FORMAL METHODS FOR HIGH-PERFORMANCE LINEAR ALGEBRA LIBRARIES*

John A. Gunnels, Robert A. van de Geijn
The University of Texas at Austin
Austin, TX, USA

Abstract A colleague of ours, Dr. Timothy Mattson of Intel, once made the following observation: "Literature professors read literature. Computer Science professors should at least occasionally read code." The point he was making was that in order to write superior prose one needs to read good (and bad) literature. Analogously, it is our thesis that exposure to elegant (and ugly) programs tends to yield the insights which are necessary if one wishes to produce consistently well-written code.

Since the advent of high-performance distributed-memory parallel computing, the need for intelligible code has become ever greater. Development and maintenance of libraries for these kinds of architectures is simply too complex to be amenable to conventional approaches to coding. Attempting to do so has led to the production of an abundance of inefficient, anfractuous code that is difficult to maintain and nigh-impossible to upgrade.

Having struggled with these issues for more than a decade, we have arrived at a conclusion which is somewhat surprising to us: the answer is to apply formal methods from Computer Science to the development of high-performance linear algebra libraries. The resulting approach has consistently resulted in aesthetically-pleasing, coherent code that greatly facilitates performance analysis, intelligent modularity, and the enforcement of program correctness via assertions. Since the technique is completely language-independent, it lends itself equally well to a wide spectrum of programming languages (and paradigms) ranging from C and Fortran to C++ and Java. In this paper, we illustrate our observations by looking at our Formal Linear Algebra Methods Environment (FLAME).

*This work was partially supported by the Remote Exploration and Experimentation Project at Caltech's Jet Propulsion Laboratory, which is part of NASA's High Performance Computing and Communications Program, and is funded by NASA's Office of Space Science.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35407-1_22

R. F. Boisvert et al. (eds.), *The Architecture of Scientific Software*

Keywords: FLAME, linear algebra, algorithms, formal methods, LU factorization

1. INTRODUCTION

The core curriculum of any first-rate undergraduate Computer Science department includes at least one course that focuses on the formal derivation and verification of algorithms [8]. Many of us in scientific computing may have, at some point in time, hastily dismissed this approach, arguing that this is all very nice for small, simple algorithms, but an academic exercise hardly applicable in "our world." Since it is often the case that our work involves libraries comprised of hundreds of thousands or even millions of lines of code, the knee-jerk reaction that this approach is much too cumbersome to take seriously is understandable. Furthermore, the momentum of established practices and "traditional wisdom" do little if anything to dissuade one from this line of reasoning. Yet, as the result of our search for superior methods for designing and constructing high-performance parallel linear algebra libraries, we have come to the conclusion that it is *only* through the systematic approach offered by formal methods that we will be able to deliver reliable, maintainable, flexible, yet highly efficient matrix libraries even in the relatively well-understood area of (sequential and parallel) dense linear algebra.

While some would immediately draw the conclusion that a change to a more modern programming language like C++ is at least highly desirable, if not a necessary precursor to writing elegant code, the fact is that most applications which call packages like LAPACK [2] and ScaLAPACK [3] are still written in Fortran and/or C. Interfacing such an application with a library written in C++ presents certain complications. However, during the mid-nineties, the Message-Passing Interface (MPI) introduced to the scientific computing community a programming model, object-based programming, that possesses many of the advantages typically associated with the intelligent use of an object-oriented language [18]. Using objects (e.g. communicators in MPI) to encapsulate data structures and hide complexity, a much cleaner approach to coding can be achieved. Our own work on the Parallel Linear Algebra PACKage (PLAPACK) borrowed from this approach in order to hide details of data distribution and data mapping in the realm of parallel linear algebra libraries [20]. The primary concept also germane to this paper is that PLAPACK raises the level of abstraction at which one programs so that indexing is essentially removed from the code, allowing the routine to reflect the algorithm as it is naturally presented in a classroom setting. Since our initial work on PLAPACK, we have experimented with similar

interfaces in such seemingly disparate contexts as (parallel) out-of-core linear algebra packages and a low-level implementation of the sequential Basic Linear Algebra Subprograms (BLAS) [15, 6, 5, 16, 9].

Our Formal Linear Algebra Methods Environment (FLAME) is the latest step in the evolution of these systems [11]. It facilitates the use of a programming style which is equally applicable to everything from out-of-core, parallel systems to single-processor systems where cache-management is of paramount concern.

It has become apparent that what makes our task of library development more manageable is this systematic approach to deriving algorithms coupled with the abstractions we use to make our code reflect the algorithms thus produced. Further, it is from these experiences that we can confidently state that this approach to programming greatly reduces the complexity of the resultant code and does not sacrifice high performance in order to do so.

Indeed, the formal techniques which we may have dismissed as merely academic or impractical make this possible, as we attempt to illustrate in the following sections.

2. THE CASE FOR A MORE FORMAL APPROACH

Ideally, an implementation should clearly reflect the algorithm as it is presented in a classroom setting. Even better, some of the derivation of the algorithm should be apparent in the code and different variants of an algorithm should be recognizable as slight perturbations to an algorithmic "skeleton" or base code. If the code is just a mechanically-realizable, straightforward translation of the algorithm presented in class, there should be no opportunity for the introduction of logical errors or coding bugs. (Note: while we will frequently refer to translations from algorithms to code as being mechanical or automatic, this process is currently performed by us by hand.) Presumably, it should be possible to prove the algorithms correct, thus ensuring that the code is correct.

Typically, it is difficult to prove code correct precisely because one is not certain that the code truly mirrors the algorithm. With our approach, the chasm is largely bridged by the simple yet crucial fact that some very simple syntactic rewrite rules can produce the code from an algorithm expressed as one might in a classroom, using mathematical formulae and stylized matrix depictions. Since we can prove the correctness of the algorithm we wish to employ and because the correctness of the translation from algorithm to code is at least as reliable as compiler technology, the complexity of the task at hand is greatly ameliorated.

By assuming that components adhere to explicit, "contractual obligations" [1] the algorithmic proof requires little alteration in order to be applicable to the code. In the case of a library constructed entirely through the methodology presented here, these components would be composed in like manner so as to make this task manageable. This is largely due to the fact that the approach presented here leads to a software architecture layered in such a way so as to require one to rely on the correctness of a very small number of base-level modules. Since those units are small, their correctness can be established through the application of standard formal methods. It is true that, in practice, one must accept that an application will need to interface with other libraries (for example, the vendor-supplied BLAS) that are not built in a "proof-friendly" format. However, it may still be possible to establish the correctness of a program if one is careful to impose minimal obligations on these, presumably time-tested and well-documented, pieces of code.

It should be noted that the "correctness" discussed so far does not address issues of numerical stability. We make no claim regarding the stability of the resulting algorithm.

Having said this, let us clarify through a simple example.

3. A CASE STUDY: LU FACTORIZATION

We illustrate our approach by considering LU factorization without pivoting. Given an $n \times n$ matrix A we wish to compute an $n \times n$ lower triangular matrix L with unit main diagonal and an $n \times n$ upper triangular matrix U so that $A = LU$. The original matrix A is overwritten by L and U in the process.

3.1. POSSIBLE LOOP-INVARIANTS

In order to prove correctness, we must ask is what intermediate value is in A at any particular stage of the algorithm. To answer this, partition

$$A = \left(\begin{array}{c|c} A_{TL}^{(k)} & A_{TR}^{(k)} \\ \hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right), L = \left(\begin{array}{c|c} L_{TL}^{(k)} & 0 \\ \hline L_{BL}^{(k)} & L_{BR}^{(k)} \end{array}\right), U = \left(\begin{array}{c|c} U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline 0 & U_{BR}^{(k)} \end{array}\right) \tag{1}$$

where $A_{TL}^{(k)}$, $L_{TL}^{(k)}$, and $U_{TL}^{(k)}$ are all $k \times k$ matrices. Notice that "T", "B", "L", and "R" stand for Top, Bottom, Left, and Right, respectively. Notice that

$$\left(\begin{array}{c|c} A_{TL}^{(k)} & A_{TR}^{(k)} \\ \hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right) = \left(\begin{array}{c|c} L_{TL}^{(k)} & 0 \\ \hline L_{BL}^{(k)} & L_{BR}^{(k)} \end{array}\right) \left(\begin{array}{c|c} U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline 0 & U_{BR}^{(k)} \end{array}\right)$$

$$= \left(\begin{array}{c|c} L_{TL}^{(k)} U_{TL}^{(k)} & L_{TL}^{(k)} U_{TR}^{(k)} \\ \hline L_{BL}^{(k)} U_{TL}^{(k)} & L_{BL}^{(k)} U_{TR}^{(k)} + L_{BR}^{(k)} U_{BR}^{(k)} \end{array} \right)$$

so the following equalities must hold:

$$\begin{aligned} A_{TL}^{(k)} &= L_{TL}^{(k)} U_{TL}^{(k)} && (2) \\ A_{TR}^{(k)} &= L_{TL}^{(k)} U_{TR}^{(k)} && (3) \\ A_{BL}^{(k)} &= L_{BL}^{(k)} U_{TL}^{(k)} && (4) \\ A_{BR}^{(k)} &= L_{BL}^{(k)} U_{TR}^{(k)} + L_{BR}^{(k)} U_{BR}^{(k)} && (5) \end{aligned}$$

Finally, let $\hat{A}_k$ denote the matrix which holds the current intermediate result of a given algorithm for computing the LU factorization. In the following pages we will show that different conditions on the contents of $\hat{A}_k$ logically dictate different variants for computing the LU factorization, and that these different conditions can be systematically generated. Previewing this, notice that to compute the LU factorization, the submatrices of L and U are to be computed. We assume that $\hat{A}_k$ will contain partial results towards that goal. Here are some possibilities:

Condition	$\hat{A}_k$ contains
Only (2) is satisfied.	$\left(\begin{array}{c\|c} L\backslash U_{TL}^{(k)} & A_{TR}^{(k)} \\ \hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array} \right)$
Only (2) and (3) have been satisfied.	$\left(\begin{array}{c\|c} L\backslash U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array} \right)$
Only (2) and (4) have been satisfied.	$\left(\begin{array}{c\|c} L\backslash U_{TL}^{(k)} & A_{TR}^{(k)} \\ \hline L_{BL}^{(k)} & A_{BR}^{(k)} \end{array} \right)$
Only (2), (3), and (4) have been satisfied.	$\left(\begin{array}{c\|c} L\backslash U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline L_{BL}^{(k)} & A_{BR}^{(k)} \end{array} \right)$
(2), (3), and (4) have been satisfied and as much of (5) has been computed *without computing any part of* $L_{BR}^{(k)}$ *or* $U_{BR}^{(k)}$.	$\left(\begin{array}{c\|c} L\backslash U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline L_{BL}^{(k)} & A_{BR}^{(k)} - L_{BL}^{(k)} U_{TR}^{(k)} \end{array} \right)$

Here we use the notation $L\backslash U$ to denote a lower and upper triangular matrix which are stored in a square matrix by overwriting the lower and

upper triangular parts of that matrix. (Recall that L has ones on the diagonal, which need not to be stored.)

In the subsequent subsections, we describe how to derive algorithms in which the desired conditions hold. Note that in this paper we will not concern ourselves with the question of whether or not the above conditions exhaust all possibilities. However, they do give rise to many commonly discussed algorithms. For example, they yield all algorithms depicted on the cover of, and discussed in, G.W. Stewart's recent book on matrix factorization [19].

This comes as no surprise as we, like Stewart, have adopted some common implicit assumptions about both matrix partitioning and the nature of algorithmic advancement. In this paper we have restricted ourselves to a consideration of only those algorithms whose progress is "simple." That is, each iteration of the algorithm is geographically monotonic and formulaically identical. The combination of these two properties leads to algorithms whose (inductive) proofs of correctness are straightforward and whose implementations, given our framework, are virtually fool-proof.

While it is not a central focus of this paper, we also feel that it is worthwhile to point out our dissatisfaction with the categorization schemes which have traditionally been used to label algorithms of this class. In the literature one finds, for example, left- vs. right-looking algorithms [7]. The problem is that a left-looking algorithm for one operation has a very different flavor from a left-looking algorithm for another operation. Stewart, in [19] supplies novel naming conventions for some of these algorithms. We feel that the naming conventions can be made intuitively appealing and systematic, based on the work done in the inductive step of the algorithm. We intend to explain our classification further in a subsequent paper, after we have had an opportunity to evaluate it against a larger class of algorithms.

3.2. LAZY ALGORITHM

This algorithm is often referred to as a bordered algorithm in the literature. Stewart, [19] rather colorfully, refers to it as Sherman's march.

We only discuss blocked variants of the algorithms due to space limitations. In [11] we give details for unblocked algorithms.

Algorithm: Assume that only (2) has been satisfied. The question is now how to compute $\hat{A}_{k+b}$ from $\hat{A}_k$ for some small block size b (i.e.

$1 < b \ll n$). To answer this, repartition

$$A = \left(\begin{array}{c|c} A_{TL}^{(k)} & A_{TR}^{(k)} \\ \hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right) = \left(\begin{array}{c|c|c} A_{00}^{(k)} & A_{01}^{(k)} & A_{02}^{(k)} \\ \hline A_{10}^{(k)} & A_{11}^{(k)} & A_{12}^{(k)} \\ \hline A_{20}^{(k)} & A_{21}^{(k)} & A_{22}^{(k)} \end{array}\right) \tag{6}$$

where $A_{00}^{(k)}$ is $k \times k$ (and thus equal to $A_{TL}^{(k)}$), and $A_{11}^{(k)}$ is $b \times b$. Repartition L, U, and $\hat{A}_k$ conformally. Notice our assumption is that $\hat{A}_k$ holds

$$\hat{A}_k = \left(\begin{array}{c|c} L\backslash U_{TL}^{(k)} & A_{TR}^{(k)} \\ \hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right) = \left(\begin{array}{c|c|c} L\backslash U_{00}^{(k)} & A_{01}^{(k)} & A_{02}^{(k)} \\ \hline A_{10}^{(k)} & A_{11}^{(k)} & A_{12}^{(k)} \\ \hline A_{20}^{(k)} & A_{21}^{(k)} & A_{22}^{(k)} \end{array}\right)$$

The desired contents of $\hat{A}_{k+b}$ are given by

$$\hat{A}_{k+b} = \left(\begin{array}{c|c} \hat{A}_{TL}^{(k+b)} & \hat{A}_{TR}^{(k+b)} \\ \hline \hat{A}_{BL}^{(k+b)} & \hat{A}_{BR}^{(k+b)} \end{array}\right) = \left(\begin{array}{c|c|c} L\backslash U_{00}^{(k)} & U_{01}^{(k)} & A_{02}^{(k)} \\ \hline L_{10}^{(k)} & L\backslash U_{11}^{(k)} & A_{12}^{(k)} \\ \hline A_{20}^{(k)} & A_{21}^{(k)} & A_{22}^{(k)} \end{array}\right)$$

Thus, it suffices to compute $U_{01}^{(k)}$, $L_{10}^{(k)}$, $L_{11}^{(k)}$, and $U_{11}^{(k)}$.

To derive how to compute these quantities, consider

$$\begin{aligned} A &= \left(\begin{array}{c|c|c} A_{00}^{(k)} & A_{01}^{(k)} & A_{02}^{(k)} \\ \hline A_{10}^{(k)} & A_{11}^{(k)} & A_{12}^{(k)} \\ \hline A_{20}^{(k)} & A_{21}^{(k)} & A_{22}^{(k)} \end{array}\right) \\ &= \left(\begin{array}{c|c|c} L_{00}^{(k)} & 0 & 0 \\ \hline L_{10}^{(k)} & L_{11}^{(k)} & 0 \\ \hline L_{20}^{(k)} & L_{21}^{(k)} & L_{22}^{(k)} \end{array}\right) \left(\begin{array}{c|c|c} U_{00}^{(k)} & U_{01}^{(k)} & U_{02}^{(k)} \\ \hline 0 & U_{11}^{(k)} & U_{12}^{(k)} \\ \hline 0 & 0 & U_{22}^{(k)} \end{array}\right) \\ &= \left(\begin{array}{c|c|c} L_{00}^{(k)}U_{00}^{(k)} & L_{00}^{(k)}U_{01}^{(k)} & L_{00}^{(k)}U_{02}^{(k)} \\ \hline L_{10}^{(k)}U_{00}^{(k)} & L_{10}^{(k)}U_{01}^{(k)}+L_{11}^{(k)}U_{11}^{(k)} & L_{10}^{(k)}U_{02}^{(k)}+L_{11}^{(k)}U_{12}^{(k)} \\ \hline L_{20}^{(k)}U_{00}^{(k)} & L_{20}^{(k)}U_{01}^{(k)}+L_{21}^{(k)}U_{11}^{(k)} & L_{20}^{(k)}U_{02}^{(k)}+L_{21}^{(k)}U_{12}^{(k)}+L_{22}^{(k)}U_{22}^{(k)} \end{array}\right) \end{aligned}$$

This yields the equalities

$$\left(\begin{array}{c|c|c} A_{00}^{(k)} = L_{00}^{(k)}U_{00}^{(k)} & A_{01}^{(k)} = L_{00}^{(k)}U_{01}^{(k)} & A_{02}^{(k)} = L_{00}^{(k)}U_{02}^{(k)} \\ \hline A_{10}^{(k)} = L_{10}^{(k)}U_{00}^{(k)} & A_{11}^{(k)} = L_{10}^{(k)}U_{01}^{(k)} + L_{11}^{(k)}U_{11}^{(k)} & A_{12}^{(k)} = L_{10}^{(k)}U_{02}^{(k)} + L_{11}^{(k)}U_{12}^{(k)} \\ \hline A_{20}^{(k)} = L_{20}^{(k)}U_{00}^{(k)} & A_{21}^{(k)} = L_{20}^{(k)}U_{01}^{(k)} + L_{21}^{(k)}U_{11}^{(k)} & A_{22}^{(k)} = L_{20}^{(k)}U_{02}^{(k)} + L_{21}^{(k)}U_{12}^{(k)} + L_{22}^{(k)}U_{22}^{(k)} \end{array}\right) \tag{7}$$

Thus,

1 To compute $U_{01}^{(k)}$ we solve the triangular system $L_{00}^{(k)}U_{01}^{(k)} = A_{01}^{(k)}$. The result can overwrite $A_{01}^{(k)}$.

2 To compute $L_{10}^{(k)}$ we solve the triangular system $L_{10}^{(k)}U_{00}^{(k)} = A_{10}^{(k)}$. The result can overwrite $A_{10}^{(k)}$.

3 To compute $L_{11}^{(k)}$ and $U_{11}^{(k)}$ we update $A_{11}^{(k)} \leftarrow A_{11}^{(k)} - L_{10}^{(k)}U_{01}^{(k)} = A_{11}^{(k)} - A_{10}^{(k)}A_{01}^{(k)}$ after which the result can be factored into $L_{11}^{(k)}$ and $U_{11}^{(k)}$ using the unblocked algorithm. The result can overwrite $A_{11}^{(k)}$.

The preceding discussion motivates the algorithm in Fig. 1(b) for overwriting the given $n \times n$ matrix A with its LU factorization.

3.3. EAGER ALGORITHM

This algorithm is often referred to as classical Gaussian elimination.

Let us assume that (2), (3), and (4) have been satisfied, and as much of (5) as possible without completing any more of the factorization $L_{BR}U_{BR}$. Repartition A, L, U, and $\hat{A}_k$ conformally as in (6). Notice our assumption is that $\hat{A}_k$ holds

$$\hat{A}_k = \left(\begin{array}{c||c} L\backslash U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline\hline L_{BL}^{(k)} & \hat{A}_{BR}^{(k)} \end{array}\right) = \left(\begin{array}{c||c|c} L\backslash U_{00}^{(k)} & U_{01}^{(k)} & U_{02}^{(k)} \\ \hline\hline L_{10}^{(k)} & A_{11}^{(k)} - L_{10}^{(k)}U_{01}^{(k)} & A_{12} - L_{10}^{(k)}U_{02}^{(k)} \\ \hline L_{20}^{(k)} & A_{21}^{(k)} - L_{20}^{(k)}U_{01}^{(k)} & A_{22}^{(k)} - L_{20}^{(k)}U_{02}^{(k)} \end{array}\right)$$

The desired contents of $\hat{A}_{k+b}$ are given by

$$\begin{aligned} \hat{A}_{k+b} &= \left(\begin{array}{c||c} \hat{A}_{TL}^{(k+b)} & \hat{A}_{TR}^{(k+b)} \\ \hline\hline \hat{A}_{BL}^{(k+b)} & \hat{A}_{BR}^{(k+b)} \end{array}\right) \\ &= \left(\begin{array}{c|c||c} L\backslash U_{00}^{(k)} & U_{01}^{(k)} & U_{02}^{(k)} \\ \hline L_{10}^{(k)} & L\backslash U_{11}^{(k)} & U_{12}^{(k)} \\ \hline\hline L_{20}^{(k)} & L_{21}^{(k)} & A_{22}^{(k)} - L_{20}^{(k)}U_{02}^{(k)} - L_{21}^{(k)}U_{12}^{(k)} \end{array}\right) \end{aligned}$$

Thus, it suffices to compute $L\backslash U_{11}^{(k)}$, $L_{21}^{(k)}$, $U_{12}^{(k)}$, and to update $\hat{A}_{22}^{(k)} = A_{22}^{(k)} - L_{20}^{(k)}U_{02}^{(k)} - L_{21}^{(k)}U_{12}^{(k)}$. Recalling the equalities in (7) we decide

1 To compute $L_{11}^{(k)}$ and $U_{11}^{(k)}$ we factor $\hat{A}_{11}^{(k)}$ which already contains $A_{11}^{(k)} - L_{10}^{(k)}U_{01}^{(k)}$. The result can overwrite $\hat{A}_{11}^{(k)}$.

Partition $A = \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)$

where A_{TL} is 0×0

do until A_{BR} is 0×0

Repartition

$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c||c|c} A_{00} & A_{01} & A_{02} \\ \hline\hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where A_{11} is $b \times b$

(a) Eager:
$A_{11} \leftarrow \{L\backslash U\}_{11} = LU(A_{11})$
$A_{12} \leftarrow U_{12} = L_{11}^{-1} A_{12}$
$A_{21} \leftarrow L_{21} = A_{21} U_{11}^{-1}$
$A_{22} \leftarrow A_{22} - L_{21} U_{12}$

(b) Lazy:
View A_{00} as $L\backslash U_{00}$
$A_{01} \leftarrow L_{01} = L_{00}^{-1} A_{01}$
$A_{10} \leftarrow L_{10} = A_{10} U_{00}^{-1}$
$A_{11} \leftarrow L\backslash U_{11} = LU(A_{11} - L_{10}U_{01})$

(c) Row-lazy:
View A_{00} as $L\backslash U_{00}$
$A_{10} \leftarrow L_{10} = A_{10} U_{00}^{-1}$
$A_{11} \leftarrow L\backslash U_{11} = LU(A_{11} - L_{10}U_{01})$
$A_{12} \leftarrow U_{12} = L_{11}^{-1}(A_{12} - L_{10}U_{02})$

(d) Column-lazy:
View A_{00} as $L\backslash U_{00}$
$A_{01} \leftarrow U_{01} = U_{00}^{-1} A_{01}$
$A_{11} \leftarrow L\backslash U_{11} = LU(A_{11} - L_{10}U_{01})$
$A_{21} \leftarrow L_{21} = (A_{21} - L_{20}U_{01}) U_{11}^{-1}$

(e) Row-column-lazy:
$A_{11} \leftarrow L\backslash U_{11} = LU(A_{11} - L_{10}U_{01})$
$A_{12} \leftarrow U_{12} = L_{11}^{-1}(A_{12} - L_{10}U_{02})$
$A_{21} \leftarrow L_{21} = (A_{21} - L_{20}U_{01}) U_{11}^{-1}$

Continue with

$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c||c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

enddo

Figure 1 Blocked versions of LU factorization without pivoting for five commonly encountered variants. The different variants share the skeleton which partitions and repartitions the matrix. Executing the operations in one of the five boxes yields a specific algorithm.

2 To compute $U_{12}^{(k)}$ we update $\hat{A}_{12}^{(k)}$ which already contains $A_{12}^{(k)} - L_{10}^{(k)}U_{02}^{(k)}$ by solving $L_{11}^{(k)}U_{12}^{(k)} = \hat{A}_{12}^{(k)}$, overwriting the original $\hat{A}_{12}^{(k)}$.

3 To compute $L_{21}^{(k)}$ we update $A_{21}^{(k)}$ which already contains $A_{21}^{(k)} - L_{20}^{(k)}U_{01}^{(k)}$ by solving $L_{21}^{(k)}U_{11}^{(k)} = \hat{A}_{21}^{(k)}$, overwriting the original $\hat{A}_{21}^{(k)}$.

4 We then update $\hat{A}_{22}^{(k)}$ which already contains $A_{22}^{(k)} - L_{20}^{(k)}U_{02}^{(k)}$ with $\hat{A}_{22}^{(k)} - L_{21}^{(k)}U_{12}^{(k)}$ overwriting the original $\hat{A}_{22}^{(k)}$.

The resulting algorithm is given in Fig. 1(a).

3.4. OTHER ALGORITHMS

We briefly discuss the three algorithms which may be derived from the remaining three loop-invariants.

Row-lazy algorithm: As a point of reference, Stewart [19] calls this algorithm Pickett's charge south.

Let us assume that only (2) and (3) have been satisfied. Now it suffices to compute $L_{10}^{(k)}$, $L\backslash U_{11}^{(k)}$, and $U_{12}^{(k)}$. Using the same techniques as before one derives the algorithm in Fig. 1 (c). Again, this algorithm overwrites the given $n \times n$ matrix A with its LU factorization.

Column-lazy algorithm This algorithm is referred to as left-looking in [7] while Stewart [19] calls it Pickett's charge east.

Let us assume that only (2) and (4) have been satisfied. Now it suffices to compute $U_{01}^{(k)}$, $L\backslash U_{11}^{(k)}$, and $L_{21}^{(k)}$. Using the same techniques as before one derives the algorithm in Fig. 1 (d). Again, this algorithm overwrites the given $n \times n$ matrix A with its LU factorization.

Row-column-lazy algorithm This algorithm is often referred to as Crout's methods in the literature.

Let us assume that only (2), (3), and (4) have been satisfied. This time, it suffices to compute $L\backslash U_{11}^{(k)}$, $U_{12}^{(k)}$, and $L_{21}^{(k)}$, yielding the algorithm in Fig. 1 (e). Again, this algorithm overwrites a given $n \times n$ matrix A with its LU factorization.

4. SO MANY ALGORITHMS, SO LITTLE TIME

The primary motivating force behind developing a systematic framework for deriving algorithms is that, depending on architecture and/or matrix dimensions, different algorithms may exhibit different perfor-

mance characteristics. So, an algorithm which performs admirably on one architecture and/or a particular problem size may prove to be an inferior algorithm when implemented on another architecture or applied to a problem with dissimilar dimensions.

In [9] we show that the efficient, transportable implementation of matrix multiplication on a sequential architecture with a hierarchical memory requires a hierarchy of matrix algorithms whose organization, in a very real sense, mirrors that of the memory system under consideration. Perhaps surprisingly, this is necessary even when the problem size is fixed. In the same paper we describe a methodology for composing these routines. In this way, minimal coding effort is required to attain superior performance across a wide spectrum of algorithms and problem sizes. Analogously, in [10] we demonstrate that an efficient implementation of parallel matrix multiplication requires both multiple algorithms and a method for selecting the appropriate algorithm for the presented case if one is to handle operands of various sizes and shapes. In [16] we came to a similar conclusion in the context of out-of-core factorization algorithms.

Taken together, these concerns and discoveries have motivated us to create an enabling technology for automating the entire process. What we have developed, in FLAME as it couples with our design philosophy, is a major step in that direction.

5. FLAME: CODING THE ALGORITHM

In an effort to make the code look like the algorithms given in Fig. 1, while simultaneously accounting for the constraints imposed by C and Fortran, we have developed FLAME. Rather than going into great detail regarding this library of routines, we give the FLAME code for the lazy algorithm in Fig. 2.

The reader will recognize the skeleton which the five variants share, with only the code between the lines marked by `/* ***···*** */` differing between implementations. A is being passed to the routine as a data structure, `A`, which describes all attributes of this matrix, such as dimensions and method of storage. Inquiry routines like `FLA_Obj_length` are used to extract information such as the row dimension of the matrix. Finally, `ATL`, `A00`, etc. are simply references into the original array described by `A`.

If one is familiar with the coding conventions used to name the BLAS kernels, it is clear that by substituting different updates to submatrices in lines 24–35, one can achieve the different variants which implement LU factorization without pivoting.

```
#include "FLAME.h"

void FLA_LU_nopivot_lazy( FLA_Obj A, nb_alg )
{
  FLA_Obj     ATL, ATR,   A00, A01, A02,
              ABL, ABR,   A10, A11, A12,
                          A20, A21, A22;

  FLA_Part_2x2( A,  &ATL, /**/ &ATR,
                  /* ************** */
                  &ABL, /**/ &ABR,
            /* with */ 0, /* by */ 0, /* submatrix */ FLA_TL );

  while ( b=min(FLA_Obj_length( ABR ), nb_alg) != 0 ){

    FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
                        /* ************** */   /* ******************** */
                                /**/            &A10, /**/ &A11, &A12,
                          ABL, /**/ ABR         &A20, /**/ &A21, &A22,
            /* with */ b, /* by */ b, /* A11 split from */ FLA_BR );

    /* ************************************************************* */

    FLA_Trsm( FLA_RIGHT, FLA_UPPER_TRIANGULAR,
              FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
              ONE, A00, A10 );
    FLA_Trsm( FLA_LEFT,  FLA_LOWER_TRIANGULAR,
              FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
              ONE, A00, A01 );
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
              MINUS_ONE, A10, A01, ONE, A11 );
    FLA_LU_nopivot_level2( A11 );

    /* ************************************************************* */

    FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,      A00, A01, /**/ A02,
                                    /**/            A10, A11, /**/ A12,
                           /* ************** */ /* ****************** */
                              &ABL, /**/ &ABR,      A20, A21, /**/ A22,
            /* with A11 added to submatrix */ FLA_TL );
  }
}
```

Figure 2 A skeleton for the C implementation of many of the blocked LU factorization algorithms using FLAME.

A number of issues related to this style of coding are discussed in detail in [11], including:

Proving the code correct: We argue that the approach we take to deriving the algorithm allows the algorithm to be proved correct. We claim that since the code closely resembles the algorithm, the proof for

the algorithm largely carries over to a proof for the correctness of the code.

Fortran: Again using MPI as an inspiration, a Fortran interface is available for FLAME. Examples of Fortran code are available on the FLAME web page (see "Additional Information" at the end of the paper).

Pivoting: We show how LU factorization with partial pivoting can be elegantly expressed in our algorithmic format.

Performance: We show that the FLAME approach to coding linear algebra algorithms does not hinder high performance.

6. RELATED WORK

Advances in software engineering for scientific applications has often been led by libraries for dense linear algebra operations. These advances include interfaces like the BLAS and libraries like EISPACK [17], LINPACK [4], and LAPACK [2].

More recently, a great deal of work has been devoted to implementing the bulk of dense linear algebra algorithms by optimizing only matrix-matrix multiplication [14]. This work was based on a careful study of memory hierarchies and how best to construct an efficient implementation of the entire BLAS library with a minimum of coding. FLAME has been used to quickly and reliably duplicate and extend these efforts.

The true complexity of indexing became painfully obvious with the advent of distributed memory architectures and linear algebra libraries for these kinds of machines [3]. PLAPACK was developed in a response to dealing with this complexity and that process lead to many of the insights in this paper.

A number of recent efforts explore hierarchical data structures for storing matrices [13, 12]. The idea is that by storing matrices by blocks rather than row- or column-major ordering, data re-use in caches can be enhanced. By combining this with recursive algorithms which exploit this data structure, impressive performance improvements have been demonstrated. Notice that these more complex data structures for sequential algorithms introduce a complexity similar to that encountered when data are distributed to the memories of a distributed-memory architecture. Since PLAPACK effectively addressed that problem for those architectures, we have strong evidence that FLAME can be extended to accomodate more complex data structures for hierarchical memories.

7. CONCLUSION

In this paper, we have illustrated that a more formal approach to the design and implementation of matrix algorithms, combined with the

right level of abstraction for coding, leads to a software architecture for linear algebra libraries which is dramatically different than the one resulting from the more traditional approaches used by packages such as LINPACK, LAPACK, and ScaLAPACK. The approach is such that the library developer is forced to give careful attention to the derivation of the algorithm. The benefit is that the code is a direct translation of the resulting algorithm, greatly reducing opportunities for the introduction of common bugs related to indexing. Our experience shows that there is no significant loss of performance. Indeed, since more flexible algorithms can now be developed we often observe a performance benefit from the approach.

Notice that throughout the paper we concentrate on the correctness of the algorithm. As we said earlier in the paper, this is not the same as proving that the algorithm is numerically stable. While we do not claim that our methodology will automatically generate stable algorithms, we do claim that the skeleton used to express the algorithm, and to implement the code, can be used to implement extant algorithms with known numerical properties. It also facilitates the discovery and implementation of new algorithms for which numerical properties can be subsequently established.

The nature of the design process and the tight coupling between algorithm and implementation have many advantages. Currently, we are pursuing a project wherein we exploit this systematic approach in order to automatically generate entire (parallel) linear algebra libraries along with run-time estimates for the cost of the generated routines.

Additional Information

For more information: `http://www.cs.utexas.edu/users/flame/`.

Acknowledgments

We thank Fred Gustavson of IBM Research for his feedback on this paper.

References

[1] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM TOSEM*, 6(3):213–249, 1997.

[2] E. Anderson, Z. Bai, J. Demmel, J. E. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide.* SIAM, Philadelphia, 1992.

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

[4] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide.* SIAM, Philadelphia, 1979.

[5] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[6] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[7] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers.* SIAM, Philadelphia, PA, 1991.

[8] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[9] John Gunnels, Greg Henry, and Robert van de Geijn. Toward dynamic high-performance matrix multiplication kernels. Technical report, Department of Computer Sciences, The University of Texas at Austin, in preparation.

[10] John Gunnels, Calvin Lin, Greg Morrow, and Robert van de Geijn. A flexible class of parallel matrix multiplication algorithms. In *Proceedings of First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (1998 IPPS/SPDP '98)*, pages 110–116, 1998.

[11] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. Formal linear algebra methods environment (flame):overview. FLAME Working Note #1 CS-TR-00-28, Department of Computer Sciences, The University of Texas at Austin, NOV 2000.

[12] F. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling. Recursive blocked data formats and blas's for dense linear algebra algorithms. In B. Kågström et al., editor, *Applied Parallel Computing, Large Scale Scientific and Industrial Problems*, volume 1541 of *Lecture Notes in Computer Science*, pages 195–206. Springer-Verlag, 1998.

[13] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, November 1997.

[14] B. Kagstrom, P. Ling, and C. Van Loan. GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. *TOMS*, 24(3):268–302, 1998.

[15] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5(3):308–323, Sept. 1979.

[16] Wesley C. Reiley and Robert A. van de Geijn. POOCLAPACK: Parallel Out-of-Core Linear Algebra Package. Technical Report CS-TR-99-33, Department of Computer Sciences, The University of Texas at Austin, Nov. 1999.

[17] B. T. Smith et al. *Matrix Eigensystem Routines – EISPACK Guide.* Lecture Notes in Computer Science 6. Springer-Verlag, New York, second edition, 1976.

[18] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference.* The MIT Press, 1996.

[19] G. W. Stewart. *Matrix Algorithms Volume 1: Basic Decompositions.* SIAM, 1998.

[20] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package.* The MIT Press, 1997.

DISCUSSION

Speaker: Robert van de Geijn

Mladen Vouk : There is an initial gain in productivity if formal methods are used. However, to optimize the result, one needs to do additional work. What is the ratio of formal development effort to traditional effort, and what fraction of the traditional effort has to be *added* to optimize the "formal" code?

Robert van de Geijn : Regarding the ratio between development time using a traditional approach versus our more formal approach, a full implementation of all sequential level-3 BLAS (excluding matrix-matrix multiplication) can easily require months of time on the part of an experienced programmer, and would require extensive testing before correctness were established. Using our approach, one person (myself) implemented all these operations in a matter of about 10 hours, including testing, and I will give $100 to anyone who finds a bug in the code. Regarding the additional optimization, the implementation of the LU factorization with pivoting took about an hour. Optimizing the two routines that really improved performance, the pivoting routine and the level-2 BLAS based LU factorization with pivoting, takes 10-20 lines of code, less than 30 minutes of coding.

Fred Gustavson : A comment: I consider your formal approach to be a significant contribution because it captures in a succinct manner the underlying linear algebra theory, translating it into C and Fortran programs in an automatic way. Additionally, I point out that linear algebra algorithms can be expressed recursively and that you get different algorithms than the ones that your formal approach considers. Finally, it is possible to represent a matrix using a different data structure than those currently supported by C and/or Fortran. Expressing your formal algorithms in terms of these new data representations should lead to higher performance on today's processors.

Robert van de Geijn : In some sense recursion is covered since implementation of the smaller subproblems is not explicitly addressed, and is typically accomplished through recursive calls. Indeed, recursion was used to improve performance. A more direct treatment of this would be a nice addition to the theory. As for the new data representations, notice that the issues one has to deal are similar to those encountered on distributed memory architectures. Since much of the inspiration for FLAME came from our Parallel Linear Algebra Package (PLAPACK), we believe our formal approach can very elegantly and effectively address more complex data representations.

Margaret Wright : Developers of existing codes, like LAPACK and ScaLAPACK, might argue that users do not need to read the underlying library code, and hence addressing readability is not important.
Robert van de Geijn : This is fine if one treats a library as a black box, but often modifications must be made to existing functionality.
David Walker : Where is parameter and error checking performed in your matrix multiply code?
Robert van de Geijn : In PLAPACK we separate the parameter checking into a separate routine, which can be called optionally. In FLAME we intend to take the same approach.
Vladimir Getov : You mentioned that your implementation following the formal methods approach outperforms ScaLAPACK. Could you provide some insight as to why your performance results are better?
Robert van de Geijn : By coding at a higher level of abstraction, one can implement more complex algorithms, which attain better performance. While these same algorithms can sometimes also be implemented using more traditional coding styles, e.g., using ScaLAPACK, the indexing becomes sufficiently complex that the resulting codes are frequently not manageable.