Numerical Linear Algebra Algorithms and Software

Jack J. Dongarra
Victor Eijkhout

‡ Department of Computer Science
University of Tennessee
107 Ayres Hall
Knoxville, TN 37996-1301
§ Mathematical Sciences Section
Oak Ridge National Laboratory
P. O. Box 2008, Bldg. 6012
Oak Ridge, TN 37831-6367

Numerical Linear Algebra Algorithms and Software

Jack J. Dongarra
Victor Eijkhout

1 Introduction

The increasing availability of advanced-architecture computers has a significant effect on all spheres of scientific computation, including algorithm research and software development in numerical linear algebra. Linear algebra—in particular, the solution of linear systems of equations—lies at the heart of most calculations in scientific computing. This chapter discusses some of the recent developments in linear algebra designed to exploit these advanced-architecture computers. We discuss two broad classes of algorithms: those for dense, and those for sparse matrices. A matrix is called sparse if it has a substantial number of zero elements, making specialized storage and algorithms necessary.

Much of the work in developing linear algebra software for advanced-architecture computers is motivated by the need to solve large problems on the fastest computers available. In this chapter, we focus on four basic issues: (1) the motivation for the work; (2) the development of standards for use in linear algebra and the building blocks for libraries; (3) aspects of algorithm design and parallel implementation; and (4) future directions for research.

As representative examples of dense matrix routines, we will consider the Cholesky and LU factorizations, and these will be used to highlight the most important factors that must be considered in designing linear algebra software for advanced-architecture computers. We use these factorization routines for illustrative purposes not only because they are relatively simple, but also because of their importance in several scientific and engineering applications that make use of boundary element methods. These applications include electromagnetic scattering and computational fluid dynamics problems, as discussed in more detail in Section 2.1.2.

For the past 15 years or so, there has been a great deal of activity in the area of algorithms and software for solving linear algebra problems. The goal of achieving high performance on codes that are portable across platforms has largely been realized by the identification of linear algebra kernels, the Basic Linear Algebra Subprograms (BLAS). We will discuss the EISPACK, LINPACK, LAPACK, and ScaLAPACK libraries which are expressed in successive levels of the BLAS.

The key insight of our approach to designing linear algebra algorithms for advanced architecture computers is that the frequency with which data are moved between different levels of the memory hierarchy must be minimized in order to attain high performance. Thus, our main algorithmic approach for exploiting both vectorization and parallelism in our implementations is the use of block-partitioned algorithms, particularly in conjunction with highly-tuned kernels for performing matrix-vector and matrix-matrix operations (the Level 2 and 3 BLAS).

2 Dense Linear Algebra Algorithms

2.1 Overview of Dense Algorithms

Common operations involving dense matrices are the solution of linear systems

$$Ax = b,$$

the least squares solution of over- or underdetermined systems

$$\min_x \|Ax - b\|,$$

and the computation of eigenvalues and -vectors

$$Ax = \lambda x.$$

Although these problems are formulated as matrix-vector equations, their solution involves a definite matrix-matrix component. For instance, in order to solve a linear system, the coefficient matrix is first factored as

$$A = LU$$

(or $A = U^T U$ in the case of symmetry) where L and U are lower and upper triangular matrices respectively. It is a common feature of these matrix-matrix operations that they take, on a matrix of size $n \times n$, a number of operations proportional to n^3, a factor n more than the number of data elements involved.

Thus we are led to identify three levels of linear algebra operations:

- Level 1: vector-vector operations such as the update $y \leftarrow y + \alpha x$ and the inner product $d = x^T y$. These operations involve (for vectors of length n) $O(n)$ data and $O(n)$ operations.
- Level 2: matrix-vector operations such as the matrix-vector product $y = Ax$. These involve $O(n^2)$ operations on $O(n^2)$ data.
- Level 3: matrix-matrix operations such as the matrix-matrix product $C = AB$. These involve $O(n^3)$ operations on $O(n^2)$ data.

These three levels of operations have been realized in a software standards known as the Basic Linear Algebra Subprograms (BLAS) [17, 18, 46]. Although BLAS routines are freely available on the net, many computer vendors supply a tuned, often assembly-coded, BLAS library optimised for their particular architecture. See also section 4.3.

The relation between the number of operations and the amount of data is crucial for the performance of the algorithm. We discuss this in detail in section 3.1.

2.1.1 Loop rearranging

The operations of BLAS Levels 2 and 3 can be implemented using doubly and triply nested loops respectively. With simply modifications, this means that for Level 2 each algorithms has two, and for Level 3 six different implementations [23]. For instance, solving a lower triangular system $Lx = y$ is mostly written

for $i = 1 \dots n$
    $t = 0$
    for $j = 1 \dots i-1$
        $t \leftarrow t + \ell_{ij} x_j$
    $x = \ell_{ii}^{-1}(y_i - t)$

but can also be written as

for $j = 1 \dots n$
    $x_j = \ell_{jj}^{-1} y_j$
    for $i = j+1 \dots n$
        $y_i \leftarrow y_i - \ell_{ij} x_j$

(The latter implementation overwrites the right hand side vector y, but this can be eliminated.)

While the two implementations are equivalent in terms of numbers of operations, there may be substantial differences in performance due to architectural considerations. We note, for instance, that the inner loop in the first implementation uses a row of L, whereas the innner loop in the second traverses a column. Since matrices are usually stored with either rows or columns in contiguous locations, column storage the historical default inherited from the FORTRAN programming language, the performance of the two can be radically different. We discuss this point further in section 3.1.

2.1.2 Uses of LU Factorization in Science and Engineering

A major source of large dense linear systems is problems involving the solution of boundary integral equations [26]. These are integral equations defined on the boundary of a region of interest. All examples of practical interest compute some intermediate quantity on a two-dimensional boundary and then use this information to compute the final desired quantity in three-dimensional space. The price one pays for replacing three dimensions with two is that what started as a sparse problem in $O(n^3)$ variables is replaced by a dense problem in $O(n^2)$.

Dense systems of linear equations are found in numerous applications, including:

- airplane wing design;
- radar cross-section studies;
- flow around ships and other off-shore constructions;

- diffusion of solid bodies in a liquid;
- noise reduction; and
- diffusion of light through small particles.

The electromagnetics community is a major user of dense linear systems solvers. Of particular interest to this community is the solution of the so-called radar cross-section problem. In this problem, a signal of fixed frequency bounces off an object; the goal is to determine the intensity of the reflected signal in all possible directions. The underlying differential equation may vary, depending on the specific problem. In the design of stealth aircraft, the principal equation is the Helmholtz equation. To solve this equation, researchers use the *method of moments* [37, 62]. In the case of fluid flow, the problem often involves solving the Laplace or Poisson equation. Here, the boundary integral solution is known as the *panel method* [38, 39], so named from the quadrilaterals that discretize and approximate a structure such as an airplane. Generally, these methods are called *boundary element methods*.

Use of these methods produces a dense linear system of size $O(N)$ by $O(N)$, where N is the number of boundary points (or panels) being used. It is not unusual to see size $3N$ by $3N$, because of three physical quantities of interest at every boundary element.

A typical approach to solving such systems is to use LU factorization. Each entry of the matrix is computed as an interaction of two boundary elements. Often, many integrals must be computed. In many instances, the time required to compute the matrix is considerably larger than the time for solution.

The builders of stealth technology who are interested in radar cross-sections are using direct Gaussian elimination methods for solving dense linear systems. These systems are always symmetric and complex, but not Hermitian.

For further information on various methods for solving large dense linear algebra problems that arise in computational fluid dynamics, see the report by Alan Edelman [26].

2.2 Block Algorithms and Their Derivation

It is comparatively straightforward to recode many of the dense linear algebra algorithms so that they use Level 2 BLAS. Indeed, in the simplest cases the same floating-point operations are done, possibly even in the same order: it is just a matter of reorganizing the software. To illustrate this point, we consider the Cholesky factorization algorithm, which factors a symmetric positive definite matrix as $A = U^T U$. We consider Cholesky factorization because the algorithm is simple, and no pivoting is required on a positive definite matrix.

Suppose that after $j-1$ steps the block A_{00} in the upper lefthand corner of A has been factored as $A_{00} = U_{00}^T U_{00}$. The next row and column of the factorization can then be computed by writing $A = U^T U$ as

$$\begin{pmatrix} A_{00} & b_j & A_{02} \\ . & a_{jj} & c_j^T \\ . & . & A_{22} \end{pmatrix} = \begin{pmatrix} U_{00}^T & 0 & 0 \\ v_j^T & u_{jj} & 0 \\ U_{02}^T & w_j & U_{22}^T \end{pmatrix} \begin{pmatrix} U_{00} & v_j & U_{02} \\ 0 & u_{jj} & w_j^T \\ 0 & 0 & U_{22} \end{pmatrix}$$

where b_j, c_j, v_j, and w_j are column vectors of length $j-1$, and a_{jj} and u_{jj} are scalars. Equating coefficients of the j^{th} column, we obtain

$$\begin{aligned} b_j &= U_{00}^T v_j \\ a_{jj} &= v_j^T v_j + u_{jj}^2. \end{aligned}$$

Since U_{00} has already been computed, we can compute v_j and u_{jj} from the equations

$$\begin{aligned} U_{00}^T v_j &= b_j \\ u_{jj}^2 &= a_{jj} - v_j^T v_j. \end{aligned}$$

The computation of v_j is a triangular system solution, a BLAS Level 2 operation. Thus, a code using this will have a single call replacing a loop of Level 1 calls or a doubly nested loop of scalar operations.

This change by itself is sufficient to result in large gains in performance on a number of machines—for example, from 72 to 251 megaflops for a matrix of order 500 on one processor of a CRAY Y-MP. Since this is 81% of the peak speed of matrix-matrix multiplication on this processor, we cannot hope to do very much better by using Level 3 BLAS.

We can, however, restructure the algorithm at a deeper level to exploit the faster speed of the Level 3 BLAS. This restructuring involves recasting the algorithm as a **block algorithm**—that is, an algorithm that operates on **blocks** or submatrices of the original matrix.

2.2.1 Deriving a Block Algorithm

To derive a block form of Cholesky factorization, we partition the matrices as shown in Figure 1, in which the diagonal blocks of A and U are square, but of differing sizes. We assume that the first block has already been factored as $A_{00} = U_{00}^T U_{00}$, and that we now want to determine the second block column of U consisting of the blocks U_{01} and U_{11}. Equating submatrices in the second block of columns, we obtain

$$\begin{aligned} A_{01} &= U_{00}^T U_{01} \\ A_{11} &= U_{01}^T U_{01} + U_{11}^T U_{11}. \end{aligned}$$

Hence, since U_{00} has already been computed, we can compute U_{01} as the solution to the equation

$$U_{00}^T U_{01} = A_{01}$$

$$\left[\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{01}^T & A_{11} & A_{12} \\ \hline A_{02}^T & A_{12}^T & A_{22} \end{array}\right] = \left[\begin{array}{c|c|c} U_{00}^T & 0 & 0 \\ \hline U_{01}^T & U_{11}^T & 0 \\ \hline U_{02}^T & U_{12}^T & U_{22}^T \end{array}\right] * \left[\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right]$$

Figure 1: Partitioning of A, U^T, and U into blocks. It is assumed that the first block has already been factored as $A_{00} = U_{00}^T U_{00}$, and we next want to determine the block column consisting of U_{01} and U_{11}. Note that the diagonal blocks of A and U are square matrices.

by a call to the Level 3 BLAS routine STRSM; and then we can compute U_{11} from

$$U_{11}^T U_{11} = A_{11} - U_{01}^T U_{01}.$$

This involves first updating the symmetric submatrix A_{11} by a call to the Level 3 BLAS routine SSYRK, and then computing its Cholesky factorization. Since Fortran does not allow recursion, a separate routine must be called, using Level 2 BLAS rather than Level 3. In this way, successive blocks of columns of U are computed.

But that is not the end of the story, and the code given above is not the code actually used in the LAPACK routine SPOTRF. We mentioned earlier that for many linear algebra computations there are several algorithmic variants, often referred to as i-, j-, and k-variants, according to a convention introduced in [15, 23] and explored further in [53, 54]. The same is true of the corresponding block algorithms.

It turns out that the j-variant chosen for LINPACK, and used in the above examples, is not the fastest on many machines, because it performs most of the work in solving triangular systems of equations, which can be significantly slower than matrix-matrix multiplication. The variant actually used in LAPACK is the i-variant, which relies on matrix-matrix multiplication for most of the work.

Table 1 summarizes the results.

Table 1: Speed (Megaflops) of Cholesky Factorization $A = U^T U$ for $n = 500$

	CRAY T-90, 1 proc.	CRAY T-90, 4 proc.
j-variant: LINPACK	376	392
j-variant: using Level 3 BLAS	1222	2306
i-variant: using Level 3 BLAS	1297	3279

3 The influence of computer architecture on performance

3.1 Discussion of architectural features

In section 2.1.1 we noted that for BLAS Levels 2 and 3 several equivalent implementations of the operations exist. These differ, for instance, in whether they access a matrix operand by rows or columns in the inner loop. In FORTRAN, matrices are stored by columns, so accessing a column corresponds to accessing consecutive memory elements. 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.

We will now give a simplified discussion on the various architectural issues that influence the choice of algorithm. The following is, of necessity, a simplified account of the state of affairs for any particular architecture.

At first, we concentrate only on 'non-blocked' algorithms. In blocked methods, discussed in more detail below, every algorithm has two levels on which we can consider loop arranging: the block level, and the scalar level. Often, the best arrangement on one level is not the best on the other. The next two subsections concern themselves with the scalar level.

3.1.1 Using consecutive elements

The decision how to traverse matrix elements should usually be taken so as to use elements that are consecutive in storage. There are at least three architectural reasons for this.

page swapping By using consecutive memory elements, instead of ones at some stride distance of each other, the amount of memory page swapping is minimized.

memory banks If the processor cycle is faster than the memory cycle, and memory consists of interleaved banks, consecutive elements will be in different banks. By contrast, taking elements separated a distance equal to the number of banks, all elements will come from the same bank. This will reduce the effective performance of the algorithm to the memory speed instead of the processor speed.

cache lines Processors with a memory cache typically do not bring in single elements from memory to cache, but move them one 'cache line' at a time. A cache line consists of a small number of consecutive memory elements. Thus, using consecutive memory storage elements means that a next element will already be in cache and does not have to be brought into cache. This cuts down on memory traffic.

Whether consecutive elements correspond to rows or columns in a matrix depends on the programming language used. In Fortran, columns are stored consecutively, whereas C has row elements contiguous in memory.

The effects of column orientation are quite dramatic: on systems with virtual or cache memories, the LINPACK library codes (section 4.4.2), which are written in FORTRAN and which are column-oriented, will significantly outperform FORTRAN codes that are not column-oriented. In the C language, however, algorithms should be formulated with row-orientation. We note that textbook examples of matrix algorithms are usually given in a row-oriented manner.

3.1.2 Cache reuse

In many contemporary architectures, memory bandwidth is not enough to keep the processor working at its peak rate. Therefore, the architecture incorporates some cache memory, a relatively small store of faster memory. The memory bandwidth problem is now shifted to bringing the elements into cache, and this problem can be obviated almost entirely if the algorithm can re-use cache elements.

Consider for instance a matrix-vector product $y = Ax$. The doubly nested loop has an inner statement

$$y_i \leftarrow y_i + a_{ij}a_j$$

implying three reads and one write from memory for two operations. If we write the algorithm as

$$y_* = x_1 a_{1*} + x_2 a_{2*} + \cdots$$

we see that, keeping y in cache[1] and reusing the elements of x, we only need to load the column of A, making the asymptotic demand on memory one element load once x and y have been brought into cache.

3.1.3 Blocking for cache reuse

Above, we saw in the Cholesky example how algorithms can naturally be written in terms of Level 2 operations. In order to use Level 3 operations, a more drastic rewrite is needed.

Suppose we want to perform the matrix-matrix multiplication $C = AB$, where all matrices are of size $n \times n$. We divide all matrices in subblocks of

[1]Since many level-1 caches are write-through, we wouldn't actually keep y in cache, but rather keep a number of elements of it in register, and reuse these registers by unrolling the '$*$' loop.

size $k \times k$, and let for simplicity's sake k divide n: $n = km$. Then the triply nested scalar loop becomes, in one possible rearrangement

for $i = 1 \ldots m$
    for $k = 1 \ldots m$
        for $j = 1 \ldots m$
            $C_{ij} \leftarrow C_{ij} + A_{ik}B_{kj}$

where the inner statement is now a size k matrix-matrix multiplication.

If the cache is now large enough three of these smaller matrices, we can keep C_{ij} and A_{ik} in cache[2], while successive blocks B_{kj} are being brought in. The ratio of memory loads to operations is then (ignoring the loads of the elements of C and A, which are amortised) k^2/k^3, that is, $1/k$.

Thus, by blocking the algorithm, and arranging the loops so that blocks are reused in cache, we can achieve high performance in spite of a low memory bandwidth.

3.2 Target Architectures

The EISPACK and LINPACK software libraries were designed for supercomputers used in the 1970s and early 1980s, such as the CDC-7600, Cyber 205, and Cray-1. These machines featured multiple functional units pipelined for good performance [41]. The CDC-7600 was basically a high-performance scalar computer, while the Cyber 205 and Cray-1 were early vector computers.

The development of LAPACK in the late 1980s was intended to make the EISPACK and LINPACK libraries run efficiently on shared memory, vector supercomputers. The ScaLAPACK software library will extend the use of LAPACK to distributed memory concurrent supercomputers. The development of ScaLAPACK began in 1991 and is had its first public release of software by the end of 1994.

The underlying concept of both the LAPACK and ScaLAPACK libraries is the use of block-partitioned algorithms to minimize data movement between different levels in hierarchical memory. Thus, the ideas discussed in this chapter for developing a library for dense linear algebra computations are applicable to any computer with a hierarchical memory that (1) imposes a sufficiently large startup cost on the movement of data between different levels in the hierarchy, and for which (2) the cost of a context switch is too great to make fine grain size multithreading worthwhile. Our target machines are, therefore, medium and large grain size advanced-architecture computers. These include "traditional" shared memory, vector supercomputers, such as the Cray C-90 and T-90, and MIMD distributed memory concurrent supercomputers, such as the SGI Origin 2000, IBM SP, Cray T3E, and HP/Convex Exemplar concurrent systems.

Future advances in compiler and hardware technologies are expected to make multithreading a viable approach for masking communication costs. Since the blocks in a block-partitioned algorithm can be regarded as separate threads, our

[2] Again, with a write-through level-1 cache, one would try to keep C_{ij} in registers.

approach will still be applicable on machines that exploit medium and coarse grain size multithreading.

4 Dense Linear Algebra Libraries

4.1 Requirements on High-Quality, Reusable, Mathematical Software

In developing a library of high-quality subroutines for dense linear algebra computations the design goals fall into three broad classes:

- performance
- ease-of-use
- range-of-use

4.1.1 Performance

Two important performance metrics are *concurrent efficiency* and *scalability*. We seek good performance characteristics in our algorithms by eliminating, as much as possible, overhead due to load imbalance, data movement, and algorithm restructuring. The way the data are distributed (or decomposed) over the memory hierarchy of a computer is of fundamental importance to these factors. Concurrent efficiency, ϵ, is defined as the concurrent speedup per processor [32], where the concurrent speedup is the execution time, T_{seq}, for the best sequential algorithm running on one processor of the concurrent computer, divided by the execution time, T, of the parallel algorithm running on N_p processors. When direct methods are used, as in LU factorization, the concurrent efficiency depends on the problem size and the number of processors, so on a given parallel computer and for a fixed number of processors, the running time should not vary greatly for problems of the same size. Thus, we may write,

$$\epsilon(N, N_p) = \frac{1}{N_p} \frac{T_{\mathrm{seq}}(N)}{T(N, N_p)} \qquad (1)$$

where N represents the problem size. In dense linear algebra computations, the execution time is usually dominated by the floating-point operation count, so the concurrent efficiency is related to the performance, G, measured in floating-point operations per second by,

$$G(N, N_p) = \frac{N_p}{t_{\mathrm{calc}}} \, \epsilon(N, N_p) \qquad (2)$$

where t_{calc} is the time for one floating-point operation. For iterative routines, such as eigensolvers, the number of iterations, and hence the execution time, depends not only on the problem size, but also on other characteristics of the input data, such as condition number. A parallel algorithm is said to be scalable

[34] if the concurrent efficiency depends on the problem size and number of processors only through their ratio. This ratio is simply the problem size per processor, often referred to as the granularity. Thus, for a scalable algorithm, the concurrent efficiency is constant as the number of processors increases while keeping the granularity fixed. Alternatively, Eq. 2 shows that this is equivalent to saying that, for a scalable algorithm, the performance depends linearly on the number of processors for fixed granularity.

4.1.2 Ease-Of-Use

Ease-of-use is concerned with factors such as portability and the user interface to the library. Portability, in its most inclusive sense, means that the code is written in a standard language, such as Fortran or C, and that the source code can be compiled on an arbitrary machine to produce a program that will run correctly. We call this the "mail-order software" model of portability, since it reflects the model used by software servers such as *netlib* [19]. This notion of portability is quite demanding. It requires that all relevant properties of the computer's arithmetic and architecture be discovered at runtime within the confines of a compilable Fortran code. For example, if it is important to know the overflow threshold for scaling purposes, it must be determined at runtime *without overflowing*, since overflow is generally fatal. Such demands have resulted in quite large and sophisticated programs [24, 44], which must be modified frequently to deal with new architectures and software releases. This "mail-order" notion of software portability also means that codes generally must be written for the worst possible machine expected to be used, thereby often degrading performance on all others. Ease-of-use is also enhanced if implementation details are largely hidden from the user, for example, through the use of an object-based interface to the library [21].

4.1.3 Range-Of-Use

Range-of-use may be gauged by how numerically stable the algorithms are over a range of input problems, and the range of data structures the library will support. For example, LINPACK and EISPACK deal with dense matrices stored in a rectangular array, packed matrices where only the upper or lower half of a symmetric matrix is stored, and banded matrices where only the nonzero bands are stored. In addition, some special formats such as Householder vectors are used internally to represent orthogonal matrices. In the second half of this paper we will focus on sparse matrices, that is matrices with many zero elements, which may be stored in many different ways.

4.2 Portability, Scalability, and Standards

Portability of programs has always been an important consideration. Portability was easy to achieve when there was a single architectural paradigm (the serial von Neumann machine) and a single programming language for scientific programming (Fortran) embodying that common model of computation. Architectural and linguistic diversity have made portability much more difficult, but no less important, to attain. Users simply do not wish to invest significant amounts of time to create large-scale application codes for each new machine. Our answer is to develop portable software libraries that hide machine-specific details.

In order to be truly portable, parallel software libraries must be *standardized.* In a parallel computing environment in which the higher-level routines and/or abstractions are built upon lower-level computation and message-passing routines, the benefits of standardization are particularly apparent. Furthermore, the definition of computational and message-passing standards provides vendors with a clearly defined base set of routines that they can implement efficiently.

From the user's point of view, portability means that, as new machines are developed, they are simply added to the network, supplying cycles where they are most appropriate.

From the mathematical software developer's point of view, portability may require significant effort. Economy in development and maintenance of mathematical software demands that such development effort be leveraged over as many different computer systems as possible. Given the great diversity of parallel architectures, this type of portability is attainable to only a limited degree, but machine dependences can at least be isolated.

LAPACK is an example of a mathematical software package whose highest-level components are portable, while machine dependences are hidden in lower-level modules. Such a hierarchical approach is probably the closest one can come to software portability across diverse parallel architectures. And the BLAS that are used so heavily in LAPACK provide a portable, efficient, and flexible standard for applications programmers.

Like portability, *scalability* demands that a program be reasonably effective over a wide range of number of processors. Maintaining scalability of parallel algorithms, and the software libraries implementing them, over a wide range of architectural designs and numbers of processors will likely require that the fundamental granularity of computation be adjustable to suit the particular circumstances in which the software may happen to execute. Our approach to this problem is block algorithms with adjustable block size. In many cases, however, polyalgorithms[3] may be required to deal with the full range of architectures and processor multiplicity likely to be available in the future.

Scalable parallel architectures of the future are likely to be based on a distributed memory architectural paradigm. In the longer term, progress in hardware development, operating systems, languages, compilers, and communications may make it possible for users to view such distributed architectures (without significant loss of efficiency) as having a shared memory with a global address space. For the near term, however, the distributed nature of the underlying hardware will continue to be visible at the programming level; therefore,

[3]In a polyalgorithm the actual algorithm used depends on the computing environment and the input data. The optimal algorithm in a particular instance is automatically selected at runtime.

efficient procedures for explicit communication will continue to be necessary. Given this fact, standards for basic message passing (send/receive), as well as higher-level communication constructs (global summation, broadcast, etc.), become essential to the development of scalable libraries that have any degree of portability. In addition to standardizing general communication primitives, it may also be advantageous to establish standards for problem-specific constructs in commonly occurring areas such as linear algebra.

The BLACS (Basic Linear Algebra Communication Subprograms) [16, 22] is a package that provides the same ease of use and portability for MIMD message-passing linear algebra communication that the BLAS [17, 18, 46] provide for linear algebra computation. Therefore, we recommend that future software for dense linear algebra on MIMD platforms consist of calls to the BLAS for computation and calls to the BLACS for communication. Since both packages will have been optimized for a particular platform, good performance should be achieved with relatively little effort. Also, since both packages will be available on a wide variety of machines, code modifications required to change platforms should be minimal.

4.3 The BLAS as the Key to Portability

At least three factors affect the performance of compilable code.

1. **Vectorization/Cache Reuse.** Designing vectorizable algorithms in linear algebra is usually straightforward. Indeed, for many computations there are several variants, all vectorizable, but with different characteristics in performance (see, for example, [15]). Linear algebra algorithms can approach the peak performance of many machines—principally because peak performance depends on some form of chaining of vector addition and multiplication operations or cache reuse, and this is just what the algorithms require. However, when the algorithms are realized in straightforward Fortran77 or C code, the performance may fall well short of the expected level, usually because Fortran compilers fail to minimize the number of memory references—that is, the number of vector load and store operations or effectively reuse cache.

2. **Data movement.** What often limits the actual performance of a vector, or scalar, floating-point unit is the rate of transfer of data between different levels of memory in the machine. Examples include the transfer of vector operands in and out of vector registers, the transfer of scalar operands in and out of a high-speed, the movement of data between main memory and a high-speed cache or local memory, paging between actual memory and disk storage in a virtual memory system, and interprocessor communication on a distributed memory concurrent computer.

3. **Parallelism.** The nested loop structure of most linear algebra algorithms offers considerable scope for loop-based parallelism. This is the principal

Table 2: Speed in Mflop/s of Level 2 and Level 3 BLAS operations on a CRAY C90

(all matrices are of order 1000; U is upper triangular)

Number of processors:	1	2	4	8	16
Level 2: $y \leftarrow \alpha Ax + \beta y$	899	1780	3491	6783	11207
Level 3: $C \leftarrow \alpha AB + \beta C$	900	1800	3600	7199	14282
Level 2: $x \leftarrow Ux$	852	1620	3063	5554	6953
Level 3: $B \leftarrow UB$	900	1800	3574	7147	13281
Level 2: $x \leftarrow U^{-1}x$	802	1065	1452	1697	1558
Level 3: $B \leftarrow U^{-1}B$	896	1792	3578	7155	14009

type of parallelism that LAPACK and ScaLAPACK presently aim to exploit. On shared memory concurrent computers, this type of parallelism can sometimes be generated automatically by a compiler, but often requires the insertion of compiler directives. On distributed memory concurrent computers, data must be moved between processors. This is usually done by explicit calls to message passing routines, although parallel language extensions such as Coherent Parallel C [30] and Split-C [13] do the message passing implicitly.

These issues can be controlled, while obtaining the levels of performance that machines can offer, through use of the BLAS, introduced in section 2.1.

The Level 1 BLAS are used in LAPACK, but for convenience rather than for performance: they perform an insignificant fraction of the computation, and they cannot achieve high efficiency on most modern supercomputers. Also, the overhead entailed in calling the BLAS reduces the efficiency of the code. This reduction is negligible for large matrices, but it can be quite significant for small matrices. Fortunately, the level 1 BLAS can be removed from the smaller, more frequently used LAPACK codes in a short editing session.

The Level 2 BLAS can achieve near-peak performance on many vector processors, such as a single processor of a CRAY X-MP or Y-MP, or Convex C-2 machine. However, on other vector processors such as a CRAY-2 or an IBM 3090 VF, the performance of the Level 2 BLAS is limited by the rate of data movement between different levels of memory.

The Level 3 BLAS overcome this limitation. This third level of BLAS performs $O(n^3)$ floating-point operations on $O(n^2)$ data, whereas the Level 2 BLAS perform only $O(n^2)$ operations on $O(n^2)$ data. The Level 3 BLAS also allow us to exploit parallelism in a way that is transparent to the software that calls them. While the Level 2 BLAS offer some scope for exploiting parallelism, greater scope is provided by the Level 3 BLAS, as Table 2 illustrates.

The BLAS can provide portable high performance through being a standard that is available on many platforms. Ideally, the computer manufacturer has provided an assembly-coded BLAS tuned for that particular architecture, but there is a standard implementation available that can simply be compiled and linked. Using this standard BLAS may improve the efficiency of programs when they 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 for matrices of quite small order, and for large orders the savings are quite significant.

4.4 Overview of dense linear algebra libraries

Over the past twenty-five years, we have has been directly involved in the development of several important packages of dense linear algebra software: EISPACK, LINPACK, LAPACK, and the BLAS. Most recently, we have been involved in the development of ScaLAPACK, a scalable version of LAPACK for distributed memory concurrent computers. In this section, we give a brief review of these packages—their history, their advantages, and their limitations on high-performance computers.

4.4.1 EISPACK

EISPACK is a collection of Fortran subroutines that compute the eigenvalues and eigenvectors of nine classes of matrices: complex general, complex Hermitian, real general, real symmetric, real symmetric banded, real symmetric tridiagonal, special real tridiagonal, generalized real, and generalized real symmetric matrices. In addition, two routines are included that use singular value decomposition to solve certain least-squares problems.

EISPACK is primarily based on a collection of Algol procedures developed in the 1960s and collected by J. H. Wilkinson and C. Reinsch in a volume entitled *Linear Algebra* in the *Handbook for Automatic Computation* [64] series. This volume was not designed to cover every possible method of solution; rather, algorithms were chosen on the basis of their generality, elegance, accuracy, speed, or economy of storage.

Since the release of EISPACK in 1972, over ten thousand copies of the collection have been distributed worldwide.

4.4.2 LINPACK

LINPACK is a collection of Fortran subroutines that analyze and solve linear equations and linear least-squares problems. The package solves linear systems whose matrices are general, banded, symmetric indefinite, symmetric positive definite, triangular, and tridiagonal square. In addition, the package computes the QR and singular value decompositions of rectangular matrices and applies them to least-squares problems.

LINPACK is organized around four matrix factorizations: LU factorization, pivoted Cholesky factorization, QR factorization, and singular value decomposition. The term LU factorization is used here in a very general sense to mean the factorization of a square matrix into a lower triangular part and an upper triangular part, perhaps with pivoting. Next we describe the organization and factors influencing LINPACK's efficiency.

LINPACK uses column-oriented algorithms to increase efficiency by preserving locality of reference. By column orientation we mean that the LINPACK codes always reference arrays down columns, not across rows. This works because Fortran stores arrays in column major order. This means that as one proceeds down a column of an array, the memory references proceed sequentially in memory. Thus, if a program references an item in a particular block, the next reference is likely to be in the same block. See further section 3.1.1. LINPACK uses the Level 1 BLAS; see section 4.3.

Since the release of LINPACK, over twenty thousand copies of the collection have been distributed worldwide.

4.4.3 LAPACK

LAPACK [14] provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) are also provided, as are related computations such as reordering of the Schur factorizations and estimating condition numbers. Dense and banded matrices are handled, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices, in both single and double precision.

The original goal of the LAPACK project was to make the widely used EISPACK and LINPACK libraries run efficiently on shared-memory vector and parallel processors. On these machines, LINPACK and EISPACK are inefficient because their memory access patterns disregard the multilayered memory hierarchies of the machines, thereby spending too much time moving data instead of doing useful floating-point operations. LAPACK addresses this problem by reorganizing the algorithms to use block matrix operations, such as matrix multiplication, in the innermost loops [2, 14]. These block operations can be optimized for each architecture to account for the memory hierarchy [1], and so provide a transportable way to achieve high efficiency on diverse modern machines. Here we use the term "transportable" instead of "portable" because, for fastest possible performance, LAPACK requires that highly optimized block matrix operations be already implemented on each machine. In other words, the correctness of the code is portable, but high performance is not—if we limit ourselves to a single Fortran source code.

LAPACK can be regarded as a successor to LINPACK and EISPACK. It has virtually all the capabilities of these two packages and much more besides. LAPACK improves on LINPACK and EISPACK in four main respects: speed, accuracy, robustness and functionality. While LINPACK and EISPACK are based on the vector operation kernels of the Level 1 BLAS, LAPACK was designed at the outset to exploit the Level 3 BLAS —a set of specifications for Fortran subprograms that do various types of matrix multiplication and the solution of triangular systems with multiple right-hand sides. Because of the coarse granularity of the Level 3 BLAS operations, their use tends to promote high efficiency on many high-performance computers, particularly if specially coded implementations are provided by the manufacturer.

LAPACK is designed to give high efficiency on vector processors, high-performance "superscalar" workstations, and shared memory multiprocessors. LAPACK in its present form is less likely to give good performance on other types of parallel architectures (for example, massively parallel SIMD machines, or MIMD distributed memory machines), but the ScaLAPACK project, described in Section 4.4.4, is intended to adapt LAPACK to these new architectures. LAPACK can also be used satisfactorily on all types of scalar machines (PCs, workstations, mainframes).

LAPACK, like LINPACK, provides LU and Cholesky factorizations of band matrices. The LINPACK algorithms can easily be restructured to use Level 2 BLAS, though restructuring has little effect on performance for matrices of very narrow bandwidth. It is also possible to use Level 3 BLAS, at the price of doing some extra work with zero elements outside the band [20]. This process becomes worthwhile for large matrices and semi-bandwidth greater than 100 or so.

4.4.4 ScaLAPACK

The ScaLAPACK software library extends the LAPACK library to run scalably on MIMD, distributed memory, concurrent computers [10, 11]. For such machines the memory hierarchy includes the off-processor memory of other processors, in addition to the hierarchy of registers, cache, and local memory on each processor. Like LAPACK, the ScaLAPACK routines are based on block-partitioned algorithms in order to minimize the frequency of data movement between different levels of the memory hierarchy. The fundamental building blocks of the ScaLAPACK library are distributed memory versions of the Level 2 and Level 3 BLAS, and a set of Basic Linear Algebra Communication Subprograms (BLACS) [16, 22] for communication tasks that arise frequently in parallel linear algebra computations. In the ScaLAPACK routines, all interprocessor communication occurs within the distributed BLAS and the BLACS, so the source code of the top software layer of ScaLAPACK looks very similar to that of LAPACK.

5 Future Research Directions in Dense Algorithms

Traditionally, large, general-purpose mathematical software libraries have required users to write their own programs that call library routines to solve specific subproblems that arise during a computation. Adapted to a shared-memory parallel environment, this conventional interface still offers some potential for hiding underlying complexity. For example, the LAPACK project incorporates parallelism in the Level 3 BLAS, where it is not directly visible to the user.

But when going from shared-memory systems to the more readily scalable distributed memory systems, the complexity of the distributed data structures required is more difficult to hide from the user. Not only must the problem decomposition and data layout be specified, but different phases of the user's problem may require transformations between different distributed data structures.

These deficiencies in the conventional user interface have prompted extensive discussion of alternative approaches for scalable parallel software libraries of the future. Possibilities include:

1. Traditional function library (i.e., minimum possible change to the status quo in going from serial to parallel environment). This will allow one to protect the programming investment that has been made.

2. Reactive servers on the network. A user would be able to send a computational problem to a server that was specialized in dealing with the problem. This fits well with the concepts of a networked, heterogeneous computing environment with various specialized hardware resources (or even the heterogeneous partitioning of a single homogeneous parallel machine).

3. General interactive environments like Matlab or Mathematica, perhaps with "expert" drivers (i.e., knowledge-based systems). With the growing popularity of the many integrated packages based on this idea, this approach would provide an interactive, graphical interface for specifying and solving scientific problems. Both the algorithms and data structures are hidden from the user, because the package itself is responsible for storing and retrieving the problem data in an efficient, distributed manner. In a heterogeneous networked environment, such interfaces could provide seamless access to computational engines that would be invoked selectively for different parts of the user's computation according to which machine is most appropriate for a particular subproblem.

4. Domain-specific problem solving environments, such as those for structural analysis. Environments like Matlab and Mathematica have proven to be especially attractive for rapid prototyping of new algorithms and systems that may subsequently be implemented in a more customized manner for higher performance.

5. Reusable templates (i.e., users adapt "source code" to their particular applications). A template is a description of a general algorithm rather than the executable object code or the source code more commonly found in a conventional software library. Nevertheless, although templates are

general descriptions of key data structures, they offer whatever degree of customization the user may desire.

Novel user interfaces that hide the complexity of scalable parallelism will require new concepts and mechanisms for representing scientific computational problems and for specifying how those problems relate to each other. Very high level languages and systems, perhaps graphically based, not only would facilitate the use of mathematical software from the user's point of view, but also would help to automate the determination of effective partitioning, mapping, granularity, data structures, etc. However, new concepts in problem specification and representation may also require new mathematical research on the analytic, algebraic, and topological properties of problems (e.g., existence and uniqueness).

We have already begun work on developing such templates for sparse matrix computations. Future work will focus on extending the use of templates to dense matrix computations.

We hope the insight we gained from our work will influence future developers of hardware, compilers and systems software so that they provide tools to facilitate development of high quality portable numerical software.

The EISPACK, LINPACK, and LAPACK linear algebra libraries are in the public domain, and are available from *netlib*. For example, for more information on how to obtain LAPACK, send the following one-line email message to `netlib@ornl.gov`:

```
send index from lapack
```

or visit the web site at *http://www.netlib.org/lapack/*. Information for EISPACK, LINPACK, and ScaLAPACK can be similarly obtained.

6 Sparse Linear Algebra Methods

6.1 Origin of sparse linear systems

The most common source of sparse linear systems is the numerical solution of partial differential equations. Many physical problems, such as fluid flow or elasticity, can be described by partial differential equations. These are implicit descriptions of a physical model, describing some internal relation such as stress forces. In order to arrive at an explicit description of the shape of the object or the temperature distribution, we need to solve the PDE, and for this we need numerical methods.

6.1.1 Discretised partial differential equations

Several methods for the numerical solution of PDEs exist, the most common ones being the methods of finite elements, finite differences, and finite volumes. A common feature of these is that they identify discrete points in the physical object, and give a set of equations relating these points.

Typically, only points that are physically close together are related to each other in this way. This gives a matrix structure with very few nonzero elements per row, and the nonzeros are often confined to a 'band' in the matrix.

6.1.2 Sparse matrix structure

Matrices from discretised partial differential equations contain so many zero elements that it pays to find a storage structure that avoids storing these zeros. The resulting memory savings, however, are offset by an increase in programming complexity, and by descreased efficiency of even simple operations such as the matrix vector product.

More complicated operations, such as solving a linear system, with such a sparse matrix present a next level of complication, as both the inverse and the LU factorisation of a sparse matrix are not as sparse, thus needing considerably more storage. Specifically, the inverse of the type of sparse matrix we are considering is a full matrix, and factoring such a sparse matrix fills in the band completely.

Example: central differences in d dimensions, n points per line, matrix size $N = n^d$, bandwidth $q = n^{d-1}$ in natural ordering, number of nonzero $\sim n^d$, number of matrix elements $N^2 = n^{2d}$, number of elements in factorisation $N^{1+(d-1)/d}$.

6.2 Basic elements in sparse linear algebra methods

Methods for sparse systems use, like those for dense systems, vector-vector, matrix-vector, and matrix-matrix operations. However, there are some important differences.

For iterative methods, discussed in section 8, there are almost no matrix-matrix operations. See [43] for an exception. Since most modern architectures prefer these Level 3 operations, the performance of iterative methods will be limited from the outset.

An even more serious objection is that the sparsity of the matrix implies that indirect addressing is used for retrieving elements. For example, in the popular row-compressed matrix storage format, the matrix-vector multiplication looks like

for $i = 1 \ldots n$
  $p \leftarrow$ pointer to row i
  for $j = 1, n_i$
    $y_i \leftarrow y_i + a(p + j)x(c(p + j))$

where n_i is the number of nonzeros in row i, and $p(\cdot)$ is an array of column indices. A number of such algorithms for several sparse data formats are given in [6].

Direct methods can have a BLAS 3 component if they are a type of dissection method. However, in a given sparse problem, the more dense the matrices are, the smaller they are on average. They are also not general full matrices, but only banded. Thus we don't expect very high performance on such methods either.

7 Direct solution methods

For the solution of a linear system one needs to factor the coefficient matrix. Any direct method is a variant of Gaussian elimination. As remarked above, for a sparse matrix, this fills in the band in which the nonzero elements are contained. In order to minimise the storage needed for the factorisation, research has focused on finding suitable orderings of the matrix. Re-ordering the equations by a symmetric permutation of the matrix does not change the numerical properties of the system in many cases, and it can potentially give large savings in storage. In general, direct methods do not make use of the numerical properties of the linear system, and thus their execution time is affected in a major way by the structural properties of the input matrix.

7.1 Matrix graph theory

The most convenient way of talking about matrix orderings or permutations is to consider the matrix 'graph' [55]. We introduce a node for every physical variable, and nodes i and j are connected in the graph if the (i, j) element of the matrix is nonzero. A symmetric permutation of the matrix then corresponds to a numbering of the nodes, while the connections stay the same. With these permutations, one hopes to reduce the 'bandwidth' of the matrix, and thereby the amount of fill generated by the factorisation.

7.2 Cuthill-McKee ordering

A popular ordering strategy is the Cuthill-McKee ordering, which finds levels or wavefronts in the matrix graph. This algorithm is easily described:

1. Take any node as starting point, and call that 'level 0'.

2. Now successively take all nodes connected to the previous level, and group them into the next level.

3. Iterate this until all nodes are grouped into some level; the numbering inside each level is of secondary importance.

This ordering strategy often gives a smaller bandwidth than the natural ordering, and there are further advantages to having a level structure, e.g., for out-of-core solution or for parallel processing. Often, one uses the 'reverse Cuthill-McKee' orderings [50].

7.3 Minimum degree

An explicit reduction of bandwidth is effected by the minimum degree ordering, which at any point in the factorisation chooses the variable with the smallest number of connections. Considering the size of the resulting fill-in is used as a tie-breaker.

7.4 Nested dissection

Instead of trying to minimise fill-in by reducing the bandwidth, one could try a direct approach. The 'nested dissection' ordering recursively splits the matrix graph in two, thus separating it into disjoint subgraphs. Somewhat more precisely, given a graph, this algorithm relies on the existence of a 'separator': a set of nodes such that the other nodes fall into two mutually unconnected subgraphs. The fill from first factoring these subgraphs, followed by a factorisation of the separator, is likely to be lower than for other orderings.

It can be shown that for PDEs in two space dimensions this method has a storage requirement that is within a log-factor of that for the matrix itself, that is, very close to optimal [33]. This proof is easy for PDEs on rectangular grids, but with enough graph theory it can be generalised [49, 48]. However, for problems in three space dimensions, the nested dissection method is no longer optimal.

An advantage of dissection-type methods is that they lead to large numbers of uncoupled matrix problems. Thus, to an extent, parallelisation of such methods is easy. However, the higher levels in the tree quickly have fewer nodes than the number of available processors. In addition to this, they are also the larger subproblems in the algorithm, thereby complicating the parallelisation of the method.

Another practical issue is the choice of the separator set. In a model case this is trivial, but in practice, and in particular in parallel, this is a serious problem, since the balancing of the two resulting subgraphs depends on this choice. Recently, so-called '2nd eigenvector methods' have become popular for this [56].

8 Iterative solution methods

Direct methods, as sketched above, have some pleasant properties. Foremost is the fact that their time to solution is predictable, either a priori, or after determining the matrix ordering. This is due to the fact that the method does not rely on numerical properties of the coefficient matrix, but only on its structure. On the other hand, the amount of fill can be substantial, and with it the execution time. For large scale applications, the storage requirements for a realistic size problem can simply be prohibitive.

Iterative methods have far lower storage demands. Typically, the storage, and the cost per iteration with it, is of the order of the matrix storage. However, the number of iterations strongly depends on properties of the linear system, and is at best known up to an order estimate; for difficult problems the methods may not even converge due to accumulated round-off errors.

8.1 Basic iteration procedure

In its most informal sense, an iterative method in each iteration locates an approximation to the solution of the problem, measures the error between the approximation and the true solution, and based on the error measurement improves on the approximation by constructing a next iterate. This process repeats until the error measurement is deemed small enough.

8.2 Stationary iterative methods

The simplest iterative methods are the 'stationary iterative methods'. They are based on finding a matrix M that is, in some sense, 'close' to the coefficient matrix A. Instead of solving $Ax = b$, which is deemed computationally infeasible, we solve $Mx_1 = b$. The true measure of how well x_1 approximates x is the error $e_1 = x_1 - x$, but, since we do not know the true solution x, this quantity is not computable. Instead, we look at the 'residual': $r_1 = Ae_1 = Ax_1 - b$, which is a computable quantity. One easily sees that the true solution satisfies $x = A^{-1}b = x_1 - A^{-1}r_1$, so, replacing A^{-1} with M^{-1} in this relation, we define $x_2 = x_1 - M^{-1}r_1$.

Stationary methods are easily analysed: we find that $r_i \to 0$ if all eigenvalues $\lambda = \lambda(I - AM^{-1})$ satisfy $|\lambda| < 1$. For certain classes of A and M this inequality is automatically satisfied [36, 61].

8.3 Krylov space methods

The most popular class of iterative methods nowadays is that of 'Krylov space methods'. The basic idea there is to construct the residuals such that the n-th residual r_n is obtained from the first by multiplication by some polynomial in the coefficient matrix A, that is,

$$r_n = P_{n-1}(A)r_1.$$

The properties of the method then follow from the properties of the actual polynomial [3, 7, 9].

Most often, these iteration polynomials are chosen such that the residuals are orthogonal under some inner product. From this, one usually obtains some minimisation property, though not necessarily a minimisation of the *error*.

Since the iteration polynomials are of increasing degree, it is easy to see that the main operation in each iteration is one matrix-vector multiplication. Additionally, some vector operations, including inner products in the orthogonalisation step, are needed.

8.3.1 The issue of symmetry

Krylov method residuals can be shown to satisfy the equation

$$r_n \in \operatorname{span}\{Ar_{n-1}, r_{n-1}, \ldots, r_1\}.$$

This brings up the question whether all $r_{n-1}, \ldots, r_1\}$ need to be stored in order to compute r_n. The answer is that this depends on the symmetry of the coefficient matrix. For a symmetric problem, the r_n vectors satisfy a three-term recurrence. This was the original Conjugate Gradient method [40].

For nonsymmetric problems, on the other hand, no short recurrences can exist [29], and therefore, all previous residuals need to be stored. Some of these methods are OrthoDir and OrthoRes [65].

If the requirement of orthogonality is relaxed, one can derive short-recurrence methods for nonsymmetric problems [31]. In the Biconjugate Gradient method, two sequences $\{r_n\}$ and $\{s_n\}$ are derived that are mutually orthogonal, and that satisfy three-term recurrences.

A disadvantage of this latter method is that it needs application of the transpose of the coefficient matrix. In environments where the matrix is only operatively defined, this may exclude this method from consideration. Recently developed methods, mostly based on the work of [59, 60], obviate this consideration.

8.3.2 True minimisation

The methods mentioned so far minimise the error (over the subspace generated) in some matrix-related norm, but not in the Euclidean norm. We can effect a true minimisation by collecting the residuals generated so far, and finding a minimising convex combination. This leads to one of the most popular methods nowadays: GMRES [58]. It will always generate the optimal iterate, but for this it requires storage of all previous residuals. In practice, truncated or restarted version of GMRES are popular.

8.4 Preconditioners

The matrix M that appeared in the section on stationary iterative methods can play a role in Krylov space methods too. There, it is called a ‘preconditioner’, and it acts to improve spectral properties of the coefficient matrix that determine the convergence speed of the method. In a slight simplification, one might say that we replace the system $Ax = b$ by

$$(AM^{-1})(Mx) = b.$$

(Additionally, the inner product is typically changed.) It is generally recognised that a good preconditioner is crucial to the performance of an iterative method.

The requirements on a preconditioner are that it should be easy to construct, a system $Mx = b$ should be simple to solve, and in some sense M should be an approximation to A. These requirements need to be balanced: a more accurate preconditioner is usually harder to construct and more costly to apply, so any decrease in the number iterations has to be set against a longer time per iteration, plus an increased setup phase.

The holy grail of preconditioners is finding an 'optimal' preconditioner: one for which the number of operations for applying it is of the order of the number of variables, while the resulting number of iterations is bounded in the problem size. There are very few optimal preconditioners.

8.4.1 Simple preconditioners

Some preconditioners need no construction at all. For instance, the Jacobi preconditioner consists of simply the matrix diagonal D_A. Since in PDE applications the largest elements are on the diagonal, one expects some degree of accuracy from this. Using not just the diagonal, but the whole lower triangular part $D_A + L_A$ of the coefficient matrix, an even more accurate method results. Since this triangular matrix is nonsymmetric, it is usually balanced with the upper triangular part as $(D_A + L_A)D_A^{-1}(D_A + U_A)$.

8.4.2 Incomplete factorisations

A successful strategy for preconditioners results from mimicking direct methods, but applying some approximation process to them. Thus, the so-called 'incomplete factorisation' methods ignore fill elements in the course of the Gaussian elimination process. Two strategies are to ignore elements in fixed positions, or to drop elements that are deemed small enough to be negligible. The aim is here to preserve at least some of the sparsity of the coefficient matrix in the factorisation, while giving something that is close enough to the full factorisation.

Incomplete factorisations can be very effective, but there are a few practical problems. For the class of M-matrices, these methods are well-defined [52], but for other, even fairly common classes of matrices, there is a possibility that the algorithm breaks down [42, 45, 51].

Also, factorisations are inherently recursive, and coupled with the sparseness of the incomplete factorisation, this gives very limited parallelism in the algorithm using a natural ordering of the unknowns. Different orderings may be more parallel, but take more iterations [25, 27, 43].

8.4.3 Analytically inspired preconditioners

In recent years, a number of preconditioners have gained in popularity that are more directly inspired by the continuous problem. First of all, for a matrix from an elliptic PDE, one can use a so-called 'fast solver' as preconditioner [12, 28, 63].

A particularly popular class of preconditioners based on the continuous problem, is that of 'domain decomposition' methods. If the continuous problem was elliptic, then decomposing the domain into simply connected pieces leads to elliptic problems on these subdomains, tied together by internal boundary conditions of some sort.

For instance, in the Schur complement domain decomposition method [8], thin strips of variables are assigned a function as interface region, and the original problem reduces to fully independent problems on the subdomains, connected by a system on the interface that is both smaller and better conditioned, but more dense, than the original one. While the subdomains can trivially be executed in parallel, the interface system poses considerable problems.

Choosing overlapping instead of separated subdomains leads to the class of Schwarz methods [47]. The original Schwarz method on two domains proposed solving one subdomain, deriving interface conditions from it for the other subdomain, and solving the system there. Repetition of this process can be shown to converge. In a more parallel variant of this method, all subdomains solve their system simultaneously, and the solutions on the overlap regions are added together.

Multilevel methods do not operate by decomposing the domain. Rather, they work on a sequence of nested discretisation, solving the coarser ones as a starting point for solving the finer levels. Under certain conditions such methods can be shown to be close to optimal [4, 35]. However, they require explicit knowledge of the operator and boundary conditions. For this reason, people have investigated algebraic variants [5, 57]. In both cases, these methods can be parallelised by distributing each level over the processors, but this may not be trivial.

9 Libraries and standards in sparse methods

Unlike in dense methods, there are few standards for iterative methods. Most of this is due to the fact that sparse storage is more complicated, admitting of more variation, and therefore less standardised. Whereas the (dense) BLAS has been accepted for a long time, sparse BLAS is not more than a proposal under research.

9.1 Storage formats

As is apparent from the matrix-vector example in section 6.2, storage formats for sparse matrices include not just the matrix elements, but pointer information describing where the nonzero elements are placed in the matrix. A few storage formats are in common use (for more details see [6]):

Aij format In the 'Aij' format, three arrays of the same length are allocated: one containing the matrix elements, and the other two containing the i and j coordinates of these elements. No particular ordering of the elements is implied.

Row/column-compressed In the row-compressed format one array of integers is allocated in addition to the matrix element, giving the column indices of the nonzero elements. Since all elements in the same row are stored contiguously, a second, smaller, array is needed giving the start points of the rows in the two larger arrays.

Compressed diagonal If the nonzero elements of the matrix are located, roughly or exactly, along subdiagonals, one could use contiguous storage for these diagonals. There are several diagonal storage formats. In the simplest, describing a contiguous block of subdiagonals, only the array of matrix elements is needed; two integers are sufficient to describe which diagonals have been stored.

There exist blocked versions of these formats, for matrices that can be partitioned into small square subblocks.

9.2 Sparse libraries

Since sparse formats are more complicated than dense matrix storage, sparse libraries have an added level of complexity. This holds even more so in the parallel case, where additional indexing information is needed to specify which matrix elements are on which processor.

There are two fundamentally different approaches for handling this complexity. Some sparse libraries require the user to set up the matrix and supply it to the library, while all handling is performed by the library. This requires the user to store data in a format dictated by the library, which might involve considerable work.

On the other hand, the library might do even the matrix setup internally, hiding all data from the user. This gives total freedom to the user, but it requires the library to supply sufficient access functions so that the user can perform certain matrix operations, even while not having access to the object itself.

References

[1] E. Anderson and J. Dongarra. Results from the initial release of LAPACK. Technical Report LAPACK working note 16, Computer Science Department, University of Tennessee, Knoxville, TN, 1989. `http://www.netlib.org/lapack/lawns/lawn16.ps`.

[2] E. Anderson and J. Dongarra. Evaluating block algorithm variants in LAPACK. Technical Report LAPACK working note 19, Computer Science Department, University of Tennessee, Knoxville, TN, 1990. `http://www.netlib.org/lapack/lawns/lawn19.ps`.

[3] O. Axelsson and A.V. Barker. *Finite element solution of boundary value problems. Theory and computation.* Academic Press, Orlando, Fl., 1984.

[4] O. Axelsson and P. Vassilevski. Algebraic multilevel preconditioning methods, I. *Numer. Math*, 56:157–177, 1989.

[5] Owe Axelsson and Victor Eijkhout. The nested recursive two-level factorization method for nine-point difference matrices. *SIAM J. Sci. Stat. Comput.*, 12:1373–1400, 1991.

[6] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, Philadelphia PA, 1994. `http://www.netlib.org/templates/templates.ps`.

[7] Garrett Birkhoff and Robert E. Lynch. *Numerical solution of elliptic problems.* SIAM, Philadelphia, 1984.

[8] P. Bjørstad and O. Widlund. Iterative methods for the solution of elliptic problems on regions partitioned in to substructures. *SIAM J. Numer. Anal.*, 23:1097–1120, 1986.

[9] Tony Chan and Henk van der Vorst. Linear system solvers: Sparse iterative methods. In D. Keyes et al, editor, *Parallel Numerical Algorithms, Proc. of the ICASW/LaRC Workshop on Parallel Numerical Algorithms, May 23-25, 1994*, pages 91–118. Kluwer Publ., Dordrecht, The Netherlands, 1997.

[10] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. IEEE Computer Society Press, 1992.

[11] J. Choi, J. J. Dongarra, and D. W. Walker. The design of scalable software libraries for distributed memory concurrent computers. In J. J. Dongarra and B. Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing.* Elsevier Science Publishers, 1993.

[12] Paul Concus and Gene H. Golub. Use of fast direct methods for the efficient numerical solution of nonseparable elliptic equations. *SIAM J. Numer. Anal.*, 10:1103–1120, 1973.

[13] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Introduction to Split-C: Version 0.9. Technical report, Computer Science Division – EECS, University of California, Berkeley, CA 94720, February 1993.

[14] J. Demmel. LAPACK: A portable linear algebra library for supercomputers. In *Proceedings of the 1989 IEEE Control Systems Society Workshop on Computer-Aided Control System Design*, December 1989.

[15] J. J. Dongarra. Increasing the performance of mathematical software through high-level modularity. In *Proc. Sixth Int. Symp. Comp. Methods in Eng. & Applied Sciences, Versailles, France*, pages 239–248. North-Holland, 1984.

[16] J. J. Dongarra. LAPACK Working Note 34: Workshop on the BLACS. Computer Science Dept. Technical Report CS-91-134, University of Tennessee, Knoxville, TN, May 1991. `http://www.netlib.org/lapack/lawns/lawn16.ps`.

[17] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[18] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An extended set of Fortran basic linear algebra subroutines. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988.

[19] J. J. Dongarra and E. Grosse. Distribution of mathematical software via electronic mail. *Communications of the ACM*, 30(5):403–407, July 1987.

[20] J. J. Dongarra, Peter Mayes, and Giuseppe Radicati di Brozolo. The IBM RISC System/6000 and linear algebra operations. *Supercomputer*, 44(VIII-4):15–30, 1991.

[21] J. J. Dongarra, R. Pozo, and D. W. Walker. An object oriented design for high performance linear algebra on distributed memory architectures. In *Proceedings of the Object Oriented Numerics Conference*, 1993.

[22] J. J. Dongarra and R. A. van de Geijn. Two-dimensional basic linear algebra communication subprograms. Technical Report LAPACK working note 37, Computer Science Department, University of Tennessee, Knoxville, TN, October 1991.

[23] J.J. Dongarra, F.C. Gustavson, and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26:91–112, 1984.

[24] J. Du Croz and M. Pont. The development of a floating-point validation package. In M. J. Irwin and R. Stefanelli, editors, *Proceedings of the 8th Symposium on Computer Arithmetic, Como, Italy, May 19-21, 1987*. IEEE Computer Society Press, 1987.

[25] I.S. Duff and G.A. Meurant. The effect of ordering on preconditioned conjugate gradients. *BIT*, 29:635–657, 1989.

[26] A. Edelman. Large dense numerical linear algebra in 1993: The parallel computing influence. *International Journal Supercomputer Applications*, 7:113–128, 1993.

[27] Victor Eijkhout. Analysis of parallel incomplete point factorizations. *Lin. Alg. Appl.*, 154–156:723–740, 1991.

[28] Howard C. Elman and Martin H. Schultz. Preconditioning by fast direct methods for non self-adjoint nonseparable elliptic equations. *SIAM J. Numer. Anal.*, 23:44–57, 1986.

[29] V. Faber and T. Manteuffel. Orthogonal error methods. *SIAM J. Numer. Anal.*, 24:170–187, 1987.

[30] E. W. Felten and S. W. Otto. Coherent parallel C. In G. C. Fox, editor, *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 440–450. ACM Press, 1988.

[31] R. Fletcher. Conjugate gradient methods for indefinite systems. In G.A. Watson, editor, *Numerical Analysis Dundee 1975*, pages 73–89, New York, 1976. Springer Verlag.

[32] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice Hall, Englewood Cliffs, N.J., 1988.

[33] Alan George and Joseph H-W. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1981.

[34] A. Gupta and V. Kumar. On the scalability of FFT on parallel computers. In *Proceedings of the Frontiers 90 Conference on Massively Parallel Computation.* IEEE Computer Society Press, 1990. Also available as technical report TR 90-20 from the Computer Science Department, University of Minnesota, Minneapolis, MN 55455.

[35] W. Hackbusch. *Multi-Grid Methods and Applications.* Springer-Verlag, 1985.

[36] Louis A. Hageman and David M. Young. *Applied Iterative Methods.* Academic Press, New York, 1981.

[37] R. Harrington. Origin and development of the method of moments for field computation. *IEEE Antennas and Propagation Magazine*, June 1990.

[38] J. L. Hess. Panel methods in computational fluid dynamics. *Annual Reviews of Fluid Mechanics*, 22:255–274, 1990.

[39] J. L. Hess and M. O. Smith. Calculation of potential flows about arbitrary bodies. In D. Küchemann, editor, *Progress in Aeronautical Sciences, Volume 8.* Pergamon Press, 1967.

[40] M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Nat. Bur. Stand. J. Res.*, 49:409–436, 1952.

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

[42] A. Jennings and G.M. Malik. Partial elimination. *J. Inst. Maths Applics*, 20:307–316, 1977.

[43] M.T. Jones and P.E. Plassmann. Parallel solution of unstructed, sparse systems of linear equations. In R.F. Sincovec, D.E. Keyes, M.R. Leuze, L.R. Petzold, and D.A. Reed, editors, *Proceedings of the Sixth SIAM conference on Parallel Processing for Scientific Computing*, pages 471–475, Philadelphia. SIAM.

[44] W. Kahan. Paranoia. Available from netlib [19]: `http://www.netlib.org/paranoia`.

[45] D.S. Kershaw. The incomplete cholesky-conjugate gradient method for the iterative solution of systems of linear equations. *J. Comp. Phys.*, 26:43–65, 1978.

[46] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw.*, 5:308–323, 1979.

[47] P.L. Lions. On the Schwarz alternating method. i. In Roland Glowinski, Gene H. Golub, Gérard Meurant, and Jacques Periaux, editors, *Domain Decomposition Methods for Partial Differential Equations, proceedings of the First Internation Symposium, Paris, January 7–9, 1987*, pages 1–42, Philadelphia, 1988. SIAM.

[48] Richard J. Lipton, Donald J. Rose, and Robert Endre Tarjan. Generalized nested dissection. *SIAM J. Numer. Anal.*, 16:346–358, 1979.

[49] Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM J. Appl. Math.*, 36:177–189, 1979.

[50] Joseph W-H Liu and Andrew H. Sherman. Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices. *SIAM J. Numer. Anal.*, 13:198–213, 1973.

[51] T.A. Manteuffel. An incomplete factorization technique for positive definite linear systems. *Math. Comp.*, 34:473–497, 1980.

[52] J.A. Meijerink and H.A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric m-matrix. *Math Comp*, 31:148–162, 1977.

[53] J.M. Ortega. The ijk forms of factorization methods I. Vector computers. *Parallel Computing*, 7:135–147, 1988.

[54] J.M. Ortega and C.H. Romine. The ijk forms of factorization methods II. Parallel systems. *Parallel Computing*, 7:149–162, 1988.

[55] Seymour V. Parter. The use of linear graphs in gaussian elimination. *SIAM Review*, 3:119–130, 1961.

[56] Alex Pothen, Horst D. Simon, and Kang-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal of Matrix Analysis and Applications*, 11(3):430–452, July 1990.

[57] J.W. Ruge and K. Stüben. Algebraic multigrid. In Stephen F. McCormick, editor, *Multigrid Methods*. SIAM, 1987. chapter 4.

[58] Yousef Saad and Martin H. Schultz. GMRes: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7:856–869, 1986.

[59] Peter Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 10:36–52, 1989.

[60] Henk van der Vorst. Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13:631–644, 1992.

[61] Richard S. Varga. *Matrix Iterative Analysis*. Prentice-Hall Inc., Englewood Cliffs, NJ, 1962.

[62] J. J. H. Wang. *Generalized Moment Methods in Electromagnetics*. John Wiley & Sons, New York, 1991.

[63] O. Widlund. On the use of fast methods for separable finite difference equations for the solution of general elliptic problems. In D.J. Rose and R.A. Willoughby, editors, *Sparse matrices and their applications*, pages 121–134. Plenum Press, New York, 1972.

[64] J. Wilkinson and C. Reinsch. *Handbook for Automatic Computation: Volume II - Linear Algebra*. Springer-Verlag, New York, 1971.

[65] David M. Young and Kang C. Jea. Generalized conjugate-gradient acceleration of nonsymmetrizable iterative methods. *Lin. Alg. Appl.*, 34:159–194, 1980.