The LINPACK Benchmark: Past, Present, and Future*

Jack J. Dongarra†, Piotr Luszczek†, and Antoine Petitet‡

July 2002

Abstract

This paper describes the LINPACK Benchmark [41] and some of its variations commonly used to assess performance of computer systems. Aside from the LINPACK benchmark suite, the TOP500 [43], and the HPL [48] code are presented. The latter is frequently used to obtained results for TOP500 submissions. Information is also given on how to interpret results of the benchmark and how the results fit into performance evaluation process.
Keywords: BLAS, benchmarking, high performance computing, HPL, linear algebra, LINPACK, TOP500

1 Introduction

The original LINPACK Benchmark is, in some sense, an accident. It was originally designed to assist users of the LINPACK package [15] 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 [15] in 1979. The appendix comprised of data for one commonly used path in the LINPACK software package. Results were provided for a matrix problem of size 100, on a collection of widely used computers (23 computers in all). This was done so users could estimate the time required to solve their matrix problem by extrapolation.

Over the years additional performance data was added, more as a hobby than anything else, and today the collection includes over 1300 different computer systems. In addition to the number of computers increasing, the scope of the benchmark has also expanded. The benchmark report describes the performance for solving a general dense matrix problem $Ax = b$ in 64-bit floating-point arithmetic at three levels of problem size and optimization opportunity: 100 by 100 problem (inner loop optimization), 1000 by 1000 problem (three loop optimization - the whole program), and a scalable parallel problem. The names and rules for running the LINPACK suite of benchmarks is given in Table 1.

2 The LINPACK Package and Original LINPACK Benchmark

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

The LINPACK package was based on another package, called the Level 1 Basic Linear Algebra Subroutines (BLAS) [40]. 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.

In the LINPACK Benchmark, a matrix of size 100 was originally used because of memory limitations with the computers that were in use in 1979. Such a matrix has 10000 floating-point elements and could have been accommodated in most environments of that time. At the time it represented a *large enough* problem.

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 between -1 and 1. The random number generator used in the benchmark is not sophisticated; rather its major attribute is its compactness.

*This work was sponsored in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract DE-AC05-96OR22464, and in part by the University of Tennessee.

†University of Tennessee, Department of Computer Science, Knoxville, TN 37996-3450, U.S.A., Phone: (+865) 974-8295, Fax: (+865) 974-

Benchmark name	Matrix dimension	Optimizations allowed	Parallel Processing
LINPACK 100	100	compiler	–[a]
LINPACK 1000[b]	1000	manual	–[c]
LINPACK Parallel	1000	manual	Yes
HPLinpack[d]	arbitrary	manual	Yes

[a] Compiler parallelization possible.
[b] Also known as TPP (Toward Peak Performance) or Best Effort
[c] Multiprocessor implementations allowed.
[d] Highly-Parallel LINPACK Benchmark is also known as NxN LINPACK Benchmark or High Parallel Computing (HPC).

Table 1: Overview of nomenclature and rules for the LINPACK suite of benchmarks.

Operation type	Operation count
addition	328350
multiplication	333300
reciprocal	99
absolute value	5364
comparison	4950
comparison with zero	5247

Table 2: Double precision operations counts for LINPACK 100's `DGEFA` routine.

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 (time_n) required to solve such problems on a given machine can be approximated with the LINPACK number (time_{100}) by the following extrapolation formula:

$$\text{time}_n = \frac{\text{time}_{100} \cdot n^3}{100^3}.$$

Operation counts for the most computationally intensive routine of the benchmark are given in Table 2. The table shows, that even for a small matrix (of order 100), multiplications and additions dominate the total operation count. The extrapolation formula is also useful because, on most modern CPUs, floating-point multiplications and additions take (almost) the same number of cycles.

3 Performance Characterization and Improvement

3.1 Concepts

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.

8296, E-mail: {dongarra,luszczek}@cs.utk.edu
‡Sun Microsystems, Inc., Paris, France

From this point onwards, by performance we mean the number of millions of floating point operations per second often measured in terms of Megaflops, (Mflop/s). In the context of the LINPACK benchmark, Gigaflops (Gflop/s) are also used as the number of billions of floating point operations per second. It is customary to include both additions and multiplications in the count of Mflop/s, and the operands are assumed to be 64-bit floating-point values.

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 completed in a period of time, usually the cycle time of the machine. For example, an Intel Pentium III with a cycle time of 750 MHz has two floating point units: an adder and multiplier. During each cycle the results of either the adder or multiplier can be completed, and thus the peak performance is:

$$R_{\text{peak}} = \frac{1 \text{ operation}}{1 \text{ cycle}} \cdot 750 \text{ MHz} = 750 \text{ Mflop/s}.$$

Machine	Cycle time [MHz]	Peak Performance [Mflop/s]	LINPACK 100 Performance [Mflop/s]	System Efficiency [%]
Intel Pentium III	750	750	138	18.4
Intel Pentium 4	2,530	5,060	1190	23.5
Intel Itanium	800	3,200	600	18.5
AMD Athlon	1,200	2,400	557	23.3
Compaq Alpha	500	1,000	440	44.0
IBM RS/6000	450	1,800	503	27.9
NEC SX-5	250	8,000	856	10.7
Cray SV-1	300	1,200	549	45.7

Table 3: Theoretical peak and LINPACK 100 performance numbers of various CPUs.

Table 3 shows the peak performance for a number of high-performance computers. We treat the peak theoretical performance as a limit that is guaranteed by the manufacturer not to be exceeded by programs – a sort of *speed of light* for a given computer. The LINPACK Benchmark illustrates this point quite well. In practice, as Table 3 shows, there may be a significant difference between peak theoretical and actual performance [14]. We are not claiming that Table 3 reflects the overall performance of a given system. On the contrary, we believe that no single number ever can. It does, however, reflect the performance of a dedicated machine for solving a dense system of linear equations. Since the dense matrix problem is very regular, the performance achieved is quite high, possibly still too high for some common applications to achieve and to be characterized by. Yet, LINPACK numbers give a good correction of peak performance.

In the following sections, we focus on performance improving techniques which are relevant to the LINPACK benchmark: *loop unrolling* and *data reuse*.

3.2 Loop Unrolling

It is a frequently observed fact that the bulk of the central processor time for a program is localized in 3% or less of the source code [46]. 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, simple techniques 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 [21, 39]. When a loop is unrolled, its contents is replicated one or more times, with appropriate adjustments to array indices and loop increments. Loop unrolling enhances performance, because there is the direct reduction in loop overhead (the increment, test, and branch function). For advanced computer architectures (employing segmented or pipe-lined functional units), the greater density of non-overhead operations permits higher levels of concurrency within a particular segmented unit (eg., unrolling could allow more than one multiplication to be concurrently active on a segmented machine such as the IBM Power processor). Furthermore, unrolling often increases concurrency between independent functional units on computers so equipped or ones with fused multiple-add instructions (the IBM Power processor, which has independent multiplier and adder units, could

obtain concurrency between addition for one element and multiplication for the following element). However, on machines with vector instructions, the unrolling technique has the opposite effect. Compilers would always try to detect vector operations in the loop, but the unrolling inhibits it and the resulting vector code might become scalar and consequently the performance degrades.

3.3 Vector Operations

To observe data reuse patterns 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, vector operations are performed 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 $2/3n^3$, and the number of data references, both loads and stores, is $2/3n^3$. Thus, for every add/multiply pair we must perform a load and store of the elements, unfortunately obtaining little 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. On super-scalar computers this results in a large amount of data movement and updates. To achieve high-performance rates, this *operation-to-memory-reference rate* must be higher.

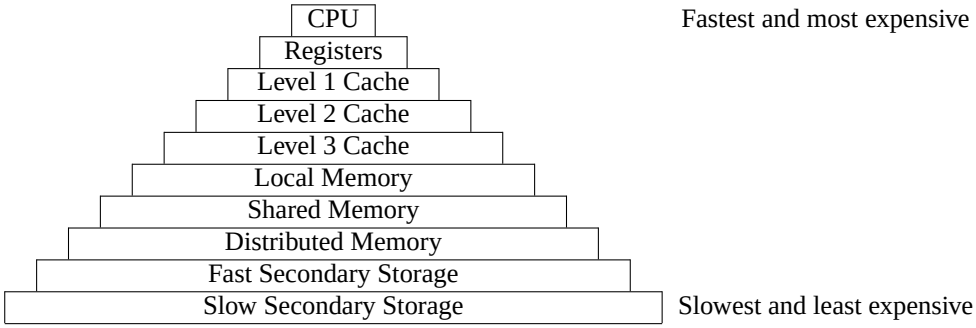

Figure 1: Computer storage hierarchy.

In some sense this is a problem with doing simple vector operations on a vector or super-scalar machine. The bottleneck is in moving data and the rate of execution is limited by this quantity. We can see this by examining the rate of data transfers and the peak performance. The Level 1 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. Today's computer architectures usually have multiple stages in the memory hierarchy as shown in Figure 1. One can gain high performance by restructuring algorithms to exploit this hierarchical organization. To come close to gaining peak performance, one must optimize the use of the lowest level of memory (i.e., retain information as long as possible before the next access to lower level of memory hierarchy), obtaining as much reuse as possible.

3.4 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. The Level 2 BLAS [17] 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 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 [18, 20, 23]. 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,
2. within the operations on each block, scalar or vector operations may be performed in parallel.

3.5 Matrix-Matrix Operations

To accommodate portability of matrix-matrix operations, a set of Level 3 BLAS have been developed; targeted at the matrix-matrix operations [16]. If the vectors and matrices involved are of order n, then the original BLAS (Level 1) include operations that are of order $O(n)$, the extended or Level 2 BLAS provide operations of order $O(n^2)$, and the latest BLAS provide operations of order $O(n^3)$ (hence the use of the term Level 3 BLAS). There is a long history of block algorithms: early algorithms utilized a small main memory, with tape or disk as secondary storage [4, 9, 10, 13, 26, 42]. More recently, several researchers have demonstrated the effectiveness of block algorithms on a variety of modern computer architectures with vector-processing or parallel-processing capabilities [5, 6, 8, 9, 20, 23, 36, 49, 50]. Additionally, full blocks (and hence the multiplication of full matrices) might appear as a subproblem when handling large sparse systems of equations [13, 19, 27, 33]. Finally, it has been shown that matrix-matrix operations can be exploited further. LU factorization (the method of choice for the LINPACK benchmark code) can be formulated recursively [35]. The recursive formulation achieves better performance [53] than a block algorithm [2]. This is due to lower memory traffic of the recursive method which is achieved through better utilization of Level 3 BLAS. The result carries on to the case of out-of-core computations [54]. Interestingly, the recursive algorithm cannot be implemented in standard Fortran 77 (due to the lack of recursive functions and subroutines) unless explicit code for handling frame stack is provided [35] or explicit calculation of update sequence is performed. In Fortran 90, on the other hand, implementation requires careful consideration of the runtime data copying process which may significantly decrease the performance. A much more elegant implementation may be easily achieved in C.

4 LINPACK Benchmark Suite Evolution

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.

As mentioned earlier, performance is a complex issue. To accommodate its evaluation, the LINPACK benchmark suite provides three separate benchmarks that can be used to evaluate computer performance on a dense system of linear equations: the first for a 100 by 100 matrix, the second for a 1000 by 1000 matrix. The third benchmark, in particular, is dependent on the algorithm chosen by the manufacturer and the amount of memory available on the computer being benchmarked. For details refer to Table 1.

In the case of LINPACK 100, the problem size was relatively small and no changes were allowed to the LINPACK software. Moreover, no explicit attempt was made to use special hardware features or to exploit vector capabilities or multiple processors. The compilers on some machines may, of course, generate optimized code that itself accesses special features. Thus, as described before, many high-performance machines may not have reached their asymptotic execution rates. However, the benchmark is still important because it quite well approximates performance rates of numerically intensive codes written by the user and optimized by an optimizing compiler.

The fact that a vendor-supplied code could achieve much higher performance rates than any compiler-optimized code is reflected in the LINPACK benchmark suite by LINPACK 1000. To begin with, the problem size is larger

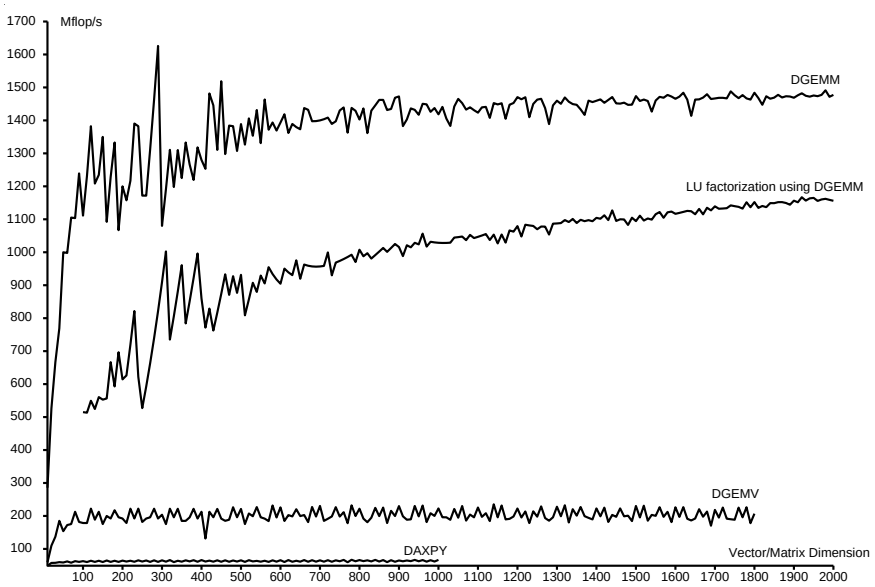

Figure 2: Asymptotic performance rate of BLAS routines `DAXPY` (Level 1), `DGEMV` (Level 2), and `DGEMM` (Level 3) and LU factorization based on BLAS Level 3 on AMD Athlon 1200 Mhz processor (BLAS routines implementation comes from ATLAS 3.2.1 [22, 25]).

(matrix of order 1000). In addition, modifying (or even replacing) the algorithm and software is permitted to achieve as high an execution rate as possible. Thus, the hardware has more opportunity for reaching so called *near-asymptotic rates*. Figure 2 illustrates the concept of asymptotic rate: BLAS routines exhibit higher performance rate as the matrix size increases and algorithm set up overhead becomes negligible. But at certain point the performance rate seizes to get larger as the system riches its optimal behavior for a given routine. Other linear algebra, including LINPACK's LU factorization shown for comparison in Figure 2, have their asymptotic rates as well.

To guard against excessive optimization encroaching upon correctness of the solution, constraints are imposed on the setup of the matrix and numerical properties of the solution. This is achieved by requiring the use of the driver code from LINPACK 100 which generates random matrix entries, calls the routines to solve the problem (these may be replaced by the user), verifies that the answer is correct, and computes the total number of operations (independently of the method) as $2n^3/3+2n^2$ (where $n = 1000$). The answer is correct if it has the same relative accuracy as standard techniques such as Gaussian elimination used in the LINPACK package. By relative accuracy we mean that the scaled residual is a slowly growing function of matrix dimension n. For completeness, we only mention that this is achieved through the following standard result which holds regardless of conditioning of A:

$$\frac{\|Ax-b\|}{\|A\|\cdot\|x\|\cdot n\cdot\varepsilon} = O(1)$$

where: $A \in \mathbf{R}^{n\times n}$; $x, b \in \mathbf{R}^n$, ε is machine precision for 64-bit floating point arithmetic ($\varepsilon = \max_{x>0} \text{float}(1+x) = 1$, $\text{float}(x)$ is machine representation of x), and $\|\cdot\|$ is any consistent matrix and vector norm.

With the arrival of parallel computers yet another requirement of LINPACK benchmark had to be reconsidered. The, so called, HPLinpack benchmark allows for matrix dimension n to be made as large as necessary so that asymptotic performance can be achieved. The following quantities are reported for each system:

- R_{max} the performance in Gflop/s for the largest problem run on a machine,
- N_{max} the size of the largest problem run on a machine,
- $N_{1/2}$ the size where half the R_{max} execution rate is achieved,
- R_{peak} the theoretical peak performance Gflop/s for the machine.

To summarize, the rules for HPLinpack are: solve systems of equations by some method allow the problem size n to vary, and measure the execution time for each size problem. In computing the floating-point execution rate, use $2n^3/3+2n^2$ operations indpendent of the actual method used (if Gaussian elimination is chosen then partial pivoting must be used), compute and report a residual for the accuracy of solution as $\|Ax-b\|/(\|A\|\cdot\|x\|)$.

5 Top500 List

Statistics on high-performance computers are of major interest to manufacturers and potential users. They wish to know not only the number of systems installed, but also the location of the various supercomputers within the high-performance computing community and the applications for which a computer system is being used. Such statistics can facilitate the establishment of collaborations, the exchange of data and software, and provide a better understanding of the high-performance computer market.

Statistical lists of supercomputers are not new. Every year since 1986, Hans Meuer [43] has published system counts of the major vector computer manufacturers, based principally on those at the Mannheim Supercomputer Seminar. Statistics based merely on the name of the manufacturer are no longer useful, however. New statistics are required that reflect the diversification of supercomputers, the enormous performance difference between low-end and high-end models, the increasing availability of massively parallel processing (MPP) systems, and the strong increase in computing power of the high-end models of workstations such as symmetric multiprocessors (SMP).

To provide this new statistical foundation, the TOP500 list was created in 1993 to assemble and maintain a list of the 500 most powerful computer systems. It first edition it was partially based partially on statistical lists published by others for different purposes [1, 38] while today it relies on submissions from computer system users and vendors. The list is compiled twice a year with the help of high-performance computer experts, computational scientists, manufacturers, and the Internet community in general.

It is true that, in the list, computers are ranked by their performance on the HPLinpack benchmark. However, in an attempt to obtain uniformity across all computers in performance reporting, the algorithm used in solving the system of linear equations in the benchmark routine must conform to the standard operation count for LU factorization with partial pivoting. In particular, the operation count for the algorithm must be $2/3n^3+O(n^2)$ floating point operations. This excludes the use of a fast matrix multiply algorithm like Strassen's [3, 34, 47, 52] or Coppersmith and Winograd's [12] methods for matrix multiplication. Even though there is no specific requirement for the method used for measuring performance, there exists a reference implementation of the benchmark called HPL [48]. While meant only as a guideline, it is being widely used to provide data for the TOP500 list since it uses external routines for matrix-matrix operations which are supplied by the vendor and are very well tuned for a given computer system. Detailed description of HPL is provided in the following section. As closing remarks, we would like to mention other software packages and technologies that can be used for TOP500 submission. They include: HPF [29, 30], PESSL [37], PLAPACK [57], ScaLAPACK [7] or even LAPACK [2] combined with either OMP [45] or `pthreads` [28].

6 HPL

This section gives a rather detailed description of the HPL code. However, to limit the size of this exposition, substantial amount of technical information given is omitted for which the reader is referred to the supplied references.

6.1 Overview

HPL is a portable implementation of the HPLinpack benchmark written in C. At the same time, it can be regarded as a software package that generates, solves, checks and times the solution process of a random dense linear system of equations on distributed-memory computers. The package uses 64-bit floating point arithmetic and portable routines for linear algebra operations and message passing. The former ones can either be BLAS or Vector Signal Image Processing Library (VSIPL) [51] while the latter ones are from MPI [31, 32]. The true advantage of HPL is the fact that it allows selection of multiple factorization algorithms. Figure 3 shows an outline of HPL's driver code, which is modeled after the original LINPACK 100 benchmark code.

```
/* Generate and partition matrix data among MPI computing nodes. */
/* ... */

MPI_Barrier(...); /* All the nodes start at the same time. */

HPL_ptimer(...);  /* Start wall-clock timer. */

HPL_pdgesv(...);  /* Solve system of equations. */

HPL_ptimer(...);  /* Stop  wall-clock timer. */

MPI_Reduce(...);  /* Obtain the maximum wall-clock time. */

/* Gather statistics about performance rate (base on the maximum wall-clock time)
 and accuracy of the solution. */
/* ... */
```

Figure 3: Main computational steps performed by HPL to obtain the HPLinpack benchmark rating.

6.2 Algorithm

HPL generates and solves a linear system of equations of order n:

$$Ax = b; \quad A \in \mathbf{R}^{n \times n}; \; x, b \in \mathbf{R}^n$$

by first computing LU factorization with row partial pivoting of the n by $n+1$ coefficient matrix $[A, b]$:

$$P_r[A, b] = [[L \cdot U], y]; \quad P_r, L, U \in \mathbf{R}^{n \times n}; \; y \in \mathbf{R}^n.$$

Since the row pivoting (represented by the permutation matrix P_r) and the lower triangular factor L are applied to b as the factorization progresses, the solution x is obtained in one step by solving the upper triangular system:

$$Ux = y.$$

The lower triangular matrix L is left unpivoted and the array of pivots is not returned.

Figure 4 shows 2-D block cyclic data distribution used by HPL. The data is distributed onto a two-dimensional grid (of dimensions P by Q) of processes according to the block-cyclic scheme to ensure good load balance as well as the scalability of the algorithm. The n by $n+1$ coefficient matrix is logically partitioned into blocks (each of dimension n_B by n_B; where n_B is referred to as *blocking factor*), that are cyclically dealt onto the P by Q process grid. This is done in both dimensions of the matrix.

P_0	P_1	P_0	P_1
P_2	P_3	P_2	P_3
P_0	P_1	P_0	P_1
P_2	P_3	P_2	P_3

Figure 4: Ownership of dense subblocks in two-dimensional block cyclic data distribution used by HPL. The number of processors is 4 (named P_0, P_1, P_2, and P_3), they are organized in 2 by 2 grid ($P = Q = 2$). The number of subblocks is 4 in both dimensions ($n/n_B = 4$).

The right-looking variant [18] has been chosen for the main loop of the LU factorization. This computation is logically partitioned with the same block size n_B that was used for the data distribution. This means that, at each iteration of the loop, a panel of n_B columns is factored, and the trailing submatrix is updated.

At a given iteration of the main loop, and because of the cartesian property of the distribution scheme, each panel factorization occurs in one column of processes. This particular part of the computation lies on the critical path of the

overall algorithm. For this operation, the user is offered a choice of three (Crout, left- and right-looking) recursive variants based on matrix-matrix multiply. The software also allows the user to choose in how many sub-panels the current panel should be divided at each recursion level. Furthermore, one can also select at run-time the recursion stopping criterion in terms of the number of columns left to factor. When this threshold is reached, the sub-panel will then be factored using one of the three (Crout, left- or right-looking) factorization algorithms based on matrix-vector operations. Finally, for each panel's column, the pivot search and the associated swap and broadcast operations of the pivot row are combined into one single communication step. A binary-exchange (leave-on-all) reduction performs these three operations at once what reduces the number of messages exchange but requires high-performance network to perform well.

Once the panel factorization has been performed, the factored panel of columns is broadcast to the other process columns. There are many possible broadcast algorithms and the software currently offers the following variants:

- Increasing ring,
- Modified increasing ring,
- Increasing two-ring,
- Modified increasing two-ring,
- Bandwidth-reducing,
- Modified bandwidth-reducing.

The modified variants relieve the next processor (the one that would participate in factorization of the panel after the current one) from the burden of sending messages (otherwise it has to receive as well as send matrix update data). The ring variants propagate the update data in a single pipeline fashion, whereas the two-ring variants propagate data in two pipelines concurrently. The bandwidth-reducing variants [11, 24, 34, 55, 56] divide a message to be sent into a number of pieces and scatter it across a single row of the grid of processors so that more messages are exchanged but the total volume of communication is independent of number of processors. This becomes particularly important when the computing nodes are much faster relative to the interconnect.

Once the current panel has been broadcast (or during the broadcast operation) the trailing submatrix has to be updated. As mentioned before, the panel factorization lies on the critical path. This means that when the kth panel has been factored and then broadcast, the next most urgent task to complete is factorization and broadcast of panel $k+1$. This technique is often referred to as a *look-ahead* (or *send-ahead*) in the literature. HPL allows to select various depths of look-ahead. By convention, a depth of zero corresponds to having no look-ahead, in which case the trailing submatrix is updated by the panel currently broadcast. Look-ahead consumes some extra memory to keep all the panels of columns currently in the look-ahead pipe. Our experimental results show that a look-ahead of depth 1 or 2 is most likely to achieve the best performance gain.

The update of the trailing submatrix by the last panel in the look-ahead pipe is performed in three phases. First, the pivots must be applied to form the current row panel of U. Second, upper triangular solve using the column panel occurs. Finally, the updated part of U needs to be broadcast to each process within a single column so that the local rank update of size n_B can take place. It has been decided to combine the swapping and broadcast of U at the cost of replicating the solve. HPL provides two algorithms for this communication operation: one is based on the binary-exchange algorithm and the second one on bandwidth-reducing techniques. The former variant is a modified leave-on-all reduction operation. The latter one has communication volume complexity that solely depends on the size of U (the number of process rows only impacts the number of messages being exchanged) and, consequently, should outperform the previous variant for large problems on large machine configurations. In addition, both of the previous variants may be combined in a single run of the code.

After the factorization has ended, the backward substitution remains to be done. HPL uses look-ahead of depth one to do this. The right hand side is forwarded in process rows in a decreasing-ring fashion, so that we solve $Q \cdot n_B$ entries at a time. At each step, this shrinking piece of the right-hand-side is updated. The process just above the one owning the current diagonal block of the matrix updates its last n_B entries of vector x, forwards it to the previous process column, and then broadcasts it in the process column in a decreasing-ring fashion. The solution is then updated and sent to the previous process column. The solution of the linear system is left replicated in every process row.

To verify the result, the input matrix and right-hand side are regenerated. The following scaled residuals are computed (ε is the relative machine precision):

$$
\begin{aligned}
r_n &= \frac{\|Ax-b\|_\infty}{\|A\|_1 \cdot n \cdot \varepsilon} \\
r_1 &= \frac{\|Ax-b\|_\infty}{\|A\|_1 \cdot \|x\|_1 \cdot \varepsilon} \\
r_\infty &= \frac{\|Ax-b\|_\infty}{\|A\|_\infty \cdot \|x\|_\infty \cdot \varepsilon}
\end{aligned}
$$

A solution is considered numerically correct when all of these quantities are of order $O(1)$.

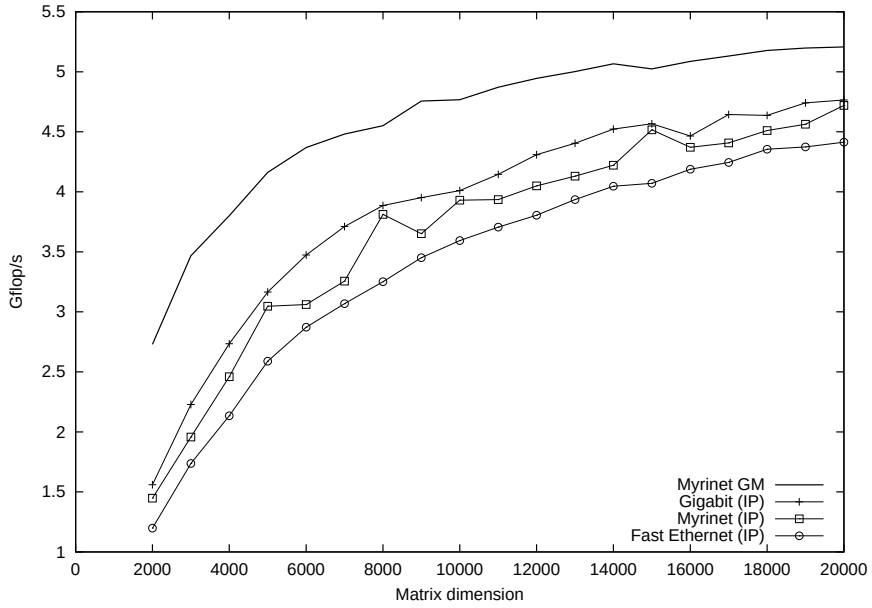

Figure 5: Comparison of HPL performance with three different interconnects: Gigabit Ethernet, Myrinet, and Fast Ethernet; on top of IP and Myrinet's native layer GM. The process grid consist of 4×4 Pentium III 550 MHz CPUs.

6.3 Performance Results

The first set of performance results is shown in Figure 5. They, just like Figure 2, reveal asymptotic behavior of performance on a cluster of Pentium III 550 MHz processors. In addition, the figure shows how differences between messaging software (Myrinet GM and TCP/IP stack) and hardware (Myrinet, Gigabit Ethernet, and Fast Ethernet) influence benchmarking results.

CPU	EV67 667 Mhz
OS	True64 ver. 5
C compiler	`cc` ver. 6.1
C flags	`-arch host -tune host -std -O5`
MPI	native (linker flags: `-lmpi -lelan`)
BLAS	CXML
Date	September 2000

Table 4: Description of the Compaq cluster used in tests.

Performance tests were also performed on a larger system – a Compaq cluster installed at Oak Ridge National Laboratory in Tennessee, U.S.A. This cluster is listed at position 90 on the June 2001 TOP500 list. Its description and performance is given in Tables 4 and 5. The LINPACK benchmark numbers for this system are presented in Table 6. All of the results, including the TOP500 submission for this system, were obtained with HPL. HPL was also used to benchmark IBM system at Oak Ridge National Lab which is listed at position 8 on the June 2002 TOP500. The system has 864 IBM Power4 processors. Performance data of this processor is given in Table 7.

Processor Grid Dimension	Matrix Dimension			
	5000	10000	25000	53000
8 by 8	26.37	45.00	60.99	67.50

Table 5: Performance (in Gflop/s) of HPL on a Compaq cluster with 64 computing nodes.

CPUs/Nodes	$N_{1/2}$	Nmax	Rmax [Gflop/s]	E [%]
1/1	150	6625	1.14	100
4/1	800	12350	4.36	95.6
16/4	2300	26500	17.0	93.2
64/16	5700	53000	67.5	92.5
256/64	14000	106000	263.6	90.1

Table 6: LINPACK benchmark numbers for the Compaq cluster obtained using HPL.

7 The Future of the Benchmark

Rather than embarking in a risky fortune-telling, we will describe recent trends in high performance computing which, we believe, will shape the near future of the LINPACK benchmark suite.

One of such trends is the ongoing improvement of CPU hardware which results in making Moore's law [44] reality. However, it is interesting to observe how the peak performance race (marketed to the public in the form of ever increasing clock rates) leaves behind much more practical metrics such as LINPACK numbers. Table 7 shows evolution of characteristics of the key hardware components and performance numbers of two widely used families of superscalar processors. The performance numbers were given as percentages of the peak to show the decreasing trend of achievable computing power. Partially, it can attributed to insufficient amount of level 1 cache which cannot provide enough buffering of the increased clock rates of the CPU and the system bus. Also, the tuning of vendor libraries lags behind the hardware development. The latter can be alleviated by self-tuning libraries such as ATLAS [22, 25] and so we are most likely to see proliferation of such "intelligent software" which encapsulates years of experience of experts and automates deployment effort of existing software to new hardware.

8 Acknowledgements

The authors acknowledge the use of the Oak Ridge National Laboratory Compaq cluster, funded by the Department of Energy's Office of Science and Energy Efficiency programs and IBM supercomputer based on Power4 microprocessor provided as a part of the Department of Energy's Scientific Discovery through Advanced Computing (SCiDAC) program.

References

[1] G. Ahrendt. Weekly postings to `comp.sys.super`, 1993.

	Intel				IBM	
Operation	Pentium III	Pentium III	P4	P4	Power3	Power4
Clock [MHz]	550	933	1700	2530	375	1300
L1 cache (I+D) [KB]	16+16	16+16	12+8	12+8	32+64	32+64
System Bus [MHz]	100	133	400	533	100	333
Peak Performance [Mflop/s]	550	933	3400	5060	1500	5200
`DGEMM` Performance [%]	73	71	64	69	86	67
LINPACK 1000 [%]	59	57	41	46	80	55
HPLinpack [%]	58	56	N/A	N/A	58	51

Table 7: Hardware and performance characteristics of two families of processors from two different vendors.

[2] Edward Anderson, Z. Bai, C. Bischof, Suzan L. Blackford, James W. Demmel, Jack J. Dongarra, J. Du Croz, A. Greenbaum, Sven Hammarling, A. McKenney, and Danny C. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, Third edition, 1999.

[3] D. Bailey, K. Lee, and H. Simon. Using strassen's algorithm to accelerate the solution of linear systems. *The Journal of Supercomputing*, 4:357–371, 1990.

[4] D. W. Barron and H. P. F. Swinnerton-Dyer. Solution of simultaneous linear equations using a magnetic tape store. *Computer J.*, 3:28–33, 1990.

[5] M. Berry, K. Gallivan, W. Harrod, W. Jalby, S. Lo, U. Meier, B. Philippe, and A. Sameh. Parallel algorithms on CEDAR system. Technical Report Report No. 581, CSRD, 1986.

[6] C. Bischof and Charles F. Van Loan. The WY representation for products of Householder matrices. *SIAM SISSC*, 8(2), March 1987.

[7] L. Suzan Blackford, J. Choi, Andy Cleary, Eduardo F. D'Azevedo, James W. Demmel, Inderjit S. Dhillon, Jack J. Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, Ken Stanley, David W. Walker, and R. Clint Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[8] I. Bucher and T. Jordan. Linear algebra programs for use on a vector computer with a secondary solid state storage device. In R. Vichnevetsky and R. Stepleman, editors, *Advances in Computer Methods for Practical Differential Equations*, pages 546–550. IMACS, 1984.

[9] D. A. Calahan. Block-oriented local-memory-based linear equation solution on the CRAY-2: Uniprocessor algorithms. In U. Schendel, editor, *Proceedings International Conference on Parallel Processing*, pages 375–378. IEEE Computer Society Press, August 1986.

[10] B. Chartres. Adaption of the Jacobi and Givens methods for a computer with magnetic tape backup store. Technical Report 8, University of Sydney, 1960.

[11] A. Chtchelkanova, J. Gunnels, G. Morrow, J. Overfelt, and R. van de Geijn. Parallel implementation of blas: General techniques for level 3 blas. *Concurrency: Practice and Experience*, 9(9):837–857, 1997.

[12] Don Coppersmith and Samuel Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Computation*, 9:251–280, 1990.

[13] A. K. Dave and Iain S. Duff. Sparse matrix calculations on the CRAY-2. Technical Report Report CSS 197, AERE Harwell, 1986.

[14] Jack J. Dongarra. Performance of various computers using standard linear equations software. Technical Report CS-89-85, University of Tennessee, 2002. (An updated version of this report can be found at `http://www.netlib.org/benchmark/performance.ps`).

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

[16] Jack J. Dongarra, J. Du Croz, Iain S. Duff, and Sven Hammarling. A set of Level 3 FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, March 1990.

[17] Jack J. Dongarra, J. Du Croz, Sven Hammarling, and R. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, March 1988.

[18] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. SIAM, Philadelphia, PA, 1998.

[19] Jack J. Dongarra, Victor Eijkhout, and Piotr Luszczek. Recursive approach in sparse matrix LU factorization. In *Proceedings of the 1st SGI Users Conference*, pages 409–418, Cracow, Poland, October 2000. ACC Cyfronet UMM. Accepted for publication in Scientific Programming.

[20] Jack J. Dongarra and T. Hewitt. Implementing dense linear algebra algorithms using mutlitasking on CRAY X-MP-4. *SIAM J. Sci Stat Comp.*, 7(1):347–350, January 1986.

[21] Jack J. Dongarra and A. Hinds. Unrolling loops in Fortran. *Software-Practice and Experience*, 9:219–226, 1979.

[22] Jack J. Dongarra, Antoine Petitet, and R. Clint Whaley. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–25, 2001.

[23] Jack J. Dongarra and Danny C. Sorensen. Linear algebra on high-performance computers. In U. Schendel, editor, *Proceedings Parallel Computing 85*, pages 3–32. North Holland, 1986.

[24] Jack J. Dongarra, R. van de Geijn, and David W. Walker. Scalability issues in the design of a library for dense linear algebra. *Journal of Parallel and Distributed Computing*, 22(3):523–537, 1994. (Also LAPACK Working Note No. 43).

[25] Jack J. Dongarra and R. Clint Whaley. Automatically tuned linear algebra software (ATLAS). In *Proceedings of SC'98 Conference*, 1998.

[26] J. DuCroz, S. Nugent, J. Reid, and D. Taylor. Solving large full sets of linear equations in a paged virtual store. *ACM Transactions on Mathematical Software*, 7(4):527–536, 1981.

[27] Iain S. Duff. Full matrix techniques in sparse Gaussian elimination. In *Numerical Analysis Proceedings*, number 912 in Lecture Notes in Mathematics, pages 71–84, Dundee, 1981. Springer-Verlag, Berlin, 1981.

[28] International Organization for Standardization. Information technology – Portable operating system interface (POSIX) – Part 1: System Application Programming Interface (API) [C language]. ISO/IEC 9945-1:1996, Geneva, Switzerland, 1996.

[29] High Performance Fortran Forum. High performance fortran language sepcivication. version 1.1. Technical report, Rice University, November 1994.

[30] High Performance Fortran Forum. High performance fortran language sepcivication. version 2.0. Technical report, Rice University, January 1997.

[31] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard (version 1.1), 1995. Available at: `http://www.mpi-forum.org/`.

[32] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997. Available at: `http://www.mpi-forum.org/`.

[33] A. George and H. Rashwan. Auxiliary storage methods for solving finite element systems. *SIAM SISSC*, 6:882–910, 1985.

[34] B. Grayson and R. van de Geijn. A high performance parallel Strassen implementation. *Parallel Processing Letters*, 6(1):3–12, 1996.

[35] Fred G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, November 1997.

[36] IBM. *Engineering and Scientific Subroutine Library*. IBM, 1986. Program Number: 5668-863.

[37] IBM. *Parallel Engineering and Scientific Subroutine Library for AIX Version 2 Release 3*. IBM, 2001.

[38] D. Kahaner. Kahaner report on supercomputer in japan. Technical report, The Computer Science Department, University of Arizona, 1992. Available at `ftp://ftp.cs.arizona.edu/japan/kahaner.reports/jsuper.92`.

[39] Donald Ervin Knuth. An empirical study of Fortran programs. *Software-Practice and Experience*, 1:105–133, 1971.

[40] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5:308–323, 1979.

[41] The LINPACK 1000x1000 benchmark program. (See `http://www.netlib.org/benchmark/1000d` for source code.).

[42] A. C. McKellar and E. G. Coffman Jr. Organizing matrices and matrix operations for paged memory systems. *Communications of the ACM*, 12(3):153–165, 1969.

[43] Hans W. Meuer, Erik Strohmaier, Jack J. Dongarra, and H.D. Simon. *Top500 Supercomputer Sites*, 17th edition edition, November 2 2001. (The report can be downloaded from `http://www.netlib.org/benchmark/top500.html`).

[44] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 19 1965.

[45] Openmp: Simple, portable, scalable smp programming. `http://www.openmp.org/`.

[46] D. Pager. Some notes on speeding up certain loops by software, firmware, and hardware means. *IEEE Trans. on Comp.*, pages 97–100, January 1972.

[47] Marcin Paprzycki and C. Cyphers. Using strassen's matrix multiplication in high performance solution of linear systems. *Computers Math. Applic.*, 31(4/5):55–61, 1996.

[48] Antoine Petitet, R. Clint Whaley, Jack J. Dongarra, and Andy Cleary. *HPL – A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers*. Innovative Computing Laboratory, September 2000. Available at `http://icl.cs.utk.edu/hpl/` and `http://www.netlib.org/benchmark/hpl/`.

[49] Y. Robert and P. Suguazerro. The LU decomposition algorithm and its efficient fortran implementation on the IBM 3090 vector multiprocessor. Technical Report ECSEC Report ICE-0006, IBM, March 1987.

[50] R. Schreiber. *Engineering and Scientific Subroutine Library. Module Design Specification*. SAXPY Computer Corporation, 255 San Geronimo Way, Sunnyvale, CA 94086, 1986.

[51] David A. Shwartz, Randall R. Judd, William J. Harrod, and Dwight P. Manley. VSIPL 1.02 API, February 26 2002. Available at: `http://www.vsipl.org/`.

[52] V. Strassen. Gaussian elimination is not optimal. *Numerical Mathematics*, 13:354–356, 1969.

[53] Sivan Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal on Matrix. Anal. Appl.*, 18(4), 1997.

[54] Sivan Toledo and Fred G. Gustavson. The design and implementation of SOLAR, a portable library of rscalable out-of-core linear algebra computations. In *Proceedings of the 4th Annual Workshop on I/O in Parallel and Distributed Systems*, pages 28–40. ACM Press, May 1996.

[55] R. van de Geijn. Massively parallel LINPACK benchmark on the Intel Touchstone DELTA and iPSC/860 systems. In *1991 Annual Users Conference Proceedings*, Dallas, Texas, 1991. Intel Supercomputer Users Group.

[56] R. van de Geijn and J. Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[57] Robert A. van de Geijn. *Using PLAPACK*. The MIT Press, 1997.