CONCURRENCY: PRACTICE AND EXPERIENCE, VOL. 3(6), 655–666 (DECEMBER 1991)

LAPACK: a portable linear algebra library for high-performance computers

JAMES DEMMEL
Mathematics Department and Computer Science Division
University of California
Berkeley, CA 94720, USA

SUMMARY

The goal of the LAPACK project is to design and implement a portable linear algebra library for efficient use on a variety of high-performance computers. The library is based on the widely used LINPACK and EISPACK packages for solving linear equations, eigenvalue problems, and linear least-squares problems, but extends their functionality in a number of ways. The major methodology for making the algorithms run faster is to restructure them to perform block matrix operations (e.g. matrix–matrix multiplication) in their inner loops. These block operations may be optimized to exploit the memory hierarchy of a specific architecture. In particular, we discuss algorithms and benchmarks for the singular value decomposition.

1. INTRODUCTION

The University of California at Berkeley, the University of Tennessee, the Courant Institute of Mathematical Sciences, the Numerical Algorithms Group, Ltd, Rice University, Argonne National Laboratory, and Oak Ridge National Laboratory are developing a transportable linear algebra library in Fortran 77. The library is intended to provide a uniform set of subroutines to solve the most common linear algebra problems and to run efficiently on a wide range of high-performance computers.

The LAPACK library (shorthand for Linear Algebra Package) will provide routines for solving systems of simultaneous linear equations, least-squares solutions of over-determined systems of equations, and eigenvalue problems. The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) will also be provided, as will related computations such as reordering of the factorizations and condition numbers (or estimates thereof). Dense and banded matrices will be provided for, but not general sparse matrices. In all areas, similar functionality will be provided for real and complex matrices.

The new library will be based on the successful EISPACK[35,27] and LINPACK[14] libraries, integrating the two sets of algorithms into a unified, systematic library. A great deal of effort has also been expended to incorporate design methodologies and algorithms that make the LAPACK codes more appropriate for today's high-performance architectures. The LINPACK and EISPACK codes were written in a fashion that, for the most part, ignored the cost of data movement. Most of today's high-performance machines, however, incorporate a memory hierarchy[23,29,37] to even out the difference in speed of memory accesses and vectorized floating-point operations. As a result, codes must be careful about reusing data in order not to run at memory speed instead of

1040-3108/91/060655-12$06.00

Received 17 December 1990
Revised 4 June 1991

floating-point speed. LAPACK codes have been carefully restructured to reuse as much data as possible in order to reduce the cost of data movement. Further improvements are the incorporation of new and improved algorithms for the solution of eigenvalue problems[10,20].

LAPACK is designed to be efficient and transportable across a wide range of computing environments, with special emphasis on modern high-performance computers. While we do not hope for LAPACK codes to be optimal for all architectures, we expect high performance over a wide range of machines. By relying on the Basic Linear Algebra Subprograms (BLAS)[22,15,31] the codes can be 'tuned' to a given architecture by efficient—and, in all likelihood machine-dependent—implementations of these kernels. Machine-specific optimizations are limited to those kernels, and the user interface is uniform across machines. We shall also distribute test and timing routines to verify the installation of the LAPACK codes on a particular architecture and to allow for easy comparison with existing software.

In addition to higher speed, LAPACK is designed to provide higher accuracy for a number of problems. We do this by replacing the conventional notion of normwise backward error with componentwise relative backward error. This approach better respects the sparsity and scaling structure of the original problem. This leads to new perturbation theory, algorithms and error analysis for a number of problems, and is discussed in section 5 below.

Netlib[18] has demonstrated how useful and important it is for libraries to be easily available, and preferably on line. We intend to distribute the new library in a similar way, for no cost or a nominal cost only.

The rest of this paper is outlined as follows. Section 2 describes the BLAS and explains why their use can speed up algorithms. Section 3 describes block algorithms and shows in some detail how to reorganize the singular value decomposition (SVD). Section 4 contains benchmark results for the SVD and other routines on a variety of machines. Section 5 outlines our general approach to achieving high accuracy. Section 6 reviews the target machines for which LAPACK is designed to run most efficiently. Finally, Section 7 outlines future plans to extend the library, including the challenges faced in adapting the codes to distributed-memory machines.

In addition to that of the author, this represents work of E. Anderson, Z. Bai, J. Barlow, C. Bischof, P. Deift, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, W. Kahan, L.-C. Li, A. McKenney, D. Sorensen, C. Tomei and K. Veselić.

2. BASIC LINEAR ALGEBRA SUBPROGRAMS

The BLAS were originally introduced in the construction of LINPACK[31]. These BLAS did operations only on vectors of data, such as a dot product or a saxpy (adding a scalar multiple of one vector to another). We refer to these vector–vector operations as Level 1 BLAS. The Level 1 BLAS permit efficient implementation on scalar machines, but the granularity is too low for effective use on most vector or parallel machines.

More recently, higher-level BLAS have been specified that perform operations of higher granularity and so offer more opportunity for optimization on different architectures. The Level 2 BLAS[17] perform matrix–vector operations such as matrix–vector multiplication and rank-one updates. The Level 3 BLAS[16] perform matrix–

matrix operations such as matrix–matrix multiplication, solving triangular systems with multiple right-hand sides, and rank-k matrix updates.

To appreciate why these Level 2 and Level 3 BLAS with larger granularity offer better opportunities for efficiency, one must understand memory hierarchies. All machines (not just supercomputers) have a hierarchy of memory levels—for example, with registers at the top, followed by cache, main memory, and finally disk storage at the bottom. Toward the top of the hierarchy, memory is smaller, more expensive, and faster. Since operations such as multiplication and addition must be done at the top level, data has to move up through the various levels to the top to be processed, and then down again to be stored. The result is that data at higher levels is available only after some delay and (because of memory bank conflicts) may not be available at a rate fast enough to feed the arithmetic units. Clearly, an algorithm that minimizes the memory traffic in the hierarchy will run faster.

One way to measure the amount of this memory traffic is the ratio of flops (floating-point operations) to memory references in an algorithm. The larger this ratio, the longer a piece of data may be kept at the top of the hierarchy on average. Let us use this measure to compare the three operations of saxpy (Level 1 BLAS), matrix–vector multiplication (Level 2 BLAS), and matrix–matrix multiplication (Level 3 BLAS), where all vectors and matrices are of dimension n. Simple counting yields the ratios 2/3 for saxpy, 2 for matrix–vector multiplication, and $n/2$ for matrix–matrix multiplication. The large ratio for matrix–matrix multiply represents a surface-to-volume effect, doing $O(n^3)$ operations on $O(n^2)$ data. Hence, matrix–matrix multiplication offers much greater opportunity for exploiting the memory hierarchy than the lower-level BLAS routines. Table 1 illustrates this fact with some benchmark results.

Table 1. Speed of the BLAS on various architectures (all values are in Mflops)

	Alliant FX/8 (8 processors)	IBM 3090/VF (1 processor)	Cray 2S (1 processor)
Peak speed	94	108	488
Level 1 BLAS	14	26	121
Level 2 BLAS	26	60	350
Level 3 BLAS	43	80	437

Fortran implementations of all the BLAS are available; to get the full benefit, however, the BLAS should be optimized for each architecture. We encourage the computer manufacturers to perform these optimizations; the data in Table 1 are for such optimized implementations. We also expect that the LAPACK project will reveal the need for a few additional basic routines whose performance may need to be optimized for different architectures and may be regarded as extensions to the current set of BLAS (e.g. applying a sequence of plane rotations to a matrix).

3. BLOCK ALGORITHMS

To exploit the Level 3 BLAS, one usually must express the algorithm in terms of operations on submatrices, or 'blocks', as compared to vector- or scalar-oriented operations. We have developed such block routines for Gaussian elimination

and Cholesky, QR decomposition (with and without pivoting), the nonsymmetric eigenproblem (both reduction to Hessenberg form and QR iteration), and the symmetric eigenproblem (reduction to tridiagonal form). Work is continuing on block algorithms for the SVD and generalized eigenproblems. See Reference 19 for details. A good survey of block algorithms is in Reference 26. In this section we discuss the SVD in detail.

The singular value decomposition of an m by n real matrix A is a factorization $A = U\Sigma V^T$, where U is m by m and orthogonal, V is n by n and orthogonal, and $\Sigma = \text{diag}(\sigma_1,\ldots,\sigma_s)$ is m by n and diagonal. Here $s = \min(m,n)$ and $\sigma_1 \geq \cdots \geq \sigma_s \geq 0$. The σ_i are called *singular values*, the columns of V the *right singular vectors*, and the columns of U the *left singular vectors*. A similar, somewhat cheaper and more compact version is to take V_1 the first s columns of V, U_1 the first s columns of U, and Σ_1 the top left s by s corner of Σ so $A = U_1\Sigma_1 V_1^T$.

If m and n do not differ too much in size (are within a factor of, say, 5/3 of one another[28]), then the SVD is computed in two stages as follows. We assume without loss of generality that $m \geq n$. First, we compute orthogonal matrices U_2 and V_2 so that $B = U_2^T A V_2$ where B is s by s and *bidiagonal*, i.e. nonzero only on the main diagonal and on the first superdiagonal. This can be done in a finite number of steps as described below, and requires about $8n^3/3$ flops if A is square.

The second stage computes orthogonal U_3 and V_3 so that $\Sigma_1 = U_3^T B V_3$. This is an iterative scheme which stops when the computed Σ_1 is close enough to diagonal. Altogether we get $A = (U_2U_3)\Sigma_1(V_2V_3)^T$, the SVD. The first stage of the algorithm may be rewritten in terms of the Level 2 and 3 BLAS; we describe this below. The second stage is harder to vectorize or parallelize, in particular if one is only interested in computing Σ. Our main algorithm is sequential, and described briefly at the end of Section 5. If only Σ is desired, this part of the algorithm takes $O(n^2)$ flops. Thus, it is asymptotically negligible compared to the first stage. However, as we shall see in Section 4, our current inability to speed up stage two means $O(n^2)$ can dominate $O(n^3)$ for surprisingly large n.

For the rest of this section, we describe the Level 2 and 3 BLAS versions of the reduction to bidiagonal form.

3.1. Reduction to bidiagonal form using Level 2 BLAS

We begin by describing the conventional algorithm, and then show how to rewrite it using Level 2 BLAS. Our basic tool is the *Householder reflection* or *Householder matrix*, an orthogonal matrix of the form

$$H_u = I - 2uu^T \qquad \|u\|_2 = 1 \tag{1}$$

Given a vector x of size n, it costs just $O(n)$ work to compute a vector u such that $H_u x = x - 2u(u^T x)$ has zeros in entries 2 through n. More generally, one can compute a vector u which is nonzero in entries k through n such that Hx is zero in entries $k+1$ through n. We can use this to reduce A to bidiagonal form as follows.

At the first step we choose u_1 so that the first column of $H_{u_1}A$ is zero below the diagonal, and then v_1 so that the first row of $H_{u_1}AH_{v_1}$ is zero in columns 3 through n. This leaves the zeros in column 1 unchanged.

At the beginning of step i we have pre- and postmultiplied A by $i-1$ Householder

matrices to get $A_{i-1} = H_{u_{i-1}} \cdots H_{u_1} A H_{v_1} \cdots H_{v_{i-1}}$, with zeros below the diagonal in columns 1 through $i-1$ and to the right of the first superdiagonal in rows 1 through $i-1$. In step i we choose u_i to zero out below the diagonal in column i of $H_{u_i} A_{i-1}$ and then v_i to zero out row i in columns $i+2$ through n in $H_{u_i} A_{i-1} H_{v_i}$. Both these multiplications leave previously created zeros unchanged. Finally, at step $n-1$, we need only premultiply by $H_{u_{n-1}}$.

The basic operation is therefor pre- and postmultiplying a matrix A by Householder transformations H_u and H_v, respectively. This can be done easily with Level 2 BLAS as follows:

(1) Compute u from the first column of A.
(2) $x^T = 2u^T \cdot A$ (matrix–vector multiply).
(3) $A = A - u \cdot x^T$ (rank-1 matrix update).
(4) Compute v from the first row of A.
(5) $y = 2A \cdot v$ (matrix–vector multiply).
(6) $A = A - y \cdot v^T$ (rank-1 matrix update).

The entries of the bidiagonal matrix are byproducts of the computations of u and v. This algorithm is performed by subroutine SGEBD2 in LAPACK.

3.2. Reduction to bidiagonal form using Level 3 BLAS

We now describe how the above algorithm can be changed to use matrix–matrix operations. Briefly, in order to update the first b rows and columns of A, all of A must be read from memory, but only the first b rows and columns written. Thus we may reduce the first b rows and columns of A to bidiagonal form, accumulating the information we need to update the rest of the matrix in m by b work matrices U and Y, and n by b work matrices V and X. Then, using matrix–matrix multiplication, we can compute $A_b = A - UX^T - YV^T$. These two rank-b updates replace the $2b$ rank-1 updates $A - ux^T - yv^T$ in the Level 2 BLAS algorithm.

We describe how U, Y, X and X are computed. The columns of U, V, X and Y will just be the vectors u_i, v_i, x_i and y_i computed by the Level 2 algorithm. Let $U_i = [u_1, \ldots, u_i]$, and define V_i, X_i and Y_i similarly. We wish to compute these four matrices for $i = b$. Assuming U_i, V_i, X_i and Y_i have been computed, we show how to compute U_{i+1}, V_{i+1}, X_{i+1}, and Y_{i+1}. By assumption A_i, if we were to compute it, would be $A - U_i X_i^T - Y_i V_i^T$. We use the notation $A(:, k)$ to mean the kth column of A and $A(k, :)$ to mean the kth row of A. We compute as follows:

(1) Compute the $i+1$st column of A_i: $z = A(:, i+1) - U_i \cdot (X_i(i+1, :))^T - Y_i \cdot (V_i(i+1, :))^T$ (two matrix–vector multiplies).
(2) Compute u_{i+1} from z.
(3) $x_{i+1}^T = 2[u_{i+1}^T \cdot A - (u_{i+1}^T \cdot U_i) \cdot X_i^T - (u_{i+1}^T \cdot Y_i) \cdot V_i^T]$ (five matrix–vector multiplies).
(4) Compute the $i+1$st row of A_i: $w = A(i+1, :) - U_i(i+1, :) \cdot X_i^T - Y_i(i+1, :) \cdot V_i^T$ (two matrix–vector multiplies).
(5) Compute v_{i+1} from w.
(6) $y_{i+1} = 2[A \cdot v_{i+1} - U_i \cdot (X_i^T \cdot v_{i+1}) - Y_i \cdot (V_i^T \cdot v_{i+1})$ (five matrix–vector multiplies).

When we have computed U_b, V_b, X_b and Y_b, we update $A = A - U_b \cdot X_b^T - Y_b \cdot V_b^T$ with two matrix–matrix multiplications. Then we repeat the process on the remainder of A. Note that we access all of A only twice in the displayed algorithm, to compute $u_{i+1}^T \cdot A$ and $A \cdot v_{i+1}$. The other matrix–vector multiplies deal with smaller matrices.

The need to choose a block size b occurs throughout LAPACK. The optimal b depends on the algorithm, matrix dimension, and machine architecture. Furthermore, on multiprocessor machines, possibly conflicting issues of individual processor performance and overall load balancing must be reconciled. A discussion of these issues and a suggestion for a methodology to overcome this problem can be found in Reference 6. Determining optimal, or near optimal, block sizes for different environments is a major research topic for the LAPACK project.

4. BENCHMARKS

The first version of LAPACK software was released for beta-testing in April 1989. This software included software for general, positive definite, and symmetric indefinite systems and for QR decomposition without pivoting.

In Tables 2–4 we present results for which most or all of the BLAS were optimized for the particular architecture. SGETRF is the LAPACK routine for triangular factorization of a general matrix with partial pivoting, SPOTRF performs Cholesky factorization of a positive definite symmetric matrix, and SGEQRF does QR factorization without pivoting. Also shown are SGEMV (matrix–vector multiply) and SGEMM (matrix–matrix multiply), since these are the 'speed limits' for the algorithms written in terms of the Level 2 BLAS and Level 3 BLAS, respectively. All codes were run in single precision (32 bits on the Convex and 64 bits on the Cray). All results are in Mflops.

Table 2 gives the results for the Convex C210, with an algorithm block size of $n_b = 1$. As the table shows, there is no difference between the Level 2 and Level 3 BLAS versions. Since matrix–vector and matrix–matrix multiply are equally fast on this machine in their current implementations, nothing is gained by going to the Level 3 BLAS.

Tables 3 and 4 give results for the Cray Y-MP for one and eight processors, respectively. Here $n_b = 64$ for SGETRF and SPOTRF and $n_b = 16$ for SGEQRF. The maximum speed of a single processor of a Cray Y-MP is 333 Mflops. Thus, we see that for large-enough matrix dimensions, the single-processor code runs at at 90% efficiency. When all eight processors are used, the code attains 73% to 80% efficiency.

We conclude with some preliminary benchmarks for the SVD code, and compare it to the LINPACK routine SSVDC (see Table 5). We break the LAPACK times down into two parts: reduction to bidiagonal form using Level 2 BLAS (subroutine SGEBD2) and computing the singular values of a bidiagonal matrix (subroutine SBDSQR). As mentioned in Section 3, SGEBD2 performs about $8n^3/3$ flops and SBDSQR only $O(n^2)$. The LINPACK routine SSVDC performs both bidiagonal reduction and bidiagonal SVD, so we only have one set of data for it.

These tests were done on a single processor of a Cray Y-MP. The rows labeled 'Total' mean total LAPACK time or megaflops. The 'Speed-up' is of LAPACK over LINPACK. The most interesting data is in Table 6. SGEBD2 runs over 90 times faster than SBDSQR for $n = 400$. So even though SGEBD2 is performing vastly more flops that SBDSQR, it takes 0.6 seconds compared to SBDSQR's 1.4 seconds. Extrapolating SBDSQR's time to grow proportionally to n^2 and SGEBD2's to n^3, we expect them to take the same

Table 2. LAPACK on a Convex C210

Routine	Matrix dimension				
	32	64	128	256	512
SGEMV	34	43	47	47	47
SGEMM	38	44	47	47	47
SGETRF	6	12	21	30	36
SPOTRF	8	20	33	40	44
SGEQRF	12	21	27	33	38

Table 3. LAPACK on a Cray Y-MP, one processor

Routine	Matrix dimension					
	32	64	128	256	512	1024
SGETRF	40	108	195	260	290	304
SPOTRF	34	95	188	259	289	301
SGEQRF	54	139	225	275	294	301

Table 4. LAPACK on a CRAY Y-MP, eight processors

Routine	Matrix dimension					
	32	64	128	256	512	1024
SGETRF	32	90	205	375	1039	1974
SPOTRF	29	84	273	779	1592	2115
SGEQRF	50	133	328	807	1476	1937

Table 5. SVD (seconds) on a Cray Y-MP, one processor

Routine	Matrix dimension		
	100 × 100	200 × 200	400 × 400
SGEBD2	0.014	0.08	0.60
SBDSQR	0.096	0.36	1.40
Total	0.110	0.44	2.00
SSVDC	0.140	0.58	2.60
Speed-up	1.3	1.3	1.3

Table 6. SVD (Mflops) on a Cray Y-MP, one processor

Routine	Matrix dimension		
	100 × 100	200 × 200	400 × 400
SGEBD2	200	260	280
SBDSQR	3	3	3
Total	27	52	88
SSVDC	22	39	68

time at about $n = 1000$. Thus unless we can significantly speed up the $O(n^2)$ part of the computation, it will dominate the $O(n^3)$ part for rather large n.

For complex data the speed-up of LAPACK over LINPACK improved to 1.6. For $n = 400$, going from real to complex data slowed LAPACK down by a factor of 1.5 and slowed LINPACK down by a factor of 1.9.

We expect further improvements when we benchmark our Level 3 BLAS code for bidiagonal reduction.

5. HIGH-ACCURACY LINEAR ALGEBRA ALGORITHMS

One objective of the LAPACK project is to provide linear algebra algorithms of extremely high accuracy. To discuss the new algorithms, we shall need some notation.

We let H denote the problem for which we seek a solution for some problem; we denote the solution by $f(H)$. For example, $f(H)$ may denote the eigenvalues, eigenvectors, singular values, or singular vectors of the matrix H. If H denotes the pair (A,b), then $f(H)$ may denote the solution of the linear system $Ax = b$, perhaps in a least-squares sense if A is singular or not square. In general, $f(H)$ cannot be computed exactly and hence is approximated by an algorithm whose output we denote $\hat{f}(H)$. We also let ε denote the machine precision.

Analyzing the accuracy of an algorithm $\hat{f}$ for f consists of two parts. First, we use *perturbation theory*, where we bound the difference $f(H + \delta H) - f(H)$ in terms of δH. This part depends only on f and not the algorithm that approximates it. Second, we use *error analysis*, which attempts to show that the computed solution $\hat{f}(H)$ is close to $f(H + \delta H)$ for some bounded δH. Showing that $\hat{f}(H) = f(H + \delta H)$ for some bounded δH is called *backward error analysis*, but is by no means the only way to proceed.

There is a great deal of choice in the measures we choose to bound errors and measure distances. In conventional error analysis as introduced by Wilkinson, we bound $\|f(H + \delta H) - f(H)\|$ in terms of $\|\delta H\|$, and show $\hat{f}(H) = f(H + \delta H)$ where $\|\delta H\| \leq O(\varepsilon)\|H\|$. Here, $\|\cdot\|$ denotes a norm, like the one-norm or Frobenius norm. Typically one proves a formula of the form $\|f(H + \delta H) - f(H)\| \leq \kappa(f,H) \cdot \|\delta H\| + O(\|\delta H\|^2)$, where $\kappa(H)$ is called the *condition number of H with respect to f*. In this formulation, it is easy to see that $\kappa(f,H)$ is simply the norm of the gradient of f at H: $\|\nabla f(H)\|$; other scalings are possible. Thus, combining the perturbation theory and error analysis, one can write

$$\|f(H + \delta H) - f(H)\| \leq O(\varepsilon)\kappa(f,H) \cdot \|H\| + O(\varepsilon^2)$$

The drawback of this approach is that it does not respect the structure of the original data. In particular, if the original data is sparse or graded (large in some entries, small in others), bounding δH only by norm can give very pessimistic results. A trivial example is solving a diagonal system of equations. Each component of the solution is computed to full accuracy by a single divide operation, but the conventional condition number is the ratio of the largest to smallest diagonal entries and may be arbitrarily large.

Instead of bounding δH by its norm $\|\delta H\|$, one may instead use the measure $rel_H(\delta H) \equiv \max_{ij} |\delta H_{ij}|/|H_{ij}|$, the largest relative change in any entry (we use the notation rel_H to indicate the dependence on H). This measure respects sparsity, since δH_{ij} must be zero if H_{ij} is zero, and also grading, since every entry is perturbed by an

amount small compared to its magnitude. For example, in the case of diagonal linear equation solving, one can easily see that a perturbation δH of size $rel_H(\delta H)$ in the matrix can only change the solution relatively by $rel_H(\delta H)$ in each component, and that the algorithm is backward stable with $rel_H(\delta H) \leq \varepsilon$. Thus, the new perturbation theory and error analysis with respect to $rel_H(\delta H)$ accurately predict that each component of the solution is computed to full relative accuracy.

We have successfully developed new perturbation theory, algorithms, and error analysis for the measure $rel_H(\delta H)$ for much of numerical linear algebra. We cannot always guarantee to solve problems as though we had a small $rel_H(\delta H)$, but the algorithms can in all cases monitor their accuracy and produce useful error bounds. The algorithms are usually small variations on conventional algorithms, perhaps with a slightly different stopping criterion, although the bidiagonal SVD algorithm has a quite new component. In all cases the algorithms run approximately as fast as their conventional counterparts, sometimes a little slower and sometimes a little faster. Since they are based on the conventional algorithms, all the techniques using the Level 3 BLAS apply to them.

This approach has been applied to linear equation solving[2], linear least-squares problems[3], the bidiagonal SVD[11,9], the tridiagonal symmetric eigenproblem[30,4], the dense symmetric positive definite eigenproblem[12], and the dense definite generalized eigenproblem[4,12]. We have similar but slightly weaker results for the dense SVD and generalized SVD[12]. These algorithms either will be included directly in LAPACK or can be easily constructed by using LAPACK subroutines as 'building blocks'.

We briefly describe the results for the bidiagonal SVD. The standard algorithm[14] computes each singular value σ_i of a bidiagonal matrix B with an error bound $O(\varepsilon)||B||_2$. Large singular values, i.e. those near $||B||_2$, are therefore computed to high relative accuracy, whereas tiny ones may be computed with no relative accuracy at all. In contrast, our new algorithm can compute each singular value to nearly full precision, no matter how tiny it is. It does this with no degradation in speed on average, and is often faster[11]. It also computes the singular vectors corresponding to groups of nearby tiny singular values much more accurately than the standard algorithm. The proof of this last property relies on the fact that the algorithm is the integer time evaluation of a Hamiltonian differential equation, which may be used to prove that the rounding errors accumulate quite slowly[9].

6. TARGET MACHINES

The LAPACK library will be designed primarily to perform efficiently on machines with a modest number of processors (say, 1–100), each having a powerful vector-processing capability. These machines include all of the most powerful computers currently available and in use for general-purpose scientific computing: Cray-2, Cray X-MP, Cray Y-MP, CYBER 205, Fujitsu VP, IBM 3090/VF, NEC SX, Hitachi S-820, Alliant FX/80, Convex C-1, Convex C-2, Stardent, Sequent Symmetry, Encore Multimax, and BBN Butterfly. We hope that the library will also perform well on a wider class of parallel machines, including the Intel iPSC/860, and NCUBE. On conventional serial machines, the performance of the library is expected to be at least as good as that of the current LINPACK and EISPACK codes. Thus the library will be suitable across the whole range of machines from personal computers to supercomputers to experimental architectures.

We do not claim that the strategy of using Level 2 or Level 3 BLAS will necessarily attain optimal performance on all these machines; indeed, some algorithms can be structured in several different ways, all calling Level 3 BLAS, but with different performance characteristics. In such cases we shall choose the structure that provides the best 'average' performance over the range of target machines. Currently we are limiting machine-dependent optimizations to the BLAS to retain portability across architectures. We encourage vendors to provide implementations of the BLAS kernels that are optimized for their particular architecture. While users are free to develop their own versions of the LAPACK codes, we believe that the possible performance gain will be limited on the more conventional architectures.

On the more experimental architectures (in particular, distributed-memory machines), the restriction of optimization to the BLAS might be too limiting. In particular, it might be advantageous to introduce parallelism at the top-level of the algorithm instead of inside the BLAS. To aid users in experimenting on their particular architecture, the LAPACK codes have been carefully designed in a modular fashion and with the objective of minimizing data movement. Since data movement is the key issue in distributed-memory as well as shared-memory machines, the LAPACK codes should be easily 'tunable' to more experimental architectures.

Several of the algorithms we intend to implement[20] will require more than loop-based parallelism. These algorithms will rely upon the simplified SCHEDULE mechanism[21] to invoke parallelism. These ideas might also be used to express top-level parallelism in a portable fashion. We are also closely following the activities of the Parallel Computing Forum[25] which has been formed by computer vendors, software developers, national laboratories, and universities to exchange technical information and to document agreements on constructs for programming parallel applications for shared-memory multiprocessors.

7. FUTURE WORK

Our first software release in April 1989 distributed codes for linear equation solving and QR decomposition to over 20 beta-test sites. Our second release, occurred in April 1990 and includes software for iterative refinement, the non-symmetric eigenproblem, symmetric eigenproblem, and SVD. Banded problems, generalized eigenproblems and the generalized SVD, condition estimation for the eigenproblem, and low-rank updates of various decompositions followed in mid-1991. We expect the final public release in late 1991.

For the longer term, we have identified a number of research directions. First, we are interested in extending our approach to distributed-memory machines. These are more challenging than the shared-memory machines we have been working on, because of the additional cost of communication between different processors and memories. Second, we would like to systematically develop parameterized software that is both portable and efficient. In Section 3 we identified the block size n_b as one such parameter. Other parameters include the access order (which of the six versions of block Gaussian elimination is best) and features of the machine arithmetic (round-off level, overflow and underflow thresholds, presence or absence of guard digits, etc.). Third, we wish to identify features of computer architectures that either help or hinder production of good numerical software. Two examples of helpful features are the ability to access rows and

columns of matrices with similar speeds, and friendly error recovery such as the overflow flag in IEEE standard floating-point arithmetic[1]. We also wish to provide performance evaluation tools for new architectures. Finally, we plan to provide C and Fortran 8x versions of the library as well.

ACKNOWLEDGEMENTS

The work is supported by NSF grants ASC-8715728, ASC-9005933 and CCR-8552474, and by DARPA grant F49620-87-C0065.

REFERENCES

1. *IEEE Standard for Binary Floating Point Arithmetic*, ANSI/IEEE, New York, Std 754-1985 edition (1985).
2. M. Arioli, J. Demmel and I. S. Duff, 'Solving sparse linear systems witwh sparse backward error', *SIAM J. Matrix Anal. Appl.*, **10**(2), 165–190, April (1989).
3. M. Arioli, I. S. Duff and P. P. M. de Rijk, 'On the augmented system approach to sparse least-squares problems', *Num. Math.*, **55**, 667–684 (1989).
4. Jesse Barlow and James Demmel. 'Computing accurate eigensystems of scaled diagonally dominant matrices', *SIAM J. Num. Anal.*, **27**(3), 762–791, June (1990).
5. Chris Bischof, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling and Danny Sorensen, 'LAPACK provisional contents', Mathematics and Computer Science Division Report ANL-88-38, Argonne National Laboratory, Argonne, IL, September (1988). (LAPACK Working Note #5.)
6. Christian H. Bischof., 'Adaptive blocking in the QR factorization'. *The Journal of Supercomputing*, **3**(3), 193–208 (1989).
7. Christian H. Bischof and Charles F. Van Loan. 'The WY representation for products of Householder matrices', *SIAM Journal on Stat. and Sci. Compt.*, **8**, s2–s13 (1987).
8. Chandler Davis and W. Kahan, 'The rotation of eigenvectors by a perturbation iii', *SIAM Journal on Numerical Analysis*, **7**, 248–263 (1970).
9. P. Deift, J. Demmel, L.-C. Li and C. Tomei, 'The bidiagonal singular values decomposition and Hamiltonian mechanics', Computer Science Dept. Technical Report 458, Courant Institute, New York, NY, July (1989). (To appear in *SIAM J. Num. Anal.*)
10. James Demmel, Jeremy Du Croz, Sven Hammarling and Danny Sorensen. 'Guides for the design of symmetric eigenroutines, SVD, iterative refinement and condition estimation', Mathematics and Computer Science Division Report ANL/MCS-TM-111, Argonne National Laboratory, Argonne, IL, February (1988). (LAPACK Working Note #4.)
11. James Demmel and W. Kahan, 'Accurate singular values of bidiagonal matrices', *SIAM J. Sci. Stat. Comp.*, **11**(5), 873–912, September (1990).
12. James Demmel and K. Veselić. 'Jacobi's method is more accurate than QR'. (To appear in *SIAM J. Matrix Anal. Appl.*)
13. J. Dongarra and D. Sorensen, 'A fully parallel algorithm for the symmetric eigenproblem', *SIAM J. Sci. Stat. Comp.*, **8**(2), 139–154, March (1987).
14. J. J. Dongarra, J. R. Bunch, C. B. Moler and G. W. Stewart, *LINPACK Users' Guide*. SIAM Press, Philadelphia (1979).
15. Jack Dongarra, Jeremy Du Croz, Iain Duff and Sven Hammarling, 'A set of Level 3 basic linear algebra subprograms', *ACM TOMS*, **16**(1), 1–18 March 1990.
16. Jack Dongarra, Jeremy Du Croz, Iain Duff and Sven Hammarling, 'A proposal for a set of level 3 basic linear algebra subprograms', *SIGNUM Newsletter*, **22**(3), 2–14, February (1987).
17. Jack Dongarra, Jeremy Du Croz, Sven Hammarling and Richard J. Hanson, 'An extended set of fortran basic linear algebra subroutines', *ACM Transactions on Mathematical Software*, **14**(1), 1–17, March (1988).
18. Jack Dongarra and Eric Grosse, 'Distribution of mathematical software by electronic mail', *Communications of the ACM*, **30**(5), 403–407 (1987).

19. Jack Dongarra, Sven Hammarling and Danny Sorensen, 'Block reduction of matrices to condensed forms for eigenvalue computations', *Journal of Computational and Applied Mathematics*, **27**, 215–227 (1989).
20. Jack Dongarra and Danny Sorensen, 'A fully parallel algorithm for the symmetric eigenvalue problem', *SIAM Journal on Stat. and Sci. Compt.*, **8**(2), s139–s154 (1987).
21. Jack Dongarra and Danny Sorensen, 'A portable environment for developing parallel programs', *Parallel Computing*, **5**(1&2), 175–186 (1987).
22. Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling and Richard J. Hanson, 'An extended set of Fortran basic linear algebra subprograms', *ACM Transactions on Mathematical Software*, **14**(1), 1–17 (1988).
23. Jack J. Dongarra and Iain S. Duff, 'Advanced computer architectures', Technical Report CS-89-90, University of Tennessee (1989).
24. Jeremy Du Croz, Private communication (1987).
25. The Parallel Computing Forum, *PCF Fortran: Language Definition*. Kuck and Associates, Champaign, IL (1988).
26. K. Gallivan, R. Plemmons and A. Sameh, 'Parallel algorithms for dense matrix computations', *SIAM Review*, **32**, 54–135 (1990).
27. B. Garbow, J. Boyle, J. Dongarra and C. Moler. *Matrix Eigensystem Routines — EISPACK Guide Extension*. Vol. 51 of *Lecture Notes in Computer Science*, Springer Verlag, New York (1977).
28. Gene H. Golub and Charles F. Van Loan, *Matrix Computations*, 2nd edn, The Johns Hopkins Press, Baltimore, MD (1989).
29. Kai Hwang and Fayé A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York (1984).
30. W. Kahan, 'Accurate eigenvalues of a symmetric tridiagonal matrix', Computer Science Dept. Technical Report CS41, Stanford University, Stanford, CA, July (1966). (Revised June 1968.)
31. C. Lawson, R. Hanson, D. Kincaid and F. Krogh, 'Basic linear algebra subprograms for fortran usage', *ACM Transactions on Mathematical Software*, **5**, 308–323 (1979).
32. S.-S. Lo, B. Phillipe and A. Sameh, 'A multiprocessor algorithm for the symmetric eigenproblem', *SIAM J. Sci. Stat. Comp.*, **8**(2), 155–165 (1987).
33. Robert Schreiber and Charles Van Loan, 'A storage efficient WY representation for products of Householder transformations', *SIAM Journal on Stat. and Sci. Compt.*, **10**(1), 53–57 (1989).
34. A. Van Der Sluis, 'Condition numbers and equilibration of matrices', *Numerische Mathematik*, **14**, 14–23 (1969).
35. B. Smith, J. Boyle, J. Dongarra, B. Garbow, Y. Ikebe, V. Klema and C. B. Moler, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd edn, Springer-Verlag, New York (1976).
36. Per Stenström, 'Reducing contention in shared-memory multiprocessors', *IEEE Computer*, **21**(11), 26–37 (1988).
37. Harold Stone, *High-Performance Computer Architecture*, Addison-Wesley, Reading, MA (1987).