Linear Algebra Libraries for High-Performance Computers: A Personal Perspective

Jack Dongarra
University of Tennessee and Oak Ridge National Laboratory

Linpack software for solving linear algebra problems on high-performance computers was released to the public in 1979. It has recently been joined by Lapack, which embodies the ideas of data locality and reuse.

Parallel processing is the most promising approach to designing and building high-performance computers. Parallel machines with hundreds of moderate-sized processors or thousands of very simple processors are commercially available and are being used to solve practical problems at rates comparable to the most powerful conventional supercomputers.

For more than 15 years, my colleagues and I have been developing linear algebra software for high-performance computers. We began the Linpack project in the mid-1970s with the goal of producing a package of mathematical software for solving systems of linear equations.[1] We took a careful look at how you put together a package of mathematical software and tried to design a package that would be effective on state-of-the-art computers at that time — the (scalar) CDC 7600 and the IBM system 370. Because vector machines were just beginning to emerge, we also provided some vector routines.

Linpack incorporated other features as well. Rather than simply collecting or translating existing algorithms, we reworked them. We also used a column orientation that provided greater efficiency than the traditional row orientation, and we published a users guide with directions and examples for addressing different problems. The result was a carefully designed package of mathematical software, which we released to the public in 1979.

1063-6552/93/0200-0017 $3.00 © 1993 IEEE

Table 1. Linpack Benchmark on high-performance computers.

Machine	Peak Mflops	Actual Mflops	System efficiency
IBM RS/6000-550	84	26	.31
Convex C-3810 (1 processor)	120	44	.37
Cray-1	160	27	.075
Cray X-MP/1	235	121	.51
Cray Y-MP/1 (1 processor)	333	145	.44
IBM ES/9000-520 VF	444	60	.14
ETA-10G (1 processor)	644	93	.14
Cray X-MP/4 (4 processors)	941	178	.19
Cray C-90/1 (1 processor)	952	387	.41
Convex C-3880 (8 processors)	960	86	.090
NEC SX-2	1,300	43	.033
Cray-2 (4 processors)	1,951	129	.066
Cray Y-MP/8 (8 processors)	2,664	275	.10
Hitachi S-820/80	3,000	107	.036
NEC SX-3/14 (1 processor)	5,500	314	.057
Fujitsu VP-2600/10 (1 processor)	5,000	249	.083
Cray C-90/16 (16 processors)	15,238	479	.031

Table 2. Mflops and memory bandwidth.

Machine	Peak Mflops	Peak Transfer (Mwords/sec)	Ratio
Alliant FX/80	188	22	0.12
Convex C-210	50	25	0.5
Cray-1	160	80	0.5
Cray X-MP/4	940	1,411	1.5
Cray Y-MP/8	2,667	4,000	1.5
Cray C-90-16	15,238	22,857	1.5
Cray-2S	1,951	970	0.5
Cyber 205	400	600	1.5
ETA-10G	644	966	1.5
Fujitsu VP-400	1,066	1,066	1.0
Hitachi 820/80	3,000	2,000	0.67
IBM 3090/600-VF	798	400	0.5
NEC SX-2	1,300	2,000	1.5

Table 3. Memory latency.

Machine	Latency cycles
Cray-1	15
Cray X-MP	14
Cray Y-MP	17
Cray C-90	23
Cray-2	50
Cray-2S	35
Cyber 205	50
Fujitsu VP-400	31

The Linpack Benchmark

Perhaps the best-known part of that package — indeed, some people think it is Linpack — is the so-called Linpack Benchmark that appeared in the appendix to the users guide.[1,2] It was intended to give users an idea of how long it would take to solve certain problems. We measured the time required to solve a system of equations of order 100 (a problem size we knew could be run on all the machines of interest), and we listed those times and gave some guidelines for extrapolating execution times for about 20 machines.

We gathered the times from two Linpack routines: one to factor a matrix (SGEFA), the other to solve a system of equations (SGESL). These routines, called the Basic Linear Algebra Subprograms (BLAS),[3] are where most of the floating-point computation takes place. The routine that sits in the center of that computation is a SAXPY, which takes a multiple of one vector and adds it to another vector.

Table 1 shows the Linpack Benchmark timings for some high-performance computers. Compared with their peak performance, the actual performance of these machines was quite disappointing, in spite of the fact that we used a highly vectorized algorithm on machines with vector architectures. Why were the results so bad? The answer has to do with the rate at which the machine can transfer information to and from the memory device. If we increase computational power without a corresponding increase in memory, memory access can cause serious bottlenecks.

Table 2 lists the peak megaflop rate for various machines, as well as the peak transfer rate from memory to registers (in megawords per second). Since the SAXPY operation requires three references and produces two operations, we need a ratio of 3/2 to run at

good rates. The Cray C-90 does not do badly in this respect: Each processor can transfer 1.43 gigawords (64-bit words) per second, and the complete system, from memory into the registers, runs at 22.8 gigawords per second. But for many of these machines, there is an imbalance. The Alliant FX/80 was a particularly bad case: It had a peak rate of 188 megaflops but could transfer only 22 megawords from memory, making it very hard to get peak performance (the company is no longer in business). The bottom line is: Megaflops are easy, but bandwidth — the rate at which data is moved to where the operations are performed — is difficult.

The Cray-1's performance today for the Linpack Benchmark is 27 megaflops, compared with 12 megaflops with the same Fortran code in 1983. The improved performance comes not from the machine or the applications software, but from enormous improvements in techniques for vectorizing programs over the past 10 years. In other words, the compiler can produce better optimized code.

Memory latency also affects performance: How many cycles does it take to transfer information after we make a request? Table 3 lists the memory latency for seven machines, with times ranging from 14 to 50 cycles. Obviously, a memory latency of 50 cycles will affect the algorithm's performance.

Standards development

Several years ago, the linear algebra community developed a de facto standard for identifying basic operations in its algorithms and software. We hoped that many manufacturers would implement the standard on their machines, so we could then draw on the power of this portable software library.

We began with the BLAS for vector-vector operations; we now call them the Level 1 BLAS.[3] We later defined a standard for some rather simple matrix-vector calculations: the Level 2 BLAS.[4] Still later, the basic matrix-matrix operations were identified, and the Level 3 BLAS were defined.[5] (See Figure 1.) We developed three standards to take advantage of the fact that machines have a memory hierarchy, and that the faster memory is at the top (see Figure 2). To get as much use of or access to the data as possible, we would like to keep it at the top. The higher-level BLAS let us do just that. The Level 2 BLAS offer the potential for two floating-point operations for every reference; with the Level 3 BLAS we get essentially n operations for every two accesses, or the maximum possible (see Table 4).

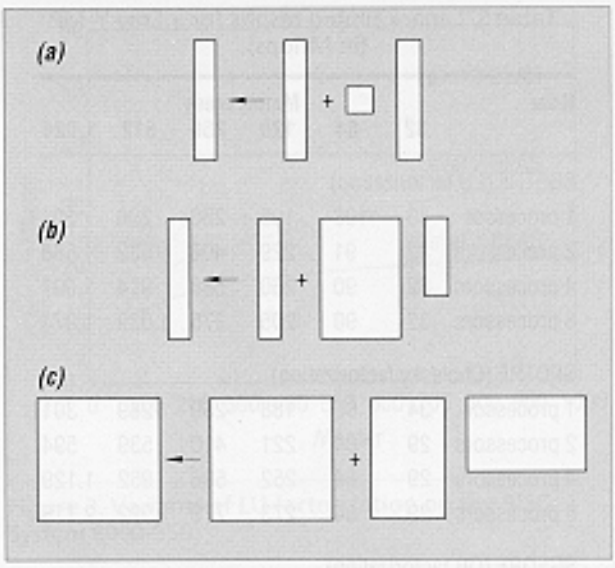

Figure 1. (a) The Level-1 BLAS for vector-vector operations ($y \leftarrow y + \alpha x$; also dot product, ...); (b) the Level-2 BLAS for matrix-vector operations ($y \leftarrow y + Ax$; also triangular solve, rank-1 update); (c) the Level-3 BLAS for matrix-matrix operations ($A \leftarrow A + BC$; also block triangular solve, rank-k update).

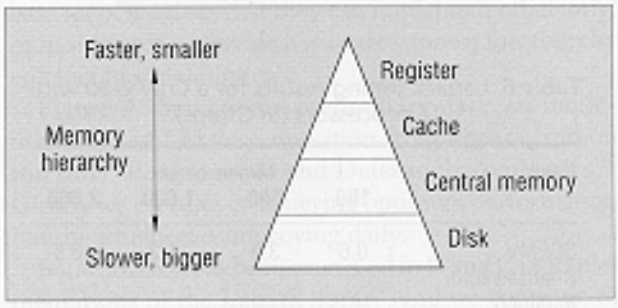

Figure 2. Memory hierarchy.

Table 4. Capabilities of higher-level BLAS.

BLAS	Memory references	Flops	Flops/memory reference
Level 1 $y \leftarrow y + \alpha x$	$3n$	$2n$	2/3
Level 2 $y \leftarrow y + Ax$	n^2	$2n^2$	2
Level 3 $A \leftarrow A + BC$	$4n^2$	$2n^3$	$n/2$

On some parallel machines, the higher-level BLAS also provide increased granularity, the possibility of parallel operations, and lower synchronization costs. Of course, nothing comes free. We must rewrite our algorithms to use the BLAS effectively. In particular, we

Table 5. Lapack timing results for a Cray Y-MP (in Mflops).

Name	Matrix order					
	32	64	128	256	512	1,024
SGETRF (LU factorization)						
1 processor	40	108	195	260	290	304
2 processors	32	91	229	408	532	588
4 processors	32	90	260	588	914	1,097
8 processors	32	90	205	375	1,039	1,974
SPOTRF (Cholesky factorization)						
1 processor	34	95	188	259	289	301
2 processors	29	84	221	410	539	594
4 processors	29	84	252	598	952	1,129
8 processors	29	84	273	779	1,592	2,115
SGEQRF (QR factorization)						
1 processor	54	139	225	275	294	301
2 processors	50	134	256	391	505	562
4 processors	50	136	292	612	891	1,060
8 processors	50	133	328	807	1,476	1,937

Table 6. Lapack timing results for a Cray C-90 with 16 processors (in Gflops).

Name	Matrix order			
	100	500	1,000	2,000
SGEMV (matrix-vector multiply)	0.6	3.6	11.2	13.5
SGEMM (matrix-vector multiply)	10.7	13.9	14.2	14.2
SGETRF (LU factorization)	0.4	4.1	7.4	10.0
SPOTRF (LL^T factorization)	0.4	4.7	8.4	11.3
SGEQRF (QR factorization)	0.3	3.8	7.6	10.9
SGEHRD (reduct to Hessenberg)	0.4	4.7	8.9	11.5
SGEBRD (reduct to bidiagonal form)	0.3	3.7	7.5	10.8
SSYTRD (reduct to tridiagonal form)	0.3	3.2	6.5	9.8

need to develop blocked-partitioned algorithms that can exploit the matrix-matrix operations.[6]

The development of these "blocked" algorithms is a fascinating example of history repeating itself. In the 1960s, machines with very small main memories used tapes as primary storage. The programmer reeled in information from the tapes, put it into memory, and tried to get as much access as possible before sending it out again. Today, we are reorganizing our algorithms with that same idea, but instead of tapes and main memory, we are dealing with vector registers, caches, and so forth.

Lapack

Lapack is a new linear algebra library that embodies these ideas of locality of reference and data reuse. Built on top of the Level 1, 2, and 3 BLAS, Lapack is based on algorithms that minimize memory access while solving systems of equations and eigenvalue problems efficiently across a wide range of high-performance computers. The institutions involved in the Lapack project include the University of Tennessee, the University of California at Berkeley, Numerical Algorithms Group, New York University's Courant Institute, Rice University, Argonne National Laboratory, and Oak Ridge National Laboratory. Our work in algorithm design has been supported by tool development projects throughout the country, especially at Rice University and the University of Illinois. Other projects have helped with what we might call logic or performance debugging: trying to understand what an algorithm is doing when it runs on a parallel machine to give the implementor a better feeling for where to focus attention.[7]

We recently completed the package, and we are beginning to see some impressive results. Table 5 shows some timing results for some Lapack routines written in Fortran and provided by Cray. On one processor of a Cray Y-MP, the LU factorization routine runs at 40 megaflops for a matrix of order 32, and at 300 megaflops for a matrix of order 1,000. On eight processors, this same routine actually slows down to only 32 megaflops (obviously, we should not use eight processors to solve this matrix problem). But when we go to large-order matrices, the execution rate is nearly 2 gigaflops — and on a very portable piece of software. We get the same effect for LLT and QR factorization. (Note that we are doing the same number of operations as with the unblocked algorithms; we are not cheating in terms of the megaflop rate here.) Table 6 shows some timing

results for the Level-3 BLAS routines (provided by Cray and written in Cray assembly language) and for some Lapack routines; the performance is impressive.

For comparison, Figure 3 plots the speed of LU decomposition using a Fortran implementation of the Level-3 BLAS on the IBM RISC machine RS/6000-550. This one-processor workstation runs at around 50 megaflops on larger-order matrices. Clearly the BLAS help not only on the high-performance machines, but also on RISC machines for exactly the same reason: Data is being used or reused in its cache.

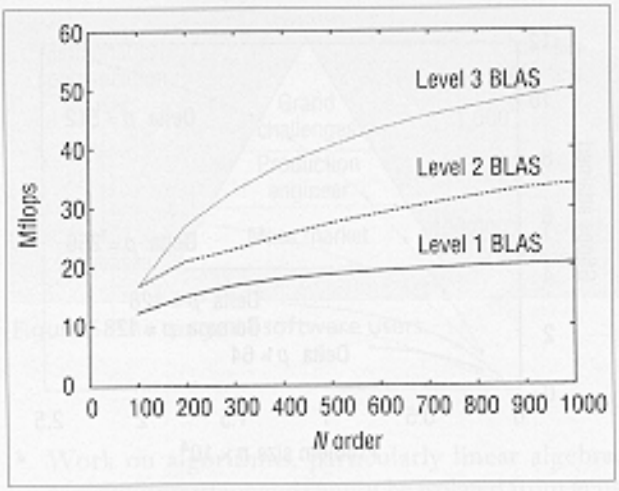

Figure 3. Variants of LU factorization on the RISC System 6000-550.

Algorithm design

When we restructure an algorithm, the basic algorithm remains the same. When we changed the Linpack algorithms to block form, for example, we affected only the locality of how we reference data and the independence of the operations we are focusing on (the matrix-matrix operations). When we design an algorithm, on the other hand, the basic algorithm changes. Let's consider a divide-and-conquer technique for finding the eigenvalues and eigenvectors of a symmetric tridiagonal matrix.[8] In other fields, this technique is sometimes called domain decomposition. It involves tearing or partitioning the problem into small, independent pieces, finding the eigenvalue of each piece, combining the eigenvalues of pairs of pieces in parallel, and then pairing successive pairs until the complete set is determined. We redesigned this algorithm to run in parallel very efficiently; on a Cray-2, we are getting a factor-of-4 speedup, and sometimes better (see Table 7). What's more, the algorithm is more efficient than even the "best" sequential algorithm on a sequential architecture.

The future of Lapack

We have already started looking at "cosmetic changes" for Lapack, adapting it semiautomatically for distributed-memory, highly parallel architectures. Our work on blocked operations will be appropriate, because they minimize communication and provide a good surface-to-volume ratio. We also expect to need yet another set of routines, this one based on a set of message-passing standards for linear algebra, called the Basic Linear Algebra Communication Subprograms, (BLACS).[9] Once again, these operations will draw on what has been done in the community.

The main advantages of a message-passing standard are portability and ease of use. The benefits of standardization are particularly apparent in a distributed-memory communication environment where higher level routines or abstractions are built on lower level message-passing routines. Furthermore, a message-passing standard would give vendors a clearly defined base set of routines that they can implement efficiently, or in some cases provide hardware support for, thereby enhancing scalability.

Figure 4 shows some preliminary data: an implementation of LU decomposition from Lapack, run on the Intel iPSC Gamma and Delta multicomputers.[10] Clearly we are not yet achieving optimum performance, but the situation is improving daily.

Some interest has been expressed in C and C++ implementations of the Lapack library, and we continue to track Fortran 90 and High-Performance Fortran; we have a pilot project to develop a core set of routines for each of these languages.

Table 7. Ratio of execution time for Eispack TQL2 algorithm over divide-and-conquer algorithm.

No. of processors	100 E/(p)	100 (1)/(p)	200 E/(p)	200 (1)/(p)	300 E/(p)	300 (1)/(p)
1	1.35	1	1.45	1	1.53	1
2	2.55	1.88	2.68	1.84	2.81	1.84
3	3.39	2.51	3.71	2.55	3.79	2.48
4	4.22	3.12	4.60	3.17	50.3	3.28

E = Eispack TQL2
(1) = parallel divide-and-conquer algorithm on one processor
(P) = parallel divide-and-conquer algorithm on P processors

7. J. Dongarra et al., "A Tool to Aid in the Design, Implementation, and Understanding of Matrix Algorithms for Parallel Processors," *J. Parallel and Distributed Computing*, Vol. 9, No. 2, June 1990, pp. 185-202.

8. J. Dongarra and D. Sorensen, "A Fully Parallel Algorithm for the Symmetric Eigenproblem," *SIAM J. Scientific Statistical Computing*, Vol. 8, No. 2, Mar. 1987, pp. 139-154.

9. J. Dongarra, "Lapack Working Note 34: Workshop on the BLACS," Technical Report CS-91-134, Computer Science Dept., University of Tennessee, Knoxville, Tenn., 1991.

10. D. Walker, J. Dongarra, and R. van de Geijn, "A Look at Scalable Dense Linear Algebra Libraries," *Proc. Scalable High-Performance Computing Conf.*, IEEE Press, Piscataway, N.J., 1992, pp. 372-379.

11. E. Anderson et al., *Lapack User's Guide*, Soc. for Industrial and Applied Math., Philadelphia, 1992.

Jack Dongarra holds a joint appointment as distinguished professor of computer science in the Computer Science Department of the University of Tennessee, and distinguished scientist in the Mathematical Sciences Section of Oak Ridge National Laboratory, under the UT/ORNL Science Alliance program. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced computer architectures, programming methodologies, and tools for parallel computers. He helped to design and implement Eispack, Linpack, the BLAS, Lapack, and PVM/Hence, and he is currently involved in designing algorithms and techniques for high-performance computer architectures.

Dongarra received a PhD in applied mathematics from the University of New Mexico in 1980, an MS in computer science from the Illinois Institute of Technology in 1973, and a BS in mathematics from Chicago State University in 1972. He is a member of the IEEE, ACM, and the Society for Industrial and Applied Mathematics, and he is editor-in-chief of the *International Journal of Supercomputer Applications*.

Readers can contact Dongarra at the Dept. of Computer Science, University of Tennessee, Knoxville, TN 37996-1301; Internet, dongarra@cs.utk.edu

1993 Gordon Bell Prize

For Outstanding Achievements in the Application of Parallel Processing to Practical Scientific and Engineering Problems

The due date for entries is **May 31, 1993**. Finalists will be announced by June 30, 1993, and winners will be announced at Supercomputing 93 in November.

Oil exploration and extraction, integrated-circuit design, evolution research, space-shuttle modeling, turbulence simulation, and mortgage-rate determination are some of the applications addressed by previous prize recipients.

Awards totaling $3,000 will be given. Entries in the following four categories will be considered:

- **Performance:** The entrant will be expected to convince the judges that the submitted program is running faster than any other comparable engineering or scientific application. Suitable evidence will be the megaflop rate based on actual operation counts or the solution of the same problem with properly tuned code on a machine of known performance, such as a Cray Y-MP. If neither of these measurements can be made, the submitter should document the performance claims as well as possible.
- **Price/performance:** The entrant must show that the performance of the application divided by the list price of the smallest system needed to achieve the reported performance is better than that of any other entry. Performance measurements will be evaluated as for the performance prize. Only the cost of the CPUs, memory, and any peripherals critical to the application need be included in the price. For example, if the job can be run on diskless compute servers, the cost of disks, keyboards, and displays need not be included.
- **Compiler parallelization:** Judges are looking for the combination of compiler and application that generates the greatest speedup. Speedup will be measured by dividing the wall-clock time of the parallel run by that of a good serial implementation of the same job. These may be the same program if the entrant can convince the judges that the serial code is a good choice for a uniprocessor. Compiler directives and new languages are permitted. However, anyone submitting an entry in other than a standard, sequential language will have to convince the judges that the parallelism was detected by the compiler, not by the programmer.
- **Speedup:** By determining how much faster a job runs on N processors than on just one of the processors in a parallel machine, speedup serves as a measure of how effectively the parallelism is being used. The more nearly the speedup approaches the number of processors N, the more impressive the results. But high speedup is hard to achieve, even with a very large number of processors. The speedup award recognizes those who overcome these difficulties.

General conditions include demonstrating the utility of the program and machine. The judges will also consider how much the entry advances the state of the art in some field.

Contestants should send a three- or four-page executive summary by **May 31, 1993** to:

1993 Gordon Bell Prize
c/o Marilyn Potes
IEEE Computer Society
10662 Los Vaqueros Cir., PO Box 3014
Los Alamitos, CA 90720-1264