[32] D. Sorensen and P. Tang. On the orthogonality of eigenvectors computed by divide-and-conquer techniques. Mathematics and Computer Science Division MCS-P152-0490, Argonne National Lab, Argonne, IL, May 1990. to appear in SIAM J. Num. Anal.

[33] J. H. Wilkinson. *The Algebraic Eigenvalue Problem.* Oxford University Press, 1965.

[19] J. Dongarra and M. Sidani. A parallel algorithm for the non-symmetric eigenvalue problem. Computer Science Dept. Technical Report CS-91-137, University of Tennessee, Knoxville, TN, 1991.

[20] J. Dongarra and D. Sorensen. A fully parallel algorithm for the symmetric eigenproblem. *SIAM J. Sci. Stat. Comput.*, 8(2):139–154, March 1987.

[21] J. Du Croz and M. Pont. The development of a floating-point validation package. In M. J. Irwin and R. Stefanelli, editors, *Proceedings of the 8th Symposium on Computer Arithmetic*, Como, Italy, May 19-21 1987. IEEE Computer Society Press.

[22] B. S. Garbow, J. M. Boyle, J. J. Dongarra, and C. B. Moler. *Matrix Eigensystem Routines – EISPACK Guide Extension*, volume 51 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1977.

[23] G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, 1983.

[24] N. J. Higham. Exploiting fast matrix multiplication within the Level 3 BLAS. *ACM Trans. Math. Soft.*, 16:352–368, 1990.

[25] W. Kahan. Paranoia. available from Netlib[18].

[26] W. Kahan. Analysis and refutation of the international standard iso/iec for language compatible arithmetic, 1991.

[27] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.

[28] S. Linnainmaa. Software for doubled-precision floating point computations. *ACM Trans. Math. Soft.*, 7:272–283, 1981.

[29] M. Payne and B. Wichmann. Information technology - programming languages - language compatible arithmetic. Project jtc1.22.28, ISO/IEC JTC1/SC22/WG11, 1 March 1991. First Committee Draft (Version 3.1).

[30] D. Priest. Algorithms for arbitrary precision floating point arithmetic. In P. Kornerup and D. Matula, editors, *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 132–145, Grenoble, France, June 26-28 1991. IEEE Computer Society Press.

[31] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines – EISPACK Guide*, volume 6 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1976.

[5] Z. Bai and J. Demmel. On a block implementation of Hessenberg multishift QR iteration. *International Journal of High Speed Computing*, 1(1):97–112, 1989. (also LAPACK Working Note #8).

[6] Z. Bai, J. Demmel, and A. McKenney. On the conditioning of the nonsymmetric eigenproblem: Theory and software. Computer Science Dept. Technical Report 469, Courant Institute, New York, NY, October 1989. (LAPACK Working Note #13).

[7] D. H. Bailey, K. Lee, and H. D. Simon. Using Strassen's algorithm to accelerate the solution of linear systems. *J. Supercomputing*, 4:97–371, 1991.

[8] S. Batterson. Convergence of the shifted QR algorithm on 3 by 3 normal matrices. *Num. Math.*, 58:341–352, 1990.

[9] C. Bischof. Adaptive blocking in the QR factorization. *J. Supercomputing*, 3(3):193–208, 1989.

[10] C. Bischof and P. Lacroute. An adaptive blocking strategy for matrix factorizations. In H. Burkhart, editor, *Lecture Notes in Computer Science 457*, pages 210–221, New York, NY, 1990. Springer Verlag.

[11] J. Bunch, J. Dongarra, C. Moler, and G. W. Stewart. *LINPACK User's Guide*. SIAM, Philadelphia, PA, 1979.

[12] T. Dekker. A floating point technique for extending the available precision. *Num. Math.*, 18:224–242, 1971.

[13] J. Demmel. Underflow and the reliability of numerical software. *SIAM J. Sci. Stat. Comput.*, 5(4):887–919, Dec 1984.

[14] J. Demmel. LAPACK: A portable linear algebra library for supercomputers. In *Proceedings of the 1989 IEEE Control Systems Society Workshop on Computer-Aided Control System Design*, Tampa, FL, Dec 1989. IEEE.

[15] J. Demmel and N. J. Higham. Stability of block algorithms with fast Level 3 BLAS. to appear in ACM TOMS.

[16] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[17] J. Dongarra, J. Du Croz, S. Hammarling, and Richard J. Hanson. An extended set of fortran basic linear algebra subroutines. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[18] J. Dongarra and E. Grosse. Distribution of mathematical software via electronic mail. *Communications of the ACM*, 30(5):403–407, July 1987.

9 Future Work

We have recently begun work on a new version of LAPACK. We intend to pursue all the goals listed above, in particular

- Producing a version for distributed memory parallel machines,
- Adding more routines satisfying new componentwise relative stability bounds,
- Adding condition estimators and error bounds for all quantities computable by the library,
- Producing routines designed to exploit exception handling features of IEEE arithmetic, and
- Producing Fortran 90 and C versions of the software.

We hope the insight we gained in this project will influence future developers of hardware, compilers and systems software so that they provide tools to facilitate development of high quality portable numerical software.

10 Acknowledgements

The authors acknowledge the contributions of the many contributors to the LAPACK project: E. Anderson, Z. Bai, C. Bischof, P. Deift, J. DuCroz, A. Greenbaum, S. Hammarling, E. Jessup, L.-C. Li, A. McKenney, D. Sorensen, P. Tang, C. Tomei, and K. Veselić.

References

[1] E. Anderson and J. Dongarra. Results from the initial release of LAPACK. Computer science dept. technical report cs-89-89, University of Tennessee, Knoxville, 1989. (LAPACK Working Note #16).

[2] E. Anderson and J. Dongarra. Evaluating block algorithm variants in LAPACK. Computer science dept. technical report cs-90-103, University of Tennessee, Knoxville, 1990. (LAPACK Working Note #19).

[3] ANSI/IEEE, New York. *IEEE Standard for Binary Floating Point Arithmetic*, Std 754-1985 edition, 1985.

[4] M. Assadullah, J. Demmel, S. Figueroa, A. Greenbaum, and A. McKenney. On finding eigenvalues and singular values by bisection. LAPACK Working Note. in preparation.

submatrix or growing number of columns. A third example is to let P, M and N grow with $M = O(P)$ and $N = O(P^2)$, and I constant. This corresponds to keeping a constant number of columns (or rows) per memory. Finally, we can keep N and M constant and let I and $P = O(I)$ grow. This corresponds to solving more independent problem instances of the same size, and keeping the same sized submatrix on each processor. Thus, depending on what kinds of scalability we wish to support, we may have to support many data layouts.

If ever there were a case for semi-automatically generated polyalgorithms, this may be it. The danger in choosing to support only a few of the plethora of possibilities is that the decision may turn into a self fulfilling prophecy so that the other memory mappings are of little use.

It is still unclear which programming model is best, and how many of these diverse data layouts need to be supported. *Jack: should we say anything more about parallel programming models?*

8 Suggestions for Architectures and Programming Languages

We have listed a number of suggestions for architectures and programming languages in earlier sections; we summarize them here:

1. Ability to express parallelism in a high level language.

2. Ability to perform floating point operations reasonably efficiently in double the largest input precision, even if only simulated in software using that input precision exclusively.

3. Access to efficiently implemented exception handling facilities, particularly infinity arithmetic. Trap handlers are a poor substitute.

4. Carefully implemented complex arithmetic and BLAS.

5. A standard set of floating point enquiries sufficiently detailed to describe the features of the last items, and unambiguously. Perhaps NextAfter [3] is the key.

6. BLAS for dealing with quasi-triangular matrices.

Note that a complete implementation of IEEE arithmetic would satisfy suggestions 2 and 3 above.

All in one ($p = 1,\ b = 1$)	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Blocked ($p = 4,\ b = 4$)	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4
Cyclic ($p = 4,\ b = 1$)	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Block cyclic ($p = 4,\ b = 2$)	1	1	2	2	3	3	4	4	1	1	2	2	3	3	4	4
Irregular	1	4	1	3	2	4	1	1	3	2	4	4	1	3	2	3

A multidimensional array may have each dimension stored in a different one of the layouts above, as shown in the following examples, labeled as above:

1212	1212	1212	1212
3434	3434	3434	3434
1212	1212	1212	1212
3434	3434	3434	3434
1212	1212	1212	1212
3434	3434	3434	3434
1212	1212	1212	1212
3434	3434	3434	3434

1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4
1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4

The first version of LAPACK was designed to handle *single problem instances*, e.g. a single system of linear equations to solve. On massively parallel machines one can expect users to want to solve many problems simultaneously. One way to do this is to use a multi-dimensional array, where two of the dimensions are the matrix dimensions and the others index independent problems.

Data layout is closely related to efficiency, because it is related to scalability. To be more precise, let $E(N, P, M, I)$ be the efficiency of the code as a function of problem size $N = n^2$ ($n =$ matrix dimension), number of processors P, memory size per processor M and number of independent problem instances I. Scalability means that as these four parameters grow, E should stay acceptably large, say at least .5 (i.e. at least half as fast as the best possible code for that machine). With four parameters, there are several ways they could grow, reflecting different uses of the library. For example, suppose P and N grow with $N = O(P)$, and M and I remain constant. This corresponds to adding more identical processors to the system and letting the problem size grow proportionally to the total memory. This can only be done with a data layout where each memory contains a small constant size submatrix of the whole matrix. A second example is to let M and N grow with $N = O(M)$, and P and I constant. This corresponds to adding memory to each processor, and letting the problem size grow proportionally. Here the data layout requires each memory to hold a constant fraction of the entire matrix, perhaps a growing the enormous intellectual effort that has been spent on codes like SLAMCH [25, 21].

However, there is a tradeoff between this kind of portability on the one hand and efficiency, accuracy and robustness on the other. Many if not most machines now supply IEEE arithmetic. As mentioned above, there are numerous places where significantly faster, more accurate and more robust code could have been written had we been able to assume that IEEE arithmetic *and standard high level language access to its exception handling features* were available. Unfortunately, no such standard high level language access exists yet. There have been attempts at such a standard [29, 26] but they fall far short of what is needed and could even make writing efficient portable code harder by mandating a standard environment antagonistic to what we need.

A deleterious by-product of the present situation is the near absence of any payoff for manufacturers who supply careful and complete IEEE arithmetic implementations, because little software exists that takes advantage of its features. Unless such software is written, manufacturers will have little incentive to implement these features, which then may even disappear from future versions of the standard.

We intend to produce IEEE-exploiting versions of those LAPACK codes which could benefit from special features of IEEE arithmetic. This includes condition estimators, eigenvector algorithms, and others. Not only will this code perform much better than the current portable code, but it will provide incentives to manufacturers to implement IEEE arithmetic with full access to its exception-handling features.

7 New Goals and Methods: Scope

In conventional libraries, as well as in the first version of LAPACK, dense rectangular matrices are stored in essentially one standard data structure: A statement like "DIMENSION A(20,10)" used to indicate that A is a rectangular array stored in consecutive memory locations (or contains a matrix stored in groups of evenly spaced consecutive memory locations). This is no longer a reasonable model on distributed memory machines, because there is no longer any such standard memory mapping. There are a number of competing parallel programming models (SPMD vs. MPMD, SIMD vs. MIMD, explicit message passing vs. implicit message passing, send/receive vs. put/get, etc.) and a large number of ways in which data can be distributed among memories [**?**]. For example, a one-dimensional array could be laid out in at least four different regular ways, with datum i stored in memory $\lfloor i/b \rfloor \mathrm{mod} p + 1$, where p is the number of memories used, and b is a blocking parameter. Various examples are shown below for $0 \leq i \leq 15$ (each box represents a data item, and the number inside is the number of the memory in which it is stored):

precision operations [12, 28, 32, 30]. However, this means that we must either assume the arithmetic is sufficiently accurate, which is not portable, or decide at run time whether the arithmetic is sufficiently accurate and then either do the simulated precision doubling or return an error flag. Making this decision at run-time is quite challenging, because there is no simple characterization of which arithmetics are sufficiently accurate. The desired simulation works, for example, with IEEE arithmetic, IBM 370 arithmetic, or VAX arithmetic, but requires different correctness proofs in each case. It does not work with Cray arithmetic. Thus it almost appears that we must be able to determine the floating point architecture at run-time in sufficient detail to determine the machine manufacturer.

The routine for determining floating point properties at run time, SLAMCH, has several other difficult tasks. It must also determine the overflow and underflow thresholds OVFL and UNFL, in particular without overflowing. Unfortunately, there can be different effective over/underflow thresholds depending on the operation and on the software. For example, the Cray divides a/b essentially using reciprocal approximation and multiplication $a * (1/b)$. If a and b are both tiny, then $1/b$ may overflow even though the true quotient a/b is quite moderate in value. The Cray and NEC machines both implement complex division in the simplest possible way, without branches: $\frac{a+ib}{c+id} = \frac{ac+bd}{c^2+d^2} + i\frac{bc-ad}{c^2+d^2}$. Thus even if the true quotient is modest in size, the computation can overflow if either c or d exceeds $\text{OVFL}^{1/2}$ in magnitude or both are sufficiently less than $\text{UNFL}^{1/2}$ in magnitude. This effectively cuts the exponent range in half. Similarly, there is a Level 1 BLAS routine called SNRM2 [27] which computes the Euclidean length of a vector: $(\sum_i x_i^2)^{1/2}$. The Cray uses this straightforward implementation which can again fail unnecessarily if any $|x_i| > \text{OVFL}^{1/2}$ or all $|x_i| < \text{UNFL}^{1/2}$. As a result of all these and other details, and the fact that new hardware and compilers are constantly appearing, SLAMCH is currently 2000 lines long and growing.

6 New Goals and Methods: Portability

As stated above, we can ask for portability of correctness (or of accuracy and robustness), or of performance. We have nearly abandoned portability of performance because of the need for machine dependent BLAS and block sizes. However, we do supply strictly portable Fortran BLAS and default block sizes which may provide adequate performance in some cases, but probably not peak performance on many architectures.

On the other hand, we have tried strictly to maintain portability of correctness. The "mail order software" model described above recognizes that code developed on one machine is often embedded (and hidden) in an application on another machine, and then used on a third. Consequently, it would be unreasonable to expect a user acquiring a code to modify all its subparts to ensure they run correctly on her machine. Since no standard language mechanism exists yet for making environmental enquiries about floating point properties, etc., all this must be done at run time. This explains if not justifies

Some modern parallel architectures are designed to support particular communication patterns and so may execute one algorithm (call it Algorithm A) much less efficiently than another (Algorithm B), even though on conventional computers A may have been as fast or faster than B. If Algorithm A is stable and Algorithm B is not, this means that the new architecture will not be able to run simultaneously as fast as possible and correctly in all cases. Thus one is tempted, in the interest of speed, to use an unstable algorithm. Since "the fast drives out the slow even if the fast is wrong", many users will prefer the faster algorithm despite occasional inaccuracy. So we are motivated to find a way to use unstable algorithm B provided we can check quickly whether it gets an accurate answer, and only occasionally resort to the slower alternative. For example, consider finding the eigenvalues of an n by n symmetric tridiagonal matrix T with diagonal entries $a_1, \ldots, a_n$ and offdiagonal entries $b_1, \ldots, b_{n-1}$. A standard bisection-based method uses the fact that the number of eigenvalues of T less than σ is the number of negative d_i where $d_i = (a_i - \sigma) - b_{i-1}^2/d_{i-1}$ (we take $b_0 = 0$ and $d_0 = 1$) [23]. Evaluating this recurrence straightforwardly requires $O(n)$ time and is stable. Using a parallel-prefix algorithm the d_i can be evaluated in $O(\log n)$ time but no stability proof exists. So we need either a stability proof for the $O(\log n)$ algorithm or a fast way to check the accuracy of the computed eigenvalues at the end of the computation. Similar issues arise with other tree-based algorithms.

Fifth, there is at least one important routine which *requires* double the input precision in some intermediate calculations to compute the answer correctly [32]. This is the solution of the so-called secular equation in the divide and conquer algorithm for the symmetric tridiagonal eigenproblem. This is somewhat surprising, since all other algorithms for this problem require only the input precision in all intermediate calculations. In fact, we must be careful to say what it means to require double precision, since in principal all computations could be done simulating arbitrary precision using integers: We mean in fact that there is an intermediate quantity in the algorithm which must be computed to high relative accuracy despite catastrophic cancellation in order to guarantee stability. (There are other examples where we were able to find an adequate single precision algorithm only after great effort, whereas an algorithm using a little double precision arithmetic was obvious. So even though double precision is not necessary in these cases, it would have made software design much easier.)

This requirement for double the input precision impacts library design as follows. Our original design goal was *not* to use mixed precision arithmetic. This traditional goal arose both because standard Fortran compilers were not required to supply a double precision complex data type, and because of the desire to use the same algorithm whether the input precision were single precision or double precision. (The use of mixed precision would have required quadruple precision for double precision input, and quadruple is rarely available.) An alternative is to simulate double precision using single (and quadruple using double). Provided the underlying arithmetic is accurate enough, there are a number of standard techniques for simulating "doubled precision" arithmetic using a few single the computation and again deduce that the problem is very ill-conditioned. If we use trapping, we need to be able to handle the trap and resume execution, not just terminate. To be fast, this cannot involve an expensive operating system call. Similarly, infinity-arithmetic must be done at normal hardware floating point speed, not via software, lest performance suffer devastation.

Many but not all machines support IEEE arithmetic. Many that claim to do so support neither the user readable overflow flag they should nor user handleable traps. And those that do support these things often use intolerably slow software implementations. Thus, we did not supply IEEE-exploiting routines in the first version of LAPACK. However, we intend to do so in future versions. This raises the question of portability, which we return to in section 6.

5 New Goals and Methods: Accuracy and Robustness

During work on LAPACK we have found better, or at least different, ways to understand the traditional goals described in section 2. The first improvement in accuracy and stability involved replacing the norms traditionally used for backward stability analysis. For example, consider solving $Ax = b$. As we said before, traditionally we have only guaranteed that the computed $\hat{x}$ satisfied $(A + E)\hat{x} = b + f$ where E and f were small in norm compared to A and b, respectively. If A were sparse, there was no guarantee that E would be sparse. Similarly, if A had both very large and very small entries, some entries of E could be very large compared to the corresponding entries of A. In other words, the usual methods did not respect the sparsity or scaling of the original problem.

Instead, LAPACK uses a method which (except for certain rare cases) guarantees *componentwise relative backward stability*: this means that $|E_{ij}| = O(\varepsilon)|A_{ij}|$ and $|f_k| = O(\varepsilon)|b_k|$. This respects both sparsity and scaling, and can result in a much more accurate $\hat{x}$. We have done this for various problems in LAPACK, including the bidiagonal singular value decomposition and symmetric tridiagonal eigenproblem. Future releases of LAPACK will extend this to other routines as well [14].

Second, we intend to supply condition estimators (i.e. error bounds) for every quantity computed by the library. This includes, for example, eigenvalues, eigenvectors and invariant subspaces [6]. Some problems remain for future releases (the generalized nonsymmetric eigenproblem).

Third, we determined that Strassen-based matrix multiplication is adequately accurate to achieve traditional normwise backward stability [15]. Strassen's method is not as accurate as conventional matrix multiplication when the matrices are badly row or column scaled, but if either the matrices are already reasonably scaled or if the bad scaling is first removed, it is adequate. Thus it may be used in Level 3 BLAS implementations [24, 7].

Fourth, there is possibly a tradeoff between stability and speed in certain algorithms.

methods are currently implemented only as serial codes. We intend to supply parallel versions in future releases.

Third, the Hessenberg eigenvalue algorithm has proven quite difficult to parallelize. We have a partially blocked implementation of the QR algorithm but the speedup is modest [5]. There has been quite recent progress [19], but it remains an open problem to produce a highly parallel and reliably stable and convergent algorithm for this problem and for the generalized Hessenberg eigenvalue problem.

Fourth is the issue of performance tuning, in particular choosing the block size parameters. In principal, the optimal block size could depend on the machine, problem dimension, and other problem parameters such as leading matrix dimension. We have a mechanism (subroutine ILAENV) for choosing the block size based on all this information. But we still need a better way to choose the block size. We used brute force during beta testing of LAPACK, running exhaustive tests on different machines, with ranges of block sizes and problem dimensions. This has produced a large volume of test results, too large for thorough human inspection.

There appear to be at least three ways to choose block parameters. First, we could take the exhaustive tests we have done, find the optimal block sizes, and store them in tables in subroutine ILAENV; each machine would require its own special tables. Second, we could devise an automatic installation procedure which could run just a few benchmarks and automatically produce the necessary tables. Third, we could devise algorithms which tuned themselves at run-time, choosing parameters automatically [9, 10]. The choice of method depends on the degree of portability we desire; we return to this in section 6 below.

Finally, we have determined that floating point exception handling impacts efficiency. Since overflow is a fatal exception on some machines, completely portable code must avoid it at all costs. This means extra tests, branches, and scaling must be inserted if spurious overflow is possible at all, and these slow down the code. For example, the condition estimators in LAPACK provide error bounds, and more generally warn about inaccurate answers to ill-conditioned problems. It is therefore important that these routines resist overflow. Since their main operation is (generally) solving a triangular system of equations, we cannot use the standard Level 2 BLAS triangular equation solver [17] because it is unprotected against overflow. Instead, we have another triangular solver written in Fortran including scaling in the inner loop. This gives us a double performance penalty, since we cannot use optimized BLAS, and since we must do many more floating point operations and branches. The same issues arise in computing eigenvectors.

If we could assume we had IEEE arithmetic [3], none of this would be necessary. Instead, we would run with the usual BLAS routine. If an overflow occurred, we could either trap, or else substitute an ∞ symbol, set an "overflow flag" and continue computing using the rules of infinity-arithmetic. If we trapped, we could immediately deduce that the problem is very ill-conditioned, and terminate early returning a large condition number. If we continued with infinity-arithmetic, we could check the overflow flag at the end of real and complex matrices, in both single and double precision. LAPACK will be in the public domain and available from Netlib some time in 1991.

The library is written in standard Fortran 77. The high performance is attained by calls to block matrix operations, such as matrix-multiply, in the innermost loops [14, 2]. These operations are standardized as Fortran subroutines called the Level 3 BLAS (Basic Linear Algebra Subprograms [16]). Although standard Fortran implementations of the Level 3 BLAS are available on Netlib, high performance can generally be attained only by using implementations optimized for each particular architecture. In particular, all parallelism (if any) is embedded in the BLAS and transparent to the user.

Besides depending upon locally implemented Level 3 BLAS, good performance also requires knowledge of certain machine-dependent *block sizes*, which are the sizes of the submatrices processed by the Level 3 BLAS. For example, if the block size is 32 for the Gaussian Elimination routine on a particular machine, then the matrix will be processed in groups of 32 columns at a time. Details of the memory hierarchy determine the block size that optimizes performance [1].

4 New Goals and Methods: Efficiency

The most important fact is that the Level 3 BLAS have turned out to be a satisfactory mechanism for producing fast transportable code for most dense linear algebra computations on high performance *shared memory* machines. (Dealing with distributed memory machines is future work we describe below.) Gaussian elimination and its variants, QR decomposition, and reductions to Hessenberg, tridiagonal and bidiagonal forms (as preparation for finding eigenvalues and singular values) all admit efficient block implementations [1, 2]. Such codes are often nearly as fast as full assembly language implementations for sufficiently large matrices, but approach their asymptotic speeds more slowly. Parallelism, embedded in the BLAS, is generally useful only on sufficiently large problems, and can in fact slow down processing on small problems. This means that the number of processors exercised should ideally be a function of the problem size, something not always taken into account by existing BLAS implementations.

However, the BLAS do not deal with all problems, even in the shared memory world. First, the real nonsymmetric eigenvalue problem involves solving systems with quasi-triangular matrices (block triangular matrices with 1 by 1 and 2 by 2 blocks). These are not handled by the BLAS and so must be written in Fortran. As a result, the real nonsymmetric eigenproblem runs relatively slowly compared to the complex nonsymmetric eigenproblem, which has only standard triangular matrices.

Second, finding eigenvalues of a symmetric tridiagonal matrix, and singular values of a bidiagonal matrix, can not exploit blocking. For these problems, we invented other methods which are potentially quite parallel [20, 4]. However, since the parallelism is not embedded in the BLAS, and since standard Fortran 77 cannot express parallelism, these fixed operation counts like Gaussian elimination, but also for routines that iterate (to find eigenvalues). Why this should be so for some eigenroutines is still not completely understood; worse, some nonconvergent examples have been discovered only recently [8].

Portability in its most inclusive sense means that the code is written in a standard language (say Fortran), and that the source code can be compiled on an arbitrary machine with an arbitrary Fortran compiler to produce a program that will run correctly and efficiently. We call this the "mail order software" model of portability, since it reflects the model used by software servers like Netlib [18]. This notion of portability is quite demanding. It demands that all relevant properties of the computer's arithmetic and architecture be discovered at runtime within the confines of a Fortran code. For example, if the overflow threshold is important to know for scaling purposes, it must be discovered at runtime *without overflowing*, since overflow is generally fatal. Such demands have resulted in quite large and sophisticated programs [25, 21] which must be modified continually to deal with new architectures and software releases. The mail order software notion of portability also means that codes generally must be written for the worst possible machine expected to be used, thereby often degrading performance on all the others.

Finally, *wide scope* refers to the range of input problems and data structures the code will support. For example, LINPACK and EISPACK deal with dense matrices (stored in a rectangular array), packed matrices (where only the upper or lower half of a symmetric matrix is stored), and band matrices (where only the nonzero bands are stored). In addition, there are some special internally used formats such as Householder vectors to represent orthogonal matrices. Then there are sparse matrices which may be stored in innumerable ways, but in this paper we will limit ourselves to dense and band matrices, the mathematical types addressed by LINPACK, EISPACK and LAPACK.

3 LAPACK Overview

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 called LAPACK (short for Linear Algebra Package). The library is intended to provide a coordinated set of subroutines to solve the most common linear algebra problems and to run efficiently on a wide range of high-performance computers.

LAPACK will provide routines for solving systems of simultaneous linear equations, least-squares solutions of linear 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 handled, but not general sparse matrices. In all areas, similar functionality will be provided for

The breadth of material we will cover does not permit us to describe or justify all our claims in detail. Instead we give an overview, and relegate details to future papers.

2 Traditional Library Design Goals

The traditional goals of good library design are the following:

- stability and robustness,
- efficiency,
- portability, and
- wide scope.

Let us consider these in more detail in the context of libraries for numerical linear algebra, particularly LINPACK and EISPACK. The terms have traditional interpretations:

In linear algebra, *stability* refers specifically to *backward stability with respect to norms* as developed by Wilkinson [33, 23]. In the context of solving a linear system $Ax = b$, for example, this means that the computed solution $\hat{x}$ solves a perturbed system $(A + E)\hat{x} = b + f$ where $\|E\| = O(\varepsilon)\|A\|$ and $\|f\| = O(\varepsilon)\|b\|$[1]. Similarly, in finding eigenvalues of a matrix A the computed eigenvalues are the exact eigenvalues of $A + E$ where again $\|E\| = O(\varepsilon)\|A\|$ [2] *Robustness* refers to avoiding overflows and dangerous underflows. In particular, it means that if the inputs are "far" from over/underflow, and the true answer is far from over/underflow, then the program should not overflow (which generally halts execution) or underflow in such a way that the answer is much less accurate than in the presence of roundoff alone. For example, in standard Gaussian elimination with pivoting, intermediate underflows do not change the bounds for $\|E\|$ and $\|f\|$ above so long as A, b and x are far enough from underflow themselves [13].

Among other things, *efficiency* means that the performance (floating point operations per second, or flops) should not degrade for large problems; this property is frequently called *scalability*. When using direct methods as in LINPACK and EISPACK, it also means that the running time should not vary greatly for problems of the same size (though occasional examples where this occurs are sometimes dismissed as "pathological cases"). Maintaining performance on large problems means, for example, avoiding unnecessary page faults. This was a problem with EISPACK, and was fixed in LINPACK by using column oriented code which accesses matrix entries in consecutive memory locations in columns (since Fortran stores matrices by column) instead of by rows. Running time depends almost entirely on a problem's dimension alone, not just for algorithms with

[1] The constants in $O(\varepsilon)$ depend on dimensionality in a way that is important in practice but not here.

[2] This is one version of backward stability. More generally one can say that an algorithm is backward stable if the answer is not worse than what would be be computed exactly from a slightly perturbed input, even if one cannot construct this slightly perturbed input.

1 Introduction

The original goal of the LAPACK [14] project was to modernize the widely used LINPACK [11] and EISPACK [31, 22] numerical linear algebra libraries to make them run efficiently on shared memory vector and parallel processors. On these machines, LINPACK and EISPACK are inefficient because their memory access patterns do not respect the multilayered memory hierarchies of the machines, thereby spending too much time moving data instead of doing useful floating point operations. LAPACK tries to cure this by reorganizing the algorithms to use block matrix operations. These block operations can be optimized for each architecture to account for the memory hierarchy, and so provide a transportable way to achieve high efficiency on diverse modern machines.

We say "transportable" instead of "portable" because for fastest possible performance LAPACK requires that highly optimized block matrix operations be already implemented on each machine by the manufacturers or someone else. In other words the correctness of the code is portable, but high performance is not if we limit ourselves to a single (Fortran) source code. Thus we have modified the traditional and honorable goal of portability in use among numerical library designers, where both correctness and performance were retained as the source code was moved to new machines, because it is no longer appropriate on modern architectures.

Portability is just one of the many traditional design goals of numerical software libraries we reconsidered and sometimes modified in the course of designing LAPACK. Other goals are stability (or accuracy), robustness against over/underflow, portability of correctness (in contrast to portability of performance), and scope (which input data structures to support). Recent changes in computer architectures and numerical methods have permitted us to to strengthen these goals in many cases, resulting in a library more capable than before. These changes include the availability of massive parallelism, IEEE floating point arithmetic, new high accuracy algorithms, and better condition estimation techniques. We have also identified tradeoffs among the goals, as well as certain architectural and language features whose presence (or absence) makes achieving these goals easier.

Section 2 reviews traditional goals of library design. Section 3 gives an overview of the LAPACK library. The next three sections discuss how traditional design goals and methods have been modified: Section 4 deals with efficiency, section 5 with stability and robustness, section 6 with portability, and section 7 with scope. Section 8 lists particular architectural and programming language features that bear upon the goals. Section 9 describes future work on distributed memory machines.

We will use the notation $\|x\|$ to refer to the largest absolute component of the vector x, and $\|A\|$ to be the corresponding matrix norm (the maximum absolute row sum). ε will denote the machine roundoff, UNFL the underflow threshold (smallest positive normalized floating point number) and OVFL the overflow threshold (the largest finite floating point number).

On Designing Portable High Performance Numerical Libraries

James Demmel*
W. Kahan†
Computer Science Division and Mathematics Department
University of California
Berkeley, CA 94720

Jack Dongarra‡
Computer Science Department
University of Tennessee
Knoxville, TN 37996

July 18, 1991

Abstract

High quality portable numerical libraries have existed for many years. These libraries, such as LINPACK and EISPACK, were designed to be accurate, robust, efficient and portable in a Fortran environment of conventional uniprocessors, diverse floating point arithmetics, and limited input data structures. These libraries are no longer adequate on modern high performance computer architectures. We describe their inadequacies and how we are addressing them in the LAPACK project, a library of numerical linear algebra routines designed to supplant LINPACK and EISPACK. We shall show how the new architectures lead to important changes in the goals as well as the methods of library design.

*Supported by the NSF via grants DCR-8552474, ASC-8715728 and ASC-9005933.
†Supported by the NSF via grant ASC-9005933.
‡Supported by the NSF via grants ASC-8715728 and ASC-9005933.