M2AN. MATHEMATICAL MODELLING AND NUMERICAL ANALYSIS
- MODÉLISATION MATHÉMATIQUE ET ANALYSE NUMÉRIQUE

P. CARNEVALI
G. RADICATI
Y. ROBERT
P. SGUAZZERO

Efficient FORTRAN implementation of the gaussian elimination and Householder reduction algorithms on the IBM 3090 vector multiprocessor

M2AN. Mathematical modelling and numerical analysis - Modélisation mathématique et analyse numérique, tome 23, n° 1 (1989), p. 63-86

<http://www.numdam.org/item?id=M2AN_1989__23_1_63_0>

MATHEMATICAL MODELLING AND NUMERICAL ANALYSIS
MODÉLISATION MATHÉMATIQUE ET ANALYSE NUMÉRIQUE
(Vol. 23, n° 1, 1989, p. 63 à 86)

EFFICIENT FORTRAN IMPLEMENTATION OF THE GAUSSIAN ELIMINATION AND HOUSEHOLDER REDUCTION ALGORITHMS ON THE IBM 3090 VECTOR MULTIPROCESSOR (*)

by P. CARNEVALI (1), G. RADICATI (1),
Y. ROBERT (1,2), P. SGUAZZERO (1)

Abstract. — *Let A be a real dense $m \times n$ matrix, and b a vector with m components. Assume that $m \geq n$ and rank $(A) = n$. In the case $m = n$, the well-known Gaussian elimination method with partial pivoting is the most commonly used algorithm to solve the linear system $Ax = b$. In the case $m \geq n$, the Householder reduction scheme provides with an efficient solution of the linear least squares problem,* $\min_x \| Ax - b \|$ (GV 83). *In the paper we discuss the efficient implementation, both vector and parallel, of these two algorithms, on the* IBM 3090 *Vector Multiprocessor* (Buc 86, Tuc 86).

Vectorization

We recast the two standard algorithms in terms of high-level matrix-matrix modules. More precisely, for Gaussian elimination, we use a modified version of the Dongarra et al. (DGK 84) *kji*-SAXPY *algorithm termed Rank-r Update and described in* (RS 86), *and for Householder reduction we implement a Block-r generalization of the algorithm, as proposed by Berry* et al. (BGH 86). *Interestingly enough, the two implementations share the same generic computational kernel.*

Parallel implementation

We first show that the Rank-r Update and Block-r Reduction algorithms can be expressed with the same task graph. Then we present some performance data of a two- to six-processors implementation of our two algorithms. Basically, at step k, there remain $n - kr$ columns of the matrix A to update. Rather than equally distributing the updating of these columns among the processors, we give less columns to one of them, which in turn has in charge to prepare the next r columns for step $k + 1$ (RS 86). *As a consequence, there is almost no sequential part in the algorithm. Furthermore, since we use schemes which update r columns at a time, we only have n/r synchronization points, instead of n as in a classical algorithm.*

(*) Received in June 1987.

(1) IBM Eur. Center for Sc. and Eng. Comp. (ECSEC) via Giorgione 159, 00147 Roma, Italy.

(2) CNRS, Laboratoire TIM3-INPG, 46, avenue F.-Viallet, 38031 Grenoble Cedex, France.

M²AN Modélisation mathématique et Analyse numérique 0399-0516/89/016324/$ 4.40

Résumé. — *Soit A une matrice réelle dense de taille $m \times n$, et soit b un vecteur à m composantes, avec $m \geqslant n$ et rang $(A) = n$. Dans le cas $m = n$, l'algorithme d'élimination de Gauss avec pivotage partiel est le plus couramment utilisé pour résoudre le système linéaire $Ax = b$. Dans le cas $m \geqslant n$, l'algorithme de décomposition* QR *de Householder fournit la solution du problème aux moindres carrés linéaires* $\min_x \| Ax - b \|$ (GV 83). *Dans cet article, nous proposons une implémentation* FORTRAN *efficace de ces deux algorithmes, à la fois vectorielle et parallèle, sur le Multiprocesseur Vectoriel* IBM 3090 (Buc 86, Tuc 86).

Vectorisation

*Nous reformulons les deux algorithmes standards en termes de modules de large granularité (comme la multiplication de deux matrices). Pour l'élimination de Gauss, nous utilisons une version modifiée du schéma kji-*SAXPY *de Dongarra* et al. (DGK 84), *dénommée Rank-r LU Update et décrite dans* (RS 86). *Pour la décomposition de Householder, nous employons une méthode par blocs appelée Block-r QR Update qui généralise l'algorithme usuel (voir Berry* et al. (BGH 86)). *Ces deux implémentations possèdent le même noyau de calcul.*

Implémentation parallèle

Nous montrons tout d'abord que les algorithmes Rank-r LU Update et Block-r QR Update peuvent être analysés à l'aide du même graphe de tâches. Nous présentons ensuite les performances obtenues en utilisant de 2 à 6 processeurs. L'idée de base est la suivante : à l'étape k, on doit modifier $n - kr$ colonnes de la matrice A. Plutôt que de partager également le travail entre tous les processeurs, nous affectons moins de colonnes à l'un d'entre eux, qui en revanche a en charge de préparer l'étape suivante. Par suite, il n'y a presque aucune partie séquentielle dans l'algorithme. En outre, comme nous utilisons des schémas de calcul qui modifient r colonnes à chaque étape, nous avons seulement $\frac{n}{r}$ points de synchronisation, contre n pour un algorithme classique.

INTRODUCTION

Let A be a real dense $m \times n$ matrix, and b a vector with m components. Assume that $m \geqslant n$ and *rank* $(A) = n$. In the case $m = n$, the well-known Gaussian elimination method with partial pivoting is the most commonly used algorithm to solve the linear system $Ax = b$. In the case $m \geqslant n$, the Householder reduction scheme provides with an efficient solution of the linear least squares problem $\min_x \| Ax - b \|$ (GV 83).

In the paper we describe an efficient FORTRAN implementation, both vector and parallel, of these two algorithms, on the IBM 3090 Vector Multiprocessor (Buc 86, Tuc 86).

We first discuss the vectorization. Let us choose $m = n = 1\,000$ for the sake of illustration. A straightforward FORTRAN implementation (in double precision) reaches about 21 Mflops for both algorithms. We show that doubling this performance (and even better) can be achieved by more elaborate, still entirely FORTRAN written, procedures. Following Berry *et al.* (BGH 86), we recast the two algorithms in terms of high-level matrix-

matrix modules. More precisely, for Gaussian elimination, we use a modified version of the Dongarra *et al.* (DGK 84) *kji*-SAXPY algorithm termed Rank-r LU Update and described in (RS 86), and for Householder reduction we implement a Block-r generalization of the algorithm, termed Block-r QR Update.

Interestingly enough, the two implementations share the same generic computational kernel : at the k-th step in both algorithms, we perform a rank-r modification of the right bottom $(m - kr) \times (n - kr)$ block of the matrix A.

Consider the previous example with $m = n = 1\,000$: we are lead to a performance of 58 Mflops in the case of Gaussian elimination, and of 44 Mflops in the case of Householder reduction. This difference is mainly due to the possibility of using only tryadic operations *multiply-and-adds* in the Rank-r Update algorithm, which are implemented by a single vector instruction. On the contrary, there remain scalar products to be computed in the Block-r Reduction scheme, for which the FORTRAN compiler (FORT 86 release 2.1.1) generates three vector instructions (with an important scalar overhead).

The last sections of the report deal with a parallel implementation of the Rank-r LU Update and Block-r QR Update algorithms. We first show that these two algorithms can be expressed with the same task graph. Such a graph has been introduced in the case of Gaussian elimination by Lord *et al.* (LKK 83) and Cosnard *et al.* (CMRT 86), to model the precedence constraints which direct the execution ordering.

We then present some performance data of a two- to six-processor implementation of the two algorithms. Basically, at step k, there remain $n - kr$ columns of the matrix A to update. Rather than equally distributing the updating of these columns among the processors, we give less columns to one of them, which in turn has in charge to prepare the next r columns for step $(k + 1)$ (RS 86). As a consequence, there is almost no sequential part in the algorithm. Furthermore, since we use schemes which update r columns at a time, we only have n/r synchronization points, instead of n as in a classical algorithm.

Reporting good speed-ups (such as 1.9 for the two-processor execution), we conclude that our two schemes, which are very efficient for an uniprocessor implementation, are also very suitable for parallel execution.

STANDARD GAUSSIAN ELIMINATION AND HOUSEHOLDER REDUCTION

Let A be a real dense $m \times n$ matrix, with $m \geq n$ and *rank* $(A) = n$. When discussing Gaussian elimination, we assume that $m = n$ (in this case A is non-singular), while when discussing Householder reduction we let

$m \geq n$. In both cases the corresponding decomposition method can be expressed in the following compact form :

DO $k = 1$, *kmax*
(1) prepare k-th transformation from column k of A
(2) apply k-th transformation to columns $k + 1$ to n

We specify below the transformations which are computed in each method. Note that $kmax = n - 1$ for Gaussian elimination and $kmax = min\ (n, m - 1)$ for Householder reduction.

Gaussian elimination

The following algorithm computes the LU decomposition of A with partial pivoting $PA = LU$, where P is a permutation matrix, L is lower triangular with unit diagonal and U is upper triangular.

DO $k = 1, n - 1$
(1) prepare k-th transformation with column k of A
($1a$) search for the pivot index ipiv(k) = 1 in column k
($1b$) interchange $a(k, k)$ and $a(k, 1)$
($1c$) scale elements in position $k + 1$ to n by factor $-1/a(k, k)$
(2) apply k-th transformation to columns $k + 1$ to n
($2a$) interchange rows k and 1 :
($2b$) apply a rank $-$ 1 transformation :

```
DO j = k + 1, n
DO i = k + 1, n
    a(i, j) = a(i, j) + a(i, k)* a(k, j).
```

This scheme is the usual *kji*-SAXPY scheme, as identified by Dongarra *et al.* (DGK 84). At step k, the k-th column of L is computed and a rank $-$ 1 transformation is applied to the right bottom $(n - k) \times (n - k)$ block of A.

Transformation ($2b$) consists of a double loop of SAXPY operations (the innermost loop), but it can be better described in terms of a matrix operation, namely a rank $-$ 1 update, represented by the double loop over j and i.

The number of floating-point operations executed in the algorithm is approximately $2\ n^3/3$.

Householder reduction

The following algorithm computes the QR decomposition $A = QR$ of A, where Q is orthogonal and R is upper triangular. The lower triangular part of A is overwritten by Q stored in factored form, and its strictly upper triangular part is overwritten by R. The diagonal of R is stored in an auxiliary array AR. Scaling factors of the Householder transformation are stored in an auxiliary array BETA. See (GV 83) for more details :

DO $k = 1$, min $(n, m-1)$
(1) prepare k-th transformation with column k of A
($1a$) compute the norm of elements in position $k+1$ to n
($1b$) compute $a(k, k)$, $ar\ (k)$ and beta (k)
($1c$) scale elements in position k to n by *sqrt* (beta (k))
(2) apply k-th transformation to columns $k+1$ to n
($2a$) compute scalar product of column k with all columns j:

```
DO j = k + 1, n
    s(j) = 0 . dO
    DO i = k, m
        s(j) = s(j) + a(i, k)* a(k, j)
```

($2b$) apply a rank $-$ 1 transformation :

```
DO j = k + 1, n
DO i = k, m
    a(i, j) = a(i, j) - s(j)* a(i, k)
```

The operation count for this algorithm is roughly $2\,n^2(m - n/3)$.

Performances

Let $m = n$ for the sake of comparison. A straightforward FORTRAN implementation reveals similar performances for the two algorithms, as is reported in Table 1. Note however that the operation count is not the same : Householder reduction is twice as expensive as Gaussian elimination for square matrices.

TABLE 1
Performance in Mflops for standard implementation.

Matrix size	*Gaussian elimination*	*Householder reduction*
400 x 400	20.4	20.4
600 x 600	21.2	21.0
800 x 800	21.5	21.4
1000 x 1000	21.9	21.6

To get better performances, we want to recast the two algorithms in terms of high-granularity modules so as to minimize the number of memory accesses. Moving from BLAS vector-vector routines (LHKK 79), such as an AXPY operation $Y = Y + \alpha \times Y$, to BLAS2 matrix-vector operations, such as a matrix-vector product, or even BLAS3 matrix-matrix computations, such as the multiplication of two matrices, makes it easy for the vectorizing compiler to minimize data movement in addition to using vector operations. This is mainly due to the fact that the results of a vector-vector operation can be temporary stored in a vector register, reutilized immediately, and

need not be stored back in the memory until the completion of the computation (DS 86).

Such a restructuring has been described in (RS 86) in the case of Gaussian elimination. A brief presentation of the corresponding algorithm, termed *Rank-r LU Update* in (RS 86), and of its efficient vector implementation is included in the next section for the sake of completeness.

For Householder reduction, we implement a Block-*r* generalization of the usual QR algorithm, as proposed by Berry *et al.* (BGH 86). The resulting algorithm, termed *Block-r QR Update,* reveals to share the same computational kernel as the Rank-*r* LU Update algorithm.

EFFICIENT VECTOR IMPLEMENTATION

From the description of the two standard algorithms, we see that we can easily recast them in terms of matrix-vector modules. However, to gain a third level of granularity, we have to move from processing a single column to processing blocks of columns : we need to replace the *kmax* phases of the computation, each of them updating a given column, by *kmax*/*r* macro-phases, each of them updating a block of *r* columns of the original matrix. This leads to the following scheme :

DO $k = r, kmax, r$
  (1) prepare the (k/r)-th macro-transformation
    using columns $k - r + 1$ to k of A
  (2) apply this macro-transformation to columns $k + 1$ to n

Step (2) now corresponds to $O(r \times (n - k)^2)$ floating-point operations. Moving to the new scheme requires some technical modifications both for Gaussian elimination and Householder reduction. In what follows we detail further steps (1) and (2) for both methods. Note that the upper bound *kmax* in the *k*-loop can be chosen as $kmax = kmax - mod\,(kmax, r)$ (where $kmax = min\,(m, n - 1)$), or a smaller number if we do not want to deal with too small matrices : in this case we execute the last steps of the elimination with the standard scheme.

The Rank-*r* LU Update algorithm

Basically, we want to process *r* columns at a time, and postpone the updating of the remaining columns until the completion of this pre-processing phase. That is to say, we perform a standard Gaussian elimination on the current block of *r* columns and keep track of pivot indices, so as to apply the corresponding rank-*r* transformation to the rest of the matrix, when the processing is complete. See (DS 86) for a very similar algorithm in a parallel environment.

Applying a rank-r transformation to the remaining columns requires that all the r elements of each pivoting row have been interchanged during the computation of the transformation. On the contrary, we would only interchange elements after the diagonal in the usual algorithm. We undo these temporary interchanges after having applied the transformation.

We are lead to the following FORTRAN procedure (for the sake of clarity, we do not include the testing for small pivots) :

```
      SUBROUTINE LU(A, LDA, N, IPIV, IRANK)
C
      IMPLICIT REAL*8 (A - H, O - Z)
      REAL*8 A(LDA, *)
      INTEGER IPIV (*)
C
C --- IRANK = the block size r
C
      KMAX = N - MOD(N, IRANK) - 32
      DO 10 K = IRANK, KMAX, IRANK
         KP1 = K + 1
         CALL PREP(A, LDA, N, IPIV, IRANK, K)
         CALL APPLY(A, A, LDA, N, IPIV, IRANK, K, KP1, N)
         CALL UNDO(A, LDA, N, IPIV, IRANK, K)
   10 CONTINUE
C
      CALL DGE000(A, LDA, N, IPIV, KP1)
      RETURN
      END
```

The subroutine PREP corresponds to step (1) of the algorithm : we generate a rank-r transformation :

```
      SUBROUTINE PREP(A, LDA, N, IPIV, IRANK, K)
C
      IMPLICIT REAL*8 (A - H, O - Z)
      REAL*8 A(LDA, *)
      INTEGER IPIV (*)
C
C     PREP processes columns K - IRANK + 1 to K
C
      DO 70 KK = K - IRANK + 1, K
         L = KK
         T = DABS(A(KK, KK))
         DO 10 I = KK + 1, N
            IF (DABS(A(I, KK)) . LE . T) GO TO 10
            T = DABS(A(I, KK))
            L = I
   10    CONTINUE
         IPIV(KK) = L
         IF (L . EQ . KK) GO TO 30
         DO 20 J = K - IRANK + 1, K
            T = A(L, J)
            A(L, J) = A(KK, J)
            A(KK, J) = T
```

```
 20     CONTINUE
 30     CONTINUE
        T = - 1 . DO/A(KK, KK)
        DO 40 I = KK + 1, N
            A(I, KK) = T*A(I, KK)
 40     CONTINUE
        DO 60 J = KK + 1, K
            T = A(KK, J)
        DO 50 I = KK + 1, N
            A(I, J) = A(I, J) + T*A(I, KK)
 50     CONTINUE
 60   CONTINUE
 70 CONTINUE
    RETURN
    END
```

Note that the DO 40 and DO 50 loops are automatically vectorized by the compiler.

The subroutine APPLY applies the rank-r transformation to the remaining columns :

```
  PROCESS DIRECTIVE('*VDIR :')
      SUBROUTINE APPLY(A, B, LDA, N, IPIV, IRANK, K, KLIM1, KLIM2)
C
      IMPLICIT REAL*8 (A - H, O - Z)
      REAL*8 A(LDA, N), B(LDA, N)
C
C     update rows K - IRANK + 1 to K
C
C *VDIR : PREFER VECTOR
      DO 30 J = KLIM1, KLIM2
          DO 20 KK = K - IRANK + 1, K
              L = IPIV(KK)
              ACC = A(L, J)
              A(L, J) = A(KK, J)
              DO 10 L = K - IRANK + 1, KK - 1
                  ACC = ACC + A(KK, L)*A(L, J)
  10          CONTINUE
              A(KK, J) = ACC
  20      CONTINUE
  30  CONTINUE
C
C     apply rank-r transformation
C
C *VDIR : PREFER VECTOR
      DO 60 I = K + 1, N
          DO 50 J = KLIM1, KLIM2
              ACC = A(I, J)
              DO 40 KK = K - IRANK + 1, K
                  ACC = ACC + B(I, KK)*B(KK, J)
```

```
40          CONTINUE
            A(I, J) + ACC
50      CONTINUE
60  CONTINUE
    RETURN
    END
```

Considering the first block of nested DO loops, we have forced vectorization to occur in the outermost DO 30 loop on columns. The two others loops are very short and should be executed in scalar mode.

Considering now the second block, we have forced vectorization to occur in the outermost I loop. Consecutive sections (of length 128) of vector A(*, J) are loaded from main memory into a vector register and stored inside until their update by means of the KK loop is complete. Note that it is necessary to use explicitely a temporary register (which is expanded to a vector-one) in the FORTRAN code : otherwise the compiler would not detect that sections of vectors A(*, J) can be kept in the vector registers throughout their updating. In the subroutine APPLY, matrices A and B are formally distinct but *de facto* synonym : this prevents the compiler from declaring the DO 40 loop recursive, hence not vectorizable.

The routine UNDO is responsible for undoing temporary exchanges that were needed for generating the rank-r transformation. We execute it in scalar mode. Finally, subroutine DGE000 is a standard Gaussian elimination routine to end up the computation.

All the code above can be made very modular, and r can be viewed as a parameter of the algorithm. Unfortunately, it is impossible to force the compiler to vectorize the DO 30 loop on J in the routine APPLY, due to the non-constant induction variables in the two inner loops. We have then two alternatives : either we let this block execute in scalar mode, or we borrow a technique from (DH 79) (DE 84) (DD 85) are unroll the DO 20 and DO 10 loops. Of course the second solution will lead to improved performances, especially if we take care, while computing the rank-r transformation in routine PREP, to record the data we need into two temporary arrays, namely

$$ATEMP(i, j) = A(k - i, k - j)\ , \quad ITEMP(i) = IPIV(k - i)\ , \quad 0 \leqslant i, j \leqslant r - 1\ .$$

Such an implementation is detailed in (RS 86). But the price to pay for squeezing the most out of the routine APPLY is that the unrolling makes the code dependent on parameter r.

When quoting the performance of the Rank-r LU Update algorithm below, we make reference to the unrolled implementation. However, we point out that the main computational block of the LU procedure is the second block of nested loops in APPLY, which needs not be unrolled (this block typically represents 80 % of the total execution time). To give a rough estimate, the modular solution is 10 % less efficient than the unrolled one.

We find out experimentally that for big enough matrices, the larger r, the better performance we have, as illustrated in Table 2 below : we report the performances (expressed in Mflops) obtained for five values of r ranging from 4 to 32, and for several sizes of matrices, ranging from 400 to 1 000. We also include the speed of the assembly coded routine DGEF of the ESSL library (ESSL 86) in Table 2 for the sake of comparison.

TABLE 2
Rank-r LU Update algorithm.

Matrix size	*Rank 4*	*Rank 8*	*Rank 16*	*Rank 24*	*Rank 32*	*DGEF*
400 x 400	32.2	38.7	46.5	48.5	48.0	61.7
600 x 600	33.4	40.4	49.0	52.2	52.4	66.1
800 x 800	34.3	41.9	51.6	55.0	56.0	69.0
1000 x 1000	35.0	42.9	53.0	57.1	58.1	70.1

It can be seen from Table 7 that we reach 80 % of the speed of the assembly routine DGEF. Performance improvement over the standard implementation is 150 % for the largest values of r.

The Block-*r* QR Update algorithm

The basic principle of the algorithm is just the same as before : we have in mind to compute r Householder vectors at a time, and then apply the corresponding r transformations to the remaining columns.

The key observation (BV 85) is that the product of r Householder transformations $P_i = I - uu^t$ (where $u^t u = 2$) can be written in the form $Q_r = I - V_r U_r^t$, where $U_r = (u_1, u_2, \ldots, u_r)$ and $V_r = (P_r V_{r-1}, u_r)$ are matrices of size $m \times r$.

The Block-r QR Update algorithm has a structure very similar to the Rank-r LU Update algorithm, as evidenced by the following FORTRAN subroutine :

```
      SUBROUTINE QR(A, LDA, M, N, R, BETA, KBLOCK)
C
      IMPLICIT REAL*8 (A - H, O - Z)
      REAL*8 A(LDA, *), R(*), BETA(*), U(1001, 32), V(1001, 32)
C
C     KBLOCK = THE BLOCK SIZE R
C
      KMAX = N - MOD(N, KBLOCK) - 32
      DO 10 K   KBLOCK, NMAX, KBLOCK
         KP1 = K + 1
         CALL PREP(A, LDA, M, N, R, BETA, KBLOCK, K, U, V)
         CALL APPLY(A, LDA, M, N, KBLOCK, K, U, V, K + 1, N)
   10 CONTINUE
C
      CALL DHR000(A, LDA, M, N, R, BETA, KP1)
      RETURN
      END
```

For sake of simplicity, we assume in the procedure that the leading dimension LDA of A is bounded by 1001 and that we do not use values of r greater than 32. The routine DHR000 is a standard Householder reduction routine to end up the computation. The routine PREP generates the matrices U and V which will be needed in the routine APPLY :

```
@PROCESS DIRECTIVE('*VDIR :')
      SUBROUTINE PREP(A, LDA, M, N, R, BETA, KBLOCK, K, U, V)
C
      IMPLICIT REAL*8 (A - H, O - Z)
      REAL*8 A(LDA, *), R(*), BETA(*), U(1001, 32), V(1001, 32), S(32)
C
C     PREP PROCESSES COLUMNS K-KBLOCK + 1 TO K
C
      KSTART = K - KBLOCK + 1
C--- clean out first components of U and V
C*VDIR : PREFER SCALAR
      DO 20 J = 1, KBLOCK
C*VDIR : PREFER SCALAR
         DO 10 I = KSTART, K
            U(I, J) = 0 . DO
            V(I, J) = 0 . DO
   10    CONTINUE
   20 CONTINUE
C--- main loop on columns
      DO 150 KK = KSTART, K
C---     compute new vector u
         ALPHA = 0 . DO
         DO 30 I = KK, M
            ALPHA = ALPHA + A(I, KK)*A(I, KK)
   30    CONTINUE
         ALPHA = DSQRT(ALPHA)
         IF (A(KK, KK) . GT . 0 . DO) ALPHA = - ALPHA
         R(KK) = ALPHA
         A(KK, KK) = A(KK, KK) - ALPHA
         BETA(K) = - 1 . DO/(ALPHA * A(KK, KK))
C---     store u in corresponding column of matrices U and V
         SQBETA = DSQRT(BETA(K))
         DO 40 I = KK, M
            U(I, KK - KSTART + 1) = SQBETA * A(I, KK)
            V(I, KK - KSTART + 1) = U(I, KK - KSTART + 1)
   40    CONTINUE
C---     update matrix V
C*VDIR : PREFER SCALAR
         DO 50 J = 1, KK - KSTART
            S(J) = 0 . DO
   50    CONTINUE
C*VDIR : PREFER VECTOR
         DO 70 I = KK, M
```

```
            ACC = U(I, KK - KSTART + 1)
            DO 60 J = 1, KK - KSTART
               S(J) = S(J) + ACC*V(I, J)
 60         CONTINUE
 70       CONTINUE
C*VDIR : PREFER VECTOR
          DO 90 I = KK, M
            ACC = U(I, KK - KSTART + 1)
            DO 80 J = 1, KK - KSTART
               V(I, J) = V(I, J) - S(J)*ACC
 80         CONTINUE
 90       CONTINUE
C---      prepare next column if not processing the last one
          IF (KK . NE . K) THEN
C*VDIR : PREFER SCALAR
            DO 100 J = 1, KK - KSTART
               S(J) = 0 . DO
 100        CONTINUE
C *VDIR : PREFER VECTOR
            DO 120 I = KK, M
               ACC = A(I, KK + 1)
               DO 110 J = 1, KK - KSTART
                  S(J) = S(J) + ACC*V(I, J)
 110           CONTINUE
 120        CONTINUE
C *VDIR : PREFER VECTOR
            DO 140 I = KK, M
               ACC = A(I, KK + 1)
               DO 130 J = 1, KK - KSTART
                  ACC = ACC - S(J)*V(I, J)
 130           CONTINUE
 140        CONTINUE
          ENDIF
 150 CONTINUE
     RETURN
     END
```

The DO 30 and DO 40 loops are automatically vectorized by the compiler. The DO 30 loop gives us an opportunity to explain how scalar products are vectorized by the compiler (FORT 86, release 2.1.1) : when computing the scalar product of two vectors X and Y, the compiler generates a code quivalent to the following one for each section (of length 128) of X and Y :

```
load X into vector register VR
multiply VR by Y and store in VR
zero partial sums
```

```
accumulate partial sums
sum partial sums
```

As a result, we have three vector instructions and an important scalar overhead : an assembly code would zero and sum the partial sums outside the segmentation loop (i.e. only once rather than for each section), and would use only two vector instructions per section : a load of vector X, and a compound instruction to multiply by vector Y and accumulate partial sums (VECT 86).

In the routine APPLY we first compute the scalar product of each of the remaining columns by each of the vectors in matrix U. We then perform a rank-*r* transformation very similar to that in Gaussian elimination :

```
      PROCESS DIRECTIVE('*VDIR :')
C     SUBROUTINE APPLY(A, LDA, M, N, U, V, KBLOCK, K, U, V, KLIM1, KLIM2)
      IMPLICIT REAL*8 (A - H, O - Z)
      REAL*8 A(LDA, *), U(1001, 32), V(1001, 32), S(32, 1001)
C
      KSTART = K - KBLOCK + 1
C
C     compute scalar products and store in matrix S
C
C*VDIR : PREFER VECTOR
      DO 20 J = KLIM1, KLIM2
         DO 10 KK = 1, KBLOCK
            S(KK, J) = 0 . DO
   10    CONTINUE
   20 CONTINUE
C *VDIR : PREFER VECTOR
      DO 50 I = KSTART, M
         DO 40 J = KLIM1, KLIM2
            DO 35 KK = 1, KBLOCK
               S(KK, J) = S(KK, J) + A(I, J)*U(I, KK)
   30       CONTINUE
   40    CONTINUE
   50 CONTINUE
C
C     apply rank-r transformation
C*VDIR : PREFER VECTOR
      DO 80 I = KSTART, M
         DO 70 J = KLIM1, KLIM2
            ACC = A(I, J)
            DO 60 KK = 1, KBLOCK
               ACC = ACC - S(KK, J)*V(I, KK)
   60       CONTINUE
          A(I, J) = ACC
   70    CONTINUE
   80 CONTINUE
      RETURN
      END
```

TABLE 3
Block-r QR Update algorithm.

Matrix size	*Block 4*	*Block 8*	*Block 16*	*Block 24*	*Block 32*
400 x 400	30.0	36.3	39.6	39.3	38.6
600 x 600	30.7	37.0	41.0	41.8	41.6
800 x 800	31.1	37.6	41.8	42.8	43.2
1000 x 1000	31.4	38.2	42.5	43.7	43.9

Table 3 below reports some performance data. We choose the same values of r and the same problem sizes as in Table 2. Little difference in performances is observed for rectangular matrices, provided that the number of columns n is greater than the vector section size.

Performance improvement over the standard scheme is 100 % : for large values of r, we double the execution speed. However, we no longer achieve similar performances as for Gaussian elimination.

Comparing the routines APPLY in the Rank-r and Block-r algorithms, we see that both routines are divided into two parts. The two second parts are very similar, and very efficiently implemented by the compiler, which uses compound vector instructions *multiply-and-add* or *multiply-and-substract*. In the case of Gaussian elimination, the first part of the routine is optimized as well by the compiler, owing to the unrolling that we described. On the other hand, the first part of the Householder routine consists of computing scalar-products, for which the compiler generates a rather inefficient code.

Vectorization epilogue

In order to sum up our results related to the vectorization of the Gaussian elimination and Householder reduction schemes, we build up the following Table 4, where we report the performance (in Mflops) of the best implementation for each scheme, and its speed-up over the straightforward implementation.

TABLE 4
Vectorization epilogue.

Matrix size	*Rank-r LU Update*		*Block-r QR Update*	
	Mflops	*Speed-up*	*Mflops*	*Speed-up*
400 x 400	48.5	*2.38*	39.6	*1.94*
600 x 600	52.4	*2.47*	41.8	*1.99*
800 x 800	56.0	*2.60*	43.2	*2.02*
1000 x 1000	58.1	*2.65*	43.9	*2.04*

To sum up even more concisely these results, we can say that the use of high-granularity modules (and unrolling techniques in the case of Gaussian elimination) has caused a performance improvement of 150 % for the Rank-r LU Update algorithm, and of 100 % for the *Block-r QR Update* algorithm.

PARALLEL IMPLEMENTATION

We present in this section some performance data of a two- to six-processor implementation of the Rank-r LU Update and Block-r QR Update algorithms. We first describe the parallel decomposition of both algorithms, and we report their performance on the IBM 3090-600e VF (Tuc 86).

Parallel decomposition

As we have previously discussed, the main body of the two procedures can be concisely expressed as follows :

```
DO k = r, KMAX, r
  (1) prepare the (k/r)-th macro-transformation :
      process columns k - r + 1 to k
       call PREP(A, ..., K, ...)
  (2) apply this macro-transformation :
      update columns k + 1 to n
       call APPLY(A, ..., K, ..., K + 1, N)
```

For Gaussian elimination we can include in (1) the undoing of the temporary exchanges that were performed for the previous transformation. Routine PREP now begins with the additional statement :

```
CALL UNDO(A, LDA, N, IPIV, IRANK, K - IRANK)
```

We let *Prep*(k) denote the task of preparing the (k/r)-th macro-transformation (processing of columns $k - r + 1$ to k, plus if needed the undoing of exchanges in columns $k - r$ to $k - 1$). We consider this task as an indivisible unit of computational activity : we do not want to split the execution of this task among several processors. Its granularity is proportional to the value chosen for the parameter r.

Similarly, *Apply*(k, j) and *Apply*(k, j_1, j_2) denote respectively the task of updating column j and the task of updating columns j_1 to j_2 for the (k/r)-th transformation. We have full freedom in choosing the values of j_1 and j_2, hence the number of tasks *Apply* for any value of k, but each of them will be assigned to a single processor.

Precedence constraints

Parallel algorithms for Gaussian elimination with partial pivoting on SIMD or MIMD machines have been extensively studied in the literature. See (LKK 83) and (CMRT 86) among others. Moving from rank-1 to rank-r transformations does not modify the precedence constraints of the algorithm :

— task $Prep(k)$ must complete execution prior to any task $Apply(k, ., .)$ commencing execution

— all tasks $Apply(k, j)$ for $k + 1 \leq j \leq k + r$ must complete execution prior to task $Prep(k + r)$ commencing execution.

Given k, all tasks $Apply(k, j)$ can be executed concurrently. We describe below two ways of distributing the execution of these tasks among the processors.

Straightforward decomposition

The easiest way to decompose the algorithm is to split equally the execution of the tasks $Apply(k, .)$ among the processors. Assume that we have p processors, where $2 \leq p \leq 6$. Using the Multitasting Facility primitives (see Appendix E in (FORT 86)), we are lead to the following kernel for parallel execution :

DO $k = r$, $kmax$, r

(1) on main processor, prepare rank-r transformation
CALL PREP

(2) on main processor, compute bounds for the p copies
of routine APPLY to be executed in parallel :
KCOL = (N − K)/P
processor i will execute task Apply(k, KLIM1(i), KLIM2(i))
where KLIM1(1) = K + 1
KLIM1(i) = KLIM2(i − 1) + 1 for $i > 1$
KLIM2(i) = KLIM1(i) + KCOL for $1 < i < p$
KLIM2(p) = N

(3) in parallel, update one p-th fraction of the remaining
columns on each processor :
DO $i = 1, p$
CALL DSPTCH(APPLY, ..., KLIM1(i), KLIM2(i))
CONTINUE

(4) synchronize
CALL SYNCRO

CONTINUE

This kernel should give a good insight of the procedure. However, a few modifications to this scheme are necessary to prevent read- and write-conflicts when several processors simultaneously try to access data which are used in all copies of the routine APPLY (such conflicts are due to interferences between the caches of the processors). More precisely, just after calling the routine PREP, we duplicate the temporary arrays needed by all processors, so that each of them works on its own copy of data. For Householder reduction we make $p-1$ copies of matrices U and V. For Gaussian elimination, we duplicate arrays ATEMP and ITEMP $p-1$ times, and we also makes $p-1$ copies of another temporary array W of dimensions $LDA \times r$ into which we record the r columns of the matrix A which are used by all processors (those of index $(k-1)r+1$ to kr at the k-th step). As a consequence, the last kernel of nested loops in routine APPLY becomes the following in the Rank-r LU Update algorithm :

```
      DO 60 I = K + 1, N
         DO 50 J = KLIM1, KLIM2
            ACC = A(I, J)
            DO 40 KK = 1, IRANK
               ACC = ACC + B(K + 1 - KK, J)*W(I, KK)
40          CONTINUE
            A(I, J) = ACC
50       CONTINUE
60    CONTINUE
```

Balanced decomposition

In the previous decomposition, the subroutine PREP is executed sequentially on the main processor, and only the computations relative to the routine APPLY have been parallelized. It is true that these computations represent the main computational body of the whole procedure, but some performance improvement can be achieved by a better repartition of the work among the p processors.

The underlying idea is very simple : rather than splitting the updating of the remaining columns of matrix A into p equal blocks, we could distinguish one processor, say processor 1, to be assigned less columns than the other ones. In turn, processor 1 would have in charge to prepare the next phase of the algorithm (the next instance of routine PREP) once it has finished its own (smaller) amount of updating. Assuming that preparing a column in PREP costs the same as updating one in APPLY, we want the first processor to update r less columns than the other ones do. The main kernel can now be expressed as follows :

```
initialization : on main processor, prepare the first r columns :
CALL PREP(... K = 1 ...)
-----------------------------------------------------------------------------
parallel body :
DO K = r, kmax, r
  (1) on main processor, compute bounds for the p copies
      of routine APPLY to be executed in parallel :
        KCOL = (N - K)/p
        processor i will execute task Apply(k, KLIM1(i), KLIM2(i))
        where KLIM1(1) = K + 1
              KLIM2(1) = KLIM1(1) + KCOL - r + r/p
              KLIM1(i) = KLIM2(i - 1) + 1              for  i > 1
              KLIM2(i) = KLIM1(i) + KCOL + r/p         for  1 < i < p
              KLIM2(p) = N
  (2) duplicate temporary arrays
  (3) in parallel, update remaining columns
      and prepare next step :
      DO i = 2, p - 1
        CALL DSPTCH(APPLY, ..., KLIM1(i), KLIM2(i))
      CONTINUE
      CALL APPLY(..., KLIM1(1), KLIM2(1))
      CALL PREP(... K + r ...)
  (4) synchronize
      CALL SYNCRO
CONTINUE
-----------------------------------------------------------------------------
termination on main processor
CALL APPLY(... K + r ...)
CALL UNDO(... K + r ...) if Gaussian elimination
CALL DGE000(...) or DHR000(...)
```

There is almost no sequential part in this implementation. The main processor updates r columns less than the other ones, hence it has enough time to prepare the next transformation while the other processors are still updating columns for the current transformation.

Mixed strategy

Unfortunately, we have to stop the execution of the previous algorithm before the matrix becomes too small. Assume for instance that $p = 6$. We need to assign at least r columns to processor 1 and 2 r to the other five ones, so that there must remain 11 r columns to update. If $r = 32$, this means that we must stop the algorithm when the remaining matrix is of size 350 ! Even for smaller values of r, we are lead to a comparable bound, since a loop on

columns has been vectorized in the first part of routine APPLY, and we do not want this loop to be too short.

On the other hand, the large granularity of the tasks $Apply(k, ., .)$ makes it worth to dispatch parts of the updating until there remain, say, 80 to 100 columns to update in the matrix. That is why we adopt a mixed strategy : we start with the balanced scheme and process the matrix until reaching the previous bound. At this point we move to the straightforward decomposition and use a modified version of routine APPLY, in which the only vector loops are the DO I loops on rows (the DO 60 loop for Gaussian elimination the DO 50 and DO 80 loops for Householder reduction). The very final part of the computation is performed using a standard scalar routine on a single processor.

Technically, if $p \geqslant 3$, we use the balanced scheme until there remain $p \times (32 + r)$ columns in the matrix. Then we update the next $32(p-2) + (p-1)\,r$ columns using the straightforward decomposition before moving to a uniprocessor scalar standard elimination. For $p = 2$ we simply use the balanced scheme with $kmax = n - mod\,(n, r) - 96$. Clearly the following three objectives

— having tasks of large granularity
— using as long as possible all processors in parallel
— taking full benefit of the vectorization facilities on each processor

are contradictory. The previous strategy represents a compromise between them. Fortunately, the heart of the procedure lies in the very first steps of the execution, since the number of operations to be performed at each step is proportional to the square of the size of the matrix to be updated. During these first steps, the three previous objectives can be fullfilled simultaneously ! Still, we must be prepared to some performance degradation, especially for medium size matrices.

Implementation on the IBM 3090-600e VF

The parallel implementations of the Rank-r LU Update and Block-r QR Update algorithms have been tested on the six-processor machine IBM 3090-600e VF.

In Tables 5 to 9 below, we report the performance (expressed in Mflops) of both algorithms together with the speed-up over their uniprocessor version. Each table corresponds to a given value of p. The size of the problem matrix ranges from 400 to 1 000. We report data for $r = 24$, the value experimentally found to lead to the best performance for $1\,000 \times 1\,000$ matrices. In the tables, we also give the *efficiency* of the parallelization, defined as the ratio of the speed-up over the number of processors (two to six in our case).

TABLE 5
Parallel 2-processor implementation.

Matrix size	*With 2 processors*					
	Rank-24 LU Update			*Block-24 QR Update*		
	Mflops	*Speed-up 2 Pro/1 Pro*	*Efficiency*	*Mflops*	*Speed-up 2 Pro/1 Pro*	*Efficiency*
400 x 400	87.4	*1.80*	*0.90*	68.3	*1.74*	*0.87*
600 x 600	98.1	*1.88*	*0.94*	78.8	*1.89*	*0.94*
800 x 800	105.0	*1.91*	*0.95*	82.0	*1.91*	*0.96*
1000 x 1000	109.4	*1.92*	*0.96*	84.8	*1.94*	*0.97*

TABLE 6
Parallel 3-processor implementation.

Matrix size	*With 3 processors*					
	Rank-24 LU Update			*Block-24 QR Update*		
	Mflops	*Speed-up 3 Pro/1 Pro*	*Efficiency*	*Mflops*	*Speed-up 3 Pro/1 Pro*	*Efficiency*
400 x 400	116.5	*2.40*	*0.80*	87.2	*2.22*	*0.74*
600 x 600	137.6	*2.64*	*0.88*	106.5	*2.55*	*0.85*
800 x 800	148.8	*2.70*	*0.90*	113.8	*2.66*	*0.89*
1000 x 1000	156.1	*2.73*	*0.91*	119.0	*2.72*	*0.91*

TABLE 7
Parallel 4-processor implementation.

Matrix size	*With 4 processors*					
	Rank-24 LU Update			*Block-24 QR Update*		
	Mflops	*Speed-up 4 Pro/1 Pro*	*Efficiency*	*Mflops*	*Speed-up 4 Pro/1 Pro*	*Efficiency*
400 x 400	137.8	*2.84*	*0.71*	102.6	*2.61*	*0.65*
600 x 600	171.3	*3.28*	*0.82*	132.0	*3.16*	*0.79*
800 x 800	188.8	*3.43*	*0.86*	143.8	*3.36*	*0.84*
1000 x 1000	199.4	*3.49*	*0.87*	153.4	*3.51*	*0.88*

TABLE 8
Parallel 5-processor implementation.

Matrix size	*With 5 processors*					
	Rank-24 LU Update			*Block-24 QR Update*		
	Mflops	*Speed-up 5 Pro/1 Pro*	*Efficiency*	*Mflops*	*Speed-up 5 Pro/1 Pro*	*Efficiency*
600 x 600	197.3	*3.78*	*0.76*	152.0	*3.64*	*0.73*
800 x 800	221.6	*4.02*	*0.80*	169.8	*3.96*	*0.79*
1000 x 1000	236.0	*4.13*	*0.83*	181.3	*4.14*	*0.83*

TABLE 9
Parallel 6-processor implementation.

Matrix size	*With 6 processors*					
	Rank-24 LU Update			*Block-24 QR Update*		
	Mflops	*Speed-up 6 Pro/1 Pro*	*Efficiency*	*Mflops*	*Speed-up 6 Pro/1 Pro*	*Efficiency*
600 x 600	217.4	*4.16*	*0.69*	170.2	*4.07*	*0.68*
800 x 800	248.5	*4.51*	*0.75*	194.4	*4.54*	*0.76*
1000 x 1000	269.3	*4.72*	*0.79*	210.4	*4.81*	*0.80*

The previous tables are not entirely representative of our results, since the best value of r depends upon the matrix size and the number of processors. For instance with 6 processors, the best performances are obtained with $r = 8$ for 600×600 matrices, and with $r = 16$ for 800×800 matrices. However the tables should give a good insight of the speed-ups that can be achieved. Note that the more processors we use, the larger the matrix size needed to achieve a given speed-up.

In figures 1 and 2, we sum up more concisely our results and give a better insight to the parallelization. In figure 1, we plot the efficiency of the Rank-24 LU Update algorithm for 600×600 and $1\,000 \times 1\,000$ matrices, for 2 to 6 processors. We report in figure 2 similar quantities for the Block-24 QR Update algorithm.

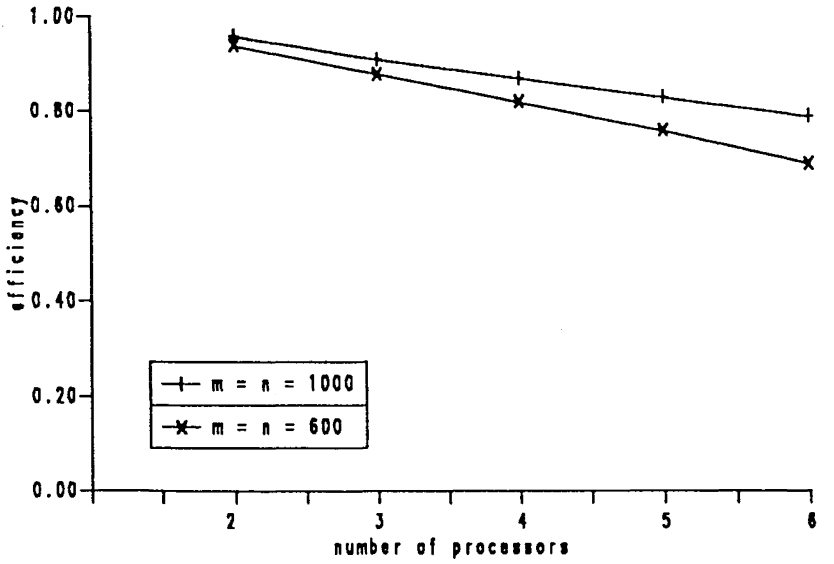

Figure 1. — Efficiency of the Rank-24 LU Update Algorithm.

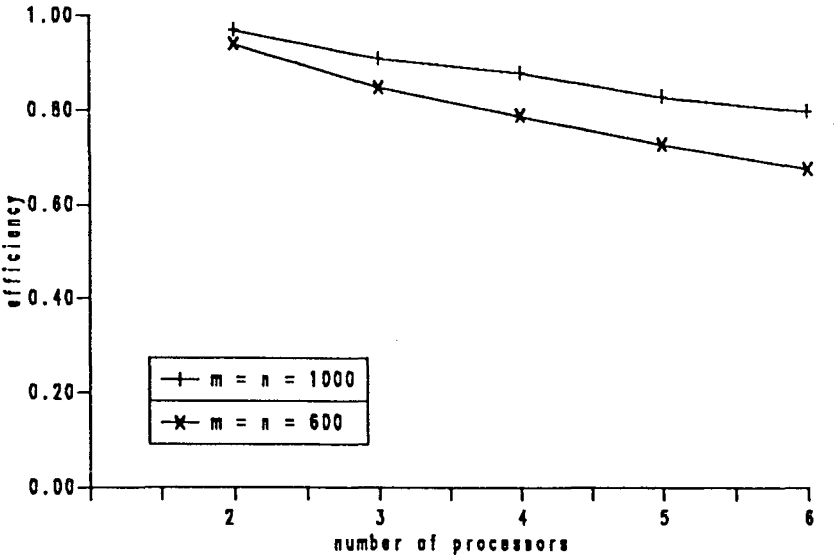

Figure 2. — Efficiency of the Block-24 QR Update Algorithm.

The good speed-ups (and efficiencies) that we report demonstrate the suitability of the Rank-r and Block-r algorithms to a parallel execution. Indeed, two nice features of the parallel implementations we have dealt with, are the following :

— the algorithms are composed of high-granularity tasks

— few synchronization points are required (less than $\frac{n}{r}$ for a rank-r or block-r algorithm).

CONCLUSION

In this report, we have first discussed the FORTRAN implementation on the uniprocessor IBM 3090 VF of the Gaussian elimination and Householder reduction algorithms. Recasting the original algorithms in terms of BLAS3 high-granularity modules, such as matrix-matrix multiplication, has permitted to exploit at best the vectorizing facilities provided by the computer. Indeed, we have designed elaborated, still entirely FORTRAN written procedures, which achieve a performance improvement of more than 100 % over the initial naive implementations.

We have shown in the second part of the report that the *Rank-r LU Update* and *Block-r QR Update* schemes are very suitable for parallel

execution : speed-ups ranging from 1.9 (for two processors) to 4.8 (for 6 processors) have been reported for a parallel execution on the IBM 3090-600e VF.

Using the six-processor implementation of the *Rank-r Update* algorithm, we are lead to a performance of 265.2 Mflops for the solution (in double precision) of a system of equations of order 1 000, as specified in the benchmark « Toward Peak Performance » of (Don 84).

BIBLIOGRAPHY

[BGH 86] M. BERRY, K. GALLIVAN, W. HARROD, W. JALBY, S. LO, U. MEIER, B. PHILIPPE and A. H. SAMEH, *Parallel algorithms on the CEDAR system,* in CONPAR 86 (G. Goos and J. Hartmanis eds.) pp. 25-39, Lecture Notes in Computer Science 237, Springer Verlag (1986).

[BV 85] C. BISCHOFF and C. VAN LOAN, *The WY representation for products of Householder matrices,* Cornell University, Report DCS-85-681 (1985).

[Buc 86] W. BUCHHOLZ, *The IBM System/370 vector architecture,* IBM Systems Journal 25, 1 (1986) pp. 51-62.

[CMRT 86] M. COSNARD, M. MARRAKCHI, Y. ROBERT and D. TRYSTRAM, *Gaussian elimination algorithms for MIMD computers,* in CONPAR 86 (G. Goos and J. Hartmanis eds.) pp. 247-254, Lecture Notes in Computer Science 237, Springer Verlag (1986).

[DD 85] C. DALY and J. J. DU CROZ, *Performance of a subroutine library on vector processing machines,* Computer Physics Communications 37 (1985) pp. 181-186.

[Don 84] J. J. DONGARRA, *Performance of various computers using standard linear equations software in a Fortran environment,* Argonne National Laboratory Report MCA-TM-23 (1984, updated December 1986).

[DE 84] J. J. DONGARRA and S. C. EISENSTAT, *Squeezing the most out of an algorithm in Cray Fortran,* ACM Trans. Math. Software 10, 3 (1984) pp. 221-230.

[DGK 84] J. J. DONGARRA, F. G. GUSTAVSON and A. KARP, *Implementing linear algebra algorithms for dense matrices on a vector pipeline machine,* SIAM Review 12, 1 (1984) pp. 91-112.

[DH 79] J. J. DONGARRA and A. R. HINDS, *Unrolling loops in Fortran,* Software - Practice and Experience 9 (1979) pp. 219-229.

[DS 86] J. J. DONGARRA and D. C. SORENSEN, *Linear algebra on high-performance computers,* in Parallel Computing 85 (M. Feilmeier *et al.* eds.), pp. 221-230, Elsevier Science Publishers B. V. (1986).

[ESSL 86] *Engineering and Scientific Subroutine Library,* Order No. SC23-0184-0, available through IBM branch offices (1986).

[FORT 86] *VS FORTRAN Version 2 Programming Guide (Release 1.1),* Order No. SC26-4222-1, available through IBM branch offices (1986).

[GV 83] G. H. GOLUB and C. F. VAN LOAN, *Matrix computations,* The John Hopkins University Press (Baltimore, MA, 1983).

[LHKK 79] C. LAWSON, R. HANSON, D. KINCAID and F. KROGH, *Basic linear algebras subprograms for Fortran usage,* ACM Trans. Math. Software 5 (1979) pp. 308-371.

[LKK 83] R. E. LORD, J. S. KOWALIK and S. P. KUMAR, *Solving linear algebraic equations on an MIMD computer,* J. ACM 30, 1 (1983) pp. 103-117.

[RS 86] Y. ROBERT and P. SGUAZZERO, *The LU decomposition algorithm and its efficient FORTRAN implementation on the IBM 3090 vector multiprocessor,* IBM ECSEC Technical Report (March 1987).

[Tuc 86] S. G. TUCKER, *The IBM 3090 system : an overview,* IBM Systems Journal 25, 1 (1986) pp. 4-19.

[VECT 86] *IBM System/370 Vector Operations,* Order No. SA22-7125-0, available through IBM branch offices (1986).