LAPACK: Linear Algebra Software for Supercomputers[1]

Christian H. Bischof
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439–4801
bischof@mcs.anl.gov

Argonne Preprint MCS-P236-0491
Appeared in Proc. 2nd ODIN Symposium, A. Schreiner and W. Ewinger, Eds., pp. 101-120, 1991.

This paper presents an overview of the LAPACK library, a portable, public-domain library to solve the most common linear algebra problems. This library provides a uniformly designed set of subroutines for solving systems of simultaneous linear equations, least-squares problems, and eigenvalue problems for dense and banded matrices. We elaborate on the design methodologies incorporated to make the LAPACK codes efficient on today's high-performance architectures. In particular, we discuss the use of block algorithms and the reliance on the Basic Linear Algebra Subprograms (BLAS). We present performance results that show the suitability of the LAPACK approach for vector uniprocessors and shared-memory multiprocessors. We also address some issues that have to be dealt with in tuning LAPACK for specific architectures. Lastly, we present results that show that the LAPACK software can be adapted with little effort to we distributed-memory environments, and discuss future efforts resulting from this project.

1 Introduction and Scope

The LAPACK (shorthand for Linear Algebra Package) library is a group effort to develop a portable public-domain 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 project was initialized by Jack Dongarra (University of Tennessee) and Jim Demmel (University of California at Berkeley). In addition, Ed Anderson (University of Tennessee), Zhaoujun Bai (University of Kentucky), Jeremy Du Croz (NAG Ltd.), Anne Greenbaum (New York University), Sven Hammarling (NAG Ltd.), Danny Sorensen (Rice University) and Chris Bischof (Argonne National Laboratory) made up the initial LAPACK team. During the past three years many more people have become involved in the project and provided us with algorithms, suggestions, and benchmark results.

LAPACK provides routines for solving

- systems of simultaneous linear equations;

[1] The work of the author was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38. The LAPACK project is also supported by the National Science Foundation under grant ASC-8715728.

- over- and underdetermined systems of equations;
- symmetric and unsymmetric eigenvalue problems, simple and generalized; and
- singular value problems.

Dense and banded matrices are provided for, but not general sparse matrices. In all areas, similar functionality will be provided for real and complex matrices. The LAPACK package also includes test and timing routines to verify the installation of the LAPACK codes on a particular architecture and to assess their performance (see [5]).

As a result, LAPACK serves many purposes.

- It is an easy-to-use library that solves linear algebra problems efficiently and reliably. Driver routines will be supplied to make the solution of common problems as easy as possible.
- Since source code is freely accessible, it is also a toolbox for the algorithm developer. It contains a wealth of reliable, well-documented, and integrated subroutines; and the algorithmic properties of the many new algorithms will be documented in the accompanying literature.
- The LAPACK codes can be used as a benchmark to compare the floating-point performance of various computers. Because of their considerable size, the codes are also good compiler and floating-point arithmetic test suites.
- Lastly, the accuracy and robustness of the LAPACK codes provide a standard against which competing implementations and algorithms can be measured.

The new library will extend the successful EISPACK [23, 30] and LINPACK [16] 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 that make the LAPACK algorithms more appropriate for today's high-performance architectures. In particular, LAPACK codes have been carefully restructured to reduce the cost of data movement as much as possible.

LAPACK is designed to be efficient and transportable across a wide range of computing environments, with special emphasis on tightly coupled shared-memory multiprocessors as, for example, the Alliant FX/8, IBM 3090/VF, CRAY-2, or CRAY Y-MP multiprocessors. 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) [17, 19, 27] 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.

A detailed description of the LAPACK package is given in [8].

2 The Basic Linear Algebra Subprograms (BLAS)

The Basic Linear Algebra Subprograms (BLAS) provide an interface for the elementary matrix and vector operations. The first BLAS [27], which we call Level 1 BLAS, implement common vector-vector operations such as a dot product, or a "saxpy,"

$$y \leftarrow y + \alpha x,$$

where x and y are vectors and α is a scalar. The Level 2 BLAS [19] provide matrix-vector operations such as matrix-vector multiplication and rank-one updates. The development of the Level 2 BLAS was motivated by vector-processing machines. Many of the frequently used algorithms of numerical linear algebra can readily be coded so that the bulk of the computation is performed by calls to the Level 2 BLAS routines.

Unfortunately, this approach is often not well suited to computers with a memory hierarchy (such as global memory, cache or local memory, and vector registers) and parallel-processing computers. (For a description of many advanced-computer architectures, see [20, 26, 31].) Data at low levels of the memory hierarchy can be accessed immediately, whereas 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. For this reason it is imperative to reuse data as much as possible to cut down on data movement overhead.

This goal can be achieved by expressing a computation in terms of matrix-matrix operations. The Level 3 BLAS [17] provide the matrix-matrix operations needed for linear algebra. Together with the Level 1 and 2 BLAS, they provide a well-defined interface for the elementary matrix and vector operations and add to the portability, modularity, and ease of maintenance of the software.

As an example, consider the ratio of memory references to arithmetic operations for a "saxpy" (a Level 1 BLAS), a matrix-vector multiply (a Level 2 BLAS), and a matrix-matrix multiply (a Level 3 BLAS). For n-vectors and $n \times n$ matrices, it is 3:2, 1:2, and 1.5:n, respectively,

We see that the use of higher-level BLAS requires less data movement. In particular, for the Level 3 BLAS, we achieve a *surface-to-volume effect* for the ratio of operations to data movement. The superiority of the Level 3 BLAS is borne out by the performance numbers in Figure 2, which shows the performance of a matrix-matrix multiply (the multiplication of a $100 \times k$ by a $k \times n$ matrix, with $k = 96$ for the Siemens and $k = 100$ for the Crays) versus a matrix-vector multiply (the multiplication of a $100 \times n$ matrix by an n-vector) on the CRAY-2 (with 4 processors), the CRAY Y-MP (with 8 processors), and the Siemens S600/10. It should be noted that the performance numbers for the Siemens S600/10 are preliminary, since the BLAS employed are simply the Fortran BLAS that had been optimized for the Siemens VP series [24, 25].

For all machines a matrix-matrix multiply is preferable to a series of matrix-vector multiplies. The reason is that the memory bandwidth of high-performance supercomputers is significantly lower than the speed of the processors. For example, on the IBM 3090/VF, data has to be resident in cache before it can be used as an operand. The CRAY-2, in contrast, has 16 kwords of so-called local memory, from which data can be accessed in four clock cycles, whereas references to global memory experience a startup overhead of 63 cycles for a vector load or store. An additional bottleneck on those machines is the fact that there is only one path into main memory which has to be used for both load and stores. In contrast, each processor of an CRAY X-MP and CRAY Y-MP has two load pipes and one store pipe into main memory, greatly alleviating this I/O bottleneck. In general, one can obtain a good idea of the usefulness of higher-level BLAS by comparing the peak performance of a machine with its peak transfer rate into main memory. This ratio varies between 1.5 for the CRAY Y-MP and 0.12 for the Alliant FX/80, with values between 0.5 and 1 being common. Whenever the ratio of peak performance to peak memory bandwidth is low, it is imperative to reuse data in fast memory in order not to degrade the computation rate by waiting for memory accesses. The Level 3 BLAS allow for efficient reuse of fast memory and hence provide an algorithmic tool that ensures good performance on such machines.

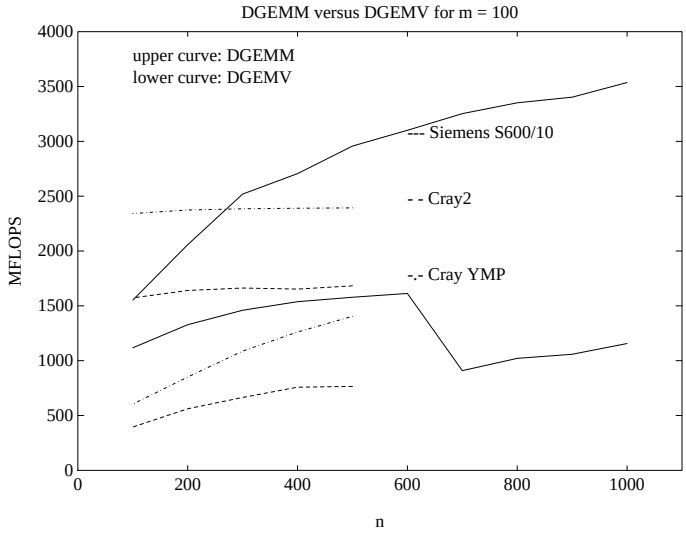


Figure 1: Speed of Matrix-Matrix and Matrix-Vector Multiply

3 Block Algorithms

The LINPACK and EISPACK codes were written in a fashion that, for the most part, ignored the cost of data movement. The preceding section, however, has shown that considerable performance improvements can be achieved by using matrix-matrix kernels.

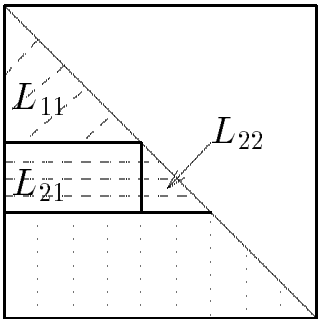
In some algorithms (e.g., computing the eigenvalues of a symmetric tridiagonal matrix), use of such kernels is not feasible. In the majority of algorithms, however, there is scope for using the Level 2 and Level 3 BLAS. To exploit the Level 3 BLAS, one usually must express the algorithm at the top level in terms of operations on submatrices (the so-called blocks) as compared to vector- or scalar-oriented operations. Many references to block algorithms can be found in [12, 21, 22].

As an example of a block algorithm, we consider algorithms for computing the Cholesky decomposition

$$A = LL^T,$$

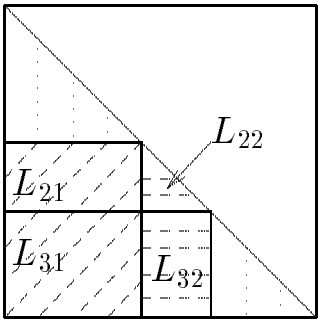
where A is a symmetric positive definite matrix and L is an lower triangular matrix. There are several ways of computing this factorization; and if one overwrites the lower triangle of A with L, one arrives at the variants shown in Figure 2. A is partitioned conformably. In the top-looking version we overwrite A_{21} and A_{22}, using L_{11}; in the left-looking version we overwrite A_{22} and A_{32}, using L_{21} and L_{31}; and in the right-looking version we overwrite A_{22} and A_{32}, updating A_{33}. If A is partitioned such that L_{22} is a scalar, we obtain the usual unblocked versions of the algorithms; $L_{22} = \sqrt{A_{22}}$, and the solution of the equation systems involving L_{22} is simply a scalar division.

To arrive at a block algorithm, we consider A as composed of submatrices as indicated in Figure 2,



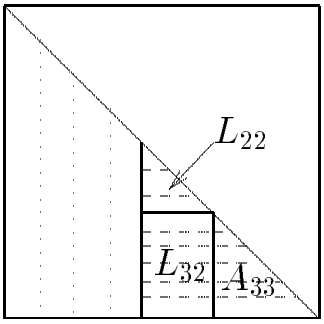
Top-looking Variant

$L_{21} \leftarrow A_{21} L_{11}^{-T}$
$A_{22} \leftarrow A_{22} - L_{21} L_{21}^{T}$
$L_{22} \leftarrow$ Cholesky factor of A_{22}



Left-looking Variant

$A_{22} \leftarrow A_{22} - L_{21} L_{21}^{T}$
$L_{22} \leftarrow$ Cholesky factor of A_{22}
$A_{32} \leftarrow A_{32} - L_{31} L_{21}^{T}$
$L_{32} \leftarrow A_{32} L_{22}^{-T}$



Right-looking Variant

$L_{22} \leftarrow$ Cholesky factor of A_{22}
$L_{32} \leftarrow A_{32} L_{22}^{-T}$
$A_{33} \leftarrow A_{33} - L_{32} L_{32}^{T}$

Dotted areas are "not" references, diagonally dashed areas are "read," and horizontally dashed areas are "read" and "written."

Figure 2: Three Variants for the Block Formulation of the Cholesky Factorization

Table 1: Distribution of Floating-Point Operations in Different Variants of the Block Cholesky Algorithm on a 500×500 matrix, Using blocksize 64

Cholesky Variant	Unblocked Cholesky	STRSM	SSYRK	SGEMM
top-looking	1.6%	82.1%	16.3%	0%
left-looking	1.6%	16.7%	16.3%	65.3%
right-looking	1.6%	16.7%	81.6%	0%

Table 2: Memory Access Cost (in Kwords) of Different Variants of the Block Cholesky Algorithm

Variant	Reads	Writes	Reads+Writes
top-looking	536	140	676
left-looking	697	249	946
right-looking	513	389	902
unblocked	21082	125	21208

instead of scalar entries or vectors. Let us assume for the sake of simplicity that A can be partitioned into N subblocks of size $nb \times nb$ each. It is characteristic of block factorization algorithms that one needs an "unblocked" code as well — in this case, the Cholesky factorization of a diagonal subblock. Here we employ the algorithm that uses Level 2 BLAS kernels, but the key issue is that we employ this algorithm only on the (comparatively small) $nb \times nb$ diagonal subblocks.

The three Cholesky variants require the same number of floating-point operations and are numerically equivalent, yet they differ in their use of the BLAS kernels and in the number of reads and writes they require. In detail, these versions require

1. the unblocked Cholesky code;
2. the BLAS routine STRSM, which implements a triangular solver with multiple right-hand sides (i.e., $X \leftarrow L^{-1}B$);
3. the BLAS routine SSYRK, which implements a symmetric rank-k update (i.e., $A \leftarrow A - BB^T$); and
4. the BLAS routine SGEMM, which implements a matrix-matrix multiply-and-add (i.e., $C \leftarrow \alpha AB + \beta C$).

The distribution of work among those kernels is shown in Table 1 for a 500×500 matrix partitioned into blocks of size 64. We see that the top-looking version does nearly all its work in solving triangular systems, the left-looking version favors matrix-matrix multiply, and the right-looking version is biased towards symmetric rank-k updates.

For memory accesses, we have the situation shown in Table 2, again on a 500×500 matrix partitioned into blocks of size 64. The surface-to-volume ratio mentioned before becomes apparent when

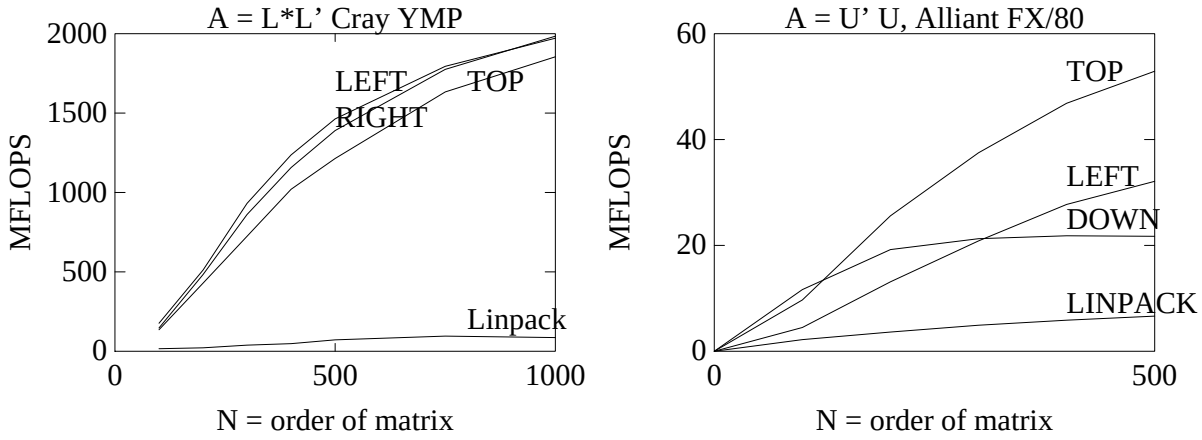


Figure 3: Comparison of Block Cholesky Variants with LINPACK Code

we compare the number of memory accesses required for the blocked versions with the number for the unblocked version. Of the blocked versions, the top-looking version requires the least number of memory accesses. The left-looking version requires the greatest total number of memory references, but fewer writes than the right-looking version. This latter feature may be advantageous in shared-memory multiprocessors where cache consistency is guaranteed by the use of "write-through" caches [26]. On those architectures, read accesses to cached data can be satisfied in one cycle, but write accesses are immediately flushed to memory; as a result, write accesses can be much slower than read accesses.

The performance of those variants on an eight-processor CRAY Y-MP is shown in Figure 3. This figure also shows the performance on an Alliant FX/80 of the corresponding variant if we overwrite the upper triangle of A (i.e., compute $A = U^T U$). We note that, because of the transposing of the matrix, the left-looking version for $A = LL^T$ corresponds to the top-looking version for $A = UU^T$. These particular versions have proven to be very effective because of their reliance on matrix-matrix multiplication, the operation of choice for many architectures. We also see that, depending on the architecture chosen, the differences between the various versions can be quite substantial. For a discussion of these issues for other decompositions, see [6]. For the LAPACK release, we tried to choose the variant that provides the best "average" performance over the range of target machines.

We also note that there may be quite a difference between the variant that overwrites the upper or lower triangle, because of the different memory access patterns. For example, on the Siemens S600/10 we observe the behavior shown in Figure 4. If we overwrite the upper triangle of a symmetric matrix, the blocked code (dashed line) is always inferior to the unblocked code (solid line), whereas if we overwrite the lower triangle, the blocked code (dash-dotted line) is always superior to the unblocked code (dotted line). Again, we stress that these performance numbers are preliminary.

Lastly, in Figure 5 we show the performance of the NEC SX/2, the CRAY-2/S, the CRAY Y-MP,

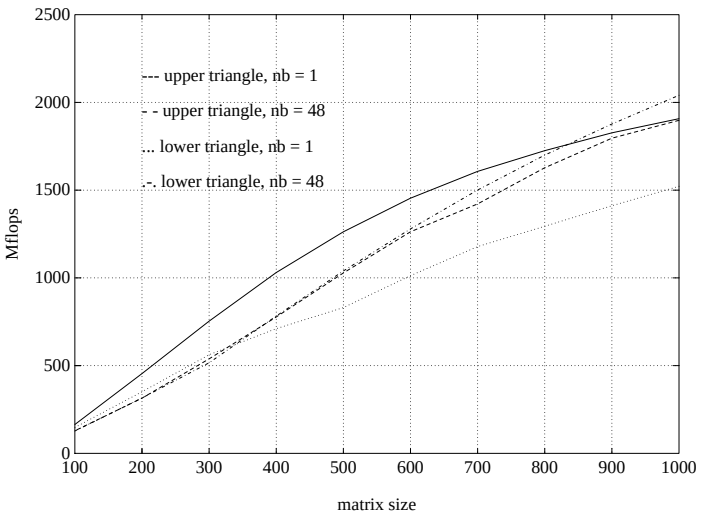


Figure 4: Performance of Cholesky Factorization Variants on Siemens S600/10

and the Siemens S600/10 on LU factorizations $A = LU$ of size up to 500. The peak performance of these machines is 1.3 Gflops, 2 Gflops, 4 Gflops, and 5 Gflops, respectively. On a 1000×1000 problem, the Siemens S600/10 reached 2.5 Gflops, even using just the recompiled Fortran BLAS of the Siemens VP series. We see that the blocked code usually performs better than the unblocked code (with the exception of the NEC SX/2, where the unblocked code is always better, largely because of the implementation of the BLAS). Further, as the numbers for the Crays indicate, both versions are substantially better than the LINPACK codes. The performance of the Siemens S600/10 on a set of other common matrix transformations is shown in Figure 6. The block size has been chosen to achieve best possible performance, and as can be seen, it may differ for the various algorithms. At any rate, however, the new codes perform substantially better than the EISPACK codes.

To achieve block algorithms for the Cholesky and LU factorizations is relatively simple: it is essentially a restructuring of loops. As a result, efforts are under way [15, 28] to achieve this effect by means of compiler transformations. There are other block algorithms, however (for example, the block multishift algorithm of Bai and Demmel [7] and the QR factorization algorithm for rank-deficient matrices of Bischof [11]), where new algorithms have been invented in order to be able to exploit the power of matrix-matrix kernels. For example, the performance of the QR algorithm for rank-deficient matrices on the CRAY Y-MP is shown in Figure 7. We here compare the LINPACK codes, the traditional algorithm implemented with the Level 2 BLAS, and the new block algorithm. Many references to block algorithms can be found in [12, 21, 22].

We also note that the LAPACK codes are, in many respects, significantly more reliable than other available software products of comparable scope. For example, the eigenvalue solvers and singular

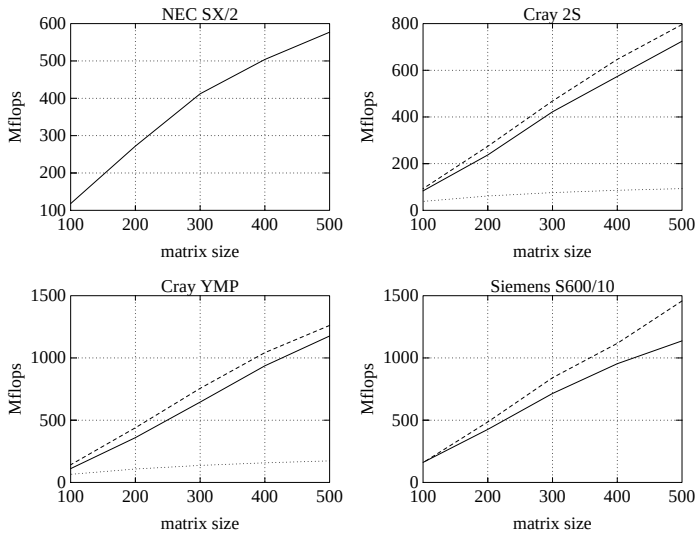


Figure 5: Blocked (dashed line), Unblocked (solid line), and LINPACK (dotted line) versions of LU factorization

value solvers deliver eigenvalues and singular values to full *relative* precision, whereas EISPACK, for example, delivers only full absolute precision. That is, if λ is the true eigenvalue, and $\hat{\lambda}$ is the computed one, then relative accuracy guarantees that

$$\frac{|\lambda - \hat{\lambda}|}{\lambda} = O(\text{ machine precision}),$$

whereas absolute accuracy guarantees only that

$$|\lambda - \hat{\lambda}| = O(\text{ machine precision}).$$

Therefore, with absolute accuracy, small eigenvalues may not be accurate at all, whereas their accuracy will be guaranteed in LAPACK. These issues are discussed in more detail in [4]. Great care has also been devoted to proper scaling of data, in order to be able to fully exploit the dynamic floating-point range of the underlying architecture for input data, and to avoid generating any spurious over- or underflows during the computation.

4 Tuning LAPACK

The LAPACK routines rely on several parameters that are machine-specific and often also problem-specific. For example, for a given problem, we need to know for what problem sizes the blocked algorithm is advantageous and, if the block algorithm is preferable, what the optimal blocksize is.

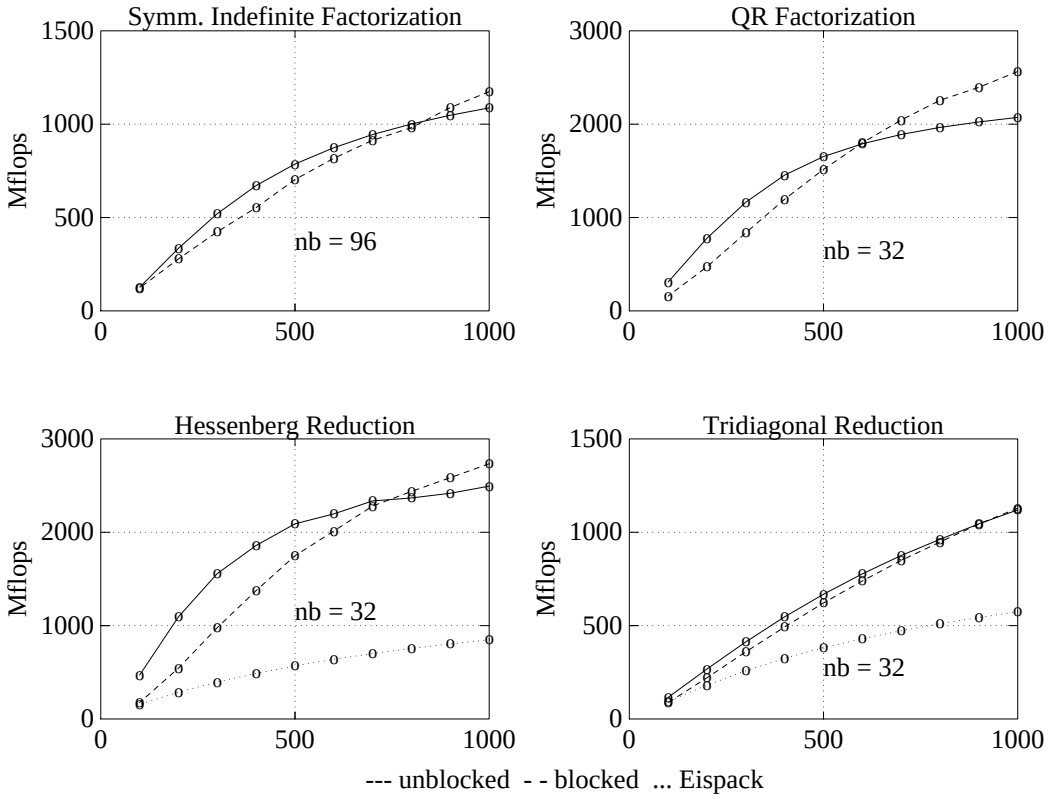


Figure 6: Performance of the Siemens S600/10 on Various Reductions on Matrices of Size 100 through 1000

.

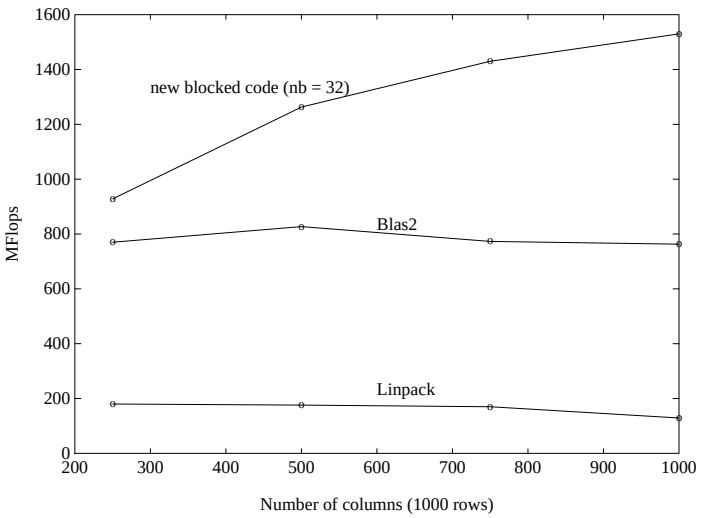


Figure 7: Performance of QR factorization with Restricted Pivoting in Comparison with Implementations of Traditional Algorithms on the CRAY Y-MP

With respect to the crossover point between blocked and unblocked algorithms, it is instructional to revisit the performance plots of the preceding sections. For some algorithms on some machines, for example, the blocked code is always better (for example, LU factorization on a CRAY-2); for other machines, it is always worse (LU factorization on a NEC SX/2); and for others, the blocked algorithm is superior from a certain problem size on (for example Hessenberg reduction on a Siemens S600/10). The last example is the common case. We give another example in Figure 8, which shows the behavior of the blocked algorithm (with blocksize $nb = 32$) and the unblocked algorithm (with blocksize $nb = 1$) on one processor of the CRAY-2/S, as well as the performance obtained when we switch from the blocked to the unblocked algorithm after the size of the submatrix still to be reduced has dropped below 144. The resulting hybrid algorithm performs better than either the blocked or the unblocked version.

Another issue that deserves closer scrutiny is the choice of the optimal block size. For most dense matrix algorithms, the choice of block size is immaterial to their numerical reliability, yet it is crucial for their computational performance. As an example, consider block algorithms for orthogonal decompositions such as the QR decomposition $A = QR$ where Q is orthogonal and R is upper triangular. For an $m \times n$ matrix partitioned into blocks of size nb, one has to compute $O(m \cdot n \cdot nb)$ extra floating-point operations in the blocked algorithm, compared to the unblocked algorithm [10, 14, 29]. These extra operations are offset by the higher speed of the Level 3 BLAS kernels (compared to the Level 2 BLAS kernels) that can now be employed. But obviously there is a tradeoff. Generation of block transformations becomes more expensive as the block size increases,

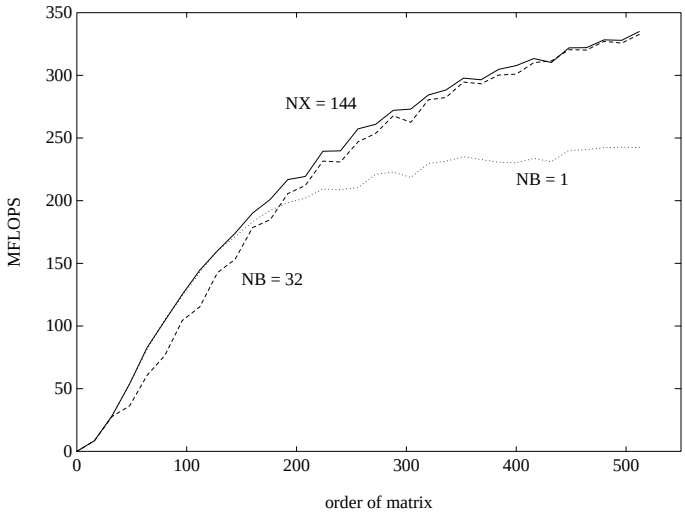


Figure 8: Effect of Crossover Point on the QR Factorization on one Processor of the Cray 2-S

whereas the block transformations perform faster with increasing block size. Take, for example, the QR factorization on an IBM RS/6000-550 workstation. As the plots in Figure 9 show, there can be quite a bit of variation in the performance achieved with different block sizes. Fortunately, the differences are often not too severe for the range of machines LAPACK is currently being run on.

We obtain access to such machine-specific information through an environment inquiry routine ILAENV, which, when given the name of the subprogram and its input parameters, returns the crossover point for the blocked versus unblocked algorithm, the optimal blocksize, and a number of other parameters. We will supply default values for this routine based on our experimental results, but we hope that the users of LAPACK will enhance this routine based on the experience they gather on their machine. We also note that the choice of an optimal blocking strategy is nontrivial. One has several possibilities for implementing blocking in a program: the use of a fixed block size (as currently done in LAPACK), the use of a varying block size determined by some a priori strategy, and the use of a varying block size determined dynamically in the course of the factorization. In [9], we suggested a methodology called *adaptive blocking* for finding "good" block sizes for pipelined factorization algorithms on distributed-memory multiprocessors. This technique was refined in [13], where we developed a recursion formula for the optimal blocking strategy.

5 Outlook

The last test release of the LAPACK package will be sent out in May of 1991, and the package will be formally released in the summer of 1991. Of course, during this effort, a variety of worthwhile

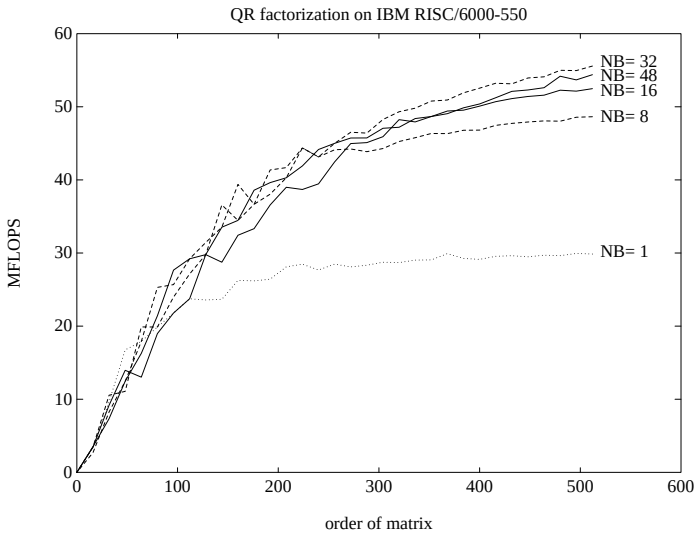


Figure 9: QR Factorization on IBM RS/6000-550 for Various Blocksizes

avenues opened up, some of which we aim to pursue in a successor project to LAPACK [3]. The goals of this new project are to

- add linear algebra routines to solve new problems;
- develop distributed-memory versions of selected LAPACK routines;
- develop C versions of the more heavily used routines, and construct a Fortran 90 interface for the driver programs;
- rewrite selected routines to exploit special properties of computer arithmetic, in particular, the IEEE Standard Floating-Point Arithmetic; and
- develop a systematic performance evaluation suite based on LAPACK.

Some of this work is already well under way. Ed Anderson, Annamaria Benzoni, Jack Dongarra, Steve Moulton, Susan Ostrouchov, Bernard Tourancheau, and Robert van de Geijn of the University of Tennessee have been defining a standard communication library for dense linear algebra computations on distributed-memory machines, the so-called BLACS (Basic Linear Algebra Communication Subprograms) [1]. Using a prototype implementation of the BLACS, they have implemented the QR, LU, Cholesky, and symmetric indefinite factorizations, as well as the reductions to Hessenberg and tridiagonal form [18, 2]. Performance results on a 128-node Intel iPSC/860 distributed-memory multiprocessor are shown in Figure 10. The performance reduction is mostly due to the high com-

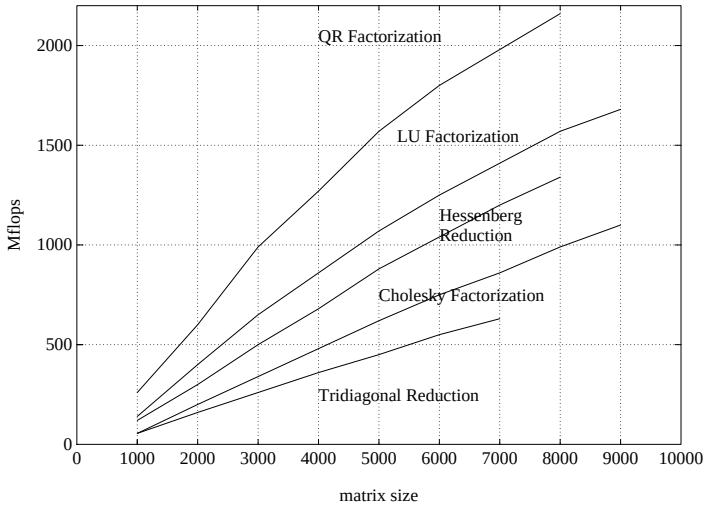


Figure 10: Performance of Selected Matrix Reductions on a 128-node Intel iPSC/860 Multiprocessor (single-precision assembler BLAS provided by Kuck and Associates)

munication costs for the iPSC/860. The time for sending a message of n single precision numbers is roughly $\alpha + n\beta$, where $\alpha = 135\mu$sec, and $\beta = 1.5\mu$sec, whereas a floating-point operation requires about 0.02μsec. However, the communication performance of the Intel Touchstone DELTA system is expected to be two orders of magnitude better. These numbers confirm that distributed-memory multiprocessors based on off-the-shelf processor technology can now reach the performance previously reserved for custom-designed vector supercomputers.

Acknowledgments

We thank Ed Anderson and Bob Van De Geijn of the University of Tennessee, as well as Klaus Geers of the University of Karlsruhe, for supplying us with many of the benchmark data.

References

[1] E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. van de Geijn. Basic Linear Algebra Communication Subprograms. In *Proceedings of the Sixth Distributed-Memory Computing Conference*. ACM Press, 1991.

[2] E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. van de Geijn. Basic Linear Algebra Communication Subprograms. In *Proceedings of the*

Fifth Siam Conference for Parallel Processing in Scientific Computing, Philadelphia, 1991. SIAM.

[3] E. Anderson, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, S. Hammarling, and W. Kahan. Prospectus for an extension to LAPACK: A portable linear algebra library for high-performance computers. Technical Report LAPACK Working Note #26, CS-90-118, Computer Science Department, The University of Tennessee, 1990.

[4] Edward Anderson, Zhaojun Bai, Christian Bischof, James Demmel, Jack Dongarra, Jeremy DuCroz, Anne Greenbaum, Sven Hammarling, Alan McKenney, and Danny Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In Joanne Martin, editor, *SUPERCOMPUTING '90*, pages 2–10, New York, 1990. ACM Press. Also LAPACK Working Note #20, CS-90-105, Computer Science Department, The University of Tennessee.

[5] Edward Anderson and Jack Dongarra. Installing and testing the initial release of LAPACK — Unix and non-Unix versions. Technical Report MCS-TM-130, Mathematics and Computer Science Division, Argonne National Laboratory, May 1989.

[6] Edward Anderson and Jack Dongarra. Evaluating block algorithm variants in LAPACK. Technical Report CS–90–103, Computer Science Department, The University of Tennessee, April 1990.

[7] Zhajoun Bai and James 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, ANL–MCS–TM–127, Mathematics and Computer Science Division, Argonne National Laboratory.

[8] Christian Bischof, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, and Danny Sorensen. LAPACK Working Note #5: Provisional contents. Technical Report ANL–88–38, Argonne National Laboratory, Mathematics and Computer Science Division, September 1988.

[9] Christian H. Bischof. Adaptive blocking in the QR factorization. *The Journal of Supercomputing*, 3(3):193–208, 1989.

[10] Christian H. Bischof. A block QR factorization algorithm using restricted pivoting. In *Proceedings SUPERCOMPUTING '89*, pages 248–256, Baltimore, Md., 1989. ACM Press.

[11] Christian H. Bischof. A block QR factorization algorithm for rank-deficient matrices. In Jack J. Dongarra, Paul Messina, Danny C. Sorensen, and Robert G. Voigt, editors, *Parallel Processing for Scientific Computing*, pages 9–14, Philadelphia, 1990. SIAM.

[12] Christian H. Bischof. *Fundamental Linear Algebra Computations on High-Performance Computers*, volume 250 of *Informatik Fachberichte*, pages 167–182. Springer Verlag, Berlin, 1990.

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

[14] Christian H. Bischof and Charles F. Van Loan. The WY representation for products of Householder matrices. *SIAM Journal on Scientific and Statistical Computing*, 8:s2–s13, 1987.

[15] Steven Carr and Ken Kennedy. Blocking linear algebra codes for memory hierarchies. In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, Philadelphia, 1990. To appear.

[16] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide*. SIAM Press, Philadelphia, 1979.

[17] Jack Dongarra, Jeremy Du Croz, Iain Duff, and Sven Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. Preprint MCS-P1-0888, Mathematics and Computer Science Division, Argonne National Laboratory, August 1988.

[18] Jack Dongarra and Susan Ostrouchov. LAPACK block factorization algorithms on the Intel iPSc/860. Technical Report LAPACK Working Note #24, TR CS-90-115, Computer Science Department, The University of Tennessee, October 1990.

[19] 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.

[20] Jack J. Dongarra and Iain S. Duff. Advanced computer architectures. Technical Report ANL–MCS–TM–57, Mathematics and Computer Science Division, Argonne National Laboratory, 1989. Revision 2.

[21] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. Van der Vorst. *Solving Linear Systems on Vector and Shared-Memory Computers*. SIAM, Philadelphia, 1991.

[22] Kyle Gallivan, Robert Plemmons, and Ahmed Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32(1):54–135, 1990.

[23] B. Garbow, J. Boyle, J. Dongarra, and C. Moler. *Matrix Eigensystem Routines — EISPACK Guide Extension*, volume 51 of *Lecture Notes in Computer Science*. Springer Verlag, New York, 1977.

[24] Nikolaus Geers. Optimization of Level 2 BLAS for Siemens VP systems. Technical Report 37.39, University of Karlsruhe, Computer Center, 1989.

[25] Helke Grasemann. Optimization of Level 3 BLAS for Siemens VP systems. Technical Report 38.89, University of Karlsruhe, Computer Center, 1989.

[26] Kai Hwang and Fayé A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, New York, 1984.

[27] C. L. Lawson, R. J. Hanson, R. J. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[28] Robert Schreiber and Jack Dongarra. Automatic blocking of nested loops. Technical Report CS–90–108, Computer Science Department, The University of Tennessee, 1990.

[29] Robert Schreiber and Charles Van Loan. A storage efficient WY representation for products of Householder transformations. *SIAM Journal on Scientific and Statistical Computing*, 10(1):53–57, 1989.

[30] B. Smith, J. Boyle, J. Dongarra, B. Garbow, Y. Ikebe, V. Klema, and C. B. Moler. *Matrix Eigensystem Routines — EISPACK Guide*. Springer Verlag, New York, 2nd edition, 1976.

[31] Harold Stone. *High-Performance Computer Architecture*. Addison-Wesley, Reading, Massachussetts, 1987.