Parallel Computing 3 (1986) 25–34
North-Holland

Implementation of some concurrent algorithms for matrix factorization *,**

J.J. DONGARRA

Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, U.S.A.

A.H. SAMEH

Department of Computer Science, University of Illinois, Urbana/Champaign, IL 61801, U.S.A.

D.C. SORENSEN

Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, U.S.A.

Received August 1985

Abstract. Three parallel algorithms for computing the QR-factorization of a matrix are presented. The discussion is primarily concerned with implementation of these algorithms on a computer that supports tightly coupled parallel processes sharing a large common memory. The three algorithms are a Householder method based upon high-level modules, a Windowed Householder method that avoids fork-join synchronization, and a Pipelined Givens method that is a variant of the data-flow type algorithms offering large enough granularity to mask synchronization costs. Numerical experiments were conducted on the Denelcor HEP computer. The computational results indicate that the Pipelined Givens method is preferred and that this is primarily due to the number of array references required by the various algorithms.

Keywords. Denelcor HEP, performance analysis.

1. Introduction

This paper discusses implementations of various forms of the QR factorization on the Denelcor HEP. The motivation for examining these implementations was to investigate performance issues that we might expect to face in developing mathematical software for linear algebra problems on emerging parallel architectures. The Denelcor HEP is particularly well suited for such a study because it offers the possibility of very fine grain parallelism through low overhead synchronization primitives. We point out certain synchronization problems that arise within the more tightly coupled variations of the algorithm and offer a comparison of the performance of these variations.

2. The Denelcor HEP computer

Our experiments were carried out on Denelcor HEP computers located at Argonne National Laboratory, Los Alamos National Laboratory, and the Ballistics Research Laboratory. The

* An earlier version of this paper appeared in the proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences in January 1985.

** 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-33.

Denelcor HEP is the first commercially available machine of the MIMD variety. It supports tightly coupled parallel processing, and is quite different from an SIMD machine (i.e., a vector or array processor). The fully configured computing system offered by Denelcor consists of up to 16 processing elements (PEMs) sharing a large global memory through a crossbar switch. Within a single PEM, parallelism is achieved through pipelining independent serial instructions streams called processes. The principal pipeline that handles the numerical and logical operations consists of synchronous functional units that have been segmented into an eight-stage pipe. The storage functional unit (SFU) operates asynchronously from this execution pipeline. Processes are synchronized through marking memory and register locations with the full or empty state. This means that a process requesting access to a memory location may be blocked until that location is marked full by another process. Such suspension of a process takes place only in the SFU and therefore does not interfere with other processes that are ready to execute instructions. The reader is referred to the article [6] by Jordan for further details on the HEP architecture.

3. Variations of the *QR*-factorization

We examined three variations of the *QR* factorization. They were the Householder, Windowed Householder, and Pipelined Givens methods. Concurrency was exploited in the Householder method by expressing the factorization in terms of two high-level modules: a matrix times vector operation and add a rank one matrix to a matrix operation. Column operations involved in these two computations were performed in parallel using a fork-join synchronization. The Windowed Householder method is an attempt to reduce the serial bottleneck introduced through the fork-join synchronization required at the beginning and end of each major reduction step. The method creates a fixed number of processes that compete to either compute or apply Householder transformations in a round-fashion to appropriate columns of the matrix. The Pipelined Givens method represents an attempt to capture the efficiency of a dataflow algorithm in utilizing processes at a level of granularity that is coarser than more traditional dataflow algorithms.

4. The Householder method

The problem that each of these methods is designed to solve is the following. 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

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

On serial machines, Householder's method is preferred because its complexity in terms of floating point operations is roughly half that of Givens' method. However, this is at the expense of roughly twice as many array references as in Givens' method, and this effect is noted on many serial machines where there is a nontrivial cost associated with an array reference. This aspect of the two algorithms becomes extremely important on machines that have memory accesses through some switching mechanism, because array references begin to dominate the calculations in these situations.

The version of Householder's method we present here provides a good example of an approach to programming linear algebra libraries in such a way that they are transportable across a wide variety of architectures and yet are reasonably efficient on a given machine. This

approach—the high-level module approach—is described in detail in [2,4]. Within this scheme a library is restructured in terms of a few modules that are targeted for efficiency on a given machine. All of the special coding necessary to take advantage of parallelism or vectorization is contained within the modules. Householder's method consists of constructing transformations of the form

$$I - 2ww^{\mathrm{T}}$$

with the unit vector w constructed to transform the first column of a given matrix into a multiple of the first coordinate vector e_1. At the kth stage of the algorithm one has

$$Q_{k-1}^{\mathrm{T}} A = \begin{pmatrix} R_{k-1} & S_{k-1} \\ 0 & A_{k-1} \end{pmatrix}.$$

and w_k is constructed such that

$$\left(I - 2w_k w_k^{\mathrm{T}}\right) A_{k-1} = \begin{pmatrix} \rho_k & s_k^{\mathrm{T}} \\ 0 & A_k \end{pmatrix}.$$

The factorization is then updated to the form

$$Q_k^{\mathrm{T}} A = \begin{pmatrix} R_k & S_k \\ 0 & A_k \end{pmatrix}, \quad \text{with } Q_k^{\mathrm{T}} = \begin{pmatrix} I & 0 \\ 0 & I - 2w_k w_k^{\mathrm{T}} \end{pmatrix} Q_{k-1}^{\mathrm{T}}.$$

This is the basic algorithm used in LINPACK [3] for computing the QR factorization of a matrix. It should be noted that this algorithm can be made efficient on a given machine if we have good routines for the following two operations:

$$z^{\mathrm{T}} = w^{\mathrm{T}} A \quad (\text{vector} \times \text{matrix})$$

and

$$\hat{A} = A - 2wz^{\mathrm{T}} \quad (\text{rank-one-modification}).$$

One might choose a higher level module than this, but it turns out that one more operation,

$$z = Aw \quad (\text{matrix} \times \text{vector}),$$

is all that is needed to obtain all of the major routines in LINPACK. Thus, efficient coding of these three routines is all that is needed to transport the entire package from one machine to another. For a vector machine such as the CRAY-1 the vector times matrix operation should be coded in the form

$$z^{\mathrm{T}} \leftarrow z^{\mathrm{T}} + \omega_i a_i^{\mathrm{T}}, \quad i = 1, 2, \ldots, m,$$

where a_i is the ith row of A and ω_i is the ith component of w. For an MIMD machine such as the Denelcor HEP, this operation should be coded in the form

$$\zeta_j = w^{\mathrm{T}} \hat{a}_j, \quad j = 1, 2, \ldots, n \ (\text{in parallel}),$$

where $\hat{a}_j$ is the jth column of A and ζ_j is the jth component of z. This coding has been done, and it has been possible to achieve super-vector speeds on the CRAY-1 and near-maximum speedup on the Denelcor HEP. Far more spectacular results are possible on the HEP if one codes the modules for efficiency in assembly language. This is partially due to the inefficient FORTRAN compiler currently in use under HEP/OS. However, one can do a very good job of managing index calculations and memory references if the matrix vector product is written in assembly language. Details of these techniques are reported in [8].

In this paper we consider only FORTRAN implementations. Within this framework, it is possible to achieve a 20% increase in performance on the HEP by more sophisticated scheduling

of the creation and application of the Householder transformations. That is the subject of the next section. However, in evaluating the merit of such a performance increase, one must consider that each subroutine of the package would require such a recoding effort and that maintainability and integrity of the package would suffer from the process.

5. Windowed Householder method

The Windowed Householder method performs the same numerical operations as the standard Householder method but attempts to keep all available processors busy by executing from a pool of tasks consisting of two types: 'compute a transformation' and 'apply a transformation'. The task of computing a transformation has precedence over the task of applying a transformation during the selection process. However, synchronization is required to ensure that the transformation for the kth column gets computed before the transformation for the $k+1$st column and to ensure that the $k+1$st transformation is not applied to a column until the kth transformation has been applied.

One may think of a window of fixed width moving down the diagonal of the unreduced matrix, as illustrated in Fig. 1. When this window is positioned over column k of the matrix, the transformation to zero out the kth column is computed and applied to the remaining columns in the window. The index k is recorded as the last transformation computed, and the window is moved to the $k+1$st column. At the same time, additional processes are competing to apply the transformations that have been computed by the window to the remaining columns of the matrix that lie to the right of the current window position.

6. Pipelined Givens method

The Pipelined Givens method is the most highly tuned method of the three algorithms discussed here. It is influenced by the more traditional data flow and systolic array algorithms proposed and investigated in [5,7]. The idea is to construct an algorithm with chunks of computation that are at a granularity level of a few FORTRAN statements rather than at the binary operation level present in more traditional data flow algorithms. The concept is quite similar in spirit to the large grained data flow techniques proposed by Babb [1]. However, the details of the implementation are quite different from Babb's systematic approach. The method we present assumes a globally shared memory together with a low overhead synchronization mechanism as attributes of the target architecture for this algorithm. The HEP is, of course, the machine that is principally considered. However, the algorithm would be applicable to any architecture that would support a model of computation containing these two features.

The technique might be contrasted with the Householder method based on high-level

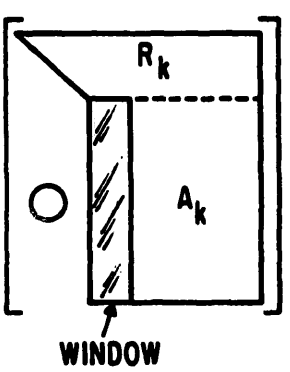

Fig. 1. Windowed Householder method.

modules. If one were to raise the level of 'module' to the highest level subroutine calls within a linear algebra library, then this method would be the method of choice on machines supporting the low overhead synchronization and shared data requirements mentioned above. The reason for this is twofold. As we shall demonstrate with our computational results, memory references are likely to play a far more important role in accessing algorithm performance than they have on serial machines. This 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. Moreover, there is the opportunity to adjust the level of granularity through the specification of a given parameter in order to property 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} which are constructed to zero out the ijth element of the matrix A. Such a matrix may be thought of as a 2×2 orthogonal matrix of the form

$$G = \begin{pmatrix} \gamma & \sigma \\ -\sigma & \gamma \end{pmatrix}.$$

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

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

represents a $2 \times n$ matrix, then a zero can be introduced with left multiplication by G into the β position through proper choice of γ and σ. 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 (with the possible exception of the ii, ij, ji, and jj entries). The matrices G_{ij} are used to reduce A to upper triangular form in the following order:

$$(G_{n-1,n} \cdots G_{2n} G_{1n})(G_{n-2,n-1} \cdots G_{2,n-1} G_{1,n-1}) \cdots (G_{23} G_{13})(G_{12}) A = R.$$

The order of the zeroing pattern may be seen in the following 5×5 example:

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

where the symbol $\otimes_j$ means that entry was zeroed out by the jth transformation. This order is important if one wishes to 'pipeline' the row reduction process. This 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 new row may enter the pipe immediately after the row ahead has been processed by the first segment. Each row proceeds one behind the other until the entire matrix has been processed. However, because of data dependencies, at no time can these rows get out of order once they have entered the pipe. Moreover, at a particular location within a segment, one of two operations must be done: compute a transformation or apply one.

The method is more easily grasped if one considers the following three diagrams. In Fig. 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 components of the next two rows of A that must be reduced. In Fig. 3 we see the row $(\alpha\ \alpha\ \alpha\ \alpha\ \alpha)$ being passed through the triangle R during the reduction process. The first

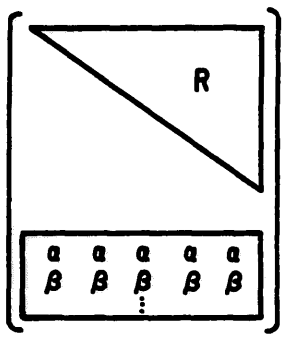

Fig. 2. Partially reduced matrix.

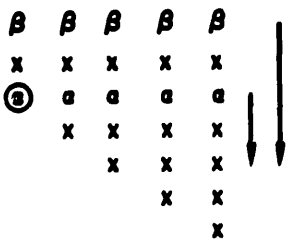

Fig. 3. Pipelined row reduction.

entry has been zeroed by computing and applying the appropriate Givens transformations as described above, and we are ready to zero out the second entry. In a serial algorithm this row would be completely reduced to zero before the row ($\beta\ \beta\ \beta\ \beta\ \beta$) could be referenced. In the parallel version the processing of this row is begun immediately after the α-row. The first row of R is modified during the introduction of a zero in the first position of the α-row so it is important that the processing of the β-row is suitably synchronized with the processing of the α-row.

To accomplish this synchronization and also achieve a way to adjust the level of granularity of the algorithm, we consider the matrix R as a linear array. In Fig. 4 we depict this linear array with natural row boundaries marked with : and pipe segments marked with |. The length of a segment is

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

The number of segments is a parameter to the program. The α and β arrays are represented as in Fig. 3 with the α array entering the second segment and the β array entering the first segment. A row must gain entry to the next segment before releasing the current segment in order to keep the rows in order. 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 data flow algorithm presented in [5]. The subroutine ROWRED in the Appendix shows how this is accomplished on the HEP in FORTRAN. The variables in ROWRED that are preceded by a $ (e.g. $KWHERE) are *asynchronous* variables. These variables are in a *full* or an *empty* state during execution. If an asynchronous variable appears to the left of an assignment statement it must be empty in order for the statement to execute. If it appears to the right of an assignment statement it must be full before the statement will execute. This mechanism allows concurrent processes to communicate and synchronize through data memory.

Table 1 indicates the effect of adjusting the number of segments in the array R. The first column contains the number of segments. The remaining columns give the corresponding execution time (in seconds) of the factorization of a matrix with $n = 50, 100, 150, 200$ columns respectively.

The results of these experiments seem to indicate that the performance of the algorithms is not extremely sensitive to the number of segments, but that it does appear desirable to get the

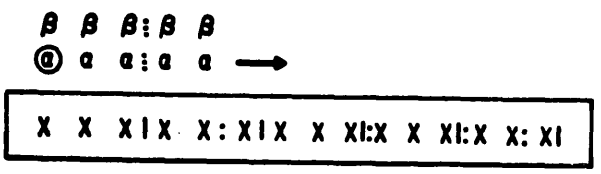

Fig. 4. R as a segmented pipe.

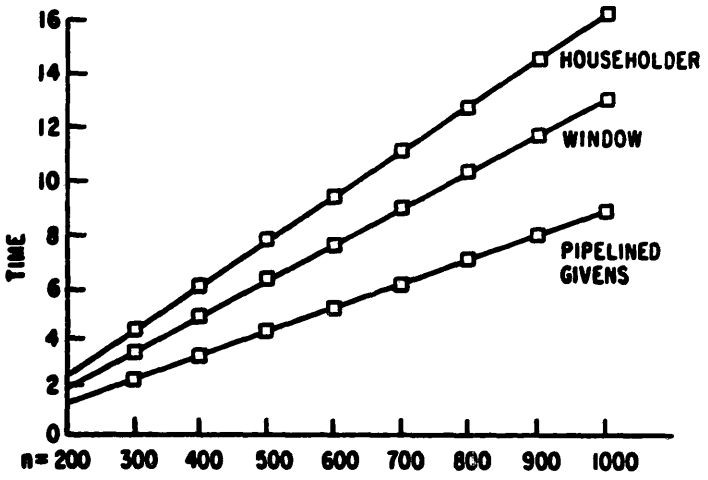

Fig. 5. A comparison of parallel QR-factorizations.

Table 1
Adjusting number of segments in R

Segments	$N = 50$	$N = 100$	$N = 150$	$N = 200$
10	3.83	13.17	27.95	47.79
20	3.15	10.92	22.79	38.40
30	2.83	10.11	21.44	36.39
40	2.84	10.09	21.33	35.11
50	2.85	10.08	21.28	35.99
60	2.85	10.07	21.26	35.93
70	2.86	10.08	21.24	35.90
80	2.88	10.08	21.23	35.87
90	2.88	10.09	21.24	35.86
100	2.90	10.10	21.25	35.85

length of a segment roughly equal to n (the number of columns). The optimum time for each n seems to be around $n/2$ segments. The results shown here have the number of rows of the matrix fixed at $m = 500$ and the number of active processes set at 20. This number of processes appeared to give the best performance, however, the results were fairly insensitive within the range of 10 to 20 processes.

7. Comparison of the methods

While extensive runs were made with these codes, their relative performance is best summarized with a graph. In Fig. 5 we represent the performance in terms of execution time of the three QR factorization variants on matrices of order $m \times 200$ where $m = 200, 300, \ldots, 1000$. The experiments show the Pipelined Givens method to be roughly twice as efficient as the Householder method. The Windowed Householder method achieved roughly a 20% reduction in execution over the Householder method.

8. Conclusions

Experiments were carried out in the relatively well understood setting of linear algebra in order to isolate peculiarities of the HEP environment in particular and to gain understanding

about efficiency of algorithms on parallel processors in general. Some recent work on efficient use of the HEP register memory to construct vector operations in software [4,8] can improve the performance of the Householder routine tremendously but similar coding of the Pipelined Givens method has not yet been carried out. Comparable increases in performance are expected when this has been done. To obtain fair comparisons, we stuck to FORTRAN for these experiments. This decision may affect the results in that array references totally dominate the performance of these algorithms.

Appendix

```
      SUBROUTINE ROWRED(M,N,R,ACOL,PTAG,KUSH,KTASK,MARK)
C
C     PURPOSE:
C        THIS SUBROUTINE TAKES THE COLUMN VECTOR ACOL
C        AND PULLS IT THROUGH THE TRIANGLE R REDUCING ACOL TO ZERO
C        AND MODIFYING R BY THE GIVENS METHOD.
C
C     INPUT PARAMETERS
C
C     M INTEGER
C
C     N AN INTEGER VARIABLE DEFINING THE LENGTH OF ACOL AND DIMENSION OF
C       THE TRIANGULAR MATRIX CONTAINED IN R.
C
C     R A REAL LINEAR ARRAY CONTAINING AN N BY N UPPER TRIANGULAR MATRIX
C       IN PACKED FORM.  STORAGE SCHEME IS CONCATONATION OF SUCCESSIVE ROWS
C       OF R.
C
C    ACOL IS A REAL LINEAR ARRAY OF LENGTH AT LEAST N CONTAINING
C       THE VECTOR TO BE REDUCED BY GIVENS TRANSFORMATIONS.
C
C    PTAG IS AN INTEGER VARIABLE CONTAINING THE IDENTIFIER OF THE CALLING
C      WORK ROUTINE.
C
C    KUSH IS A POSITIVE INTEGER DEFINING THE LENGTH OF A PIPELINE SEGMENT
C      KUSH SHOULD BE LARGE ENOUGH TO ABSORB THE COST OF PIPELINE
C      SYNCHRONIZATION.
C
C    KTASK IS THE INDEX OF THE VECTOR PASSED.  IT IS ASSUMED THAT KTASK
C      REFERS TO THE NUMBER OF THE COLUMN OF THE MATRIX A(TRANSPOSE)
C      THE VALUE OF KTASK IS USED TO SET A STOP VALUE ON THE REDUCTION
C      AND TO INDICATE THE NEXT TASK TO THE FOLLOWING CALL TO ROWRED.
C
C    MARK CONTAINS THE INDEX OF THE LAST SEGMENT OF REFERENCED BY THIS
C      COPY OF ROWRED DURING THE PERFORMANCE OF KTASK.  THE SEGMENTS ARE
C      CONTROLLED BY $KWHERE(*)  ROWRED MUST OWN (IE HAVE READ) $KWHERE(J)
C      IN ORDER TO MODIFY SEGMENT J OF THE ARRAY R.
C
C     D.C. SORENSEN
C     MCSD
C     ARGONNE NATIONAL LABORATORY
C     OCT 1984
C
      INTEGER M,N,PTAG,KUSH,KTASK,MARK
C     REAL ACOL(*),R(*)
C     SET UP THE SYNCHRONIZATION BLOCK
C
      INTEGER K,$LOCK,$START,$NACT,$KWHERE
      COMMON/SYNC/K,$LOCK,$START,$NACT,$KWHERE(450)
      COMMON/DBG/$WLOCK,WLOCK,$DLOCK,DLOCK,ICOUNT,WARRAY(5,500)
C
C     DECLARE LOCAL VARIABLES
C
      INTEGER DUMMY,IJSTOP,NEXT,L,LSTOP,J,LL,INC,KTP1
      REAL GAMMA,SIGMA,ONE,ZERO,TAU,TMAX,AA,RR
```

```
      DATA ONE,ZERO /1.0,0.0/
      LSTOP  = MIN0(KTASK,N)
      MARKP1 = MARK
      KTP1 = KTASK + 1
      L  = 1
      IJ =  1
      IJSTOP = IJ + KUSH
      J  = IJ
      INC = N
      LL = 0
C
C     PULL THE COLUMN A THROUGH R
C
  150 CONTINUE
         IF (IJSTOP .GT. IJ)
     1   THEN
C
C
C           DECIDE HERE TO COMPUTE OR TO APPLY REFLECTOR
C           IF IJ .EQ. L THEN A NEW REFLECTOR SHOULD BE COMPUTED
C           OTHERWISE THE OLD ONE SHOULD BE APPLIED.
C
            IF (L .GT. IJ)
     1      THEN
C               APPLY A REFLECTOR
C
C               INPUTS ARE  IJ - RUNNING INDEX IN R
C                           J - RUNNING INDEX IN ACOL
C
C                        GAMMA AND SIGMA DEFINE THE CURRENT REFLECTOR
C
C
                AA = ACOL(J)
                RR =R(IJ)
                ACOL(J) = GAMMA*AA - SIGMA*RR
                R(IJ) = SIGMA*AA + GAMMA*RR
                IJ = IJ + 1
                J  = J + 1
C
C               END APPLY REFLECTOR
C
C               OUTPUT    R(IJ) MODIFIED
C                         ACOL(J) MODIFIED
C
            ELSE
C               COMPUTE A REFLECTOR
C
C                    INPUTS
C
C                    L - LOCATION OF CURRENT ROW START IN R
C                    LL - LOCATION OF CURRENT ROW START IN ACOL
C                    INC - INCREMENT TO THE NEXT ROW START IN R
C                    LSTOP - REDUCTION IS FINISHED WHEN LSTOP ROWS
C                            OF R HAVE BEEN PROCESSED (LL .GT. LSTOP)
C                    N     - REDUCTION IS FINISHED WHEN THE N-TH REFLECTOR
C                            HAS BEEN COMPUTED
C
C                    L AND LL ALIGN ELEMENTS OF R AND ACOL FOR THE "LL-TH ROW
C                    ELIMINATION STEP.
C
                     LL = LL + 1
                     IF (LL .GT. LSTOP) GO TO 1000
                     SIGMA = ONE
                     GAMMA = ZERO
                     TMAX =
                      SIGN(ONE,R(IJ))*MAX1(ABS(ACOL(LL)),ABS(R(IJ)))
                     IF (TMAX .EQ. ZERO) GO TO 250
                     TAU =
                      TMAX*SQRT((ACOL(LL)/TMAX)**2 + (R(IJ)/TMAX)**2)
                        GAMMA = R(IJ)/TAU
                        SIGMA = ACOL(LL)/TAU
                        R(IJ) = TAU
```

```
  250               CONTINUE
                    IJ = IJ + 1
                    J  = LL + 1
                    L = L + INC
                    INC = INC - 1
                    IF (LL .GE. N) GO TO 1000
C
C               END COMPUTE REFLECTOR

            ENDIF
          ELSE
C
C           BEGIN SYNCRONIZATION OF SEGMENTED PIPE
C
C           INPUT
C                 IJ .EQ. IJSTOP TO GAIN ENTRY
C                 KUSH - THE LENGTH OF A SEGMENT
C                 MARK - THE INDEX OF THE CURRENT SEGMENT OF R OWNED
C                        BY THIS PROCESS.
C                                /
                  IJSTOP = IJ + KUSH
C
C           WAIT UNTIL PROCESS AHEAD HAS EXITED THE NEXT SECTION
C
            MARKP1 = MARK + 1
            DUMMY = $KWHERE(MARKP1)
C
C           LET THE NEXT PROCESS ENTER THIS SECTION
C
            $KWHERE(MARK) = KTP1
            MARK = MARKP1
C
C           END SYNCHRONIZATION
C
          ENDIF
          GO TO 150
C
 1000 CONTINUE
C
C     EXIT LEAVING $KWHERE FULL
C
      $KWHERE(MARKP1) = KTP1
C
C     GO FIND ANOTHER JOB
C
      RETURN
C
C     LAST CARD OF ROWRED
C
      END
```

References

[1] R.G. Babb II, Parallel processing with large grain data flow techniques, *IEEE Comput.* **17** (7) (1984) 55–61.

[2] J.J. Dongarra and R. Hiromoto, A collection of parallel linear equations routines for the Denelcor HEP, *Parallel Comput.* **1** (2) (1984) 133–142.

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

[4] J.J. Dongarra and D.C. Sorensen, A parallel linear algebra library for the Denelcor HEP, Argonne National Laboratory Report MCS-TM-33, 1984.

[5] M. Gentleman and H.T. Kung, Matrix triangularization by systolic arrays, *Proc. SPIE 298 Real-Time Signal Processing IV*, San Diego, CA (1981).

[6] H.F. Jordan, Experience with pipelined multiple instruction streams, *IEEE Proc* (January 1984) 113–123.

[7] A. Sameh, Parallel algorithms in numerical linear algebra, *Proc. Crest Conference on Design of Numerical Algorithms for Parallel Processing*, Bergamo, Italy (Academic Press, New York, 1985).

[8] D.C. Sorensen, Buffering for vector performance on a pipelined MIMD machine, *Parallel Comput.* **1** (2) (1984) 143–164.