Parallel Computing 1 (1984) 133–142
North-Holland

A collection of parallel linear equations routines for the Denelcor HEP *

Jack J. DONGARRA

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

Robert E. HIROMOTO

Computing Division, Los Alamos National Laboratory, Los Alamos, NM 87545, U.S.A.

Received June 1984

Abstract. This paper describes the implementation and performance results for a few standard linear algebra routines on the Denelcor HEP computer. The algorithms used here are based on high-level modules that facilitate portability and perform efficiently in a wide range of environments. The modules are chosen to be of a large enough computational granularity so that reasonably optimum performance may be insured. The design of algorithms with such fundamental modules in mind will also facilitate their replacement by others more suited to gain the desired performance on a particular computer architecture.

Keywords. HEP computer, linear algebra routines, assembly language programming, performance analysis parallel algorithms, parallel computer.

We have been using the Denelcor HEP (Heterogenous Element Processor) to implement a modest set of parallel routines to handle some common problems that arise when dealing with dense matrices in linear algebra: matrix multiplication, Cholesky decomposition of a positive definite matrix, LU factorization with partial pivoting, and QR factorization of a general matrix. Jordan [3] describes the architecture and programming environment of the Denelcor HEP, and Stewart [5] provides a complete description of the algorithms discussed here. Part of the experiment was to examine the ease of taking a collection of algorithms, expressed in terms of high-level modules, and implementing then on a computer with parallel constructions, such as the Denelcor HEP. Our hope was to gain near-optimum performance from these routines by implementing only the underlying modules using parallel constructs. We look on our experience as an experiment in producing portable algorithms that have a high level of granularity in their structure and high performance on a wide variety of computer architectures.

The basic algorithms used here are the same as those reported in a paper by Dongarra and Eisenstat [1] (with the exception of QR factorization). These algorithms are based on standard procedures in linear algebra. They have been written to retain much of the original mathematical formulation and are based on matrix-vector operations. Designing the algorithms in terms of such operations is the hard part of an implementation. By understanding the algorithm in terms

* This work was supported in part by the Applied Mathematical Sciences Research Program (KC-04-02) of the Office of Energy Research of the U.S. Department of Energy under contracts W-31-109-Eng-38 and W-7405-ENG-36.

of the mathematical steps necessary to reduce or solve a problem, we can uncover the structure at a high level. When programming from an algorithmic description, we have a tendency to focus on small details during an implementation. To unveil the desired structure, we must look higher, avoiding the details, and must concentrate on operations that embody the computational components of an algorithm.

To produce a parallel version for the HEP, we replaced only three lower level modules: matrix-vector multiplication ($y = y + Ax$), vector-matrix multiplication ($y^T = y^T + x^T A$), and a rank one update to a matrix ($A = A + xy^T$). These modules represent a high level of granularity in the algorithm in the sense that they are based on matrix-vector operations, $O(n^2)$ work, not just vector operations, $O(n)$ work.

The parallelism in matrix-vector multiplication was obtained by performing m independent inner products with a matrix of size $m \times n$ and a vector of length n. For the vector-matrix multiplication, n independent inner products were performed with a vector of length m and a matrix of size $m \times n$. The parallelism in the rank one update was obtained by performing n operations, a scalar times a vector added to a column of the matrix.

The technique used in this parallel implementation was based on a concept called 'self-scheduling' of parallel processes [2]. In self-scheduling, a number of parallel processes are created and allowed to asynchronously access a unique processing index value that points to a specific parallel segment of the computation to be done. The accessing procedure allows only one process to gain exclusive read/write privileges of the loop index, while all other similarly contending processes are momentarily blocked from such access. Before the controlling process relinquises the loop index, however, it updates the index value: positioning it to point to the next parallel processing segment. As a process becomes free or completes its curreent task, this technique allows it to self-schedule itself for the next parallel segement of computation. Rather than preassigning blocks of parallel computations across processes, the self-scheduling technique allows each process to acquire more work at the earliest possible moment, thereby, reducing the idle time between completed parallel processes. Listed in the appendix are two source codes that illustrate the use of the high-level module SSDOT, making up the kernal for these algorithms. For clarity the subroutines SMXPY and SSDOT appearing in the appendix have been annotated to describe the technique of self-scheduling.

The routines were run first with straight sequential FORTRAN 77 versions of the modules, using no parallel constructions, and then with the sequential modules replaced by their parallel counterparts. The parallel algorithms, written in an extended version of FORTRAN 77, require the same number of floating-point operations and have identical properties with respect to roundoff errors as their sequential counterparts. There was, however, an increase in memory utilization by the parallel programs. This increase was required in order to support the necessary parallel processing environment. For each parallel process the processing environment must provide in part for the guaranteed and exclusive access of all local variables associated within a particular routine. As a means to implement this local assessing structure, all local variables for each parallel subroutine were duplicated by the operating system (for as many parallel processes to be spawned) and placed into replicated, pre-assigned memory locations during the creation of that specific parallel routine. In our problems the additional memory requirements was typically less than one percent over their sequential counterparts. This slight increase reflects the memory space required for the execution of 32 parallel processes. Althought this increase is small, the potential for a significant increase is quite evident if the user requires a large number of parallel subroutines each having a large number of local variables. The optimum results are shown in Table 1 for problems with matrices of order 200.

Significant improvements in the parallel version can be seen over the sequential implementation, as much as a factor of 8.53. An important point here is that only small changes were made to the programs to gain these speedups. By replacing the fundamental modules, which would be

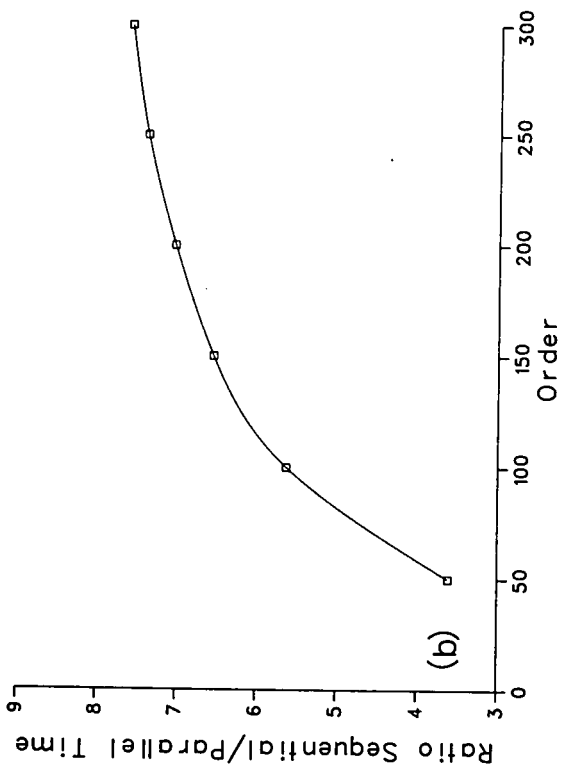
Table 1
Experimental results for matrices of order 200

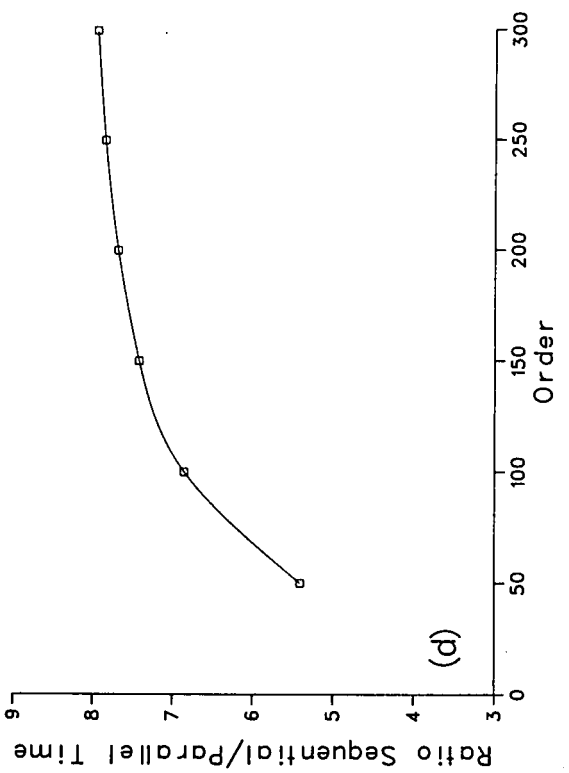
Algorithm	Ratio execution time Sequential/parallel
Matrix multiply	8.53
Cholesky decomposition	7.07
LU factorization	6.84
QR factorization	7.82

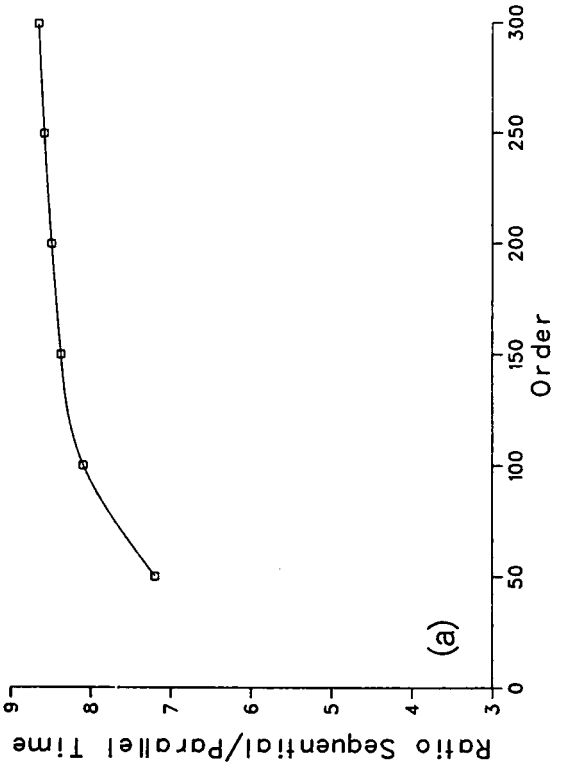
transparent to a user, we have implemented a modest yet important collection of programs on the HEP with minimal effort and with no change to the basic algorithm. These programs may not reach the maximum performance possible, but the software effort has been quite small in relation to the gains in performance. We realize it may be possible to achieve even better performance by investing more time and effort in the parallel implementation. As the algorithms become more complicated, however, the improvement becomes limited by the amount of work associated with the nonparallel parts – partial pivoting, scaling, and norm calculations. In a parallel implementation on the HEP with one Process Execution Module (PEM), we can expect at most a factor 10 speedup over the sequential counterpart. Figures 1(a)–(d), on the other hand, show our results for matrices whose order varied over the range from 50 to 300 with 24 processes executing in parallel. We see from these results that the large computational granularity (the matrix order) contributes directly to the overall gains in performance for these parallel algorithms. Figures 2(a)–(d) on the other hand show our results for a matrix of order 200 executing from 1 to 32 processes in parallel. By fixing the computational granularity, we clearly see how well the HEP single PEM system supports the parallelism as the number of parallel processes increases. We may further deduce from Fig. 2 the intrinsic parallelism of these algorithms by noting the near linear speedups (indicating minimal hardware overhead) resulting from the execution of from one to five parallel processes.

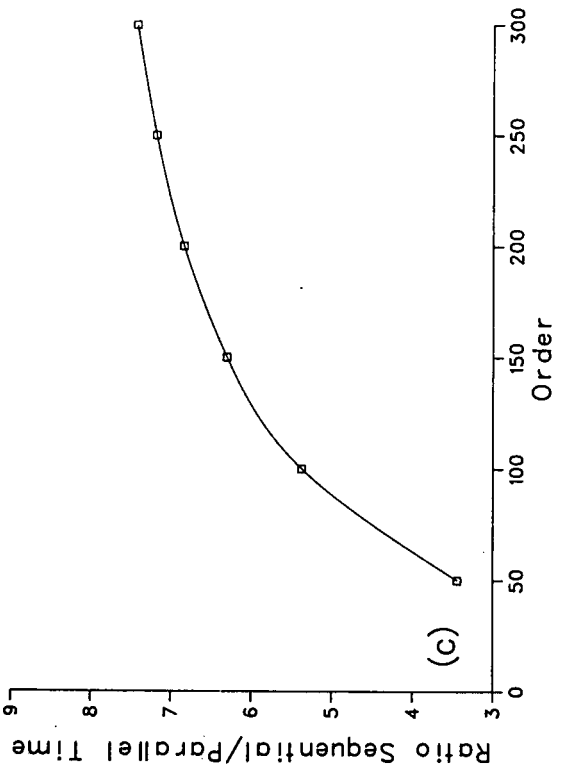
One of our goals is to avoid locking the algorithms into one computer's architecture, however fast that one may be; another goal is to design the algorithms at a level that the fundamental modules need only be replaced to gain the desired performance. The module concept allows us to divide a large problem into small, easily understood pieces that can be programmed separately and verified at each step of the development process. These pieces are then chosen, perhaps repeated, to solve various aspects of the larger problem. The success of this approach in efficiently solving problems across a wide spectrum of computers depends on how well the modules can be chosen so that the modules are at a high enough level to allow a significant number of arithmetic operations to be performed.

We have for the past few years been using a set of routines called the Basic Linear Algebra Subprograms (BLAS) [4]. These routines focus on the vector level; that is, they operate on one-dimensional arrays. Typical of these operations are functions such as inner product, a multiplication of a vector added to another vector, and vector scaling. The BLAS are well suited for operations that occur on some of the vector processors, but they are not the best choice for certain other vector processors, multiprocessors, or parallel processing computers. The next higher level up from simple vector operations is the matrix-vector operations. Operations such as a matrix times a vector and rank one changes to a matrix not only embody the operations described by the BLAS, but also have the advantage of providing enough computational granularity for efficient parallel processing. With vectorizing techniques, the basic idea is to produce vector functions out of the inner loops of algorithms such as the BLAS. By contrast, when an algorithm is parallelized, one focuses on the outer loops. By constructing algorithms from modules that have a high level of granularity, it will be possible to allow for either implementation in a straightforward and simple manner.


Ratio Sequential/Parallel Time
Order
(b)


Ratio Sequential/Parallel Time
Order
(d)


Ratio Sequential/Parallel Time
Order
(a)


Ratio Sequential/Parallel Time
Order
(c)

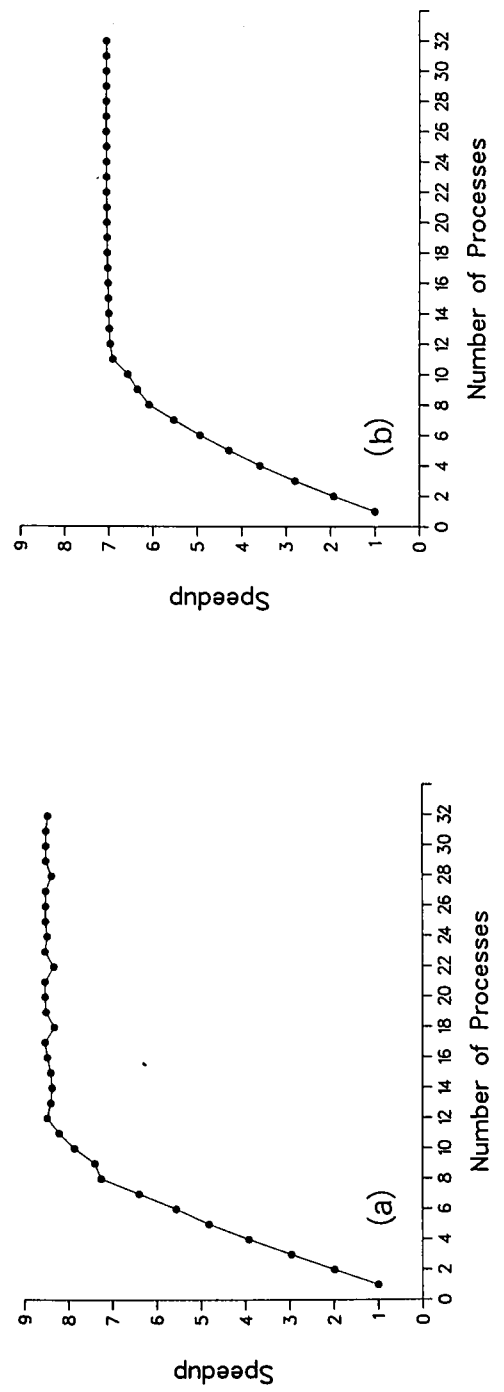


Fig. 2. Parallel speedup performances plotted against number of processes for fixed matrix of order 200. Matrix multiply (a), Cholesky decomposition (b), LU factorization (c), QR factorization.

With such a wide variety of computer systems and architectures in use or proposed, there is a very real challenge for people designing algorithms, namely, how to write software that is both efficient and portable. The solution lies in the granularity of the task. Programs expressed in terms of modules with a high level of granularity reflect less of the detail and retain more of the basic mathematical formulation. This allows for a wider range of efficient implementations because the computational intense parts are isolated in high-level modules. These modules can be dealt with separately, perhaps retargeting them for quite different architectures, making the overall algorithms efficient on the targeted architecture.

Appendix

We list the source of routines that were implemented on the Denelcor HEP. Each routine contains documentation describing its purpose as well as its parameter definitions. The source listings for routines SMXPY and SSDOT have also been annotated in order to clarify the technique of self-scheduling as employed in this particular study.

```
      SUBROUTINE MM (A, LDA, N1, N3, B, LDB, N2, C, LDC)
C
      INTEGER LDA, N1, N3, LDB, N2, LDC
      REAL A(LDA,*), B(LDB, *), C(LDC, *)
C
C  PURPOSE:
C  Multiply matrix B times matrix C and store the result in matrix A.
C
C  PARAMETERS:
C
C     A      REAL (LDA, N3), matrix of N1 rows and N3 columns
C     LDA    INTEGER,        leading dimension of array A
C     N1     INTEGER,        number of rows in matrices A and B
C     N3     INTEGER,        number of columns in matrices A and C
C     B      REAL (LDB, N2), matrix of N1 rows and N2 columns
C     LDB    INTEGER,        leading dimension of array B
C     N2     INTEGER,        number of columns in matrix the B, and number
C                            of rows in the matrix C
C     C      REAL (LDC, N3), matrix of N2 rows and N3 columns
C     LDC    INTEGER,        leading dimension of array C
C
C  -----------------------------------------------------------------------
C
      DO 20 J = 1, N3
         DO 10 I = 1, N1
            A(I, J) = 0.0
10       CONTINUE
         CALL SMXPY (N2, A(1, J), N1, LDB, C(1, J), B)
20    CONTINUE
C
      RETURN
      END
      SUBROUTINE LLT (A, LDA, N, ROWI, INFO)
C
```

```
      INTEGER LDA, N, INFO
      REAL A(LDA, *), ROWI (*), T
C
C  PURPOSE:
C     Form the Cholesky factorization A = L * L^T of a symmetric positive
C     definite matrix A with factor L overwriting A.
C
C  PARAMETERS:
C     A      REAL(LDA, N),   matrix to be decomposed; only the lower triangle need
C                            be supplied; the upper triangle is not referenced
C     LDA    INTEGER,        leading dimension of array A
C     N      INTEGER,        number of rows and columns in the matrix A
C     ROWI   REAL(N),        work array
C     INFO   INTEGER,        = 0 for normal return
C                            = I if Ith leading minor is not positive definite
C
C  ------------------------------------------------------------------
C
      INFO = 0
      DO 30 I = 1, N
C
C     Subtract multiples of preceding columns from Ith column of A
C
         DO 10 J = 1, I - 1
            ROWI(J) = -A(I, J)
   10    CONTINUE
         CALL SMXPY (N - I + 1, A(I, I), I - 1, LDA, ROWI, A(I, 1))
C
C     Test for non-positive definite leading minor
C
         IF  (A(I, I) .LE. 0.0) THEN
            INFO = I
            GO TO 40
         ENDIF
C
C     Form Ith column of L
C
      T = 1.0/SQRT(A(I, I))
      A(I, I) = T
      DO 20 J = I + 1, N
         A(J, I) = T * A(J, I)
   20 CONTINUE
   30 CONTINUE
   40 RETURN
      END
```

Annotated listings of subroutines SMXPY and SSDOT

To be noted are the dollar ($) signed variables, formally termed asynchronous variables, in these routines. A key feature of the HEP architecture is the addition of an extra bit in register and data memory locations. This bit when accessed by the $ declaration of a variable, allows

any process to read that variable's allows any process to read that variable's content only if the bit is in the full (1) state, and concurrently blocks other processes from gaining read access by setting the bit to the empty (0) state. Similarly, any process may write into the content of the asynchronous variable only if the bit is in the empty state, and as before blocks other processes from gaining write access by setting the bit to the full state. By such a unique mechanism, synchronization (and in this case self-scheduling) may be achieved with little or no programming effort.

```
      SUBROUTINE SMXPY (N1, Y, N2, LDM, X, M)
C
      INTEGER LDM, N1, N2
      REAL Y(*), X(*), M(LDM, *)
C
C     PURPOSE:
C
C       Form y = y + M * x, where x and x are vectors and
C       M is a matrix.
C
C     PARAMETERS:
C
C       N1      INTEGER,        number of rows in Y and the matrix M.
C       Y       REAL(N1),       vector to accumulate the product M * x.
C       N2      INTEGER,        number of rows in X and columns in the matrix M.
C       LDM     INTEGER,        leading dimension of the array M.
C       X       REAL (N2),      vector used to form y = y + M * x.
C       M       REAL (N1, N2),  matrix used to form y = y + M * x.
C
C     ----------------------------------------------------------------
C
C     This is a parallel version for the HEP
C
Asynchronous variables to be communicated to routine SSDOT
      COMMON /SYNC1/ $NPROC, $DONE, $NROW
C
      IF(N1 .LE. 0 .OR. N2 .LE. 0) RETURN
C
C     Set up for asynchronous operations
C
C     NPROC = number of processors for a task
PURGE sets asynchronous variables to empty state
      PURGE $NPROC, $DONE, $NROW
Initialize variable to have the value 1 and set "ful"
      $NROW = 1
C
Prepare to setup the number of processors
      NCREAT = MIN0(N1, NPROC)
Set the number of processes "full"
      $NPROC = NCREAT
C
C     Doall loop
```

```
C
Spawn multiple copies of this routine to run in parallel
      DO 30 IPROC = 1, NCREAT
        CREATE SSDOT (N1, N2, X, 1, M, LDM, Y)
   30 CONTINUE
C
C     Endall
C     Join and continue serial
C
Wait here until $DONE has been given a value (operation performed in routine SSDOT)
      DONTST = $DONE
      RETURN
      END
      SUBROUTINE SSDOT(N1, N2, X, INCX, M, LDM, Y)
C
      INTEGER N1, N2, INCX, LDM
      REAL X(*), M(LDM, *), Y(*)
C
C     PURPOSE:
C
C     Form one component of y, such that y = y + M * x,
C     where y and x are vectors and M is a matrix.
C
C     PARAMETERS:
C
C       N1      INTEGER,      number of rows in Y and the matrix M
C       N2      INTEGER,      number of rows in X and columns in the matrix M
C       X       REAL(N2),     vector used to form y = y + M * x
C       INCX    INTEGER,      stride used in addressing X
C       M       REAL(N1, N2), matrix used to form y = y + M * x
C       LDM     INTEGER,      leading dimensions of the array M
C       Y       REAL(N1),     vector to accumulate the product M * x
C
C     ------------------------------------------------------------------
C
C     This is a parallel version for the HEP
C
Asynchronous variables communicated from SMXPY
      COMMON/SYNC1/$NPROC, $DONE, $NROW
C
   10 CONTINUE
C
C       Pick up next row
C
Gain unique row that has to be processed
        I = $NROW
Increment to next row to be processed
        $NROW = I + 1
Check if reached the end
        IF(I. GT. N1) GO TO 30
```

```
C
Perform inner product with row i of matrix
      DO 20 J = 1, N2
         Y(I) = (Y(I)) + X(J) * M(I, J)
   20 CONTINUE
C
Get a new row to perform inner product
      GO TO 10
   30 CONTINUE
C
C     Terminate process and check if finished
C
Uniquely decrement the number of processes active
      NACTPR = $NPROC - 1
      $NPROC = NACTPR
Check to see if finished, if so set $DONE "full"
      IF(NACTPR .EQ. 0) $DONE = 1.0
C
      RETURN
      END
```

References

[1] J.J. Dongarra and S.C. Eisenstat, Squeezing the most out of an algorithm in CRAY FORTRAN, *ACM Trans. Math. Software* **3** (1984).

[2] *HEP Fortran 77 User's Guide,* Denelcor Inc., Aurora, CO, 1982.

[3] H.F. Jordan, Experience with pipelined multiple instruction streams, *Proc. of the IEEE* **72** (1) (1984).

[4] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, Basic linear algebra subprograms for FORTRAN usage, *ACM Trans. Math. Software* **5** (1979) 308–371.

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