Linear Algebra on High Performance Computers

J.J. Dongarra and D.C. Sorensen

Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, Illinois 60439

ABSTRACT

This is a survey of some work recently done at Argonne National Laboratory in an attempt to discover ways to construct numerical software for high performance computers. The numerical algorithms discussed are taken from several areas of numerical linear algebra. We discuss certain architectural features of advanced computer architectures that will affect the design of algorithms. The technique of restructuring algorithms in terms of certain modules is reviewed. This technique has proved very successful in obtaining a high level of transportability without severe loss of performance on a wide variety of both vector and parallel computers.

The module technique is demonstrably effective for dense linear algebra problems. However, in the case of sparse and structured problems it may be difficult to identify general modules that will be as effective. New algorithms have been devised for certain problems in this category. We present examples in three important areas: banded systems, sparse QR - factorization, and symmetric eigenvalue problems.

1. Introduction

This is a survey of some work recently done at Argonne National Laboratory in an attempt to discover ways to construct numerical software for high performance computers. We have concentrated on numerical linear algebra problems involving dense matrices since we feel that the algorithms for these problems are well understood in most cases. This has allowed us to focus on utilization of the new hardware rather than on development of new algorithms for

Work supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy under Contracts W-31-109-Eng-38 and DE-AC05-84OR21400.

many of the standard problems. Nevertheless, there are instances when efficient use of the architecture begs for new algorithms.

Within the last ten years many who work on the development of numerical algorithms have come to realize the need to get directly involved in the software development process. Issues such as robustness, ease of use, and portability are now standard in any discussion of numerical algorithm design and implementation. New and exotic architectures are evolving which depend on the technology of concurrent processing, shared memory, pipelining, and vector components to increase performance capabilities. Within this new computing environment the portability issue, in particular, can be very challenging. One feels compelled to structure algorithms that are tuned to particular hardware features in order to exploit these new capabilities; yet, the sheer number of different machines appearing makes this approach intractable. It is very tempting to assume that an unavoidable byproduct of portability will be an unacceptable degradation in performance on any specific machine architecture. Nevertheless, we contend that it is possible to achieve a reasonable fraction of the performance of a wide variety of different architectures through the use of certain programming constructs.

Complete portability is an impossible goal at this point in time, but it is possible to achieve a level of transportability through the isolation of machine dependent code within certain modules. Such an approach is essential in our view, to even begin to address the portability problem.

The current generation of vector computers exploits several advanced concepts to enhance their performance over conventional computers:

Fast cycle time,

Vector instructions to reduce the number of instructions interpreted,

Pipelining to utilize a functional unit fully and to deliver one result per cycle,

Chaining to overlap functional unit execution, and

Overlapping to execute more than one independent vector instruction concurrently.

The key to utilizing 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. Performance of algorithms can be dominated by the amount of memory traffic, rather than 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.

Many of the algorithms in linear algebra can be expressed in terms of a SAXPY operation: $y \leftarrow y+\alpha x$, *i.e.* adding a multiple α of a vector x to another vector y. This would result in three vector memory references for each two vector floating point operations. If this operation comprises the body of an inner loop which updates the same vector y many times then a considerable amount of unnecessary data movement will occur. Usually, a SAXPY occurring in an inner loop will indicate that the algorithm may be recast in terms of some matrix vector operation, such as $y \leftarrow y+M*x$, which is just a sequence of SAXPYs involving the columns of the matrix M and the corresponding components of the vector x. The advantage of this is the y vector and the length of the columns of M are a fixed size throughout. This makes it relatively easy to automatically recognize that only the columns of M need be moved into registers while accumulating the result y in a vector register, avoiding two of the three memory references in the inner most loop. This also allows chaining to occur on vector machines, and results in a factor of three increase in performance on the CRAY 1. The cost of the algorithm in these cases is not determined by floating point operations, but by memory references.

2. Structure of the Algorithms

In this section we discuss the way algorithms may be restructured to take advantage of the modules introduced above. Typical recasting that occurs within LINPACK and EISPACK type subroutines is discussed here. We begin with definitions and a description of the efficient implementation of the modules themselves.

2.1 The Modules.

Only three modules are required for the recasting of LINPACK in a way that achieves the super-vector performance reported above. They are

$$z = Mw \qquad (matrix \times vector).$$

$$\hat{M} = M - wz^T \qquad (rank\ one\ modification).$$

and

$$z = Tz \qquad (solve\ a\ triangular\ system).$$

Efficient coding of these three routines is all that is needed to transport the entire package from one machine to another while retaining close to top performance.

We shall describe some of the considerations that are important when coding the matrix vector product module. The other modules require similar techniques. For a vector machine such as the CRAY-1 the vector times matrix operation should be coded in the form

$$y(*) \leftarrow y(*) + M(*,j)x(j) \quad \text{for } j = 1,2,\ldots,n \,. \tag{2.1.1}$$

In (2.1.1) the * in the first entry implies this is a column operation and the intent here is that a vector register is reserved for the result while the columns of M are successively read into vector registers, multiplied by the corresponding component of x and then added to the result register in place. In terms of ratios of data movement to floating point operations this arrangement is most favorable. It involves one vector move for two vector-floating point operations. Comparing this to the three vector moves to get the same two floating point operations when a sequence of SAXPY operations are used shows the advantage of using the matrix vector operation.

This arrangement is perhaps inappropriate for a parallel machine because one would have to synchronize the access to y by each of the processes, and this would cause busy waiting to occur. One might do better to partition the vector y and the rows of the matrix M into blocks

$$\begin{bmatrix} y_1 \\ y_2 \\ \cdot \\ \cdot \\ y_k \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \cdot \\ \cdot \\ y_k \end{bmatrix} + \begin{bmatrix} M_1 \\ M_2 \\ \cdot \\ \cdot \\ M_k \end{bmatrix} x$$

and self-schedule individual vector operations on each of the blocks in parallel:

$$y_i \leftarrow y_i + M_i x \quad \text{for } i = 1,2,...,k .$$

That is, the subproblem indexed by i is picked up by a processor as it becomes available and the entire matrix vector product is reported done when all of these subproblems have been completed.

If the parallel machine has vector capabilities on each of the processors this partitioning introduces short vectors and defeats the potential of the vector capabilities for small to medium size matrices. A better way to partition in this case is

$$y \leftarrow y + (M_1 , M_2 , \cdots , M_k) \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ x_k \end{bmatrix}$$

Again, subproblems are computed by individual processors. However, in this scheme, we must either synchronize the contribution of adding in each term $M_i x_i$ or write each of these into temporary locations and hold them until all are complete before adding them to get the final result. This scheme does prove to be effective for increasing the performance of the factorization subroutines on the smaller (order less than 100) matrices. One can easily see this if the data access scheme for *LU* decomposition shown in Figure 2.1 is studied. We see that during the final stages of the factorization vector lengths become short regardless of matrix size. For the smaller matrices, subproblems with vector lengths that are below a certain performance level represent a larger percentage of the calculation. This problem is magnified when the row-wise partitioning is used.

2.2 Recasting LINPACK subroutines

We now turn to some examples of how to use the modules to obtain various standard matrix factorizations. We begin with the *LU* decomposition of a general nonsingular matrix. Restructuring the algorithm in terms of the basic modules described above is not so obvious in the case of *LU* decomposition. The approach described here is inspired by the work of Fong and Jordan [11]. They produced an assembly language code for *LU* decomposition for the CRAY-1. This code differed significantly in structure from those commonly in use because it did not modify the entire k-th reduced submatrix at each step but only the k-th column of that matrix. This step was essentially matrix-vector multiplication operation.

Dongarra and Eisenstat [4] showed how to restructure the Fong and Jordan implementation explicitly in terms of matrix-vector operations and were able to achieve nearly the same performance from a FORTRAN code as Fong and Jordan had done with their assembly language implementation. The pattern of data references for factoring a square matrix A into $PA = LU$ (with P a permutation matrix, L unit lower triangular, U upper triangular) is shown below in Figure 2.1.1.

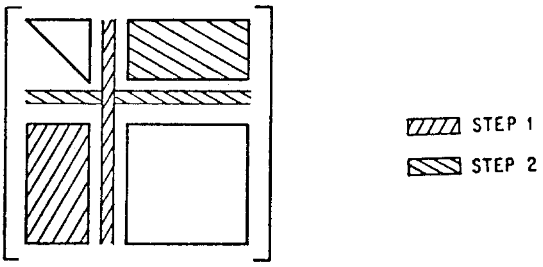


Figure 2.1.1. *LU* Data References

At the k-th step of this algorithm, a matrix formed from columns 1 through $k-1$ and rows k through n is multiplied by a vector constructed from the k-th column, rows 1 through $k-1$, with the results added to the k-th column, rows k through n. The second part of the k-th step involves a vector-matrix product, where the vector is constructed from the k-th row, columns 1 through $k-1$, and a matrix constructed from rows 1 through $k-1$ and columns $k+1$ through n, with the results added to the k-th row, columns $k+1$ through n.

One can construct the factorization by analyzing the way in which the various pieces of the factorization interact. Let us consider decomposition of the matrix A into its *LU* factorization with the matrix partitioned in the following way:

$$\begin{bmatrix} L_{11} & & \\ l_{21}^T & 1 & \\ L_{31} & l_{32} & L_{33} \end{bmatrix} \begin{bmatrix} U_{11} & u_{12} & U_{13} \\ & u_{22} & u_{23}^T \\ & & U_{33} \end{bmatrix}.$$

Multiplying L and U together and equate terms with A we have:

$$\begin{array}{lll} A_{11} = L_{11}U_{11} & a_{12} = L_{11}u_{12} & A_{13} = L_{11}U_{13} \\ a_{12}^T = l_{21}^T U_{11} & a_{22} = l_{21}^T u_{12} + u_{22} & a_{23}^T = l_{21}^T U_{13} + u_{23}^T \\ A_{31} = L_{31}U_{11} & a_{32} = L_{31}u_{12} + u_{22}l_{32} & A_{33} = L_{31}U_{13} + l_{32}u_{23}^T + L_{33}U_{33} \end{array}$$

We can now construct the various factorizations for LU decomposition by determining how to form the unknown parts of L and U given various parts of A, L and U. We give the basic algoritmic step for three variants in the following examples:

(*i*) Given the triangular matrices L_{11} and U_{11}, to construct vectors l_{21}^T and u_{12} and scalar u_{22} we must perform, $u_{12} = L_{11}^{-1}a_{11}$, $l_{21}^T = U_{11}^{-1}a_{11}^T$, $u_{22} = a_{22} - l_{21}^T u_{12}$.

Since these operations deal with triangular matrix L_{11} and U_{11} they can be expressed in terms of solving triangular systems of equations.

(*ii*) Given the rectangular matrices L_{31} and U_{12}, and the vectors l_{21}^T and u_{12}, we can form vectors l_{32} and u_{23}^T and scalar u_{22} by forming $u_{23} = a_{23}^T - l_{21}^T U_{13}$, $u_{22} = a_{22} - l_{21}^T u_{12}$, and $l_{32} = (a_{23} - L_{31}u_{12})/u_{22}$,

Since these operations deal with rectangular matrices and vectors they can be expressed in terms of simple matrix-vector operations.

(*iii*) Given the triangular matrix L_{11}, the rectangular matrix L_{31}, and the vector l_{21}^T we can construct vectors u_{12} and l_{32} and scalar u_{22} by forming $u_{12} = L_{11}^{-1}a_{12}$, $u_{22} = a_{22} - l_{21}^T u_{12}$, $l_{32} = (a_{32} - L_{31}u_{12})/u_{22}$.

These operations deal with a triangular solve and a matrix vector multiply.

The same ideas for use of high-level modules can be applied to other algorithms, including matrix multiply, Cholesky decomposition, and QR factorization.

For the Cholesky decomposition the matrix dealt with is symmetric and positive definite. The factorization is of the form

$$A = LL^T \ ,$$

where $A = A^T$ and is positive definite. If we assume the algorithm proceeds as in LU decomposition, but reference only the lower triangular part of the matrix, we have an algorithm based on matrix-vector operations which accomplishes the desired factorization.

The final method we shall discuss is the QR factorization using Householder transformations. Given a real $m \times n$ matrix A, the routine must produce an $m \times m$ orthogonal matrix Q and an $n \times n$ upper triangular matrix R such that

$$A = Q \begin{bmatrix} R \\ 0 \end{bmatrix} .$$

Householder's method consists of constructing a sequence of transformations of the form

(2.2.1) $$I - \alpha w w^T \ \text{,where } \alpha \, w^T w = 2.$$

The vector w is constructed to transform the first column of a given matrix into a multiple of the first coordinate vector e_1. At the k-th stage of the algorithm one has

$$Q_{k-1}^T A = \begin{bmatrix} R_{k-1} & S_{k-1} \\ 0 & A_{k-1} \end{bmatrix} ,$$

and w_k is constructed such that

(2.2.2) $$\left[I - \alpha_k w_k w_k^T \right] A_{k-1} = \begin{bmatrix} \rho_k & s_k^T \\ 0 & A_k \end{bmatrix} .$$

The factorization is then updated to the form

$$Q_k^T A = \begin{bmatrix} R_k & S_k \\ 0 & A_k \end{bmatrix} ,$$

with

$$Q_k^T = \begin{bmatrix} I & 0 \\ 0 & I - \alpha_k w_k w_k^T \end{bmatrix} Q_{k-1}^T .$$

However, this product is not explicitly formed, since it is available in product form if we simply record the vectors w in place of the columns they have been used to annihilate. This is the basic algorithm used in LINPACK [8] for computing the QR factorization of a matrix. This algorithm may be coded in terms of two of the modules. To see this, just note that the operation of applying a transformation shown on the left hand side of (2.2.2) above may be broken into two steps:

(2.2.3) $$z^T = w^T A \qquad (\textit{vector} \times \textit{matrix})$$

and

$$\hat{A} = A - \alpha z^T \qquad (\textit{rank one modification}).$$

2.3 Restructuring EISPACK subroutines

As we have seen, all of the main routines of LINPACK can be expressed in terms of the three modules described in Section 2.1. The same type of restructuring may be used to obtain efficient performance from EISPACK subroutines. A detailed description of this may be found in [10]. In the following discussion we just outline some of the basic ideas used there. Many of the algorithms implemented in EISPACK have the following form:

```
Algorithm (2.3.1):
For i = 1,....
   Generate matrix $T_i$
   Perform transformation $A_{i+1} \leftarrow T_i A_i T_i^{-1}$
End .
```

Because we are applying similarity transformations, the eigenvalues of A_{i+1} are those of A_i. Since the application of these similarity transformations represents the bulk of the work, it is important to have efficient methods for this operation. The main difference between this situation and that encountered with linear equations is that these transformations are applied from both sides. The transformation matrices T_i used in (2.3.1) are of different types depending upon the particular algorithm.

The simplest are the stabilized elementary transformation matrices which have the form $T = LP$, where P is a permutation matrix, required to maintain numerical stability [12,29,32], and L has the form

$$\begin{bmatrix} 1 & & & & & \\ & \cdot & & & & \\ & & 1 & & & \\ & & & 1 & & \\ & & & * & 1 & \\ & & & \cdot & & \cdot \\ & & & * & & & 1 \end{bmatrix} .$$

The inverse of L has the same structure as L and may be written in terms of a rank one modification of the identity in the following way:

$$L^{-1} = \begin{bmatrix} I & 0 \\ 0 & I - we_1^T \end{bmatrix},$$

with $e_1^T w = 0$. If we put

$$PAP^T = \begin{bmatrix} \hat{A} & B \\ C & D \end{bmatrix},$$

then

$$TAT^{-1} = \begin{bmatrix} I & 0 \\ 0 & I + we_1^T \end{bmatrix} \begin{bmatrix} \hat{A} & B \\ C & D \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & I - we_1^T \end{bmatrix}$$

$$= \begin{bmatrix} \hat{A} & B - be_1^T \\ C + wc_1^T & D + wd_1^T - fe_1^T - \delta we_1^T \end{bmatrix},$$

where $c_1^T = e_1^T C$, $d_1^T = e_1^T D$, $b = Bw$, $f = Dw$, $\delta = d_1^T w$ and e_1 is the first co-ordinate vector (of appropriate size). The appropriate module to use therefore, is the rank one modification. However, more can be done with the rank two correction that takes place in the modification of the matrix D above.

In most of the algorithms the transformation matrices T_i are Householder matrices of the form (2.2.1) shown above. This results in a rank two correction that might also be expressed as a sequence of two rank on corrections. Thus, it would be straightforward to arrange the similarity transformation as two successive applications of the scheme (2.2.3) discussed above. However, more can be done with a rank two correction as we now show.

First suppose that we wish to form $(I-\alpha ww^T)A(I-\beta uu^T)$, where for a similarity transformation $\alpha = \beta$ and $w = u$. We may replace the two rank one updates by a single rank two update using the following algorithm.

```
Algorithm 2.3.2
1. $v^T = w^T A$
2. $x = Au$
3. $y^T = v^T-(\beta w^T x)u^T$
4. Replace $A$ by $A-\beta xu^T-\alpha wy^T$
```

As a second example that is applicable to the linear equation setting, suppose that we wish to form $(I-\alpha ww^T)(I-\beta uu^T)A$, then as with Algorithm 2.3.2 we might proceed as follows.

```
Algorithm 2.3.3 :
1. $v^T = w^T A$
2. $x^T = u^T A$
3. $y^T = v^T-(\beta w^T u)x^T$
4. Replace $A$ by $A-\beta ux^T-\alpha wy^T$
```

In both cases we can see that Steps 1 and 2 can be achieved by calls to the matrix vector and vector matrix modules. Step 3 is a simple vector operation and Step 4 is now a rank-two correction, and one gets four vector memory references for each four vector floating point operations (rather than the three vector memory references for every two vector floating point operations, as in Step 2 of (2.2.3)). The increased saving is not as much as is realized with the initial substitution of SXMPY for the inner products in Step 1 of Algorithm C, but it more than pays for the additional $2n$ operations incurred at Step 3 and exemplifies a technique that might pay off in certain situations.

These techniques have been used quite successfully to increase the performance of EISPACK on various vector and parallel machines. The results of these modifications is reported in full detail in [10]. As a typical example of the performance increase possible with these techniques we offer the following table.

Comparison of EISPACK to Matrix Vector version

	order		
Routine	50	100	Machine
ELMHES	1.5	2.2	CRAY 1
ORTHES	2.5	2.5	CRAY 1
ELMBAK	2.2	2.6	CRAY 1
ORTBAK	3.6	3.3	CRAY 1
TRED1	1.5	1.5	CRAY X-MP-1
TRBAK1	4.2	3.7	CRAY X-MP-1
TRED2	1.6	1.6	CRAY X-MP-1
SVD no/vectors	1.7	2.0	Hitachi S-810/20
SVD with/vectors	1.6	1.7	Hitachi S-810/20
REDUC	1.8	2.2	Fujitsu VP-200
REBAK	4.4	5.8	Fujitsu VP-200

(All versions in Fortran)
(Speedup of matrix vector versions over the EISPACK routines.)

3. Sparsity and Structured Problems

Modules work well for full dense matrix problems, but different techniques may be needed for sparse or special structures. These techniques are likely to be specific to parallel machines and used in algorithms which typically cannot be based on the regular data structures and operations in the modules described above. We give three examples of such algorithms here. These algorithms all have portions that might take advantage of certain vector constructs, but the primary gain in all of them is through the explicit use of parallel computation. In each example there are requirements for synchronization, and in some cases additional computation may be present that would not be needed for the serial algorithm. Nevertheless, all of these have proved to be effective in terms of speed up over the corresponding serial algorithm. One of the algorithms has even provided the startling result of being faster than the corresponding serial code even when it is run on a serial machine.

3.1 Banded Systems

An important structured problem that arises in many applications such as numerical solution of certain PDE problems is the solution of banded systems of linear equations. We consider algorithms for solving narrow-banded diagonally dominant linear systems which are suitable for multiprocessors. Let the linear system under consideration be denoted by

$$Ax = f \tag{3.1.1}$$

where A is a banded diagonally dominant matrix of order n. We assume that the number of superdiagonals $m \ll n$ is equal to the number of subdiagonals. On a sequential machine such a system would be solved via Gaussian elimination without pivoting at a cost of $O(m^2n)$ arithmetic operations. We describe here an algorithm for solving this system on a multiprocessor of p processing units. Each unit may be a sequential machine, a vector machine, or an array of processors. In this paper, however, we consider only p sequential processing units.

Let the system (3.1.1) be partitioned into the block-tridiagonal form shown below

$$\begin{bmatrix} A_1 & B_1 & & & \\ C_2 & A_2 & B_2 & & \\ & \cdot & \cdot & \cdot & \\ & & \cdot & \cdot & \cdot \\ & & C_{p-1} & A_{p-1} & B_{p-1} \\ & & & C_p & A_p \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \cdot \\ \cdot \\ f_{n-1} \\ f_n \end{bmatrix} \tag{3.1.2}$$

where A_i, $1 \le i \le p-1$, is a banded matrix of order $q = \lceil n/p \rceil$ and bandwidth $2m + 1$ (same as A),

$$B_i = \begin{bmatrix} 0 & 0 \\ \hat{B}_i & 0 \end{bmatrix}, \qquad 1 \le i \le p-1 \tag{3.1.3a}$$

and

$$C_{i+1} = \begin{bmatrix} 0 & \hat{C}_{i+1} \\ 0 & 0 \end{bmatrix}, \tag{3.1.3b}$$

in which $\hat{B}_i$ and $\hat{C}_{i+1}$ are lower and upper triangular matrices, respectively, each of order m. The algorithm consists of four stages.

Stage 1

Obtain the LU-factorization

(3.1.4) $$A_i = L_iU_i \qquad 1 \le i \le p$$

using Gaussian elimination without pivoting, one processor per factorization. Here L_i is unit lower triangular and U_i is a nonsingular upper triangular matrix. Note that each A_i is also diagonally dominant.

The cost of this stage is $O(m^2n/p)$ arithmetic operations, no interprocessor communication is required.

Stage 2

If we premultiply both sides of (3.1.2) by

$$diag(A_1^{-1}, A_2^{-1}, \cdots, A_p^{-1})$$

we obtain a system of the form

(3.1.5) $$\begin{bmatrix} I_q & E_1 & & & \\ F_2 & I_q & E_2 & & \\ & \cdot & \cdot & \cdot & \\ & & \cdot & \cdot & \cdot \\ & & F_{p-1} & I_q & E_{p-1} \\ & & & F_p & I_q \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ \cdot \\ \cdot \\ g_{n-1} \\ g_n \end{bmatrix}$$

where

$$E_i = (\hat{E}_i, 0), \quad F_i = (0, \hat{F}_i),$$

in which $\hat{E}_i$ and $\hat{F}_i$ are matrices of m columns given by

$$\hat{E}_i = A_i^{-1} \begin{bmatrix} 0 \\ \hat{B}_i \end{bmatrix}$$

and

$$\hat{F}_i = A_i^{-1} \begin{bmatrix} \hat{C}_i \\ 0 \end{bmatrix}$$

and will in general be full. In other words $\hat{E}_i$, $\hat{F}_i$, and g_i are obtained by solving the linear systems

$$L_jU_j[\hat{F}_j, \hat{E}_j, g_j] = [\begin{bmatrix} 0 \\ \hat{B}_i \end{bmatrix}, \begin{bmatrix} \hat{C}_i \\ 0 \end{bmatrix}, f_i]$$

for$1 \le i \le p$, here $\hat{C}_1 = 0$ and $\hat{B}_p = 0$. Each processor $2 \le k \le p-1$ handles $2m + 1$ linear systems of the form $L_kU_kv = r$, while processors 1 and p each handles $m+1$ linear systems of the same form.

The cost at this stage is $O(m^2n/p)$ arithmetic operations, no interprocessor communications are needed.

Stage 3

Let $\hat{E}_i$ and $\hat{F}_i$ be partitioned, in turn, as follows

$$\hat{F}_i = \begin{bmatrix} P_i \\ M_i \\ Q_i \end{bmatrix}, \text{ and } \hat{E}_i = \begin{bmatrix} S_i \\ N_i \\ T_i \end{bmatrix},$$

where P_i, Q_i, S_i, and $T_i \in R^{m \times m}$. Also, let g_i and x_i be conformally partitioned:

$$g_i = \begin{bmatrix} h_{2i-2} \\ w_i \\ h_{2i-1} \end{bmatrix}, \text{ and } x_i = \begin{bmatrix} y_{2i-2} \\ z_i \\ y_{2i-1} \end{bmatrix}.$$

As an illustration we show the system (3.1.5) for p=3,

$$\left|\begin{array}{ccccccccc}
I_m & & & S_1 & & & & & \\
 & I_v & & N_1 & & & & & \\
 & & I_m & T_1 & & & & & \\
 & & P_2 & I_m & & & S_2 & & \\
 & & M_2 & & I_v & & N_2 & & \\
 & & Q_2 & & & I_m & T_2 & & \\
 & & & & & P_3 & I_m & & \\
 & & & & & M_3 & & I_r & \\
 & & & & & Q_3 & & & I_m
\end{array}\right| \left|\begin{array}{c} y_0 \\ z_1 \\ y_1 \\ y_2 \\ z_2 \\ y_3 \\ y_4 \\ z_3 \\ y_5 \end{array}\right| = \left|\begin{array}{c} h_0 \\ w_1 \\ h_1 \\ h_2 \\ w_2 \\ h_3 \\ h_4 \\ w_3 \\ h_5 \end{array}\right|$$

Observe that the unknown vectors y_1, y_2, y_3, and y_4 (each of order m) are disjoint from the rest of the unknowns. In other words, the m equations above and the m equations below each of the p–1 partitioning lines form an independent system of order $2m(p-1)$, which we shall refer to as the "reduced system" $Ky=h$, which is of the form

$$\left|\begin{array}{cccccccccc}
I_m & T_1 & 0 & 0 & & & & & & \\
P_2 & I_m & 0 & S_2 & & & & & & \\
Q_2 & 0 & I_m & T_2 & 0 & 0 & & & & \\
 & & P_3 & I_m & 0 & S_3 & & & & \\
 & & Q_3 & 0 & I_m & T_3 & & & & \\
 & & & & P_4 & I_m & & & & \\
 & & & & & & & & & \\
 & & & & & & I_m & T_{p-2} & 0 & 0 \\
 & & & & & & P_{p-1} & I_m & 0 & S_{p-1} \\
 & & & & & & Q_{p-1} & 0 & I_m & T_{p-1} \\
 & & & & & & & & P_p & I_m
\end{array}\right| \left\|\begin{array}{c} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ \cdot \\ y_{2p-5} \\ y_{2p-4} \\ y_{2p-3} \\ y_{2p-2} \end{array}\right| = \left|\begin{array}{c} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ \cdot \\ h_{2p-5} \\ h_{2p-4} \\ h_{2p-3} \\ h_{2p-2} \end{array}\right| \qquad (3.1.6)$$

The cost of the algorithm to be used for solving (3.1.6) depends on the interconnection network. Processor 1 contains T_1 and h_1, processor j, $2 \le j \le p-1$, contains P_j, Q_j, S_j, T_j, and h_{2j-2}, h_{2j-1}, and processor p contains P_p, and h_{2p-2}. Hence, if the processors are linearly connected we can only solve (3.1.6) sequentially at the cost of $O(m^3p)$ steps, where a step is the cost of an arithmetic operation or the cost of transmitting a floating-point number from one processor to either of its immediate neighbors. We should add here that since A is diagonally dominant it can be shown that (3.1.6) is also diagonally dominant and hence can be solved via Gaussian elimination without pivoting.

Stage 4

Once the y_i's are obtained, with y_1 in processor 1, y_{2j-2} and y_{2j-1} in processor j $(2 \le j \le p-1)$, and y_{2p-2} in processor p, the rest of the components of the solution vector of (3.1.5) may be computed as follows. Processor k, $1 \le k \le p$, evaluates

$$z_k = w_k - M_k y_{2k-3} - N_k y_{2k} \qquad (3.1.7)$$

with processors 1 and p performing the additional tasks

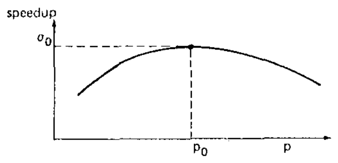
$$y_0 = h_0 - S_1 y_2,$$

and

$$y_{2p-1} = h_{2p-1} - Q_p y_{2p-3}, \qquad (3.1.8)$$

respectively (M_1 and N_p are nonexistent and are taken to be zero in this equation). The cost of this stage is $O(mn/p)$ steps, with no interprocessor communication.

It can be shown that for a linear array of processors, the speedup of this algorithm over the classical sequential algorithm behaves as shown in the figure below



where p_0 and σ_0 are $O(\sqrt{n/m})$. Stage 2 dominates the computation until p_0, then the communication costs impact the performance and Stage 3 has a greater influence.

3.2 QR - Factorization of a Sparse Matrix

The version of Householder's method for the QR-factorization of a dense matrix given in Section 4.2 is very well suited to vector and parallel-vector architectures. However, for parallel processors without vector capabilities this may not be the algorithm of choice. An alternative is to use a parallel version of Givens' method. There are many papers on this subject especially within the study of systolic arrays of processors[13,14,28]. Here we present a variant of these techniques that is suitable for parallel processors with far more computing power in a single processor than considered in the systolic array case. This method was first presented in [6] and it called the Pipelined Givens Method. The Pipelined Givens method is well suited to the architecture and synchronization mechanism of the Denelcor HEP computer. However, any parallel computer with globally shared memory and a relatively inexpensive synchronization primitive could take advantage of this method.

The reasons this algorithm is more successful than Householder's method on a such a parallel computer are twofold. As demonstrated by the computational results presented in [6], memory references play a far more important role in determining algorithm performance on a parallel machine such as the HEP than they do on serial machines. The Givens algorithm requires half as many array references as the Householder method. In addition, the Pipelined Givens method offers a greater opportunity to keep many (virtual) processors busy because it does not employ a fork-join synchronization mechanism and does not have the inherent serial bottlenecks present in the Householder method. Moreover, there is an opportunity to adjust the level of granularity through the specification of a certain parameter (discussed below) in order to mask synchronization costs with computation.

The serial variant of Givens' method that we consider is as follows. Given a real $m \times n$ matrix A, the goal of the algorithm is to apply elementary plane rotations G_{ij} that are constructed to annihilate the ji-th element of the matrix A. Such a matrix may be thought of as a

2x2 orthogonal matrix of the form

$$G = \begin{bmatrix} \gamma & \sigma \\ -\sigma & \gamma \end{bmatrix},$$

where $\sigma^2 + \gamma^2 = 1$. If

$$\begin{bmatrix} \alpha & a^T \\ \beta & b^T \end{bmatrix}$$

represents a $2 \times n$ matrix, then, with proper choice of γ and σ, a zero can be introduced into the β position with left multiplication by G. When embedded in the $n \times n$ identity, the matrix G_{ij} is of the form

$$G_{ij} = I + D_{ij} ,$$

where all elements of D_{ij} are zero except possibly the *ii*, *ij*, *ji*, and *jj* entries. The matrices G_{ij} are used to reduce A to upper triangular form in the following order:

$$\begin{bmatrix} G_{n,m} \cdots G_{2m} G_{1m} \end{bmatrix} \cdots \begin{bmatrix} G_{n-1,n} \cdots G_{2n} G_{1n} \end{bmatrix} \begin{bmatrix} G_{n-2,n-1} \cdots G_{2,n-1} G_{1,n-1} \end{bmatrix} \cdots \begin{bmatrix} G_{12} \end{bmatrix} A = \begin{bmatrix} R \\ 0 \end{bmatrix} .$$

The order of the zeroing pattern may be seen in the 6x5 example:

$$\begin{array}{ccccc}
\times & \times & \times & \times & \times \\
\otimes_1 & \times & \times & \times & \times \\
\otimes_2 & \otimes_3 & \times & \times & \times \\
\otimes_4 & \otimes_5 & \otimes_6 & \times & \times \\
\otimes_7 & \otimes_8 & \otimes_9 & \otimes_{10} & \times \\
\otimes_{11} & \otimes_{12} & \otimes_{13} & \otimes_{14} & \otimes_{15}
\end{array}$$

Figure 3.2.1. Zeroing Pattern of Givens Method

In Figure 3.2.1 the symbol $\times$ denotes a nonzero entry of the matrix and the symbol $\otimes_k$ means that entry is zeroed out by the *k*-th transformation. This order is important if one wishes to "pipeline" the row reduction process. Pipelining may be achieved by expressing R as a linear array in packed form by rows and then dividing this linear array into equal-length pipeline segments. A process is responsible for claiming an unreduced row of the original matrix and reducing it to zero by combining it with the existing R matrix using Givens transformations. A new row may enter the pipe immediately after the row ahead has been processed in the first segment. Each row proceeds one behind the other until the entire matrix has been processed. However, these rows cannot be allowed to get out of order, once they have entered the pipe, because of data dependencies. The synchronization required to preserve this order is accomplished using an array of locks, with each entry of the array protecting access to a segment of the pipe. A process gains access to the next segment by locking the corresponding entry of the lock array before unlocking the entry protecting the segment it currently occupies. Granularity may be adjusted to hide the cost of this synchronization by simply altering the length of a segment. Segment boundaries do not correspond to row boundaries in R. This feature has the advantage of balancing the amount of work between synchronization points but the disadvantage of having to decide on one of two possible computations at each location within a segment: compute a transformation or apply one.

The method is more easily grasped if one considers the following three diagrams. In Figure 3.2.2 we represent the matrix A in a partially decomposed state. The upper triangle of the array contains the current state of the triangular matrix R. The entries ($\alpha\ \alpha\ \alpha\ \alpha\ \alpha$) and the entries ($\beta\ \beta\ \beta\ \beta\ \beta$) represent the nonzero components of the next two rows of A that must be reduced.

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & & \times & \times \\ & & & & \times \\ \\ \\ \alpha & \alpha & \alpha & \alpha & \alpha \\ \beta & \beta & \beta & \beta & \beta \end{bmatrix}$$

Figure 3.2.2. Partially Reduced Matrix

A natural way to pipeline this reduction process is shown in Figure 3.2.3. There we see the row ($\alpha\ \alpha\ \alpha\ \alpha\ \alpha$) being passed through the triangle R during the reduction process, with the row ($\beta\ \beta\ \beta\ \beta\ \beta$) flowing immediately behind it. The position of β-row and the α-row interleaved within the rows of R is meant to indicate that they are ready to be combined with the first and second rows of R respectively. The first entry of the α-row has been zeroed by computing and applying the appropriate Givens transformation as described above, and we are ready to zero out the second entry. In a serial algorithm this α-row would be completely reduced to zero before beginning to reduce the β-row. However, this process may be pipelined by beginning to combine the β-row with the first row of R as soon as the α-row is ready to be combined with the second row of R. Since the first row of R is modified during the introduction of a zero in the first position of the α-row, it is important that the processing of the β-row be suitably synchronized with the processing of the α-row. In practice, after initial startup, there would be n rows in the pipe throughout the course of the computation.

$$\begin{array}{ccccc}
\times & \times & \times & \times & \times \\
\beta & \beta & \beta & \beta & \beta \\
 & \times & \times & \times & \times \\
\alpha & \alpha & \alpha & \alpha & \alpha \\
 & & \times & \times & \times \\
 & & & \times & \times \\
 & & & & \times
\end{array}$$

Figure 3.2.3. Pipelined Row Reduction

A disadvantage suffered by the scheme we have just described is that the granularity becomes finer as the process advances because the length of the nonzero entries in a row of R decreases. A better load balance and a natural way to adjust the granularity may be achieved by considering the matrix R as a linear array divided into segments of equal length.

$$(\rho_{11}\ \rho_{12}\ \rho_{13} | \rho_{14}\ \rho_{15}\ \rho_{22} | \rho_{23}\ \rho_{24}\ \rho_{25} | \rho_{33}\ \rho_{34}\ \rho_{35} | \rho_{44}\ \rho_{45}\ \rho_{55})$$

Figure 3.2.4. R as a Segmented Pipe

In Figure 3.2.4 we depict the nonzero elements of R as ρ_{ij} and note that in this linear array the natural row boundaries occur at entries ρ_{i5}. The length of a segment is 3 in this example and we denote pipe segment boundaries with |. In general we specify the number of segments desired and then the length of a segment is

$$\left\lceil \frac{n(n+1)/2}{\textit{number of segments}} \right\rceil .$$

The number of segments is an adjustable parameter in the program. The α-row and β-row are represented as in Figure 3.2.3 with the α-row entering the second segment and the β-row entering the first segment. The difference between this scheme and the one depicted in Figure 3.2.3 is that the α-row is not fully combined with the first row of R before processing of the β-row is begun. To keep the rows in order, a row must gain entry to the next segment before releasing the current segment. If the number of segments is equal to the number of nonzero elements of R, then this algorithm reduces to a variant of the more traditional dataflow algorithm presented in [13,14,28]. Computational experience reported in [6] indicates that performance is not extremely sensitive to this parameter. The optimal length of a segment appeared to be around n, but performance degraded noticeably only with extremely large or extremely small segment lengths.

We now turn to the main point of interest in this discussion, the case when the matrix A is large and sparse. The algorithm we present was developed by Heath and Sorensen [21] as an generalization of the Pipelined Givens method to the sparse case. Specifically, we assume that the matrix

$$A^TA$$

is suitably sparse. In this case there are well-established techniques [17] for determining a permutation matrix P such that

$$P^TA^TAP = R^TR$$

has a sparse Cholesky factor R. This permutation is obtained from the symbolic nonzero structure of the matrix A and is designed to reduce the number of nonzeros in the factor R as much as possible. It is of considerable interest to parallelize this symbolic step of the factorization procedure, but for this discussion we have concentrated only on parallelizing the numerical portion of the algorithm, which consists of applying Givens transformations to the matrix AP to produce R.

$$R = \begin{bmatrix} \rho_{11} & \rho_{12} & & \rho_{14} & & & \\ & \rho_{22} & & \rho_{24} & & & \\ & & \rho_{33} & & \rho_{35} & \rho_{36} & \\ & & & \rho_{44} & \rho_{45} & & \\ & & & & \rho_{55} & \rho_{56} & \rho_{57} \\ & & & & & \rho_{66} & \rho_{67} \\ & & & & & & \rho_{77} \end{bmatrix}$$

DIAG	ρ_{11}	ρ_{22}	ρ_{33}	ρ_{44}	ρ_{55}	ρ_{66}	ρ_{77}		
RNZ	ρ_{12}	ρ_{14}	ρ_{24}	ρ_{35}	ρ_{36}	ρ_{45}	ρ_{56}	ρ_{57}	ρ_{67}
XRNZ	1	3	4	6	7	9	10		
NZSUB	2	4	5	6	5	6	7		
XNZSUB	1	2	3	5	6	6			

Figure 3.2.5. Sparse Data Structure of R

The algorithm is virtually identical to the serial algorithm. There are some notable exceptions, however, an explanation of which requires an understanding of the data structure for R as illustrated in Figure 3.2.5. The *RNZ* array contains the off-diagonal nonzero entries of R in packed form. It is evident that the *RNZ* array lends itself to the same segmentation and that the row reduction process may be pipelined in almost exactly the same way as the R array in the dense case. The natural row boundaries are determined by the array *XRNZ*. The i-th entry of this array points to the location of the first nonzero in the i-th row of the full array R. The arrays *NZSUB* and *XNZSUB* are used to determine the column indices of entries in *RNZ* as described in [15]. The *RNZ* array is divided into equal length segments as shown in Figure 3.2.6.

RNZ ρ_{12} ρ_{14} | ρ_{24} ρ_{35} | ρ_{36} ρ_{45} | ρ_{56} ρ_{57} | ρ_{67} |

Figure 3.2.6. *RNZ* as a Segmented Pipe

Just as in the dense case, a process is responsible for claiming a row and then combining it with the current *R* array using Givens transformations. These processes synchronize as before: The first nonzero of the unreduced row is determined, the location of the segment containing the corresponding row boundary in *RNZ* is determined, entry is gained to that segment (by reading an asynchronous variable on the HEP), and then the row reduction is started. To preserve the correctness of the factorization, once the pipeline has been entered by a process, it must stay in proper order. A process keeps itself in proper order by gaining access to the next segment before releasing the segment it currently owns. In the dense case, every process has work to do in every segment. In the sparse case, however, there may be segments where no work is required because the sparsity pattern of the row currently being reduced allows it to skip several rows of *R* . This phenomenon is best understood when illustrated by example. Consider a row which has the initial nonzero structure

$$a = (\ 0\ \alpha\ 0\ \alpha\ 0\ 0\ 0\ ),$$

and suppose this row is to be reduced to zero against the nonzero *R* structure shown in Figure 3.2.5 with *RNZ* segmented as shown in Figure 3.2.6. The first nonzero of the row a is in position 2, so it is first combined with row number 2. This row starts at position 3, as indicated by the second entry of *XRNZ*, and position 3 is in segment number 2 in *RNZ*. The diagonal entry ρ_{22} is used together with the first nonzero in a to compute the Givens transformation, and then this transformation is applied to element ρ_{24} together with the entry in the 4-th position of a. No fill is created in a, so after the application there is one nonzero at position 4. This means that row 3 may be skipped. Row 4 begins in the 6-th position of *RNZ*, which is in segment 3. Entry is gained to segment 3, and then segment 2 is released and the factorization proceeds. In this example the next row boundary required happened to be in the adjacent segment. In general, however, there might be several segments between the relevant row boundaries. In that case, entry into each of the intervening segments must be gained and released to ensure that the proper order is maintained between the various rows being processed.

Computational results reported in [21] show that this scheme achieves near perfect speedup on typical problems such as those found in [16,20]. It has the advantage of using existing data structures that are found in SPARSPACK and thus does not require modification of the user interface in existing codes that rely on this package. Such routines can take advantage of this speedup without modification.

3.3 Eigensystems of Tridiagonal Matrices

The final problem we consider is that of determining the eigensystem of a real $n{\times}n$ symmetric matrix A, find all of the eigenvalues and corresponding eigenvectors of A. It is well known [30,32] that under these assumptions

(3.3.1) $$A = QDQ^T \quad , \quad \text{with } Q^TQ = I \, ,$$

so that the columns of the matrix Q are the orthonormal eigenvectors of A and $D = diag(\delta_1, \delta_2, \ldots, \delta_n)$ is the diagonal matrix of eigenvalues. The standard algorithm for computing this decomposition is to first use a finite algorithm to reduce A to tridiagonal form using a sequence of Householder transformations, and then to apply a version of the QR-algorithm to obtain all the eigenvalues and eigenvectors of the tridiagonal matrix[12,29,30,32]. We have already discussed a method for parallelizing the initial reduction to tridiagonal form in Section 4.3. Now, we shall describe a method for parallelizing the computation of the eigensystem of the tridiagonal matrix.

The method is based upon a divide and conquer algorithm suggested by Cuppen[3]. A fundamental tool used to implement this algorithm is a method that was developed by Bunch, Nielsen, and Sorensen[2] for updating the eigensystem of a symmetric matrix after modification by a rank one change. This rank-one updating method was inspired by some earlier work of Golub[19] on modified eigenvalue problems. The basic idea of the new method is to use rank-one modifications to tear out selected off-diagonal elements of the tridiagonal problem in order to introduce a number of independent subproblems of smaller size. The subproblems are solved at the lowest level using the subroutine TQL2 from EISPACK and then results of these problems are successively glued together using the rank-one modification routine SESUPD that we have developed based upon the ideas presented in [2].

In the following discussion we describe the partitioning of the tridiagonal problem into smaller problems by rank-one tearing. Then we describe the numerical algorithm for gluing the results back together. The organization of the parallel algorithm is laid out, and finally some preliminary computational results are presented.

Partitioning by Rank-One Tearing

The crux of the algorithm is to divide a given problem into two smaller subproblems. To do this, we consider the symmetric tridiagonal matrix

(3.3.2) $$T = \begin{bmatrix} T_1 & \beta e_k e_1^T \\ \beta e_1 e_k^T & T_2 \end{bmatrix}$$
$$= \begin{bmatrix} \hat{T}_1 & 0 \\ 0 & \hat{T}_2 \end{bmatrix} + \beta \begin{bmatrix} e_k \\ e_1 \end{bmatrix} (e_k^T \, , \, e_1^T)$$

where $1 \le k \le n$ and e_j represents the j–th unit vector of appropriate dimension. The k–th

diagonal element of T_1 has been modified to give $\hat{T}_1$ and the first diagonal element of T_2 has been modified to give $\hat{T}_2$. There are some numerical concerns here concerning possible cancellation. A way to overcome these difficulties is discussed fully in [5].

Now we have two smaller tridiagonal eigenvalue problems to solve. According to equation (3.3.2) we compute the two eigensystems

$$\hat{T}_1 = Q_1 D_1 Q_1^T , \quad \hat{T}_2 = Q_2 D_2 Q_2^T .$$

This gives

$$(3.3.3) \quad T = \begin{bmatrix} Q_1 D_1 Q_1^T & 0 \\ 0 & Q_2 D_2 Q_2^T \end{bmatrix} + \beta \begin{bmatrix} e_k \\ e_1 \end{bmatrix} (e_k^T , e_1^T)$$

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

where $q_1 = Q_1^T e_k$ and $q_2 = Q_2^T e_1$. The problem at hand now is to compute the eigensystem of the interior matrix in equation (3.3.3). A numerical method for solving this problem has been provided in [2] and we shall discuss this method in the next section.

It should be fairly obvious how to proceed from here to exploit parallelism. One simply repeats the tearing on each of the two halves recursively until the original problem has been divided into the desired number of subproblems and then the rank one modification routine may be applied from bottom up to glue the results together again.

The Updating Problem

The general problem we are required to solve is that of computing the eigensystem of a matrix of the form

$$(3.3.4) \qquad \hat{Q}\hat{D}\hat{Q}^T = D + \rho z z^T$$

where D is a real $n \times n$ diagonal matrix, ρ is a scalar, and z is a real vector of order n . It is assumed without loss of generality that z has Euclidian norm 1.

As shown in [2], if $D = diag(\delta_1, \delta_2, \cdots, \delta_n)$ with $\delta_1 < \delta_2 < \cdots < \delta_n$ and no component ζ_i of the vector z is zero, then the updated eigenvalues $\hat{\delta}_i$ are roots of the equation

$$(3.3.5) \qquad f(\lambda) \equiv 1 + \rho \sum_{j=1}^{n} \frac{\zeta_j^2}{\delta_j - \lambda} = 0.$$

Golub[19] refers to this as the secular equation and the behavior of its roots is completely described by the following graph:

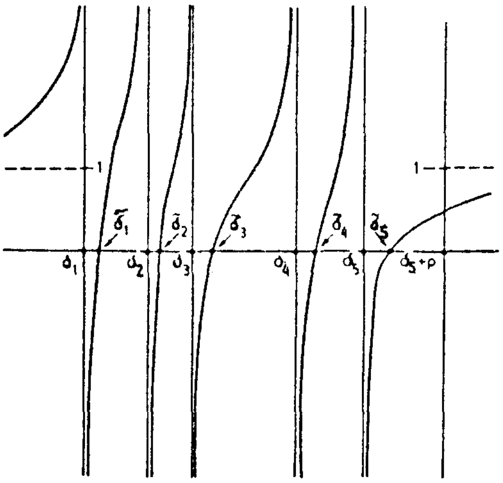


Figure 3.3.1. The Secular Equation

Moreover, as shown in [2] the eigenvectors (i.e. the columns of $\hat{Q}$ in (3.3.4)) are given by the formula

(3.3.6) $$\hat{q}_i = \gamma_i \Delta_i^{-1} z$$

with γ_i chosen to make $\|\hat{q}_i\| = 1$, and with $\Delta_i = diag(\delta_1 - \hat{\delta}_i, \delta_2 - \hat{\delta}_i, \cdots, \delta_n - \hat{\delta}_i)$. Due to this structure, an excellent numerical method may be devised to find the roots of the secular equation and as a by-product to compute the eigenvectors to full accuracy.

In the following discussion we assume that $\rho > 0$ in (3.3.5). A simple change of variables may always be used to achieve this, so there is no loss of generality. The method we shall describe was inspired by the work of More' [25] and Reinsch[26,27], and relies on the use of simple rational approximations to construct an iterative method for the solution of equation (3.2). Given that we wish to find the *i-th* root $\hat{\delta}_i$ of the function f in (3.2) we may write this function as

$$f(\lambda) = 1 + \phi(\lambda) + \psi(\lambda)$$

where

$$\psi(\lambda) = \rho \sum_{j=1}^{i} \frac{\zeta_j^2}{\delta_j - \lambda}$$

and

$$\phi(\lambda) \equiv \rho \sum_{j=i+1}^{n} \frac{\zeta_j^2}{\delta_j - \lambda}.$$

From the graph in Figure 3.3.1 it is seen that the root $\hat{\delta}_i$ lies in the open interval (δ_i, δ_{i+1}) and for λ in this interval all of the terms of ψ are negative and all of the terms of ϕ are positive. We may derive an iterative method for solving the equation

$$-\psi(\lambda) = 1 + \phi(\lambda)$$

by starting with an initial guess λ_0 in the appropriate interval and then constructing simple rational interpolants of the form

$$\frac{p}{q - \lambda}, \quad r + \frac{s}{\delta - \lambda}$$

where the parameters p, q, r, s are defined by the interpolation conditions

(3.3.7)
$$\frac{p}{q - \lambda_0} = \psi(\lambda_0), \quad r + \frac{s}{\delta - \lambda_0} = \phi(\lambda_0)$$

$$\frac{p}{(q - \lambda_0)^2} = \psi'(\lambda_0), \quad \frac{s}{(\delta - \lambda_0)^2} = \phi'(\lambda_0).$$

The new approximate λ_1 to the root $\hat{\delta}_i$ is then found by solving

(3.3.8)
$$\frac{-p}{q - \lambda} = 1 + r + \frac{s}{\delta - \lambda}$$

It is possible to construct an initial guess which lies in the open interval $(\delta_i, \hat{\delta}_i)$. A sequence of iterates $\{\lambda_k\}$ may then be constructed as we have just described with λ_{k+1} being derived from λ_k as λ_1 was derived from λ_0 above. It is proved in [3] that this sequence of iterates converges quadratically from one side of the root and does not need any safeguarding.

During the course of this iteration, the quantities $\delta_j - \lambda_k$ are maintained and the iterative corrections to λ_k are added to these differences directly. As the iteration converges the lower order bits of these quantities are corrected to full accuracy. Since these differences make up the diagonal entries of the matrix Δ_i appearing in (3.3.6), this allows computation of the updated eigenvectors to full accuracy and avoids cancellation that would occur if we first computed the roots and then formed the differences.

Another important numerical aspect of the updating problem is "deflation". There are two cases where such deflation occurs. One when two given roots are nearly equal and the other when certain components of the vector z are "small". The effects of such deflation can be dramatic, for the amount of computation required to perform the updating is greatly reduced. We shall not present the details nor the numerical motivation for deflation here. We just remark that the result of deflation is to replace the updating problem (3.3.4) with one of smaller size. This is accomplished when appropriate by applying similarity transformations consisting of several Givens transformations. If G represents the product of these transformations the result is

$$G(D + \rho zz^T)G^T = \begin{bmatrix} D_1 - \rho z_1 z_1^T & 0 \\ 0 & D_2 \end{bmatrix} + E,$$

where

$$\| E \| \leq \varepsilon \| D + \rho zz^T \|$$

with ε roughly the size of machine precision. The cumulative effect of such errors is additive and thus the final computed eigensystem $\hat{Q}\hat{D}\hat{Q}^T$ which satisfies

$$\| A - \hat{Q}\hat{D}\hat{Q}^T \| \leq \varepsilon\eta \| A \|$$

where η is order 1 in magnitude. The reduction in size of $D_1 - \rho z_1 z_1^T$ over the original rank 1 modification can be spectacular in certain cases. The details of deflation and more numerical results may be found in [5]. We shall indicate the potential in the following table. In this table we report the results of this algorithm on a tridiagonal matrix with pseudo-random nonzero entries in the interval [-1,1]. The table entries show ratios of execution time required by TQL2 (from EISPACK) to that required by the parallel algorithm on the same machine with the same compiler options and the same environment. In all cases the time reported by TQL2 was obtained by executing it as a single process. It should be emphasized that in all cases the computations were carried out as though the tridiagonal matrix had come from Householder's reduction of a dense symmetric matrix to tridiagonal form. The identity was passed in place of the orthogonal basis that would have been provided by this reduction, but the arithmetic operations performed were the same as those that would have been required to transform that basis into the eigenvectors of the original symmetric matrix.

	VAX 785/FPA	Denelcor HEP	Alliant FX/8	CRAY X-MP-1	CRAY X-MP-4
random matrix	2.6	12	15.2	1.8	4.5

order = 150

Ratio of execution time $\dfrac{\textit{TQL2 time}}{\textit{parallel time}}$

These results are remarkable because in all cases speedups greater than the number of physical processors were obtained. The gain is due to the numerical properties of the deflation portion of the parallel algorithm. In all cases the word length was 64 bits and the same level of accuracy was achieved by both methods. The measurement of accuracy used was the maximum 2-norm of the residuals $Tq - \lambda q$ and of the columns of $Q^TQ - I$. The results are typical of the performance of this algorithm on random problems with speedups becoming more dramatic as the matrix order increases. In problems of order 500 speedups of 15 have been observed on the CRAY-XMP-4 and speedups over 50 have been observed on the Alliant FX/8 which are 4 and 8 processor machines respectively. The CRAY results can actually be improved because parallelism at the root finding level was not exploited in the implementation run on the CRAY but was fully exploited on the Alliant. Finally, we remark that deflation does not occur for all matrices and examples of this are given in [5].

The Parallel Algorithm

Although it is fairly straightforward to see how to obtain a parallel algorithm, certain details are worth discussing further. We shall begin by describing the partitioning phase. This phase amounts to constructing a binary tree with each node representing a rank-one tear and hence a partition into two sub-problems. A tree of level 3 therefore represents a splitting of the original problem into 8 smaller eigenvalue problems. Thus, there are two standard symmetric tridiagonal eigenvalue problems to be solved at each leaf of the tree. Each of these problems may be spawned independently without fear of data conflicts. The tree is then traversed in reverse order with the eigenvalue updating routine SESUPD applied at each node joining the results from the left son and right son calculations. The leaves each define independent rank-one updating problems and again there is no data conflicts between them. The only data dependency at a node is that the left and right son calculations must have been completed. As this condition is satisfied, the results of two adjacent eigenvalue subproblems are ready to be joined through the rank-one updating process and this node may spawn the updating process immediately. Information required at a node to define the problem consists of the index of the element torn out together with the dimension of the left and right son problems. For example, if $n = 50$ with a tree of level 3 we have

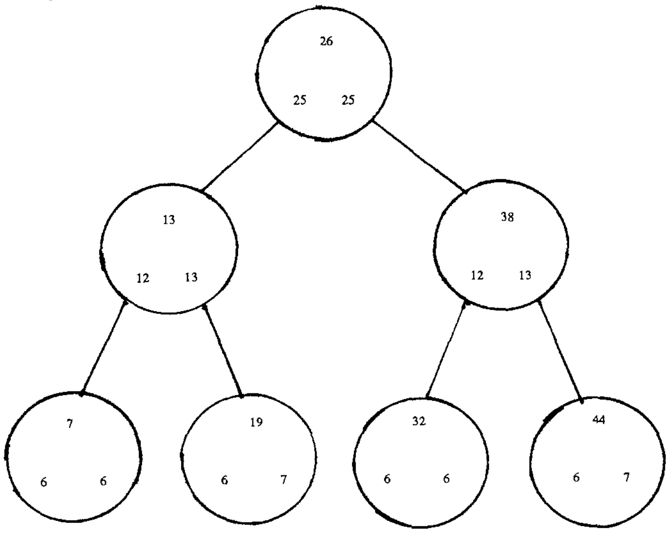


Figure 2. The Computational Tree

This tree defines 8 subproblems at the lowest level. The beginning indices of these problems are 1,7,13,19,26,32,38,44 and the dimension of each of them may be read off from left to right at the lowest level as 6,6,6,7,6,6,6,7 respectively. As soon as the calculation for the problems beginning at indices 1 and 7 have been completed a rank-one update may proceed on the problem beginning at index 1 with dimension 12. The remaining updating problems at this level begin at indices 13,26,38. There are then two updating problems at indices 1 and 26 each of dimension 25 and a final updating problem at index 1 of dimension 50.

Evidently, we lose a degree of large grain parallelism as we move up the tree. However, there is more parallelism to be found at the root finding level and the amount of this increases as we travel up the tree so there is ample opportunity for load balancing in this scheme. The parallelism at the root finding level stems from the fact that each of the root calculations is independent and requires read only access to all but one array. That is the array that contains the diagonal entries of the matrix Δ_i described above. For computational efficiency we may decide on an advantageous number of processes to create at the outset. In the example above that number was 8. Then as we travel up the tree the root-finding procedure is split into 2,4,and finally 8 parallel parts in each node at level 3, 2, 1 respectively. As these computations are roughly equivalent in complexity on a given level it is reasonable to expect to keep all processors devoted to this computation busy throughout.

4. Implementation and Library Issues

The notion of introducing parallelism at the level of the modules as presented in Section 4 presents an unpleasant situation. All of the algorithms presented here are properly considered low level library subroutines when taken in the context of a large scale applications code. If properly designed, such codes rely upon software libraries to perform calculations of the type discussed here. When designing a library, one wishes to conceal machine dependencies as much as possible from the user. Also, in the case of transporting existing libraries to new machines, one wishes to preserve user interfaces in order to avoid unnecessary modification of existing code that references library subroutines. These important considerations seem to be difficult to accommodate if we are to invoke parallelism at the level described above. It would appear that the user must be conscious of the number of parallel processes required by the library subroutines throughout his program. This is the result of physical limitations on the total number of processes allowed to be be created. Should the library routines be called from multiple branches of a parallel program, the user could inadvertently attempt to create many more processes than is allowed.

A second issue arises within the context of merely programming the more explicitly parallel algorithms discussed in Section 3. These algorithms present far more challenging synchronization requirements than the simple fork-join construct used to implement the modules on a parallel machine. How can these routines be coded in a transportable way?

A possible solution that will have impact on both situations has been inspired by work of Lusk and Overbeek on methodology for implementing transportable parallel codes. We have adapted the "pool of problems " approach they present in [23,24] to the problem of constructing and implementing transportable software libraries. We use a package called SCHEDULE that we have been developing during the period that the algorithms presented within this paper were being devised and tested. SCHEDULE is a package of Fortran subroutines designed to aid in programming explicitly parallel algorithms for numerical calculations. The design goal of SCHEDULE is to aid a programmer familiar with a Fortran programming environment to implement a parallel algorithm in a style of Fortran programming that will lend itself to transporting the resulting program across a wide variety of parallel machines. The approach relies upon the user adopting a particular style of expressing a parallel program. Once this has been done the subroutines and data structure provided by SCHEDULE will allow implementation of the parallel program without dependence on specific machine intrinsics. The user will be required to fully understand the data dependencies, parallel structure and shared memory requirements of the program.

The basic philosophy taken here is that Fortran programs are naturally broken into subroutines which identify units of computation that are self-contained and which operate on shared data structures. Typically these data structures are rectangular arrays and the portion of the data structure to be operated on is often identified by passing an element of the array that is treated within the subroutine as the first element of the array to be operated on. This standard technique is extremely useful in implementing a parallel algorithm in the style adopted in SCHEDULE. Morever, it allows one to call upon existing library subroutines without any modification, and without having to write an envelope around the library subroutine call in order to conform to some unusual data passing conventions imposed by a given parallel programming environment. One defines a shared data structure and subroutines to operate on this data structure. Then a parallel(izable) program is written in terms of calls to these subroutines which in principle may either be performed independently or according to data dependency requirements which the user is responsible for defining. Once this has been done the result is a serial program that could run in parallel if there was a way to schedule the units of computation on a system of parallel processors while obeying the data dependencies.

SCHEDULE works in a manner similar to an operating system to schedule processes that are ready to execute. It consists of two queues: a process queue and a ready queue. A process identifies a subroutine call and pointers to addresses needed to make the call. A process tag is placed upon the ready queue when its data dependencies have been satisfied. In addition to this Work routines are constructed which are capable of assuming the identity of any process that will appear on the queue. A fixed number of these routines are devoted to the library. They are activated (created, forked, etc.) at the outset of the computation and remain activated throughout the course of this computation. Within this scheme calls to matrix vector routines (for example) are not made explicitly, they are instead put on the process queue to be performed as soon as they can be picked up by one of the workers through the scheduler mechanism. Transportability is achieved because the actual references to machine specific synchronization primitives are isolated in two low level SCHEDULE and are very few in number. This together with the specific means for creating or forking processes are the only things that need to be changed when moving from one machine to another. A schematic of the abstract idea behind the scheduler is represented in Figure 6.2 below

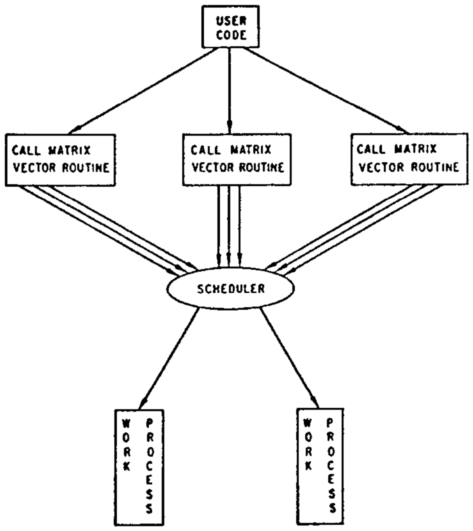

USER CODE
CALL MATRIX VECTOR ROUTINE
CALL MATRIX VECTOR ROUTINE
CALL MATRIX VECTOR ROUTINE
SCHEDULER
WORK PROCESS
WORK PROCESS

Figure 6.2. Library Scheduler

5. Conclusions

This has been a sampling of some of the ideas for algorithms and implementation techniques that we have been considering recently in the Mathematics and Computer Science Division at Argonne National Laboratory. Traditionally, our activities have involved the development of algorithms and techniques for implementing these algorithms in a transportable manner. We view the work presented here as an extension of this theme that will aid us in addressing similar problems that are arising with the advent of exotic computer architectures. We find the subject challenging and rewarding in terms of its potential. We encourage others to join us in pursuing the means to provide useful methodologies and software techniques to enable us to make effective use of the developing hardware.

6. References

[1] R. G. Babb II, *Parallel Processing with Large Grain Data Flow Techniques,* IEEE Computer, Vol. 17, No. 7 , pp. 55-61, July (1984).

[2] J.R. Bunch, C.P. Nielsen, and D.C. Sorensen, *Rank-One Modification of the Symmetric Eigenproblem,* Numerische Mathematik 31, pp. 31-48, (1978).

[3] J.J.M. Cuppen *A Divide and Conquer Method for the Symmetric Tridiagonal Eigenproblem,* Numerische Mathematik 36, pp. 177-195, (1981).

[4] J.J. Dongarra and S.C. Eisenstat, *Squeezing the Most out of an Algorithm in CRAY Fortran,* ACM Trans. Math. Software, Vol. 10, No. 3, (1984).

[5] J.J. Dongarra and D.C. Sorensen, *A Fully Parallel Algorithm for the Symmetric Eigenvalue Problem,* Argonne National Laboratory Report ANL/MCS-TM-62, (January 1986),

[6] J.J. Dongarra, A.H. Sameh, and D.C. Sorensen, "Some Implementations of the QR Factorization on an MIMD Machine", Argonne National Laboratory Report ANL/MCS-TM-25, (October 1984), to appear in Parallel Computing.

[7] J.J. Dongarra, *Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment,* Argonne National Laboratory Report MCS-TM-23, (updated August 1984).

[8] J.J. Dongarra, J.R. Bunch, C.B. Moler, and G.W. Stewart, *LINPACK Users' Guide,* SIAM Publications, Philadelphia, (1979).

[9] J.J. Dongarra, J. Du Croz, S. Hammarling, R.J. Hanson, *A Proposal for an Extended Set of Fortran Basic Linear Algebra Subroutines,* Argonne National Laboratory Report MCS/TM 41, Revision 1 (October, 1985).

[10] J.J. Dongarra, L. Kaufman, and S. Hammarling *Squeezing the Most out of Eigenvalue Solvers on High-Performance Computers,* Argonne National Laboratory Report ANL MCS-TM 46, (January 1985), to appear in Linear Algebra and Its Applications.

[11] K. Fong and T.L. Jordan, *Some Linear Algebra Algorithms and Their Performance on CRAY-1,* Los Alamos Scientific Laboratory, UC-32, (June 1977).

[12] B.S. Garbow, J.M. Boyle, J.J. Dongarra, and C.B. Moler, *Matrix Eigensystem Routines - EISPACK Guide Extension,* Lecture Notes in Computer Science, Vol. 51, Springer-Verlag, Berlin, (1977).

[13] W. Gentleman, *Error Analysis of the QR Decomposition by Givens Transformations,* Linear Algebra and Its Applications, Vol. 10, pp. 189-197, (1975).

[14] M. Gentleman and H. T. Kung, *Matrix Triangularization by Systolic Arrays,* in Proceedings SPIE 298 Real-Time Signal Processing IV, San Diego, California, (1981).

[15] A. George and M. T. Heath, *Solution of Sparse Linear Least Squares Problems Using Givens Rotations,* Linear Algebra and Its Applications, Vol. 34, pp. 69-83, (1980).

[16] J. A. George, M. T. Heath and R. J. Plemmons, *Solution of Large-Scale Sparse Least Squares Problems Using Auxiliary Storage,* SIAM Journal on Scientific and Statistical Computing, Vol. 2, pp. 416-429, (1981).

[17] A. George and J. Liu, *Computer Solution of Large Sparse Positive Definite Systems,* Prentice-Hall, Englewood Cliffs, New Jersey, (1981).

[18] W. Givens, *Numerical Computation of the Characteristic Values of a Real Symmetric Matrix,* Oak Ridge National Laboratory Report ORNL-1574, Oak Ridge, Tenn., (1954).

[19] G.H. Golub, *Some Modified Matrix Eigenvalue Problems,* SIAM Review, 15, pp. 318-334 (1973).

[20] G. H. Golub and R. J. Plemmons, *Large Scale Geodetic Least Squares Adjustments by Dissection and Orthogonal Decomposition,* Linear Algebra and Its Applications, Vol. 34, pp. 3-28, (1980).

[21] M. T. Heath and D. C. Sorensen, *A Pipelined Givens Method for Computing the QR-Factorization of a Sparse Matrix* Argonne National Laboratory Report ANL MCS-TM 47, (February 1985). Linear Algebra and Its Applications, (to appear).

[22] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, *Basic Linear Algebra Subprograms for Fortran Usage. ACM Trans. Math. Software,* 5 (1979), 308-371.

[23] E. Lusk and R. Overbeek, "Implementation of Monitors with Macros: A Programming Aid for the HEP and Other Parallel Processors", ANL-83-97, (1983).

[24] E. Lusk and R. Overbeek, "An Approach to Programming Multiprocessing Algorithms on the Denelcor HEP", ANL-83-96, (1983).

[25] J.J. More' *The Levenberg-Marquardt Algorithm: Implementation and Theory,* Proceedings of the Dundee Conference on Numerical Analysis, G.A. Watson ed. Springer-Verlag (1978).

[26] C.H. Reinsh, *Smoothing by Spline Functions,* Numerische Mathematik 10, pp. 177-183, (1967).

[27] C.H. Reinsh, *Smoothing by Spline Functions II,* Numerische Mathematik 16, pp. 451-454, (1971).

[28] A. Sameh and D. Kuck, *On Stable Parallel Linear System Solvers,* Journal of the ACM, Vol. 25, pp. 81-91, (1978).

[29] B.T. Smith, J.M. Boyle, J.J. Dongarra, B.S. Garbow, Y. Ikebe, V.C. Klema, and C.B. Moler, *Matrix Eigensystem Routines - EISPACK Guide,* Lecture Notes in Computer Science, Vol. 6, 2nd edition, Springer-Verlag, Berlin, (1976).

[30] G.W. Stewart, *Introduction to Matrix Computations,* Academic Press, New York (1973).

[31] D.C. Sorensen, *Buffering for Vector Performance on a Pipelined MIMD Machine,* Parallel Computing, Vol. 1, pp. 143-164, (1984).

[32] J.H. Wilkinson, *The Algebraic Eigenvalue Problem,* Clarendon Press, Oxford (1965).