Dense linear algebra on high-performance computers

Jeremy Du Croz
NAG Ltd., England

1 The LAPACK project

Can we provide portable software for computations in dense linear algebra which is efficient on a wide range of modern high-performance computers? If so, how? Answering these questions — and providing the desired software — is the goal of the LAPACK project, on which this paper will focus. For a wider survey of dense linear algebra on high-performance computers we refer to [9].

In more detail the aims of the LAPACK project are to develop a library of Fortran 77 subroutines which will supersede Linpack and Eispack. It will cover the solution of systems of linear equations, linear least squares problems, and eigenvalue problems — and the usual related computations such as matrix factorizations, orthogonal transformations, and estimating condition numbers or error bounds. More information about LAPACK is given in [1]. The first public release of LAPACK software is currently planned for March/April 1991.

LAPACK aims to improve on Linpack and Eispack in several ways:

- a consistent software design, integrating Linpack and Eispack into a single package
- increased functionality
- new or improved numerical algorithms
- efficiency on high performance computers

This paper will emphasize the last of those topics—*performance.*

2 Factors which affect performance

LAPACK was originally targeted to achieve good performance on single-processor vector machines and on shared-memory multi-processor machines with a modest number of powerful processors (such as an Alliant FX/8, a Convex C240 or a Cray YMP). Since the start of the project, a new class of machines has emerged for which LAPACK software is well-suited—the high-performance workstations, such as the IBM RISC/6000 or machines based on an Intel i860. (LAPACK is intended to be used across the whole spectrum of modern computers, but when considering performance, the emphasis is on machines at the more powerful end of the spectrum.)

Here we discuss the main factors which affect the performance of linear algebra software on these classes of machines.

2.1 Vectorization

Designing vectorizable algorithms in linear algebra is usually staightforward; indeed for many computations there are several variants, all vectorizable, but with different characteristics in performance (see, for example, [5]). Linear algebra algorithms can come close to the peak performance of many machines — principally because peak performance depends on some form of chaining of vector addition and multiplication operations, and this is just what the algorithms require.

However, when the algorithms are realized in straightforward Fortran 77 code, the performance may fall well short of the expected level, usually because vectorizing Fortran compilers fail to minimize the number of memory references — that is, the number of vector load and store operations. This brings us to the next factor.

2.2 Data movement

What often limits the actual performance of a vector—or scalar— floating-point unit, is the rate of transfer of data between different levels of memory in the machine. Examples are: the transfer of vector operands in and out of vector registers; the transfer of scalar operands in and out of a high-speed scalar processor; the movement of data between main memory and a high-speed cache or local memory; or paging between actual memory and disc storage in a virtual memory system.

It is desirable to maximise the ratio of floating-point operations to memory references, and to re-use data as much as possible while it is stored in the higher levels of the memory hierarchy (for example, vector registers or high-speed cache).

A Fortran programmer has no explicit control over these types of data movement.

2.3 Parallelism

The nested loop structure of most linear algebra algorithms offers considerable scope for fine-grained loop-based parallelism on shared-memory machines. This is the principal type of parallelism that LAPACK at present aims to exploit, but we return to this question in Sections 5.4 and 6.

This type of parallel processing can sometimes be generated automatically by a compiler, but often requires the insertion of compiler directives.

3 The BLAS as a portability base

How then can we hope to be able to achieve sufficient control over vectorization, data movement and parallelism in portable Fortran code, to obtain the levels of performance that machines can offer?

The LAPACK strategy for combining efficiency with portability is to construct the software as much as possible out of calls to the BLAS (Basic Linear Algebra Subprograms); the BLAS are used as building-blocks.

The efficiency of LAPACK software depends on efficient implementations of the BLAS being

provided by computer vendors (or others) for their machines. Such implementations are already provided, for example, by Alliant, Convex, Cray, IBM (incomplete) and Siemens. Thus the BLAS form a portability base for LAPACK. Above this level, all the LAPACK software (with a few exceptions mentioned in Section 5.4) is truly portable.

There are now three levels of BLAS:

Level 1 BLAS: for vector operations, such as $y \leftarrow \alpha x + y$ [11]

Level 2 BLAS: for matrix-vector operations, such as $y \leftarrow \alpha Ax + \beta y$ [4]

Level 3 BLAS: for matrix-matrix operations, such as $C \leftarrow \alpha AB + \beta C$ [3]

Here, A, B and C are matrices, x and y are vectors and α and β are scalars.

The Level 1 BLAS are used in LAPACK, but for convenience rather than for performance: they perform an insignificant fraction of the computation, and they cannot achieve high efficiency on most modern supercomputers.

The Level 2 BLAS can achieve near-peak performance on many vector-processors, such as a single processor of a Cray XMP or YMP, or Convex C2 machine. However on other vector processors, such as a Cray 2 or an IBM 3090 VF, their performance is limited by the rate of data movement between different levels of memory. This limitation is overcome by the Level 3 BLAS, which perform $O(n^3)$ floating-point operations on $O(n^2)$ data, whereas the Level 2 BLAS perform only $O(n^2)$ operations.

The BLAS also allow us to exploit parallelism in a way that is transparent to the overlying software. Even the Level 2 BLAS offer some scope for exploiting parallelism, but greater scope is provided by the Level 3 BLAS, as Table 1 illustrates.

Table 1: Speed of Level 2 and Level 3 BLAS operations on a Cray YMP

(all matrices are of order 500; U is upper triangular)

Number of processors:	1	2	4	8
Level 2: $y \leftarrow \alpha Ax + \beta y$	311	611	1197	2285
Level 3: $C \leftarrow \alpha AB + \beta C$	312	623	1247	2425
Level 2: $x \leftarrow Ux$	293	544	898	1613
Level 3: $B \leftarrow UB$	310	620	1240	2425
Level 2: $x \leftarrow U^{-1}x$	272	374	479	584
Level 3: $B \leftarrow U^{-1}B$	309	618	1235	2398

4 Block algorithms and their derivation

It is comparatively straightforward to recode many of the algorithms in Linpack and Eispack so that they call Level 2 BLAS. Indeed in the simplest cases the same floating-point operations are performed, possibly even in the same order: it is just a matter of changing the software. To illustrate this point, here is the body of the code of the Linpack routine SPOFA, which factorizes a symmetric positive-definite matrix as U^TU:

```
      DO 30 J = 1, N
         INFO = J
         S = 0.0E0
         JM1 = J - 1
         IF (JM1 .LT. 1) GO TO 20
         DO 10 K = 1, JM1
            T = A(K,J) - SDOT(K-1,A(1,K),1,A(1,J),1)
            T = T/A(K,K)
            A(K,J) = T
            S = S + T*T
10       CONTINUE
20       CONTINUE
         S = A(J,J) - S
C     ......EXIT
         IF (S .LE. 0.0E0) GO TO 40
         A(J,J) = SQRT(S)
30    CONTINUE
```

And here is the same computation recoded in LAPACK style to use the Level 2 BLAS routine STRSV (which solves a triangular system of equations). The call to STRSV has replaced the loop over K which made several calls to the Level 1 BLAS routine SDOT.

```
      DO 10 J = 1, N
         CALL STRSV( 'Upper', 'Transpose', 'Non-unit', J-1, A, LDA,
     $               A(1,J), 1 )
         S = A(J,J) - SDOT( J-1, A(1,J), 1, A(1,J), 1 )
         IF( S.LE.ZERO ) GO TO 20
         A(J,J) = SQRT( S )
   10 CONTINUE
```

This change by itself is sufficient to make big gains in performance on a number of machines—for example, from 72 to 251 megaflops for a matrix of order 500 on one processor of a Cray YMP. Since this is 81% of the peak speed of matrix-matrix multipliation on this processor, we cannot hope to do very much better by using Level 3 BLAS.

On an IBM 3090E VF (using double precision) there is virtually no difference in performance between the Linpack and the LAPACK-style code. Both run at about 23 megaflops. This is unsatisfactory on a machine on which matrix-matrix multiplication can run at 75 megaflops. To exploit the faster speed of Level 3 BLAS, the algorithms must undergo a deeper level of restructuring, and be re-cast as a *block algorithm* — that is, an algorithm which operates on *blocks* or submatrices of the original matrix.

To derive a block form of Cholesky factorization, we write the defining equation in partitioned form thus:

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{12}^T & A_{22} & A_{23} \\ A_{13}^T & A_{23}^T & A_{33} \end{pmatrix} = \begin{pmatrix} U_{11}^T & 0 & 0 \\ U_{12}^T & U_{22}^T & 0 \\ U_{13}^T & U_{23}^T & U_{33}^T \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{pmatrix}$$

Equating submatrices in the second column, we obtain:

$$\begin{aligned} A_{12} &= U_{11}^T U_{12} \\ A_{22} &= U_{12}^T U_{12} + U_{22}^T U_{22} \end{aligned}$$

Hence, if U_{11} has already been computed, we can compute U_{12} from

$$U_{12} = (U_{11}^T)^{-1} A_{12}$$

by a call to the Level 3 BLAS routine STRSM; and then we can compute U_{22} from

$$U_{22}^T U_{22} = A_{22} - U_{12}^T U_{12}$$

This involves first updating the symmetric submatrix A_{22} by a call to the Level 3 BLAS routine SSYRK, and then computing its Cholesky factorization; since Fortran does not allow recursion, a separate routine must be called (using Level 2 BLAS rather than Level 3), named SUTU2 in the code below. In this way successive blocks of columns of U are computed. In the code NB denotes the width of the blocks.

```
      DO 10 J = 1, N, NB
         JB = MIN( NB, N-J+1 )
         CALL STRSM( 'Left', 'Upper', 'Transpose', 'Non-unit', J-1, JB,
     $               ONE, A, LDA, A(1,J), LDA )
         CALL SSYRK( 'Upper', 'Transpose', JB, J-1, -ONE, A(1,J), LDA,
     $               ONE, A(J,J), LDA )
         CALL SUTU2( JB, A(J,J), LDA, INFO )
         IF( INFO.NE.0 ) GO TO 20
   10 CONTINUE
```

This code runs at 49 megaflops on a 3090, more than double the speed of the Linpack code. But that is not the end of the story.

We mentioned in Section 2.1 that for many linear algebra computations there are several vectorizable variants, often referred to as i-, j- and k-variants, according to a convention introduced in [5] and used in [10]. The same is true of the corresponding block algorithms.

It turns out that the j-variant that was chosen for Linpack, and used in the above examples, is not the fastest on many machines, because it is based on solving triangular systems of equations, which can be significantly slower than matrix-matrix multiplication. The fastest variant on a 3090 is the k-variant, which achieves 57 megaflops.

On a Cray YMP, the use of Level 3 BLAS squeezes a little more performance out of one processor, but makes a large improvement when using all 8 processors. Table 2 summarises the results.

Table 2: Speed of Cholesky factorization $A = U^T U$ for $n = 500$

Machine:	IBM 3090 VF	Cray YMP	Cray YMP
Number of processors:	1	1	8
j-variant: Linpack	23	72	72
j-variant: using Level 2 BLAS	24	251	378
j-variant: using Level 3 BLAS	49	287	1225
k-variant: using Level 3 BLAS	57	287	1438

5 Survey of block algorithms in LAPACK

Having discussed in detail the derivation of one particular block algorthm, we now describe some of the results that have been obtained so far with a variety of block algorthms. Note that all the results quoted in this paper are *preliminary* — the code of LAPACK routines has not been finalized at the time of writing.

5.1 Factorizations for solving linear equations

The well-known LU and Cholesky factorizations are the simplest block algorithms to derive. No extra floating-point operations are required, nor is any extra working storage. The same is true of the associated routines for matrix inversion, but they are much less frequently needed.

Table 3 illustrates the speed of the LAPACK routine for LU factorization, SGETRF in single precision on Cray machines, DGETRF in double precision on all other machines. A block size of 1 means that the unblocked algorithm is used, since it is faster than — or at least as fast as — a blocked algorithm.

Table 3: Speed of SGETRF/DGETRF for square matrices of order n

	No. of processors	Block size	Values of n				
			100	200	300	400	500
IBM RISC/6000	1	32	19	25	29	31	33
Alliant FX/8	8	16	9	26	32	46	57
IBM 3090J VF	1	64	23	41	52	58	63
Convex C240	4	64	31	60	82	100	112
Cray YMP	1	1	132	219	254	272	283
Cray 2	1	64	110	211	292	318	358
Siemens/Fujitsu VP 400-EX	1	64	46	132	222	309	397
NEC SX2	1	1	118	274	412	504	577
Cray YMP	8	64	195	556	920	1188	1408

Table 4 gives similar results for Cholesky factorization, extending the results given in Table 2.

For symmetric indefinite matrices, LAPACK, like Linpack, provides the Bunch-Kaufman algorithm, factorizing A as $PUDU^TP^T$, where P is a permutation matrix, and D is block diagonal with blocks of order 1 or 2. A block form of this algorithm has been derived by Sorensen, and is implemented in the LAPACK routine SSYTRF/DSYTRF. It has to duplicate a little of the computation in order to "look ahead" to determine the necessary row and column interchanges. But the extra work is more than compensated for by the faster speed of updating the matrix by blocks, as is illustrated in Table 5.

LAPACK, like Linpack, will provide LU and Cholesky factorizations of band matrices. The Linpack algorithms can easily be restructured to use Level 2 BLAS, though that has little effect on performance for matrices of very narrow bandwidth. It is also possible to use Level 3 BLAS, at the price of doing some extra work with zero elements outside the band [8]. This become worth while for matrices of large order and semi-bandwidth greater than 100 or so.

Table 4: Speed of SPOTRF/DPOTRF for matrices of order n with UPLO = 'U'

	No. of processors	Block size	Values of n				
			100	200	300	400	500
IBM RISC/6000-530	1	32	21	29	34	36	38
Alliant FX/8	8	16	10	27	40	49	52
IBM 3090J VF	1	48	26	43	56	62	67
Convex C240	4	64	32	63	82	96	103
Cray YMP	1	1	126	219	257	275	285
Cray 2	1	64	109	213	294	318	362
Siemens/Fujitsu VP 400-EX	1	1	53	145	237	312	369
NEC SX2	1	1	155	387	589	719	819
Cray YMP	8	32	146	479	845	1164	1393

Table 5: Speed of SSYTRF for matrices of order n with UPLO = 'U' on a Cray 2

Block size	Values of n				
	100	200	300	400	500
1	75	128	154	164	176
64	78	160	213	249	281

5.2 QR factorization

The traditional algorithm for QR factorization is based on the use of elementary Householder matrices, of the general form

$$H = I - \tau uu^T$$

This leads to an algorithm with very good vector performance, especially if coded to use Level 2 BLAS.

The key to developing a block form of this algorithm is to represent a product of b elementary Householder matrices of order n as a block form of Householder matrix [12]:

$$H_1 H_2 \ldots H_b = I - UTU^T$$

where U is an n-by-b matrix whose columns are the individual vectors $u_1, u_2, \ldots, u_b$, and T is an upper triangular matrix of order b. Extra work is required to compute the elements of T, but once again this is compensated for by the faster speed of applying the block form. Table 6 summarises results obtained with the LAPACK routine SGEQRF/DGEQRF.

5.3 Eigenvalue problems

Eigenvalue problems have so far provided a less fertile ground for the development of block algorithms than the factorizations so far described. Nevertheless useful improvements in performance have been obtained.

The first step in solving many types of eigenvalue problem is to reduce the original matrix to a "condensed form" by orthogonal transformations.

Table 6: Speed of SGEQRF/DGEQRF for square matrices of order n

	No. of processors	Block size	Values of n				
			100	200	300	400	500
IBM RISC/6000-530	1	32	18	26	30	32	34
Alliant FX/8	8	16	11	28	39	47	50
IBM 3090J VF	1	32	28	54	68	75	80
Convex C240	4	16	35	65	82	97	106
Cray YMP	1	1	177	253	276	286	292
Cray 2	1	32	105	208	269	303	326
Siemens/Fujitsu VP 400-EX	1	1	101	237	329	388	426
NEC SX2	1	1	217	498	617	690	768

As for QR factorization, the unblocked algorithms all use elementary Housholder matrices, and have good vector performance. Block forms of these algorithms have been developed [6], but all require additional operations, and a significant proportion of the work must still be performed by Level 2 BLAS, so there is less possibility of compensating for the extra operations.

The algorithms concerned are:

- reduction of a symmetric matrix to tridiagonal form, to solve a symmetric eigenvalue problem: LAPACK routine SSYTRD applies a symmetric block update of the form $A \leftarrow A - UX^T - XU^T$, using the Level 3 BLAS routine SSYR2K; Level 3 BLAS account for at most half the work.
- reduction of a rectangular matrix to bidiagonal form, to compute a singular value decomposition: LAPACK routine SGEBRD applies a block update of the form $A \leftarrow A - UX^T - YV^T$, using two calls to the Level 3 BLAS routine SGEMM; Level 3 BLAS account for at most half the work.
- reduction of a nonsymmetric matrix to Hessenberg form, to solve a nonsymmetric eigenvalue problem: LAPACK routine SGEHRD applies a block update of the form $A \leftarrow (I - UT^TU^T)(A - XU^T)$; Level 3 BLAS account for at most three-quarters of the work.

Note that only in the reduction to Hessenberg form is it possible to use the block Householder representation, described in subsection 5.2. Extra work must be performed to compute the n-by-b matrices X and Y that are required for the block updates — and extra workspace is needed to store them.

Nevertheless the performance gains can be worthwhile on some machines, for example, on an IBM 3090, as shown in Table 7.

Following the reduction to condensed form, there is no scope for using Level 2 or Level 3 BLAS in computing the eigenvalues and eigenvectors of a symmetric tridiagonal matrix, or in computing the singular values and vectors of a bidiagonal matrix.

However, for computing the eigenvalues and eigenvectors of a Hessenberg matrix—or rather for computing its Schur factorization— yet another flavour of block algorithm has been developed: a *multishift* QR iteration [2]. Whereas the traditional Eispack routine HQR

Table 7: Speed of reductions to condensed forms on an IBM 3090E VF

(all matrices are square of order n)

	Block size	Values of n			
		128	256	384	512
SSYTRD	1	15	22	26	27
	16	15	26	32	34
SGEBRD	1	23	26	28	29
	12	23	33	38	41
SGEHRD	1	27	29	30	30
	24	36	51	57	58

uses a double shift (and the corresponding complex routine COMQR uses a single shift), the multishift algorithm uses block shifts of higher order. It has been found that the total number of operations *decreases* as the order of shift is increased until a minimum is reached typically between 4 and 8; for higher orders the number of operations increases quite rapidly. Because the speed of applying the shift increases steadily with the order, the optimum order of shift may be in the region 8 to 16, say. (Note that for this algorithm timings are not only machine-dependent, but also data-dependent, because they depend on the pattern of iterations.) The overall gain in performance is not spectacular—the total time may be at best halved—but no faster method has yet been developed.

5.4 Other parallelizable algorithms in LAPACK

Although the main thrust of LAPACK has been to develop block algorithms wherever possible, this has not excluded a few other algorithms that promise good scope for vectorization or parallelization.

For example, routines have been developed to apply the method of bisection to compute eigenvalues or singular values, in a way that allows either vectorization or parallelization. And for the symmetric eigenvalue problem, code is being developed for the divide-and-conquer method [7].

For these algorithms the parallelism cannot be hidden in the BLAS, and some other (as yet non-standard) means must be used to express it.

6 The future — LAPACK 2

Recently the NSF has granted funding for a new project — LAPACK 2 — which will extend LAPACK in several directions:

- developing distributed-memory versions of selected routines
- rewriting selected routines to exploit special properties of IEEE arithmetic
- developing C and Fortran 90 versions of selected routines

- extending the scope of LAPACK to include, for example, generalized SVD and new accurate routines for eigenvalue problems based on Jacobi's method
- systematic performace evaluation

On present-day distributed-memory machines, we do not expect to be able to hide the parallelism inside the BLAS. We expect to have to write special code, although preserving as far as possible the basic structure of the block algorithms described above. A set of standard communication routines would be a great advantage, and an attempt to specify them has just begun — under the title BLACS (Basic Linear Algebra Communication Subprograms).

Acknowledgements

The LAPACK project is a collaborative project, involving E. Anderson (University of Tennessee), Z. Bai (University of Kentucky), C. Bischof (Argonne National Laboratory), J.W. Demmel (University of California at Berkeley), J.J. Dongarra (University of Tennessee and Oak Ridge National Laboratory), J.J. Du Croz (NAG Ltd.), A. Greenbaum (New York University), S.J. Hammarling (NAG Ltd.), A. McKenney (New York University), and D.C. Sorensen (Rice University), with contributions from many others. The project is supported by NSF Grant No. ASC-8715728.

References

[1] Anderson, E., Z. Bai, C. Bischof, J.W. Demmel, J.J. Dongarra, J.J. Du Croz, A. Greenbaum, S.J. Hammarling, A. McKenney and D.C. Sorensen, 'LAPACK: a portable linear algebra library for high-performance computers', LAPACK Working Note 20, Computer Science Dept, University of Tennessee, Knoxville, report CS-90-105, May 1990.

[2] Bai, Z. and J.W. Demmel, 'On a block implementation of Hessenberg multishift QR iteration', Int. J. High Speed Computing, 1, pp. 97–112, 1989.

[3] Dongarra, J.J., J.J. Du Croz, I.S.Duff and S.J. Hammarling, 'A set of Level 3 Basic Linear Algebra Subprograms', ACM Trans. Math. Software, 16, pp. 1–17, 1990.

[4] Dongarra, J.J., J.J. Du Croz, S.J. Hammarling and R.J. Hanson, 'An extended set of Fortran Basic Linear Algebra Subprograms', ACM Trans. Math. Software, 14, pp. 1–17, 1988.

[5] Dongarra, J.J., F. Gustavson and A. Karp, 'Implementing linear algebra algorithms for dense matrices on a vector pipeline machine', SIAM Rev., 26, pp. 91–112, 1984.

[6] Dongarra, J.J., S.J. Hammarling and D.C. Sorensen, 'Block reduction of matrices to condensed forms for eigenvalue computations', J. Comp. Appl. Math. 27, pp. 215–227, 1989.

[7] Dongarra, J.J. and D.C. Sorensen, 'A fully parallel algorithm for the symmetric eigenvalue problem', SIAM J. Sci. Stat. Comp. 8, pp. s139-s154, 1987.

[8] Du Croz, J.J., P.J.D. Mayes and G. Radicati, 'Factorizations of band matrices using Level 3 BLAS', LAPACK Working Note 21, Computer Science Dept, University of Tennessee, Knoxville, report CS-90-109, July 1990.

[9] Gallivan, K.A., R.J. Plemmons and A.H. Sameh, 'Parallel algorithms for dense linear algebra computations', SIAM Rev., 32, pp. 54–135, 1990.

[10] Golub, G.H. and C.F. Van Loan, 'Matrix Computations', 2nd ed., The Johns Hopkins University Press, 1989.

[11] Lawson, C., R.J. Hanson, D. Kincaid and F.T. Krogh, 'Basic Linear Algebra Subprograms for Fortran usage', ACM Trans. Math. Software, 5, pp. 308–323, 1979.

[12] Schreiber, R., and C.F. Van Loan, 'A storage efficient WY representation for products of Householder transformations', SIAM J. Sci. Stat. Comp. 10, pp. 53–57, 1989.