The LINPACK Benchmark: An Explanation*

Jack J. Dongarra

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439-4844

1. Introduction

In this paper we will clear up some of the confusion and mystery surrounding the LINPACK Benchmark. We will examine what is measured and describe how to interpret the results of the programs. But first a bit of history.

The LINPACK Benchmark is, in some sense, an accident. It was originally designed to assist users of the LINPACK package [9] by providing information on execution times required to solve a system of linear equations. The first "LINPACK Benchmark" report appeared as an appendix in the LINPACK Users' Guide [9] in 1979. The appendix comprised data for one commonly used path in LINPACK for a matrix problem of size 100, on a collection of widely used computers (23 in all), so users could estimate the time required to solve their matrix problem.

Over the years other data was added, more as a hobby than anything else, and today the collection includes 200 different computer systems. In addition the scope of the benchmark has also expanded. The benchmark report describes the performance at three levels of problem size and optimization opportunity: 100×100 problem - inner loop optimization, 300×300 problem - two loop optimization, and 1000×1000 problem - three loop optimization (whole problem).

1. The LINPACK Package

The LINPACK package is a collection of Fortran subroutines for solving various systems of linear equations. The software in LINPACK is based on a *decompositional* approach to numerical linear algebra. The general idea is the following. Given a problem involving a matrix, A, one factors or decomposes A into a product of simple, well-structured matrices which can be easily manipulated to solve the original problem. The package has the capability

* Work supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U. S. Department of Energy, under Contract W-31-109-Eng-38.

of handling many different matrix types and different data types, and provides a range of options.

The package itself is based on another package, called the Basic Linear Algebra Subroutines (BLAS)[25]. The BLAS address simple vector operations, such as adding a multiple of a vector to another vector (SAXPY) or forming an inner product (SDOT). Most of the floating-point work within the LINPACK algorithms is carried out by the BLAS, which makes it possible to take advantage of special computer hardware without having to modify the underlying algorithm. This approach thus achieves transportability and clarity of software without sacrificing reliability.

1. Selection of the Algorithm

Solving a system of equations requires $O(n^3)$ floating-point operations, more specifically, $2/3n^3 + 2n^2 + O(n)$ floating-point additions and multiplications. Thus, the time required to solve such problems on a given machine is simply

$$time_n = time_{100} \times \frac{n^3}{100^3}.$$

In the LINPACK Benchmark, a matrix of size 100 is used as the base since it represents a "large enough" problem. That is, the $O(n^2)$ term in the operation count does not have a major effect on the time for large n. Another reason for choosing the matrix size as 100 is that it requires only $O(100^2)$ floating-point elements and can be accommodated in most environments.

The algorithm used in the timings is based on LU decomposition with partial pivoting. The matrix type is real, general, and dense, with matrix elements randomly distributed in the range $(-1.0, 1.0)$. (The random number generator used in the benchmark is not sophisticated; rather its major attribute is its compactness. The generator produces pseudo-random elements and has a period that is rather short, 2^{14}. As a result, if one uses this generator for a matrix of order 800, the matrix will be singular to working precision.)

4. The LINPACK Benchmark

4.1 Operations

The LINPACK benchmark features two routines: SGEFA and SGESL (these are the single-precision versions of the routines; DGEFA and DGESL are the double-precision counterparts). SGEFA performs the decomposition with partial pivoting, and SGESL uses that decomposition to solve the given system of linear equations. Most of the time - $O(n^3)$ floating-point operations - is spent in SGEFA. Once the matrix has been decomposed, SGESL is used to find the solution; this requires $O(n^2)$ floating-point operations.

SGEFA and SGESL in turn call three BLAS routines: SAXPY, ISAMAX, and SSCAL. By far the major portion of time - over 90% at order 100 - is spent in SAXPY. SAXPY is used to multiply a scalar, α, times a vector, x, and add the results to another vector, y. It is called approximately $n^2/2$ times by SGEFA and $2n$ times by SGESL with vectors of varying length. The statement $y_i \leftarrow y_i + \alpha x_i$, which forms an element of the SAXPY operation, is executed approximately $\frac{n^3}{3} + n^2$ times, which gives rise to roughly $2/3n^3$ floating-point operations in the solution. Thus, the benchmark requires roughly 2/3 million floating-point operations.

The statement $y_i \leftarrow y_i + \alpha x_i$, besides the floating-point addition and floating-point multiplication, involves a few one-dimensional index operations and storage references. While the LINPACK routines SGEFA and SGESL involve two-dimensional arrays references, the BLAS refer to one-dimensional arrays. The LINPACK routines in general have been organized to access two-dimensional arrays by column. In SGEFA, the call to SAXPY passes an address into the two-dimensional array A, which is then treated as a one-dimensional reference within SAXPY. Since the indexing is *down* a column of the two-dimensional array, the references to the one-dimensional array are sequential with unit stride. This is a performance enhancement over, say, addressing across the column of a two-dimensional array. Since Fortran dictates that two-dimensional arrays be stored by column in memory, accesses to consecutive elements of a column lead to simple index calculations. References to consecutive elements differ by one word instead of by the leading dimension of the two-dimensional array.

4.2 Precision

In discussions of scientific computing, one normally assumes that floating-point computations will be carried out to 64-bit precision, that is, single precision on CDC or CRAY and double precision on IBM or UNIVAC. In the benchmark there are two sets of numbers reported, one for full precision and the other for half precision. Full precision refers to 64-bit arithmetic or the equivalent, and half precision is 32-bit floating-point arithmetic or the equivalent.

4.3 Timing Information

The results in the report reflect only one problem area: solving dense systems of equations using the LINPACK programs in a Fortran environment. In particular, since most of the time is spent in SAXPY, the benchmark is really measuring the performance of SAXPY. The average vector length for the algorithm used to compute LU decomposition with partial pivoting is $2/3n$[15]. Thus in the benchmark with $n = 100$, the average vector length is 66.

5. Loop Unrolling

It is frequently observed that the bulk of the central processor time for a program is localized in 3% or less of the source code [27]. Often the critical code (from a timing perspective) consists of one or a few short inner loops typified, for instance, by the scalar product of two vectors. On scalar computers a simple technique for optimizing of such loops should then be most welcome. "Loop unrolling" (a generalization of "loop doubling") applied selectively to time-consuming loops is just such a technique[24, 17].

When a loop is unrolled, its contents are replicated one or more times, with appropriate adjustments to array indices and loop increments. Consider, for instance, the SAXPY sequence, which adds a multiple of one vector to a second vector:

```
      DO 10 i = 1,n
         y(i) = y(i) + alpha*x(i)
   10 CONTINUE
```

Unrolled to a depth of four, it would assume the following form:

```
      m = n - MOD(n,4)
      DO 10 i = 1,m,4
         y(i) = y(i) + alpha*x(i)
         y(i+1) = y(i+1) + alpha*x(i+1)
         y(i+2) = y(i+2) + alpha*x(i+2)
         y(i+3) = y(i+3) + alpha*x(i+3)
   10 CONTINUE
```

In this recoding, four terms are computed per loop iteration, with the loop increment modified to count by fours. Additional code has to be added to process the $MOD(N,4)$ elements remaining upon completion of the unrolled loop, should the vector length not be a multiple of the loop increment. The choice of four was for illustration, with the generalization to other orders obvious from the example. Actual choice of unrolling depth in a given instance would be guided by the contribution of the loop to total program execution time and by consideration of architectural constraints.

Why does unrolling enhance loop performance? First, there is the direct reduction in loop overhead -- the increment, test, and branch function -- which, for short loops, may actually dominate execution time per iteration. Unrolling simply divides the overhead by a factor equal to the unrolling depth, although additional code required to handle "leftovers" will reduce this gain somewhat. Clearly, savings should increase with increasing unrolling depth, but the marginal savings fall off rapidly after a few terms. The reduction in overhead is the primary source of improvement on "simple" computers.

Second, for advanced architectures employing segmented functional units, the greater density of non-overhead operations permits higher levels of concurrency within a particular segmented unit. Thus, in the SAXPY example, unrolling would allow more than one multiplication to be concurrently active on a segmented machine such as the CDC 7600 or IBM

370/195. Optimal unrolling depth on such machines might well be related to the degree of functional unit segmentation.

Third, and related to the above, unrolling often increases concurrency between independent functional units on computers so equipped. Thus, in our SAXPY example, a CDC 7600, with independent multiplier and adder units, could obtain concurrency between addition for one element and multiplication for the following element, besides the segmentation concurrency obtainable within each unit.

However, with vector computers and their compilers trying to detect vector operations, the unrolling technique had the opposite effect. The unrolling inhibited vectorization of the loop, the resulting vector code became scalar, and the performance suffered. As a result, the Fortran implementation of the BLAS when run on vector machines should not be unrolled. The LINPACK benchmark notes this by a reference in the table to rolled Fortran.

5. Performance

The performance of a computer is a complicated issue, a function of many interrelated quantities. These quantities include the application, the algorithm, the size of the problem, the high-level language, the implementation, the human level of effort used to optimize the program, the compiler's ability to optimize, the age of the compiler, the operating system, the architecture of the computer, and the hardware characteristics. The results presented for benchmark suites should not be extolled as measures of total system performance (unless enough analysis has been performed to indicate a reliable correlation of the benchmarks to the workload of interest) but, rather, as reference points for further evaluations.

Performance is often measured in terms of Megaflops, millions of floating point operations per second (MFLOPS). We usually include both additions and multiplications in the count of MFLOPS, and the reference to an operation is assumed to be on 64-bit operands.

The manufacturer usually refers to peak performance when describing a system. This peak performance is arrived at by counting the number of floating-point additions and multiplications that can be a period of time, usually the cycle time of the machine. As an example, the CRAY-1 has a cycle time of 12.5 nsec. During a cycle the results of both the adder and multiplier can be completed

$$\frac{2\ operations}{1\ cycle} * \frac{1\ cycle}{12.5 nsec} = 160 MFLOPS.$$

Table 1 displays the peak performance for a number of high-performance computers.

Table 1
Theoretical Peak Performance

Machine	Cycle time, nsec	Number of Processors,	Peak Performance, MFLOPS
Culler PSC	200	1	5
Multiflow TRACE 7/200	130	1	15
CONVEX C-1	100	1	20
SCS-40	45	1	44
FPS 264	38	1	54
Alliant FX/8	170	8	94
Amdahl 500	7.5	1	133
CRAY-1	12.5	1	160
CRAY X-MP-1	9.5	1	210
IBM 3090/VF-200	18.5	2	216
Amdahl 1100	7.5	1	267
NEC SX-1E	7	1	325
CDC CYBER 205	20	1	400
CRAY X-MP-2	9.5	2	420
IBM 3090/VF-400	18.5	4	432
Amdahl 1200	7.5	1	533
NEC SX-1	7	1	650
CRAY X-MP-4	9.5	4	840
Hitachi S-810/20	14	1	840
NEC SX-2	6	1	1300
CRAY-2	4.1	4	2000

By peak theoretical performance we mean only that the manufacturer guarantees that programs will not exceed these rates, sort of a *speed of light* for a given computer. At one time, a programmer had to go out of his way to code a matrix routine that would not run at nearly top efficiency on any system with an optimizing compiler. Owing to the proliferation of exotic computer architectures, this situation is no longer true.

The LINPACK Benchmark illustrates this point quite well. In practice, as Table 2 shows, there may be a significant difference between peak theoretical and actual performance [8]:

Table 2
LINPACK Benchmark
Solving a 100 x 100 Matrix Problem

Machine	Peak Performance, MFLOPS	Actual Performance, MFLOPS	System Efficiency
Culler PSC	5	2	.40
Multiflow TRACE 7/200	15	6	.40
CONVEX C-1	20	3.0	.15
SCS-40	44	8.0	.18
FPS 264	54	5.6	.10
Alliant FX/8	94	7.6 (8 proc)	.08
Amdahl 500	133	14	.11
CRAY-1	160	12	.075
CRAY X-MP-1	210	24	.11
IBM 3090/VF-200	216	12 (1 proc)	.11(.056)
Amdahl 1100	267	16	.060
NEC SX-1E	325	35	.11
CDC CYBER 205	400	17	.043
CRAY X-MP-2	420	24 (1 proc)	.11(.057)
IBM 3090/VF-400	432	12 (1 proc)	.11(.028)
Amdahl 1200	533	18	.034
NEC SX-1	650	39	.06
CRAY X-MP-4	840	24 (1 proc)	.11(.029)
Hitachi S-810/20	840	17	.020
NEC SX-2	1300	46	.035
CRAY-2	2000	15 (1 proc)	.030(.0075)

If we examine the algorithm used in LINPACK and look at how the data are referenced, we see that at each step of the factorization process there are vector operations that modify a full submatrix of data. This update causes a block of data to be read, updated, and written back to central memory. The number of floating-point operations is $\frac{2}{3}n^3$, and the number of data references, both loads and stores, is $\frac{2}{3}n^3$. Thus, for every *add/multiply* pair we must perform a load and store of the elements, unfortunately obtaining no reuse of data. Even though the operations are fully vectorized, there is a significant bottleneck in data movement, resulting in poor performance. On vector computers this translates into two vector operations and three vector-memory references, usually limiting the performance to well below peak

rates. To achieve high-performance rates, this *operation-to-memory-reference rate* must be higher.

In some sense this is a problem with doing vector operations on a vector machine. The bottleneck is in moving data and the rate of execution are limited by these quantities. We can see this by examining the rate of data transfers and the peak performance.

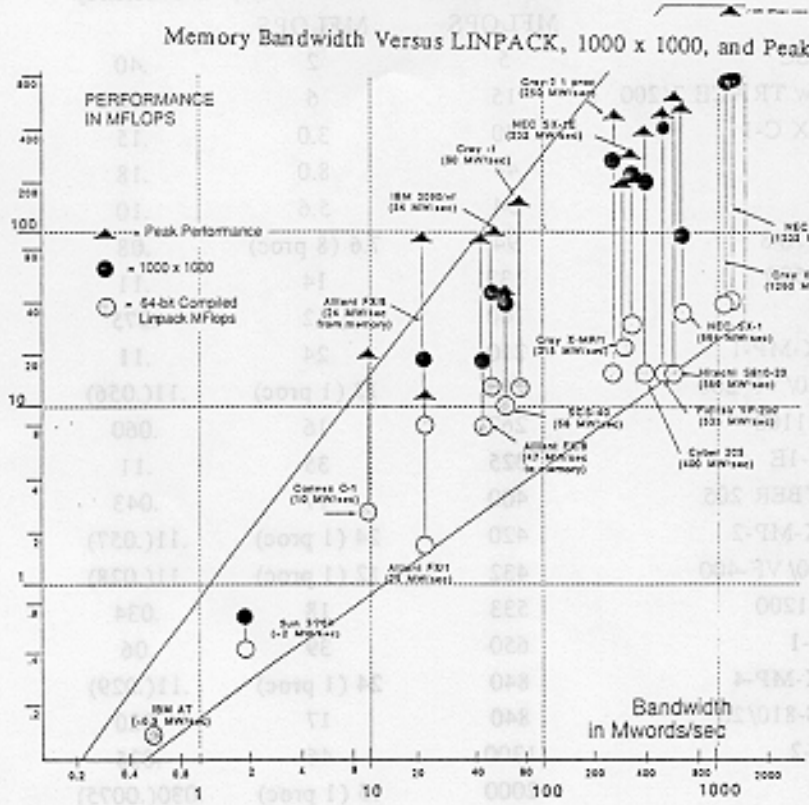

Figure 1

Memory Bandwidth and Peak Performance

To understand why the performance is so poor, considering the basic operation performed, a SAXPY. There are two parameters, $n_{\frac{1}{2}}$ and r_{∞}, that reflect the hardware performance of the idealized generic computer and give a first-order description of any real computer. These characteristic parameters are defined as follows:

$n_{\frac{1}{2}}$ - *the half-performance length* - the vector length required to achieve half the maximum

performance;

r_∞ - *the maximum or asymptotic performance* - the maximum rate of computation in units of equivalent scalar operaitons performed per second (MFLOPS).[22].

SAXPY

Machine	$n_{1/2}$	r_∞	peak
Alliant FX/1	50	3	12
CONVEX C-1	31	10	20
SCS-40	20	18	44
Alliant FX/8 (for 8 processors)	150	14	94
IBM 3090/VF (per processor)	34	53	108
CRAY 1-S	20	45	160
CRAY X-MP (per processor)	37	101	210
Fujitsu VP-100	200	140	260
NEC SX-1E	20	120	325
CYBER 205 (2-pipe)	238	170	400
CRAY-2 (per processor)	30	55	500
Fujitsu VP-200	120	190	533
NEC SX-1	30	240	650
NEC SX-2	80	575	1300

SDOT

Machine	$n_{1/2}$	r_∞	peak
Alliant FX/1	150	3	12
CONVEX C-1	56	9	20
SCS-40	300	30	44
Alliant FX/8 (for 8 processors)	220	20	94
IBM 3090/VF (per processor)	41	52	108
CRAY 1-S	200	70	160
CRAY X-MP (per processor)	270	120	210
Fujitsu VP-100	200	180	260
NEC SX-1E	300	220	325
CYBER 205 (2-pipe)	135	90	400
CRAY-2 (per processor)	80	45	500
Fujitsu VP-200	320	350	533
NEC SX-1	390	350	650
NEC SX-2	480	575	1300

The information presented on SAXPY and SDOT was generated by running the following loops as in-line code and measuring the time to perform the operations.

SAXPY	SDOT
DO 10 i = 1,n	DO 10 i = 1,n
y(i) = y(i) + alpha * x(i)	s = s + x(i) * y(i)
10 CONTINUE	10 CONTINUE

The BLAS operate only on vectors. The algorithms as implemented tend to do more data movement than is necessary. As a result, the performance of the routines in LINPACK suffers on high-performance computers where data movement is as costly as floating-point operations.

In Figure 2 we display a graph comparing the price-performance of various computer systems based on the results from the first LINPACK benchmark (an example of applications programs where there are short vectors and many references to memory, both of which limit performance on today's high-performance computers). The performance data are taken from the first table in the benchmark report, solving a matrix of order 100 using the LINPACK software in full precision, in a Fortran environment. The price information is based on the best available information for the list price of a system. Since the graph is log-log, small

changes will not affect the points greatly.

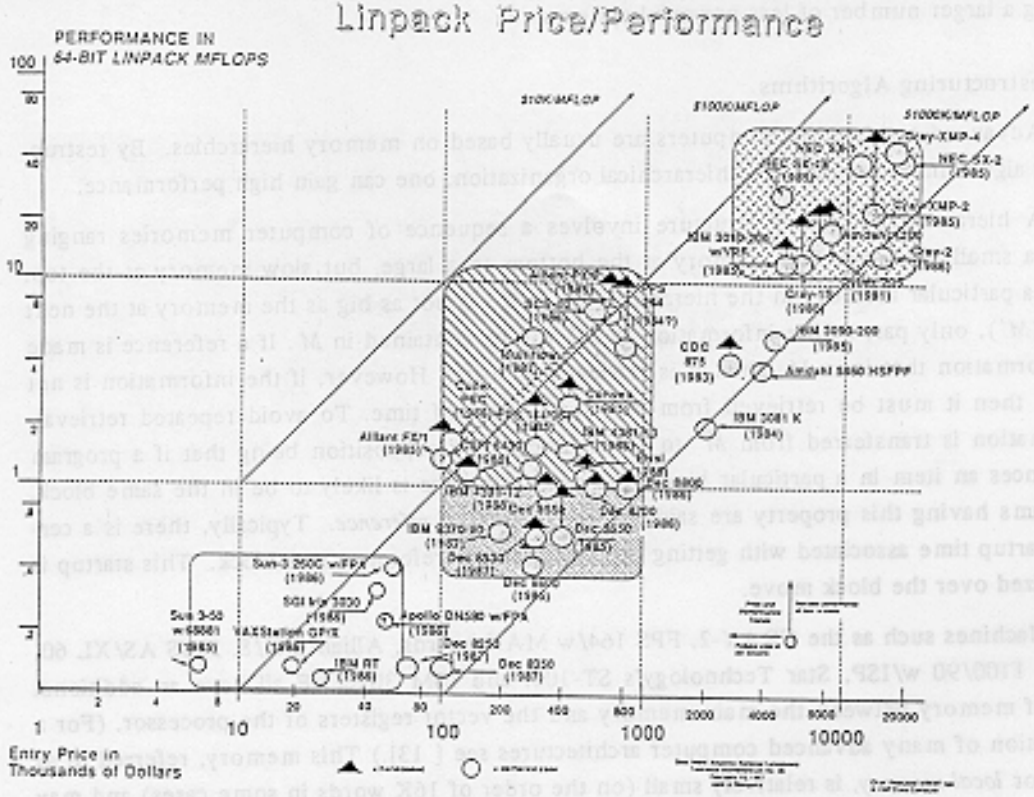

Figure 2
LINPACK Price Performance

The machines depicted in Figure 2 fall into four broad categories: supercomputer, mini-supercomputer, super-minicomputer, and workstation. The supercomputers cost between $1 million and $10 million and have a performance between 10 and 100 MFLOPS on the LINPACK benchmark. The mini-supercomputers cost between $100 thousand and $1 million and are in the range of 1 to 10 MFLOPS. The super-minis, like the mini-supercomputers, cost between $100 thousand and $1 million; however, their performance on this benchmark is an order of magnitude less (.1 to 1 MFLOPS). The workstations cost between $10 thousand and $100 thousand and are in the same range as the super-minis for performance.

As plotted, the best price-performance computers (for the LINPACK routines) will be those furthest to the left of the line that represents $100K/MFLOPS. In terms of these data, the best price-performance machines appear to be Sun-3 workstations, the Multiflow, and the Culler IPSC. Bear in mind that this price-performance benefit will be of little use to scientists

who require the large memory or full power of a supercomputer. Nevertheless, it does raise an interesting question concerning the tradeoffs between having many users share the largest and most powerful machine available, and having the same number of users distributed among a larger number of less powerful systems.

7. Restructuring Algorithms

Advanced-architecture computers are usually based on memory hierarchies. By restructuring algorithms to exploit this hierarchical organization, one can gain high performance.

A hierarchical memory structure involves a sequence of computer memories ranging from a small, but very fast memory at the bottom to a large, but slow memory at the top. Since a particular memory in the hierarchy (call it M) is not as big as the memory at the next level (M'), only part of the information in M' will be contained in M. If a reference is made to information that is in M, then it is retrieved as usual. However, if the information is not in M, then it must be retrieved from M', with a loss of time. To avoid repeated retrieval, information is transferred from M' to M in blocks, the supposition being that if a program references an item in a particular block, the next reference is likely to be in the same block. Programs having this property are said to have *locality of reference*. Typically, there is a certain startup time associated with getting the first memory reference in a block. This startup is amortized over the block move.

Machines such as the CRAY-2, FPS 164/w MAX boards, Alliant FX/8, NAS AS/XL 60, Sperry 1100/90 w/ISP, Star Technology's ST-100, and IBM 3090/VF all have an additional level of memory between the main memory and the vector registers of the processor. (For a description of many advanced computer architectures see [13].) This memory, referred to as *cache*, or *local memory*, is relatively small (on the order of 16K words in some cases) and may not be under the control of the programmer. Nevertheless, the issue is the same: to come close to gaining peak performance, one must optimize the use of this level of memory (i.e., retain information as long as possible before the next access to main memory), obtaining as much reuse as possible.

7.1 Matrix-Vector Operations

One approach to restructuring algorithms to exploit hierarchical memory involves expressing the algorithms in terms of matrix-vector operations. These operations have the benefit that they can reuse data and achieve a higher rate of execution than the vector counterpart. In fact, the number of floating-point operations remains the same; only the data reference pattern is changed. This change results in a operation-to-memory-reference rate on vector computers of effectively 2 vector floating-point operations and 1 vector-memory reference.

Table 3 shows the results when the LINPACK algorithm for solving dense systems of linear equations was recast in terms of a matrix-vector multiplication. (The table labeled

"Matrix-Vector Operations" in the benchmark report gives a more complete list of the performance of the various computers when that software is run and the time measured.) This second benchmark involves a matrix of order 300; the larger matrix is intended to allow high-performance computers to reflect their potential performance. In this benchmark, the algorithm has been built on matrix-vector operations for Level 2 BLAS[14, 11, 12], which are currently being implemented by various computer vendors.

Table 3
Comparison with Matrix-Vector Operations
Solving a 100 x 100 Matrix Problem

Machine	Performance Before, MFLOPS	Performance After, MFLOPS	System Efficiency
CONVEX C-1	3.0	9.6	.48
SCS-40	8.0	13	.30
FPS 264	5.6	24	.44
Alliant FX/8	7.6 (8 procs)	11	.11
CRAY-1	12	38	.24
CRAY X-MP-1	24	48	.23
IBM 3090/VF-200	12	24(1 proc)	.22
Amdahl 1100	16	48	.18
NEC SX-1E	35	71	.22
CDC CYBER 205	17	24	.06
CRAY X-MP-2	24 (1 proc)	37 (2 procs)	.088
IBM 3090/VF-400	12	24(1 proc)	.22
Amdahl 1200	18	52	.098
NEC SX-1	39	74	.11
CRAY X-MP-4	24 (1 proc)	74 (4 procs)	.088
Hitachi S-810/20	17	48	.057
NEC SX-2	46	100	.076
CRAY-2	14	28 (1 proc)	.056

The Level 2 BLAS were proposed in order to support the development of software that would be both portable and efficient across a wide range of machine architectures, with emphasis on vector-processing machines. Many of the frequently used algorithms of numerical linear algebra can be coded so that the bulk of the computation is performed by calls to Level 2 BLAS routines; efficiency can then be obtained by utilizing tailored implementations of the Level 2 BLAS routines. On vector-processing machines one of the aims of such

implementations is to keep the vector lengths as long as possible, and in most algorithms the results are computed one vector (row or column) at a time. In addition, on vector register machines performance is increased by reusing the results of a vector register, and not storing the vector back into memory.

Unfortunately, this approach to software construction is often not well suited to computers with a hierarchy of memory (such as global memory, cache or local memory, and vector registers) and true parallel-processing computers. For those architectures it is often preferable to partition the matrix or matrices into blocks and to perform the computation by matrix-matrix operations on the blocks. By organizing the computation in this fashion we provide for full reuse of data while the block is held in the cache or local memory. This approach avoids excessive movement of data to and from memory and gives a *surface-to-volume* effect for the ratio of operations to data movement. In addition, on architectures that provide for parallel processing, parallelism can be exploited in two ways: (1) operations on distinct blocks may be performed in parallel; and (2) within the operations on each block, scalar or vector operations may be performed in parallel.

7.2 Matrix-Matrix Operations

A set of Level 3 BLAS have been proposed; targeted at the matrix-matrix operations[10]. If the vectors and matrices involved are of order n, then the original BLAS include operations that are of order $O(n)$, the extended or Level 2 BLAS provide operations of order $O(n^2)$, and the current proposal provides operations of order $O(n^3)$; hence the use of the term Level 3 BLAS. Such implementations can, we believe, be portable across a wide variety of vector and parallel computers and also efficient (assuming that efficient implementations of the Level 3 BLAS are available). The question of portability has been much less studied but we hope, by having a standard set of building blocks, research into this area will be encourage.

In the case of matrix factorization, one must perform *matrix-matrix* operations rather than matrix-vector operations[18, 16]. There is a long history of block algorithms for such matrix problems. Both the NAG and the IMSL libraries, for example, include such algorithms (F01BTF and F01BXF in NAG; LEQIF and LEQOF in IMSL). Many of the early algorithms utilized a small main memory, with tape or disk as secondary storage[1, 6, 26, 19, 5, 7]. Similar techniques were later used to exploit common page-swapping algorithms in virtual-memory machines. Indeed, such techniques are applicable wherever there exists a hierarchy of data storage (in terms of access speed). Additionally, full blocks (and hence the multiplication of full matrices) might appear as a subproblem when handling large sparse systems of equations (for example, [20, 21, 7]).

More recently, several workers have demonstrated the effectiveness of block algorithms on a variety of modern computer architectures with vector-processing or parallel-processing capabilities, on which potentially high performance can easily be degraded by excessive

transfer of data between different levels of memory (vector registers, cache, local memory, main memory, or solid-state disks) [16, 2, 23, 28, 3, 4, 29, 5, 18].

Our own efforts have been twofold: First, we are attempting to recast the algorithms from linear algebra in terms of the Level 3 BLAS (matrix-matrix operations). This involves modifying the algorithm to perform more than one step of the decomposition process at a given loop iteration.

Second, to facilitate the transport of algorithms to a wide variety of architectures and to achieve high performance, we are isolating the computationally intense parts in high-level modules. When the architecture changes, we deal with the modules separately, rewriting them in terms of machine-specific operations; however, the basic algorithm remains the same. By doing so we can achieve the goal of a high operation-to-memory-reference ratio.

Figure 3 shows the results for the third LINPACK benchmark, which solves a matrix of order 1000. (Table 6 in the benchmark report presents these results in greater detail.) For this case, manufacturers may implement any algorithm they wish to solve the linear equation. The only restrictions are that the results be correct and that the operation count used to report MFLOPS be $2/3n^3 + 2n^2$ independent of the actual number of operations or method used to computer the solution. Most of the implementations here are based on a version of LU decomposition using matrix-matrix operations.

The graph shows three sets of numbers for various computers: the LINPACK benchmark number, the best achieved performance using any algorithm to solve a 1000 by 1000 system of equations, and the theoretical peak performance of the system. From the graph, it is clear that the LINPACK numbers fall far short of peak performance; in general, they are an order of magnitude less than the peak. The second set of points provides a measure of the best performance attainable for this problem on the given systems (that is, what degree of the manufacturer's quoted peak performance has been achieved).

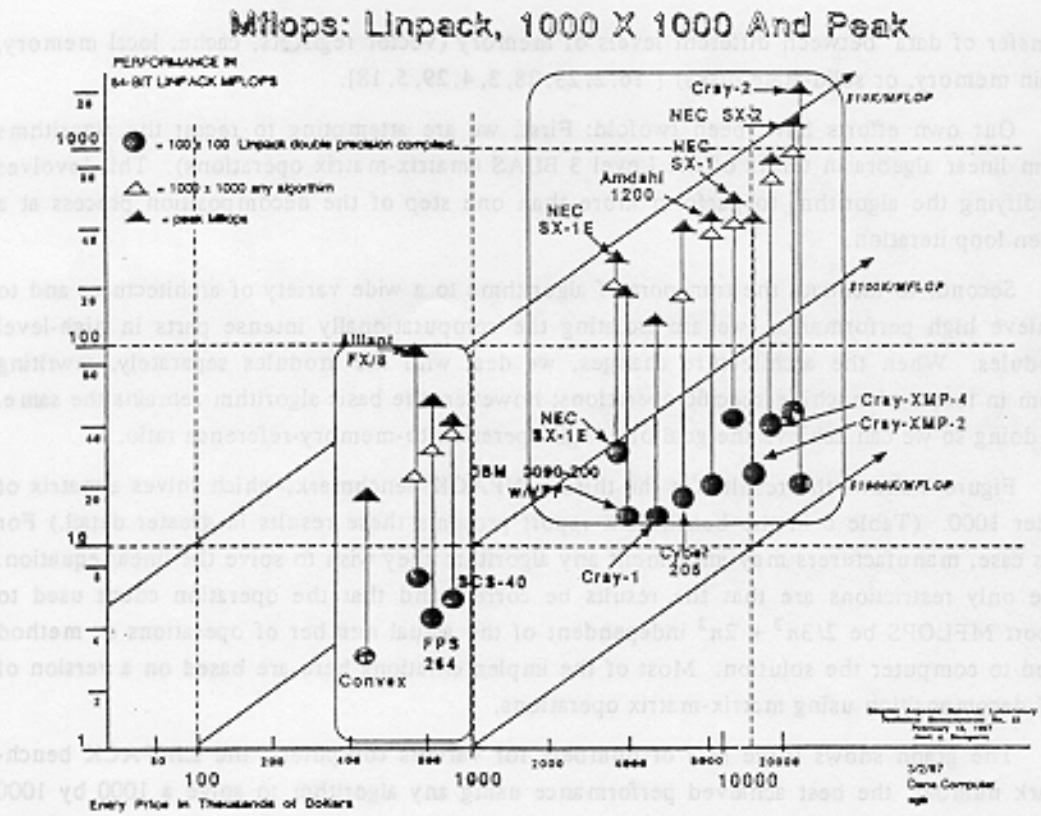

Figure 3
LINPACK / Best / Peak Performance

Concluding Remarks

Over the past several years, the LINPACK Benchmark has evolved from a simple listing for one matrix problem to an expanded benchmark describing the performance at three levels of problem size on several hundred computers. The benchmark today is used by scientists worldwide to evaluate computer performance, particularly for innovative advanced-architecture machines.

Nevertheless, a note of caution is needed. Benchmarking, whether with the LINPACK Benchmark or some other program, must not be used indiscriminately to to judge the overall performance of a computer system. Performance is a complex issue, dependent on a variety of diverse quantities including the algorithm, the problem size, and the implementation. The LINPACK Benchmark provides three separate benchmarks that can be used to evaluate computer performance on a dense system of linear equations: the first for 100 x 100 matrix, the

second for a 300 x 300 matrix, and the third for a 1000 x 1000 matrix. The third benchmark, in particular, is dependent on the algorithm chosen by the manufacturer.

References

1. D.W. Barron and H.P.F. Swinnerton-Dyer, "Solution of Simultaneous Linear Equations Using a Magnetic-Tape Store," *Computer J.*, vol. 3, pp. 28-33, 1960.

2. M. Berry, K. Gallivan, W. Harrod, W. Jalby, S. Lo, U. Meier, B. Philippe, and A. Sameh, "Parallel Algorithms on the CEDAR System," *CSRD Report No. 581*, 1986.

3. C. Bischof and C. Van Loan, "The WY Representation for Products of Householder Matrices," *SIAM SISSC*, vol. 8, 2, March, 1987.

4. I. Bucher and T. Jordan, "Linear Algebra Programs for use on a Vector Computer with a Secondary Solid State Storage Device," in *Advances in Computer Methods for Partical Differential Equations*, ed. R. Vichnevetsky and R Stepleman, pp. 546-550, IMACS, 1984.

5. D.A. Calahan, "Block-Oriented Local-Memory-Based Linear Equation Solution on the CRAY-2: Uniprocessor Algorithms," *Proceedings International Conference on Parallel Processing*, pp. 375-378, IEEE Computer Society Press, August 1986.

6. B. Chartres, "Adaption of the Jacobi and Givens Methods for a Computer with Magnetic Tape Backup Store," *University of Sydney Technical Report No. 8*, 1960.

7. A.K. Dave and I.S. Duff, "Sparse Matrix Calculations on the CRAY-2," AERE Harwell Report CSS 197 (to appear Parallel Computing), 1986.

8. J.J. Dongarra, "Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment," Argonne National Laboratory MCS-TM-23, April, 1987.

9. J.J. Dongarra, J. Bunch, C. Moler, and G. Stewart, *LINPACK Users' Guide*, SIAM Pub., Philadelphia, 1976.

10. J.J. Dongarra, J. DuCroz, I. Duff, and S. Hammarling, "A Proposal for a Set of Level 3 Basic Linear Algebra Subprograms," Argonne National Laboratory Report, ANL-MCS-TM-88, April 1987.

11. J.J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson, "An Extended Set of Fortran Basic Linear Algebra Subprograms," Argonne National Laboratory Report, ANL-MCS-TM-41 (Revision 3), November 1986.

12. J.J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson, "An Extended Set of Basic Linear Algebra Subprograms: Model Implementation and Test Programs," Argonne National Laboratory Report, ANL-MCS-TM-81, November, 1986.

13. J.J. Dongarra and I.S. Duff, "Advanced Architecture Computers," Argonne National Laboratory Report, ANL-MCS-TM-57 (Revision 1), January, 1987.

14. J.J. Dongarra and S. C. Eisenstat, "Squeezing the Most out of an Algorithm in Cray Fortran," *ACM Trans. Math. Software*, vol. 10, 3, pp. 221-230, 1984.

15. J.J. Dongarra, F. Gustavson, and A. Karp, "Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine," *SIAM Review*, vol. 26, 1, pp. 91-112, Jan. 1984.

16. J.J. Dongarra and T. Hewitt, "Implementing Dense Linear Algebra Algorithms Using Multitasking on the CRAY X-MP-4," *SIAM J. Sci Stat. Comp.*, vol. 7, 1, pp. 347-350, January, 1986.

17. J.J. Dongarra and A. Hinds, "Unrolling Loops in Fortran," *Software-Practice and Experience*, vol. 9, pp. 219-226, 1979.

18. J.J. Dongarra and D.C. Sorensen, "Linear Algebra on High-Performance Computers," in *Proceedings Parallel Computing 85*, ed. U. Schendel, pp. 3-32, North Holland, 1986.

19. J. DuCroz, S. Nugent, J. Reid, and D. Taylor, "Solving Large Full Sets of Linear Equations in a Paged Virtual Store," *TOMS* , vol. 7,4, pp. 527-536, 1981.

20. I.S. Duff, "Full Matrix Techniques in Sparse Gaussian Elimination," *Numerical Analysis Proceedings, Dundee 1981, Lecture Notes in Mathematics 912*, pp. 71-84, Springer-Verlag, Berlin, 1981.

21. A. George and H. Rashwan, "Auxiliary Storage Methods for Solving Finite Element Systems," *SIAM SISSC*, vol. 6, pp. 882-910, 1985.

22. R.W. Hockney and C.R. Jesshope, *Parallel Computers*, p. Adam Hilger Ltd, Bristol, 1981.

23. IBM, "Engineering and Scientific Subroutine Library," *IBM*, vol. Program Number: 5668-863, 1986.

24. D. Knuth, "An Empirical Study of Fortran Programs," *Software-Practice and Experience*, vol. 1, pp. 105-133, 1971.

25. C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Transactions on Mathematical Software*, vol. 5, pp. 308-323, 1979.

26. A.C. McKellar and E.G. Coffman Jr., "Organizing Matrices and Matrix Operations for Paged Memory Systems," *CACM*, vol. 12,3, pp. 153-165, 1969.

27. D. Pager, "Some Notes on Speeding Up Certain Loops by Software, Firmware, and Hardware Means," *IEEE Trans. on Comp.*, pp. 97-100, January 1972.

28. Y. Robert and P. Sguazzero, "The LU Decomposition Algorithm and Its Efficient Fortran Implementation on the IBM 3090 Vector Multiprocessor," IBM ECSEC Report ICE-0006, March 1987.

29. R. Schreiber, "Engineering and Scientific Subroutine Library, Module Design Specification," *SAXPY Computer Corporation, 255 San Geronimo Way, Sunnyvale, CA 94086*, vol. 1, 1986.