The LINPACK Benchmark: Past, Present, and Future*

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

December 2001

1 LINPACK Benchmark History

1.1 Introduction

In this paper we will clear up some of the confusion and mystery surrounding the LINPACK Benchmark [57] and some of its variations. We will cover the "original" LINPACK 100 benchmark, the TOP500 [59], the HPL [64] program that can be used in the TOP500 measurement. We will examine what is measured and describe how to interpret the results of the programs' execution. But first a bit of history.

The original LINPACK Benchmark is, in some sense, an accident. It was originally designed to assist users of the LINPACK package [25] 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 [25] in 1979. The appendix comprised 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 $Ax = b$ 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.

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

*This research was supported in part by the Applied Mathematical Sciences Research Program of the Office of Mathematical, Information, and Computational Sciences, U.S. Department of Energy under contract DE-AC05-00OR22725 with UT-Battelle, LLC and the SciDAC PERC DE-FC02-01ER25490 and NSF Alliance Subcontract 790, Prime-ACI-961-019-07.

†University of Tennessee, Department of Computer Science, Knoxville, TN 37996-3450, U.S.A., Phone: (+865)974-8295, Fax: (+865)974-8296, E-mail: {dongarra,luszczek}@cs.utk.edu

‡Sun Microsystems, Inc., Paris, France

The package itself is based on another package, called the Level 1 Basic Linear Algebra Subroutines (BLAS) [53]. The Level 1 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.

1.3 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 can be approximated by

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

In the LINPACK Benchmark, a matrix of size 100 was originally used because of memory limitations with the computer that were in use in 1979; the matrix has $O(100^2)$ floating-point elements and could be accommodated in most environments of that time. At the time it represented a "large enough" problem. (We will always assume 64-bit floating point arithmetic.)

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.)

2 The LINPACK Benchmark

2.1 Operations

The LINPACK benchmark features two routines: `DGEFA` and `DGESL` (these are the double precision versions, usually 64-bit floating point arithmetic, `SGEFA` and `SGESL` are the single-precision counter parts, usually 32 bit floating point arithmetic); `DGEFA` performs the decomposition with partial pivoting, and `DGESL` uses that decomposition to solve the given system of linear equations. Most of the execution time - $O(n^3)$ floating-point operations - is spent in `DGEFA`. Once the matrix has been decomposed, `DGESL` is used to find the solution; this requires $O(n^2)$ floating-point operations.

`DGEFA` and `DGESL` in turn call three BLAS routines: `DAXPY`, `IDAMAX`, and `DSCAL`. By far the major portion of time - over 90% at order 100 - is spent in `DAXPY`. `DAXPY` 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 `DGEFA` and $2n$ times by `DGESL` with vectors of varying length. The statement $y_i \leftarrow y_i + \alpha \cdot x_i$, which forms an element of the `DAXPY` operation, is executed approximately $n^3/3 + n^2$ times, which gives rise to roughly $2/3n^3$ floating-point operations in the solution. Thus, the $n = 100$ benchmark requires roughly $2/3$ million floating-point operations.

The statement $y_i \leftarrow y_i + \alpha \cdot x_i$, besides the floating-point addition and floating-point multiplication, involves a few one-dimensional index operations and storage references. While the LINPACK routines `DGEFA` and `DGESL` 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 `DGEFA`, the call to `DAXPY` passes an address into the two-dimensional array A, which is then treated as a one-dimensional reference within `DAXPY`. 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.

2.2 Detailed Operation Counts

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

In order to solve this matrix problem it is necessary to perform almost 700,000 floating point operations. The time required to solve the problem is divided into this number to determine the megaflops rate.

The routines `DGEFA` calls `IDAMAX`, `DSCAL` and `DAXPY`. Routine `IDAMAX`, which computes the index of a vector with largest modulus, is called 99 times, with a vector lengths running from 2 to 100. Each call to `IDAMAX` gives rise to n double precision absolute value computation and $n - 1$ double precision comparisons. The total number of operations is 5364 double precision absolute values and 4950 double precision comparisons.

`DSCAL` is called 99 times, with vector lengths running from 1 to 99. Each call to `DSCAL` performs n double precision multiplies, for a total of 4950 multiplies.

`DAXPY` does the bulk of the work. It is called n times, where n varies from 1 to 99. Each call to `DAXPY` gives rise to one double precision comparison with zero, n double precision additions, and n double precision multiplications. This leads to 4950 comparisons against 0, 328350 additions and 328350 multiplications. In addition, `DGEFA` itself does 99 double precision comparisons against 0, and 99 double precision reciprocals. The total operation count for `DGEFA` is given in Table 1.

Operation type	Operation count
add	328350
multiply	333300
reciprocal	99
absolute value	5364
$\leq$	4950
$\neq 0$	5247

Table 1: Double precision operations counts for LINPACK 100's `DGEFA` routine.

Routine `DGESL`, which is used to solve a system of equations based on the factorization from `DGEFA`, does much more modest amount of floating point operations. In `DGESL`, `DAXPY` is called in two places, once with vector lengths running from 1 to 99 and once with vector lengths running from 0 to 99. This leads to a total of 9900 double precision additions, the same number of double precision multiplications, and 199 compares against 0. `DGESL` does 100 divisions and 100 negations as well. The total operation count for `DGESL` is given in Table 2.

This leads to a total operation count for the LINPACK benchmark given in Table 3 or a grand total of 697509 floating point operations. (The LINPACK uses approximately $2/3n^3 + 2n^2$

Operation type	Operation count
add	9900
multiply	9900
divide	100
negate	100
$\neq 0$	199

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

Operation type	Operation count
add	338250
multiply	343200
reciprocal	99
divide	100
negate	100
absolute value	5364
$\leq$	4950
$\neq 0$	5446
Total	697509

Table 3: Double precision operations counts for LINPACK 100 benchmark.

operations, which for $n = 100$ has the value of 686667.)

It is instructive to look just at the contribution due to `DAXPY`. Of these floating point operations, the calls to `DAXPY` from `DGEFA` and `DGESL` account for a total of 338160 adds, 338160 multiplies, and 5147 comparisons with zero. This gives a total of 681467 operations, or over 97% of all the floating point operations that are executed.

The total time is taken up with more than arithmetic operations. In particular, there is quite a lot of time spent loading and storing the operands of the floating point operations. We can estimate the number of loads and stores by assuming that all operands must be loaded into registers, but also assuming that the compiler will do a reasonable job of promoting loop-invariant quantities out of loops, so that they need not be loaded each time within the loop. Then `DAXPY` accounts for 681468 double precision loads and 338160 double precision stores. `IDAMAX` accounts for 4950 loads, `DSCAL` for 5049 loads and 4950 stores, `DGEFA` outside of the loads and stores in the BLAS does 9990 loads and 9694 stores and `DGESL` for 492 loads and 193 stores. Thus, the total number of loads is 701949 and 352997 stores. Here again `DAXPY` dominates the statistics. The other overhead that must be accounted for is the load indexing, address arithmetic, and the overhead of argument passing and calls.

2.3 Precision

In discussions of scientific computing, one normally assumes that floating-point computations will be carried out to full precision or 64-bit floating point arithmetic. Note that this is not an issue of single or double precision as some system have 64-bit floating point arithmetic as single precision. It is a function of the arithmetic used.

2.4 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 [62]. 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 [31, 52].

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 `DAXPY` 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 `DAXPY` example, unrolling would allow more than one multiplication to be concurrently active on a segmented machine such as the IBM Power processor. 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 or ones with fused multiple-add instructions. Thus, in our `DAXPY` example, an IBM Power processor, which has 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, machines with vector instructions and their compilers trying to detect vector operations, the unrolling technique had the opposite effect. The unrolling inhibited the detection of vector operations in the loop, the resulting vector code might have become scalar, and the performance degraded.

2.5 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 (Mflop/s). We usually include both additions and multiplications in the count of Mflop/s, 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 completed in a period of time, usually the cycle time of the machine. As an example, a Pentium III with a cycle time of 750 MHz. During a cycle the results of either the adder or multiplier can be completed:

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

Table 4 shows the peak performance for a number of high-performance computers.

Machine	Cycle time [MHz]	Peak Performance [Mflop/s]
Intel Pentium III	750	750
Intel Pentium 4	1,700	1,700
Intel Itanium	800	3,200
AMD Athlon	1,200	2,400
Compaq Alpha	500	1,000
IBM RS/6000	450	1,800
NEC SX-5	250	8,000
Cray SV-1	300	1,200

Table 4: Theoretical Peak Performance of various CPUs.

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.

Machine	Peak Performance [Mflop/s]	LINPACK 100 Performance [Mflop/s]	System Efficiency [%]
Intel Pentium III	750	138	18.4
Intel Pentium 4	1,700	313	18.4
Intel Itanium	3,200	600	18.5
AMD Athlon	2,400	557	23.3
Compaq Alpha	1,000	440	44.0
IBM RS/6000	1,800	503	27.9
NEC SX-5	8,000	856	10.7
Cray SV-1	1,200	549	45.7

Table 5: LINPACK benchmark solving a 100 by 100 matrix problem.

The LINPACK Benchmark illustrates this point quite well. In practice, as Table 5 shows, there may be a significant difference between peak theoretical and actual performance [24].

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 $2/3n^3$, 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 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. 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.

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 quantities. We can see this by examining the rate of data transfers and the peak performance.

To understand why the performance is so poor, considering the basic operation performed, a `DAXPY`. There is one parameter, R_∞, that reflects the hardware performance of the idealized generic computer and gives a first-order description of any real computer. This characteristic parameter is defined as follows:

> R_∞ - the maximum or asymptotic performance - the maximum rate of computation in units of equivalent scalar operations performed per second (Mflop/s) [48].

The information on `DAXPY` and `DDOT` presented in Tables 6 and 7 was generated by running the following loops as in-line code and measuring the time to perform the operations:

```
   DAXPY

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

```
   DDOT

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

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.

Machine	DAXPY R_∞ [Mflop/s]	Peak [Mflop/s]
Intel Pentium III	56	750
Intel Pentium 4	178	1,700
Intel Itanium	28	3,200
AMD Athlon	66	2,400
Compaq Alpha	92	1,000
IBM RS/6000	100	1,800

Table 6: DAXPY's asymptotic and peak performance.

Machine	DDOT R_∞ [Mflop/s]	Peak [Mflop/s]
Intel Pentium III	82	750
Intel Pentium 4	277.5	1,700
Intel Itanium	33	3,200
AMD Athlon	80	2,400
Compaq Alpha	150	1,000
IBM RS/6000	143	1,800

Table 7: DDOT's asymptotic and peak performance

2.6 Restructuring Algorithms

Today's computer architectures usually have multiple stages in the memory hierarchy. By restructuring algorithms to exploit this hierarchical organization, one can gain high performance.

A hierarchical memory structure involves a sequence of computer memories, or caches, 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.

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 main memory), obtaining as much reuse as possible.

2.7 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 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.

2.8 Matrix-Matrix Operations

A set of Level 3 BLAS have been proposed; targeted at the matrix-matrix operations [26]. 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 encouraged.

In the case of matrix factorization, one must perform *matrix-matrix* operations rather than matrix-vector operations [28, 30, 33]. 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 [7, 17, 18, 23, 36, 58]. 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 [23, 29, 37, 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, 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) [8, 9, 16, 17, 30, 33, 50, 65, 67].

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.

Recently it has been shown that matrix-matrix operations can be exploited further. LU factorization, which is usually the method of choice for the LINPACK benchmark code, can be formulated recursively [45]. The recursive formulation performs better [70] than block algorithm [4]. It is due to a lower memory traffic of the recursive method which is achieved through better utilization of Level 3 BLAS.

2.9 Concluding Remarks on the LINPACK Benchmark

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 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 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.

3 The Parallel LINPACK Benchmark

In the past several years, the emergence of Distributed Memory (DM) computers [48] and their potential for the numerical solution of Grand Challenge problems [22, 51, 60, 61] has led to extensive research in benchmarking. Examples of DM computers include the IBM Scalable POWERparallel SP-2, the Intel Paragon, the Cray T3E, Networks and Clusters of Workstations (NoWs and CoWs). The key feature they have achieved is *scalable* performance. These scalable parallel computers comprise an ensemble of Processing Units (PUs) where each unit consists of a processor, local memories organized in a hierarchical manner, and other supporting devices. These PUs are interconnected by a point-to-point (direct) or switch-based (indirect) network. Without modifying the basic machine architecture, these distributed memory systems are capable of proportional increases in performance as the number of PUs, their memory capacity and bandwidth, and the network and I/O bandwidth are increased. As of today, DM computers are still being produced and their success is apparent when considering how common they have become. Still, their limitations have been revealed and their successors have already appeared. The latter are constructed from a small number of nodes, where each node is a small DM computer featuring a virtual shared memory. These nodes are interconnected by a simple bus- or crossbar-based interconnection network. Programming these machines as well as their production is facilitated by the relative simplicity of the interconnection network. In addition, increasing the computational capabilities of the PUs appears to be an easier task than increasing the performance of the network. As opposed to large scale DM computers where all processors are much less powerful than the whole, the collection of nodes of this hierarchical architecture is only slightly more powerful than its components. The

SGI SMP Power Challenge is an existing example of such an architecture. The scalability of these machines can simultaneously take advantage of the progresses made by the processor and network technologies as well as the hardware and/or software mechanisms implementing the virtual shared memory. It is still unclear how these machines will be programmed. Whether these machines will in the future completely replace DM computers is also a question that is difficult to answer today. In this paper, these machines will also be considered as DM computers.

In order to fully exploit the increasing computational power of DM computers, the application software must be *scalable*, that is, able to take advantage of larger machine configurations to solve larger problems with the same efficiency. The LINPACK benchmark addresses scalability by an introduction of a new category of testing rules and environment. This new category is referred to as a Highly-Parallel LINPACK (HPL) NxN benchmark [10, 13, 14, 15, 24, 71, 44, 74]. It requires solution of systems of linear equations by some method. The problem size is allowed to vary, and the best floating-point execution rate should be reported. In computing the execution rate, the number of operations should be $2n^3/3 + 2n^2$ independent of the actual method used. If Gaussian elimination is chosen, partial pivoting must be used. A residual for the accuracy of solution, given as $\|Ax - b\|/(\|A\| \cdot \|x\|)$, should also be reported. The following quantities of the benchmark are reported in the TOP500 list:

- 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 the performance of the HPL NxN benchmark contributes efficiency of the serial code executed on a single CPU as well as the parallel algorithm which makes all the CPUs cooperate. The former may be designed with the use of the optimization techniques mentioned earlier. The essential ingredients of the latter are load balancing and network latency hiding. Assuming that all of the CPUs are the same, an even work and data load may is usually achieved with a block-cyclic data layout [34, 47]. Latency hiding is most often performed with computation and communication overlapping which is facilitated by an appropriate choice of virtual CPU topology, computational sequences of LU factorization, and an implementation of collective communication algorithms. Large body of literature exists on efficient parallel implementation of BLAS [1, 2, 3, 20, 21, 38, 39, 40, 43, 49, 56, 72, 63], the factorization algorithm [5, 6, 11, 19, 34, 41, 47, 66, 68, 69, 73], and its components [12, 46, 54, 55].

4 The 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 [59] 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 workstation suppliers (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. The list is compiled twice a year with the help of high-performance computer experts, computational scientists, manufacturers, and the Internet community in general.

In the list, computers are ranked by their performance on the HPL NxN benchmark.

5 The HPL Code

5.1 Overview

HPL [64] is a software package that generates and solves a random dense linear system of equations on distributed-memory computers as it is outlined in Fig. 1. The package uses 64-bit floating point arithmetic and portable routines for linear algebra operations and message passing. Also, it gives a possibility of selecting one of multiple factorization algorithms and provides timing and an estimate of accuracy of the solution. Consequently, it can be regarded as a portable implementation of the HPL NxN benchmark. It requires implementations of MPI and either BLAS or Vector Signal Image Processing Library (VSIPL).

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

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

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

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

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

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

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

Figure 1: **Main computational steps performed by HPL to obtain the HPL NxN benchmark rating.**

The HPL package is written in C and requires implementation of either the BLAS [26, 27] or

the Vector Signal Image Processing Library (VSIPL).

5.2 Algorithm

HPL 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:

$$P[A, b] = [[L, U], y].$$

Since the row pivoting (represented by the permutation matrix P) 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.

Fig. 2 shows 2-D cyclic block 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), that are cyclically dealt onto the P by Q process grid. This is done in both dimensions of the matrix.

P0	P1	P0	P1
P2	P3	P2	P3
P0	P1	P0	P1
P2	P3	P2	P3

Figure 2: **Ownership of dense subblocks in two-dimensional block cyclic data distribution used by HPL. The number of processors is 4 (named P0, P1, P2, and P3), 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 has been chosen for the main loop of the LU factorization. This means that, at each iteration of the loop, a panel of N_B columns is factored, and the trailing submatrix is updated. Therefore, this computation is logically partitioned with the same block size N_B that was used for the data distribution.

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.

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 divide a message to be sent into a number of pieces and scatters 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 relatively much faster than 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 the factorization and broadcast of the 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. 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 two 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. The following algorithms are available for this communication operation:

- binary-exchange,
- bandwidth-reducing.

The first variant is a modified leave-on-all reduction operation. The second 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):

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

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

5.3 Scalability

HPL was designed for distributed-memory computers. They consist of processors that are connected using a message passing interconnection network. Each processor has its own memory called the local memory, which is accessible only to that processor. As the time to access a remote memory is longer than the time to access a local one, such computers are often referred to as Non-Uniform Memory Access (NUMA) machines. The interconnection network of our machine model is static, meaning that it consists of point-to-point communication links among processors. This type of network is also referred to as a direct network as opposed to dynamic networks. The latter are constructed from switches and communication links. These links are dynamically connected to one another by the switching elements to establish, at run-time, the paths between processors' memories. The interconnection network of the two-dimensional machine model considered here is a static, fully connected physical topology. It is also assumed that processors can be treated equally in terms of local performance and that the communication rate between two processors depends only on the processors being considered. Our model assumes that a processor can send or receive data on only one of its communication ports at a time (assuming it has more than one). In the literature, this assumption is also referred to as a one-port communication model. The time spent to communicate a message between two given processors is called the communication time T_c. In our machine model, T_c is approximated by a linear function of the number L of double precision floating point numbers being communicated. T_c is the sum of the time to prepare the message for transmission α and the time βL taken by the message of length L to traverse the network to its destination:

$$T_c = \alpha + \beta L.$$

Finally, the model assumes that the communication links are bi-directional, i.e., the time for two processors to send each other a message of length L is also T_c. A processor can send and/or receive a message on only one of its communication links at a time. In particular, a processor can send a message while receiving another message from the processor it is sending to at the same time.

Since this document is only concerned with regular local dense linear algebra operations, the time taken to perform one floating point operation is assumed to be modeled by three constants: γ_1, γ_2, and γ_3. These quantities are single processor approximations of floating point operation (FLOP) rates of the vector-vector, matrix-vector and matrix-matrix operations, respectively. This

approximation describes all the steps performed by a processor to complete its portion of matrix factorization. Such a model neglects all the phenomena occurring in the processor components, such as cache misses, pipeline start-ups, memory load or store, floating point arithmetic, TLB misses etc. They may influence the value of the constants and may vary as, e.g. a function of the problem size.

Similarly, the model does not make any assumptions on the amount of physical memory available on a single node. It is assumed that if a process has been spawn on a processor, one has ensured that enough memory was available on that processor. In other words, swapping will not occur during the modeled computation.

Consider an M by N panel distributed over a column of P processors. Because of the recursive formulation of the panel factorization, it is reasonable to consider that the floating point operations will be performed at matrix-matrix multiply rate. For every column in the panel a binary-exchange is performed on $2N$ data items. When this panel is broadcast, what matters is the time that the next process column will spend in this communication operation. Assuming one chooses the modified increasing ring variant, only one message needs to be taken into account. The execution time of the panel factorization and broadcast can thus be approximated by:

$$T_{pfact}(M,N) = \gamma_3 \left(\frac{M}{P} - \frac{N}{3} \right) N^2 + N \log P (\alpha + 2\beta N) + \alpha + \frac{\beta MN}{P}.$$

The update phase of an N by N trailing submatrix distributed on a P by Q process grid is achieved with a triangular solve of N right hand sides and a local update of rank N_B of the trailing submatrix. Assuming that the long variant is chosen, the execution time of the update operation can be approximated by:

$$T_{update}(N,N_B) = \gamma_3 \left(\frac{N \cdot N_B^2}{Q} + \frac{2N^2 \cdot N_B}{PQ} \right) + \alpha(\log P + P - 1) + \frac{3\beta N \cdot N_B}{Q}.$$

The constant 3 in front of the β term is obtained by counting one for the logarithmic spread phase and two for the rolling phase. In the case of bi-directional links this constant should be replaced by 2.

The number of floating point operations performed during the backward substitution is given by $N^2/(P \cdot Q)$. Because of the look-ahead, the communication cost can be approximated at each step by two messages of length N_B, i.e. the time to communicate the piece of size N_B of the solution vector from one diagonal block of the matrix to another. It follows that the execution time of the backward substitution can be approximated by:

$$T_{backs}(N,N_B) = \frac{\gamma_2 N^2}{PQ} + N \left(\frac{\alpha}{N_B} + 2\beta \right).$$

The total execution time of the algorithm described above is given by:

$$\sum_{k=0}^{N} [T_{pfact}(N - k \cdot N_B, N_B) + T_{update}(N - (k-1) \cdot N_B, N_B)] + T_{backs}(N,N_B).$$

By considering only the dominant term in α, β and γ_3:

$$T_{HPL} = \gamma_3 \frac{2N^3}{3PQ} + \beta \frac{N^2(3P+Q)}{2PQ} + \alpha \frac{N((N_B+1)\log P + P)}{N_B}.$$

The serial execution time is given by $T_{ser} = \frac{2}{3}\gamma_3 N^3$. If we define the parallel efficiency E as the ratio $T_{ser}/(PQ \cdot T_{HPL})$, we obtain:

$$E = \left(1 + \beta\frac{3(3P+Q)}{4\gamma_3 N} + \alpha\frac{3PQ((N_B+1)\log P + P)}{2N^2 \cdot N_B\gamma_3}\right)^{-1}.$$

This last equality shows that when the memory usage per processor $N^2/(PQ)$ is maintained constant, the parallel efficiency slowly decreases only because of the α term. The communication volume (the β term), however, remains constant. Due to these results, HPL is said to be scalable not only with respect to the amount of computation, but also with respect to the communication volume.

5.4 Performance Results

Tables 8 and 9 show system parameters and performance results of HPL for a cluster of workstations based on an AMD Athlon processor. Similarly, Tables 10 and 11 describe a cluster based on the Intel Pentium III processor. Performance tests were also performed on 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 12 and 13. The LINPACK benchmark numbers for this system are presented in Table 14.

CPU	AMD Athlon K7 500 Mhz
Main memory	256 MB
Interconnect	100 Mbps Switched 2 NICs per node (channel bonding)
OS	RedHat Linux 6.2 (kernel 2.2.14)
C compiler	`gcc` ver. 2.91.66 (egcs-1.1.2 release)
C flags	`-fomit-frame-pointer -O3 -funroll-loops`
MPI	MPICH 1.2.1
BLAS	ATLAS 3.0 beta
Date	September 2000

Table 8: Description of the AMD cluster used in tests.

Processor Grid Dimension	Matrix Dimension			
	2000	5000	8000	10000
1 by 4	1.28	1.73	1.89	1.95
2 by 2	1.17	1.68	1.88	1.93
4 by 1	0.81	1.43	1.70	1.80

Table 9: Performance (in Gflop/s) of HPL on an AMD cluster with 4 computing nodes.

5.5 Running and Tuning

In order to find the best performance of a given system, the largest problem size fitting in memory should be used. The amount of memory used by HPL is essentially the size of the coefficient matrix.

CPU	Intel Pentium III 550 Mhz
Main memory	512 MB
Interconnect	Myrinet
OS	RedHat Linux 6.1 (kernel 2.2.15)
C compiler	`gcc` ver. 2.91.66 (egcs-1.1.2 release)
C flags	`-fomit-frame-pointer -O3 -funroll-loops`
MPI	MPI GM ver. 1.2.3
BLAS	ATLAS ver. 3.0 beta
Date	September 2000

Table 10: Description of the Pentium cluster used in tests.

Processor Grid	Matrix Dimension					
Dimension	2000	5000	8000	10000	15000	20000
2 by 4	1.76	2.32	2.51	2.58	2.72	2.73
4 by 4	2.27	3.94	4.46	4.68	5.00	5.16

Table 11: Performance (in Gflop/s) of HPL on a Pentium cluster with up to 16 computing nodes.

HPL uses the block size N_B for the data distribution as well as for the computational granularity. From a data distribution point of view, the smallest N_B, the better the load balance, so, consequently, one definitely should avoid very large values of N_B. From a local computation point of view, too small value of N_B may limit the computational performance by a large factor because almost no data reuse will occur in the fastest level of the memory hierarchy. The number of messages will also increase. Efficient matrix-multiply routines are often internally blocked. Small multiples of this blocking factor are likely to be good block sizes for HPL.

6 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. The use of UltraSparc III server at the Department of Mathematics and Computer Science at Emory University is also greatly

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 12: Description of the Compaq cluster used in tests.

Processor Grid Dimension	Matrix Dimension			
	5000	10000	25000	53000
8 by 8	26.37	45.00.73	60.99	66.93

Table 13: 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	-
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 14: LINPACK benchmark numbers for the Compaq cluster obtained using HPL.

appreciated.

References

[1] M. Aboelaze, N. Chrisochoides, and E. Houstis. The parallelization of Level 2 and 3 BLAS operations on distributed-memory machines. Technical Report CSD-TR-91-007, Purdue University, West Lafayette, IN, 1991.

[2] R. Agarwal, S. Balle, F. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal or Research and Development*, 39(5):575–582, 1995.

[3] R. Agarwal, F. Gustavson, and M. Zubair. A high performance matrix multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. *IBM Journal or Research and Development*, 38(6):673–681, 1994.

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

[5] Cleve Ashcraft. The distributed solution of linear systems using the torus-wrap data mapping. Technical Report ECA-TR-147, Boeing Computer Services, Seattle, WA, 1990.

[6] Cleve Ashcraft. A taxonomy of distributed dense LU factorization methods. Technical Report ECA-TR-161, Boeing Computer Services, Seattle, WA, 1991.

[7] 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.

[8] 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.

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

[10] R. Bisseling and L. Loyens. Towards peak parallel LINPACK performance on 400. *Supercomputer*, 45:20–27, 1991.

[11] R. Bisseling and J. van der Vorst. Parallel LU decomposition on a transputer network. In Eds. G. van Zee and J. van der Vorst, editors, *Lecture Notes in Computer Sciences*, volume 384, pages 61–77. Springer-Verlag, 1989.

[12] R. Bisseling and J. van der Vorst. Parallel triangular system solving on a mesh network of transputers. *SIAM Journal on Scientific and Statistical Computing*, 12:787–799, 1991.

[13] J. Bolen, A. Davis, B. Dazey, S. Gupta, G. Henry, D. Robboy, G. Schiffler, D. Scott, M. Stallcup, A. Taraghi, S. Wheat (from Intel SSD), L. Fisk, G. Istrail, C. Jong, R. Riesen, and L. Shuler (from Sandia National Laboratories). Massively parallel distributed computing: World's first 281 Gigaflop supercomputer. In *Proceedings of the Intel Supercomputer Users Group*, 1995.

[14] R. Brent. The LINPACK benchmark on the AP 1000. *Frontiers*, pages 128–135, 1992.

[15] R. Brent and P. Strazdins. Implementation of BLAS Level 3 and LINPACK benchmark on the AP1000. *Fujitsu Scientific and Technical Journal*, 5(1):61–70, 1993.

[16] 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.

[17] 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.

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

[19] J. Choi, Jack J. Dongarra, Susan Ostrouchov, Antione Petitet, David Walker, and R. Clint Whaley. The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming*, 5:173–184, 1996.

[20] J. Choi, Jack J. Dongarra, and David Walker. Pumma: Parallel universal matrix multiplication algorithms on distributed-memory concurrent computers. *Concurrency: Practice and Experience*, 6(7):543–570, 1994.

[21] 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.

[22] Mathematical Committee on Physical and Engineering Sciences, editors. *Grand Challenges: High Performance Computing and Communications.* NSF/CISE, 1800 G Street NW, Washington, DC, 20550, 1991.

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

[24] Jack J. Dongarra. Performance of various computers using standard linear equations software. Technical Report CS-89-85, University of Tennessee, 1989. (An updated version of this report can be found at `http://www.netlib.org/benchmark/performance.ps`).

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

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

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

[28] 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.

[29] 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.

[30] 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.

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

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

[33] 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.

[34] Jack J. Dongarra, R. van de Geijn, and David 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).

[35] Jack J. Dongarra and Clint R. Whaley. Automatically tuned linear algebra software (ATLAS). In *Proceedings of SC'89 Conference*, 1989.

[36] 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.

[37] 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.

[38] A. Elster. Basic matrix subprograms for distributed-memory systems. In David Walker and Q. Stout, editors, *Proceedings of the Fifth Distributed-Memory Computing Conference*, pages 311–316. IEEE Press, 1990.

[39] R. Falgout, A. Skjellum, S. Smith, and C. Still. The multicomputer toolbox approach to concurrent BLAS and LACS. In *Proceedings of the Scalable High Performance Computing Conference SHPCC-92*. IEEE Computer Society Press, 1992.

[40] G. Fox, S. Otto, and A. Hey. Matrix algorithms on a Hypercube I: Matrix multiplication. *Parallel Computing*, 3:17–31, 1987.

[41] G. Geist and C. Romine. Lu factorization algorithms on distributed-memory multiprocessor architectures. *SIAM Journal on Scientific and Statistical Computing*, 9:639–649, 1988.

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

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

[44] B. Greer and G. Henry. High performance software on Intel Pentium Pro processors or micro-ops to TeraFLOPS. In *Proceedings of the SuperComputing 1997 Conference*, San Jose, California, 1997. ACM SIGARCH, IEEE Computer Society Press. ISBN: 0-89791-985-8.

[45] 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.

[46] M. Heath and C. Romine. Parallel solution triangular systems on distributed-memory multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, 9:558–588, 1988.

[47] B. Hendrickson and D. Womble. The torus-wrap mapping for dense matrix calculations on massively parallel computers. *SIAM Journal on Scientific and Statistical Computing*, 15:1201–1226, 1994.

[48] R. W. Hockney and C. R. Jesshope. *Parallel Computers*. Adam Hilger Ltd, Bristol, 1981.

[49] S. Huss-Lederman, E. Jacobson, A. Tsao, and G. Zhang. Matrix multiplication on the Intel Touchstone DELTA. *Concurrency: Practice and Experience*, 6(7):571–594, 1994.

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

[51] W. Kaufmann and L. Smarr. *Supercomputing and the Transformation of Science*. Scientific American Library, 1993.

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

[53] 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.

[54] G. Li and T. Coleman. A parallel triangular solver for a distributed-memory multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 9(3):485–502, 1988.

[55] G. Li and T. Coleman. A new method for solving triangular systems on distributed-memory message-passing multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 10(2):382–396, 1989.

[56] J. Li, R. Falgout, and A. Skjellum. A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies. *Concurrency: Practice and Experience*, 9(5):345–389, 1997.

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

[58] 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.

[59] 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`).

[60] Office of Science and Technology Policy, editors. *A Research and Development Strategy for High Performance Computing.* Executive Office of the President, 1987.

[61] Office of Science and Technology Policy, editors. *The Federal High Performance Computing Program.* Executive Office of the President, 1989.

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

[63] Antoine Petitet. *Algorithmic Redistribution Methods for Block Cyclic Decompositions.* Computer Science Department, University of Tennessee Knoxville, 1996. Ph.D. thesis.

[64] 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/hpl/`.

[65] 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.

[66] Youcef Saad. Communication complexity of the Gaussian elimination algorithm on multiprocessors. *Linear Algebra and Its Applications*, 77:315–340, 1986.

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

[68] P. Strazdins. Matrix factorization using distributed panels on the Fujitsu AP1000. In *Proceedings of the IEEE First International Conference on Algorithms And Architectures for Parallel Processing ICA3PP-95*, Brisbane, 1995.

[69] P. Strazdins. Lookahead and algorithmic blocking techniques compared for parallel matrix factorization. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems, IASTED*, Las Vegas, 1998.

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

[71] 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.

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

[73] E. van de Velde. Experiments with multicomputer LU-decomposition. *Concurrency: Practice and Experience*, 2:1–26, 1990.

[74] D. Womble, D. Greenberg, D. Wheat, and S. Riesen. LU factorization and the LINPACK benchmark on the Intel Paragon. Technical report, Sandia National Laboratories, 1994.

Appendix

Description

In the following, a number of performance results is presented for variety of CPUs. The results show:

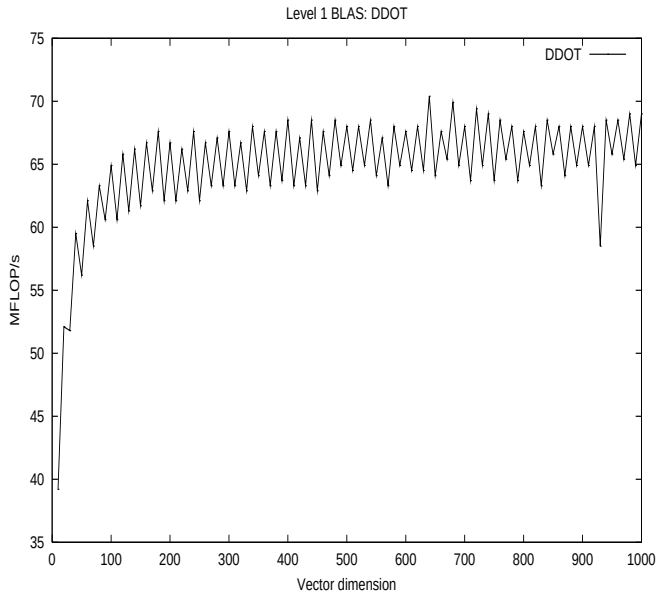
- asymptotic performance graphs of BLAS routines relevant to the HPL NxN benchmark,
- single-CPU LINPACK benchmark numbers for 1000 by 1000 problem,
- selected hardware parameters of the systems used in tests.

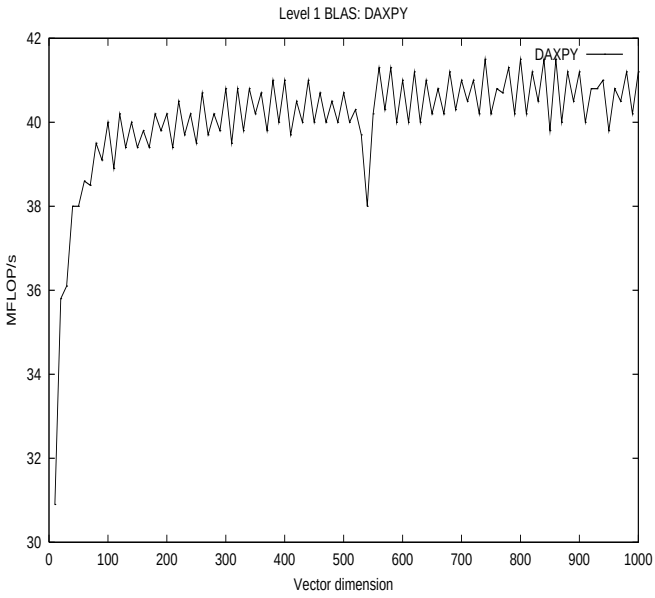
The original LINPACK benchmark used Level 1 BLAS routines: `DDOT` and `DAXPY`. As the following graphs show, their performance is severely limited on contemporary superscalar processors. The Level 2 BLAS `DGEMV` routine may be used in an implementation of a LINPACK benchmark code. As the results indicate, however, this should also be avoided as its performance is limited by the system bus throughput. The Level 3 BLAS `DGEMM` routine achieves the highest fraction of peak performance and, consequently, should be the preferred BLAS to use for the LINPACK benchmark code.

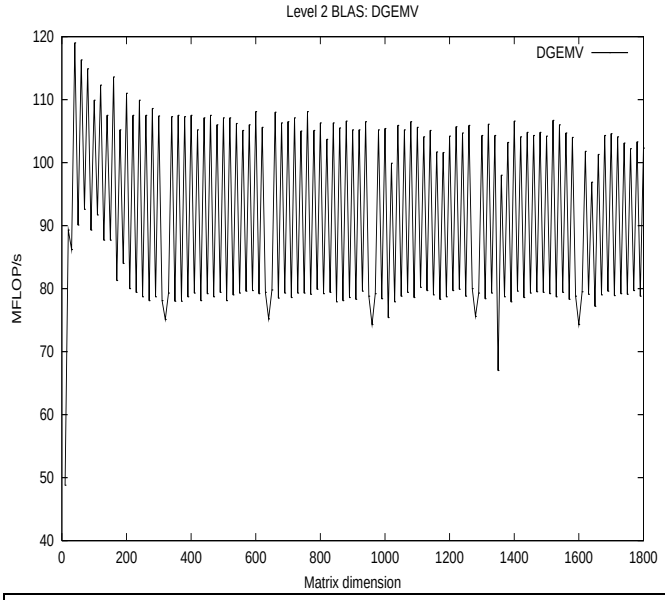
The LINPACK benchmark numbers for each of the CPUs were obtained by solving a system of 1000 simultaneous linear equations so they may be easily compared to each other. The label "Linpack source" refers to the self contained LINPACK benchmark code (available at `http://www.netlib.org/benchmark/1000d`) which relies only on compiler optimizations to increase the rate of execution. The code labeled "Linpack BLAS" requires external (possibly optimized) BLAS. The codes "Right-looking", "Left-looking", and "Crout" represent variants of LU factorization based on Level 2 BLAS routines. "Right-looking" relies on `DGER` and is equivalent to LAPACK's `DGETF2` routine [4]. "Left-looking" variant uses `DTRSV` and `DGEMV` routines. And finally, "Crout" variant uses `DGEMV` routine only. Two codes that use Level 3 BLAS are "Lapack" (which solves the problem by using LAPACK's `DGETRF` and `DGETRS` routines [4]) and "Recursive LU" which calls `DTRSM` and `DGEMM` routines and is based on recursive formulation of LU factorization.

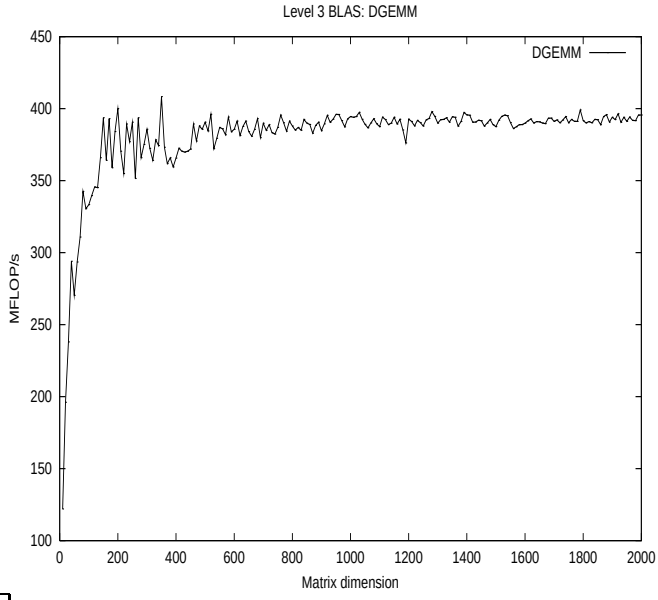
Unless stated otherwise, all codes use BLAS provided by ATLAS [32, 35] version 3.2.1 or higher (available at `http://www.netlib.org/atlas/` and `http://math-atlas.sourceforge.net/`).

Performance Results for Pentium III 550MHz





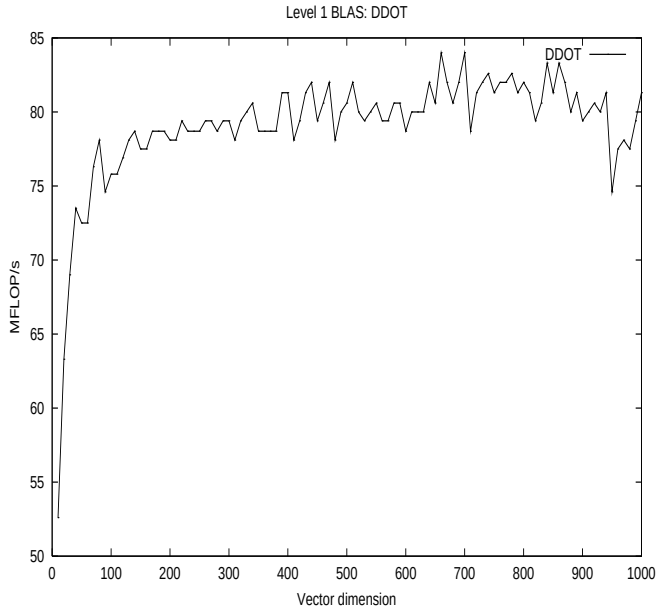


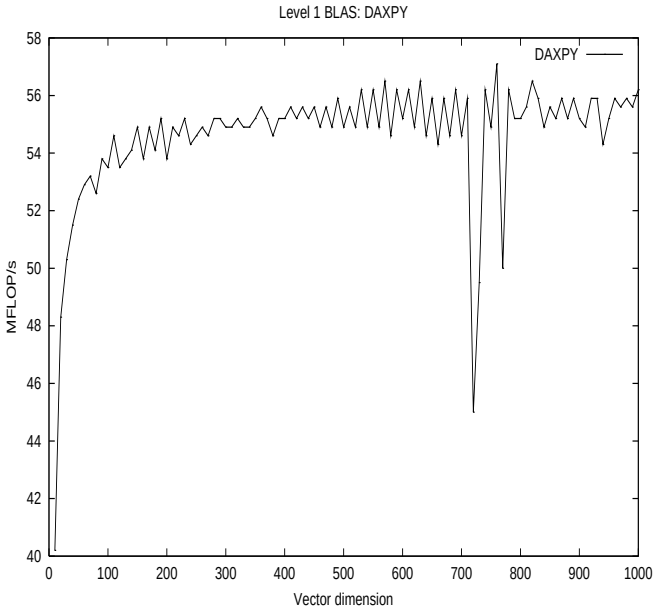


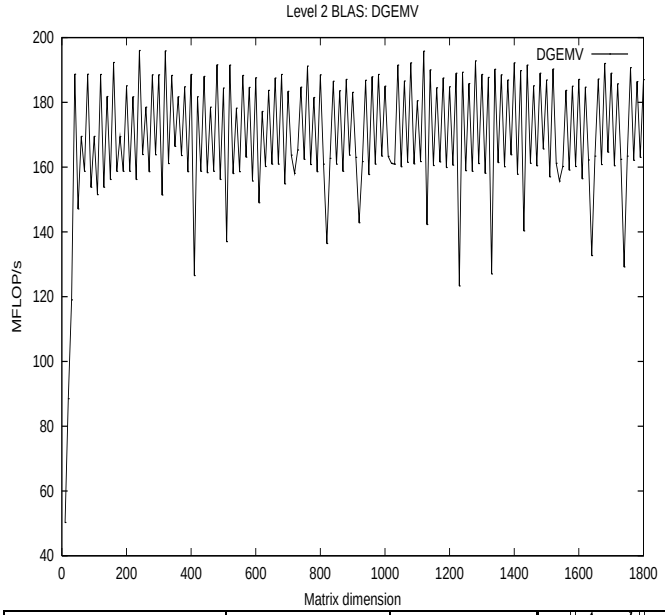
Linpack benchmark numbers			
Code	MFLOP/s	$\|Ax-b\|$	$\frac{\|Ax-b\|}{\|A\|\cdot\|x\|n\epsilon}$
Level 1 BLAS			
Linpack source	41.4	10^{-12}	10.5
Linpack BLAS	55.4	10^{-12}	10.5
Level 2 BLAS			
Right-looking	51.4	10^{-13}	2.3
Left-looking	74.7	10^{-13}	1.4
Crout	84.0	10^{-13}	1.4
Level 3 BLAS			
Lapack	278.6	10^{-13}	1.3
Recursive LU	324.6	10^{-13}	1.3

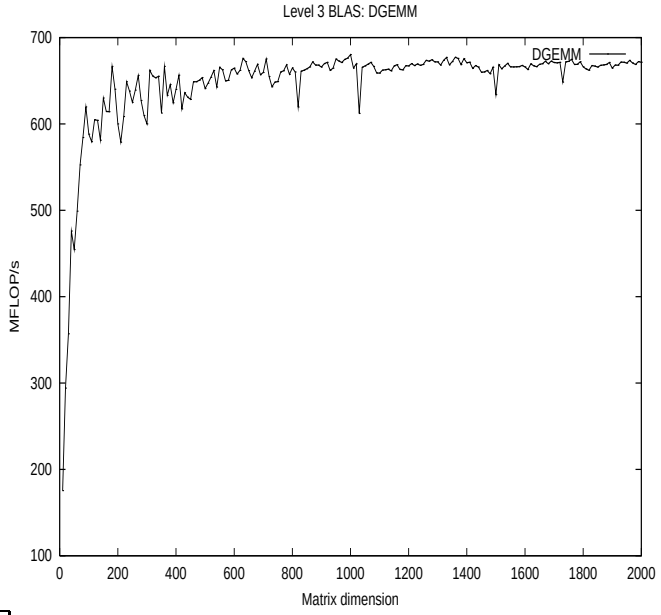
System parameters	
Clock rate [MHz]	550
Bus speed [MHz]	100
L1 cache [KB]	16+16
L2 cache [KB]	512

Performance Results for Pentium III 933MHz





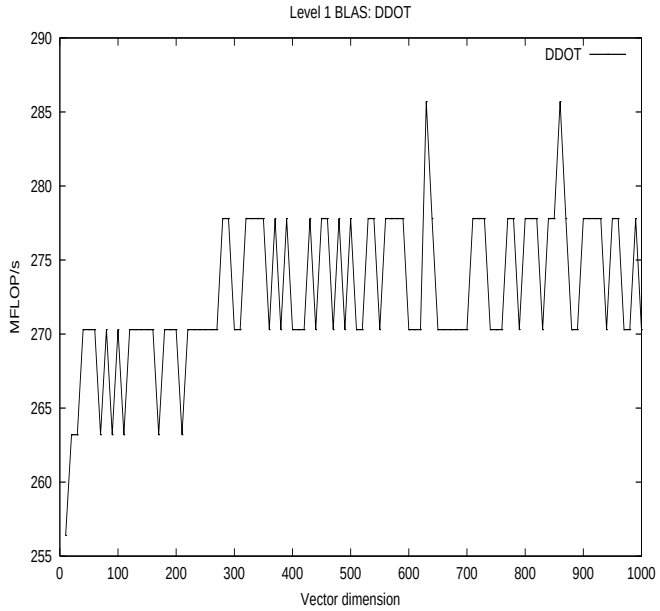


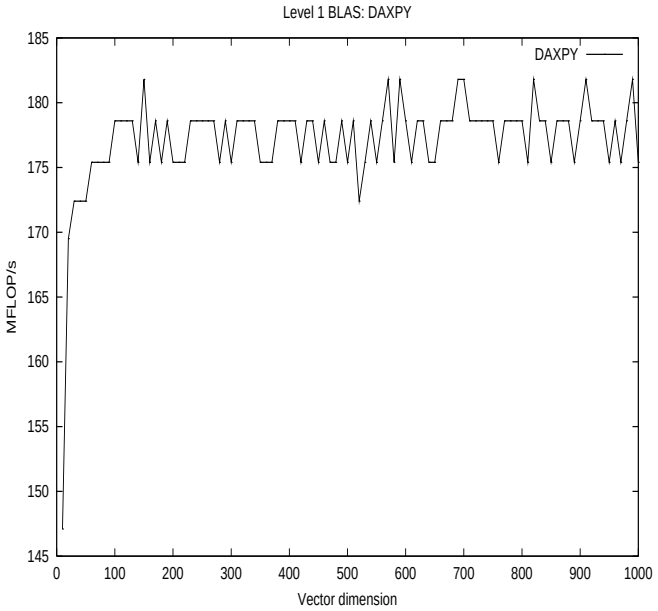


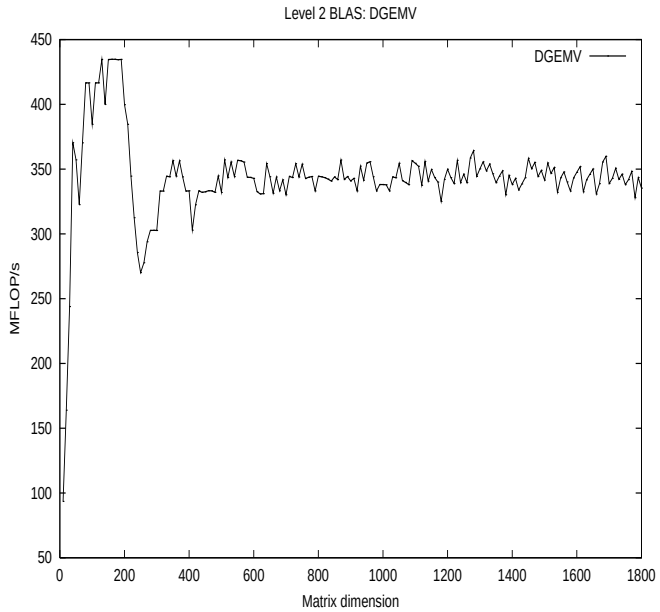
Code	MFLOP/s	$\|Ax-b\|$	$\frac{\|Ax-b\|}{\|A\|\cdot\|x\|n\epsilon}$
Level 1 BLAS			
Linpack source	49.3	10^{-12}	10.5
Linpack BLAS	53.6	10^{-12}	10.5
Level 2 BLAS			
Right-looking	86.1	10^{-13}	2.9
Left-looking	144.7	10^{-13}	1.8
Crout	115.7	10^{-13}	2.0
Level 3 BLAS			
Lapack	410.2	10^{-13}	1.5
Recursive LU	506.6	10^{-13}	1.2

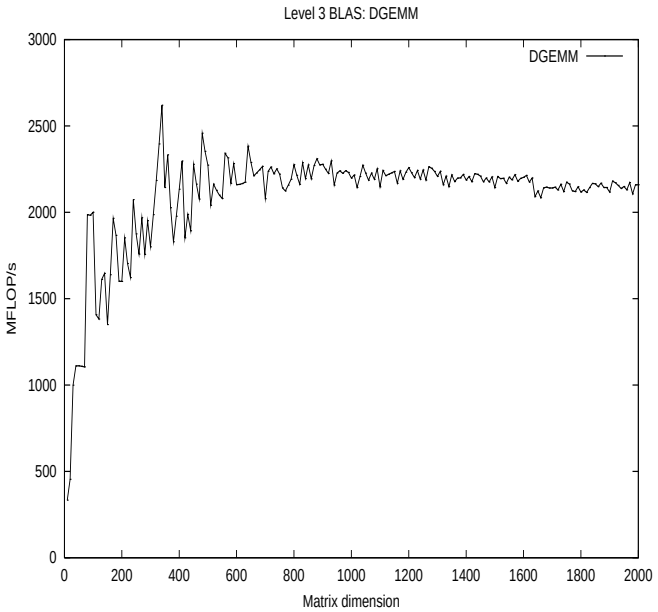
System parameters	
Clock rate [MHz]	933
Bus speed [MHz]	133
L1 cache [KB]	16+16
L2 cache [KB]	256

Performance Results for Intel P4 1700MHz





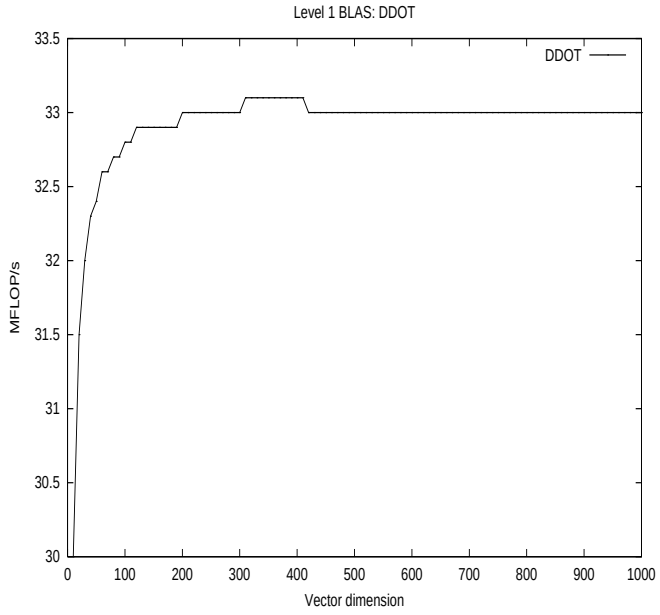


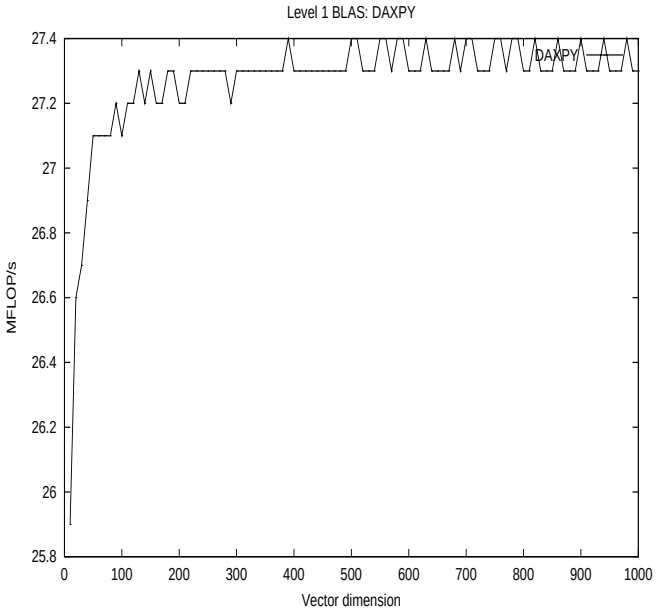


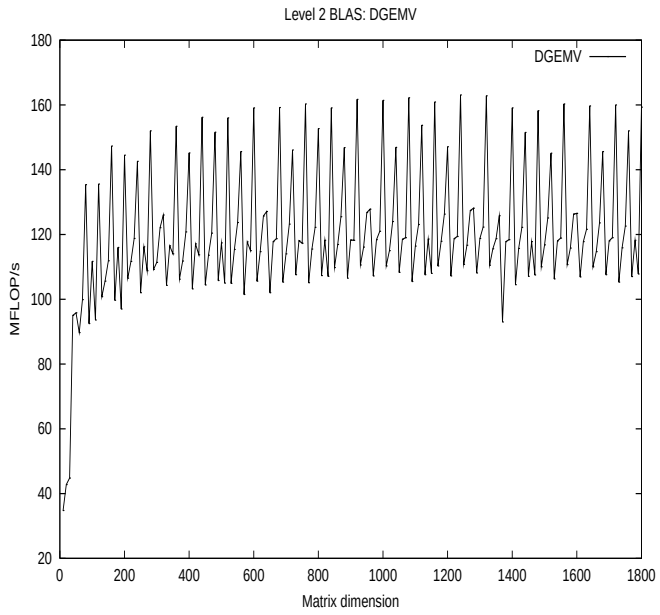
Code	MFLOP/s	$\|Ax-b\|$	$\frac{\|Ax-b\|}{\|A\|\cdot\|x\|n\epsilon}$
Level 1 BLAS			
Linpack source	182.2	10^{-12}	10.5
Linpack BLAS	238.8	10^{-12}	10.5
Level 2 BLAS			
Right-looking	199.6	10^{-13}	2.2
Left-looking	290.7	10^{-13}	1.7
Crout	312.5	10^{-13}	2.8
Level 3 BLAS			
Lapack	1262.0	10^{-13}	8.4
Recursive LU	1393.0	10^{-13}	7.2

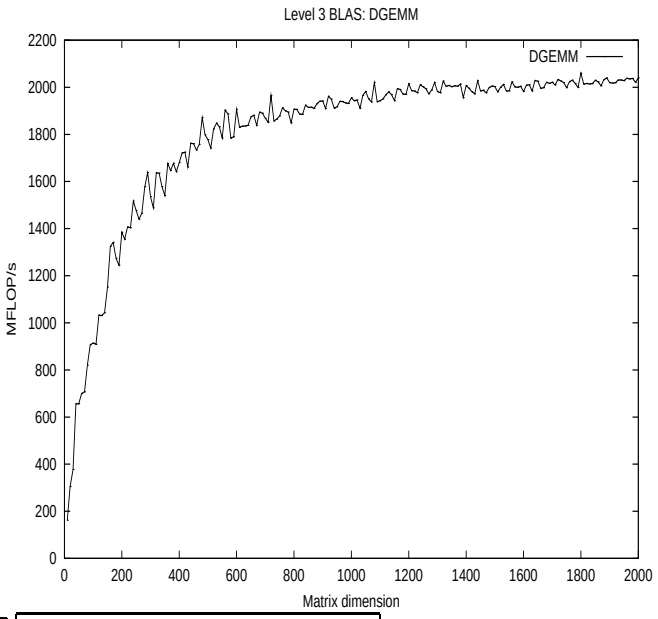
System parameters	
Clock rate [MHz]	1700
Bus speed [MHz]	400
L1 cache [KB]	12(I)+8(D)
L2 cache [KB]	256

Performance Results for Intel/HP Itanium 800MHz





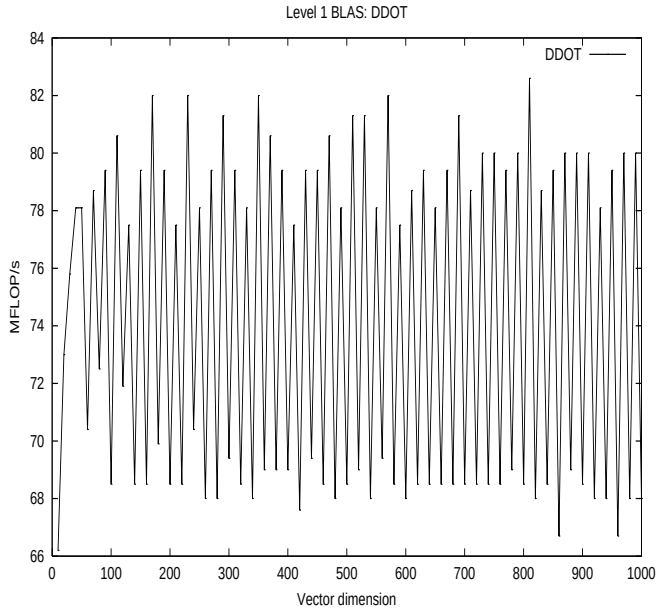


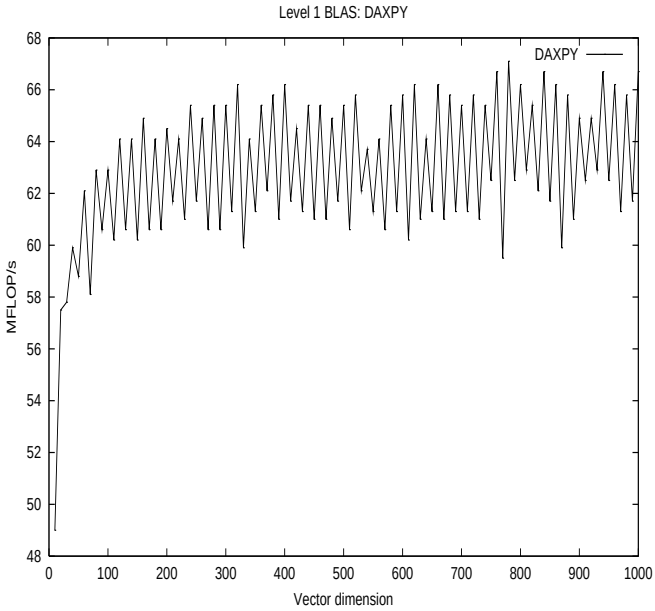


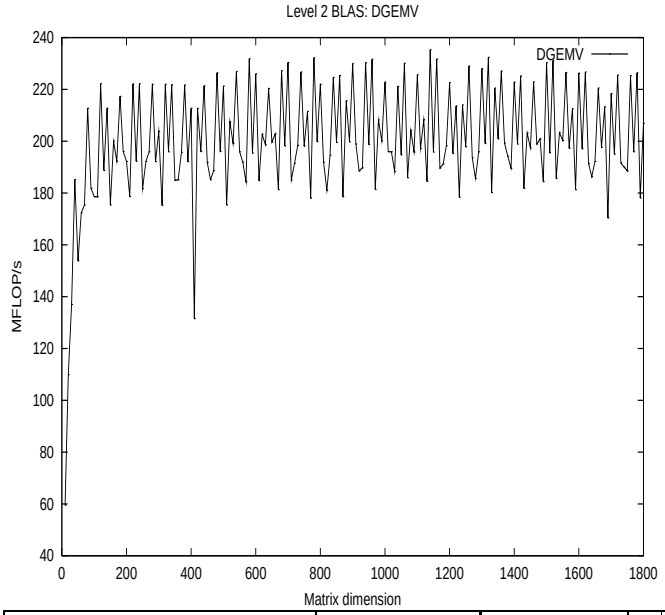
Code	MFLOP/s	$\|Ax-b\|$	$\frac{\|Ax-b\|}{\|A\|\cdot\|x\|n\epsilon}$
Level 1 BLAS			
Linpack source	39.3	10^{-12}	9.4
Linpack BLAS	36.9	10^{-12}	9.4
Level 2 BLAS			
Right-looking	40.5	10^{-13}	6.3
Left-looking	108.5	10^{-12}	10.0
Crout	123.7	10^{-13}	10.2
Level 3 BLAS			
Lapack	624.2	10^{-13}	8.5
Recursive LU	801.8	10^{-13}	6.5

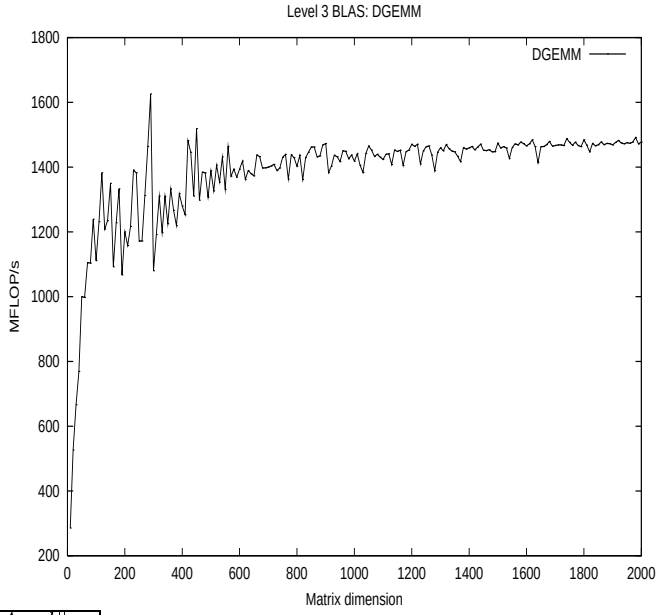
System parameters	
Clock rate [MHz]	800
Bus speed [MHz]	100
L1 cache [KB]	16+16
L2 cache [KB]	96
L3 cache [MB]	2

Performance Results for AMD Athlon 1200MHz





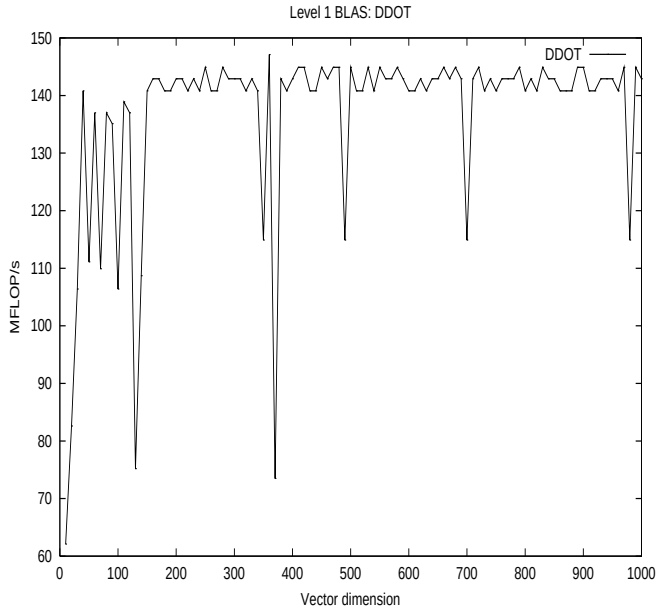


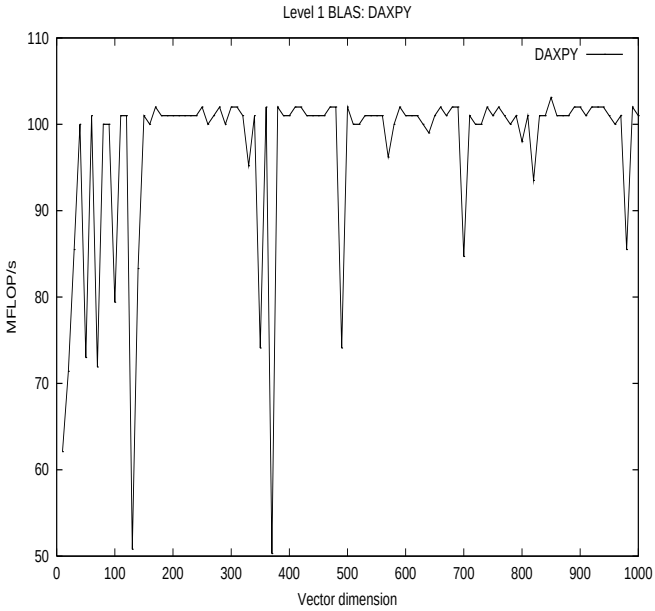


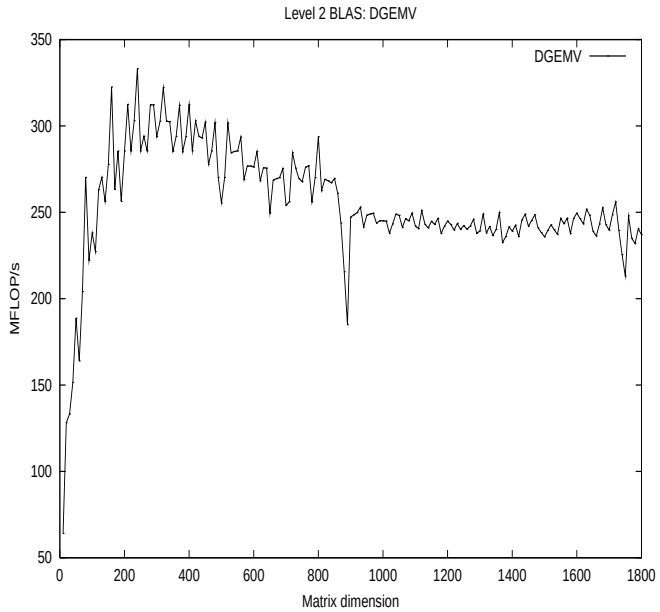
Code	MFLOP/s	$\|Ax-b\|$	$\frac{\|Ax-b\|}{\|A\|\cdot\|x\|n\epsilon}$
Level 1 BLAS			
Linpack 1000 (source)	63.0	10^{-12}	10.5
Linpack 1000 (BLAS)	91.0	10^{-12}	10.5
Level 2 BLAS			
Right-looking	124.3	10^{-13}	6.6
Left-looking	173.7	10^{-13}	7.4
Crout	137.0	10^{-13}	8.8
Level 3 BLAS			
Lapack	835.8	10^{-13}	1.6
Recursive LU	998.3	10^{-13}	1.1

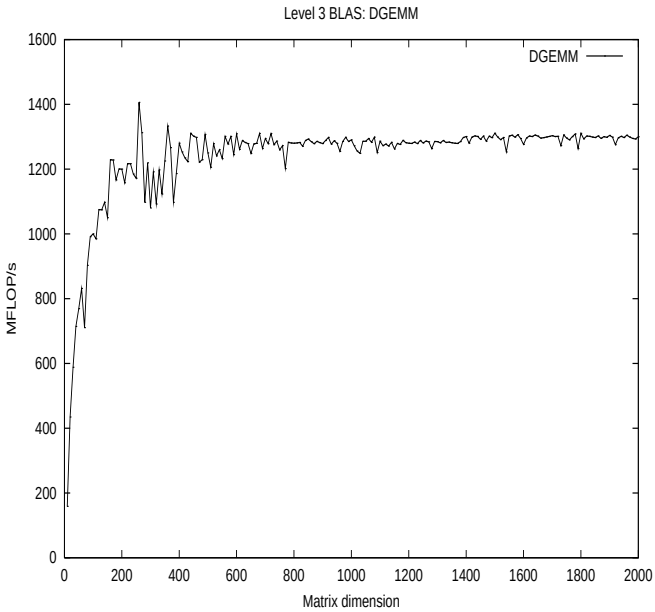
System parameters	
Clock rate [MHz]	1200
Bus speed [MHz]	200
L1 cache [KB]	64+64
L2 cache [KB]	256

Performance Results for IBM Power3 375MHz





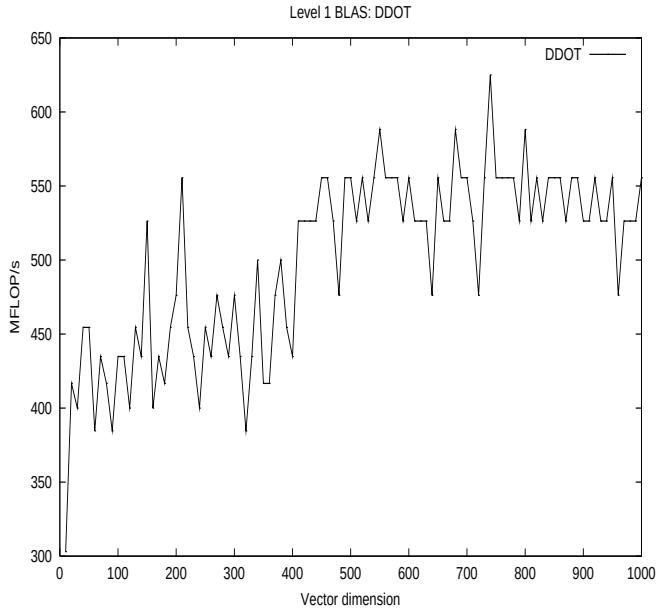


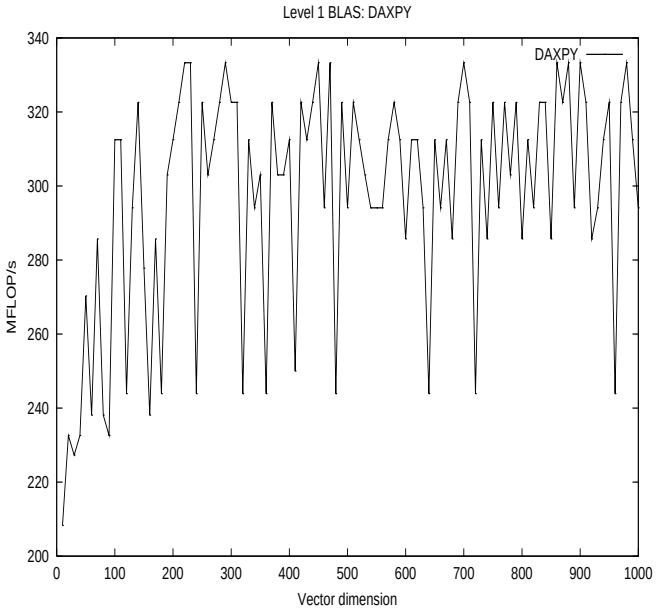


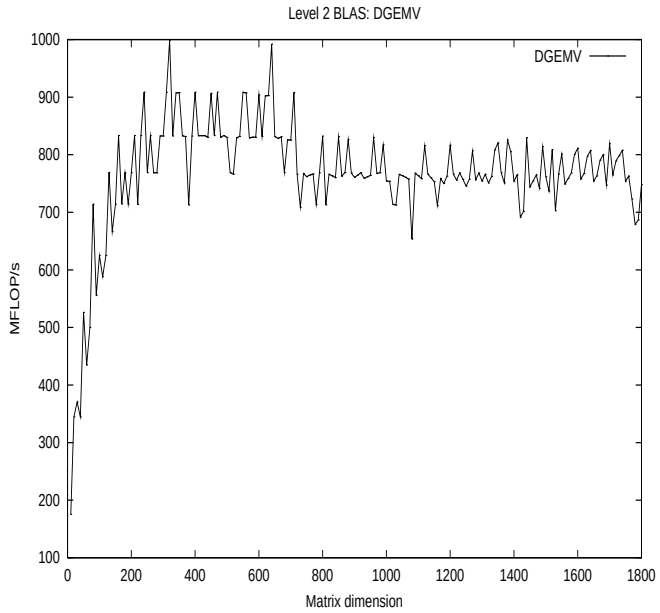
Code	MFLOP/s	$\|Ax-b\|$	$\frac{\|Ax-b\|}{\|A\|\cdot\|x\|n\epsilon}$
Level 1 BLAS			
Linpack source	169.3	10^{-12}	4.7
Linpack BLAS	165.9	10^{-12}	4.7
Level 2 BLAS			
Right-looking	174.6	10^{-13}	4.0
Left-looking	298.5	10^{-12}	4.7
Crout	290.7	10^{-12}	4.7
Level 3 BLAS			
Lapack	941.8	10^{-13}	4.0
Recursive LU	1078.0	10^{-13}	4.5

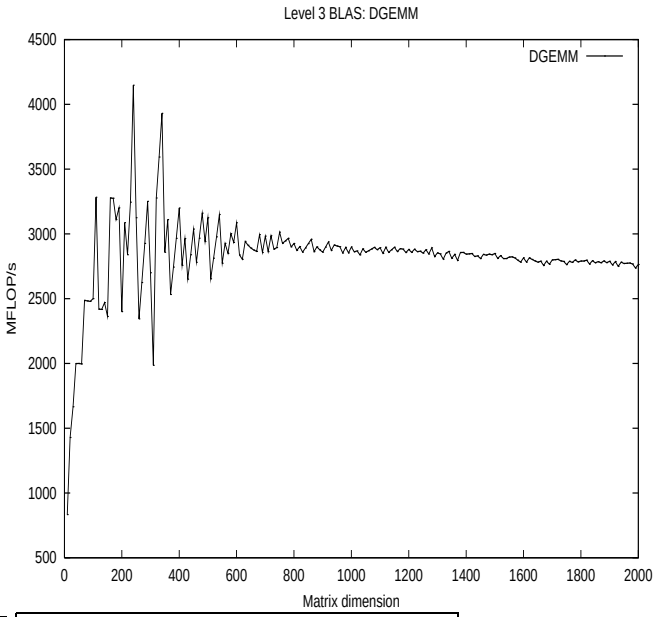
System parameters	
Clock rate [MHz]	375
Bus speed [MHz]	100
L1 cache [KB]	32(I)+64(D)
L2 cache [MB]	8

Performance Results for IBM Power4 1300MHz





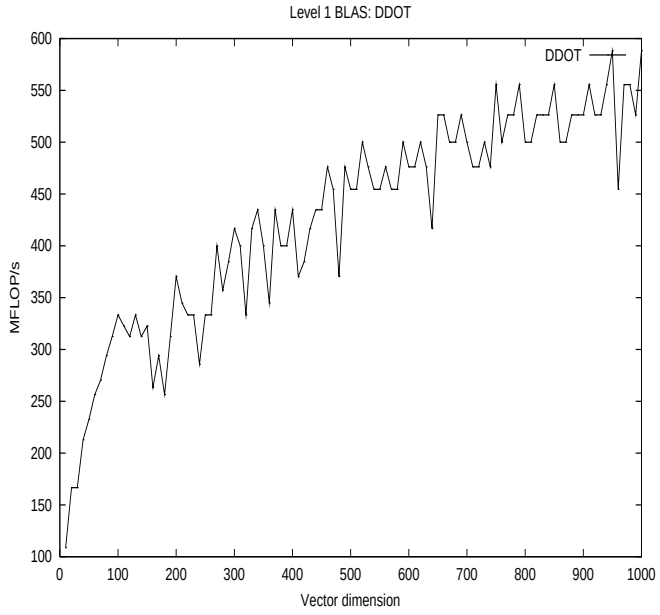


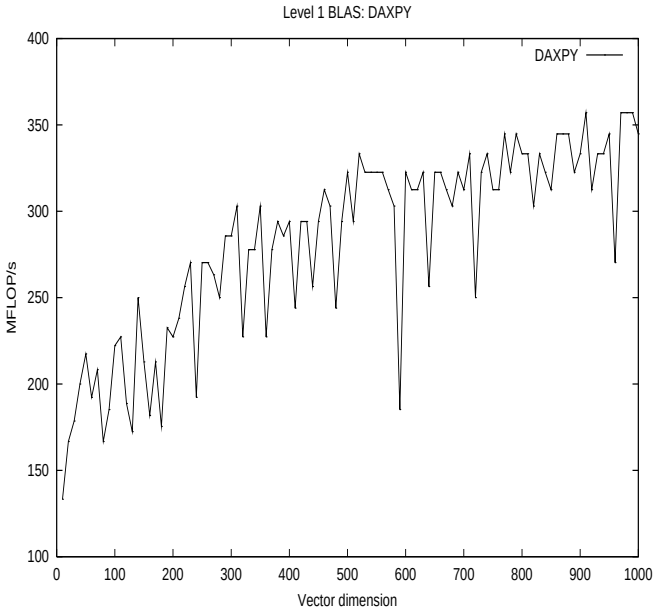


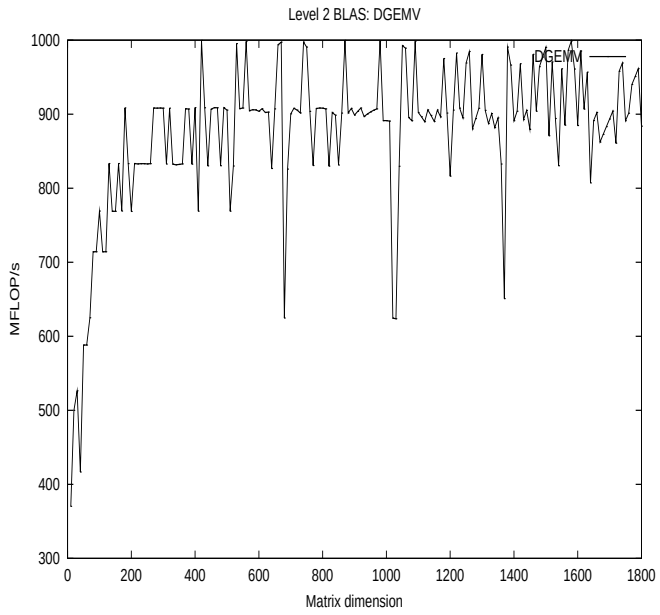
Code	MFLOP/s	$\|Ax-b\|$	$\frac{\|\|Ax-b\|\|}{\|\|A\|\|\cdot\|\|x\|\|n\epsilon}$
Level 1 BLAS			
Linpack source	301.2	10^{-12}	4.7
Linpack BLAS	294.6	10^{-12}	4.7
Level 2 BLAS			
Right-looking	288.2	10^{-12}	5.8
Left-looking	625.0	10^{-12}	6.4
Crout	566.7	10^{-12}	4.8
Level 3 BLAS			
Lapack	2157.0	10^{-12}	5.8
Recursive LU	2388.0	10^{-13}	6.8

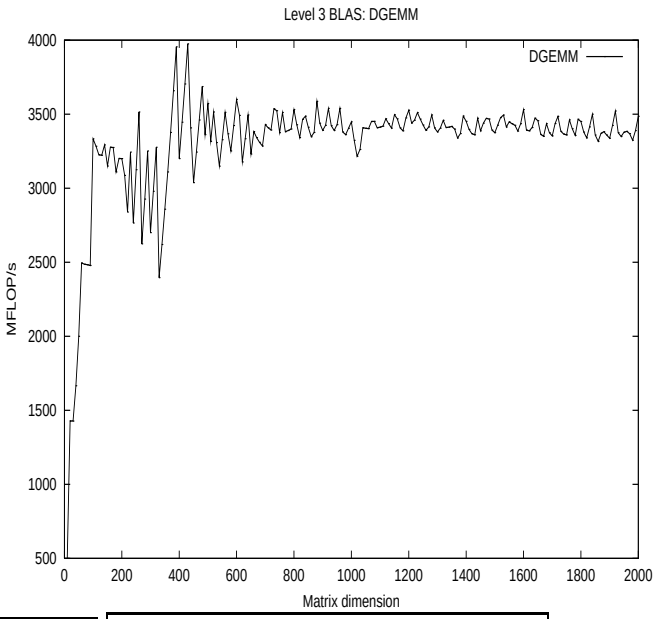
System parameters	
Clock rate [MHz]	1300
Bus speed [MHz]	333
L1 cache [KB]	64(I)+32(D)
L2 cache [MB]	1.4
L3 cache [MB]	32

Performance Results for IBM Power4 1300MHz (ESSL BLAS)





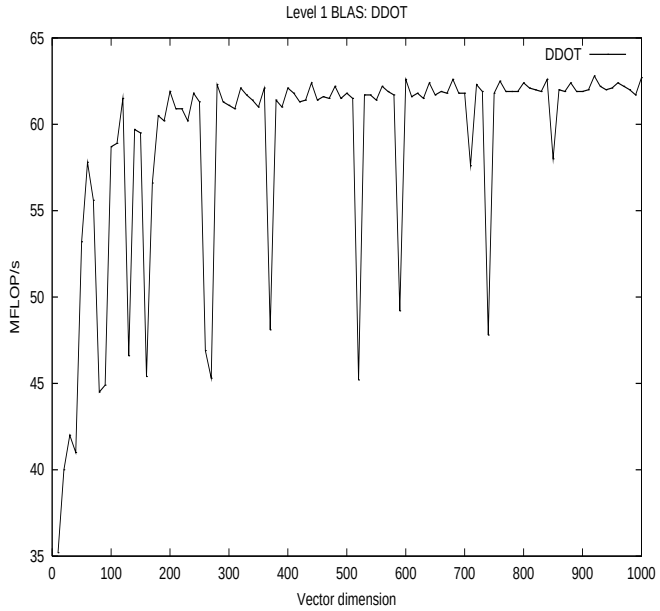


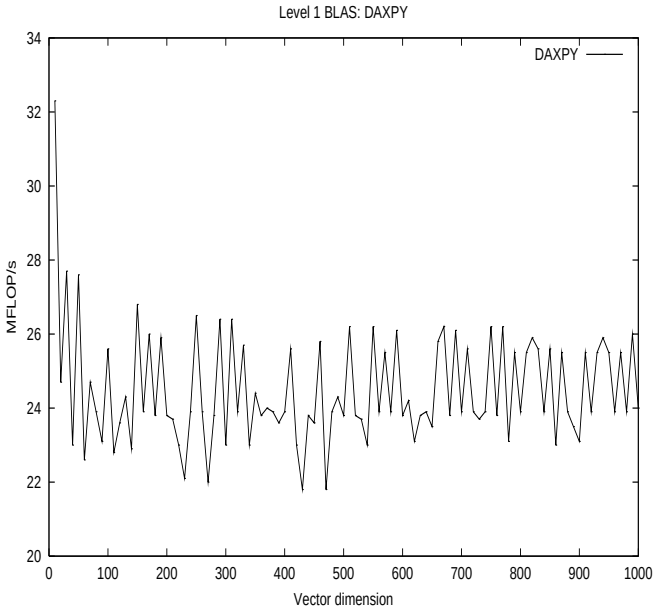


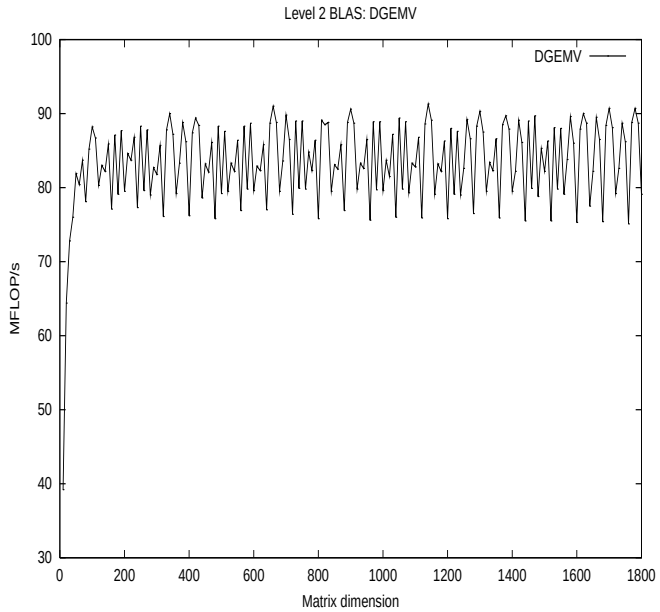
Code	MFLOP/s	$\|Ax-b\|$	$\frac{\|Ax-b\|}{\|A\|\cdot\|x\|n\epsilon}$
Level 1 BLAS			
Linpack 1000 (source)	302.6	10^{-12}	4.7
Linpack 1000 (BLAS)	331.0	10^{-12}	4.7
Level 2 BLAS			
Right-looking	339.4	10^{-12}	7.1
Left-looking	825.5	10^{-12}	7.1
Crout	668.7	10^{-12}	7.1
Level 3 BLAS			
Lapack 1000	2477.0	10^{-12}	7.1
Recursive LU 1000	2786.0	10^{-12}	6.8

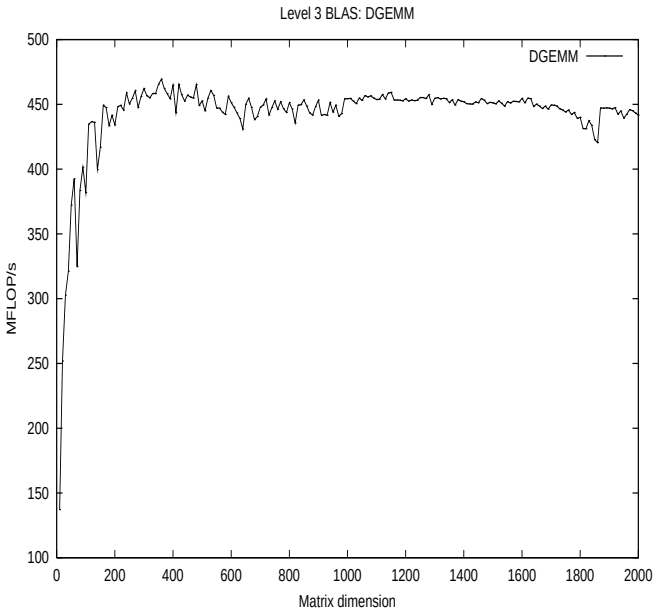
System parameters	
Clock rate [MHz]	1300
Bus speed [MHz]	333
L1 cache [KB]	64(I)+32(D)
L2 cache [MB]	1.4
L3 cache [MB]	32

Performance Results for SGI Octane R12000 IP30 270MHz





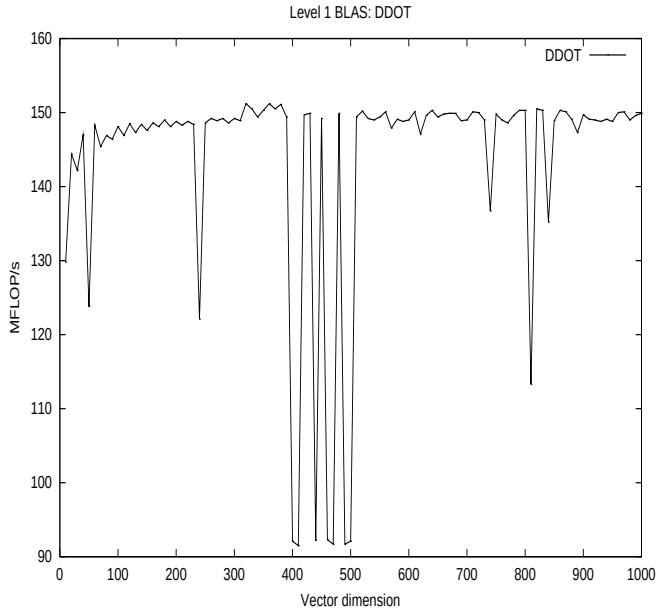


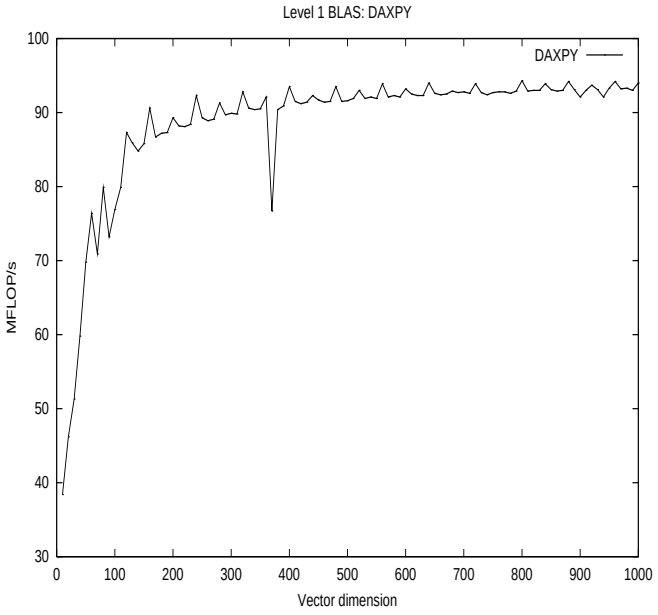


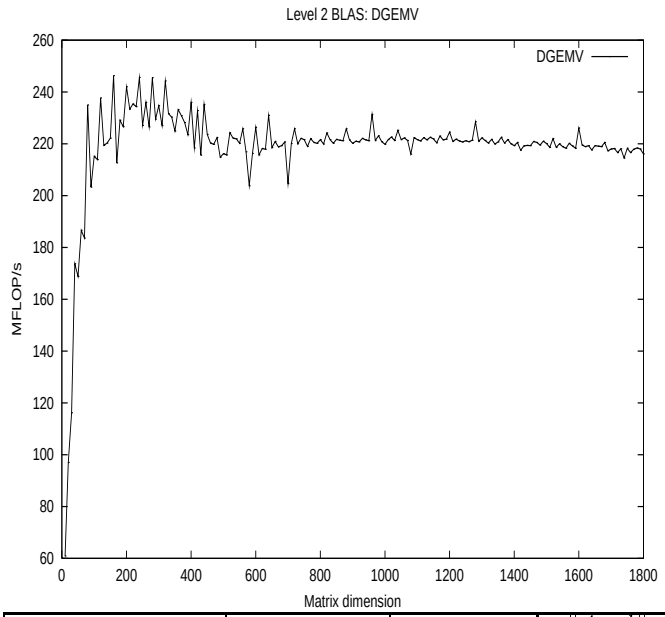
Code	MFLOP/s	$\|Ax-b\|$	$\frac{\|Ax-b\|}{\|A\|\cdot\|x\|n\epsilon}$
Level 1 BLAS			
Linpack source	60.8	10^{-13}	6.4
Linpack BLAS	31.8	10^{-13}	6.4
Level 2 BLAS			
Right-looking	38.1	10^{-13}	7.3
Left-looking	93.0	10^{-13}	8.3
Crout	88.5	10^{-13}	7.9
Level 3 BLAS			
Lapack	336.0	10^{-13}	7.2
Recursive LU	400.4	10^{-13}	6.9

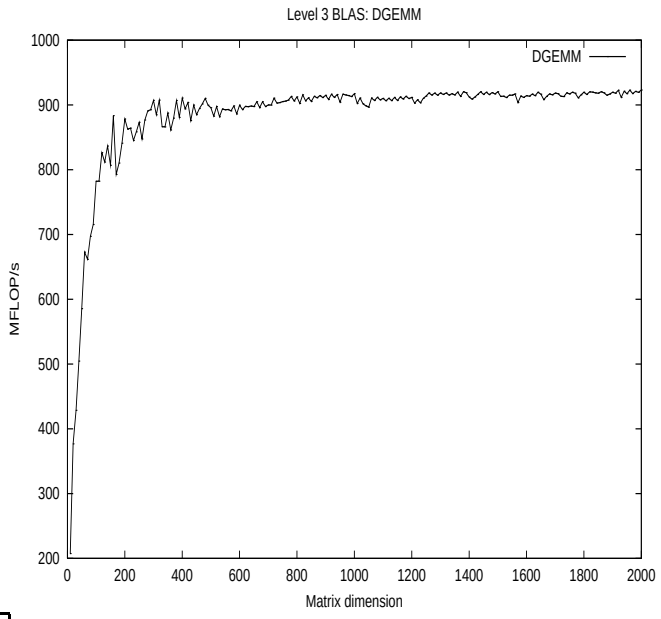
System parameters	
Clock rate [MHz]	270
Bus speed [MHz]	100
L1 cache [KB]	32+32
L2 cache [MB]	2

Performance Results for Compaq/DEC Alpha 21264 EV67 500MHz





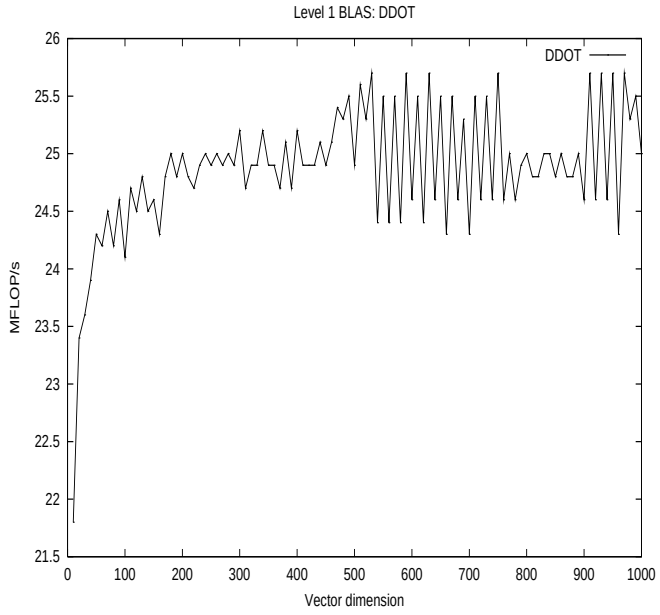


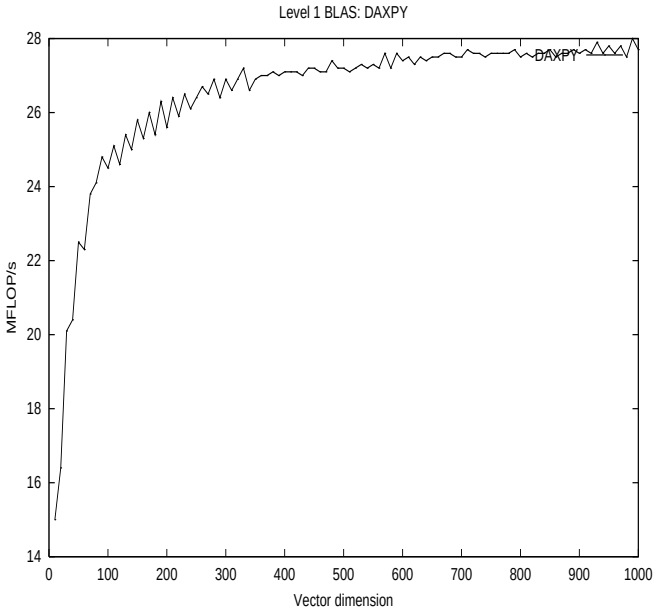


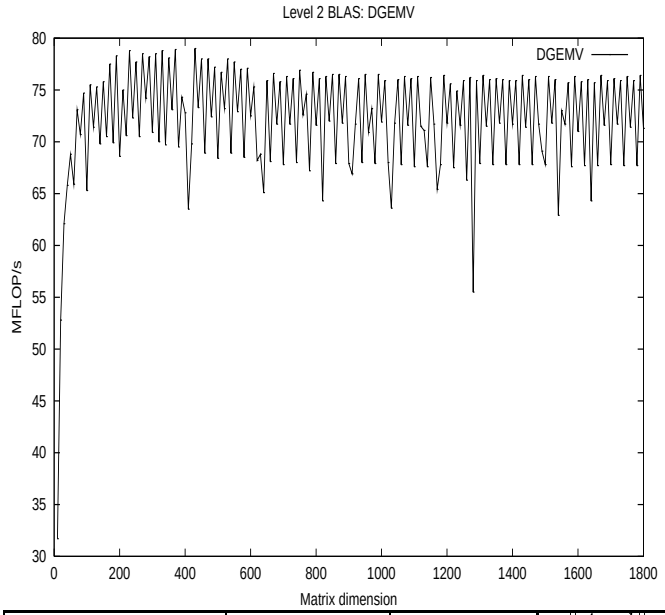
Code	MFLOP/s	$\|Ax-b\|$	$\frac{\|Ax-b\|}{\|A\|\cdot\|x\|n\epsilon}$
Level 1 BLAS			
Linpack source	187.5	10^{-13}	6.5
Linpack BLAS	167.9	10^{-13}	6.5
Level 2 BLAS			
Right-looking	159.2	10^{-13}	7.2
Left-looking	188.4	10^{-13}	8.0
Crout	214.6	10^{-13}	8.0
Level 3 BLAS			
Lapack	565.1	10^{-13}	8.5
Recursive LU	636.9	10^{-13}	7.7

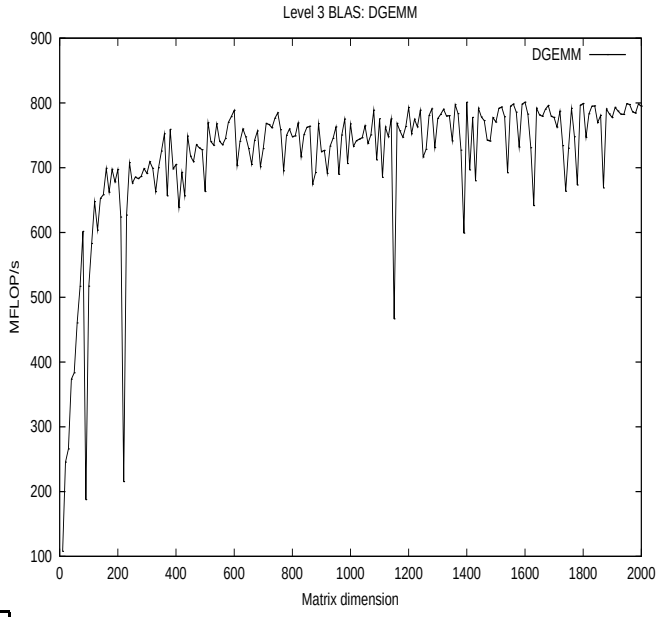
System parameters	
Clock rate [MHz]	500
Bus speed [MHz]	333
L1 cache [KB]	64+64
L2 cache [MB]	4

Performance Results for Compaq/DEC Alpha 21164 533MHz





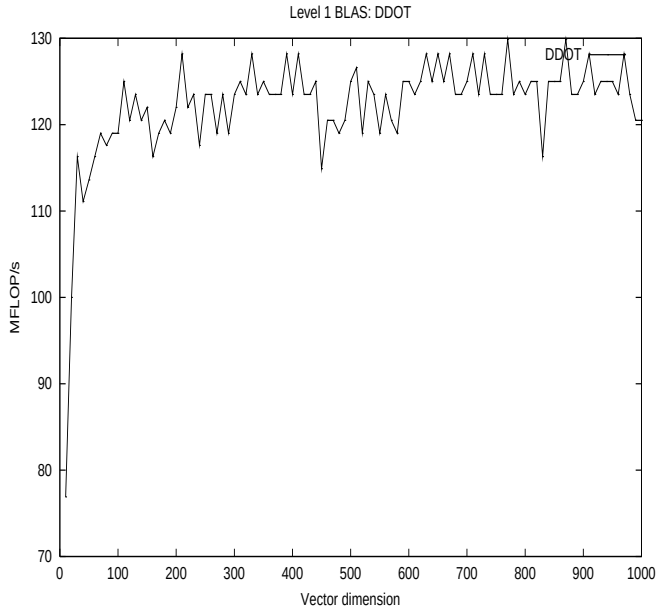


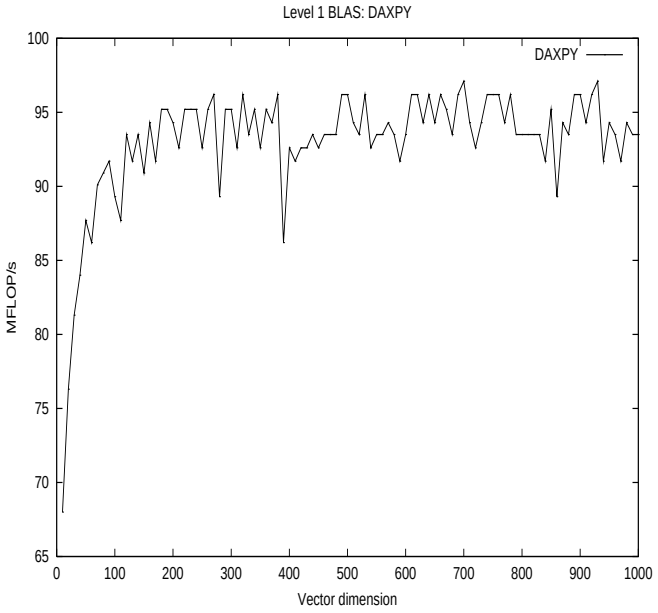


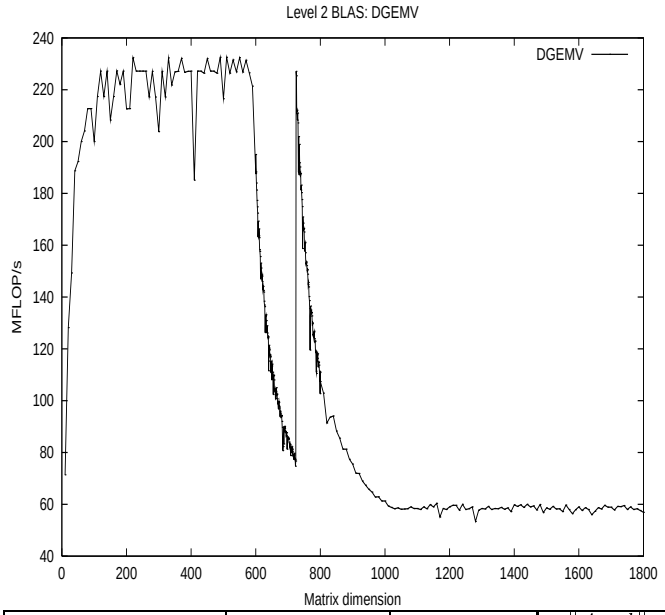
Code	MFLOP/s	$\|Ax-b\|$	$\frac{\|Ax-b\|}{\|A\|\cdot\|x\|n\epsilon}$
Level 1 BLAS			
Linpack source	34.0	10^{-13}	6.5
Linpack BLAS	48.1	10^{-13}	6.5
Level 2 BLAS			
Right-looking	45.9	10^{-13}	5.8
Left-looking	76.7	10^{-13}	8.1
Crout	74.6	10^{-13}	7.3
Level 3 BLAS			
Lapack	425.6	10^{-13}	8.1
Recursive LU	500.9	10^{-13}	6.5

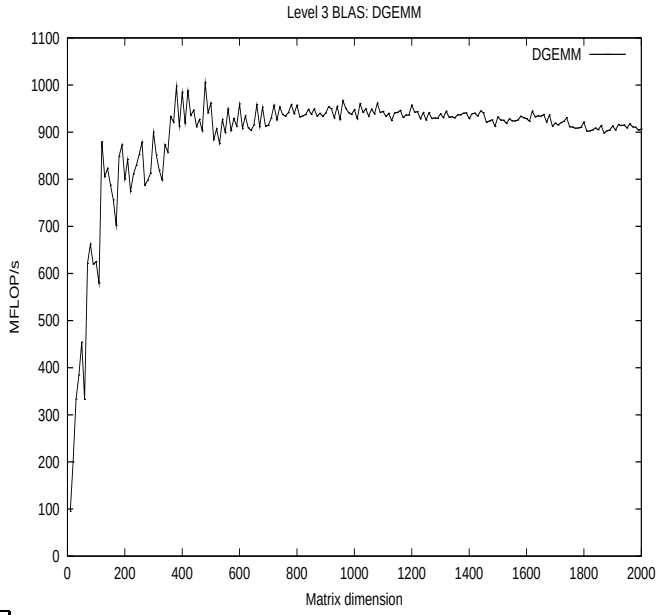
System parameters	
Clock rate [MHz]	533
Bus speed [MHz]	88
L1 cache [KB]	8+8
L2 cache [KB]	96

Performance Results for Sun UltraSparc III 750MHz





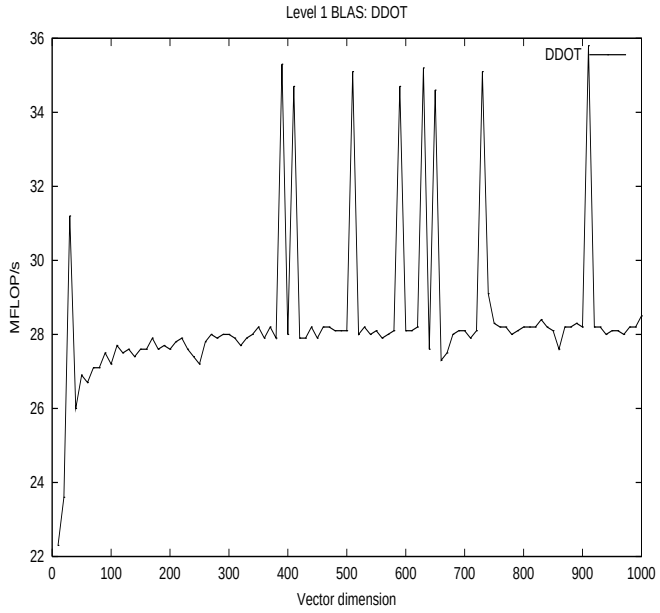


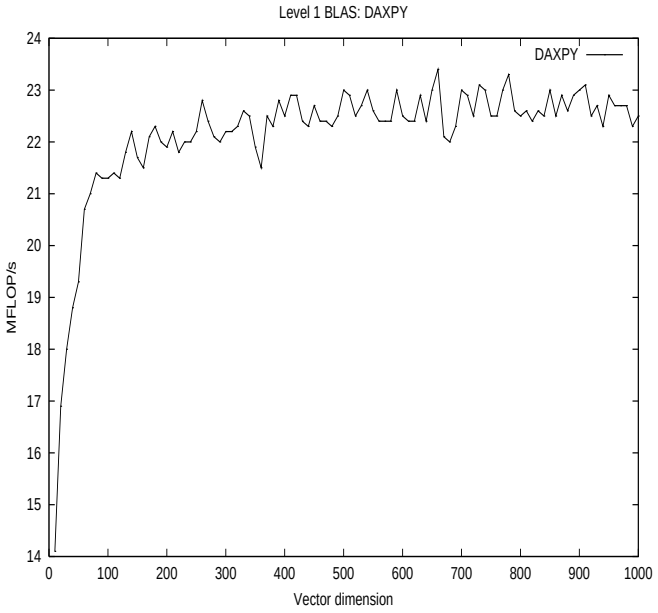


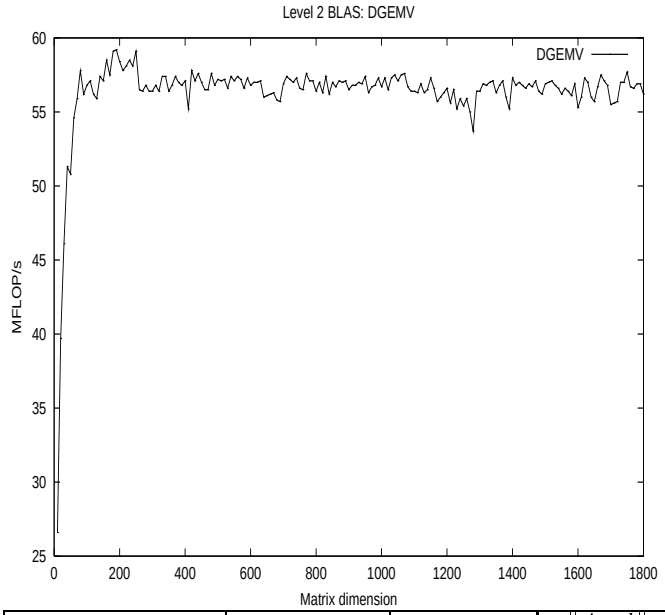
Code	MFLOP/s	$\|Ax-b\|$	$\frac{\|Ax-b\|}{\|A\|\cdot\|x\|n\epsilon}$
Level 1 BLAS			
Linpack source	171.5	10^{-13}	6.5
Linpack BLAS	123.8	10^{-13}	6.5
Level 2 BLAS			
Right-looking	150.9	10^{-12}	9.9
Left-looking	219.2	10^{-12}	9.9
Crout	205.7	10^{-13}	7.6
Level 3 BLAS			
Lapack	675.4	10^{-12}	9.9
Recursive LU	734.8	10^{-12}	9.6

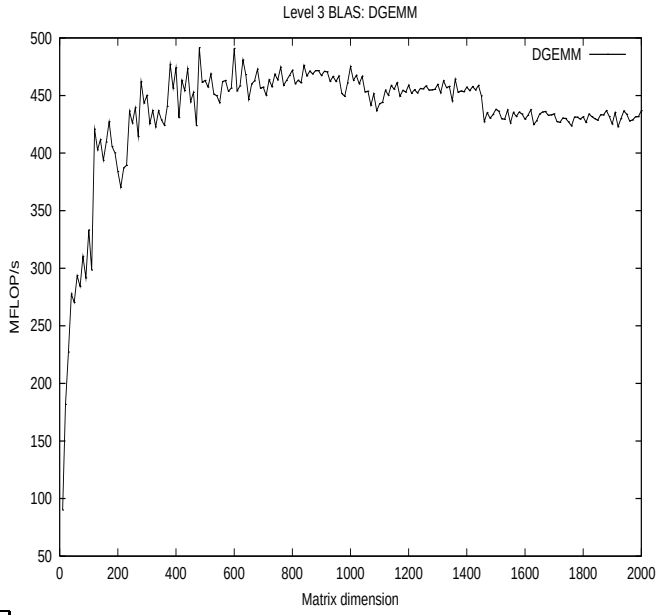
System parameters	
Clock rate [MHz]	750
Bus speed [MHz]	150
L1 cache [KB]	64(I)+32(D)
L2 cache [MB]	1

Performance Results for Sun UltraSparc II 300MHz





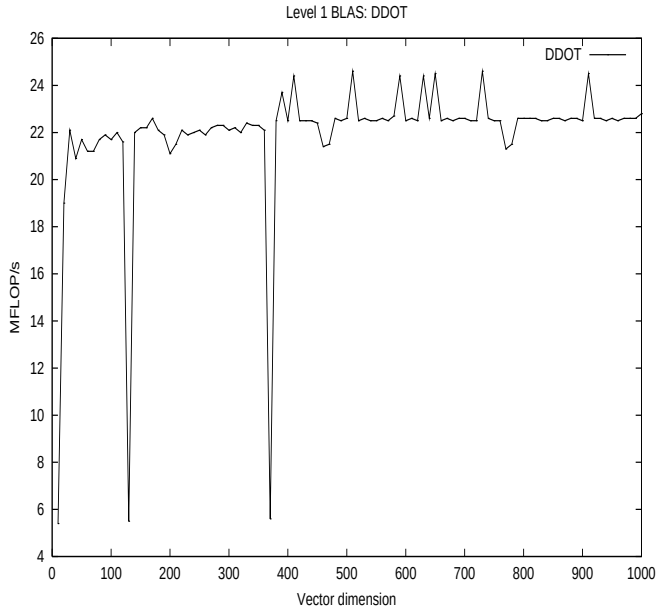


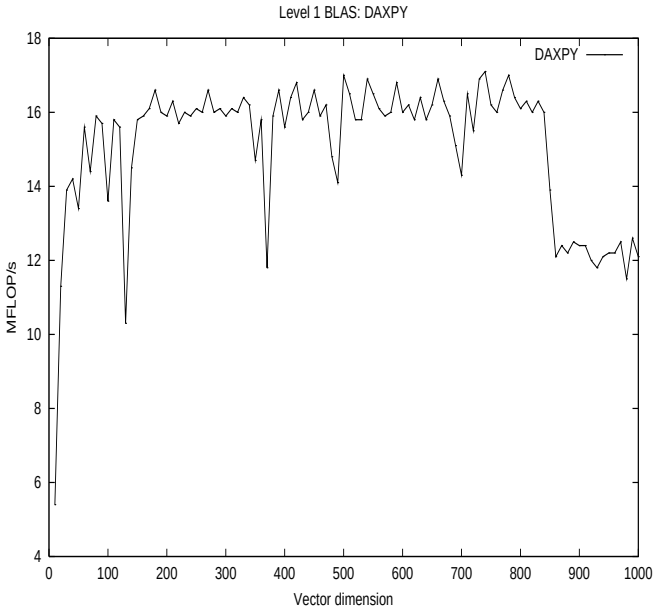


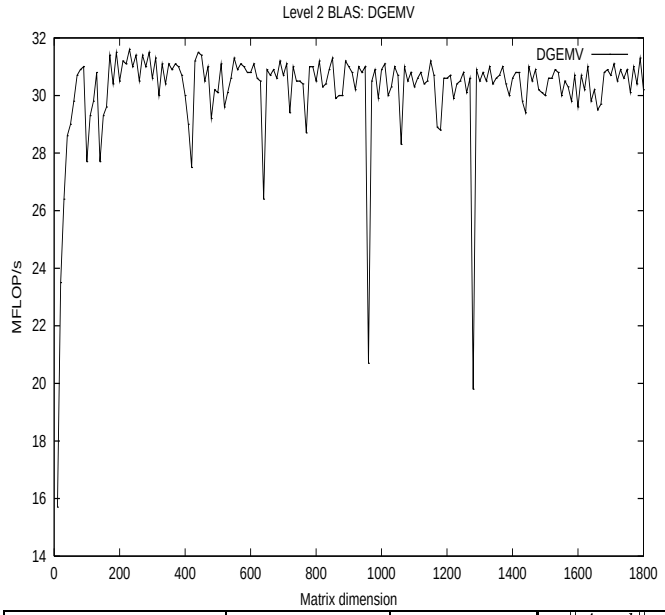
Code	MFLOP/s	$\|Ax-b\|$	$\frac{\|Ax-b\|}{\|A\|\cdot\|x\|n\epsilon}$
Level 1 BLAS			
Linpack source	42.2	10^{-13}	6.5
Linpack BLAS	37.5	10^{-13}	6.5
Level 2 BLAS			
Right-looking	37.1	10^{-13}	8.9
Left-looking	66.8	10^{-13}	7.9
Crout	65.9	10^{-13}	8.0
Level 3 BLAS			
Lapack	285.8	10^{-13}	8.8
Recursive LU	284.5	10^{-12}	9.0

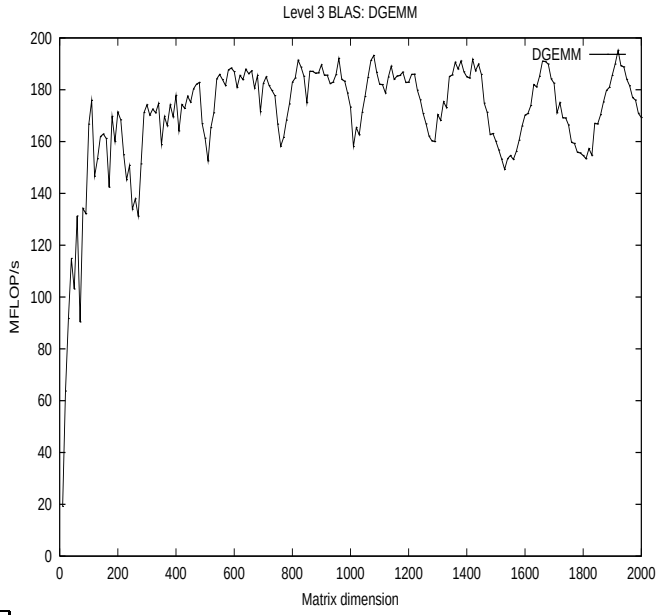
System parameters	
Clock rate [MHz]	300
Bus speed [MHz]	100
L1 cache [KB]	16+16
L2 cache [MB]	2

Performance Results for Sun UltraSparc II 250MHz





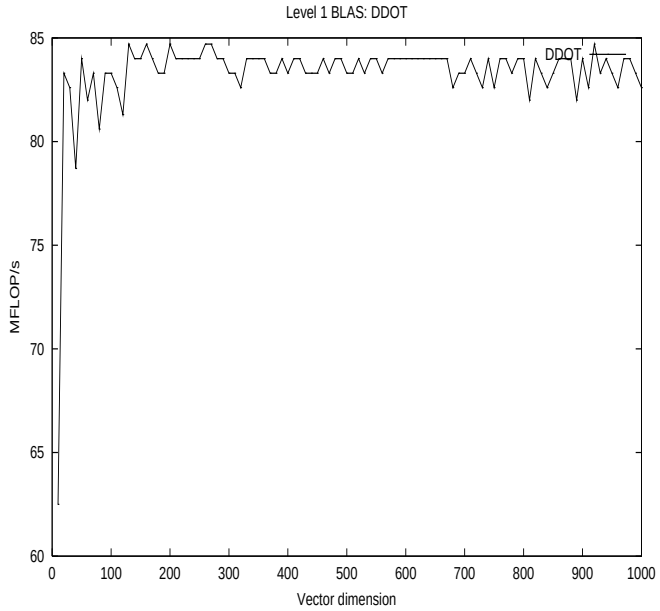


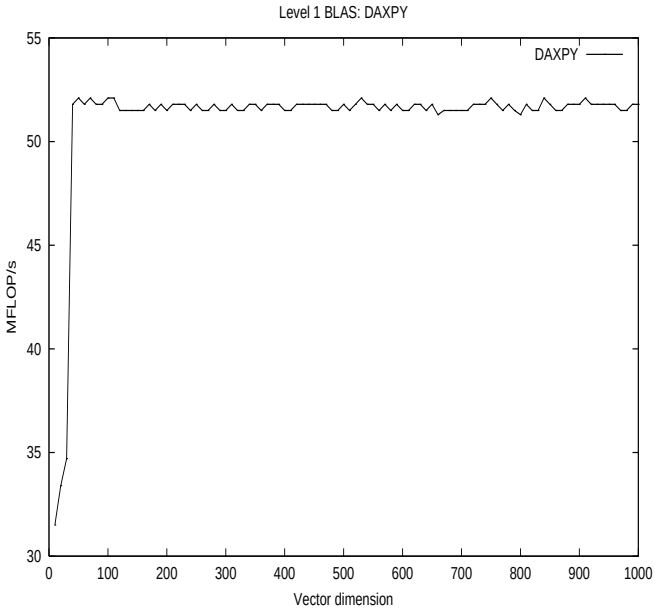


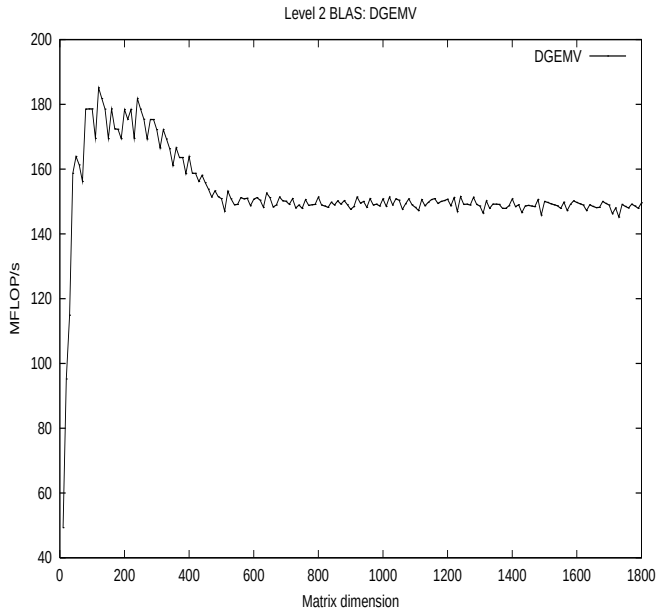
Code	MFLOP/s	$\|Ax-b\|$	$\frac{\|Ax-b\|}{\|A\|\cdot\|x\|n\epsilon}$
Level 1 BLAS			
Linpack source	21.4	10^{-13}	6.5
Linpack BLAS	18.2	10^{-13}	6.5
Level 2 BLAS			
Right-looking	18.4	10^{-13}	7.5
Left-looking	28.0	10^{-12}	10.2
Crout	28.8	10^{-12}	10.1
Level 3 BLAS			
Lapack	109.3	10^{-13}	7.5
Recursive LU	110.3	10^{-12}	1.1

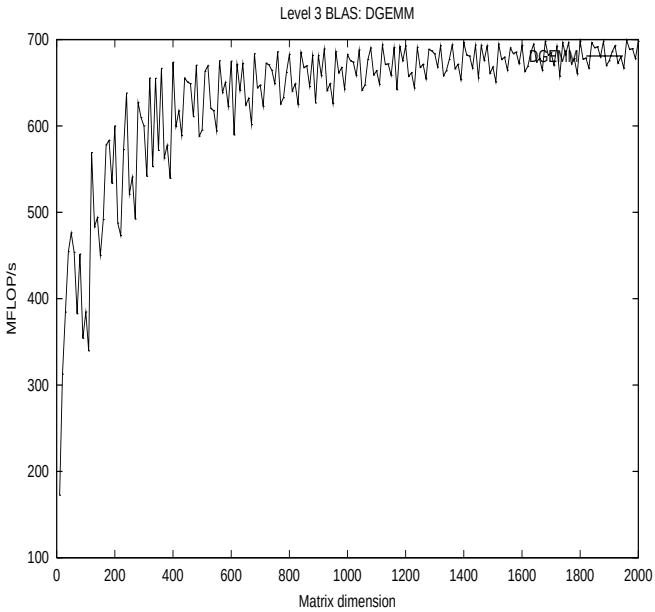
System parameters	
Clock rate [MHz]	250
Bus speed [MHz]	83
L1 cache [KB]	16+16
L2 cache [MB]	1

Performance Results for PowerPC G4 533MHz









Code	MFLOP/s	$\|Ax-b\|$	$\frac{\|Ax-b\|}{\|A\|\cdot\|x\|n\epsilon}$
Level 1 BLAS			
Linpack source	52.6	10^{-12}	9.4
Linpack BLAS	76.2	10^{-12}	9.4
Level 2 BLAS			
Right-looking	85.2	10^{-13}	8.1
Left-looking	125.2	10^{-13}	7.4
Crout	120.6	10^{-13}	8.2
Level 3 BLAS			
Lapack	412.8	10^{-13}	8.2
Recursive LU	477.6	10^{-12}	7.2

System parameters	
Clock rate [MHz]	533
Bus speed [MHz]	100
L1 cache [KB]	32+32
L2 cache [MB]	2