Evolution of Numerical Software for Dense Linear Algebra

Jack Dongarra *

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439-4801

Sven Hammarling

Numerical Algorithms Group Ltd.
Wilkinson House
Jordan Hill Road
Oxford OX2 8DR

. *gsize* + 1

To the memory of Jim Wilkinson for his inspiration and encouragement.

1. Introduction

We wish to trace the development of numerical software for dense linear algebra from the early days of computers through to work in progress for modern high-performance machines.

Jim Wilkinson was a great influence on the development of algorithms for numerical linear algebra and we highlight his influence, as well as those things that influenced his ideas, particularly in the early days. We believe that there are still lessons to be learned, or remembered, by looking at the historical development.

2. Early Days

Up to the 1950s the principal aid to computation was the mechanical calculator. The art of computation and numerical analysis in the first half of the Century is epitomized by such classic books as Whittaker and Watson [1927], Whittaker and Robinson [1924] and Southwell [1940]. Indeed Leslie Fox, Wilkinson's great friend and early colleague at NPL, was a student of Southwell and can still demonstrate a remarkable skill in the relaxation method.

Wilkinson joined NPL in 1946 working jointly for the Desk Machine Section and for Alan Turing on a project to build an Automatic Computing Engine. In his work for the Desk Machine Section, Wilkinson often used a Brunsviga for his calculation and this was an important influence on his understanding of machine arithmetic and his expectations of machine arithmetic for two reasons. Firstly and most

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

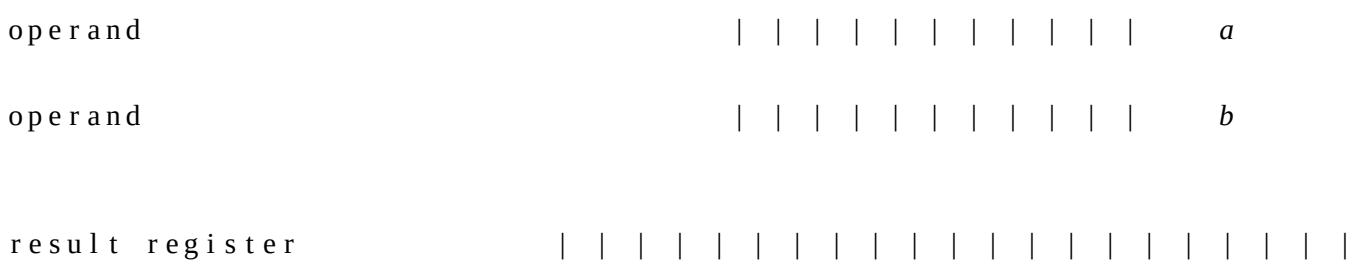
Typeset on 25 October 2018.

importantly, a hand calculator readily gives one the ability to watch the calculations proceed, something which Wilkinson certainly thought valuable:

> *"But in my experience, many people who do computing are reluctant to look at numbers.*
>
> *...*
>
> *I certainly do not want to suggest that the way to acquire the habit is to serve an apprenticeship on hand desk calculators, but we have yet to learn how to instill the relevant knowledge."*
>
> *From an interview in BYTE, February, 1985, pp 177 - 194.*

MATLAB, which we shall mention again later, is one example of a system that attempts to give this experience.

A second feature of the Brunsviga (and other similar calculators), which influenced Wilkinson is the ability to perform extended precision arithmetic, a feature that in particular allows one to perform accumulated inner products (see Figure 1).



Schematic of a hand calculator
Figure 1

The poor arithmetic on many modern computers was a disappointment to Wilkinson (see for example, p.146 of [Wilkinson, 1971]) and he, like us believed that we all owe a debt of gratitude to Velvel Kahan for his tireless crusade to put things right.

During his time at the Armament Research Department, Wilkinson had been asked to solve a system of twelve linear equations and, soon after joining NPL, a system of eighteen equations arose on which a joint operation with Fox, Goodwin, Turing and Wilkinson (a rather awesome set of processors) was mounted. The experience of solving these two systems of equations is described in Wilkinson's 1970 Turing Lecture [Wilkinson, 1971] and was undoubtably a fundamental ingredient in forming his ideas on backward error analysis.

The equations were solved using Gaussian elimination with pivoting. In both cases the equations were mildly ill-conditioned and figures were slowly lost during the elimination process until in the final equation

$$u_{n,n}x_n = \beta_n,$$

$u_{n,n}$ and β_n had lost about four figures relative to $u_{1,1}$ and β_1. Wilkinson realized that the solution must surely have lost about four figures in accuracy, but in substituting back into the original equations (using accumulated inner products) it was found that the left-hand side agreed with the right-hand side to the full ten figures of working accuracy.

The significance of the results was that, although the solution was almost certainly not accurate, it was the solution of a closely neighboring problem. That is, if $\bar{x}$ denotes the computed solution of the equations

$$Ax = b \tag{2.1}$$

then $\bar{x}$ satisfied

$$A\bar{x} = b + r, \quad with \quad \| r \| = \varepsilon \| A \| \| x \|,$$

where ε is of the order of machine accuracy.

To express this in the more familiar form used later by Wilkinson, if we let E be the matrix

$$E = \frac{r\bar{x}^T}{\bar{x}^T\bar{x}}$$

so that

$$\| E \|_F \leq \frac{\| r \|_2}{\| \bar{x} \|_2} = \varepsilon \| A \|_F,$$

$$r = E\bar{x} \quad \text{and} \quad (A + E)\bar{x} = b.$$

Since $\| r \| \leq \| E \| \| \bar{x} \|$, $\| E \|_F \geq \| r \|_2 / \| \bar{x} \|_2$ and it follows that

$$(A + E)\bar{x} = b, \; with \; \| E \|_F = \varepsilon \| A \|_F. \tag{2.2}$$

So again $\bar{x}$ is expressed as the solution of a closely neighboring problem. Wilkinson attributes the experience of solving the eighteen equations as a strong stimulus to Turing to write his famous paper on rounding errors [Turing, 1948], which with the paper by von Neumann and Goldstine [1947] helped pave the way to an understanding of Gaussian elimination.

As mentioned earlier, Wilkinson was involved in the project to design and build the ACE and this project soon absorbed all his time. The Pilot ACE machine first worked in May 1950. It had mercury delay lines each with a capacity of 32, 32-bit words and instructions that could perform operations on all 32 numbers in a delay line. By the standards of the time it had fast floating point arithmetic with one word allocated for the mantissa and one word for the exponent, and accumulated inner products. Background information to Wilkinson's involvement in building Pilot ACE can be found in [Wilkinson, 1971; Wilkinson, 1980 and Fox, 1987]. We mention the machine here because it had a profound effect on numerical linear algebra and software for numerical linear algebra due to the work of Wilkinson, who had an intimate knowledge of Pilot ACE through his work on the design, building, operation and programming of the machine.

> *"Since the use of the punched-card equipment required the use of an operator, it encouraged user participation generally, and this was a distinctive feature of Pilot ACE operation.*
>
> *...*
>
> *Speaking for myself I gained a great deal of experience from user participation, and it was this that led to my own conversion to backward error analysis."*
>
> *[Wilkinson, 1980.]*

This background with the Desk Machine Section and Pilot ACE, together with his education in Pure Mathematics enabled Wilkinson to take the fundamental step of developing analytic methods for explaining stability and for giving precise error bounds.

As an example of the result of a backward error analysis, Wilkinson showed that with Gaussian elimination for solving equation (2.1) the computed solution $\bar{x}$ satisfies (2.2) and ε satisfies a bound of the form $\varepsilon \leq g f_n u$, where g is the "growth factor", f_n is a modest function of n (the order of A) and u is the unit rounding error, or relative machine precision. Of course, in general, pivoting is needed to control the size of g, but $g = 1$ for positive definite A, or if orthogonal transformations are used in place of elementary transformations. Wilkinson gave similar details for many classes of algorithms of numerical linear algebra [Wilkinson, 1963; Wilkinson, 1965]. Although he gave precise bounds, for example an explicit expression for f_n above, his aim always was to expose the strengths and weaknesses of algorithms and to aid our understanding of numerical stability.

This work, together of course with the work of many others, such as Wallace Givens, gave a firm foundation for the development of numerical software for linear algebra.

3. The Software Basis

The first published subroutine library was a set of machine code routines for EDSAC [Wilkes, Wheeler and Gill, 1951], Cambridge University's stored-program computer which began operation on 6th May, 1949 [Wilkes, 1977]. This library contained just two linear algebra routines:

a) $z \leftarrow x \pm y$,

b) $y \leftarrow Ax$, *where* $A = A^T$ *stored in packed form*.

During the 1960's a number of Algol procedures were developed and published in the journal Numerische Mathematik. Wilkinson, together with Christian Reinsch edited a collection of these procedures, together with background material, into a volume entitled Linear Algebra in the Handbook for Automatic Computation series [Wilkinson and Reinsch, 1971]. The selected algorithms represented the best available in terms of their generality, elegance, accuracy, speed and economy of storage. The volume is now generally referred to simply as "The Handbook" and represents a landmark in the development of numerical software.

The Handbook contained 41 procedures associated with solving linear systems and 43 procedures for the eigenvalue problem. Many of the procedures used accumulated inner products and all were accompanied by comprehensive documentation. The Handbook formed the basis for a number of software projects including EISPACK, a number of linear algebra routines in the IMSL Library and the F chapters of the NAG Libraries.

4. Software Development

Wilkinson visited Nottingham University in 1968 to give advice about software for linear algebra and in February 1970 the Nottingham Algorithms Group was formed. Mark 1 of the library was released on October 1, 1971. In 1973 the Central Office moved to Oxford and NAG became the Numerical Algorithms Group. In 1976 NAG became a non-profit company limited by guarantee and in 1980 the NAG Inc. office opened in Downers Grove near Chicago. Throughout the development of NAG Wilkinson played an active role in contributing to and commenting on the linear algebra chapters of the library. In 1984 Wilkinson gave an invited lecture at NAG's AGM [Wilkinson, 1985].

center; l n n n n n.

Mark	1	2	4	11	12
No. of Routines	11	75	99	142	239

F Chapter of NAG Fortran Library

We now move over the Atlantic to look at some important software developments in the USA.

Prior to the actual publication of the *Handbook*, V. Klema and others at Argonne National Laboratory had begun translating many of the Algol procedures into Fortran. Their work became the basis for the subsequent development of EISPACK [Smith et al. 1976], a collection of Fortran subroutines that compute the eigenvalues and eigenvectors of nine classes of matrices.

The first release of EISPACK in 1974 was available in five versions: IBM Systems 370 and 360, CDC 6600 and 7600, Univac 1108, Honeywell 635, and PDP-10. Since that time, EISPACK has been extended twice. In Section 6.1, we discuss these extensions briefly, and review two other software packages of linear algebra algorithms. Wilkinson also actively supported the EISPACK project and was a frequent visitor at Argonne.

In the next section we look at the linear algebraic approach to this software and other subsequent software projects.

5. Decompositional Approach

Software for linear algebra is based on the *decompositional* approach to numerical linear algebra. To understand this approach better, let us consider the problem of solving the linear system

$$Ax = b, \tag{2.1}$$

where A is a nonsingular matrix of order n. In older textbooks this problem is treated by writing (2.1) as a system of scalar equations and eliminating unknowns in such a way that the system becomes upper triangular (Gaussian elimination) or even diagonal (Gauss-Jordan elimination). This approach has the advantage that it is easy to understand and that it leads to pretty computational tableaux suitable for hand calculation. However, it has the drawback that the level of detail obscures the very broad applicability of the method.

In contrast, the decompositional approach begins with the observation that it is possible to factor A in the form

$$A = LU, \tag{2.2}$$

where L is a lower triangular matrix with ones on its diagonal and U is upper triangular. * The solution to (2.1) can then be written in the form

$$x = A^{-1}b = U^{-1}L^{-1}b = U^{-1}y,$$

where $y = L^{-1}b$. This suggests the following algorithm for solving (2.1).

1: *Factor* A *in the form* (2.2);

2: *Solve the system* $Ly = b$; (2.3)

3: *Solve the system* $Ux = y$;

Since both L and U are triangular, steps 2 and 3 are easily done.

---

* This is not strictly true. It may be necessary to permute the rows of A (a process called pivoting) in order to ensure the existence of the factorization (2.2). In finite precision arithmetic, pivoting *must*, in general, be incorporated to ensure numerical stability.

The approach to matrix computations through decompositions has turned out to be very fruitful. First, by dividing the computation into two stages (the computation of a decomposition and the use of the decomposition to solve the problem at hand), factorization is necessary only once, representing a potentially large saving.

Second, the approach suggests ways of avoiding the explicit computation of matrix inverses or generalized inverses, which is always a computationally expensive and numerically risky procedure.

Third, the decompositional approach introduces flexibility into matrix computations. There are many decompositions, and a knowledgeable person can select the one best suited to a given applications.

Fourth, if one is given a decomposition of a matrix A and a simple change is made in A (e.g. the alteration of a row or column), one can frequently compute the decomposition of the altered matrix from the original decomposition at far less cost than the *ab initio* computation of the decomposition. This general idea of *updating* a decomposition has been an important theme during the past decade of numerical linear algebra.

Finally, the decompositional approach provides theoretical simplification and unification. This is true both inside and outside of numerical analysis. For example, the realization that the Crout, Doolittle, and square root methods all compute LU decompositions enables one to recognize that they are all variants of Gaussian elimination. Outside of numerical analysis, the spectral decomposition has long been used by statisticians as a canonical form for multivariate models.

In the next section we discuss three particular software packages for linear algebra algorithms that exploit this approach to software development.

6. Software Packages

6.1 EISPACK

EISPACK includes 13 drivers, each intended for matrices of various forms. Twelve of the drivers provide two options: compute all eigenvalues, or compute all eigenvalues and eigenvectors. One of the drivers provides for all the eigenvalues and some of the eigenvectors for a symmetric matrix. Seven of the drivers are for the standard eigenvalue problem involving a single real matrix; two of the drivers solve the standard eigenvalue problem for complex matrices; and four of the drivers solve the generalized eigenvalue problem involving two real matrices. These driver subroutines provide easy access to many of EISPACK's capabilities. The user whose problems do not make heavy demands on computer time or storage need not be concerned with any further details of EISPACK organization.

In addition to the drivers, however, there are 58 subroutines in EISPACK. The modular organization greatly reduces the amount of both source and object code that must be handled. It also provides opportunities for using EISPACK facilities in computations not envisioned during the original development. But it means that the user who desires to access these facilities is faced with a formidable list of subroutines.

EISPACK has been enhanced twice since its initial release [Garbow et al. 1977; Dongarra and Moler 1984]. The first revision, in 1976, offered the capability of handling generalized eigenvalue problems

directly. The current version, EISPACK 3, eliminates the need for machine-specific constants and reduces the probability of underflow/overflow difficulties. The basic design, however, remains the same.

6.2 LINPACK

The success of EISPACK in 1974 motivated the development of a second package of high-quality software for linear algebra problems. This package, called LINPACK, was designed for the solution of linear equations and linear least-squares problems [Dongarra et al. 1979].

When LINPACK was designed, one of its most distinctive features was efficiency. LINPACK achieves this efficiency from two sources: the column orientation of the algorithm and the use of the Basic Linear Algebra Subprograms (Level 1 BLAS). When LINPACK was designed in the late 1970's the state of the art in scientific computers were the pipelined scalar processors, such as the CDC 7600 and the IBM 360/195. In light of todays advanced computers which use memory hierarchy, vector operations and parallel processing the situation has changed somewhat.

While advanced-computer architectures have generally speeded up performance, many modern machines have also presented a new problem in coding matrix routines. This problem centers on the feature of hierarchical memory organization. Typically, a hierarchical memory structure involves a sequence of computer memories, ranging from a small, but very fast memory at the bottom to a capacious, 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. In order 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.* LINPACK uses column-oriented algorithms to preserve locality of reference. That is, the LINPACK codes always reference arrays down columns, not across rows. This approach works because FORTRAN stores arrays in column order. Thus, as one proceeds down a column of an array, the memory references proceed sequentially in memory. On the other hand, as one proceeds across a row the memory references jump across memory, the length of the jump being proportional to the length of a column. The effects of column orientation are dramatic: on scalar systems with virtual or cache memories, the LINPACK codes will significantly outperform codes that are not column oriented. However, the algorithms in LINPACK did not go far enough with the locality of reference. As we shall discuss later, expressing the operations or algorithm in terms of vector operations does not achieve the locality of reference required to attain full reuse of data and ultimate performance rates on today's large scale scientific computers.

On scalar computers LINPACK also gains efficiency from the use of the Level 1 BLAS when large matrices (typically between $n = 25$ and $n = 100$) are involved. The BLAS improve the efficiency particularly when the programs are run on nonoptimizing compilers. This is because doubly subscripted array references in the inner loop of the algorithm are replaced by singly subscripted array references in the appropriate BLAS. The effect can be seen in matrices of quite small order, and for large orders the savings are significant. Finally, improved efficiency can be achieved by coding a set of BLAS to take

advantage of the special features of the computers on which LINPACK is being run—either by producing machine language versions or taking advantage of features such as vector operations. (For further information on the BLAS, see Section 8.)

In order to improve the performance of algorithms implemented on high-performance computers, we must consider not only the total number of memory references, but also the pattern of memory references. We would like our algorithms to observe the principle of locality of reference, so that the data can be effectively utilized. A set of tools that aid in understanding a program's locality of reference have been designed by Brewer, Dongarra, and Sorensen [1988]. The tools help visualize the individual memory references that were made to the one- and two-dimensional arrays in a user's program. The goal of this work is to assist in formulating correct algorithms for high-performance computers and to aid as much as possible the process of translating an algorithm into an efficient implementation on a specific machine.

6.3 MATLAB

MATLAB is an interactive system whose basic data element is a matrix [Moler, *et al.*, 1986]. The system provides easy access to matrix software developed by the LINPACK and EISPACK projects. This allows a user to solve many numerical problems in a fraction of the time it would take to write a program in a language like Fortran or C. Furthermore, problem solutions are expressed in MATLAB almost exactly as they are written mathematically.

MATLAB has evolved over more than half a decade with suggestions and contributions from many users. In university environments it has become the standard instrumental tool used in introductory courses in applied linear algebra, as well as advanced courses in other areas. In nonacademic settings, MATLAB is used for research and for solving practical engineering and mathematical problems. Typical uses include general-purpose numerical computation, algorithm prototyping, and the solution of the special-purpose problems with matrix formulations that arise in disciplines such as automatic control theory, statistics, and digital signal processing.

Today MATLAB is distributed by MathWorks and available for a range of computer systems. They have produced a version called PRO-MATLAB which has been completely rewritten in C and has integrated graphics capability, programmable macros, IEEE arithmetic, a fast interpreter, and many new analytical commands. The initial commercial version of MATLAB was done for the PC, and called PC-MATLAB.

7. Architectural Features

The development of vector and parallel computers in the late 1970s led to a critical review of mathematical software for the solution of linear algebra equations. Many of the sequential algorithms used satisfactorily on traditional machines fail to exploit the architecture of advanced computers. In this section we review the various features of these more advanced systems and discuss how the architecture affects the potential performance of linear algebra algorithms. In Section 8 we consider recent techniques devised for utilizing advanced architectures more fully, especially the design of the BLAS. In Section 9

we discuss a new proposal, LAPACK, which is intended to most fully exploit advanced computers. Finally, in Section 10, we address the challenge facing designers of mathematical software in view of the development of massively parallel computer systems.

We review some of the basic features of traditional and more advanced computers. This review is not intended to be a complete discussion of the architecture of any particular machine. Rather, our focus is on certain features that are especially relevant to the implementation of linear algebra algorithms.

7.1 Cache

The idea of introducing a high-speed buffer memory (or cache) between the slow main memory and the arithmetic registers goes back at least to the ATLAS computer [Hockney and Jesshope 1981, p 14]. The technique was adopted by IBM for both the System 360 and the System 370 computers. In the IBM System 360 Model 85, for example, the cache (32,768 words of 162-ns semiconductor memory in the 360/85) held the most recently used data blocks of 64 bytes. If the data required by an instruction were not in the cache, the block containing it was obtained from the slower main memory (4 Mbytes of 756-ns core storage, divided into 16 different banks) and replaced the least-frequently-used block in the cache. Cache memory is still used in many large-scale calculations in which memory references tend to concentrate around limited regions of the address space. In such cases, most references will be to data in the fast cache memory, and the performance of the slow memory will be effectively that of the faster cache memory.

7.2 Pipelining

Pipeline concurrency is the name given to a system of multiple functional units, each of which is responsible for partial interpretation and execution of the instruction stream. A pipeline processor has several partially completed instructions in process at one time. Each processor stage operates on a specific part of the instruction (e.g., instruction fetch, effective address calculation, operand fetch, execution of operation specified by the instruction, and results storing).

Pipelining is analogous to an industrial assembly line where a product moves through a sequence of stations. Each station carries out one step in the manufacturing process, and each of the stations works simultaneously on different units in different phases of completion.

The goal of pipelined functional units is clearly performance. After some initial startup time, which depends on the number of stages (called the length of the pipeline, or pipe length), the functional unit can turn out one result per clock period as long as a new pair of operands is supplied to the first stage every clock period. Thus, the rate is independent of the length of the pipeline and depends only on the rate at which operands are fed into the pipeline. Therefore, if two vectors of length k are to be added, and if the floating-point adder requires 3 clock periods to complete, it would take $3 + k$ clock periods to add the two vectors together, as opposed to $3 * k$ clock periods in a conventional computer [Dongarra, Gustavson, and Karp 1984].

Pipelining was used by a number of machines in the 1960s, including the CDC 7600 and the IBM System 360/195. Later CDC introduced the STAR 100 (subsequently called the CYBER 200 series),

which also used pipelining to gain a speedup in instruction execution.

7.3 Vector Instructions

One of the most obvious concepts for achieving high performance is the use of vector instructions. By means of a single instruction, all elementwise operations that make up the total vector operation are carried out. The instructions are performed in vector registers. The machine may have k such elements in a vector register in addition to having a conventional set of registers for scalar operations. A typical sequence of instructions would be as follows:

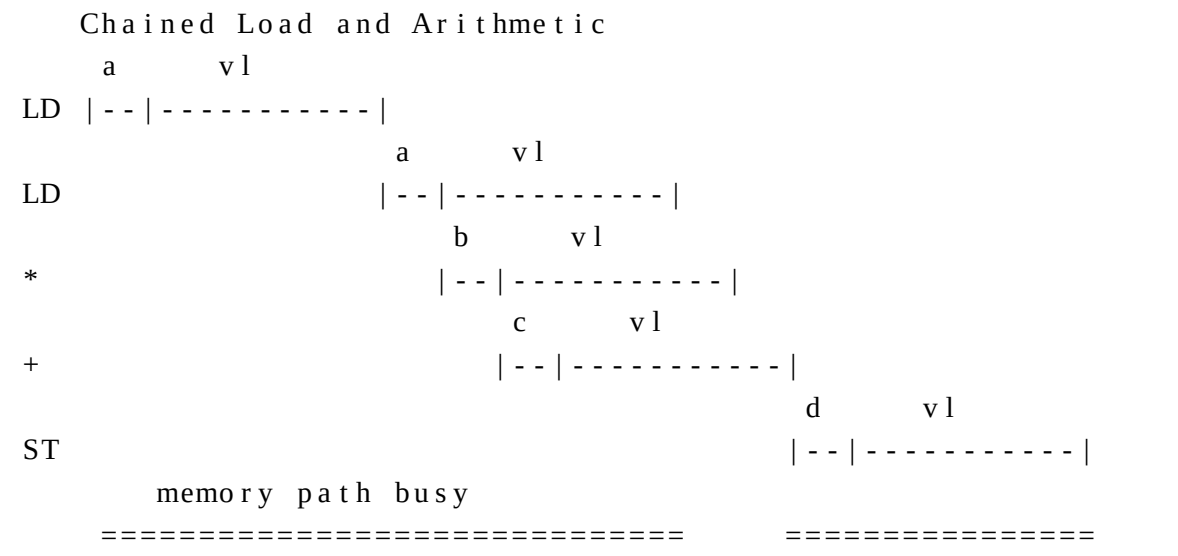
center; l. Load a scalar register from memory Load a vector register from memory Perform a scalar-vector multiplication Load a vector register from memory Perform a vector-vector addition Store the results in memory.

These six instructions would correspond to perhaps $6k + 1$ instructions on a conventional computer, where k instructions are necessary for loop branching. Clearly, then, the time to interpret the instructions has been reduced by almost a factor of k, resulting in a significant savings in overhead.

7.4 Chaining

Another feature that is used to achieve high rates of execution is chaining. Chaining is a technique whereby the output register of one vector instruction is the same as one of the input registers for the next vector instruction. If the instructions use separate functional units, the hardware will start the second vector operation during the clock period when the first result from the first operation is just leaving its functional unit. A copy of the result is forwarded directly to the second functional unit and the first execution of the second vector is started. The net result is that the execution of both vector operations takes only the second functional unit startup time longer than the first vector operation. The effect is that of having a new instruction which performs the same operation as that of the two functional units that have been chained together. On the CRAY, in addition to the arithmetic operations, vector loads from memory to vector registers can be chained with other arithmetic operations.

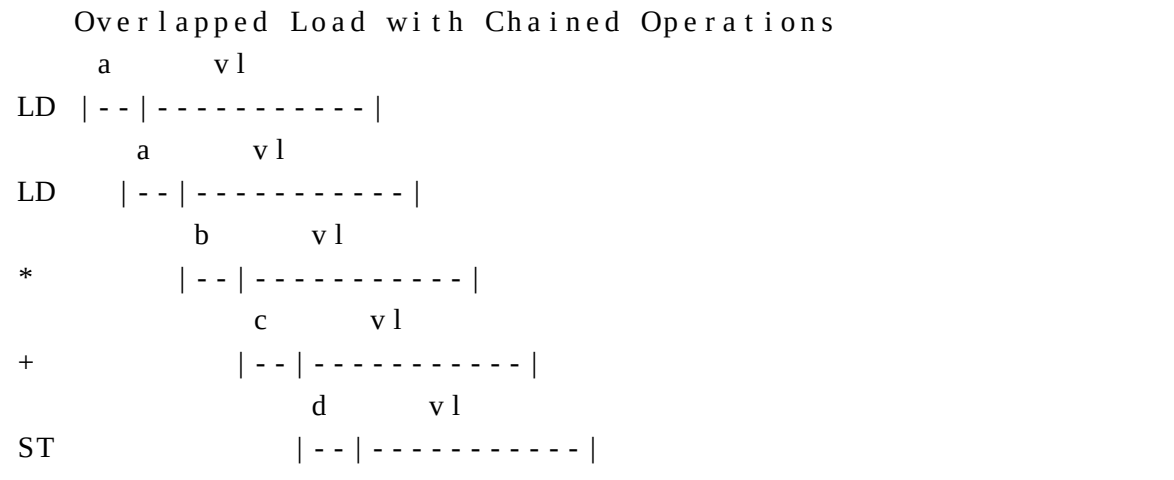
For example, let us consider a case involving a scalar-vector multiplication, followed by a vector-vector addition, where the addition operation depends on the results of the multiplication. Without chaining, but with pipelined functional units, the operation would take $a + k + m + k$ clock periods, where a is the time to start the vector addition (length of the vector addition pipeline) and m is the time to start a vector multiplication (length of the vector multiplication pipeline). With chaining, as soon as a result is produced from the adder, it is fed directly into the multiplication unit, so the total time is $a + m + k$. We may represent this process graphically as follows:



a startup time for memory load operations
b startup time for floating point addition operations
c startup time for floating point multiplication operations
d startup time for memory store operations
= memory busy

6.5 Overlapping

It is also possible to overlap operations if the two operations are independent. If a vector addition and an independent vector multiplication are to be processed, the resulting timing graph might look like the following:



To describe the time to complete a vector operation, we use the concept of a chime [Fong and Jordan 1977]. A *chime* (for *ch*aining t*ime*) is a measure of the time needed to complete a sequence of vector operations. To compute the number of chimes necessary for a sequence of operations, one divides the total time to complete the operations by the vector length. Overhead of startup and scalar work are usually ignored in counting chimes, and only the integer part is reported. For example, in the graph for unchained operations above there are two chimes, whereas in the graph for the chained operation there is one chime.

As Fong and Jordan [1977] have pointed out, there are three performance levels for algorithms on the CRAY. The two obvious ones are scalar and vector performance. Scalar performance is achieved when operations are carried out on scalar quantities, with no use of the vector functional units. Vector performance is achieved when vectors are loaded from memory into registers, operations such as multiplication or addition are performed, and the results are stored into memory. The third performance level is called supervector. This level is achieved when vectors are retained in registers, operations are performed using chaining, and the results are stored in registers. Thereby using the memory hierarchy of the machine architecture to the fullest.

Dramatic improvements in rates of execution are realized in going from scalar to vector and from vector to supervector speeds. We show below a graph of the execution rate in MFLOPS (million floating point operations per second) for *LU* decomposition of a matrix of order *n* as performed on the CRAY-1.

7.6 Loop Unrolling

When data references and the memory hierarchy are used efficiently, the hardware is being driven at close to its highest potential. Theoretically, this situation can be shown by the following example, which adds the product of a matrix and a vector to another vector:

```
center; l l.        SUBROUTINE SMXPY (N1,Y,N2,LDM,X,M)        REAL Y(*), X(*),
M(LDM,*)        DO 20 J = 1, N2              DO 10 I = 1, N1              Y(I) = Y(I) +
X(J)*M(I,J) 10        CONTINUE 20      CONTINUE         RETURN        END
```

The innermost loop is a SAXPY (adding a multiple of one vector to another) and would be detected by a good vectorizing compiler. Thus, the CRAY Fortran compiler generates vector code of the general form

center; l. Load vector Y *Load scalar* X(J) *Load vector* M(*,J) *Multiply scalar* X(J) *times vector* M(*,J) *Add result to vector* Y *Store result in* Y

Note that there are *three* vector memory references for each *two* vector floating-point operations. Since there is only one path to and from memory and the memory bandwidth is 80 million words per second, the rate of execution cannot exceed ~53 1/3 MFLOPS (less than 50 MFLOPS when vector startup time is taken into account) — *vector performance*.

Thus to attain supervector performance, it is necessary to expand the scope of the vectorizing process to more than just simple vector operations. In this case, a closer inspection reveals that the vector Y is stored and then reloaded in successive SAXPY's. If instead one accumulates Y in a vector register (up to 64 words at a time) until all of the columns of M have been processed, it is possible to avoid two of the three vector memory references in the innermost loop. The maximum rate of execution is then 160 MFLOPS (~148 MFLOPS when vector startup time is taken into account) — *supervector performance*.

Unfortunately, the CRAY CFT compiler does not detect the fact that the result can be accumulated in a register (and not stored between successive vector operations). Thus, the rate of execution is limited to vector speeds. In fact, all of todays compilers for vector machines follow the same action of transferring the vector to memory and immediately reload the same vector to a register.

But if the outer loop is unrolled [Dongarra and Eisenstat, 1986] in this case to a depth of four, and parentheses are inserted to force the arithmetic operations to be performed in the most efficient order, then the innermost loop becomes

```
l l.        DO 10 I = 1, N1        Y(I) = ((((Y(I)) + X(J-3)*M(I,J-3)) + X(J-2)*M(I,J-2))
     $                + X(J-1)*M(I,J-1)) + X(J)  *M(I,J) 10        CONTINUE
```

Now the code generated by CFT has *six* vector memory references for each *eight* vector floating-point operations. Thus the maximum rate of execution is ~106 2/3 MFLOPS (~100 MFLOPS when vector startup time is taken into account) and the actual rate is ~77 MFLOPS — *supervector performance from Fortran*.

With this approach a collection of procedures from linear algebra can be developed. The key idea is to use the kernel — SGEMV (add a vector times a matrix to another vector) to do the bulk of the work.

Since this kernel can be unrolled to give supervector performance, the procedures themselves are capable of supervector performance.

Many processes that involve elementary transformations can be described in these terms, e.g., matrix multiplication, Cholesky decomposition, and LU factorization (see [Dongarra and Eisenstat 1986]). However, the formulation is often not the natural one, which may be based on outer-products of vectors or accumulating variable-length vectors, neither of which can be super-vectorized in Fortran.

Tables 1-3 summarize the results obtained for these procedures on a CRAY 1-S (as well as on the CRAY 1-M and CRAY X-MP) when the subroutines SMXPY and SXMPY were unrolled to the specified depth. All runs used the CFT 1.11 Fortran compiler.

Table 1: 300 × 300 Matrix Multiplication

box center; c | c s s c | c s s c | c | c | c c | c | c | c n | n | n | n. Unrolled MFLOPS _

Depth	CRAY 1-M	CRAY 1-S	CRAY X-MP	_	_	_	_
1	39	40	106				
2	60	53	151	4	83	72	161
8	101	86	170	16	111	96	177

Table 2: 300 × 300 Cholesky Decomposition

box center; c | c s s c | c s s c | c | c | c c | c | c | c n | n | n | n. Unrolled MFLOPS _

Depth	CRAY 1-M	CRAY 1-S	CRAY X-MP	_	_	_	_
1	31	33	68				
2	48	45	99	4	67	60	118
8	81	70	131	16	86	78	139

Table 3a: 300 × 300 LU Decomposition with Pivoting

box center; c | c s s c | c s s c | c | c | c c | c | c | c n | n | n | n. Unrolled MFLOPS _

Depth	CRAY 1-M	CRAY 1-S	CRAY X-MP	_	_	_	_
1	28	29	56				
2	42	39	78	4	56	52	93
8	66	60	103	16	69	66	108

Table 3b: 300 × 300 LU Decomposition with Pivoting
(Using an Assembler Language Implementation of ISAMAX)

box center; c | c s s c | c s s c | c | c | c c | c | c | c n | n | n | n. Unrolled MFLOPS _

Depth	CRAY 1-M	CRAY 1-S	CRAY X-MP	_	_	_	_
1	30	32	62				
2	46	43	96	4	64	59	117
8	78	68	129	16	83	76	136

By contrast, 30 MFLOPS is often cited as a good rate for Fortran on the CRAY 1-S and 100 MFLOPS as a good rate for CAL (Cray Assembler Language) (e.g., Fong and Jordan [1977] report 107 MFLOPS for an assembler language implementation of LU decomposition with pivoting).

Similar techniques have been used by Dongarra, Kaufman, and Hammarling [1985] to modify many of the standard algorithms used in computing eigenvalues and eigenvectors of matrices. Not only do the

techniques dramatically increase the performance without resorting to assembly language, but they are beneficial in a variety of architectural settings.

7.7 Summary of Techniques

In summary, then, vector machines rely on a number of techniques to enhance their performance over conventional computers:

- vector instructions to reduce the number of instructions interpreted,
- pipelining to utilize a functional unit fully and to deliver one result per cycle,
- chaining to overlap functional unit execution,
- overlapping to execute more than one independent vector instruction concurrently, and
- loop unrolling to force arithmetic operations to be performed efficiently.

Programs that use these features properly will fully exploit the potential of the vector machine.

8. Basic Linear Algebra Subprograms

One way of achieving efficiency in the solution of linear algebra problems is through the use of the Basic Linear Algebra Subprograms. In 1973 Hanson, Krogh, and Lawson [1973] described the advantages of adopting a set of basic routines for problems in linear algebra. The BLAS, as they are now commonly called [Lawson et al. 1979], have been very successful and have been used in a wide range of software, including LINPACK and many of the algorithms published by the *ACM Transactions on Mathematical Software.* They are an aid to clarity, portability, modularity, and maintenance of software, and they have become a *de facto* standard for the elementary vector operations. The BLAS are fully described in [Lawson, Hanson, Kincaid, and Krogh 1979] and by Dodson and Lewis [1985]. Here we review their purpose and their advantages. We also discuss two recent enhancements to the BLAS.

8.1 Level 1 BLAS

The original set of BLAS perform low-level operations such as dot-product and the adding of the multiple of one vector to another. The BLAS promote efficiency by identifying frequently occurring operations of linear algebra that can be optimized on various computers, perhaps by coding them in assembly language or otherwise taking advantage of special machine properties. Use of these optimized operations can yield dramatic reductions in computation time on some computers. The BLAS also offer several other benefits:

- Robustness of linear algebra computations is enhanced by the BLAS since they take into consideration

algorithmic and implementation subtleties that are likely to be ignored in a typical application programming environment.

• Program portability is improved through standardization of computational kernels without giving up efficiency, since optimized versions of the BLAS can be used on those computers for which they exist, yet compatible standard Fortran is available for use elsewhere.

• Program readability is enhanced. The BLAS are a design tool; that is, they are a conceptual aid in coding, allowing one to visualize mathematical operations rather than the particular detailed coding required to implement the operations. By associating widely recognized mnemonic names with mathematical operations, the BLAS improve the self-documenting quality of code.

8.2 Level 2 BLAS

Special versions of the BLAS, in some cases machine code versions, have been implemented on a number of computers, thus improving the efficiency of the BLAS. However, with some of the modern machine architectures, the use of the BLAS is not the best way to improve the efficiency of higher level codes. On vector machines, for example, one needs to optimize at least at the level of matrix-vector operations in order to approach the potential efficiency of the machine; the use of the BLAS inhibits this optimization because they hide the matrix-vector nature of the operations from the compiler.

Thus, an additional set of BLAS, called the Level 2 BLAS, was designed for a small set of matrix-vector operations that occur frequently in the implementation of many of the most common algorithms in linear algebra [Dongarra et al. 1986].

The Level 2 BLAS involve $O(mn)$ scalar operations where m and n are the dimensions of the matrix involved.

The following three types of basic operation are performed by the Level 2 BLAS:

1. Matrix-vector products of the form

$$y \leftarrow \alpha Ax + \beta y, \quad y \leftarrow \alpha A^T x + \beta y, \quad \text{and} \quad y \leftarrow \alpha \bar{A}^T x + \beta y$$

where α and β are scalars, x and y are vectors and A is a matrix, and

$$x \leftarrow Tx, \quad x \leftarrow T^T x, \quad \text{and} \quad x \leftarrow \bar{T}^T x,$$

where x is a vector and T is an upper or lower triangular matrix.

2. Rank-one and rank-two updates of the form

$$A \leftarrow \alpha x y^T + A, \quad A \leftarrow \alpha x \bar{y}^T + A, \quad H \leftarrow \alpha x \bar{x}^T + H, \quad \text{and } H \leftarrow \alpha x \bar{y}^T + \bar{\alpha} y \bar{x}^T + H,$$

where H is a Hermitian matrix.

3. Solution of triangular equations of the form

$$x \leftarrow T^{-1}x,\ x \leftarrow T^{-T}x,\ \text{ and }\ x \leftarrow \bar{T}^{-T}x,$$

where T is a non-singular upper or lower triangular matrix.

Where appropriate, the operations are applied to general, general band, Hermitian, Hermitian band, triangular, and triangular band matrices in both real and complex arithmetic, and in single and double precision.

8.3 Level 3 BLAS

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.

The Level 3 BLAS proposed by Dongarra et al. [1987] are targeted at the matrix-matrix operations required for these purposes. The routines proposed are derived in a fairly obvious manner from some of the Level 2 BLAS, by replacing the vectors x and y with matrices B and C. The advantage in keeping the design of the software as consistent as possible with that of the Level 2 BLAS is that it will be easier for users to remember the calling sequences and parameter conventions.

In real arithmetic the operations proposed for the Level 3 BLAS have the following forms:

a) Matrix-matrix products

$$C \leftarrow \alpha AB + \beta C$$

$$C \leftarrow \alpha A^T B + \beta C$$

$$C \leftarrow \alpha AB^T + \beta C$$

$$C \leftarrow \alpha A^T B^T + \beta C$$

These operations are more accurately described as matrix-matrix multiply-and-add operations; they include rank-k updates of a general matrix.

b) Rank-k updates of a symmetric matrix:

$$C \leftarrow \alpha AA^T + \beta C$$

$$C \leftarrow \alpha A^T A + \beta C$$

$$C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$$

c) Multiplying a matrix by a triangular matrix:

$$B \leftarrow \alpha TB$$

$$B \leftarrow \alpha T^T B$$

$$B \leftarrow \alpha BT$$

$$B \leftarrow \alpha BT^T$$

d) Solving triangular systems of equations with multiple right-hand sides:

$$B \leftarrow \alpha T^{-1} B$$

$$B \leftarrow \alpha T^{-T} B$$

$$B \leftarrow \alpha BT^{-1}$$

$$B \leftarrow \alpha BT^{-T}$$

Here α and β are scalars, A, B and C are rectangular matrices (in some cases square and symmetric), and T is an upper or lower triangular matrix (and non-singular in (d)).

Analogous operations are proposed in complex arithmetic: conjugate transposition is specified instead of simple transposition and in (b) C is Hermitian and α and β are real.
The results of using the different levels of BLAS on the Alliant FX/8, IBM 3090 with Vector Facility, and the CRAY-2 are shown in the figures below.

center; l c l n. Alliant FX/8 (8 Processors) MFLOPS _

Peak Performance 94 LINPACK Benchmark 7.6 Level 1 BLAS ($y \leftarrow y + \alpha x$) 14 Level 2 BLAS ($y \leftarrow \beta y + \alpha Ax$) 26 Level 3 BLAS ($C \leftarrow \beta C + \alpha AB$) 43

center; l c l n. IBM 3090/VF (1 Processor) MFLOPS _

Peak Performance 108 LINPACK Benchmark 12 Level 1 BLAS ($y \leftarrow y + \alpha x$) 26 Level 2 BLAS ($y \leftarrow \beta y + \alpha Ax$) 60 Level 3 BLAS ($C \leftarrow \beta C + \alpha AB$) 80

center; l c l n. CRAY-2 (1 processor) MFLOPS _

Peak Performance 488 LINPACK Benchmark 15 Level 1 BLAS ($y \leftarrow y + \alpha x$) 121 Level 2 BLAS ($y \leftarrow \beta y + \alpha Ax$) 350 Level 3 BLAS ($C \leftarrow \beta C + \alpha AB$) 437

The following figure illustrates the advantage of the Level 3 BLAS:

center; l|l l l. BLAS Mem Ref Ops Ratio Ref:Ops $n = m = k$ _

Level 1 SAXPY $3n$ $2n$ 3 : 2 $y \leftarrow y + \alpha x$

Level 2 SGEMV $mn + n + 2m$ $2mn$ 1 : 2 $y \leftarrow \beta y + \alpha Ax$

Level 3 SGEMM $2mn + mk + kn$ $2mnk$ 2 : n $C \leftarrow \beta C + \alpha AB$

8.4 Matrix-Matrix Level Algorithms

As computer architectures become more sophisticated in their organization we are required to supply an even higher level of granularity in our algorithms to take full advantage of the highest levels of performance. A primary source of performance problems on todays advanced scientific computers is the

result of handling of data traffic in the memory hierarchy of the computer. The next level of modularity we naturally focus on is at the matrix-matrix level. The advantage here is obvious, $O(n^2)$ data to undergo $O(n^3)$ operations.

Defining the methods in terms of these modules requires us to express the algorithms in terms of applying groups of transformations to a submatrix during a step. We will sometimes perform slightly more floating point operations in these formulation, but the performance increase gained by the matrix-matrix operations will far out-weigh the additional arithmetic. As we have done earlier, we will describe formulations for Gaussian elimination terms of matrix-matrix operations.

The algorithm can be viewed at the k^{th} stage as decomposing the matrix in the form:

$$\begin{pmatrix} L_{11} & & \\ L_{21} & I & \\ L_{31} & & I \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ & A_{22}^{(k)} & A_{23}^{(k)} \\ & A_{32}^{(k)} & A_{33}^{(k)} \end{pmatrix}$$

At this stage we have decomposed part of the matrix and produced pieces of the final L and U factors L_{11}, L_{21}, L_{31}, U_{11}, U_{12}, and U_{13}. These parts will undergo no further changes. The k stage of the algorithm will modify the submatrix

$$\begin{pmatrix} A_{22}^{(k)} & A_{23}^{(k)} \\ A_{32}^{(k)} & A_{33}^{(k)} \end{pmatrix}$$

and produce:

$$\begin{pmatrix} L_{22}/U_{22} & L_{32} \\ U_{23} & \hat{A}_{33}^{(k)} \end{pmatrix}$$

The steps are as follows:

Step 1 (Construct parts of L and U)

$$P\begin{pmatrix} A_{22}^{(k)} \\ A_{32}^{(k)} \end{pmatrix} = \begin{pmatrix} L_{22} \\ L_{32} \end{pmatrix}(U_{22})$$

This step performs an LU factorization on the rectangular matrix $\begin{pmatrix} A_{22}^{(k)} \\ A_{32}^{(k)} \end{pmatrix}$. Partial pivoting is performed and each interchange of rows is carried out across the entire rectangular matrix and a record made to apply to the rest of the matrix in the next step.

Step 2 (Interchange rows in the remaining blocks)

$$\begin{pmatrix} A_{23}^{(k)} \\ A_{33}^{(k)} \end{pmatrix} \leftarrow P \begin{pmatrix} A_{23}^{(k)} \\ A_{33}^{(k)} \end{pmatrix}$$

The pivot information from Step 1 is applied to the remainder of the matrix as listed above.

Step 3 (Apply L_{22}^{-1})

$$U_{23} \equiv A_{23}^{(k)} \leftarrow L_{22}^{-1} A_{23}^{(k)}$$

A triangular solve is performed to update part of the matrix with the transformations.

Step 4 (Update remaining submatrix)

$$\hat{A}_{33}^{(k)} \leftarrow A_{33}^{(k)} - L_{32} U_{23}$$

A matrix-matrix multiply is performed to update the last block of the matrix with the transformations.

The value of such a reorganization is particularly important when the memory bandwidth of a machine is not well matched to the speed of the processors. An example of such a machine is the Alliant FX/8 computer. This computer is a globally shared memory machine with eight processors called CE's. Each of these CE's has vector capability. The memory path is from global memory through a bus to a cache which then feeds the CE's. When data comes from this cache the vector processors can produce results at a peak rate of eight floating point (64-bit) results every 170 nanoseconds. This gives a peak megaflop rate of around 45 megaflops. However, when there are memory references to data not resident in cache, the computation rate is degraded. The cache is designed to be effective when many references are made to the same area of an array. During an LU Decomposition of a large matrix the advantage of such a cache is negated because the memory references sweep through the array over and over again. Even though certain columns of the matrix have been referenced previously, they are unlikely to remain in the cache during a full step of the decomposition. This difficulty may be overcome through efficient use of vector registers.

In order to make efficient use of the vector registers one may choose the column dimension of $A_{22}^{(k)}$ to conform to the number of vector registers available. On the Alliant there are eight vector registers and seven of these may be used to hold columns of L_{32}. The remaining register must be used to accumulate results. There are three steps in the above algorithm that may be parallelized. In Step 2 we partition

$$A_{23}^{(k)} = (M_{21} , M_{22} , \cdots , M_{2p})$$

and

$$A_{33}^{(k)} = (M_{31} , M_{32} , \cdots , M_{3p})$$

with p chosen to conform to the number of processors. Since the pivoting sequence has been recorded in P it may be applied to the matrices

$$\begin{pmatrix} M_{2j} \\ M_{3j} \end{pmatrix} \leftarrow P \begin{pmatrix} M_{2j} \\ M_{3j} \end{pmatrix}, \quad j = 1, \ldots, p$$

independently. Then the products

$$M_{2j} \leftarrow L_{22}^{-1} M_{2j} \ , \ j = 1, \ldots, p$$

are computed independently in place to form blocks of U_{23} . Finally, in Step 4 we may form the matrix matrix products

$$M_{3j} \leftarrow L_{23} M_{2j} \quad j = 1, \ldots, p$$

which form the block columns of $A_{33}^{(k)}$. This step makes full use of the vector registers since the columns of L_{23} may be held in these registers and re-used repeatedly while the columns of M_{2j} are multiplied.

The following table illustrates the effectiveness of this technique. In the table we compare the results of the matrix-vector and matrix-matrix techniques. In both cases the modules have been coded in assembly language to assure full use of the vector registers. We also show results from the Fortran equivalent matrix-vector formulation.

LU Decomposition with partial pivoting

Alliant FX/8 (8 Processors)

center; c|c s s s s s c|c s s s s s c|c c c c c c|n n n n n n.

MFLOPS Order	*Implementation*	*100*	*200*	*300*	*400*	*500*	*600*
Fortran	*Matrix-vector*	*2.8*	*4.7*	*5.4*	*5.7*	*5.7*	*5.8*
Assembler	*Matrix-vector*	*8.2*	*9.8*	*11.2*	*10.7*	*11.6*	*11.3*
Assembler	*Matrix-matrix*	*6.6*	*11.8*	*15.3*	*17.2*	*18.7*	*19.8*

This table shows that the matrix-matrix modules can be very effective. Moreover, while this technique appears to be tuned to a particular situation, it is equally effective in a non-cache situation as long as there are a significant number of vector registers available. Finally, blocks of cache may be used in place of vector registers when (as with the Alliant) the memory access from cache matches the memory access from registers.

9. The LAPACK Project

The development of the higher level BLAS, as well as the large and growing variety of machine architectures available to the scientific programmer, has underscored the need for a new and transportable linear algebra library. To address this need, Demmel et al. [1987] have proposed to design and implement a package called LAPACK, based on the successful EISPACK and LINPACK projects, with the following provisions.

1. Integration of the two sets of algorithms into a unified library, with a systematic design. The new library will provide approximately the same functionality as LINPACK and EISPACK together, namely, solution of systems of simultaneous linear equations, least squares solution of overdetermined systems of equations, and solution of matrix eigenvalue problems (standard and generalized). The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) will also be provided, as will related computations such as reordering of the factorizations and condition numbers (or estimates thereof). Dense and band matrices will be provided for, but not general sparse matrices. In all areas, similar functionality will be provided for real and complex matrices. Some algorithms may be dated, especially where there is duplication or overlap in functionality between the contents of the two packages or if they are no longer thought to be useful, in which case the algorithms will be omitted or replaced.

2. Incorporation of recent algorithmic improvements. Where the state of the art is sufficiently clear, new algorithms will be added. In some cases, the relative merits of competing algorithms will be reexamined in the light of their performance on modern high-performance computers.

3. Restructuring of the algorithms to make as much use as possible of the Basic Linear Algebra Subprograms. The scope for using Level 2 or Level 3 BLAS varies among the different algorithms that are proposed. Many authors have demonstrated the effectiveness of block algorithms on many of our modern computers. Block algorithms generally require an unblocked version of the same algorithm to be available to operate on a single block. Therefore, the development of the software for LAPACK is planned in two phases: (1) develop unblocked versions of the routines, calling the Level 2 BLAS, and (2) develop blocked versions where possible, calling the Level 3 BLAS.

The proposers intend that the new package also serve as a benchmark for supercomputer performance evaluation. A report from the Committee on Supercomputer Performance and Development to the US National Research Council has recommended [1986] using a range of routines from program kernels (like the BLAS) and basic routines (like LINPACK and EISPACK) to large application codes for benchmarks.

10. Parallel Processing, Algorithm Design, and the Future

Parallelism has become a major contributor in increasing performance in recent years, and it is now clear that in the future supercomputers will involve many processors working together in parallel on a single problem. Typically, a parallel processor with globally shared memory must employ some sort of interconnection network so that all processors may access all of the shared memory. There must also be an arbitration mechanism within this memory access scheme to handle cases where two processors attempt to access the same memory location at the same time. These two requirements obviously have the effect of increasing the memory access time over that of a single processor accessing a dedicated memory of the same type. Usually this increase is substantial, especially if the processor and memory in question are at the high end of the performance spectrum. Achieving near peak performance on such computers requires algorithms that minimize data movement and reuse data that has been moved from globally shared

memory to local processor memory.

When vector rather than serial processors are used to construct a parallel computer, a new type of parallelism is encountered. These machines are able to execute independent loop bodies which employ vector instructions. The most powerful computers that exist today are of this type. They include the CRAY X-MP line and the mini-supercomputer Alliant FX/8. A major problem with using such computers efficiently is synchronization overhead. Blocking loops to exploit outer level parallelism, for example, may conflict with vector length.

Finally, a third level of complication is added when parallel-vector machines are interconnected to achieve yet another level of parallelism. This is the case for the CEDAR architecture being developed at the Center for Super Computing Research and Development at the University of Illinois at Urbana. Such a computer is intended to solve large applications problems which naturally split up into loosely coupled parts which may be solved efficiently on the cluster of parallel-vector processors [Berry et al. 1986].

The different approaches to parallelism underscore the need for a more careful selection of algorithms. In addition to restructuring algorithms to take advantage of memory hierarchy, as in the case of linear algorithm algorithms discussed above, a divide and conquer scheme can be used. The divide and conquer paradigm involves breaking a problem up into smaller subproblems that can be treated independently. Frequently, the degree of independence is a measure of the effectiveness of the algorithm since it determines the amount and frequency of communication and synchronization.

A method to find the eigenvalues of a tridiagonal matrix based upon a divide and conquer scheme was suggested by Cuppen. A fundamental tool used to implement this algorithm is a method that was developed by Bunch, Nielsen, and Sorensen for updating the eigensystem of a symmetric matrix after modification by a rank one change. This rank-one updating method was inspired by some earlier work of Golub[19] on modified eigenvalue problems. The basic idea of the new method is to use rank-one modifications to tear out selected off-diagonal elements of the tridiagonal problem in order to introduce a number of independent subproblems of smaller size. The subproblems are solved at the lowest level using the subroutine TQL2 from EISPACK and then results of these problems are successively glued together using the rank-one modification routine developed by Dongarra and Sorensen.

A surprising result of that parallel algorithm is that even when run in serial mode, the divide and conquer approach can be significantly faster than the previously best sequential algorithm on large problems, and is effective on moderate size (order ≥ 30) problems when run in serial mode.

Projects like LAPACK lay the groundwork for much needed research.

References

An Agenda for Improved Evaluation of Supercomputer Performance, US National Research Council, 1986.

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

Hillis W.D, *The Connection Machine,* MIT Press, 1985.

J. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and D. Sorensen, *Prospectus for the Development of a Linear Algebra Library for High-Performance Computers,* Argonne National Laboratory Report MCS-TM-97, September, 1987.

D. Dodson and J. Lewis, *Issues relating to extension of the Basic Linear Algebra Subprograms,* ACM SIGNUM Newsletter 20, 2-18, 1985.

J. J. Dongarra, J. Bunch, C. Moler, and G. Stewart, *LINPACK Users' Guide,* SIAM Pub., Philadelphia, 1979.

J. J. Dongarra, L. Kaufman, and S. Hammarling, *Squeezing the Most out of Eigenvalue Solvers,* Linear Algebra and Its Applications, 77, 113-136, 1986.

J. J. Dongarra and C. B. Moler, *EISPACK—A Package for Solving Matrix Eigenvalue Problems,* in: *Sources and Development of Mathematical Software,* ed. W. R. Cowell, Prentice-Hall, Englewood Cliffs, N.J., 1985.

J. J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson, *An Extended Set of Basic Linear Algebra Subprograms,* ACM Trans. Math. Software, 14,1, p 1-17, March 1988.

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

J. J. Dongarra et al. *A Proposal for a Set of Level 3 Basic Linear Algebra Subprograms,* Argonne National Laboratory Report MCS-TM-88, April.

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

K. Fong and T. L. Jordan, *Some Linear Algebra Algorithms and Their Performance on the CRAY-1,* Los Alamos Scientific Laboratory Report UC-32, June 1977.

L. Fox, *James Hardy Wilkinson (1919-1986),* Biographical Memoirs of Fellows of the Royal Society, Vol. 33, 1987.

B. S. Garbow, J. M. Boyle, J. J. Dongarra, and C. B. Moler, *Matrix Eigensystem Routines—EISPACK*

Guide Extension, in: *Lecture Notes in Computer Science,* Vol. 51, Springer-Verlag, Berlin, 1977.

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

R. W. Hockney and C. R. Jesshope, *Parallel Computers,* J. W. Arrowsmith Ltd., Bristol, Great Britain, 1981.

C. Lawson et al. 1979. *Basic Linear Algebra Subprograms for Fortran usage,* ACM Trans. on Math. Soft. 5, 153-165

C. Moler, J. Little, S. Bangert, and S. Kleiman, *PC-MATLAB for MS-DOS Personal Computers Users' Guide* The MathWorks Inc., Massachusetts, 1986.

B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, *Matrix Eigensystem Routines—EISPACK Guide,* in: *Lecture Notes in Computer Science,* Vol. 6, 2nd ed., Springer-Verlag, Berlin, 1976.

R. V. Southwell, *Relaxation Methods in Engineering Science and Theoretical Physics,* Clarendon Press, Oxford, 1940.

A. M. Turing, *Rounding-off Errors in Matrix Processes,* Quart. J. Mech, 1, pp 287-308, 1948.

J. von Neumann and H. H. Goldstine, *Numerical Inverting of Matrices of High Order,* Bull. Amer. Math. Soc., 53, pp 1021-1099, 1947.

E. T. Whittaker and G. N. Watson, *A Course in Modern Analysis,* The University Press, Cambridge, 1927.

E. T. Whittaker and G. Robinson, *The Calculus of Observations: A Treatise on Numerical Mathematics,* Blackie and Son, London, 1924.

M. V. Wilkes, *The EDSAC,* NPL Report COM 90, National Physical Laboratory, Teddington, Middlesex TW11 0LW, UK, 1977.

M. V. Wilkes, D. J. Wheeler and S. Gill, *The Preparation of Programs for an Electronic Digital Computer,* Addison-Wesley, Cambridge, Mass., 1951.

J. H. Wilkinson, *The Algebraic Eigenvalue Problem,* Oxford University Press, London, 1965.

J. H. Wilkinson and C. Reinsch, eds., *Linear Algebra,* in: *Handbook for Automatic Computation,* Vol. 2, Springer-Verlag, New York, 1971.

J. H. Wilkinson, *Rounding Errors in Algebraic Processes*, Notes on Applied Science No.32, HMSO, London, 1963.

J. H. Wilkinson, *Some Comments from a Numerical Analyst*, J. ACM, 18, pp 137-147, 1971.

J. H. Wilkinson, *Turing's Work at the National Physical Laboratory and the Construction of Pilot ACE, DEUCE, and ACE,* In A History of Computing in the Twentieth Century, Ed. N Metropolis, J. Howlett and Gian-Carlo Rota, Academic Press, New York, pp 101-114, 1980.

J. H. Wilkinson, *The State of the Art in Error Analysis*, NAG Newsletter 2/85, NAG Ltd., Wilkinson House, Jordan Hill Road, Oxford, OX2 8DR, UK, 1985.