Formal Methods for High-Performance Linear Algebra Libraries*

John Gunnels† Robert van de Geijn‡

A technical paper submitted to the Working Conference on
"Software Architectures for Scientific Computing Applications"

July 14, 2000

Abstract

A colleague of ours from industry 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.

It is true that such libraries are often called from FORTRAN or C and, therefore, interoperability requirements sometimes inhibit our ability to use more modern object-oriented languages. However, the widely-held belief that elegance must be sacrificed in order to wring out all of the available performance from a given computer architecture is neither a logical consequent of nor necessitated by this very real, practical consideration.

Having struggled with these issues for more than a decade, we have arrived at a conclusion that 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 to graphical programming languages like those used for LabView. In this paper, we illustrate our observations by looking at the development of the Parallel Linear Algebra Package (PLAPACK).

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 [6]. Many of us in scientific

*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.

†Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, gunnels@cs.utexas.edu.

‡Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, rvdg@cs.utexas.edu. (Corresponding author)

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. The momentum of established practices and "traditional wisdom" do little if anything to dissuade one from this line of reasoning. Yet, as a result of our search for superior approaches to software engineering 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 dense linear algebra. In this paper we attempt to make this case.

Advances in software engineering for scientific libraries has often been led by libraries for dense linear algebra operations. First, there was EISPACK [15], perhaps the first package that paid careful attention to the numerical stability of the underlying algorithms. In the mid-seventies, there was the introduction of the Basic Linear Algebra Subprograms (BLAS) [10]. At that time the BLAS were a set of vector operations that allowed libraries to attain high performance on vector supercomputers while remaining highly portable between platforms and simultaneously enhancing modularity and code readability. The first successful library to exploit these BLAS was LINPACK [3]. By the late eighties, it was recognized that to overcome the gap between processor and memory performance on modern microprocessors it was necessary to reformulate matrix operations in terms of matrix multiplication-like operations, the level-3 BLAS [4]. LAPACK, first introduced in the early nineties, is a high-performance package for linear algebra operations written in terms of the level-3 BLAS [1]. LAPACK offers a functionality that is a superset of LINPACK and EISPACK [15] while achieving high performance on essentially all sequential and shared-memory architectures in a portable fashion.

With the advent of distributed memory parallel architectures, a parallel version of LAPACK, called ScaLAPACK [2], was developed. A major design goal of the ScaLAPACK project was to preserve and reuse as much code from LAPACK as possible. Thus, all layers in the ScaLAPACK software architecture are designed to closely resemble similar layers in the LAPACK software architecture. It was this decision that greatly complicated the implementation of ScaLAPACK. In essence, the introduction of data distribution (across memories) creates a problem that is not unlike that of data structures required for storing sparse matrices. The mapping from indices to matrix element(s) was no longer a simple one. Combining this fact with the monolithic structure of the software quickly led to code that was highly complicated to construct and extremely difficult to maintain.

While some would immediately draw the conclusion that a change to a more modern programming language like C++ is therefore necessary, the fact is that most applications that call packages like LAPACK and ScaLAPACK are still written in FORTRAN and/or C. Interfacing such an application with a library written in C++ poses certain problems. 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 [16]. 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 [18]. 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 BLAS. 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.

Over the last seven or eight years 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. Indeed, it is exactly the techniques that we may have at one time dismissed as merely academic or impractical which make this possible, as we will attempt to illustrate in the following sections.

2 A Case for a More Formal Approach

Ideally, the implementation of an algorithm 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 "template" or base code. If the code is just a mechanically-realizable translation of the algorithm presented in class, there should be no opportunity for the introduction of logical errors or coding bugs. Presumably, it should be possible to prove the algorithms correct, thus ensuring the code is correct. Previewing the next sections, a typical algorithm for a blocked version of classical Gaussian elimination of a matrix is presented in Fig. 2(d). In Fig. 5 we show how this can be translated to a code that has the desired properties. Different variants of the LU factorization are given in Fig. 1–2. The reader should be able to see that implementation requires only minor modification of Fig. 5.

Typically, it is difficult to prove code correct precisely because one is unsure 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 re-write 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 (the proof is generally in the construction of the algorithm, but that is of little consequence) and because the correctness of translation from algorithm to code is at least as reliable as compiler technology, the complexity of the task at hand is greatly ameliorated. Namely, components are expected to live up to certain "contractual obligations." 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 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 likely 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.

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[1]. Given $n \times n$ matrix A we wish to compute $n \times n$ lower triangular matrix L with unit main diagonal and $n \times n$ upper triangular matrix U so that $A = LU$. The original matrix A is overwritten by L and U in the process.

3.1 A simple derivation

The usual derivation of an algorithm for the LU factorization proceeds as follows:

[1]We, of course, realize pivoting is generally required for stability. However, this simple example illustrates our point and pivoting can be incorporated into our scheme.

Partition

$$A = \left(\begin{array}{c||c} \alpha_{11} & a_{12}^T \\ \hline\hline a_{21} & A_{22} \end{array}\right), L = \left(\begin{array}{c||c} 1 & 0 \\ \hline\hline l_{21} & L_{22} \end{array}\right), \text{ and } U = \left(\begin{array}{c||c} \upsilon_{11} & u_{12}^T \\ \hline\hline 0 & U_{22} \end{array}\right)$$

Now $A = LU$ translates to

$$\left(\begin{array}{c||c} \alpha_{11} & a_{12}^T \\ \hline\hline a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c||c} 1 & 0 \\ \hline\hline l_{21} & L_{22} \end{array}\right) \left(\begin{array}{c||c} \upsilon_{11} & u_{12}^T \\ \hline\hline 0 & U_{22} \end{array}\right) = \left(\begin{array}{c||c} \upsilon_{11} & u_{12}^T \\ \hline\hline l_{21}\upsilon_{11} & l_{21}u_{12}^T + L_{22}U_{22} \end{array}\right)$$

so that the following equalities must hold:

$$\begin{array}{rcl||rcl} \alpha_{11} & = & \upsilon_{11} & a_{12}^T & = & u_{12}^T \\ \hline\hline a_{21} & = & \upsilon_{11} l_{21} & A_{22} & = & l_{21}u_{12}^T + L_{22}U_{22} \end{array}$$

Finally, we arrive at the following algorithm

- Overwrite α_{11} and a_{12}^T with υ_{11} and u_{12}^T, respectively (No-op).
- Update $a_{21} \leftarrow l_{21} = a_{21}/\upsilon_{11}$.
- Update $A_{22} \leftarrow A_{22} - l_{21}u_{12}^T$.
- Recursively factor $A_{22} \rightarrow L_{22}U_{22}$.

While the algorithm is formulated as a tail-recursive algorithm, it is usually implemented as a loop.

3.2 But what intermediate value is in the matrix?

One question we one must ask to show correctness is what intermediate value is in A at any particular stage of the algorithm. To answer this, partition

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

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

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

and thus the following equalities must hold:

$$A_{TL}^{(k)} = L_{TL}^{(k)}U_{TL}^{(k)} \tag{2}$$

$$A_{TR}^{(k)} = L_{TL}^{(k)}U_{TR}^{(k)} \tag{3}$$

$$A_{BL}^{(k)} = L_{BL}^{(k)}U_{TL}^{(k)} \tag{4}$$

$$A_{BR}^{(k)} = L_{BL}^{(k)}U_{TR}^{(k)} + L_{BR}^{(k)}U_{BR}^{(k)} \tag{5}$$

Finally, let $\hat{A}_k$ equal a matrix that holds the current intermediate result of a given algorithm for computing the LU factorization. We will show in the following pages that different conditions on the contents of $\hat{A}_k$ logically dictate different variants for computing the LU factorization, and these different conditions can be systematically generated. Previewing this, notice that to compute the LU factorization, the submatrices of L and U must be computed. We assume that $\hat{A}_k$ must 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* $U_{BR}^{(k)}$.	$\left(\begin{array}{c\|c} L\backslash U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline L_{BL}^{(k)} & A_{BR}^{(k)} - L_{BL}^{(k)}U_{TR}^{(k)} \end{array}\right)$

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 the above conditions exhaust all possibilities. However, they do give rise to all commonly discussed algorithms. For example, they yield all algorithms depicted on the cover of Stewart's recent book on matrix factorization [17].

3.3 Lazy Algorithm

This algorithm is often referred to as a bordered algorithm in the literature. Stewart, [17] rather colorfully, refers to it as Sherman's march.

Unblocked Algorithm

Let us assume that only (2) has been satisfied. The question becomes how to compute $\hat{A}_{k+1}$ from $\hat{A}_k$. To answer this, repartition

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

where $A_{00}^{(k)}$ is $k \times k$ (and thus equal to $A_{TL}^{(k)}$), and $\alpha_{11}^{(k)}$ is a scalar. Repartition L, U, and $\hat{A}_k$ conformally. Notice that wish to change the contents of the current matrix from $\hat{A}_k$ to $\hat{A}_{k+1}$ or

$$\left(\begin{array}{c\|c} L\backslash U_{TL}^{(k)} & A_{TR}^{(k)} \\ \hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right) = \left(\begin{array}{c\|c|c} L\backslash U_{00}^{(k)} & a_{01}^{(k)} & A_{02}^{(k)} \\ \hline a_{10}^{(k)\,T} & \alpha_{11}^{(k)} & a_{12}^{(k)\,T} \\ \hline A_{20}^{(k)} & a_{21}^{(k)} & A_{22}^{(k)} \end{array}\right) \text{ to } \left(\begin{array}{c\|c} \hat{A}_{TL}^{(k+1)} & \hat{A}_{TR}^{(k+1)} \\ \hline \hat{A}_{BL}^{(k+1)} & \hat{A}_{BR}^{(k+1)} \end{array}\right) = \left(\begin{array}{c|c\|c} L\backslash U_{00}^{(k)} & u_{01}^{(k)} & A_{02}^{(k)} \\ \hline l_{10}^{(k)\,T} & \upsilon_{11}^{(k)} & a_{12}^{(k)\,T} \\ \hline A_{20}^{(k)} & a_{21}^{(k)} & A_{22}^{(k)} \end{array}\right)$$

Thus, it suffices to compute $u_{01}^{(k)}$, $l_{10}^{(k)}$, and $\upsilon_{11}^{(k)}$. Now, since

$$\left(\begin{array}{c\|c|c} A_{00}^{(k)} & a_{01}^{(k)} & A_{02}^{(k)} \\ \hline a_{10}^{(k)\,T} & \alpha_{11}^{(k)} & a_{12}^{(k)\,T} \\ \hline A_{20}^{(k)} & a_{21}^{(k)} & A_{22}^{(k)} \end{array}\right) = \left(\begin{array}{c\|c|c} L_{00}^{(k)} & 0 & 0 \\ \hline l_{10}^{(k)\,T} & 1 & 0 \\ \hline L_{20}^{(k)} & l_{21}^{(k)} & L_{22}^{(k)} \end{array}\right) \left(\begin{array}{c\|c|c} U_{00}^{(k)} & u_{01}^{(k)} & U_{02}^{(k)} \\ \hline 0 & \upsilon_{11}^{(k)} & u_{12}^{(k)\,T} \\ \hline 0 & 0 & U_{22}^{(k)} \end{array}\right)$$

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

Partition

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

$a_{10} \leftarrow l_{10} = U_{00}^{-T} a_{10}$

$a_{01} \leftarrow u_{01} = L_{00}^{-1} a_{01}$

$\alpha_{11} \leftarrow \upsilon_{11} = \alpha_{11} - l_{10}^T u_{01}$

Continue with

$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c||c} L\backslash U_{00} & \mathbf{u_{01}} & A_{02} \\ \hline \mathbf{l_{10}^T} & \upsilon_{11} & a_{12}^T \\ \hline\hline A_{20} & a_{21} & A_{22} \end{array}\right)$$

enddo

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
Determine block size b
Partition

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

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

$A_{10} \leftarrow L_{10} = A_{10} U_{00}^{-1}$

$A_{01} \leftarrow U_{01} = L_{00}^{-1} A_{01}$

$A_{11} \leftarrow A_{11} - L_{10} U_{01}$

$A_{11} \leftarrow L\backslash U_{11} = \text{LU fact}(A_{11})$

Continue with

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

enddo

Figure 1: Unblocked and blocked versions of the lazy variant for computing the LU factorization of square matrix A without pivoting.

$$= \left(\begin{array}{c||c|c} L_{00}^{(k)} U_{00}^{(k)} & L_{00}^{(k)} u_{01}^{(k)} & L_{00}^{(k)} U_{02}^{(k)} \\ \hline\hline l_{10}^{(k)\,T} U_{00}^{(k)} & l_{10}^{(k)\,T} u_{01}^{(k)} + \upsilon_{11}^{(k)} & l_{10}^{(k)\,T} U_{02}^{(k)} + u_{12}^{(k)\,T} \\ \hline L_{20}^{(k)} U_{00}^{(k)} & L_{20}^{(k)} U_{01}^{(k)} + l_{21}^{(k)} \upsilon_{11}^{(k)} & L_{20}^{(k)} U_{02}^{(k)} + l_{21}^{(k)} u_{12}^{(k)\,T} + L_{22}^{(k)} U_{22}^{(k)} \end{array}\right)$$

we find that the following equalities must hold:

$$\begin{array}{l||l|l} A_{00}^{(k)} = L_{00}^{(k)} U_{00}^{(k)} & a_{01}^{(k)} = L_{00}^{(k)} u_{01}^{(k)} & A_{02}^{(k)} = L_{00}^{(k)} U_{02}^{(k)} \\ \hline\hline a_{10}^{(k)\,T} = l_{10}^{(k)\,T} U_{00}^{(k)} & \alpha_{11}^{(k)} = l_{10}^{(k)\,T} u_{01}^{(k)} + \upsilon_{11}^{(k)} & a_{12}^{(k)\,T} = l_{10}^{(k)\,T} U_{02}^{(k)} + u_{12}^{(k)\,T} \\ \hline A_{20}^{(k)} = L_{20}^{(k)} U_{00}^{(k)} & a_{21}^{(k)} = L_{20}^{(k)} U_{01}^{(k)} + l_{21}^{(k)} \upsilon_{11}^{(k)} & A_{22}^{(k)} = L_{20}^{(k)} U_{02}^{(k)} + l_{21}^{(k)} u_{12}^{(k)\,T} + L_{22}^{(k)} U_{22}^{(k)} \end{array} \tag{6}$$

To compute $u_{01}^{(k)}$ we must solve the triangular system $L_{00}^{(k)} u_{01}^{(k)} = a_{01}^{(k)}$. The result can overwrite $a_{01}^{(k)}$. To compute $l_{10}^{(k)}$ we must solve the triangular system $U_{00}^{(k)\,T} l_{10}^{(k)} = a_{10}^{(k)}$. The result can overwrite $a_{10}^{(k)}$. To compute $\upsilon_{11}^{(k)}$ we merely compute $\upsilon_{11}^{(k)} = \alpha_{11}^{(k)} - l_{10}^{(k)\,T} u_{01}^{(k)} = \alpha_{11}^{(k)} - a_{10}^{(k)\,T} \hat{a}_{01}^{(k)}$. The result can overwrite $\alpha_{11}^{(k)}$.

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

Blocked Algorithm

For performance reasons it becomes beneficial to derive a blocked version of the above-presented algorithm. The derivation closely follows that of the unblocked algorithm: Again assume that only (2) has been satisfied. The question is now how to directly compute $\hat{A}_{k+b}$ from $\hat{A}_k$ for some small block size b (i.e. $1 < b \ll k$). To answer this, repartition

$$A = \left(\begin{array}{c||c} A_{TL}^{(k)} & A_{TR}^{(k)} \\ \hline\hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right) = \left(\begin{array}{c||c|c} A_{00}^{(k)} & A_{01}^{(k)} & A_{02}^{(k)} \\ \hline\hline A_{10}^{(k)} & A_{11}^{(k)} & A_{12}^{(k)} \\ \hline A_{20}^{(k)} & A_{21}^{(k)} & A_{22}^{(k)} \end{array}\right) \tag{7}$$

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 that our assumption is that $\hat{A}_k$ holds

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

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

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

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

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

This yields the equalities

$$\begin{array}{c||c|c} A_{00}^{(k)} = L_{00}^{(k)}U_{00}^{(k)} & A_{01}^{(k)} = L_{00}^{(k)}U_{01}^{(k)} & A_{02}^{(k)} = L_{00}^{(k)}U_{02}^{(k)} \\ \hline\hline A_{10}^{(k)} = L_{10}^{(k)}U_{00}^{(k)} & A_{11}^{(k)} = L_{10}^{(k)}U_{01}^{(k)} + L_{11}^{(k)}U_{11}^{(k)} & A_{12}^{(k)} = L_{10}^{(k)}U_{02}^{(k)} + L_{11}^{(k)}U_{12}^{(k)} \\ \hline A_{20}^{(k)} = L_{20}^{(k)}U_{00}^{(k)} & A_{21}^{(k)} = L_{20}^{(k)}U_{01}^{(k)} + L_{21}^{(k)}U_{11}^{(k)} & A_{22}^{(k)} = L_{20}^{(k)}U_{02}^{(k)} + L_{21}^{(k)}U_{12}^{(k)} + L_{22}^{(k)}U_{22}^{(k)} \end{array} \tag{8}$$

Thus

1. To compute $U_{01}^{(k)}$ we must 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 must 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 must 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 above discussion motivates the algorithm in Fig. 1 (right) for overwriting the given $n \times n$ matrix A with its LU factorization. A careful analysis shows that the blocked algorithm does not incur even a single extra computation relative to the unblocked algorithm.

3.4 Row-Lazy Algorithm

As a point of reference, Stewart [17] calls this algorithm Pickett's charge south.

Let us assume that only (2) and (3) have been satisfied. We will now discuss only a blocked algorithm that computes $\hat{A}_{k+b}$ from $\hat{A}_k$ while maintaining these conditions.

Repartition A, L, U, and $\hat{A}_k$ conformally as in (7). Notice that our assumption is that $\hat{A}_k$ holds

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

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

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

Thus, it suffices to compute $L_{10}^{(k)}$, $L\backslash U_{11}^{(k)}$, and $U_{12}^{(k)}$. Recalling the equalities in (8) we notice that

1. To compute $L_{10}^{(k)}$ we must solve the triangular system $L_{10}^{(k)} U_{00}^{(k)} = A_{10}^{(k)}$. The result can overwrite $A_{10}^{(k)}$.
2. To compute $L_{11}^{(k)}$ and $U_{11}^{(k)}$ we must 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 $U_{11}^{(k)}$. The result can overwrite $A_{11}^{(k)}$.
3. To compute $U_{12}^{(k)}$ we must update $A_{12}^{(k)} \leftarrow A_{12}^{(k)} - L_{10}^{(k)} U_{02}^{(k)}$ after which we must solve the triangular system $L_{11}^{(k)} U_{12}^{(k)} = A_{12}^{(k)}$, overwriting the original $A_{12}^{(k)}$.

These steps and the preceding discussion lead one directly to the algorithm in 2(a).

3.5 Column-Lazy Algorithm

This algorithm is referred to as a left-looking algorithm in [5]. Stewart [17] 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 derives the algorithm in Fig. 2 (b). Again, this algorithm overwrites given $n \times n$ matrix A with its LU factorization.

3.6 Row-Column-Lazy Algorithm

This algorithm is referred to as Krout'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. 2 (c). Again, this algorithm overwrites given $n \times n$ matrix A with its LU factorization.

3.7 Eager algorithm

This algorithm is often referred to as classical Gaussian elimination.

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
Determine block size b
Partition
$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c||c|c} L\backslash U_{00} & U_{01} & U_{02} \\ \hline\hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$
where A_{11} is $b \times b$

$A_{10} \leftarrow L_{10} = A_{10} U_{00}^{-1}$

$A_{11} \leftarrow A_{11} - L_{10} U_{01}$

$A_{11} \leftarrow L\backslash U_{11} = \text{LU fact}(A_{11})$

$A_{12} \leftarrow A_{12} - L_{10} U_{02}$

$A_{12} \leftarrow U_{12} = L_{11}^{-1} A_{12}$

Continue with
$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c||c} L\backslash U_{00} & U_{01} & U_{02} \\ \hline \mathbf{L_{10}} & \mathbf{L\backslash U_{11}} & \mathbf{U_{12}} \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right)$$
enddo

(a)

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
Determine block size b
Partition
$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c||c|c} L\backslash U_{00} & A_{01} & A_{02} \\ \hline\hline L_{10} & A_{11} & A_{12} \\ \hline L_{20} & A_{21} & A_{22} \end{array}\right)$$
where A_{11} is $b \times b$

$A_{01} \leftarrow U_{01} = L_{00}^{-1} A_{01}$

$A_{11} \leftarrow A_{11} - L_{10} U_{01}$

$A_{11} \leftarrow L\backslash U_{11} = \text{LU fact}(A_{11})$

$A_{21} \leftarrow A_{21} - L_{20} U_{01}$

$A_{21} \leftarrow L_{21} = A_{21} U_{11}^{-1}$

Continue with
$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c||c} L\backslash U_{00} & \mathbf{U_{01}} & A_{02} \\ \hline L_{10} & \mathbf{L\backslash U_{11}} & A_{12} \\ \hline\hline L_{20} & \mathbf{L_{21}} & A_{22} \end{array}\right)$$
enddo

(b)

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
Determine block size b
Partition
$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c||c|c} L\backslash U_{00} & U_{01} & U_{02} \\ \hline\hline L_{10} & A_{11} & A_{12} \\ \hline L_{20} & A_{21} & A_{22} \end{array}\right)$$
where A_{11} is $b \times b$

$A_{11} \leftarrow A_{11} - L_{10} U_{01}$

$A_{11} \leftarrow L\backslash U_{11} = \text{LU fact}(A_{11})$

$A_{12} \leftarrow A_{12} - L_{10} U_{02}$

$A_{12} \leftarrow U_{12} = L_{11}^{-1} A_{12}$

$A_{21} \leftarrow A_{21} - L_{20} U_{01}$

$A_{21} \leftarrow L_{21} = A_{21} U_{11}^{-1}$

Continue with
$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c||c} L\backslash U_{00} & U_{01} & U_{02} \\ \hline L_{10} & \mathbf{L\backslash U_{11}} & \mathbf{U_{12}} \\ \hline\hline L_{20} & \mathbf{L_{21}} & A_{22} \end{array}\right)$$
enddo

(c)

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
Determine block size b
Partition
$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c||c|c} L\backslash U_{00} & U_{01} & U_{02} \\ \hline\hline L_{10} & \hat{A}_{11} & \hat{A}_{12} \\ \hline L_{20} & \hat{A}_{21} & \hat{A}_{22} \end{array}\right)$$
where A_{11} is $b \times b$

$\hat{A}_{11} \leftarrow L\backslash U_{11} = \text{LU fact}(\hat{A}_{11})$

$\hat{A}_{12} \leftarrow U_{12} = L_{11}^{-1} \hat{A}_{12}$

$\hat{A}_{21} \leftarrow L_{21} = \hat{A}_{21} U_{11}^{-1}$

$\hat{A}_{22} \leftarrow \hat{A}_{22} - L_{21} U_{12}$

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

(d)

Figure 2: Blocked versions for computing the LU factorization of square matrix A without pivoting using (a) row-lazy, (b) column-lazy, (c) row-column-lazy, and (d) eager algorithms.

Finally, 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 (7). Notice that 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

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

Thus, it suffices to compute $L\backslash U_{11}^{(k)}$, $L_{21}^{(k)}$, $U_{12}^{(k)}$, and updating $\hat{A}_{22}^{(k)} = A_{22}^{(k)} - L_{20}^{(k)}U_{02}^{(k)} - L_{21}^{(k)}U_{12}^{(k)}$. Recalling the equalities in (8) we notice that

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

2. To compute $U_{12}^{(k)}$ we must 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 $U_{12}^{(k)}$ we must update $A_{12}^{(k)}$ 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 $\hat{A}_{12}^{(k)}$.

4. We must 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 $\hat{A}_{12}^{(k)}$.

The resulting algorithm is given in Fig. 2(d). **Notice** that this final algorithm is equivalent to the algorithm derived in Section 3.1.

4 A Recipe for Deriving Algorithms

Note that the derivations of the different algorithms detailed above are extremely systematic. Indeed, the following recipe can be used:

1. State the operation to be performed.

2. Partition the operands. Notice that some justification is needed for the particular way in which they are partitioned. For the LU, this has to do with the fact that blocks of zeroes must be isolated in L and U. Details go beyond the scope of this paper.

3. Multiply out the partitioned matrices.

4. By equating submatrices on the left and right of the equal sign of the equality generated in Step 3, derive the equalities that must hold.

5. Pick a loop invariant from possible loop invariants that satisfies the equalities given in Step 4.

6. From that loop invariant derive the steps required to maintain the loop invariant while moving the algorithm forward in the desired direction. This requires the following substeps:
   (a) Repartition so as to expose the boundaries after they are moved.
   (b) Indicate the current contents for the repartitioned matrices.
   (c) Indicate the desired contents for the repartitioned matrices such that the loop invariant is maintained.
   (d) Derive the steps required to achieve the desired contents.
7. Update the partitioning of the matrices.
8. State the algorithm.
9. Classify the algorithm. We have developed a systematic way of classifying the derived algorithms. While we use this classification in the labeling of the algorithms in the previous section, we will not go into detail here.

5 So Many Algorithms, So Little Time

So, why should we be concerned with a spectrum of algorithms for a given operation rather than picking the first one that yields good performance? Unfortunately space limitations prevent us from completely addressing this question. We limit ourselves to some general comments and pointers to relevant papers.

The primary motivating force behind developing a systematic framework for deriving algorithms is that, depending on architecture and/or matrix dimensions, different algorithms exhibit different performance characteristics. An algorithm that 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 of a different size. In [8] 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 if the problem size were fixed. In that same paper we supply a methodology for intelligently composing these routines in such a way so as to achieve efficient, flexible performance with minimal coding effort. Analogously, in [9] 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 [13, 14] we came to a similar conclusion in the context of out-of-core factorization algorithms.

A second reason for a systematic approach is that it may well be that we require specialized matrix kernels for which efficient implementations do not exist (as part of libraries like the BLAS or LAPACK). In [12] we show how specialized matrix kernels speed up computations in control theory.

Finally, in [11] we show how the outlined approach can be used to show that seemingly different algorithms for matrix inversions are actually equivalent and therefore share stability properties.

6 Coding the Algorithm

In this section we briefly discuss how dense linear algebra algorithms can be translated to code. We first show a more traditional approach, as it appears in popular packages like LAPACK. Next, we present an alternative that allows coding at a level of abstraction that mirrors how we naturally present the algorithms. This second approach has been successfully used in our Parallel Linear Algebra Package (PLAPACK).

6.1 Classic implementation with the BLAS

Let us consider the blocked eager algorithm for the LU factorization presented in Fig. 2 (d). This algorithm requires an LU factorization of small matrix to factor $A_{11} \leftarrow L\backslash U_{11} = \text{LU fact.}(A_{11})$, triangular solves with multiple right-hand-sides to update $A_{12} \leftarrow U_{12} = L_{11}^{-1} A_{12}$ and $A_{21} \leftarrow L_{21} = A_{21} U_{11}^{-1}$, and a matrix-matrix multiply to update $A_{22} \leftarrow A_{22} - L_{21} U_{12}$. The triangular solves and matrix-matrix multiply are part of the Basic Linear Algebra Subprograms (BLAS) as calls to the routines `DTRSM` and `DGEMM`, respectively. The resulting code is given in Fig. 4. To understand this code, it helps to consider the partitioning of the matrix for a typical loop index j, as illustrated in Fig. 3: A_{11} is `B` by `B` and starts at element `A( J,J )`, A_{21} is `N-(J-1)-B` by `B` and starts at element `A( J+B, J )` , A_{12} is `B` by `N-(J-1)-B` and starts at element `A( J, J+B )`, and A_{22} is `N-(J-1)-B` by `N-(J-1)-B` and starts at element `A( J+B, J+B )`.

Given this picture, it is relatively easy to determine all of the parameters that must be passed to the appropriate BLAS routines.

6.2 The algorithm *is* the code

We would argue that it is relatively easy to generate the code in Fig. 4 given the algorithm in Fig. 2(d) and the picture in Fig. 3. However, the translation of the algorithm to the code is made tedious by the fact that one has to very carefully think about indices and matrix dimensions. While this is not much of a problem if one were to implement just one algorithm, it becomes a major headache when implementing all possible variants for a given operation or, in the case of a library, such as LAPACK, implementing even a single variant of a number of operations. One becomes even more acutely aware of these issues when parallel architectures enter the picture.

Consider the implementation of the eager LU factorization given in Fig. 5. It suffices to realize that A is now being passed to the routine as a data structure `A` that describes all attributes of this matrix, such as dimensions and method of storage. Inquiry routines like `PLA_Obj_global_length` are used to extract information such as the row dimension of the matrix. Finally, `A_TL`, `A_00`, etc. are simply references into the original array described by `A`. If one is familiar with the alphabet soup used to name the BLAS kernels, the code in Fig. 5 is self-explanatory given the algorithm in Fig. 2(d).

Naturally, one can argue that determining which of the two approaches to coding the algorithm might be deemed "superior" is simply a matter of taste. However, contemplate the following questions:

- What if a bug were introduced into the FORTRAN implementation? E.g. one of the `N-(J-1)-B` is accidentally changed to a `N-(J+1)-B`. This kind of bug is actually extremely hard to track down since the only clue is that the code produces the wrong answer. A similar bug cannot as easily be introduced into the code in Fig. 5. Furthermore, with this approach to coding it is easy to perform a run-time check to determine if the dimensions of the different references into A are conformal.

- What if we wanted to code all variants of the LU factorization without pivoting? Inherently, one would have to derive all algorithms, leading to descriptions like those given in Fig. 1 and Fig. 2(a)–(d). However, translating those to code like that given in Fig. 4 would require careful consideration of the picture in Fig. 3. Moreover, due to the intricate indexing involved in that approach to coding, considerable testing would be required before one could declare the code bug-free. By contrast, given the algorithms, generating code similar to that given in Fig. 5 would be straightforward. Moreover, since the code closely resembles the algorithm, one could be much more confident about its correctness even before the code had been tested.

- What if we wished to parallelize the given code? Notice that parallelizing a small subset of the functionality of LAPACK as part of the ScaLAPACK project has taken considerable effort. The code

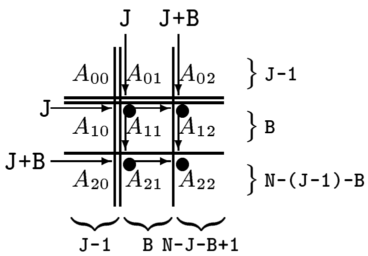

Figure 3: Partitioning of matrix A with all dimensions annotated when $A_{00} = A_{TL}$ is $(j-1) \times (j-1)$.

```
      SUBROUTINE LU_EAGER_LEVEL3( N, A, LDA, NB )

      INTEGER            N, LDA, NB, J, B
      DOUBLE PRECISION   A( LDA, * ), ONE, NEG_ONE
      PARAMETER          ( ONE = 1.0D00, NEG_ONE = -1.0D00 )

      DO J=1, N, NB
         B = MIN( N-J+1, NB )
C                                                     A11 <- L\U11 = LU fact( A11 )
         LU_EAGER_LEVEL2( B, A( J,J ), LDA )

         IF ( J+B .LE. N ) THEN

C                                                     A12 <- U12 = inv( L11 ) * A12

            DTRSM( "LEFT",  "LOWER TRIANGULAR", "NO TRANSPOSE", "UNIT DIAGONAL",
     $             ONE, B, N-(J-1)-B, A( J,J ), LDA, A( J, J+B ), LDA )

C                                                     A21 <- L21 = A21 * inv( U11 )

            DTRSM( "RIGHT", "UPPER TRIANGULAR", "TRANSPOSE",    "NONUNIT DIAGONAL",
     $             ONE, N-(J-1)-B, B, A( J,J ), LDA, A( J+B, J ), LDA )

C                                                     A22 <- A22 - A21 * A12

            DGEMM( "NO TRANSPOSE", "NO TRANSPOSE", N-(J-1)-B, N-(J-1)-B, B,
     $             NEG_ONE, A( J+B, J ), LDA, A( J, J+B ), LDA, ONE, A( J+B, J+B), LDA )
         ENDIF
      ENDDO

      RETURN
      END
```

Figure 4: FORTRAN implementation of blocked eager LU factorization algorithm using the BLAS.

```
void PLA_LU_eager( PLA_Obj A, int nb );
{
  < declarations >

  PLA_Create_constants_conf_to( A, &minus_one, NULL, &one );

  PLA_Obj_partition_4( A,          &ATL, /**/ &ATR,
                                /* ************** */
                                  &ABL, /**/ &ABR,
       /* with */ 0, /* by */ 0, /* submatrix */ PLA_SUBMATRIX_TL );

  while ( size = PLA_OBJ_GLOBAL_LENGTH( ABR ) ){
    b = min( size, nb );                             /* Determine block size b        */

    PLA_Obj_repartition_4_to_9( ATL, /**/ ATR,         &A00, /**/ &A01, &A02,
                              /* ************ */      /* ****************** */
                                       /**/             &A10, /**/ &A11, &A12,
                                ABL, /**/ ABR,         &A20, /**/ &A21, &A22,
         /* with */ b, /* by */ b, /* A11 split from submatrix */ PLA_SUBMATRIX_BR );

    PLA_LU_level2( A11 );                            /* A11 <- L\U11 = LU fact( A11 ) */

    PLA_Trsm( PLA_SIDE_LEFT, PLA_LOWER_TRIANGULAR,    /* A12 <- U12 = inv(L11) * A12   */
              PLA_NO_TRANSPOSE, PLA_UNIT_DIAG,
              one, A11, A12 );

    PLA_Trsm( PLA_SIDE_RIGHT, PLA_UPPER_TRIANGULAR,   /* A21 <- L21 = A21 * inv(U11)   */
              PLA_NO_TRANSPOSE, PLA_NONUNIT_DIAG,
              one, A11, A21 );

    PLA_Gemm( PLA_NO_TRANSPOSE, PLA_NO_TRANSPOSE,     /* A22 <- A22 - A21 * A12        */
              minus_one, A21, A12, one, A22 );

    PLA_Obj_continue_with_9_to_4( &ATL, /**/ &ATR,         A00, A01, /**/ A02,
                                            /**/               A10, A11, /**/ A12,
                                /* ************** */    /* ****************** */
                                  &ABL, /**/ &ABR,         A20, A21, /**/ A22,
         /* with A11 added to submatrix */ PLA_SUBMATRIX_TL );
  }

  < cleanup >
}
```

Figure 5: PLAPACK implementation of blocked eager LU factorization algorithm.

in Fig. 5 attains performance similar to that of the equivalent ScaLAPACK code *without requiring a single change to the code.*

6.3 Future directions

Many aspects of the approach we have described are extremely systematic: the generation of the loop-invariants, the derivation of the algorithm as well as the translation to code. Not discussed is the fact that the analysis of the run-time of the resulting algorithm on sequential or, for that matter, parallel, architectures is equally systematic. We are pursuing a project that exploits this systematic approach in order to automatically generate entire (parallel) linear algebra libraries as well as run-time estimates for the generated subroutines [7]. The goal is to create a mechanism that will automatically choose between different algorithms based on architecture and/or problem parameters.

A considerably less ambitious project, already nearing completion, allows the user to program in a language-independent manner (i.e. by writing ASCII version of the algorithms presented in this paper). Since it is our central thesis that the level of abstraction presented in this paper is the correct one, it seems an unnecessary onus to force the user to become familiar with the parameters and constraints of the underlying library. Obviously, the library must provide the necessary *functionality*, but the applications programmer should be concerned with nothing beyond the facilities provided and allowed to express those algorithms in terms of equations instead of function calls.

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 that is dramatically different than the one resulting from the more traditional approaches used by packages like 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. While we do not explicitly address the issue of performance, 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.

Additional Information

For additional information on PLAPACK and other software efforts that utilize directly or indirectly the proposed approach visit `http://www.cs.utexas.edu/users/plapack/`

References

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

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

[7] John Gunnels. *A Systematic Approach to the Design and Analysis of Parallel Dense Linear Algebra Algorithms*. PhD thesis, Department of Computer Sciences, The University of Texas, in preparation.

[8] 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.

[9] 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.

[10] 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.

[11] Enrique S. Quintana, Gregorio Quintana, Xiaobai Sun, and Robert van de Geijn. A note on parallel matrix inversion. *SISC*, to appear.

[12] Enrique S. Quintana-Orti and Robert van de Geijn. Fast parallel kernels for selected problems in control theory. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing 1999*, 1999.

[13] Wesley C. Reiley. Efficient parallel out-of-core implementation of the cholesky factorization". Technical Report CS-TR-99-33, Department of Computer Sciences, The University of Texas at Austin, Dec. 1999. Undergraduate Honors Thesis.

[14] 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.

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

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

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

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