# Formal Linear Algebra Methods Environment (FLAME) Overview*

John A. Gunnels
Robert A. van de Geijn

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

Greg M. Henry

Intel Corp.
Bldg C01-02
Beaverton, OR 97006-5733
greg.henry@intel.com

FLAME Working Note #1

November 8, 2000

**Abstract**

Since the advent of high-performance distributed-memory parallel computing, the need for intelligible code has become ever greater. The development and maintenance of libraries for these architectures is simply too complex to be amenable to conventional approaches to coding and attempting to employ traditional methodology 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 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 Formal Linear Algebra Methods Environment (FLAME) for implementing linear algebra algorithms on sequential architectures. This environment demonstrates that lessons learned in the distributed memory world can guide us toward better approaches to coding even in the sequential world.

*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.

# 1 Introduction

When considering the unmanageable complexity of computer systems, Dijkstra recently made the following observations [7]:

(i) When exhaustive testing is impossible –i.e., almost always– our trust can only be based on proof (be it mechanized or not).
(ii) A program for which it is not clear why we should trust it, is of dubious value.
(iii) A program should be structured in such a way that the argument for its correctness is feasible and not unnecessarily laborious.
(iv) Given the proof, deriving a program justified by it, is much easier than, given the program, constructing a proof justifying it.

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 [12]. 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 that call packages like LAPACK [3] and ScaLAPACK [6] 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 [26]. 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 [28]. 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 [14].

FLAME is the latest step in the evolution of these systems. It facilitates the use of a programming style that is equally applicable to everything from out-of-core, parallel systems to single-processor systems where cache-management is of paramount concern.

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. 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, it is exactly the formal 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 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. Presumably, it should be possible to prove the algorithms correct, thus ensuring that the code is correct. Previewing the next sections, algorithms for blocked variants of LU factorization of a matrix are presented in Fig. 2. In Fig. 5 we show how this can be translated into an skeleton for a code. By entering different updates of submatrices into this skeleton, given in Section 6.2, different variants of the LU factorization algorithm are realized.

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 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 (the proof is generally constructive in nature, but that is of little consequence) 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. Namely, components are expected to live up to certain "contractual obligations" [1, 4, 11]. 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.

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 $n \times n$ 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:

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

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

so that the following equalities must hold:

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

Finally, we arrive at the following algorithm

- Overwrite $\alpha_{11}$ and $a_{12}^T$ with $\upsilon_{11}$ 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 tail-recursive, it is usually implemented as a loop.

## 3.2 But what intermediate value is in the matrix?

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

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

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 <u>T</u>op, <u>B</u>ottom, <u>L</u>eft, and <u>R</u>ight, 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 when the algorithm has completed:

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

Finally, let $\hat{A}_k$ equal a matrix that 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 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:

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

Here we use the notation $\{L\backslash U\}$ to denote a lower and upper triangular matrix that 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 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 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 and discussed in G.W. Stewart's recent book on matrix factorization [27].

## 3.3 Lazy Algorithm

This algorithm is often referred to as a bordered algorithm in the literature. Stewart, [27] 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

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

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 we wish to change the contents of the current matrix from $\hat{A}_k$ to $\hat{A}_{k+1}$ or

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

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

Partition $A = \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)$
where $A_{TL}$ is $0 \times 0$
do until $A_{BR}$ is $0 \times 0$

Partition

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

$a_{01} \leftarrow u_{01} = L_{00}^{-1} a_{01}$
$a_{10}^T \leftarrow l_{10}^T = a_{10}^T U_{00}^{-1}$
$\alpha_{11} \leftarrow \upsilon_{11} = \alpha_{11} - l_{10}^T u_{01}$

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 $0 \times 0$
do until $A_{BR}$ is $0 \times 0$
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 $b \times b$
$A_{01} \leftarrow U_{01} = L_{00}^{-1} A_{01}$
$A_{10} \leftarrow L_{10} = A_{10} U_{00}^{-1}$
$A_{11} \leftarrow A_{11} - L_{10} U_{01}$
$A_{11} \leftarrow L\backslash U_{11} = \text{LU fact}(A_{11})$
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 a square matrix $A$ (without pivoting).

To derive how to compute these quantities, consider

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

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

From this equality we find that the following equalities must hold:

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

Thus, 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 $l_{10}^{(k)\,T} U_{00}^{(k)} = a_{10}^{(k)\,T}$. The result can overwrite $a_{10}^{(k)\,T}$. 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 $\alpha_{11}^{(k)}$. This motivates the algorithm in Fig. 1 (left) for overwriting given $n \times n$ matrix $A$ with its LU factorization.

To show that indeed there is a possibility of proving the algorithm correct, consider the following result:

**Theorem 1** *Consider the algorithm in Fig. 1 (left). This algorithm overwrites given $n \times n$ matrix $A$ with its LU factorization.*

**Proof:** Realize that, by design, at the top of the loop during the $k$th iteration of the loop $A$ contains the matrix $\hat{A}_k$ from the previous discussion. The derivation of the algorithm is such that it proves that given that the contents of $\hat{A}_k$ are as desired, the contents of $\hat{A}_{k+1}$ are as desired which proves the inductive step. Since $\hat{A}_0 = A$ we conclude that $\hat{A}_n = L\backslash U$. $\square$

It is possible to similarly prove the correctness of the remainder of the variants.

**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. $1 < b \ll n$). 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 $U_{11}^{(k)}$.

To derive how to compute these quantities, consider

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

This yields the equalities

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

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 [27] 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(c).

### 3.5 Column-Lazy Algorithm

This algorithm is referred to as a left-looking algorithm in [10] while Stewart [27] 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 (d). Again, this algorithm overwrites given $n \times n$ matrix $A$ with its LU factorization.

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

where $A_{TL}$ is $0 \times 0$

do until $A_{BR}$ is $0 \times 0$

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 2: Blocked versions of LU factorization without pivoting for all five commonly encountered variants. The different variants share the same skeleton that partitions and repartitions the matrix. Executing the operations in one of the five boxes yields a specific algorithm.

### 3.6 Row-Column-Lazy Algorithm

This algorithm is often 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 (e). Again, this algorithm overwrites given $n \times n$ matrix $A$ with its LU factorization.

### 3.7 Eager algorithm

This algorithm is commonly known of as classical Gaussian elimination.

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 $L_{21}^{(k)}$ we must 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 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 $\hat{A}_{22}^{(k)}$.

The resulting algorithm is given in Fig. 2(a). **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 the set of possible loop invariants that satisfy 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? 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 with dissimilar dimensions.

In [14] 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 that 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 [15] 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 [23, 24] 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 [22] we show how such specialized matrix kernels speed up computations in control theory.

Finally, in [21] we show how the approach outlined above 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 PLAPACK. FLAME represents a refinement of this methodology.

### 6.1 Classic implementation with the BLAS

Let us consider the blocked eager algorithm for the LU factorization presented in Fig. 2 (a). 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 resultant 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(a) and the picture in Fig. 3. However, the translation of the algorithm to the code is made tedious and error-prone 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 distributed memory architectures enter the picture.

In an effort to make the code look as much like the algorithms given in Fig. 2 as is possible within the confines of C and FORTRAN, we have developed the Formal Linear Algebra Methods Environment (FLAME). The skeleton that is shared by all five variants of LU factorization is given in Fig. 5. To understand the code, it suffices to realize that $A$ is 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 `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 alphabet soup used to name the BLAS kernels, it is clear that the following code segments, when entered in the appropropriate place in the code in 5, implement the different variants of the LU factorization:

**Lazy algorithm**

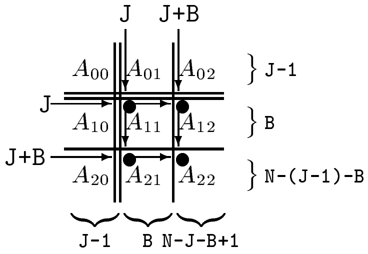


Figure 3: Partitioning of matrix $A$ with all dimensions annotated when $A_{00} = A_{TL}$ is $(j-1) \times (j-1)$.

```
      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.

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

where $A_{TL}$ is $0 \times 0$

do until $A_{BR}$ is $0 \times 0$

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 $b \times b$

---

$\vdots$

---

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

```
#include "FLA.h"

void FLA_LU_nopivot_skeleton( 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(min(FLA_Obj_length( ABR ), FLA_Obj_width( 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_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 5: An skeleton for the implementation of any of the blocked LU factorization algorithms in C using FLAME.

```
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 );
```

### Row-lazy algorithm

```
FLA_Trsm( FLA_RIGHT, FLA_UPPER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
          ONE, A00, A10 );

FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A10, A10, ONE, A11 );

FLA_LU_nopivot_level2( A11 );

FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A10, A02, ONE, A12 );

FLA_Trsm( FLA_LEFT,  FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
          ONE, A11, A12 );
```

### Column-lazy algorithm

```
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, A10, ONE, A11 );

FLA_LU_nopivot_level2( A11 );

FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A20, A10, ONE, A21 );
FLA_Trsm( FLA_RIGHT, FLA_UPPER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
          ONE, A11, A21 );
```

### Row-column-lazy algorithm

```
FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A10, A01, ONE, A11 );
FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A20, A01, ONE, A21 );
FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A10, A02, ONE, A12 );

FLA_LU_nopivot_level2( A11 );

FLA_Trsm( FLA_RIGHT, FLA_UPPER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
          ONE, A11, A21 );

FLA_Trsm( FLA_LEFT,  FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
          ONE, A11, A12 );
```

### Eager algorithm:

```
FLA_LU_nopivot_level2( A11 );

FLA_Trsm( FLA_RIGHT, FLA_UPPER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
          ONE, A11, A21 );

FLA_Trsm( FLA_LEFT,  FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
          ONE, A11, A12 );

FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A21, A12, ONE, A22 );
```

## 6.3 Proving the code correct

In Section 3.3 we showed how the correctness of the lazy algorithm can be proved and argue that the correctness of the other algorithms can be similarly derived. *If* the routines called by the described FLAME code correctly implement the operations in the algorithm, then it can be argued that the code itself is correct. Indeed, debugging is not necessary.

## 6.4 But is this really a better approach?

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? For example, suppose that one of the `N-(J-1)-B` were accidentally changed to a `N-(J+1)-B`. This kind of bug is extremely hard to track down since the only clue is that the code produces the wrong answer or causes a segmentation fault. A similar bug cannot as easily be introduced into the code implemented using FLAME. 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.

- When coding all variants of the LU factorization one inherently has to derive all algorithms, leading to descriptions like those given in Fig. 2. 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, it has been demontrated that generating all variants using FLAME is straightforward. As already mentioned, since the code closely resembles the algorithm, one can be much more confident about its correctness even before the code is ever 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 FLAME code can be transformed into PLAPACK code essentially by replacing `FLA_` by `PLA_`.

- What if we needed an out-of-core parallel version of the code? In principle, the FLAME code can be transformed into Parallel Out-of-Core Linear Algebra PACKage (POOCLAPACK) code by replacing `FLA_` by `POOCLA_`.

## 6.5 But what about 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.

## 6.6 But what about pivoting?

In Fig. 6 we show that pivoting can be easily added to, e.g., the eager LU factorization algorithm. Notice that in that implementation we also add recursion without much ado. We deem the code self-explanatory.

# 7 Performance

To illustrate that elegance does not necessarily come at the expense of performance, we measured the performance of the LU factorization with pivoting given in Fig. 6 followed by forward and backward substitution.

```
void FLA_LU( FLA_Obj A, FLA_Obj ipiv, int nb_alg )
{
  < declarations >

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

  FLA_Part_2x1( ipiv, &ipivT,
                   /* ****** */
                      &ipivB,
                /* with */ 0, /* length submatrix */ FLA_TOP );

  while ( b = min( min( FLA_Obj_length( ABR ), FLA_Obj_width( ABR ) ), nb_alg ) ){
    FLA_Repart_2x1_to_3x1( ipivT,                &ipiv0,
                        /* ***** */            /* ***** */
                                                 &ipiv1,
                           ipivB,                &ipiv2,
                /* with */ b, /* length ipiv1 split from */ FLA_BOTTOM );
    /* ************************************************************************* */
    FLA_Part_1x2( ABR,   &ABR_1, &ABR_2, /* with */ b, /* width submatrix */ FLA_LEFT );

    if ( nb_alg <= 4 ) FLA_LU_level2( ABR_1, ipiv1 );
    else               FLA_LU       ( ABR_1, ipiv1, nb_alg/2 );

    FLA_Apply_pivots( FLA_SIDE_LEFT, FLA_NO_TRANSPOSE, ipiv1, ABL );
    FLA_Apply_pivots( FLA_SIDE_LEFT, FLA_NO_TRANSPOSE, ipiv1, ABR_2 );
    /* ************************************************************************* */
    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_SIDE_LEFT, FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
              ONE, A11, A12 );

    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A21, A12, ONE, A22 );
    /* ************************************************************************* */
    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 );

    FLA_Cont_with_3x1_to_2x1( &ipivT,                  ipiv0,
                                                       ipiv1,
                            /* ***** */            /* ***** */
                              &ipivB,                  ipiv2,
                /* with ipiv1 added to */ FLA_TOP );
  }
}
```

Figure 6: FLAME recursive LU factorization with partial pivoting.

For comparison, we also measured the performance of the equivalent operations provided by ATLAS release R3.1 [29].

Some details: Performance was measured on a Pentium III based laptop with a 256K L2 cache running the Linux (Redhat 6.2) operating system. All computation was performed in 64-bit (double precision) arithmetic. For both implementations the same compiler options were used.

In Fig. 7 we report performance for four different implementations, indicated by the curves marked

`ATLAS`: This curve reports performance for the LU factorization provided by ATLAS R3.1, using the BLAS provided by ATLAS R3.1.

`ATL-FLAME`: This curve reports the performance of our LU factorization coded using FLAME built upon BLAS provided by ATLAS R3.1. The outer-most block size used for the LU factorization is 160 for these measurements. (Notice that multiples of 40 are optimal for the ATLAS matrix-matrix multiply on this architecture.)

`ITX-FLAME`: Same as the previous implementation, except that we optimized the matrix-matrix multiply (ITXGEMM). Details of this optimization are the subject of another paper [14]. This time the outer-most block size was 128. (Notice that multiples of 64 are optimal for the ITXGEMM matrix-matrix multiplication routine on this architecture.)

`ITX-FLAME-opt`: Same as the previous implementation, except that we optimized the level-2 BLAS based LU factorization of an intermediate panel as well as the pivot routine by not using the FLAME approach for those operations. For these routines we call `dscal`, `dger`, and `dswap` directly.

For all implementations, the forward and backward substitutions are provided by the ATLAS R3.1 `dtrsv` routine. The graph on the bottom shows the same data for smaller matrices is more detail.

Notice that for small matrices the unoptimized FLAME implementations perform somewhat worse, due to the overhead for manipulating the objects that encode the information about the matrices. When the level-2 BLAS based LU factorization is coded without this overhead, the performance is comparable even for small matrices. The better performance when the ITXGEMM matrix-matrix multiply is used is entirely due to the better performance of this matrix-matrix multiply.

It is important to realize that the performance difference between the implementation offered as part of ATLAS R3.1 and our own implementation is not the point of this paper: With some effort either implementation can be improved to match the performance of the other. Our primary point is that markedly less effort is required to implement these algorithms using FLAME while attaining performance comparable to that of what are widely considered to be high-performance implementations.

# 8 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 [13]. 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 an 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

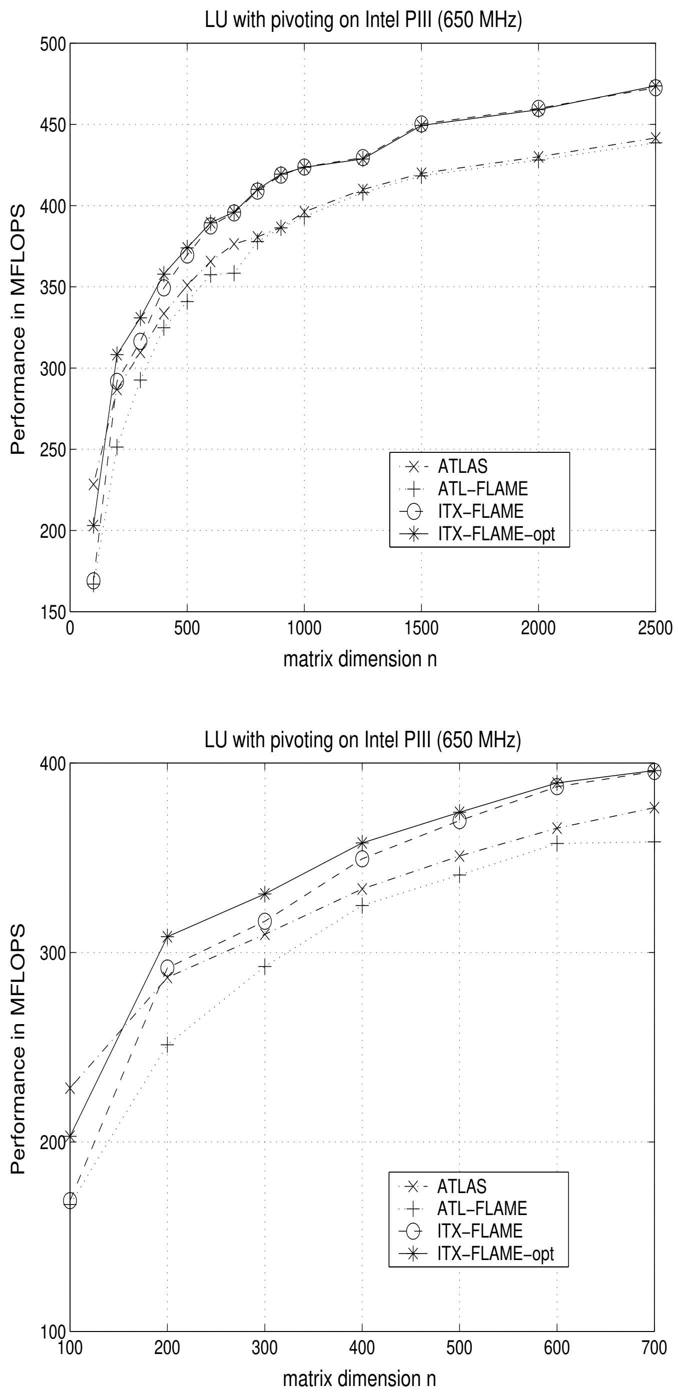


Figure 7: Performance of LU factorization with pivoting followed by forward and backward substitution.

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 at the correct level of abstraction, e.g. in terms of equations instead of function calls.

# 9 Related Work

Advances in software engineering for scientific applications has often been led by libraries for dense linear algebra operations.

## 9.1 BLAS, EISPACK, LINPACK, and LAPACK

The first such package to achieve widespread use and to embody some new techniques in software engineering was EISPACK [25]. EISPACK was also likely the first such package to pay careful attention to the numerical stability of the underlying algorithms.

The mid-seventies witnessed the introduction of the Basic Linear Algebra Subprograms (BLAS) [20]. At that time the BLAS were a set of vector operations which 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 [8]. By the mid-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 [9]. LAPACK, first released in the early nineties, is a high-performance package for linear algebra operations written in terms of the level-3 BLAS [3]. LAPACK offers a functionality that is a superset of LINPACK and EISPACK while achieving high performance on essentially all sequential and shared-memory architectures in a portable fashion.

## 9.2 GEMM-based Level 3 BLAS

Another source of inspiration for FLAME is the idea proposed by Kagstrom, Ling and Van Loan to code level 3 BLAS in terms of optimized matrix-matrix multiplication [19, 17]. 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. By constructing wrappers around _GEMM and a few lower-level BLAS routines, total coverage could be supplied for all level-3 BLAS with little coding effort. Further, the performance of this portable system was comparable to vendor-supplied BLAS in many cases.

We note that as an experiment to measure how FLAME reduces code development time, one of the authors implemented all level-3 BLAS in terms of matrix-matrix multiplication for single and double precision arithmetic in a matter of about ten hours, including testing. Performance of the implementations was again comparable to that of ATLAS R3.1. The resulting code can be found on the FLAME web page.

## 9.3 PLAPACK and ScaLAPACK

With the advent of distributed memory parallel architectures, a parallel version of LAPACK, called ScaLAPACK [6], was developed. A major design goal of the ScaLAPACK project was to preserve and re-use 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 creating and maintaining the data structures required for storing sparse matrices. The mapping from indices to matrix element(s) was no longer a simple one. Combining this complication with the monolithic structure of the software quickly led to code that was highly complicated to construct and extremely difficult to maintain.

PLAPACK achieves functionality similar to ScaLAPACK while targeting the same distributed-memory architectures using a FLAME-like approach to hide indexing and distribution details. Indeed, the inspiration for FLAME came from PLAPACK.

### 9.4 PHiPAC and ATLAS

PHiPAC was one of the first projects to seriously consider the amount of time and effort typically required to implement an optimized version of the level-3 BLAS [5]. This project considered the performance of both memory hierarchies and compilers on modern systems. ATLAS utilized PHiPAC's approach and extended it to generate almost the entire BLAS library as well as some matrix decomposition routines [30]. Both efforts resulted in code generators that use empirical information about the performance of automatically-produced kernels in order to construct routines that operate at or near the speed of vendor-supplied routines.

### 9.5 Recursive Algorithms and Data structures

A number of recent efforts explore hierarchical data structures for storing matrices [18, 2, 16]. The idea is that by storing matrices by blocks rather than row- or column-major ordering, data reuse in caches can be enhanced. By combining this with recursive algorithms that exploit this data structure, impressive performance improvements have been demonstrated.

Notice that more complex data structures for sequential algorithms introduce a complexity similar to that encountered when data is 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.

## 10 Discussion

Let us again examine the observations of Dijkstra:

> (i) When exhaustive testing is impossible –i.e., almost always– our trust can only be based on proof (be it mechanized or not).

Certainly, this succinct observation borders on truism. However, many related libraries have relied on neither exhaustive testing nor on proof. In all fairness, extensive testing is often a goal in such endevors, but has all too often been a goal sacrificed for the sake of expediency. Our system, since it lends itself to calculational proofs that border on the mechanical, obviates the need for the kind of truly exhaustive testing that more *ad hoc* systems truly require. Further, the use of simple proofs that go hand-in-hand with algorithmic construction allow us to largely ignore the uncertainty of what it means for testing to be "exhaustive."

> (ii) A program for which it is not clear why we should trust it, is of dubious value.

In this paper, and through years of experience writing parallel linear algebra libraries, we have learned this lesson the hard way. A large percentage of code and an even larger percentage of effort was devoted to the development of test code for packages like LAPACK and ScaLAPACK. Unfortunately, even where those efforts have been meticulous in their application of test suites to code bases, problems remain. It is rarely the case that the "right amount" of testing is a value that can be established in anything like a scientific manner and there is also often the rather unsettling fact that the test codes themselves can be erroneous. While there are any number of software engineering techniques that can make a real difference when that approach to coding is pursued, we believe that the more formal and systematic approach that underlies FLAME and PLAPACK has greatly reduced the need for such testing, while simultaneously increasing our confidence in the implementation.

> (iii) A program should be structured in such a way that the argument for its correctness is feasible and not unnecessarily laborious.

Notice that one of the central themes of our approach is the intertwining of algorithmic development with computational implementation. The evidence for the correctness of the algorithm lies in the constructive proof underlying its creation. To put it plainly: Since the code reflects the algorithm, the argument that the algorithm is correct carries over to an argument that the code is correct. While one could, justifiably, take issue with what "reflect" means in this case, we do not feel that any reasonable argument could be made assailing the assertion that the code "looks like" the algorithm as one would present it in a classroom setting. While this may, initially, appear to be a *strictly* superficial argument in favor of our approach, closer inspection should dissuade one from this belief. This lexical proximity implies that a straightforward set of property-preserving rewrite rules could be used to translate from proof text to code text. More importantly, FLAME's unified approach wherein each layer is constructed in a (like) manner reflective of the same simple philosophy and the fact that the base components (those constructed in something akin to a "traditional" manner) are kept small in both number and size are both facts that make FLAME amenable to traditional, mechanical, proof technology.

> (iv) Given the proof, deriving a program justified by it, is much easier than, given the program, constructing a proof justifying it.

One could view the manner in which we have addressed this point in at least two ways. The first is rather egotistical and not the view of the authors. Namely, that we have "gone Dijkstra one better" and created a system where the construction of a proof justifying the program is no more difficult than generating a program justified by the proof. The second is, in our opinion, more accurate and does not gloss over certain details inherent in the first view. The most important "detail" is to note that this observation is just that, an observation; it contains little in the way of a hidden goal, unlike the first three points. In truth, virtually any high-level programming language lends itself to the creation of code where it is certainly the case that proving the code correct is considerably more difficult than constructing the program given the proof. However, in reducing the lexical distance between the language of the proof and the language of the code, tends to make the derivation of a program a trivial matter when one has the proof in hand. We also feel that we have greatly simplified the process for proving the correctness of a code derived in the style suggested in this paper. Unfortunately, we cannot claim that we have made this an entirely symmetric case. Why not? There are a number of reasons, but two will suffice. The first is that our system is built to interface with third-party libraries (e.g. the BLAS) whose correctness is assumed, but not *known*. The second is related to the first and only seems overcautious to those not versed in formal systems. We cannot assume that the system is correct when we have no proof that the computer architecture itself is correct. Notice that we construct the entire FLAME architecture from a small number of components and have argued that a proof-of-correctness for the system could be done formally without an undue amount of effort because of the manner in which the system is modularized. However, even our lowest level components rely on certain *unproven* assumptions regarding the underlying architecture.

So, Dijkstra wins again.

# 11 Conclusion

A colleague of ours, Dr. Tim Mattson from Intel, recently made the following observation: "Literature professors read literature. Computer Science professors should, at least occasionally, read code." When one does this, certain patterns emerge and one tends to become more readily able to distinguish good code from bad.

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 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.

## Additional Information

For additional information on FLAME, ITXGEMM, PLAPACK and other software efforts that utilize directly or indirectly the discussed approach please visit

`http://www.cs.utexas.edu/users/flame/`
`http://www.cs.utexas.edu/users/flame/ITXGEMM/`
`http://www.cs.utexas.edu/users/plapack/`

## References


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

[2] Bjarne S. Andersen, Fred G. Gustavson, and Jerzy Wasniewski. A recursive formalation of Cholesky factorization of a matrix in packed storage. LAPACK Working Note 146 CS-00-441, University of Tennessee, Knoxville, May 2000.

[3] 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.

[4] Don Batory and Bart J. Geraci. Composition validation and subjectivity in genvoca generators. *IEEE Transactions on Software Engineering (special issue on Software Reuse*, pages 67–82, 1997.

[5] J. Bilmes, K. Asanovic, C.W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *In Proceedings of the International Conference on Supercomputing*. ACM SIGARC, July 1997.

[6] 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.

[7] Edsger Wybe Dijkstra. Under the spell of Leibniz's dream. Technical Report EWD1298, The University of Texas at Austin, April 2000. `http://www.cs.utexas.edu/users/EWD/`.

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

[9] 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.

[10] 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.

[11] David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] F. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling. Superscalar gemm-based level 3 blas – the on-going evolution of a portable and high-performance library. In B. Kågström et al., editor, *Applied Parallel Computing, Large Scale Scientific and Industrial Problems*, volume 1541 of *Lecture Notes in Computer Science*, pages 207–215. Springer-Verlag, 1998.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

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

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

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

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

[29] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98*, 1998.

[30] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC98*, Nov. 1998.