LAPACK Working Note 139

A Numerical Linear Algebra Problem Solving Environment Designer's Perspective

A. Petitet[*], H. Casanova[*], J. Dongarra[†], Y. Robert[‡] and R. C. Whaley[*]

October, 1998

[*]Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301

[†]Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301, and Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831

[‡]Ecole Normale Supérieure de Lyon, 69364 Lyon Cedex 07, France

Abstract

This chapter discusses the design of modern numerical linear algebra problem solving environments. Particular emphasis is placed on three essential components out of which such environments are constructed, namely well-designed numerical software libraries, software tools that generate optimized versions of a collection of numerical kernels for various processor architectures, and software systems that transform disparate, loosely-connected computers and software libraries into a unified, easy-to-access computational service.

A brief description of the "pioneers", namely the EISPACK and LINPACK software libraries as well as their successor, the Linear Algebra PACKage (LAPACK), illustrates the essential importance of block-partitioned algorithms for shared-memory, vector, and parallel processors. Indeed, these algorithms reduce the frequency of data movement between different levels of hierarchical memory. A key idea in this approach is the use of the Basic Linear Algebra Subprograms (BLAS) as computational building blocks. An outline of the ScaLAPACK software library, which is a distributed-memory version of LAPACK, highlights the equal importance of the above design principles to the development of scalable algorithms for MIMD distributed-memory concurrent computers. The impact of the architecture of high performance computers on the design of such libraries is stressed.

Producing hand-optimized implementations of even a reduced set of well designed software components such as the BLAS for a wide range of architectures is an expensive and tedious proposition. For any given architecture, customizing a numerical kernel's source code to optimize performance requires a comprehensive understanding of the exploitable hardware resources of that architecture. Since this time-consuming customization process must be repeated whenever a slightly different target architecture is available, the relentless pace of hardware innovation makes the tuning of numerical libraries a constant burden. This chapter presents an innovative approach to automating the process of producing such optimized kernels for various processor architectures.

Finally, many scientists and researchers increasingly tend nowadays to use simultaneously a variety of distributed computing resources such as massively parallel processors, networks and clusters of workstations and "piles" of PCs. This chapter describes the NetSolve software system that has been specifically designed and conceived to efficiently use such a diverse and lively computational environment and to tackle the problems posed by such a complex and innovative approach to scientific problem solving. NetSolve provides the user with a pool of computational resources. These resources are computational servers that provide run-time access to arbitrary optimized numerical software libraries. This unified, easy-to-access computational service can make enormous amounts of computing power transparently available to users on ordinary platforms.

1 Introduction

The increasing availability of advanced-architecture computers is having a very 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. In this chapter, particular attention will be paid to dense general linear system solvers, and these will be used as examples to highlight the most important factors that must be considered in designing linear algebra software for advanced-architecture computers. We use these general linear system solving algorithms for illustrative purpose not only because they are relatively simple, but also because of their importance in several scientific and engineering applications [Ede93] that make use of boundary element methods. These applications include for instance electromagnetic scattering [Har90, Wan91] and computational fluid dynamics problems [Hes90, HS67].

This chapter discusses some of the recent developments in linear algebra software designed to exploit these advanced-architecture computers. Since most of the work is motivated by the need to solve large problems on the fastest computers available, we focus on three essential components out of which current and modern problem solving environments are constructed:

1. well-designed numerical software libraries providing a comprehensive functionality and confining most machine dependencies into a small number of kernels, that offer a wide scope for efficiently exploiting computer hardware resources,
2. automatic generation and optimization of such a collection of numerical kernels on various processor architectures, that is, software tools enabling well-designed software libraries to achieve high performance on most modern computers in a transportable manner,
3. software systems that transform disparate, loosely-connected computers and software libraries into a unified, easy-to-access computational service, that is, a service able to make enormous amounts of computing power transparently available to users on ordinary platforms.

For the past twenty years or so, there has been a great deal of activity in the area of algorithms and software for solving linear algebra problems. The linear algebra community has long recognized the need for help in developing algorithms into software libraries, and several years ago, as a community effort, put together a *de facto* standard identifying basic operations required in linear algebra algorithms and software. The hope was that the routines making up this standard, known collectively as the Basic Linear Algebra Subprograms (BLAS) [LHK+79, DDH+88, DDH+90], would be efficiently implemented on advanced-architecture computers by many manufacturers, making it possible to reap the portability benefits of having them efficiently implemented on a wide range of machines. This goal has been largely realized.

The key insight of our approach to designing linear algebra algorithms for advanced-architecture computers is that the frequency with which data is 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. In general, the use of block-partitioned algorithms requires data to be moved as blocks, rather than as vectors or scalars, so that although the total amount of data moved is unchanged, the latency (or startup cost) associated with the movement is greatly reduced because fewer messages are needed to move the data. A second key idea is that the performance of an algorithm can be tuned by a user by varying the parameters that specify the data layout. On shared-memory machines, this is controlled by the block size, while on distributed-memory machines it is controlled by the block size and the configuration of the logical process mesh.

Section 2 presents an overview of some of the major numerical linear algebra software library projects aimed at solving dense and banded problems. We discuss the role of the BLAS in portability and performance on high-performance computers as well as the design of these building blocks, and their use in block-partitioned algorithms.

The Linear Algebra PACKage (LAPACK) [ABB+95], for instance, is a typical example of such a software design, where most of the algorithms are expressed in terms of a reduced set of computational building blocks, in this case called the Basic Linear Algebra Subprograms (BLAS). These computational building blocks support the creation of software that efficiently expresses higher-level block-partitioned algorithms, while hiding many details of the parallelism from the application developer. These subprograms can be optimized for each architecture to account for the deep memory hierarchies [AD89, DMR91] and pipelined functional units that are common to most modern computer architectures, and thus provide a transportable way to achieve high efficiency across diverse computing platforms. For fastest possible performance, LAPACK requires that highly optimized block matrix operations be already implemented on each machine, that is, the correctness of the code is portable, but high performance is not—if we limit ourselves to a single source code.

Speed and portable optimization are thus conflicting objectives that have proved difficult to satisfy simultaneously, and the typical strategy for addressing this problem by confining most of the hardware dependencies in a small number of heavily-used computational kernels has limitations. For instance, producing hand-optimized implementations of even a reduced set of well-designed software components for a wide range of architectures is an expensive and tedious task. For any given architecture, customizing a numerical kernel's source code to optimize performance requires a comprehensive understanding of the exploitable hardware resources of that architecture. This primarily includes the memory hierarchy and how it can be utilized to maximize data-reuse, as well as the functional units and registers and how these hardware components can be programmed to generate the correct operands at the correct time. Clearly, the size of the various cache levels, the latency of floating point instructions, the number of floating point units and other hardware constants are essential parameters that must be taken into consideration as well. Since this time-consuming customization process must be repeated whenever a slightly different target architecture is available, or even when a new version of the compiler is released, the relentless pace of hardware innovation makes the tuning of numerical libraries a constant burden.

The difficult search for fast and accurate numerical methods for solving numerical linear algebra problems is compounded by the complexities of porting and tuning numerical libraries to run on the best hardware available to different parts of the scientific and engineering community. Given the fact that the performance of common computing platforms has increased exponentially in the past few years, scientists and engineers have acquired legitimate expectations about being able to immediately exploit these available resources at their highest capabilities. Fast, accurate, and robust numerical methods have to be encoded in software libraries that are highly portable and optimizable across a wide range of systems in order to be exploited to their fullest potential.

Section 3 discusses an innovative approach [BAC+97, WD97] to automating the process of producing such optimized kernels for RISC processor architectures that feature deep memory hierarchies and pipelined functional units. These research efforts have so far demonstrated very encouraging results, and have generated great interest among the scientific computing community.

Many scientists and researchers increasingly tend nowadays to use simultaneously a variety of distributed computing resources such as massively parallel processors, networks and clusters of workstations and "piles" of PCs. In order to use efficiently such a diverse and lively computational environment, many challenging research aspects of network-based computing such as fault-tolerance, load balancing, user-interface design, computational servers or virtual libraries, must be addressed. User-friendly, network-enabled, application-specific toolkits have been specifically designed and conceived to tackle the problems posed by such a complex and innovative approach to scientific problem solving [FK98]. Section 4 describes the NetSolve software system [CD95] that provides users with a pool of computational resources. These resources are computational servers that provide run-time access to arbitrary optimized numerical software libraries. The NetSolve software system transforms disparate, loosely-connected computers and software libraries into a unified, easy-to-access computational service. This service can make enormous amounts of computing power transparently available to users on ordinary platforms.

The NetSolve system allows users to access computational resources, such as hardware and software, distributed across the network. These resources are embodied in computational servers and allow users to easily perform scientific computing tasks without having any computing facility installed on their computer. Users' access to the servers is facilitated by a variety of interfaces: Application Programming Interfaces (APIs), Textual Interactive Interfaces and Graphical User Interfaces (GUIs). As the NetSolve project matures, several promising extensions and applications will emerge. In this chapter, we provide an overview of the project and examine some of the extensions being developed for NetSolve: an interface to the Condor system [LLM88], an interface to the ScaLAPACK parallel library [BCC+97], a bridge with the Ninf system [SSN+96], and an integration of NetSolve and ImageVision [ENB96].

Future directions for research and investigation are finally presented in Section 5.

2 Numerical Linear Algebra Libraries

This section first presents a few representative numerical linear algebra packages in a chronological perspective. We then focus on the software design of the LAPACK and ScaLAPACK software libraries. The importance of the BLAS as a key to (trans)portable efficiency as well as the derivation of block-partitioned algorithms are discussed in detail.

2.1 Chronological Perspective

The EISPACK, LINPACK, LAPACK and ScaLAPACK numerical linear algebra software libraries are briefly outlined below in a chronological order. The essential features of each of these packages are in turn rapidly described in order to illustrate the reasons for this evolution. Particular emphasis is placed on the impact of the high-performance computer architecture on the design features of these libraries.

2.1.1 The Pioneers: EISPACK and LINPACK

The EISPACK and LINPACK software libraries were designed for supercomputers used in the seventies and early eighties, such as the CDC-7600, Cyber 205, and Cray-1. These machines featured multiple functional units pipelined for good performance [HJ81]. The CDC-7600 was basically a high-performance scalar computer, while the Cyber 205 and Cray-1 were early vector computers.

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 sixties and collected by J. H. Wilkinson and C. Reinsch in a volume entitled *Linear Algebra* in the *Handbook for Automatic Computation* [WR71] 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.

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. Some of these factorizations will be treated at greater length later, but, first a digression on organization and factors influencing LINPACK's efficiency is necessary.

LINPACK uses column-oriented algorithms to increase efficiency by preserving locality of reference. This means that if a program references an item in a particular block, the next reference is likely to be in the same block. 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. 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 column's length. The effects of column orientation are quite dramatic: on systems with virtual or cache memories, the LINPACK codes will significantly outperform codes that are not column oriented.

Another important influence on the efficiency of LINPACK is the use of the Level 1 BLAS [LHK+79]. These BLAS are a small set of routines that may be coded to take advantage of the special features of the computers on which LINPACK is being run. For most computers, this simply means producing machine-language versions. However, the code can also take advantage of more exotic architectural features, such as vector operations. Further details about the BLAS are presented below in Section 2.2.1.

2.1.2 LAPACK

The development of LAPACK [ABB+95] in the late eighties was intended to make the EISPACK and LINPACK libraries run efficiently on shared-memory vector supercomputers. LAPACK [Dem89] 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, along with 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. LAPACK is in the public domain and available from *netlib* [DG87].

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 [AD90, Dem89]. These block operations can be optimized for each architecture to account for the memory hierarchy [AD89, DMR91], 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 [LHK+79], LAPACK was designed at the outset to exploit the Level 3 BLAS [DDH+90] — 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.

2.1.3 ScaLAPACK

The ScaLAPACK [BCC+97] software library is extending the LAPACK library to run scalably on MIMD distributed-memory concurrent computers. 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 parallel (distributed-memory) versions of the BLAS (PBLAS) [CDO+95], and a set of Basic Linear Algebra Communication Subprograms (BLACS) [WD95] for communication tasks that arise frequently in parallel linear algebra computations. In the ScaLAPACK routines, all interprocessor communication occurs within the PBLAS and the BLACS, so that the source code of the top software layer of ScaLAPACK looks very similar to that of LAPACK.

The ScaLAPACK library contains routines for the solution of systems of linear equations, linear least squares problems and eigenvalue problems. The goals of the LAPACK project, which continue into the ScaLAPACK project, are efficiency so that the computationally intensive routines execute as fast as possible; scalability as the problem size and number of processors grow; reliability, including the return of error bounds; portability across machines; flexibility so that users may construct new routines from well designed components; and ease of use. Towards this last goal the ScaLAPACK software has been designed to look as much like the LAPACK software as possible.

Many of these goals have been attained by developing and promoting standards, especially specifications for basic computational and communication routines. Thus LAPACK relies on the BLAS [LHK+79, DDH+88, DDH+90], particularly the Level 2 and 3 BLAS for computational efficiency, and ScaLAPACK [BCC+97] relies upon the BLACS [WD95] for efficiency of communication and uses a set of parallel BLAS, the PBLAS [CDO+95], which themselves call the BLAS and the BLACS. LAPACK and ScaLAPACK will run on any machines for which the BLAS and the BLACS are available. A PVM [GBD+94] version of the BLACS has been available for some time and the portability of the BLACS has recently been further increased by the development of a version that uses MPI [MPI+94, SOH+96].

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 imposes a sufficiently large startup cost on the movement of data between different levels in the hierarchy, and for which the cost of a context switch is too great to make fine grain size multithreading worthwhile. The target machines are, therefore, medium and large grain size advanced-architecture computers. These include respectively "traditional" shared-memory vector supercomputers, such as the Cray Y-MP and C90, and MIMD distributed-memory concurrent computers, such as massively parallel processors (MPPs) and networks or clusters of workstations.

The ScaLAPACK software has been designed specifically to achieve high efficiency for a wide range of modern distributed-memory computers. Examples of such computers include the Cray T3 series, the IBM Scalable POWERparallel SP series, the Intel iPSC and Paragon computers, the nCube-2/3 computer, networks and clusters of workstations (NoWs and CoWs), and "piles" of PCs (PoPCs).

Future advances in compiler and hardware technologies in the mid to late nineties are expected to make multithreading a viable approach for masking communication costs. Since the blocks in a block-partitioned algorithm can be handled by separate threads, our approach will still be applicable on machines that exploit medium and coarse grain size multithreading.

2.2 Software Design

Developing a library of high-quality subroutines for dense linear algebra computations requires to tackle a large number of issues. On one hand, the development or selection of numerically stable algorithms in order to estimate the accuracy and/or domain of validity of the results produced by these routines. On the other hand, it is often required to (re)formulate or adapt those algorithms for performance reasons that are related to the architecture of the target computers. This section presents three fundamental ideas to this effect that characterize the design of the LAPACK and ScaLAPACK software.

2.2.1 The BLAS as the Key to (Trans)portable Efficiency

At least three factors affect the performance of portable Fortran code:

1. **Vectorization.** 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, [Don84]). 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, and this is just what the algorithms require. However, when the algorithms are realized in straightforward Fortran 77 code, the performance may fall well short of the expected level, usually because vectorizing Fortran compilers fail to minimize the number of memory references—that is, the number of vector load and store operations.

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 scalar processor, 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 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 [FO88] and Split-C [CDG+93] do the message passing implicitly.

The question arises, "How can we achieve sufficient control over these three factors to obtain the levels of performance that machines can offer?" The answer is through use of the BLAS. There are now three levels of BLAS:

Level 1 BLAS [LHK+79]: for vector-vector operations ($y \leftarrow \alpha x + y$),

Level 2 BLAS [DDH+88]: for matrix-vector operations ($y \leftarrow \alpha Ax + \beta y$),

Level 3 BLAS [DDH+90]: for matrix-matrix operations ($C \leftarrow \alpha AB + \beta C$).

Here, A, B and C are matrices, x and y are vectors, and α and β are scalars.

Table 1: Speed (Mflops) of Level 2 and Level 3 BLAS Operations on a CRAY Y-MP. All matrices are of order 500; U is upper triangular.

Number of processors:	1	2	4	8
Level 2: $y \leftarrow \alpha Ax + \beta y$	311	611	1197	2285
Level 3: $C \leftarrow \alpha AB + \beta C$	312	623	1247	2425
Level 2: $x \leftarrow Ux$	293	544	898	1613
Level 3: $B \leftarrow UB$	310	620	1240	2425
Level 2: $x \leftarrow U^{-1}x$	272	374	479	584
Level 3: $B \leftarrow U^{-1}B$	309	618	1235	2398
Peak	333	666	1332	2664

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. 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. Machines such as the CRAY Y-MP can perform two loads, a store, and a multiply-add operation all in one cycle, whereas the CRAY-2 and IBM 3090 VF cannot. For further details of how the performance of the BLAS are affected by such factors see [DDS+91]. 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 1 illustrates.

2.3 Block Algorithms and Their Derivation

It is comparatively straightforward to recode many of the algorithms in LINPACK and EISPACK so that they call 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 LU factorization algorithm, which factorizes a general matrix A in the product of the triangular factors L and U.

Suppose the $M \times N$ matrix A is partitioned as shown in Figure 1, and we seek a factorization $A = LU$, where the partitioning of L and U is also shown in Figure 1. Then we may write,

$$\begin{aligned} L_{00}U_{00} &= A_{00} \quad (1)\\ L_{10}U_{00} &= A_{10} \quad (2)\\ L_{00}U_{01} &= A_{01} \quad (3)\\ L_{10}U_{01} + L_{11}U_{11} &= A_{11} \quad (4) \end{aligned}$$

where A_{00} is $r \times r$, A_{01} is $r \times (N-r)$, A_{10} is $(M-r) \times r$, and A_{11} is $(M-r) \times (N-r)$. L_{00} and L_{11} are lower triangular matrices with ones on the main diagonal, and U_{00} and U_{11} are upper triangular matrices.

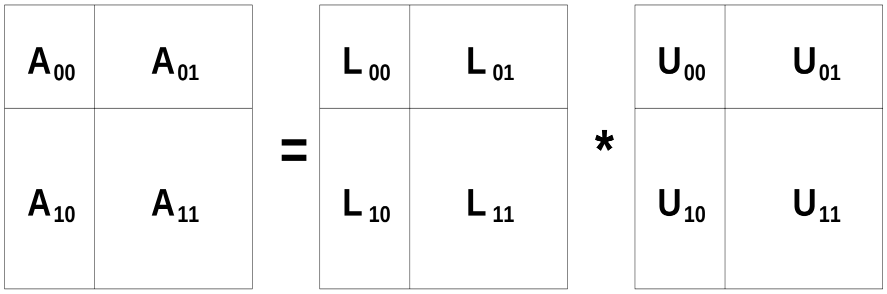

Figure 1: Block LU factorization of the partitioned matrix A. A_{00} is $r \times r$, A_{01} is $r \times (N-r)$, A_{10} is $(M-r) \times r$, and A_{11} is $(M-r) \times (N-r)$. L_{00} and L_{11} are lower triangular matrices with ones on the main diagonal, and U_{00} and U_{11} are upper triangular matrices.

Equations 1 and 2 taken together perform an LU factorization on the first $M \times r$ panel of A (i.e., A_{00} and A_{10}). Once this is completed, the matrices L_{00}, L_{10}, and U_{00} are known, and the lower triangular system in Eq. 3 can be solved to give U_{01}. Finally, we rearrange Eq. 4 as,

$$A'_{11} = A_{11} - L_{10}U_{01} = L_{11}U_{11} \quad (5)$$

From this equation we see that the problem of finding L_{11} and U_{11} reduces to finding the LU factorization of the $(M-r) \times (N-r)$ matrix A'_{11}. This can be done by applying the steps outlined above to A'_{11} instead of to A. Repeating these steps K times, where

$$K = \min\left(\lceil M/r \rceil, \lceil N/r \rceil\right), \quad (6)$$

and $\lceil x \rceil$ denotes the least integer greater than or equal to x, we obtain the LU factorization of the original $M \times N$ matrix A. For an in-place algorithm, A is overwritten by L and U – the ones on the diagonal of L do not need to be stored explicitly. Similarly, when A is updated by Eq. 5 this may also be done in place.

After k of these K steps, the first kr columns of L and the first kr rows of U have been evaluated, and the matrix A has been updated to the form shown in Figure 2, in which panel B is $(M-kr) \times r$ and C is $r \times (N-(k-1)r)$. Step $k+1$ then proceeds as follows,

1. factor B to form the next panel of L, performing partial pivoting over rows if necessary. This evaluates the matrices L_0, L_1, and U_0 in Figure 2,
2. solve the triangular system $L_0 U_1 = C$ to get the next row of blocks of U,
3. do a rank-r update on the trailing submatrix E, replacing it with $E' = E - L_1 U_1$.

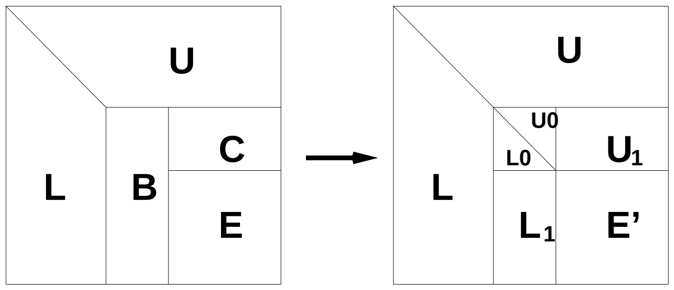

Figure 2: Stage $k+1$ of the block LU factorization algorithm showing how the panels B and C, and the trailing submatrix E are updated. The trapezoidal submatrices L and U have already been factored in previous steps. L has kr columns, and U has kr rows. In the step shown another r columns of L and r rows of U are evaluated.

The LAPACK implementation of this form of LU factorization uses the Level 3 BLAS to perform the triangular solve and rank-r update. We can regard the algorithm as acting on matrices that have been partitioned into blocks of $r \times r$ elements. No extra floating point operations nor extra working storage are required for simple block algorithms [DDS+91, GPS90].

2.4 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 and range-of-use.

2.4.1 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 [FJL+88], where the concurrent speedup is the execution time, $T_{\rm seq}$, 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,

$$\epsilon(N, N_p) = \frac{1}{N_p} \frac{T_{\text{seq}}(N)}{T(N, N_p)} \tag{7}$$

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,

$$G(N, N_p) = \frac{N_p}{t_{\text{calc}}} \epsilon(N, N_p) \tag{8}$$

where t_{calc} is the time for one floating point operation. Occasional examples where variation does occur are sometimes dismissed as "pathological cases". 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.

Table 2 illustrates the speed of the LAPACK routine for LU factorization of a real matrix, SGETRF in single precision on CRAY machines, and DGETRF in double precision on all other machines. Thus, 64-bit floating point arithmetic is used on all machines tested. A block size of one means that the unblocked algorithm is used, since it is faster than – or at least as fast as – a block algorithm. In all cases, results are reported for the block size which is mostly nearly optimal over the range of problem sizes considered.

Table 2: SGETRF/DGETRF speed (Mflops) for square matrices of order n

Machine	Block	Values of n				
(No. of processors)	size	100	200	300	400	500
IBM RISC/6000-530 (1)	32	19	25	29	31	33
Alliant FX/8 (8)	16	9	26	32	46	57
IBM 3090J VF (1)	64	23	41	52	58	63
Convex C-240 (4)	64	31	60	82	100	112
CRAY Y-MP (1)	1	132	219	254	272	283
CRAY-2 (1)	64	110	211	292	318	358
Siemens/Fujitsu VP 400-EX (1)	64	46	132	222	309	397
NEC SX2 (1)	1	118	274	412	504	577
CRAY Y-MP (8)	64	195	556	920	1188	1408

LAPACK [ABB+95] 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). LAPACK can also be used satisfactorily on all types of scalar machines (PCs, workstations, mainframes). The ScaLAPACK project, described in Section 2.1.3, adapts LAPACK to distributed-memory architectures.

A parallel algorithm is said to be scalable [GK90] 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. 8 shows that this is equivalent to saying that, for a scalable algorithm, the performance depends linearly on the number of processors for fixed granularity.

Figure 3 shows the scalability of the ScaLAPACK implementation of the *LU* factorization on the Intel XP/S Paragon computer. Figure 3 shows the speed in Mflops per node of the ScaLAPACK *LU* factorization routine for different computer configurations. This figure illustrates that when the number of nodes is scaled by a constant factor, the same efficiency or speed per node is achieved for equidistant problem sizes on a logarithmic scale. In other words, maintaining a constant memory use per node allows efficiency to be maintained. This scalability behavior is also referred to as *isoefficiency,* or *isogranularity.*) In practice, however, a slight degradation is acceptable. The ScaLAPACK driver routines, in general, feature the same scalability behavior up to a constant factor that depends on the exact number of floating point operations and the total volume of data exchanged during the computation. More information on ScaLAPACK performance can be found in [BCC+97, BW98].

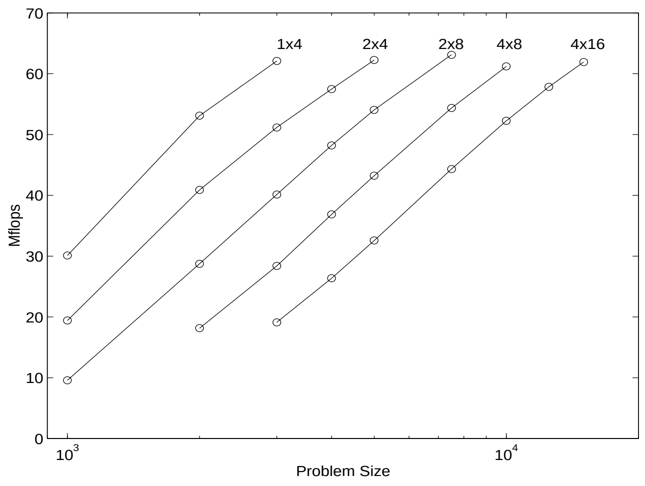

Figure 3: LU Performance per Intel XP/S MP Paragon node

2.4.2 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, 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* [DG87]. 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 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 [DP87, Kah87] 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 [DPW93]. In addition, software for distributed-memory computers should work correctly for a large class of data decompositions. The ScaLAPACK library has, therefore, adopted the block cyclic decomposition [BCC+97] for distributed-memory architectures.

2.4.3 Range-Of-Use

The 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. There are also sparse matrices, which may be stored in many different ways; but in this chapter we focus on dense and banded matrices, the mathematical types addressed by LINPACK, EISPACK, LAPACK and ScaLAPACK.

3 Automatic Generation of Tuned Numerical Kernels

This section describes an approach for the automatic generation and optimization of numerical software for processors with deep memory hierarchies and pipelined functional units. The production of such software for machines ranging from desktop workstations to embedded processors can be a tedious and time consuming customization process. The research efforts presented below aim at automating much of this process. Very encouraging results generating great interest among the scientific computing community have already been demonstrated. In this section, we focus on the ongoing Automatically Tuned Linear Algebra Software (ATLAS) [WD97] project developed at the University of Tennessee (see `http://www.netlib.org/atlas/`). The ATLAS initiative adequately illustrates current and modern research projects on automatic generation and optimization of numerical software such as PHiPAC [BAC+97]. After having developed the motivation for this research, the ATLAS methodology is outlined within the context of a particular BLAS function, namely the general matrix-multiply operation. Much of the technology and approach presented below applies to other BLAS and on basic linear algebra computations in general, and may be extended to other important kernel operations. Finally, performance results on a large collection of computers are presented and discussed.

3.1 Motivation

Straightforward implementation in Fortan or C of computations based on simple loops rarely achieve the peak execution rates of today's microprocessors. To realize such high performance for even the simplest of operations often requires tedious, hand-coded, programming efforts. It would be ideal if compilers where capable of performing the optimization needed automatically. However, compiler technology is far from mature enough to perform these optimizations automatically. This is true even for numerical kernels such as the BLAS on widely marketed machines which can justify the great expense of compiler development. Adequate compilers for less widely marketed machines are almost certain not to be developed.

Producing hand-optimized implementations of even a reduced set of well-designed software components for a wide range of architectures is an expensive proposition. For any given architecture, customizing a numerical kernel's source code to optimize performance requires a comprehensive understanding of the exploitable hardware resources of that architecture. This primarily includes the memory hierarchy and how it can be utilized to provide data in an optimum fashion, as well as the functional units and registers and how these hardware components can be programmed to generate the correct operands at the correct time. Using the compiler optimization at its best, optimizing the operations to account for many parameters such as blocking factors, loop unrolling depths, software pipelining strategies, loop ordering, register allocations, and instruction scheduling are crucial machine-specific factors affecting performance. Clearly, the size of the various cache levels, the latency of floating point instructions, the number of floating point units and other hardware constants are essential parameters that must be taken into consideration as well. Since this time-consuming customization process must be repeated whenever a slightly different target architecture is available, or even when a new version of the compiler is released, the relentless pace of hardware innovation makes the tuning of numerical libraries a constant burden.

The difficult search for fast and accurate numerical methods for solving numerical linear algebra problems is compounded by the complexities of porting and tuning numerical libraries to run on the best hardware available to different parts of the scientific and engineering community. Given the fact that the performance of common computing platforms has increased exponentially in the past few years, scientists and engineers have acquired legitimate expectations about being able to immediately exploit these available resources at their highest capabilities. Fast, accurate, and robust numerical methods have to be encoded in software libraries that are highly portable and optimizable across a wide range of systems in order to be exploited to their fullest potential.

For illustrative purpose, we consider the Basic Linear Algebra Subprograms (BLAS) described in Section 2.2.1. As shown in Section 2, the BLAS have proven to be very effective in assisting portable, efficient software for sequential, vector, shared-memory and distributed-memory high-performance computers. However, the BLAS are just a set of specifications for some elementary linear algebra operations. A reference implementation in Fortran 77 is publically available, but it is not expected to be efficient on any particular architecture, so that many hardware or software vendors provide an "optimized" implementation of the BLAS for specific computers. Hand-optimized BLAS are expensive and tedious to produce for any particular architecture, and in general will only be created when there is a large enough market, which is not true for all platforms. The process of generating an optimized set of BLAS for a new architecture or a slightly different machine version can be a time consuming and expensive process. Many vendors have thus invested considerable resources in producing optimized BLAS for their architectures. In many cases near optimum performance can be achieved for some operations. However, the coverage and the level of performance achieved is often not uniform across all platforms.

3.2 The ATLAS Methodology

In order to illustrate the ATLAS methodology, we consider the following matrix-multiply operation $C \leftarrow \alpha AB + \beta C$, where α and β are scalars, and A, B and C are matrices, with A an M-by-K matrix, B a K-by-N matrix and C an M-by-N matrix. In general, the arrays A, B, and C containing respectively the matrices A, B and C will be too large to fit into cache. It is however possible to arrange the computations so that the operations are performed with data for the most part in cache by dividing the matrices into blocks [DMR91]. ATLAS isolates the machine-specific features of the operation to several routines, all of which deal with performing an optimized "on-chip" matrix multiply, that is, assuming that all matrix operands fit in Level 1 (L1) cache. This section of code is automatically created by a code generator which uses timings to determine the correct blocking and loop unrolling factors to perform optimally. The user may directly supply the code generator with as much detail as desired, i.e. size of the L1 cache size, blocking factor(s) to try, etc; if such details are not provided, the code generator will determine appropriate settings via timings. The rest of the code produced by ATLAS does not change across architectures; it is presented in Section 3.2.1. It handles the looping and blocking necessary to build the complete matrix-matrix multiply from the on-chip multiply. The generation of the on-chip multiply routine is discussed in Section 3.2.2. It is obvious that with this many interacting effects, it would be difficult, if not impossible to predict a priori the best blocking factor, loop unrolling, etc. ATLAS provides a code generator coupled with a timer routine which takes in some initial information, and then tries different strategies for loop unrolling and latency hiding and chooses the case which demonstrated the best performance.

3.2.1 Building the General Matrix Multiply from the On-Chip Multiply

In this section, the routines necessary to build a general matrix-matrix multiply using a fixed-size on-chip multiply are described. Section 3.2.2 details the on-chip multiply and its code generator. For this section, it is enough to assume the availability of an efficient on-chip matrix-matrix multiply of the form $C \leftarrow A^T B$. This multiply is of fixed size, i.e. with all dimensions set to a system-specific value, N_B (M = N = K = N_B). Also available are several "cleanup" codes, which handle the cases caused by dimensions which are not multiples of the blocking factor.

The first decision to be taken by the general matrix multiply is whether the problem is large enough to benefit from our special techniques. The ATLAS algorithm requires copying of the operand matrices; if the problem is small enough, this $O(N^2)$ cost, along with miscellaneous overheads such as function calls and multiple layers of looping, can actually make the "optimized" general matrix multiply slower than the traditional three do loops. The size required for the $O(N^3)$ costs to dominate these lower order terms varies across machines, and so this switch point is automatically determined at installation time. For these very small problems, a standard three-loop multiply with some simple loop unrolling is called. This code will also be called if the algorithm is unable to dynamically allocate enough space to do the blocking (see below for further details).

Assuming the matrices are large enough, ATLAS presently features two algorithms for performing the general, off-chip multiply. The two algorithms correspond to different orderings of the main loops. In the first algorithm, the outer loop is over M, i.e., the rows of A and the second loop is over N, i.e., the columns of B. In the second algorithm, this order is reversed. The common dimension of A and B (i.e., the K loop) is currently always the innermost loop. Let us define the input matrix looped over by the outer loop as the outer or outermost matrix; the other input matrix will therefore be the inner or innermost matrix. In the first algorithm, A is thus the outer matrix and B is the inner matrix. Both algorithms have the option of writing the result of the on-chip multiply directly to the matrix, or to an output temporary $\hat{C}$. The advantages to writing to $\hat{C}$ rather than C are:

1. address alignment may be controlled (i.e., one can ensure during the dynamic memory allocation that one begins on a cache-line boundary),
2. Data is contiguous, eliminating possibility of unnecessary cache-thrashing due to ill-chosen leading dimension (assuming the cache is non-write-through).

The disadvantage of using $\hat{C}$ is that an additional write to C is required after the on-chip operations have completed. This cost is minimal if many calls to the on-chip multiply are made (each of which writes to either C or $\hat{C}$), but can add significantly to the overhead when this is not the case. In particular, an important application of matrix multiply is the rank-K update, where the write to the output array C can be a significant portion of the cost of the algorithm. Writing to $\hat{C}$ essentially doubles the write cost, which is unacceptable. The routines therefore employ a heuristic to determine if the number of times the on-chip multiply will be called in the K loop is large enough to justify using $\hat{C}$, otherwise the answer is written directly to C.

Regardless of which matrix is outermost, the algorithms try to dynamically allocate enough space to store the $N_B \times N_B$ output temporary, $\hat{C}$ (if needed), one panel of the outermost matrix, and the entire inner matrix. If this fails, the algorithms attempt to allocate enough space to hold $\hat{C}$, and one panel from both A and B. The minimum workspace required by these routines is therefore $2KN_B$, if writing directly to C, and ${N_B}^2 + 2KN_B$ if not. If this amount of workspace cannot be allocated, the previously mentioned small case code is called instead. If there is enough space to copy the entire innermost matrix, we see several benefits to doing so:

- Each matrix is copied only one time,
- If all of the workspaces fit into L2 cache, we get complete L2 reuse on the innermost matrix,
- Data copying is limited to the outermost loop, protecting the inner loops from unneeded cache thrashing.

Of course, even if the allocation succeeds, using too much memory might result in unneeded swapping. Therefore, the user can set a maximal amount of workspace that ATLAS is allowed to have, and ATLAS will not try to copy the innermost matrix if this maximum workspace requirement is exceeded.

If enough space for a copy of the entire innermost matrix is not allocated, the innermost matrix will be entirely copied for each panel of the outermost matrix, i.e. if A is the outermost matrix, the matrix B will be copied $\lceil M/N_B \rceil$ times. Further, the usable size of the Level 2 (L2) cache is reduced (the copy of a panel of the innermost matrix will take up twice the panel's size in L2 cache;

the same is true of the outermost panel copy, but that will only be seen the first time through the secondary loop). Regardless of which looping structure or allocation procedure used, the inner loop is always along K. Therefore, the operation done in the inner loop by both routines is the same, and it is shown in Figure 4.

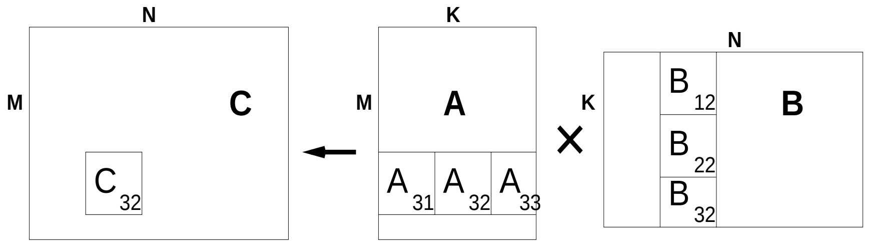

Figure 4: One step of the general matrix-matrix multiply

When a call to the matrix multiply is made, the routine must decide which loop structure to call (i.e., which matrix to put as outermost). If the matrices are of different size, L2 cache reuse can be encouraged by deciding the looping structure based on the following criteria:

- If either matrix will fit completely into L2 cache, put it as the innermost matrix (we get L2 cache reuse on the entire inner matrix),
- If neither matrix fits completely into L2 cache, put the one with the largest panel that will fit into L2 cache as the outermost matrix (we get L2 cache reuse on the panel of the outer matrix).

By default, the code generated by ATLAS does no explicit L2 blocking (the size of the L2 cache is not known anywhere in the code), and so these criteria are not presently used for this selection. Rather, if one matrix must be accessed by row-panels during the copy, that matrix will be put where it can be copied most efficiently. This means that if one has enough workspace to copy it up front, the matrix will be accessed column-wise by putting it as the innermost loop and copying the entire matrix; otherwise it will be placed as the outermost loop, where the cost of copying the row-panel is a lower order term. If both matrices have the same access patterns, B will be made the outermost matrix, so that C is accessed by columns.

3.2.2 Generation of the On-Chip Multiply

As previously mentioned, the ATLAS on-chip matrix-matrix multiply is the only code which must change depending on the platform. Since the input matrices are copied into blocked form, only one transpose case is required, which has been chosen as $C \leftarrow A^T B + C$. This case was chosen (as opposed to, for instance $C \leftarrow AB + C$), because it generates the largest (flops)/(cache misses) ratio possible when the loops are written with no unrolling. Machines with hardware allowing a smaller ratio can be addressed using loop unrolling on the M and N loops (this could also be addressed by permuting the order of the K loop, but this technique is not presently used in ATLAS.

In a multiply designed for L1 cache reuse, one of the input matrices is brought completely into the L1 cache, and is then reused in looping over the rows or columns of the other input matrix. The present ATLAS code brings in the array A, and loops over the columns of B; this was an arbitrary choice, and there is no theoretical reason it would be superior to bringing in B and looping over the rows of A. There is a common misconception that cache reuse is optimized when both input matrices, or all three matrices, fit into L1 cache. In fact, the only win in fitting all three matrices into L1 cache is that it is possible, assuming the cache is not write-through, to save the cost of pushing previously used sections of C back to higher levels of memory. Often, however, the L1 cache *is* write-through, while higher levels are not. If this is the case, there is no way to minimize the write cost, so keeping all three matrices in L1 does not result in greater cache reuse. Therefore, ignoring the write cost, maximal cache reuse for our case is achieved when all of A fits into cache, with room for at least two columns of B and one cache line of C. Only one column of B is actually accessed at a time in this scenario; having enough storage for two columns assures that the old column will be the least recently used data when the cache overflows, thus making certain that all of A is kept in place (this obviously assumes the cache replacement policy is least recently used). While cache reuse can account for a great amount of the overall performance win, it is obviously not the only factor. For the on-chip matrix multiplication, other relevant factors are outlined below.

Instruction cache overflow: Instructions are cached, and it is therefore important to fit the on-chip multiply's instructions into the L1 cache. This means that it won't be possible to completely unroll all three loops, for instance.

Floating point instruction ordering: When we discuss floating point instruction ordering in this section, it will usually be in reference to *latency hiding*. Most modern architectures possess pipelined floating point units. This means that the results of an operation will not be available for use until s cycles later, where s is the number of stages in the floating point pipe (typically 3 or 5). Remember that the on-chip matrix multiply is of the form $C \leftarrow A^T B + C$; individual statements would then naturally be some variant of `C[i] += A[j] * B[k]`. If the architecture does not possess a fused multiply/add unit, this can cause an unnecessary execution stall. The operation `register = A[j] * B[k]` is issued to the floating point unit, and the add cannot be started until the result of this computation is available, s cycles later. Since the add operation is not started until the multiply finishes, the floating point pipe is not utilized. The solution is to remove this dependence by separating the multiply and add, and issuing unrelated instructions between them. This reordering of operations can be done in hardware (out-of-order execution) or by the compiler, but this will sometimes generate code that is not quite as efficient as doing it explicitly. More importantly, not all platforms have this capability, and in this case the performance win can be large.

Reducing loop overhead: The primary method of reducing loop overhead is through loop unrolling. If it is desirable to reduce loop overhead without changing the order of instructions, one must unroll the loop over the dimension common to A and B (i.e., unroll the K loop). Unrolling along the other dimensions (the M and N loops) changes the order of instructions, and thus the resulting memory access patterns.

Exposing parallelism: Many modern architectures have multiple floating point units. There are two barriers to achieving perfect parallel speedup with floating point computations in such a case. The first is a hardware limitation, and therefore out of our hands: All of the floating point units will need to access memory, and thus, for perfect parallel speedup, the memory fetch will usually also need to operate in parallel. The second prerequisite is that the compiler recognizes opportunities for parallelization, and this is amenable to software control. The fix for this is the classical one employed in such cases, namely unrolling the M and/or N loops, and choosing the correct register allocation so that parallel operations are not constrained by false dependencies.

Finding the correct number of cache misses: Any operand that is not already in a register must be fetched from memory. If that operand is not in the L1 cache, it must be fetched from further down the memory hierarchy, possibly resulting in large delays in execution. The number of cache misses which can be issued simultaneously without blocking execution varies between architectures. To minimize memory costs, the maximal number of cache misses should be issued each cycle, until all memory is in cache or used. In theory, one can permute the matrix multiply to ensure that this is true. In practice, this fine a level of control would be difficult to ensure (there would be problems with overflowing the instruction cache, and the generation of such precision instruction sequence, for instance). So the method used to control the cache-hit ratio is the more classical one of M and N loop unrolling.

3.3 ATLAS Performance Results

In this section we present double precision (64-bit floating point arithmetic) timings across various platforms. The timings presented here are different than many BLAS timings in that the cache is flushed before each call, and the leading dimensions of the arrays are set to greater than the number of rows of the matrix. This means that the performance numbers shown below, even when timing the same routine (for instance the vendor-supplied general matrix multiply routine) are lower than those reported in other papers. However, these numbers are in general a much better estimate of the performance a user will see in his application. More complete performance results and analysis can be found in [WD97].

Figure 5 shows the performance of ATLAS versus the vendor-supplied matrix multiply (where available) for a 500×500 matrix multiply.

Figure 6 shows the performance of LAPACK's LU factorization. For each platform three results are shown in the figure: (1) LU factorization time linking to ATLAS matrix multiply, (2) LU factorization time linking to vendor supplied BLAS, (3) LU factorization time linking only to the reference Fortran 77 BLAS. These results demonstrate that the automatically generated ATLAS routine provide good performance in practice.

4 Network-Enabled Solvers

Thanks to advances in hardware, networking infrastructure and algorithms, computing intensive problems in many areas can now be successfully attacked using networked, scientific computing. In the networked computing paradigm, vital pieces of software and information used by a computing process are spread across the network, and are identified and linked together only at run time. This is in contrast to the current software usage model where one acquires a copy (or copies) of

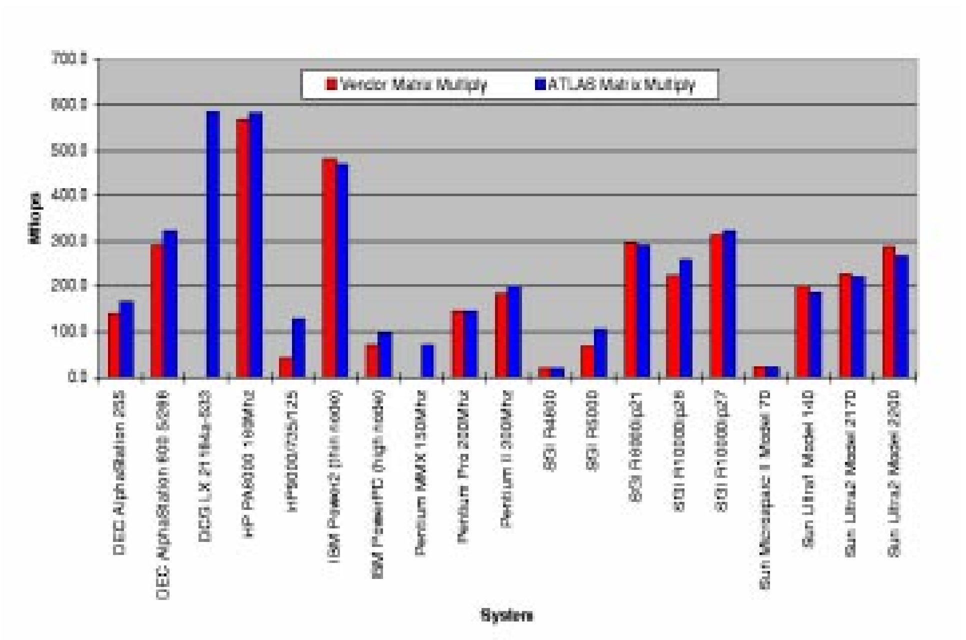

Figure 5: 500x500 matrix multiply performance across multiple architectures

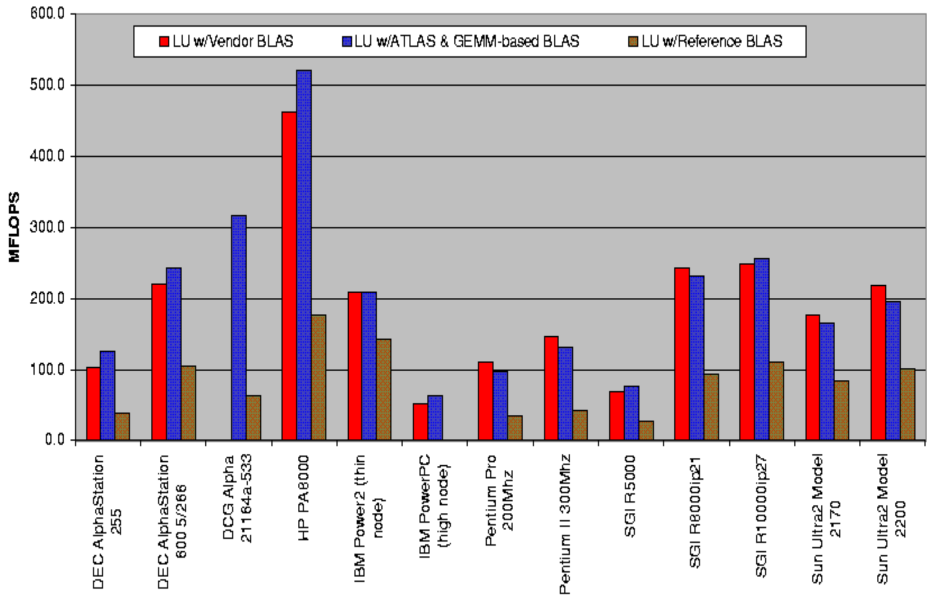

Figure 6: 500x500 LU factorization performance across multiple architectures

task-specific software package for use on local hosts. In this section, as a case study, we focus on the ongoing NetSolve project developed at the University of Tennessee and at the Oak Ridge National Laboratory (see `http://www.cs.utk.edu/netsolve`). This project adequately illustrates the current and modern research initiatives on network-enabled solvers. We first present an overview of the NetSolve project and examine some extensions being developed for NetSolve: an interface to the Condor system [LLM88], an interface to the ScaLAPACK parallel library [BCC+97], a bridge with the Ninf System [SSN+96], and an integration of NetSolve and ImageVision [ENB96].

4.1 The NetSolve System

The NetSolve system uses the *remote computing* paradigm: the program resides on the server; the user's data is sent to the server, where the appropriate programs or numerical libraries operate on it; the result is then sent back to the user's machine.

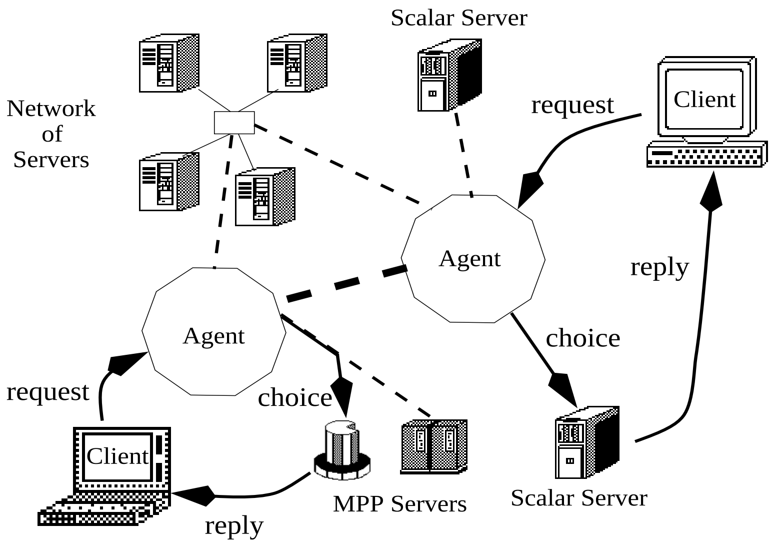

Figure 7: NetSolve's organization

Figure 7 depicts the typical layout of the system. NetSolve provides users with a pool of computational resources. These resources are computational servers that have access to ready-to-use numerical software. As shown in the figure, the computational servers can be running on single workstations, networks of workstations that can collaborate for solving a problem, or Massively Parallel Processor (MPP) systems. The user is using one of the NetSolve client interfaces. Through these interfaces, the user can send requests to the NetSolve system asking for a numerical computation to be carried out by one of the servers. The main role of the NetSolve agent is to process this request and to choose the most suitable server for this particular computation. Once a server has been chosen, it is assigned the computation, uses its available numerical software, and eventually returns the results to the user. One of the major advantages of this approach is that the agent performs load-balancing among the different resources.

As shown in Figure 7, there can be multiple instances of the NetSolve agent on the network, and different clients can contact different agents depending on their locations. The agents can exchange information about their different servers and allow access from any client to any server if desirable. NetSolve can be used either via the Internet or on an intranet, such as inside a research department or a university, without participating in any Internet based computation. Another important aspect of NetSolve is that the configuration of the system is entirely flexible: any server/agent can be stopped and (re-)started at any time without jeopardizing the integrity of the system.

4.1.1 The Computational Resources

When building the NetSolve system, one of the challenges was to design a suitable model for the computational servers. The NetSolve servers are configurable so that they can be easily upgraded to encompass ever-increasing sets of numerical functionalities. The NetSolve servers are also pre-installed, meaning that the end-user does not have to install any numerical software. Finally, the NetSolve servers provide uniform access to the numerical software, in the sense that the end-user has the illusion that he or she is accessing numerical subroutines from a single, coherent numerical library.

To make the implementation of such a computational server model possible, a general, machine-independent way of describing a numerical computation as well as a set of tools to generate new computational modules as easily as possible have been designed. The main component of this framework is a *descriptive language* which is used to describe each separate numerical functionality of a computational server. The description files written in this language can be compiled by NetSolve into actual computational modules executable on any UNIX or NT platform. These files can then be exchanged by any institution wanting to set up servers: each time a new description file is created, the capabilities of the entire NetSolve system are increased.

A number of description files have been generated for a variety of numerical libraries: ARPACK, FitPack, ItPack, MinPack, FFTPACK, LAPACK, BLAS, QMR, Minpack and ScaLAPACK. These numerical libraries cover several fields of computational science; Linear Algebra, Optimization, Fast Fourier Transforms, etc.

4.1.2 The Client Interfaces

A major concern in designing NetSolve was to provide several interfaces for a wide range of users. NetSolve can be invoked through C, Fortran, Java, Matlab [Mat92] and Mathematica [Wol96]. In addition, there is a Web-enabled Java GUI. Another concern was keeping the interfaces as simple as possible. For example, there are only two calls in the MATLAB interface, and they are sufficient to allow users to submit problems to the NetSolve system. Each interface provides asynchronous calls to NetSolve in addition to traditional synchronous or blocking calls. When several asynchronous requests are sent to a NetSolve agent, they are dispatched among the available computational resources according to the load-balancing schemes implemented by the agent. Hence, the user—with virtually no effort—can achieve coarse-grained parallelism from either a C or Fortran program, or from interaction with a high-level interface. All the interfaces are described in detail in the "NetSolve's Client User's Guide" [CD95].

4.1.3 The NetSolve Agent

Keeping track of what software resources are available and on which servers they are located is perhaps the most fundamental task of the NetSolve agent. Since the computational servers use the same framework to contribute software to the system (see Section 4.1.1), it is possible for the agent to maintain a database of different numerical functionalities available to the users.

Each time a new server is started, it sends an application request to an instance of the NetSolve agent. This request contains general information about the server and the list of numerical functions it intends to contribute to the system. The agent examines this list and detects possible discrepancies with the other existing servers in the system. Based on the agent's verdict, the server can be integrated into the system and available for clients.

The goal of the NetSolve agent is to choose the best-suited computational server for each incoming request to the system. For each user request, the agent determines the set of servers that can handle the computation and makes a choice between all the possible resources. To do so, the agent uses computation-specific and resource-specific information. Computation-specific information is mostly included in the user request whereas resource-specific information is partly static (server's host processor speed, memory available, etc.) and partly dynamic (processor workload). Rationale and further detail on these issues can be found in [BCD96], as well as a description of how NetSolve ensures fault-tolerance among the servers.

Agent-based computing seems to be a promising strategy. NetSolve is evolving into a more elaborate system and a major part of this evolution is to take place within the agent. Such issues as user accounting, security, data encryption for instance are only partially addressed in the current implementation of NetSolve and already is the object of much work. As the types of hardware resources and the types of numerical software available on the computational servers become more and more diverse, the resource broker embedded in the agent need to become increasingly sophisticated. There are many difficulties in providing a uniform performance metric that encompasses any type of algorithmic and hardware considerations in a metacomputing setting, especially when different numerical resources, or even entire frameworks are integrated into NetSolve. Such integrations are described in the following sections.

4.2 Integration of Computational Resources into NetSolve

In this section, we present how various computational resources can be integrated into NetSolve. As explained in Section 4.1.1, traditional software libraries are easy to integrate into the NetSolve system. We present however how four very different and more complex computational resources have been integrated. We selected a workstation manager environment, a parallel numerical library, a global-wide computing infrastructure similar to NetSolve itself, and finally a general purpose image processing application.

4.2.1 Interface to the Condor System

Condor [LLM88], developed at the University of Wisconsin, Madison, is an environment that can manage very large collections of distributively owned workstations. Its development has been motivated by the ever increasing need for scientists and engineers to exploit the capacity of such collections, mainly by taking advantage of otherwise unused CPU cycles. Interfacing NetSolve and Condor is a very natural idea. NetSolve provides remote easy access to computational resources through multiple, attractive user interfaces. Condor allows users to harness the power of a pool of machines while using otherwise wasted CPU cycles. The users at the consoles of those machines are not penalized by the scheduling of Condor jobs. If the pool of machines is reasonably large, it is usually the case that Condor jobs can be scheduled almost immediately. This could prove to be very interesting for a project like NetSolve. Indeed, NetSolve servers may be started so that they grant local resource access to outside users. Interfacing NetSolve and Condor could then give priority to local users and provide underutilized only CPU cycles to outside users.

A Condor pool consists of any number of machines, that are connected by a network. Condor daemons constantly monitor the status of the individual computers in the cluster. Two daemons run on each machine, the *startd* and the *schedd*. The *startd* monitors information about the machine itself (load, mouse/keyboard activity, etc.) and decides if it is available to run a Condor job. The *schedd* keeps track of all the Condor jobs that have been submitted to the machine. One of the machine, the *Central Manager*, keeps track of all the resources and jobs in the pool. When a job is submitted to Condor, the scheduler on the central manager matches a machine in the Condor pool to that job. Once the job has been started, it is periodically checkpointed, can be interrupted and migrated to a machine whose architecture is the same as the one of the machine on which the execution was initiated. This organization is partly depicted in Figure 8. More details on the Condor system and the software layers can be found in [LLM88].

Figure 8 shows how an entire Condor pool can be seen as a single NetSolve computational resource. The Central Manager runs two daemons in addition to the usual *startd* and *schedd*: the *negotiator* and the *collector*. A machine also runs a customized version of the NetSolve server. When this server receives a request from a client, instead of creating a local child process running a computational module, it uses the Condor tools to submit that module to the Condor pool. The negotiator on the Central Manager then chooses a target machine for the computational module. Due to fluctuations in the state of the pool, the computational module can then be migrated among the machines in the pool. When the results of the numerical computation are obtained, the NetSolve server transmits that result back to the client.

The actual implementation of the NetSolve/Condor interface was made easy by the Condor tools provided to the Condor user. However, the restrictions that apply to a Condor job concerning system calls were difficult to satisfy and required quite a few changes to obtain a Condor-enabled NetSolve server. A major issue however still needs to be addressed; how does the NetSolve agent perceive a Condor pool as a resource? Finding the appropriate performance prediction technique is at the focus of the current NetSolve/Condor collaboration.

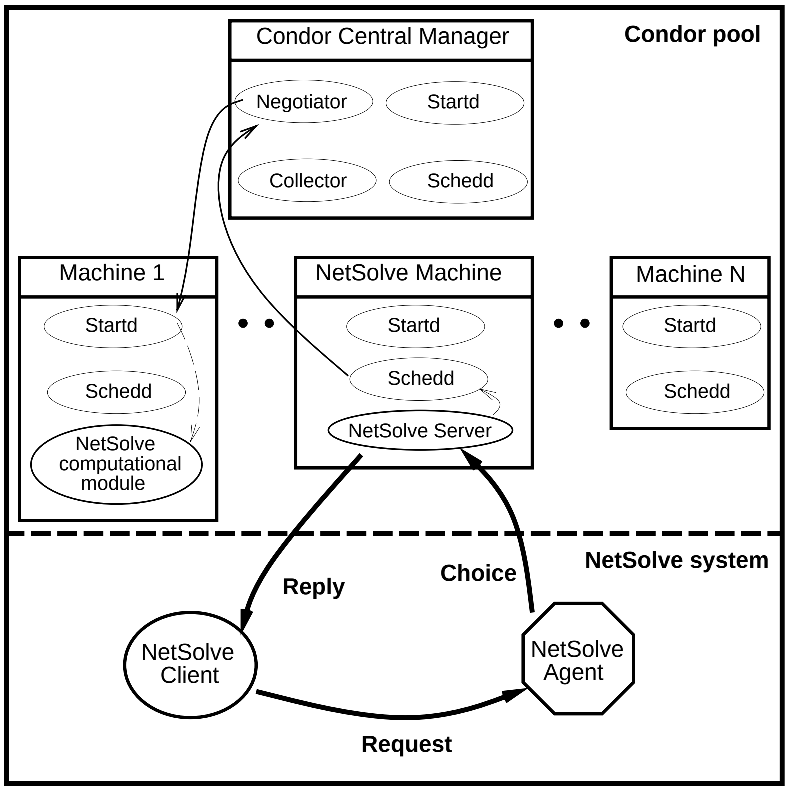

Figure 8: NetSolve and Condor

4.2.2 Integrating Parallel Numerical Libraries

Integrating software libraries designed for distributed-memory concurrent computers into NetSolve allows a workstation's user to access massively parallel processors to perform large computations. This access can be made extremely simple via NetSolve and the user may not even be aware that he or she is using a parallel library on such a computer. As an example, we describe in this section, how the ScaLAPACK library [BCC+97] has been integrated into the NetSolve system.

As briefly described in Section 2.1.3, the Scalable Linear Algebra Package (ScaLAPACK) is a library of high-performance linear algebra routines for distributed-memory message-passing MIMD computers as well as networks or clusters of workstations supporting PVM [GBD+94] or MPI [SOH+96]. It is a continuation of the LAPACK [ABB+95] project, and contains routines for solving systems of linear equations, least squares problems, and eigenvalue problems. ScaLAPACK views the underlying multi-processor system as a rectangular process *grid*. Global data is mapped to the local memories of the processes in that grid assuming specific data-distributions. For performance and load balance reasons, ScaLAPACK uses the two-dimensional block cyclic distribution scheme for dense matrix computations. Inter-process communication within ScaLAPACK is done via the Basic Linear Algebra Communication Subprograms (BLACS) [WD95].

Figure 9 is a very simple description of how the NetSolve server has been customized to use the ScaLAPACK library. The customized server receives data input from the client in the traditional way. The NetSolve server uses BLACS calls to set up the ScaLAPACK process grid. ScaLAPACK requires that the data already be distributed among the processors prior to any library call. This is the reason why each user input is first distributed on the process grid according to the block cyclic decomposition when necessary. The server can then initiate the call to ScaLAPACK and wait until completion of the computation. When the ScaLAPACK call returns, the result of the computation is distributed on the two-dimensional process grid. The server then gathers that result and sends it back to the client in the expected format. This process is completely transparent to the user who does not even realize that a parallel execution has been taking place.

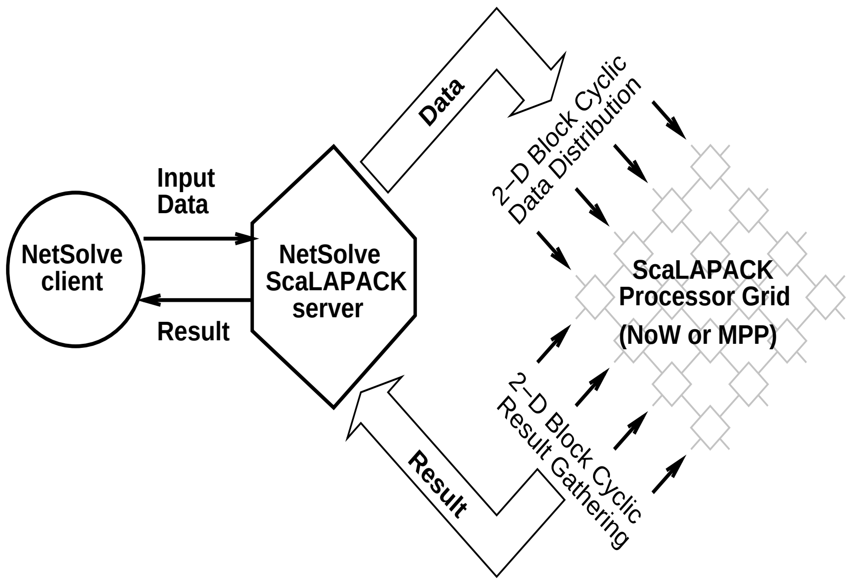
NetSolve client
Input Data
Result
NetSolve ScaLAPACK server
Data
Result
2–D Block Cyclic Data Distribution
2–D Block Cyclic Result Gathering
ScaLAPACK Processor Grid
(NoW or MPP)

Figure 9: The ScaLAPACK NetSolve Server Paradigm

This approach is very promising. A client can use MATLAB on a PC and issue a simple call like `[x] = netsolve('eig',a)` and have an MPP system use a high-performance library to perform a large eigenvalue computation. A prototype of the customized server running on top of PVM [GBD+94] or MPI [SOH+96] has been designed. There are many research issues arising with integrating parallel libraries in NetSolve, including performance prediction, choice of processor-grid size, choice of numerical algorithm, processor availability, accounting, etc.

4.2.3 NetSolve and Ninf

Ninf [SSN+96], developed at the Electrotechnical Laboratory, Tsukuba, is a global network-wide computing infrastructure project which allows users to access computational resources including hardware, software, and scientific data distributed across a wide area network with an easy-to-use interface. Computational resources are shared as Ninf remote libraries and are executable at remote Ninf servers. Users can build an application by calling the libraries with the Ninf Remote Procedure Call, which is designed to provide a programming interface similar to conventional function calls in existing languages, and is tailored for scientific computation. In order to facilitate location transparency and network-wide parallelism, the Ninf MetaServer maintains global resource information regarding computational server and databases. It can therefore allocate and schedule coarse-grained computations to achieve good global load balancing. Ninf also interfaces with existing network service such as the WWW for easy accessibility. Clearly, NetSolve and Ninf bear strong similarities both in motivation and general design. Allowing the two systems to coexist and collaborate should lead to promising developments.

Some design issues prevent an immediate seamless integration of the two softwares (conceptual differences between the NetSolve agent and the Ninf Metaserver, problem specifications, user interfaces, data transfer protocols, etc.). In order to overcome these issues, the Ninf team started developing two *adapters*: a NetSolve-Ninf adapter and a Ninf NetSolve-adapter. Thanks to those adapters, Ninf clients can use computational resources administrated by a NetSolve system and vice-versa.

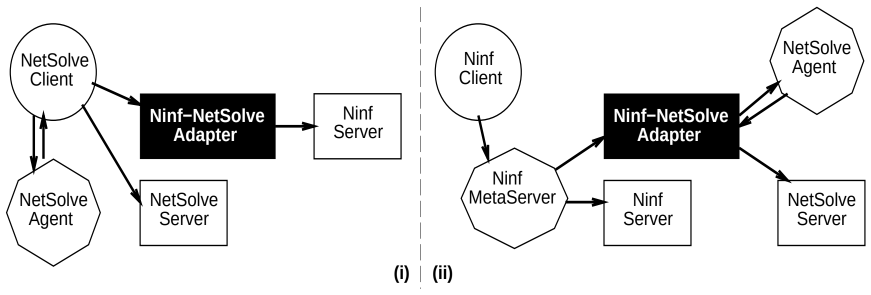

Figure 10: Going (i) from NetSolve to Ninf and (ii) from Ninf to NetSolve

Figure 10(i) shows the Ninf-NetSolve adapter allowing access to Ninf resource from a NetSolve client. The adapter is just seen by the NetSolve agent as any other NetSolve server. When a NetSolve client sends a request to the agent, it can then be told to use the NetSolve adapter. The adapter performs protocol translation, interface translation, and data transfer, asks a Ninf server to perform the required computation and returns the result to the user.

In Figure 10(ii), the NetSolve-Ninf adapter can be seen by the Ninf MetaServer as a Ninf server, but in fact plays the role of a NetSolve client. This is a little different from the Ninf-NetSolve adapter because the NetSolve agent is a resource broker whereas the Ninf MetaServer is a proxy server. Once the adapter receives the result of the computation from some NetSolve server, it transfers that result back to the Ninf client.

There are several advantages of using such adapters. Updating the adapters to reflects the evolutions of NetSolve or Ninf seems to be an easy task. Some early implementation evaluations tend to show that using either system via an adapter causes acceptable overheads, mainly due to additional data transfers. Those first experiments appear encouraging and will definitely be extended to effectively enable an integration of NetSolve and Ninf.

4.2.4 Extending ImageVision by the Use of NetSolve

In this section, we describe how NetSolve can be used as a building block for a general purpose framework for basic image processing, based on the commercial ImageVision library [ENB96]. This project is under development at the ICG institute at Graz University of Technology, Austria. The scope of the project is to make basic image processing functions available for remote execution over a network. The goals of the project include two objectives that can be leveraged by NetSolve. First, the resulting software should prevent the user from having to install complicated image processing libraries. Second, the functionalities should be available via Java-based applications. The ImageVision Library (IL) [ENB96] is an object-oriented library written in C++ by Silicon Graphics, Inc. (SGI) and shipped with newer workstations. It contains typical image processing routines to efficiently access, manipulate, display, and store image data. ImageVision has been judged quite complete and mature by the research team at ICG and seems therefore a good choice as an "engine" for building a remote access image processing framework. Such a framework will make IL accessible from any platform (and not only from SGI workstations) and is described in [Obe97].

The reasons why NetSolve has been a first choice for such a project are diverse. First, NetSolve is easy to understand, use, and extend. Second, NetSolve is freely available. Third, NetSolve provides language binding to Fortran, C, and Java. And finally, NetSolve's agent-based design allows load monitoring and balancing among the available servers. New NetSolve computational modules corresponding to the desired image processing functionalities will be created and integrated into the NetSolve servers. A big part of the project at ICG is to build a Java GUI to IL.

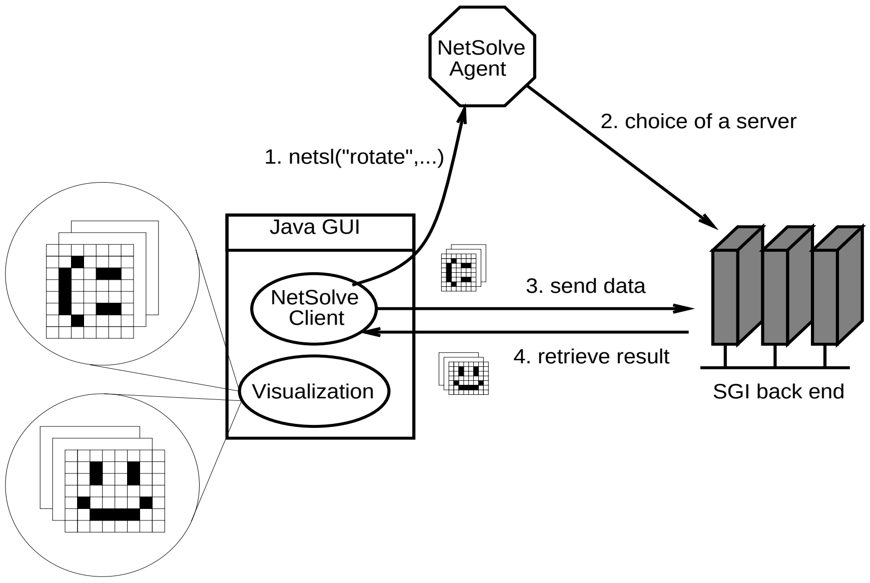

Figure 11: ImageVision and NetSolve

Figure 11 shows a simple example of how ImageVision can be accessed via NetSolve. A Java GUI can be built on top of the NetSolve Java API. As shown on the figure, this GUI offers visualization capabilities. For computations, it uses an embedded NetSolve client and contacts SGI servers that have access to IL. The user of the Java GUI does not realize that NetSolve is the back end of the system, or that he or she uses a SGI library without running the GUI on a SGI machine! The protocol depicted in the figure is of course simplistic. In order to obtain acceptable levels of performance, the network traffic needs to be minimized. There are several ways of attacking this problem: keeping a “state” in the server, combine requests, reference images with URLs for instance, etc.

5 Conclusions

This chapter presented some of the recent developments in linear algebra software designed to exploit advanced-architecture computers. We focused on three essential components out of which current and modern problem solving environments are constructed: well-designed numerical software libraries, automatic generators of optimized numerical kernels and flexible, easy-to-access software systems enabling the hardware and software computational resources. Each of these components was concretely illustrated with existing and/or ongoing research projects. We summarize below the most important features of these components. 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 scientific problem solving environments.

5.1 Well-Designed Numerical Software Libraries

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.

Like portability, *scalability* demands that a program be reasonably effective over a wide range of number of processors. The scalability of parallel algorithms, and software libraries based on 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. The ScaLAPACK approach to this problem is block algorithms with adjustable block size.

Scalable parallel architectures of the present and 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 networks 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. Today, however, the distributed nature of the underlying hardware continues to be visible at the programming level; therefore, 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.), have 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.

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.

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. One of the major design goal of *High Performance Fortran* (HPF) [KLS+94] was to achieve (almost) a transparent program portability to the user, from shared-memory multiprocessors up to distributed-memory parallel computers and networks of workstations. But writing efficient numerical kernels with HPF is not an easy task. First of all, there is the need to recast linear algebra kernels in terms of block operations (otherwise, as already mentioned, the performance will be limited by that of Level 1 BLAS routines). Second, the user is required to explicitly state how the data is partitioned amongst the processors. Third, 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. Hence, the HPF programmer may well choose to call ScaLAPACK routines just as he called LAPACK routines on sequential processors with a memory hierarchy. To facilitate this task, an interface has been developed [BDP+98]. The design of this interface has been made possible because ScaLAPACK is using the same block-cyclic distribution primitives as those specified in the HPF standards. Of course, HPF can still prove a useful tool at a higher level, that of parallelizing a whole scientific operation, because the user will be relieved from the low level details of generating the code for communications.

5.2 Automatic Generation and Optimization of Numerical Kernels on Various Processor Architectures

The ATLAS package presently available on netlib is organized around the matrix-matrix multiplication. This operation is the essential building block of all of the Level 3 BLAS. Initial research using publicly available matrix-multiply-based BLAS implementations [KLV93, DDP94] suggests that this provides a perfectly acceptable Level 3 BLAS. As time allows, we can avoid some of the $O(N^2)$ costs associated with using the matrix-multiply-based BLAS by supporting the Level 3 BLAS directly in ATLAS. We also plan on providing the software for complex data types.

We have preliminary results for the most important Level 2 BLAS routine (matrix-vector multiply) as well. This is of particular importance, because matrix vector operations, which have $O(N^2)$ operations and $O(N^2)$ data, demand a significantly different code generation approach than that required for matrix-matrix operations, where the data is $O(N^2)$, but the operation count is $O(N^3)$. Initial results suggest that ATLAS will achieve comparable success with optimizing the Level 2 BLAS as has been achieved for Level 3 (this means that the ATLAS timings compared to the vendor will be comparable; obviously, unless the target architecture supports many pipes to memory, a Level 2 BLAS operation will not be as efficient as the corresponding Level 3 BLAS operation).

Another avenue of ongoing research involves sparse algorithms. The fundamental building block of iterative methods is the sparse matrix-vector multiply. This work leverages the present research (in particular, make use of the dense matrix-vector multiply). The present work uses compile-time adaptation of software. Since matrix-vector multiply may be called literally thousands of times during the course of an iterative method, run-time adaptation is also investigated. These run-time adaptations may include matrix dependent transformations [Tol97], as well as specific code generation.

ATLAS has demonstrated the ability to produce highly optimized matrix multiply for a wide range of architectures based on a code generator that probes and searches the system for an optimal set of parameters. This avoids the tedious task of generating by hand routines optimized for a specific architecture. We believe these ideas can be expanded to cover not only the Level 3 BLAS, but Level 2 BLAS as well. In addition there is scope for additional operations beyond the BLAS, such as sparse matrix-vector operations, and FFTs.

5.3 The NetSolve Problem Solving Environment

We have discussed throughout this chapter how NetSolve can be customized, extended, and used for a variety of purposes. We first described in Sections 4.2.1 and 4.2.2 how NetSolve can encompass new types of computing resources, resulting in a more powerful and flexible environment and raising new research issues. We next discussed in Section 4.2.3 how NetSolve and Ninf can be merged into a single metacomputing environment. Finally, in Section 4.2.4, we gave an example of an entire application that uses NetSolve as an operating environment to build general image processing framework. All these developments take place at different levels in the NetSolve project and have had and will continue to have an impact on the project itself, causing it to improve and expand.

The scientific community has long used the Internet for communication of email, software, and documentation. Until recently there has been little use of the network for actual computations. This situation is changing rapidly and will have an enormous impact on the future. Novel user interfaces that hide the complexity of scalable parallelism 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 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).

Software and Documentation Availability

Most of the software mentioned in this document and the corresponding documentations are in the public domain, and are available from *netlib* (`http://www.netlib.org/`) [DG87]. For instance, the EISPACK, LINPACK, LAPACK, BLACS, ScaLAPACK, and ATLAS software packages are in the public domain, and are available from *netlib*. Moreover, these publically available software packages can also be retrieved by e-mail. For example, to obtain more information on LAPACK, one should send the following one-line email message to `netlib@ornl.gov`: `send index from lapack`. Information for other packages can be similarly obtained. Real-time information on the NetSolve project can be found at the following web address `http://www.cs.utk.edu/netsolve`.

References

[ABB+95] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorensen, LAPACK User's Guide (second edition), SIAM, Philadelphia PA, 1995

[AD89] E. Anderson and J. Dongarra, Results from the Initial Release of LAPACK, LAPACK Working Note No. 16, Technical Report University of Tennessee, Knoxville, TN, 1989

[AD90] E. Anderson and J. Dongarra, Evaluating Block Algorithm Variants in LAPACK, LAPACK Working Note No. 19, Technical Report University of Tennessee, Knoxville, TN, 1990

[BAC+97] J. Bilmes, K. Asanović, C.W. Chin and J. Demmel, Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology, in Proceedings of the International Conference on Supercomputing, ACM SIGARC, Vienna, Austria, 1997

[BCC+97] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker and R. C. Whaley, ScaLAPACK Users' Guide, SIAM, Philadelphia PA, 1997

[BCD96] S. Browne, H. Casanova and J. Dongarra, Providing Access to High Performance Computing Technologies, Lecture Notes in Computer Science 1184, Editors J. Wasniewski, J. Dongarra, K. Madsen and D. Olesen, Springer-Verlag, Berlin, 1996

[BDP+98] L. Blackford, J. Dongarra, C. Papadopoulos, and R. C. Whaley, Installation Guide and Design of the HPF 1.1 interface to ScaLAPACK, SLHPF, LAPACK Working Note No. 137, Technical Report UT CS-98-396, University of Tennessee, Knoxville, TN, 1998

[CDG+93] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken and K. Yelick, Introduction to Split-C: Version 0.9, Computer Science Division – EECS, University of California, Berkeley, CA 94720, 1993

[BW98] L. S. Blackford and R. C. Whaley, ScaLAPACK Evaluation and Performance at the DoD MSRCs, LAPACK Working Note No. 136, Technical Report UT CS-98-388, University of Tennessee, Knoxville, TN, 1998

[CDO+95] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker and R. C. Whaley, A Proposal for a Set of Parallel Basic Linear Algebra Subprograms, LAPACK Working Note No. 100, Technical report UT CS-95-292, University of Tennessee, Knoxville, TN, 1995

[CD95] H. Casanova and J. Dongarra, NetSolve: A Network Server for Solving Computational Science Problems, Technical report UT CS-95-313, University of Tennessee, Department of Computer Science, Knoxville, TN, 1995

[DDH+88] J. Dongarra, J. Du Croz, S. Hammarling and R. Hanson, An Extended Set of Fortran Basic Linear Algebra Subroutines, ACM Transactions on Mathematical Software, Volume 14(1), 1988

[DDH+90] J. Dongarra, J. Du Croz, S. Hammarling and I. Duff, A Set of Level 3 Basic Linear Algebra Subprograms, ACM Transactions on Mathematical Software, Volume 16(1), 1990

[DDP94] M. Dayde, I. Duff and A. Petitet, A Parallel Block Implementation of Level 3 BLAS for MIMD Vector Processors, ACM Transactions on Mathematical Software, Volume 20(2), 1994

[DDS+91] J. Dongarra, I. Duff, D. C. Sorensen and H. A. Van der Vorst, Solving Linear Systems on Vector and Shared Memory Computers, SIAM Publications, Philadelphia, PA, 1991

[DG87] J. Dongarra and E. Grosse, Distribution of Mathematical Software via Electronic Mail, Communications of the ACM, Volume 30(5), 1987 (See `http://www.netlib.org/`)

[Dem89] J. Demmel, LAPACK: A Portable Linear Algebra Library for Supercomputers, Proceedings of the 1989 IEEE Control Systems Society Workshop on Computer-Aided Control System Design, 1989

[DMR91] J. Dongarra, P. Mayes and G. Radicati di Brozolo, The IBM RISC System 6000 and Linear Algebra Operations, *Supercomputer*, 8(4):15–30, 1991

[Don84] J. Dongarra, Increasing the Performance of Mathematical Software through High-Level Modularity, Proceedings Sixth Int. Symp. Comp. Methods in Eng. & Applied Sciences, Versailles, France, North-Holland, 1984

[DP87] J. Du Croz and M. Pont, The Development of a Floating-Point Validation Package, Proceedings of the 8th Symposium on Computer Arithmetic, IEEE Computer Society Press, Como, Italy, 1987

[DPW93] J. Dongarra, R. Pozo and D. Walker, An Object Oriented Design for High Performance Linear Algebra on Distributed Memory Architectures, Proceedings of the Object Oriented Numerics Conference, 1993

[ENB96] G. Eckel, J. Neider and E. Bassler, ImageVision Library Programming Guide, Silicon Graphics, Inc., Mountain View, CA, 1996

[Ede93] A. Edelman, Large Dense Numerical Linear Algebra in 1993: The Parallel Computing Influence, International Journal of Supercomputing Applications, Vol. 7, No. 2, 1993

[FJL+88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker, Solving Problems on Concurrent Processors, Volume 1, Prentice Hall, Englewood Cliffs, N.J., 1988

[FK98] The Grid – Blueprint for a New Computing Infrastructure, Eds. I. Foster and C. Kesselman, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1998

[FO88] E. Felten and S. Otto, Coherent Parallel C, Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications", ACM Press, 1988

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, PVM : Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing, The MIT Press Cambridge, Massachusetts, 1994

[GK90] A. Gupta and V. Kumar, On the Scalability of FFT on Parallel Computers, Proceedings of the Frontiers 90 Conference on Massively Parallel Computation, IEEE Computer Society Press, 1990

[GPS90] K. Gallivan, R. Plemmons and A. Sameh, Parallel Algorithms for Dense Linear Algebra Computations, SIAM Review, 32(1), 1990

[HJ81] R. W. Hockney and C. R. Jesshope, Parallel Computers, Adam Hilger Ltd., Bristol, UK, 1981

[HS67] J. Hess and M. Smith, Calculation of Potential Flows about Arbitrary Bodies, in D. Küchemann, editor, Progress in Aeronautical Sciences, Vol. 8, Pergamon Press, 1967

[Har90] R. Harrington, Origin and Development of the Method of Moments for Field Computation, IEEE Antennas and Propagation Magazine, 1990

[Hes90] J. Hess, Panel Methods in Computational Fluid Dynamics, Annal Reviews of Fluid Mechanics, Vol. 22, 1990

[Kah87] W. Kahan, Paranoia, Available from netlib [DG87]

[KLS+94] C. Koebel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel, The High Performance Fortran Handbook, The MIT Press, Cambridge, Massachusetts, 1994

[KLV93] B. Kågström, P. Ling and C. Van Loan, Portable High Performance GEMM-based Level 3 BLAS, in R. F. Sincovec et al., editor, *Parallel Processing for Scientific Computing*, SIAM, Philadelphia, 1993

[LHK+79] C. Lawson, R. Hanson, D. Kincaid and F. Krogh, Basic Linear Algebra Subprograms for Fortran Usage, ACM Trans. Math. Softw., Volume 5, 1979

[LLM88] M. Litzkow and M. Livny and M.W. Mutka, Condor - A Hunter of Idle Workstations, Proceedings of the 8th International Conference of Distributed Computing Systems, 1988

[Mat92] The Math Works Inc., MATLAB Reference Guide, The Math Works Inc., 1992

[MPI+94] Message Passing Interface Forum, MPI: A Message-Passing Interface standard, International Journal of Supercomputer Applications, Volume 8(3/4), 1994

[Obe97] M. Oberhuber, `http://www.icg.tu-graz.ac.at/mober/pub`, Integrating ImageVision into NetSolve, 1997

[SOH+96] M. Snir, S. Otto, S. Huss-Lederman, D. Walker and J. Dongarra, *MPI: The Complete Reference*, MIT Press, Cambridge, Massachusetts, 1996

[SSN+96] S. Sekiguchi, M. Sato, H. Nakada, S. Matsuoka and U. Nagashima, Ninf : Network based Information Library for Globally High Performance Computing, Proceedings of Parallel Object-Oriented Methods and Applications (POOMA), Santa Fe, 1996

[Tol97] S. Toledo, Improving Instruction-Level Parallelism in Sparse Matrix-Vector Multiplication Using Reordering, Blocking, and Prefetching, in *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, SIAM, 1997

[WD95] R. C. Whaley and J. Dongarra, A User's Guide to the BLACS v1.1, LAPACK Working Note No. 94, Technical Report UT CS-95-281, University of Tennessee, Knoxville, 1995 (See also `http://www.netlib.org/blacs/`)

[WD97] R. C. Whaley and J. Dongarra, Automatically Tuned Linear Algebra Software, LAPACK Working Note No. 131, Technical Report UT CS-97-366, University of Tennessee, Knoxville, TN, 1997 (See also `http://www.netlib.org/atlas/`) (Note: A revised version of this paper will appear in the Proceedings of Supercomputing '98, ACM SIGARCH and IEEE Computer Society)

[WR71] J. Wilkinson, C. Reinsch, Handbook for Automatic Computation: Volume II - Linear Algebra, Springer-Verlag, New York, 1971

[Wan91] J. Wang, Generalized Moment Methods in Electromagnetics, John Wiley & Sons, New-York, 1991

[Wol96] S. Wolfram, The Mathematica Book, Third Edition, Wolfram Median, Inc. and Cambridge University Press, 1996