Numerical Algoritms and Libraries

Jack J. Dongarra
Danny Sorensen

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.

This chapter discusses some of the recent developments in numerical algorithms and software libraries 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 scientific 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) [**?**, **?**, **?**], 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.

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 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 1 discusses an innovative approach [**?**, **?**] 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 [?].

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

0.1 Software Design

Developing a library of high-quality subroutines for scientific problems 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.

0.2 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.

0.2.1 Performance

0.2.2 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* [?]. 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 [?, ?] 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 [?]. In addition, software for distributed-memory computers should work correctly for a large class of data decompositions.

0.2.3 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. There are also sparse matrices, which may be stored in many different ways.

1 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) [?] 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 [?]. 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.

1.1 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 **??**. As shown in Section **??**, 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.

1.2 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 [**?**]. 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 handles the looping and blocking necessary to build the complete matrix-matrix multiply from the on-chip multiply. 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.

1.3 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) [?] 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 [**?**]. 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.

1.4 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 [**?**, **?**] 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 [**?**], 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.

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/`) [?]. 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`.

1. Traditional Libraries. The ultimate development of fully mature parallel scalable libraries will necessarily depend on breakthroughs in many other supporting technologies. Development of scalable libraries cannot wait, however, until all of the enabling technologies are in place. The reason is twofold: (1) the need for such libraries for existing and near-term parallel architectures is immediate, and (2) progress in all of the supporting technologies will be critically dependent on feedback from concurrent efforts in library development.

The linear algebra community has long recognized that we needed something to help us in developing our algorithms into software libraries. Several years ago, as a community effort, we put together a *de facto* standard for identifying basic operations required in our algorithms and software. Our hope was that the standard would be implemented on the machines by many manufacturers and that we would then be able to draw on the power of having that implementation in a rather portable way. We began with those BLAS operations designed for basic matrix computations. Since on a parallel system message passing is critical we have been involved with the development of message passing standards. Both PVM and MPI have helped in the establishment of standards and the promotion of portable software that is critical for software library work.

2. User Interfaces. As computer architectures and programming paradigms become increasingly complex, it becomes desirable to hide this complexity as much as possible from the end user. The traditional user interface for large, general-purpose mathematical and scientific libraries is to have users write their own programs (usually in Fortran) that call on library routines to solve specific subproblems that arise during the course of the computation. When extended to run on parallel architectures, this approach has only a limited ability to hide the underlying architectural and programming complexity from the user. As we extend the conventional notion of mathematical and scientific libraries to scalable architectures, we must rethink the conventional concept of user interface and devise alternate approaches that are capable of hiding architectural, algorithmic, and data complexity from users.

One possible approach is that of a "problem solving environment," typified by current packages like MATLAB, which would provide an interactive, graphical interface for specifying and solving scientific problems, with both algorithms and data structures hidden from the user because the package itself is responsible for storing and retrieving the problem data in an efficient distributed manner. Such an approach seems especially appropriate in keeping with the trend toward graphical workstations as the primary user access to computing facilities, together with networks of computational resources that include various parallel computers and conventional supercomputers. The ultimate hope would be to provide seamless access to such computational engines that would be invoked selectively for different parts of the user's computation according to whichever machine is most appropriate for a particular subproblem. We envision at least two interfaces for a library in linear algebra. One would be along conventional lines (LAPACK-style) for immediate use in conventional programs that are being ported to novel machines, and the other would be in the form of a problem solving environment (MATLAB-style). The two proposed interface styles are not inconsistent or incompatible: the problem solving environment can in fact be built on top of software that is based on a more conventional interface.

3. Heterogeneous Networking. Current trends in parallel architectures, high-speed networks, and personal workstations suggest that the computational environment of the future for working scientists will require the seamless integration of heterogeneous systems into a coherent problem-solving environment. Graphical workstations will provide the standard user interface, with a variety of computational engines and data storage devices distributed across a network. The diversity of parallel architectures means that inevitably different computational tasks will be more efficient on some than on others, with no single architecture uniformly superior. Thus, we expect the "problem-solving environment" envisioned above eventually to migrate to a heterogeneous network of workstations, file servers, and parallel computation servers. The various computational tasks required to solve a given problem would automatically and transparently be targeted to the most appropriate computational engine on the network. System resources would be shared among many users, but in a somewhat different manner than conventional timesharing computer systems. We have already made important first steps toward achieving these goals with systems like PVM and MPI, which supplies the low-level services necessary to coordinate the use of multiple workstations and other computers for individual jobs, and this system could serve as the foundation for a complete problem-solving environment of the type we envision.

Network computing techniques such as NetSolve offers the ability to look for computational resources on a network for a submitted problem (which can be a single LAPACK, ScaLAPACK or Matlab function call), choose the best one available, solve it (with retry for fault tolerance) and return the answer to the user. This system is available for Fortran, C, and Matlab users.

4. Software Tools and Standards. An ambitious development effort in scalable libraries will require a great deal of supporting infrastructure. Moreover, the portability of any library is critically dependent on adherence to standards. In the case of software for parallel architectures, precious few standards exist, so new standards must evolve along with the research and development. A particularly important area for scalable distributed-memory architectures is internode communication. The BLAS have proven to be very effective in assisting portable, efficient software for sequential and some of the current class of high-performance computers. We are investigating the possibility of expanding the set of standards that have been developed. There is a need for a light weight interface to much of the functionality of traditional BLAS. In addition, iterative and sparse direct methods require additional functionality not in traditional BLAS. Numerical methods for dense matrices on parallel computers require high efficiency kernels that provide functionality similar to that in traditional BLAS on sequential machines.

Software tools are also of great importance, both for developers to use in designing and tuning the library software, and for end-users to monitor the efficiency of their applications.

Conclusions

1. In spite of a lack of enabling technologies, library development cannot wait for research in programming languages, compilers, software tools, and other areas to mature, but must be done in conjunction with work in these areas. The the time to begin is now.

2. The user–library interface needs rethinking. It is not clear that the conventional library interface will be adequate to hide the underlying complexity from the user.

3. Object-oriented programming will be required to develop portable libraries that allow the user to work at an appropriate conceptual level.

4. Work on algorithms, particularly linear algebra, is important and cannot be isolated from general library development.

5. Language standards are important. The lack of language standards is the most significant obstacle to the development of communication libraries. A language standard must emerge before a software tool "development sweep" can begin.

These are some of the major research issues in developing scalable parallel linear algebra libraries.

References