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).

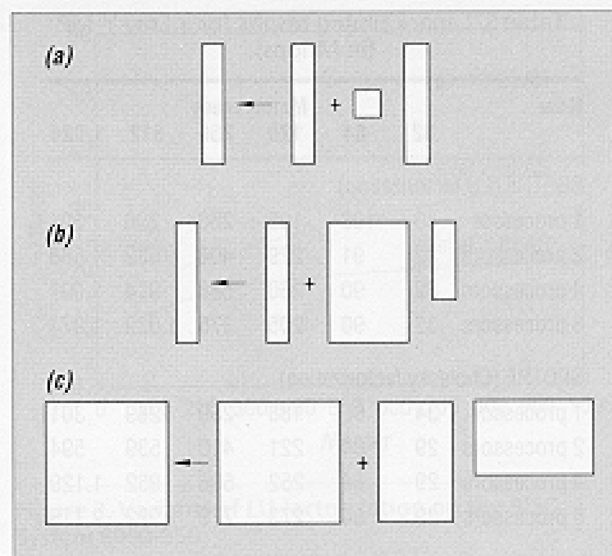

Figure 1. (a) The Level-1 BLAS for vector-vector operations ($y \leftarrow y + \alpha x$; also dot product, ...); (b) the Level-2 BLAS for matrix-vector operations ($y \leftarrow y + Ax$; also triangular solve, rank-1 update); (c) the Level-3 BLAS for matrix-matrix operations ($A \leftarrow A + BC$;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

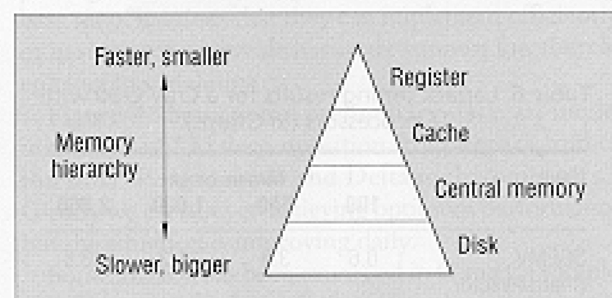

Figure 2. Memory hierarchy.

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