1 - (u_1^T u_2)v_1\ . \end{aligned}$$

We could then explicitly form A_3 as a symmetric rank four update as follows:

$$\begin{aligned} A_3 &= A_2 - u_2 v_2^T - v_2 u_2^T \\ &= A_1 - u_1 v_1^T - v_1 u_1^T - u_2 v_2^T - v_2 u_2^T\ . \end{aligned}$$

We could have continued the process and in general found for a rank $2p$ update:

$$A_{p+1} = A_1 - UV^T - VU^T\ ,$$

where

$$U = (u_1, u_2, ..., u_p)$$

$$V = (v_1, v_2, ..., v_p)$$

$$v_p = y_p - \tfrac{1}{2}(y_p^T u_p)u_p$$

$$y_{p+1} = (A_1 - UV^T - VU^T)u_{p+1}$$

$$a_{p+1}^{(p+1:n)} = a_{p+1}^{(p+1:n)} - \sum_{i=1}^{p}(v_i^{(p+1)}u_i + u_i^{(p+1)}v_i) .$$

Thus, A_{p+1} can be formed by a rank $2p$ symmetric update that is rich in matrix-matrix operations.

Algorithm 1

U and V are temporary $n \times p$ arrays, which are reused for each iteration of the k loop
n is the order of the matrix
p is the blocking
$N = (n-2)/p$
for $k = 1, N$
    $s = (k-1)p + 1$
    for $j = s, s+p-1$
        $a_j = a_j - \sum_{i=s}^{j-1}(v_i^{(j+1)}u_i + u_i^{(j+1)}v_i)$
        $\alpha_j = -sign(a_j^{(j+1)}) \, || a_j^{(j+1:n)} ||_2$
        $u_j^{(j+1)} = sqrt(1 - a_j^{(j+1)}/\alpha_j)$
        $u_j^{(j+2:n)} = -a_j^{(j+2:n)}/\alpha_j$
        $y_j = (A_s - U_{j-1}V_{j-1}^T - V_{j-1}U_{j-1}^T)u_j$
        $v_j = y_j - \tfrac{1}{2}(y_j^T u_j)u_j$
        $U_j = \left(U_{j-1}, u_j \right)$
        $V_j = \left(V_{j-1}, v_j \right)$
    end
    perform symmetric rank 2p update on submatrix
    $A^{(s+p:n,s+p:n)} = \left(A - UV^T - VU^T \right)^{(s+p:n,s+p:n)}$
end

U_k is a lower trapezoidal matrix with the first column having its first non-zero element in position $s + 1$ and the p-th column having its first non-zero element in position $s + p$. Y_k is a lower trapezoidal matrix with the first column having its first non-zero element in position s and the p-th column having its first non-zero element in position $s + p - 1$.

Notice that to aggregate the Householder transformations during the construction of the vector y_j we perform a matrix-vector multiplication with the submatrix A_s in the j loop.

Algorithm 1 constructs k block transformations and applies it to the matrix. We will call this a ‘‘*right-looking algorithm.*’’ Notice that at each of the k stages we are updating a submatrix of size

$n-s+1 \times n-s+1$. We can further reduce the amount of data referenced by the following algorithm.

Algorithm 2

for $k = 1, N$

$s = (k-1)p + 1$

Apply the previous $k-1$ *block transformations to* $A^{(s:n,s:s+p-1)}$

Compute U_k and V_k

end

At each stage of this algorithm we are only modifying a $n-s+1 \times p$ matrix. We will call this algorithm the *''left-looking algorithm.''* This algorithm will require an access to the submatrix A_s in the loop, however it avoids an update of the matrix at the

3. Reduction to Hessenberg Form

Not surprisingly, the same approach can be used in the reduction to Hessenberg form. Here we have

$$H = P_{n-2} \cdots P_2 P_1 A P_1 P_2 \cdots P_{n-2} ,$$

where H is upper Hessenberg. The idea of using a rank 2 or higher update in this context was discussed in [9]. Here it is convenient to use slightly modified formulas to those in [9] given by

$$y_i = A_i^T u_i \ , \quad z_i = A^i u_i$$

$$v_i = y_i - ½(z_i^T u_i) u_i \ , \ \ w_i = z_i - ½(y_i^T u_i) u_i$$

$$A_{i+1} = A_i - u_i v_i^T - w_i u_i^T .$$

When A is symmetric, $y_i = z_i$, and $v_i = u_i$ and these equations are as in the tridiagonal case. The vector u_i is computed from the same equations as for the tridiagonal case. Here A is updated as

$$A_{p+1} = A_1 - UV^T - WU^T \ ,$$

where

$$U = (u_1, u_2, \cdots, u_p) \ , \ \ V = (v_1, v_2, \cdots, v_p) \ , \ \ W = (w_1, w_2, \cdots, w_p) \text{ and}$$

$$y_{p+1} = (A_1^T - VU^T - UW^T) u_{p+1} \ , \ \ z_{p+1} = (A_1 - UV^T - WU^T) u_{p+1} \ .$$

U, V, and Y are trapezoidal, but Z and W are not.

4. Reduction to Bidiagonal Form

A problem that is closely associated with the eigenvalue problem is to compute the Singular Value Decomposition (SVD) of a real $m \times n$ matrix A. This decomposition is directly related to the symmetric eigenvalue problem in that the singular values of A are the square roots of the eigenvalues of the symmetric positive semidefinite matrix $A^T A$. It is numerically preferable to avoid formation of $A^T A$ and the algorithm of choice involves an initial reduction of A to upper bidiagonal form B through a sequence of Householder transformations to obtain

$$A = UBV^T$$

with U and V orthogonal and B upper bidiagonal.

This initial reduction may be treated with an algorithm similar to those already presented. In this case

$$B = P_{n-2}^T \cdots P_2^T P_1^T A Q_1 Q_2 \cdots Q_{n-2} \; ,$$

where each $P_j = I - u_j u_j^T$ is an $m \times m$ Householder transformation and each $Q_j = I - v_j v_j^T$ is an $n \times n$ Householder transformation. Again we may achieve efficient memory utilization by aggregating a sequence of transformations, say p of them, so that the matrix will be updated by a matrix of rank $2p$. However, there are data dependencies within this reduction that require additional attention.

Let us suppose for the moment that the sequences $\{ u_j \}$ and $\{ v_j \}$ can be computed at will. In general,

$$(I - uu^T)A(I - vv^T) = A - uw^T - yv^T \quad ,$$

where

$$y = Av \quad , \quad z = A^T u \quad \text{and} \quad w = z - (u^T y)v \quad .$$

Thus, a straightforward extension of the tridiagionalization scheme presented in Section 2 gives the following algorithm:

Algorithm 3

U and Y are temporary $m \times p$ arrays, which are reused for each iteration of the k loop
V and W are temporary $n \times p$ arrays, which are reused for each iteration of the k loop
n is the number of columns in the matrix
m is the number of rows in the matrix
p is the blocking
$N = (n-2)/p$
for $k = 1, N$
$s = (k-1)p + 1$
for $j = s, s+p-1$
compute u_j
compute v_j

$$y_j = \left(A - U_{j-1} W_{j-1}^T - Y_{j-1} V_{j-1}^T \right) v_j$$

$$h_j = \left(A - U_{j-1} W_{j-1}^T - Y_{j-1} V_{j-1}^T \right)^T u_j$$

$$w_j = h_j - (u_j^T y_j) v_j$$

$$U_j = \left(U_{j-1} , u_j \right)$$

$$V_j = \left(V_{j-1} , v_j \right)$$

$$W_j = \left(W_{j-1} , w_j \right)$$

$$Y_j = \left(Y_{j-1} , y_j \right)$$

end
perform rank 2p update on submatrix
$$A^{(s+p:n, s+p:n)} = (A - UW^T - YV^T)^{(s+p:n, s+p:n)}$$
end

Unfortunately, it is not so straightforward to compute u_j and v_j at will. At step j of the usual bidiagonalization process, the vector u_j is nonzero in the $j-th$ entry. Hence application on the left by the corresponding Householder transformation alters the $j-th$ row of the reduced matrix A_j and knowledge of this row is required to compute the vector v_j which is nonzero in the $j+1-st$ entry. The dependencies now become even more complicated because it would appear that the transformation corresponding to v_j must be applied from the right before u_{j+1} can be computed and so on. However, we note that the above algorithm will be valid if there is an independent formula for computing the v_j since the u_j may be computed as in the previous algorithms by knowing the $j-th$ column of the reduced matrix. Indeed, there is an independent formula for computing the v_j which may be found by noting that

$$V^T A^T A V = B^T B = T$$

where $V = Q_1 Q_2 \cdots Q_{n-2}$ is precisely the sequence of transformations that would be computed in Algorithm 1 to reduce the matrix $A^T A$ to tridiagonal form $T = B^T B$. This leads to the following procedure for computing the v_j

Procedure compute v_j

$$
\begin{aligned}
& z_j = A_s^T a_j^{(s:s+p-1)} - \sum_{i=s}^{j-1} (v_i x_i^{(j+1)} + x_i v_i^{(j+1)}) \\
& \zeta_j = -sign(z_j^{(j+1)}) \,||\, z_j^{(j+1:n)} \,||_2 \\
& v_j^{(j+1)} = sqrt(1 - z_j^{(j+1)}/\zeta_j) \\
& v_j^{(j+2:n)} = -z_j^{(j+2:n)}/\zeta_j \\
& t_j = (A_s^T A_s - V_{j-1} X_{j-1}^T - X_{j-1} V_{j-1}^T) v_j ,\, A_s = A^{(s:m,s:n)} \\
& x_j = t_j - \tfrac{1}{2}(t_j^T v_j) v_j \quad . \\
& X_j = \left[X_{j-1}, x_j \right]
\end{aligned}
$$

Computation of the u_j only depends upon the $j-th$ column of the reduced matrix being in place before the $j-th$ step. Therefore, the column oriented formula given in Algorithm 1 may be adapted to give

Procedure compute u_j

$$
\begin{aligned}
& a_j = a_j - \sum_{i=s}^{j-1} (u_i w_i^{(j+1)} + y_i v_i^{(j+1)}) \\
& \alpha_j = -sign(a_j^{(j)}) \,||\, a_j^{(j:m)} \,||_2 \\
& u_j^{(j)} = sqrt(1 - a_j^{(j)}/\alpha_j) \\
& u_j^{(j+1:m)} = -a_j^{(j+1:m)}/\alpha_j
\end{aligned}
$$

If these two procedures are substituted for "compute u" and "compute v" in Algorithm 3 then it be well defined. In all of these we do not explicitly form the indicated matrix products. Instead, the matrix vector products are accumulated.

5. Relationship to the *WY* Factorization

The algorithm presented here for aggregating Householder transformations is closely related to the *WY* representation for the product of Householder matrices presented by Bischof and Van Loan[2]. The relationship is most clearly seen in the contexts of the QR factorization of a general rectangular matrix. The *WY* representation has the following form:

QR Factorization (Bischof and Van Loan)

n is the number of columns in the matrix
p is the blocking
$N = n/p$
for $k = 1, N$
  $s = (k-1)p + 1$
  for $j = s, s+p-1$
    $a_j = a_j - \sum_{i=s}^{j-1} z_i^{(j)} u_i \quad where \ z_i = A_i u_i$
    compute Householder vector u_j
    $Y_k^{(s:j)} = (Y_k^{(s:j-1)} - u_j u_j^T Y_k^{(s:j-1)}, -2u_j)$
  end
  perform rank $2p$ *update on sub−matrix*
  $A^{(s+p:n,s+p:n)} = A^{(s+p:n,s+p:n)} - UY^T A^{(s+p:n,s+p:n)}$
end

If one implements the reduction along the lines of the algorithm described in Section 2, the factorization can be described as follows:

QR Factorization (Alternative Algorithm)

n is the order of the matrix
p is the blocking
$N = n/p$
for $k = 1, N$
  $s = (k-1)p + 1$
  for $j = s, s+p-1$
    $a_j = a_j - \sum_{i=s}^{j-1} v_i^{(j+1)} u_i$
    compute Householder vector u_j
    $v_j = (A_s^T - V_k^{(s:j-1)} U_k^{(s:j-1)\,T}) u_j$
  end
  perform rank $2p$ *update on submatrix*
  $A^{(s+p:n,s+p:n)} = \left(A - UV^T \right)^{(s+p:n,s+p:n)}$
end

The two differences between the block algorithms are in the formation of v_j and Y_j and also in the update of the matrix A. For the Alternate Algorithm the vector v_j is updated using the submatrix A_s and the Bischof and Van Loan Algorithm uses information from u_j and $Y_k^{(s:j-1)}$. Thus the Bischof and Van Loan Algorithm will have fewer accesses to the data. In the update of the matrix A for the Bischof and Van Loan factorization, the update is of the form

$$A^{(s+p:n,s+p:n)} = A^{(s+p:n,s+p:n)} - UY^T A^{(s+p:n,s+p:n)} ;$$

and for the alternative factorization, the update is of the form

$$A^{(s+p:n,s+p:n)} = A^{(s+p:n,s+p:n)} - UV^T .$$

The alternative algorithm incorporates the information about the matrix A in the matrix V.

We can describe the reduction to tridiagonal form for the symmetric eigenvalue problem using the Bischof and Van Loan approach as

$$A_{p+1} = (I - USU^T)A_1(I - USU^T).$$

If we multiply out and combine terms, we can reduce the expression to

$$A_{p+1} = A_1 - ZU^T - UZ^T$$

where

$$W = A_1 US^T \text{ and } Z = W - \tfrac{1}{2} USU^T W$$

which is of the form described in Section 2. An implementation for the reductions along these lines is described by W. Harrod at the University of Illinois [12].

With the WY representation it is simple to apply the set of transformations to another matrix, as is required in back substitution for the eigenvector computation; one simply applies $(I - WY^T)$ to the matrix. To apply the transformation using the formulation in Section 2, one can use the Householder vectors to construct the matrix S such that $I - USU^T$ can be used to apply the transformations to the eigenvectors of the symmetric tridiagonal matrix, back transforming them to the eigenvectors of the original problem. The matrix S is a p by p upper triangular matrix whose k-th column is formed as follows:

$$\begin{aligned}(I - USU^T)(I - uu^T) &= I - uu^T - USU^T + USU^T uu^T \\ &= I - [U, u] \begin{pmatrix} SU^T - SU^T uu^T \\ u^T \end{pmatrix} \\ &= I - [U, u] \begin{pmatrix} S & -SU^T u \\ 0 & 1 \end{pmatrix} \begin{pmatrix} U^T \\ u^T \end{pmatrix}\end{aligned}$$

So the new column of S is

$$\begin{pmatrix} -SU^T u \\ 1 \end{pmatrix}.$$

Pipelining Reduction to Condensed Form with Determination of Eigenvalues

Recently, algorithms have been developed based upon divide and conquer strategies for the determination of eigenvalues and singular values for a matrix in condensed form[11, 14]. These methods are also rich in matrix-matrix operations and mesh very well with the block reductions presented here. This is accomplished through pipelining the initial reduction phase with the computation of eigenvalues and back-transformation of eigenvectors. These considerations are of little consequence on serial computers but have significant performance advantages on parallel-vector processors.

We use the block reduction algorithm as described above to introduce zeros in a block of the matrix, say we are at the k-th stage and have just introduced zeros into the k-th block. As we start the next block reduction, on the $k+1$-st block, we can start in parallel the eigenvalue computation on that part of the tridiagonal matrix generated from the k-th block reduction of the matrix. When we have completed eigenvalue computations from two tridiagonal segments, we can use the technique applied in the divide and conquer algorithm as described by Dongarra and Sorensen[11] to determine the eigenvalues and eigenvectors of a pair of tridiagonal matrices. Then the eigenvalues of successive pairs of blocks can be found, then pairs of pairs, etc., until the full set is determined. When the reduction to condensed form and the divide and conquer strategy are used in this pipelined fashion a highly efficient parallel algorithm can be constructed.

This discussion is made more explicit in the following example. We consider a symmetric matrix that is to be partitioned into four block columns as shown in the figure below.

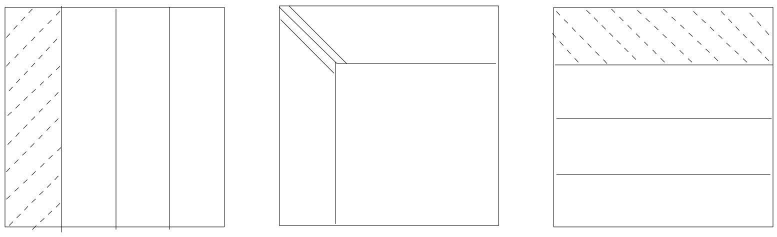

Figure 1 Partitioned Matrix

Let us associate H_k with the process of reducing the $k-th$ block A_k of the partitioned matrix to tridiagonal form using Householder transformations. Thus H_k executes a block step of Algorithm 1 (see the k-loop) on the block A_k In this algorithm we have the possibility of spawning parallel processes. Processes may cooperate in applying the resulting transformation shown in

$$A^{(s+p:n,s+p:n)} = A^{(s+p:n,s+p:n)} - U_k V_k^T - V_k U_k^T$$

in parallel. Let us denote these parallel processes by M_{kj} so that process M_{kj} is responsible for a portion of the work in the matrix multiply in the performance of Algorithm 1 by process H_k.

On completion of process H_k the tridiagonal matrix T_k is exposed and the algorithm $TQL2$ may be applied to compute the eigensystem of T_k after the rank one tearing has been computed. Let us denote this process by E_k.

Once processes E_1 and E_2 have completed, then the eigensystem of

$$\begin{pmatrix} T_1 & 0 \\ 0 & T_2 \end{pmatrix} = \begin{pmatrix} Q_1 & 0 \\ 0 & Q_2 \end{pmatrix} \left(\begin{pmatrix} D_1 & 0 \\ 0 & D_2 \end{pmatrix} + \beta \begin{pmatrix} q_1 \\ q_2 \end{pmatrix} (q_1^T, q_2^T) \right) \begin{pmatrix} Q_1^T & 0 \\ 0 & Q_2^T \end{pmatrix}$$

may be computed using the rank one updating scheme. Similarly, once processes E_3 and E_4 have completed, the eigensystem of $diag(T_3, T_4)$ may be computed. Finally the entire eigensystem may be obtained through rank one updating of these two eigensystems. Let us denote these processes as U_1, U_2 and U_0 respectively.

With the proper storage arrangements these processes obey the following large grain control flow graph:

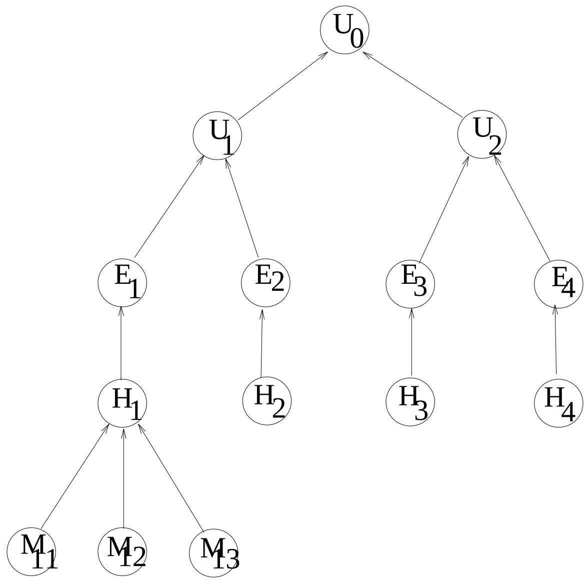

Figure 2 Large Grain Control Flow Graph

In this control flow graph a *node* denotes a process. That is, for example, a subroutine name together with the pointers to the data which the subroutine is to operate upon. A process P becomes schedulable or ready to execute when there are no incoming arcs to the node representing process P. This signifies that all of the data dependencies for process P have been satisfied through the completion of the processes that it was dependent upon. This graph; indicates that processes M_{1j} can execute immediately. Once they have completed then H_1 may report to H_2 and this process may execute and spawn processes M_{2j}. At the same time H_1 reports to process E_1 and it may begin execution. When E_1 and E_2 have both completed than process U_1 may start and so on.

To accumulate a matrix of eigenvectors, the successive Householder transformations must be multiplied from left to right in the order they are applied

$$Q = \prod_{i=1}^{n-1} \left(I - 2u_i u_i^T \right)$$

and we observe that when accumulated this way, successive applications of $I - u_i u_i^T$ effect only the last $n-i+1$ columns of Q. Thus, application of the Givens transformations associated with E_1 may be applied as soon as the products of the Householder transformations associated with H_1 have been multiplied out. These transformations may be applied independently of the computation of H_k for $k > 1$ because these matrices effect columns that are independent of the columns effected by E_i.

When we do not wish to find eigenvectors there is no reason to store the product Q of these Householder transformations. Nor is it necessary to accumulate the product of the successive eigenvector transformations resulting from the updating problem. That is, we do not need to overwrite Q with

$$Q \leftarrow Q \begin{pmatrix} Q_1 & 0 \\ 0 & Q_2 \end{pmatrix} \hat{Q}$$

where Q_1, Q_2 and $\hat{Q}$ are the matrices appearing in equations (2.2) and (3.1) above. Instead, we may simply discard Q. Then the vector q_1 may be formed as $\hat{T}_1$ is transformed to D_1 in (2.2) by accumulating the products of the transformations constructed in TQL2 that make up Q_1 against the vector e_k. If there is more than one division then Q_1 will have been calculated with the updating scheme. In this case we do not calculate all of Q_1 but instead use the component-wise formula for the eigenvectors to pick off the appropriate components needed to form q_1.

6.

7. Operations Counts and Storage

An analysis of the number of floating-point operations (counting additions and multications) for the reduction to tridiagonal form of the standard algorithm reveals an operation count of

$$\frac{4}{3}n^3 + \frac{7}{2}n^2 + \frac{1}{6}n - 25 \; flops.$$

In aggregating the transformations to perform the block reduction additional work is required in the formulation of y_j in Algorithm 1. The additional work for a block size p amounts to:

$$(2p - \frac{3}{2})n^2 + (\frac{-2}{3}p^2 - 2p + \frac{13}{6})n + (\frac{-4}{3}p^2 - 4p)$$

floating point additions and multiplications being performed.

The algorithm can be organized so that the vectors u_i overwrite the lower part of the matrix (as we do in the standard version of the software), but additional workspace of size $n \times p$ is required to store the current block of V.

8. Experimental Results

The following results were generated on an Alliant FX/8 computer using eight processors. The Alliant FX/8 is a parallel processor where each of the processors has vector registers and can perform vector operations.

Table 1
Ratio of execution times (speedups) between the EISPACK routine and the blocked version on the Alliant FX/8

Order	Ratio TRED1/TREDB	Ratio ORTHES/ORTHSB
100	1.94	2.59
200	2.39	3.01
300	2.40	3.23
400	2.39	3.35
500	2.36	3.46

The following results were generated on the CRAY X-MP using one processor.

Table 2
Ratio of execution times (speedups) between the EISPACK routine and the blocked version on the Cray X-MP

Order	Ratio TRED1/TREDB	Ratio ORTHES/ORTHSB
100	1.03	1.29
200	1.10	1.52
300	1.21	1.65
400	1.23	1.79
500	1.28	1.92

1. M. Berry, K. Gallivan, W. Harrod, W. Jalby, S. Lo, U. Meier, B. Philippe, and A. Sameh, ‘‘Parallel Algorithms on the CEDAR System,’’ *CSRD Report No. 581* (1986).

2. C. Bischof and C. Van Loan, ‘‘The WY Representation for Products of Householder Matrices,’’ *SIAM SISSC* **8, 2** (March, 1987).

3. I. Bucher and T. Jordan, ‘‘Linear Algebra Programs for use on a Vector Computer with a Secondary Solid State Storage Device,’’ pp. 546-550 in *Advances in Computer Methods for Partical Differential Equations*, ed. R. Vichnevetsky and R Stepleman, IMACS (1984).

4. D.A. Calahan, ‘‘Block-Oriented Local-Memory-Based Linear Equation Solution on the CRAY-2:

Uniprocessor Algorithms,'' *Proceedings International Conference on Parallel Processing*, pp. 375-378, IEEE Computer Society Press (August 1986).

5. J.J. Dongarra, J. DuCroz, I. Duff, and S. Hammarling, ''A Proposal for a Set of Level 3 Basic Linear Algebra Subprograms,'' Argonne National Laboratory Report, ANL-MCS-TM-88 (April 1987).

6. J.J. Dongarra and I.S. Duff, ''Advanced Architecture Computers,'' Argonne National Laboratory Report, ANL-MCS-TM-57 (Revision 1) (January, 1987).

7. J.J. Dongarra and S. C. Eisenstat, ''Squeezing the Most out of an Algorithm in Cray Fortran,'' *ACM Trans. Math. Software* **10, 3**, pp. 221-230 (1984).

8. J.J. Dongarra and T. Hewitt, ''Implementing Dense Linear Algebra Algorithms Using Multitasking on the CRAY X-MP-4,'' *SIAM J. Sci Stat. Comp.* **7, 1**, pp. 347-350 (January, 1986).

9. J.J. Dongarra, L. Kaufman, and S. Hammarling, ''Squeezing the Most out of Eigenvalue Solvers on High-Performance Computers,'' *Linear Algebra and Its Applications* **77**, pp. 113-136 (1986).

10. J.J. Dongarra and D.C. Sorensen, ''Linear Algebra on High-Performance Computers,'' pp. 3-32 in *Proceedings Parallel Computing 85*, ed. U. Schendel, North Holland (1986).

11. J.J. Dongarra and D.C. Sorensen, ''A Fully Parallel Algorithm for the Symmetric Eigenvalue Problem,'' *SIAM SISSC* **8, 2** (March, 1987).

12. Willian Harrod, *Private Communications*, 1987.

13. IBM, ''Engineering and Scientific Subroutine Library,'' *IBM* **Program Number: 5668-863** (1986).

14. L. Jessup and D. Sorensen, ''A Parallel Algorithm for Computing the Singular Value Decomposition of a Matrix,'' Argonne National Laboratory Report, ANL-MCS-TM-102 (November 1987).

LAPACK Working Notes

LAPACK Working Note #1

James Demmel, Jack J. Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, and Danny Sorensen, ''*Prospectus for the Development of a Linear Algebra Library for High-Performance Computers,*'' Argonne National Laboratory, Mathematics and Computer Science Division, Technical Memorandum No. 97, September, 1987.

LAPACK Working Note #2

Jack J. Dongarra, Sven J. Hammarling, and Danny C. Sorensen ''*Block Reduction of Matrices to Condensed Forms for Eigenvalue Computations,*'' Argonne National Laboratory, Mathematics and Computer Science Division, Technical Memorandum No. 99, September,