FLAME: Formal Linear Algebra Methods Environment

JOHN A. GUNNELS
The University of Texas at Austin
IBM T.J. Watson Research Center
and
FRED G. GUSTAVSON
IBM T.J. Watson Research Center
and
GREG M. HENRY
Intel Corporation
and
ROBERT A. VAN DE GEIJN
The University of Texas at Austin

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 implementation. Attempts to employ traditional methodology have led, in our opinion, to the production of an abundance of anfractuous code that is difficult to maintain and almost impossible to upgrade.

Having struggled with these issues for more than a decade, we have concluded that a solution is to apply a technique from theoretical computer science, formal derivation, to the development of high-performance linear algebra libraries. We think the resulting approach results in aesthetically-pleasing, coherent code that greatly facilitates intelligent modularity and high performance while enhancing confidence in its correctness. Since the technique is language-independent, it lends itself equally well to a wide spectrum of programming languages (and paradigms) ranging from C and Fortran to C++ and Java. In this paper, we illustrate our observations by looking at the Formal Linear Algebra Methods Environment (FLAME), a framework that facilitates the derivation and implementation of linear algebra algorithms on sequential architectures. This environment demonstrates that lessons learned in the distributed-memory world can guide us toward better approaches even in the sequential world.

We present performance experiments on the Intel (R) Pentium (R) III processor that demonstrate that high performance can be attained by coding at a high level of abstraction.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: —*Algorithm Design and*

Authors' addresses: John A. Gunnels, IBM T.J. Watson Research Center P.O. Box 218 Yorktown Heights, N.Y. 10598 gunnels@us.ibm.com. Fred G. Gustavson, IBM T.J. Watson Research Center P.O. Box 218 Yorktown Heights, N.Y. 10598 gustav@watson.ibm.com. Robert A. van de Geijn, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, rvdg@cs.utexas.edu. Greg M. Henry, Intel Corp., Bldg EY2-05, 5350 NE Elam Young Pkwy, Hillsboro, OR 97124-6461, greg.henry@intel.com.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0098-3500/20YY/1200-0001 $5.00

Analysis; *Efficiency*; *User interfaces*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain specific architectures*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*

General Terms: Algorithms;Design;Performance;Theory

Additional Key Words and Phrases: Formal derivation, libraries, linear algebra, performance

1. INTRODUCTION

When considering the unmanageable complexity of computer systems, Dijkstra recently made the following observations [Dijkstra 2000]:

(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.

These comments relate to issues that, in context of linear algebra libraries, are orthogonal to the concerns related to numerical stability.

In this paper we make a number of new contributions to the development linear algebra libraries with regards to the above observations:

—By borrowing from Dijkstra's own contributions to computing science, we show how to systematically derive families of algorithms for a given matrix operation.

—The derivation leads to a structured statement of the algorithms that mirrors how the algorithms are often explained in a classroom setting.

—The derivation of the algorithms provides a proof of the correctness of the algorithms.

—By implementing the algorithms so that the code mirrors the algorithm that is the end-product of this derivational process, opportunities for the introduction of error are reduced. As a result, the proof of the correctness of the algorithm allows us to assert the correctness of the code.

While the resulting infrastructure, the Formal Linear Algebra Methods Environment (FLAME), has allowed us to quickly and reliably implement components of a high-performance linear algebra library, it can equally well benefit library users who need to customize a given routine or to extend the functionality of their own library.

1.1 Related work

Advances in software engineering for scientific applications have often been led by libraries for dense linear algebra operations. The first such package to achieve widespread use and to embody new techniques in software engineering was EISPACK [Smith et al. 1976]. EISPACK was also likely the first such package to

pay careful attention to the numerical stability of the underlying algorithms. The mid-1970s witnessed the introduction of the Basic Linear Algebra Subprograms (BLAS) [Lawson et al. 1979]. At that time the BLAS were a set of vector operations that allowed libraries to attain high performance on computers possessing a flat memory while remaining highly portable between platforms and simultaneously enhancing modularity and code readability. The first successful library to exploit these BLAS was LINPACK [Dongarra et al. 1979]. By the late 1980s, it was recognized that in order to overcome the gap between processor and memory performance on modern microprocessors it was necessary to reformulate matrix algorithms in terms of matrix-matrix multiplication-like operations, the level-3 BLAS [Dongarra et al. 1990]. LAPACK [Anderson et al. 1992], first released in the early 1990s, is a high-performance package for linear algebra operations written in terms of the level-3 BLAS. 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.

A major simplification in the implementation of the level-3 BLAS themselves came from the observation that they can be cast in terms of optimized matrix-matrix multiplication [Agarwal et al. 1994; Gustavson et al. 1998b; Kågström et al. 1998]. Further, the performance of the resulting, more portable, system was comparable to the vendor-supplied BLAS in many cases.

With the advent of distributed-memory parallel architectures, a parallel version of LAPACK, ScaLAPACK [Choi et al. 1992], 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 were designed to resemble similar layers in the LAPACK software architecture. It was this decision that complicated the implementation of ScaLAPACK. The introduction of data distribution (across memories) creates a problem analogous to 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 led to code that was laborious to construct and is difficult to maintain. The Parallel Linear Algebra Package (PLAPACK) achieves a functionality similar to that of ScaLAPACK, targeting the same distributed-memory architectures while using a FLAME-like approach to hide details related to indexing into and distribution of matrices [van de Geijn 1997]. Indeed, the primary inspiration for FLAME came from earlier work on PLAPACK.

A number of recent library efforts have explored the notion of utilizing hierarchical data structures for storing matrices [Andersen et al. 2000; Gustavson et al. 1998a; Gustavson 1997; 2001]. The central idea is that, by storing matrices by blocks rather than by row- or column-major ordering, data preparation (copying) for good cache re-use is virtually eliminated. Combining this with recursive and standard algorithms that exploit these data structures, impressive performance improvements have been demonstrated. Notice that more complex data structures for sequential algorithms introduce a complexity similar to that encountered when data are distributed to the memories of a distributed-memory architecture. Since PLAPACK effectively addressed that problem for those architectures, we have strong

evidence that FLAME can be extended to accommodate more complex data structures in the context of hierarchical memories.

1.2 A case for systematic derivation and simplicity of code

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 linear algebra packages 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-1990s, 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 [Snir et al. 1996]. Using objects (e.g. communicators in MPI) to encapsulate data structures and hide complexity, a much cleaner approach can be achieved.

Our own work on PLAPACK borrowed from this approach in order to hide details of data distribution and data mapping in the realm of parallel linear algebra libraries. 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 derived in a classroom setting. Since our initial work on PLAPACK, we have experimented with similar interfaces in such contexts as (parallel) out-of-core linear algebra packages [Gunter et al. 2001; Reiley and van de Geijn 1999] and a low-level implementation of the sequential Basic Linear Algebra Subprograms (BLAS) [Gunnels et al. 2001; Gunnels and van de Geijn 2001b].

One strong motivation for systematically deriving algorithms and reducing the complexity of translating these algorithms to code comes from the fact that for a given operation a different algorithm may provide higher performance depending on the architecture and/or the problem dimensions. In [Gunnels et al. 2001] we showed that the efficient, transportable implementation of matrix-matrix multiplication on a sequential architecture with a hierarchical memory requires a hierarchy of matrix algorithms whose organization, in a very real sense, mirrors that of the memory system under consideration. Perhaps surprisingly, this is necessary even when the problem size is fixed. In the same paper we described a methodology for composing these routines. In this way, minimal coding effort is required to attain superior performance across a wide spectrum of algorithms, architectures, and problem sizes. Analogously, in [Gunnels et al. 1998] it was demonstrated that an efficient implementation of parallel matrix multiplication requires both multiple algorithms and a method for selecting an appropriate algorithm for the presented case if one is to handle operands of various sizes and shapes. In [Gunter et al. 2001; Reiley 1999] we arrived at a similar conclusion in the context of out-of-core factorization algorithms and their implementation using the Parallel Out-of-Core Linear Algebra PACKage (POOCLAPACK). To summarize our experiences: as high-performance architectures incorporate cache, local, shared, and distributed memories all within one system, multiple algorithms for a single operation become necessary for optimal performance. Traditional approaches make the implementation of libraries that span all possibilities almost impossible.

FLAME is the latest step in the evolution of these systems. We consider FLAME to be an environment in the sense that it encourages the developer to systematically construct a family of algorithms for a given matrix operation. Ideally, the steps that lead to the algorithms are carefully documented, providing the proof that the algorithms are correct. Only after its correctness can be asserted should the algorithm be translated to code. Since the code mirrors the end-product of the algorithmic derivation process, its correctness can be asserted as well, and minimal debugging and testing will be necessary. Once the code delivers the correct results, functionality can be extended and/or performance optimizations can be incorporated. We illustrate FLAME in the simplest setting, for sequential algorithms. Minor modifications to PLAPACK and POOCLAPACK allow the porting to distributed-memory architectures and/or out-of-core computations with essentially no change to the code.

1.3 Overview

In Section 2 we review some basic insights from formal derivation. In Section 3 we use the LU factorization without pivoting to illustrate the steps we use to develop a family of algorithms for a given operation. In Section 4 we summarize our systematic steps for deriving linear algebra algorithms. In Section 5 we show how library extensions added to the C programming language, together with careful formatting, allows one to write code that reflects the algorithm. The fact that the techniques can be applied to a more difficult operation like LU factorization with partial pivoting is demonstrated in Section 6. In Section 7 we show that it is not the case that high performance is compromised by raising the level of abstraction at which one codes. Future directions and conclusions are discussed in Sections 8 and 9.

2. THE CORRECTNESS OF LOOPS

In a standard text by Gries and Schneider used to teach discrete mathematics to undergraduates in computer science we find the following material ([Gries and Schneider 1992], pages 236–237):

> We prefer to write a while loop using the syntax
>
> **do** $B \rightarrow S$ **od**
>
> where Boolean expression B is called the *guard* and statement S is called the *repetend*.
> [The l]oop is executed as follows: If B is *false*, then execution of the loop terminates; otherwise S is executed and the process is repeated.
> Each execution of repetend S is called an *iteration*. Thus, if B is initially *false*, then 0 iterations occur.

The text goes on to state:

> We now state and prove the fundamental invariance theorem for loops. This theorem refers to an assertion P that holds before and after each iteration (provided it holds before the first). Such a predicate is called a *loop-invariant*.
>
> (12.43) **Fundamental invariance theorem.** Suppose

—$\{P \wedge B\}S\{P\}$ holds – i.e. execution of S begun in a state in which P and B are *true* terminates with P *true* – and
—$\{P\}$ **do** $B \rightarrow S$ **od** *true* – i.e. execution of the loop begun in a state in which P is *true* terminates.
Then $\{P\}$ **do** $B \rightarrow S$ **od** $\{P \wedge \neg B\}$ holds. [In other words, if the loop is entered in a state where P is *true*, it will complete in a state where P is *true* and guard B is *false*.]

The text proceeds to prove this theorem using the axiom of mathematical induction.

Let us translate the above programming construct into our setting, which we use to accommodate linear algebra algorithms. Consider the loop

while B **do**
S
enddo

where B is some condition and S is the body of the loop, the above theorem says that

—The loop is entered in a state where some condition P holds, and
—for each iteration, P holds at the top of the loop, and
—the body of the loop S has the property that if it is executed starting in a state where P holds it completes in a state where P holds.

Then if the loop completes, it will do so in a state where conditions P and $\neg B$ both hold. Naturally, P and B are chosen such that $P \wedge \neg B$ implies that the desired linear algebra operation has been computed.

A method that formally derives a loop (i.e., iterative implementation) approaches the problem of determining the body of the loop as follows: First, one must determine conditions B and P. Next, the body S should be developed so that it maintains condition P while making progress towards completing the iterative process (eventually B should become *false*). The operations that comprise S follow naturally from simple manipulation of equalities and equivalences using matrix algebra. Thanks to the fundamental invariance theorem, this approach implies correctness of the loop.

What we will argue in this paper is that for a large class of dense linear algebra algorithms there is a systematic way of determining different conditions P that allow us develop loops to compute a given linear algebra operation. The different conditions yield different algorithmic variants for computing the operation. We demonstrate this through the example of LU factorization without pivoting. Once we have demonstrated the techniques in this simpler setting, we will also argue, although somewhat more informally, the correctness of a hybrid iterative/recursive LU factorization with partial pivoting in Section 6.

3. A CASE STUDY: LU FACTORIZATION

We illustrate our approach by considering LU factorization without pivoting. Given a non-singular, $n \times n$ matrix ,A, we wish to compute an $n \times n$ lower triangular matrix L with unit main diagonal and an $n \times n$ upper triangular matrix U so that $A = LU$.

The original matrix A is overwritten by L and U in the process. We will denote this operation by

$$A \leftarrow \hat{A} = \mathrm{LU}(A)$$

to indicate that A is overwritten by the LU factors of A. Because FLAME produces many variants of LU factorization, it is worthwhile to emphasize the fact that, if exact arithmetic is performed, all variants will result in identical results. To see this assume that $L_1U_1 = L_2U_2$ are two *different* factorizations. Multiplying both sides by L_2^{-1} on the left and U_1^{-1} on the right yields $L = L_2^{-1}L_1 = U_2U_1^{-1} = U$, where L is unit lower-triangular and U upper-triangular. Now, $L = U$ implies $L = U = I$. It follows that $L_1 = L_2$ and $U_1 = U_2$, so our assumption has been contradicted and the proof of uniqueness is complete.

3.1 A classical derivation

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

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

Now $A = LU$ translates to

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

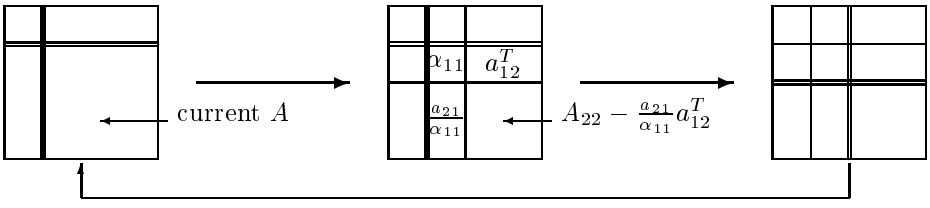
so the following equalities hold:

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

Thus, 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 $A_{22} \leftarrow A_{22} - l_{21}u_{12}^T$.
—Factor $A_{22} \rightarrow L_{22}U_{22}$ (recursively or iteratively).

The algorithm is usually implemented as a loop, as illustrated in Fig. 1. When presented in a classroom setting, this algorithm is typically accompanied by the following progression of pictures:



Here the double lines indicate how far the computation has progressed through the matrix. At the current stage the active part of the matrix resides in the lower-right quadrant of the left picture. Next, the different parts to be updated are identified and the updates given (middle picture). Finally, the boundary that indicates

partition $A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)$ **where** A_{TL} **is** 0×0

do until A_{BR} **is** 0×0

repartition $\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & a_{01} & A_{02} \\ \hline\hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$ **where** α_{11} **is a scalar**

$\alpha_{11} \leftarrow v_{11} = \alpha_{11}$ (no-op)
$a_{12}^T \leftarrow u_{12}^T = a_{12}^T$ (no-op)
$a_{21} \leftarrow l_{21} = a_{21}/v_{11}$
$A_{22} \leftarrow A_{22} - l_{21}u_{12}^T$

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

enddo

Fig. 1. Unblocked lazy algorithm for LU factorization.

how far the computation has progressed is moved forward (right picture). It is this sequence of three pictures that we will try to capture in the derivation, the specification of the algorithm, and the implementation of the algorithm.

3.2 But what is the loop-invariant?

Notice that in the above algorithm the original matrix is overwritten by intermediate results until finally it contains L and U. Let $\hat{A}$ indicate the matrix in which the LU factorization is computed, keeping in mind that $\hat{A}$ overwrites A as part of the algorithm. Notice that after k iterations of the algorithm in Fig. 1, $\hat{A}$ contains a partial result. We will denote this partial result by $\hat{A}_k$.

In order to prove correctness, one question we must ask is what intermediate value, $\hat{A}_k$, is in $\hat{A}$ at any particular stage of the algorithm. More precisely, we will ask the question of what the contents are at the beginning of the loop that implements the computation of the factorization (e.g., the loop in Fig. 1). To answer this question, partition the matrices as follows:

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

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

Notice that

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

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

so that the following equalities must hold:

$$A_{TL}^{(k)} = L_{TL}^{(k)} U_{TL}^{(k)} \quad (1)$$
$$A_{TR}^{(k)} = L_{TL}^{(k)} U_{TR}^{(k)} \quad (2)$$
$$A_{BL}^{(k)} = L_{BL}^{(k)} U_{TL}^{(k)} \quad (3)$$
$$A_{BR}^{(k)} = L_{BL}^{(k)} U_{TR}^{(k)} + L_{BR}^{(k)} U_{BR}^{(k)} \quad (4)$$

We now show that different conditions on the contents of $\hat{A}$ dictate different algorithmic variants for computing the LU factorization, and that these different conditions can be systematically generated from Eqns. 1–4.

Notice that in Eqns. 1–4 the following partial results towards the computation of the factorization can be identified:

$$L\backslash U_{TL}^{(k)}, \quad L_{BL}^{(k)}, U_{TR}^{(k)}, \quad L_{BL}^{(k)} U_{TR}^{(k)}, \quad \text{and} \quad L\backslash U_{BR}^{(k)}$$

Here we use the notation $L\backslash U$ to denote lower and upper triangular matrices 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 that need not be stored. We restrict our study to algorithms that employ Gaussian elimination and do not involve redundant computations. Further, we require that one or more of the partial results contributing to the final computation have been computed. A few observations:

—If $L_{TL}^{(k)}$ has been computed, the elements of $U_{TL}^{(k)}$ has been computed as well.

—Since $L_{BL}^{(k)} = A_{BL}^{(k)} U_{TL}^{(k)\,-1}$, data dependency considerations imply that $U_{TL}^{(k)}$ must be computed before $L_{BL}^{(k)}$.

—Similarly, since $U_{TR}^{(k)} = L_{TL}^{(k)\,-1} A_{TR}^{(k)}$, data dependency analysis implies that $L_{TL}^{(k)}$ needs to be computed before $U_{TR}^{(k)}$.

—Since the computation overwrites A, if $L_{BL}^{(k)} U_{TR}^{(k)}$ has been computed, $\hat{A}_{BR}^{(k)}$ must contain $A_{BR}^{(k)} - L_{BL}^{(k)} U_{TR}^{(k)}$.

—If $L_{BR}^{(k)}$ has been computed, we assume that $U_{BR}^{(k)}$ has been computed as well (see first bullet).

—If $L\backslash U_{BR}^{(k)}$ has been computed, $A_{BR}^{(k)} - L_{BL}^{(k)} U_{TR}^{(k)}$ must have been computed first.

Taking into account the above observations, we give possible contents of $\hat{A}_k$ in Table I. The first and last conditions indicate that no computation has been performed or the final result has been computed, neither of which is a reasonable condition to maintain as part of the loop. This leaves five loop-invariants which, we will see, lead to five different variants for LU factorization.

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 many commonly discussed algorithms. In fact, in [Dongarra et al. 1984] six variants, called the ijk orders, of $A = LU$ are listed. The jki form is commonly known as a

Table I. Possible loop-invariants for LU factorization without pivoting.

Condition	$\hat{A}_k$ contains
No computation has occurred.	$\left(\begin{array}{c\|\|c} A_{TL}^{(k)} & A_{TR}^{(k)} \\ \hline\hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right)$
Only (1) is satisfied.	$\left(\begin{array}{c\|\|c} L\backslash U_{TL}^{(k)} & A_{TR}^{(k)} \\ \hline\hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right)$
Only (1) and (2) have been satisfied.	$\left(\begin{array}{c\|\|c} L\backslash U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline\hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right)$
Only (1) and (3) have been satisfied.	$\left(\begin{array}{c\|\|c} L\backslash U_{TL}^{(k)} & A_{TR}^{(k)} \\ \hline\hline L_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right)$
Only (1), (2), and (3) have been satisfied.	$\left(\begin{array}{c\|\|c} L\backslash U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline\hline L_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right)$
(1), (2), and (3) have been satisfied and as much of (4) has been computed *without computing any part of* $L_{BR}^{(k)}$ *or* $U_{BR}^{(k)}$.	$\left(\begin{array}{c\|\|c} L\backslash U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline\hline L_{BL}^{(k)} & A_{BR}^{(k)} - L_{BL}^{(k)} U_{TR}^{(k)} \end{array}\right)$
(1), (2), (3), and (4) have all been satisfied.	$\left(\begin{array}{c\|\|c} L\backslash U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline\hline L_{BL}^{(k)} & L\backslash U_{BR}^{(k)} \end{array}\right)$

left-looking algorithm while the ikj form is left-looking on A^T . Together, they correspond to the row- and column-lazy variants discussed in this paper. The kij and kji forms both correspond to what has been traditionally called the right-looking algorithm; here, both would be deemed forms of the eager algorithm, one a column- and one a row-oriented version. The ijk and jik forms are more commonly known as the Doolittle (Crout) algorithm and correspond to row- and column-oriented versions of the row-column-lazy variant considered in this document. The lazy algorithm discussed in this paper has no corresponding variant in the ijk family of algorithms. Further, the conditions delineated above yield all algorithms depicted on the cover of, and discussed in, G.W. Stewart's recent book on matrix factorization [Stewart 1998]. This comes as no surprise as we, like Stewart, have adopted some common implicit assumptions about both matrix partitioning and the nature of algorithmic advancement. Our *a priori* assumptions were only slightly less constricting than those imposed by the authors who employed the ijk scheme mentioned above. In this paper we have restricted ourselves to a consideration of only those algorithms whose progress is "simple." That is, each iteration of the algorithm is geographically monotonic and formulaically identical. The combination of these two properties leads to algorithms whose (inductive) proofs of correctness are straightforward and whose implementations, given our framework, are virtually foolproof.

We will label any algorithm "Lazy" if it does the least amount of computation possible in the inductive step and "Eager" if it performs as much work as possible at that point. We explain our classification further in [Gunnels and van de Geijn 2001a]. It needs to be evaluated against a large class of algorithms before we make

any definitive claims regarding is usefulness.

3.3 Lazy algorithm

This algorithm is often referred to as a bordered algorithm in the literature. Stewart, [Stewart 1998] rather colorfully, refers to it as Sherman's march.

Unblocked Algorithm. Let us assume that only (1) has been satisfied. To determine the body of the loop (statement S), the question becomes how to update the contents of $\hat{A}$:

$$\left(\begin{array}{c|c} \hat{A}_{BR}^{(k)} & \hat{A}_{TR}^{(k)} \\ \hline \hat{A}_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right) = \left(\begin{array}{c|c} L\backslash U_{BR}^{(k)} & A_{TR}^{(k)} \\ \hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array}\right)$$
$$\longrightarrow \left(\begin{array}{c|c} \hat{A}_{BR}^{(k+1)} & \hat{A}_{TR}^{(k+1)} \\ \hline \hat{A}_{BL}^{(k+1)} & \hat{A}_{BR}^{(k+1)} \end{array}\right) = \left(\begin{array}{c|c} L\backslash U_{BR}^{(k+1)} & A_{TR}^{(k+1)} \\ \hline A_{BL}^{(k+1)} & A_{BR}^{(k+1)} \end{array}\right)$$

To answer this, repartition

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

where $A_{00}^{(k)}$ is $k \times k$ (and thus equal to $A_{TL}^{(k)}$), and $\alpha_{11}^{(k)}$ is a scalar. Repartition $\hat{A}_k$, L, and U similarly. This repartitioning identifies submatrices that must be updated in order to be able to move the boundary (indicated by the double lines) forward. Notice that using this new partitioning, $\hat{A}_k$ currently contains

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

After moving the double lines, the partitioning of A becomes

$$\left(\begin{array}{c|c} A_{TL}^{(k+1)} & A_{TR}^{(k+1)} \\ \hline A_{BL}^{(k+1)} & A_{BR}^{(k+1)} \end{array}\right) = \left(\begin{array}{c|c} \begin{pmatrix} A_{00}^{(k)} & a_{01}^{(k)} \\ a_{10}^{(k)\,T} & \alpha_{11} \end{pmatrix} & \begin{pmatrix} A_{02}^{(k)} \\ a_{12}^{(k)\,T} \end{pmatrix} \\ \hline \begin{pmatrix} A_{20}^{(k)} & a_{21}^{(k)} \end{pmatrix} & A_{22}^{(k)} \end{array}\right)$$

and the partitionings of $\hat{A}_{k+1}$, L, and U change similarly. Thus, $\hat{A}_{k+1}$ must contain

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

In summary, in order to maintain the loop-invariant, the contents of $\hat{A}$ must be updated like

$$\left(\begin{array}{c|c|c} L\backslash U_{00}^{(k)} & \left(\begin{array}{c|c} a_{01}^{(k)} & A_{02}^{(k)} \end{array}\right) \\ \hline \left(\begin{array}{c} a_{10}^{(k)\,T} \\ \hline A_{20}^{(k)} \end{array}\right) & \left(\begin{array}{c|c} \alpha_{11}^{(k)} & a_{12}^{(k)\,T} \\ \hline a_{21}^{(k)} & A_{22}^{(k)} \end{array}\right) \end{array}\right) \quad \rightarrow \quad \left(\begin{array}{c|c} \left(\begin{array}{c|c} L\backslash U_{00}^{(k)} & u_{01}^{(k)} \\ \hline l_{10}^{(k)\,T} & v_{11}^{(k)} \end{array}\right) & \left(\begin{array}{c} A_{02}^{(k)} \\ \hline a_{12}^{(k)\,T} \end{array}\right) \\ \hline \left(\begin{array}{c|c} A_{20}^{(k)} & a_{21}^{(k)} \end{array}\right) & A_{22}^{(k)} \end{array}\right)$$

Thus, it suffices to compute $u_{01}^{(k)}$, $l_{10}^{(k)}$, and $v_{11}^{(k)}$, overwriting the corresponding parts $a_{01}^{(k)}$, $a_{10}^{(k)}$, and $\alpha_{11}^{(k)}$.

To determine how to compute these quantities, consider

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

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

$$\begin{array}{l|l|l} A_{00}^{(k)} = L_{00}^{(k)} U_{00}^{(k)} & a_{01}^{(k)} = L_{00}^{(k)} u_{01}^{(k)} & A_{02}^{(k)} = L_{00}^{(k)} U_{02}^{(k)} \\ \hline a_{10}^{(k)T} = l_{10}^{(k)T} U_{00}^{(k)} & \alpha_{11}^{(k)} = l_{10}^{(k)T} u_{01}^{(k)} + v_{11}^{(k)} & a_{12}^{(k)T} = l_{10}^{(k)T} U_{02}^{(k)} + u_{12}^{(k)T} \\ \hline A_{20}^{(k)} = L_{20}^{(k)} U_{00}^{(k)} & a_{21}^{(k)} = L_{20}^{(k)} U_{01}^{(k)} + l_{21}^{(k)} v_{11}^{(k)} & A_{22}^{(k)} = L_{20}^{(k)} U_{02}^{(k)} + l_{21}^{(k)} u_{12}^{(k)T} + L_{22}^{(k)} U_{22}^{(k)} \end{array} \tag{5}$$

To compute $u_{01}^{(k)}$ one 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 solve the triangular system $l_{10}^{(k)\,T} U_{00}^{(k)} = a_{10}^{(k)\,T}$. The result can overwrite $a_{10}^{(k)\,T}$. To determine v_{11} we merely compute $v_{11}^{(k)} = \alpha_{11}^{(k)} - l_{10}^{(k)\,T} u_{01}^{(k)}$. The result can overwrite $\alpha_{11}^{(k)}$. This motivates the algorithm in Fig. 2 (left) for overwriting a given non-singular, $n \times n$ matrix A with its LU factorization.

To demonstrate that in deriving the algorithm we have constructively proven its correctness, consider the following:

THEOREM 3.1. *The algorithm in Fig. 2 (left) overwrites a given non-singular, $n \times n$ matrix, A, with its LU factorization.*

PROOF. To prove this theorem, we merely invoke the Fundamental invariance theorem. Here the guard B is $A_{BR} \neq 0 \times 0$, predicate P is

$$\hat{A} \text{ contains } = \left(\begin{array}{c|c} L\backslash U_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \quad \text{where} \quad L\backslash U_{TL} \text{ is } k \times k$$

and the statement S is the body of the loop in Fig. 2 (left).

First, notice that the statement

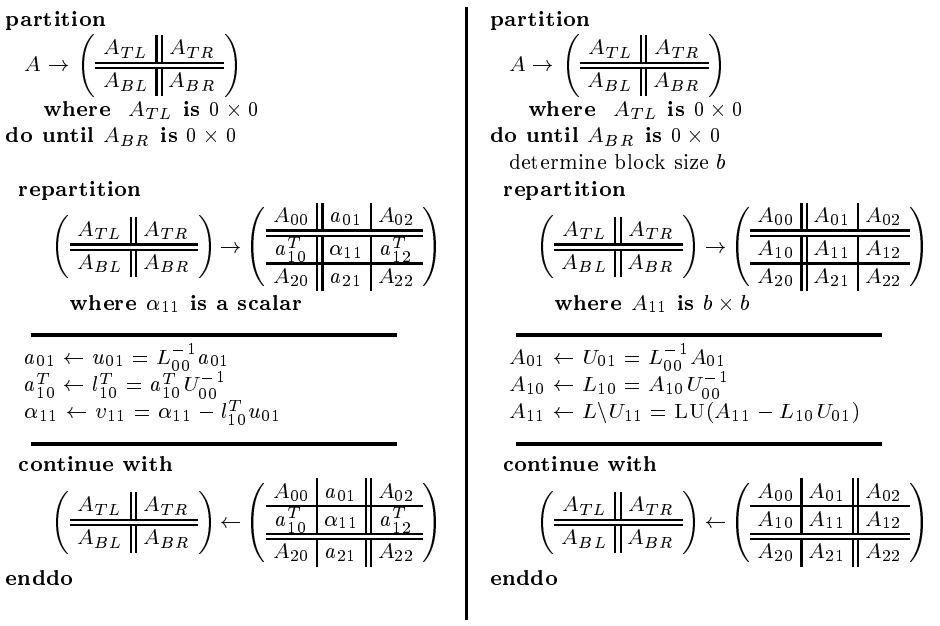


Fig. 2. Unblocked and blocked versions of the lazy variant for computing the LU factorization of a square matrix A (without pivoting).

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

has the property that after its execution P holds since $L\backslash U_{TL}$, A_{TR}, and A_{BL} are all empty (they have row and/or column dimensions equal to zero) and $A_{BR} = A$. Thus, just before the loop is first entered

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

and we conclude that P holds when $k = 0$.

Recall that the body of the loop was developed so that $\{P \wedge B\} S \{P\}$ holds, i.e. if the condition holds at the top of the loop, then it holds at the bottom of the loop (just before the enddo). Also, since at each step the size of A_{BR} decreases by one, guard B will eventually become *false*, $\{P\}$ **do** $B \rightarrow S$ **od** *true* holds (i.e. execution of the loop begun in a state in which P is *true* terminates). We have shown that all of the conditions of the Fundamental invariance theorem hold. We therefore conclude that if the loop is entered in a state where P holds, it will complete in a state where P is true and guard B is false.

This means that $\hat{A}$ contains $\left(\begin{array}{c|c} L\backslash U_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$ where A_{BR} is 0×0 and completion of the loop transpires when $k = n$. Thus the final contents of the matrix are $\hat{A} = L\backslash U_{TL}$ where L_{TL} and U_{TL} are unit-lower and upper-triangular matrices of

order n. We conclude that upon exiting the loop, the matrix has been overwritten by 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 (1) has been satisfied. The question is now how to compute $\hat{A}_{k+b}$ from $\hat{A}_k$ for some small block size b (i.e. $1 < b \ll n$). To answer this, repartition

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

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

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

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

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

Thus,

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

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

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

The preceding discussion motivates the algorithm in Fig. 2 (right) and Fig. 3(b) for overwriting the given non-singular, $n \times n$ 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.

The proof of the following theorem is similar to that of Theorem 3.1.

THEOREM 3.2. *The algorithm in Fig. 2 (right) overwrites a given non-singular, $n \times n$ matrix, A, with its LU factorization.*

3.4 Row-lazy algorithm

As a point of reference, Stewart [Stewart 1998] calls this algorithm Pickett's charge south.

Let us assume that only (1) and (2) 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 (6). 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

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

Thus, it suffices to compute $L_{10}^{(k)}$, $L\backslash U_{11}^{(k)}$, and $U_{12}^{(k)}$. Recalling the equalities in (7) we notice that

(1) To compute $L_{10}^{(k)}$ we can 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 can 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 can update $A_{12}^{(k)} \leftarrow A_{12}^{(k)} - L_{10}^{(k)}U_{02}^{(k)}$ after which we 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 Fig. 3(c).

The proof of the following theorem is similar to that of Theorem 3.1.

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

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

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

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

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

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

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

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

Fig. 3. Blocked versions of LU factorization without pivoting for five commonly encountered variants. The different variants share the skeleton that partitions and repartitions the matrix. Executing the operations in one of the five boxes yields a specific algorithm.

THEOREM 3.3. *The algorithm in Fig. 3(c) overwrites a given non-singular, $n \times n$ matrix, A, with its LU factorization.*

3.5 Column-lazy algorithm

This algorithm is referred to as a left-looking algorithm in [Dongarra et al. 1991] while Stewart [Stewart 1998] calls it Pickett's charge east.

Let us assume that only (1) and (3) have been satisfied. Now it suffices to compute $U_{01}^{(k)}$, $L\backslash U_{11}^{(k)}$, and $L_{21}^{(k)}$. Using the same techniques as before one derives the algorithm in Fig. 3 (d). Again, this algorithm overwrites the given non-singular, $n \times n$ matrix, A, with its LU factorization.

The proof of the following theorem is similar to that of Theorem 3.1.

THEOREM 3.4. *The algorithm in Fig. 3(d) overwrites a given non-singular, $n \times n$*

matrix, A, with its LU factorization.

3.6 Row-column-lazy algorithm

This algorithm is often referred to as Crout's methods in the literature [Crout 1941].

We assume that only (1), (2), and (3) 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. 3 (e). Again, this algorithm overwrites a given non-singular, $n \times n$ matrix, A, with its LU factorization.

The proof of the following theorem is similar to that of Theorem 3.1.

THEOREM 3.5. *The algorithm in Fig. 3(e) overwrites a given non-singular, $n \times n$ matrix, A, with its LU factorization.*

3.7 Eager algorithm

This algorithm is often referred to as classical Gaussian elimination.

We proceed under the assumption that (1), (2), and (3) have been satisfied, and as much of (4) as possible has been computed, without completing the computation of any part of L_{BR} and U_{BR}. Repartition A, L, U, and $\hat{A}_k$ conformally as in (6). Notice, our assumption is that $\hat{A}_k$ holds

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

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

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

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

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

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

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

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

The resulting algorithm is given in Fig. 3(a). Notice that this algorithm is the blocked equivalent to the algorithm derived in Section 3.1.

The proof of the following theorem is similar to that of Theorem 3.1.

THEOREM 3.6. *The algorithm in Fig. 3(a) overwrites a given non-singular, $n \times n$ matrix, A, with its LU factorization.*

4. A RECIPE FOR DERIVING ALGORITHMS

The derivations of the different algorithmic variants of LU factorization, detailed above, were extremely systematic. The following recipe was 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 LU factorization, this has to do with the fact that blocks of zeroes must be isolated in L and U, as they are triangular matrices.
(3) Multiply out all matrix products corresponding to this partitioning.
(4) Equate the submatrix relations that result from the partitioning of Step 3. These define computations that the algorithm must perform in order to maintain correctness.
(5) Pick a loop-invariant from the set of possible loop-invariants that satisfy the equations given in Step 4. Notice that this loop-invariant plays the role of an induction hypothesis.
(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) Determine the computations required to transform (update) the contents indicated in 6b to those indicated in 6c, (Naturally, it must be verified that these computations are possible.)
(7) Update the partitioning of the matrices.
(8) Continue until the partitioning yields the null matrix for the "BR" submatrix.
(9) Classify the algorithm. We have developed a systematic way of classifying the derived algorithms based upon the nature of the inductive step of the algorithm. While we use this classification in the labeling of the algorithms in the previous section, we will not go into further detail here.

A more complete recipe for a broader class of linear algebra operations can be found in [Gunnels and van de Geijn 2001a].

We again point out that the recipe implicitly provides a proof of correctness for the algorithm since Steps 5–6d emulate the proof by mathematical induction. Further, the technique employed for deriving these variants of LU factorization generalizes to other factorization algorithms, e.g. Cholesky and QR.

5. ENCODING THE ALGORITHM IN C

In this section we briefly discuss how dense linear algebra algorithms, as presented in Figs. 1–3, can be translated into code. We first show a more traditional approach

as it appears in popular packages like LAPACK. Next, we present an alternative framework that allows implementation at a higher level of abstraction that mirrors how we naturally present the algorithms. This second approach has been successfully used in PLAPACK and our FLAME framework represents a refinement of this methodology.

5.1 Classic implementation with the BLAS

Let us consider the blocked eager algorithm for the LU factorization presented in Fig. 3 (a). This algorithm requires an LU factorization of a small matrix, $A_{11} \leftarrow L\backslash U_{11} = \text{LU fact.}(A_{11})$, 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) (calls to the routines `DTRSM` and `DGEMM`, respectively). To understand this code, it helps to consider the partitioning of the matrix for a typical loop index j, as illustrated in Fig. 4: 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)`. The resultant code is given in Fig. 5.

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

5.2 The algorithm *is* the code

We would argue that it is relatively easy to generate the code in Fig. 5 given the algorithm in Fig. 3(a) and the picture in Fig. 4. However, the translation of the algorithm to the code is made tedious and error-prone by the fact that one has to think very carefully about indices and matrix dimensions. While this is not much of a problem if one only had to implement just one algorithm, real difficulties may arise when implementing a number of possible algorithmic variants for a given operation or, in the case of a library such as LAPACK, implementing even a single such variant of each of a large number of operations. One becomes even more acutely aware of these issues when distributed-memory architectures enter the picture, as in ScaLAPACK.

In an effort to make the code look like the algorithms given in Fig. 3, while simultaneously accounting for the constraints imposed by C and Fortran, we have developed FLAME. The algorithmic and code skeletons shared by the five variants for the LU factorization, developed earlier in this paper, are given in Figs. 6 and 7, respectively. 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, in this case the row dimension of the matrix. Finally, `ATL`, `A00`, etc. are simply references into the original array described by `A`.

If one is familiar with the coding conventions used to name the BLAS kernels, it is clear that the following code segments, when entered in the appropriate place (lines 22-34) in the code in Fig. 7, implement the different variants of the LU factorization:

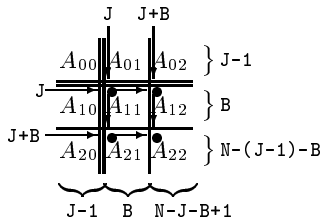


Fig. 4. Partitioning of matrix A with all dimensions annotated when $A_{00} = A_{TL}$ is $(j-1) \times (j-1)$.

```
  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 )
     CALL LU_EAGER_LEVEL2( B, A( J,J ), LDA )

     IF ( J+B <= N ) THEN
C                                              A12 <- U12 = inv( L11 ) * A12

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

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

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

C                                              A22 <- A22 - A21 * A12

        CALL 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
```

Fig. 5. Fortran implementation of blocked eager LU factorization algorithm using the BLAS. (Find the bug without referring to Fig. 4 or the text!)

Lazy algorithm

```
23  FLA_Trsm(FLA_RIGHT, FLA_UPPER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
24           ONE, A00, A10);
25  FLA_Trsm(FLA_LEFT,  FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
26           ONE, A00, A01);
27  FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A10, A01, ONE, A11);
28  FLA_LU_nopivot_level2(A11);
```

Partition $A = \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)$
where A_{TL} is 0×0
do until A_{BR} is 0×0
Repartition
$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c||c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right)$
where A_{11} is $b \times b$

insert update here

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

Fig. 6. Algorithm skeleton for LU factorization without pivoting.

```
1   #include "FLAME.h"
2
3   void FLA_LU_nopivot_skeleton( FLA_Obj A, nb_alg )
4   {
5     FLA_Obj     ATL, ATR,   A00, A01, A02,
6                 ABL, ABR,   A10, A11, A12,
7                             A20, A21, A22;
8
9     FLA_Part_2x2( A,  &ATL, /**/ &ATR,
10                  /* ************** */
11                    &ABL, /**/ &ABR,
12              /* with */ 0, /* by */ 0, /* submatrix */ FLA_TL );
13
14    while ( b=min(min(FLA_Obj_length( ABR ), FLA_Obj_width( ABR )), nb_alg) != 0 )
15     {
16      FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
17                          /* ************** */   /* ******************** */
18                                 /**/             &A10, /**/ &A11, &A12,
19                            ABL, /**/ ABR         &A20, /**/ &A21, &A22,
20              /* with */ b, /* by */ b, /* A11 split from */ FLA_BR );
21      /* ************************************************************************ */
                        insert code for update here
31      /* ************************************************************************ */
32      FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,      A00, A01, /**/ A02,
33                                      /**/            A10, A11, /**/ A12,
34                               /* ************** */ /* ****************** */
35                                 &ABL, /**/ &ABR,     A20, A21, /**/ A22,
36              /* with A11 added to submatrix */ FLA_TL );
37    }
38  }
```

Fig. 7. A code skeleton for the C implementation of many of the blocked LU factorization algorithms using FLAME.

Row-lazy algorithm

```
23  FLA_Trsm(FLA_RIGHT, FLA_UPPER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
24           ONE, A00, A10);
25  FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A10, A01, ONE, A11);
26  FLA_LU_nopivot_level2(A11);
27  FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A10, A02, ONE, A12);
28  FLA_Trsm(FLA_LEFT,  FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
29           ONE, A11, A12);
```

Column-lazy algorithm

```
23  FLA_Trsm(FLA_LEFT,  FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
24           ONE, A00, A01);
25  FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A10, A01, ONE, A11);
26  FLA_LU_nopivot_level2(A11);
27  FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A20, A01, ONE, A21);
28  FLA_Trsm(FLA_RIGHT, FLA_UPPER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
29           ONE, A11, A21);
```

Row-column-lazy algorithm

```
23  FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A10, A01, ONE, A11);
24  FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A20, A01, ONE, A21);
25  FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A10, A02, ONE, A12);
26  FLA_LU_nopivot_level2(A11);
27  FLA_Trsm(FLA_LEFT,  FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
28           ONE, A11, A12);
29  FLA_Trsm(FLA_RIGHT, FLA_UPPER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
30           ONE, A11, A21);
```

Eager algorithm

```
23  FLA_LU_nopivot_level2( A11 );
24  FLA_Trsm(FLA_LEFT,  FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
25           ONE, A11, A12);
26  FLA_Trsm(FLA_RIGHT, FLA_UPPER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
27           ONE, A11, A21);
28  FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A21, A12, ONE, A22);
```

5.3 Positive features of the FLAME approach

Naturally, one can argue that determining which of the two methods for coding the algorithms might be deemed "superior" is simply a matter of taste. However, to support our case, we list the following questions and/or observations:

—What if a bug were introduced into the code in Fig. 5? Indeed, in that code we "accidentally" replaced `N-(J-1)-B` with `N-J-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 is not as easily introduced into the code implemented using FLAME since it does not contain indices. Furthermore, with this approach it is easy to perform a run-time check in order to determine if the dimensions of the different matrix operands passed to a routine are consistent.

—When coding all variants of the LU factorization one inherently has to derive all algorithms, leading to descriptions like those given in Fig. 3. However, translating

those to code like that given in Fig. 5 would require *several* careful considerations of the picture in Fig. 4. Moreover, due to the detailed and extensive indexing involved in that approach, considerable testing would be required before one could declare the code bug-free. By contrast, given the algorithms, it has been argued that generating all variants using FLAME is straightforward. As has already been mentioned, since the code closely resembles the algorithm, one can be much more confident about its correctness before the code is 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_`. This highlights the one-to-one correspondence between FLAME and PLAPACK codes; this correspondence is found to be lacking when one considers LAPACK and ScaLAPACK codes in the same light.

—What if we needed a parallel out-of-core 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_`.

5.4 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, given at the end of this paper.

5.5 Proving the implementation correct

In Section 3.3 we proved correctness of the lazy algorithm and in subsequent subsections of Section 3 asserted that the correctness of the other algorithms can be established in much the same way. If the routines called by the described FLAME code correctly implement the operations implied by their names, then it can be argued that the code itself is correct. Indeed, debugging is not necessary.

There are a number of reasons that we feel comfortable in making such a bold assertion. The justifications for the statement rely upon features of both our systematic algorithmic design methodology, the library supporting the implementation of the algorithm, and to the relationship between the two.

The manner in which we systematically generate algorithms relies, primarily, on two design pillars, which together make up FLAME. The first is that we have limited the class of problems under consideration to those in linear algebra. The second is that our algorithms consistently build upon the fundamental invariance theorem. This restriction leads to the development of algorithms whose correctness can be established.

Naturally, FLAME is designed to express these systematically generated algorithms in a manner that is both concise and unambiguous. Therefore, the FLAME code can be made to mirror the algorithms thus produced. This leads one to conclude that the two most common sources of error are eliminated. The translation from algorithm to code is easily automatable because of the one-to-one relation between the two, so that a very common mistake, namely the code not reflecting the algorithm (when one considers a textual version of the algorithm as it might be

presented in a textbook), can be obviated. A second common mistake made with such codes, indexing errors, is eliminated from the top-level expression of FLAME code because FLAME does no explicit indexing. To be certain, there are a *few* support routines within FLAME that perform indexing. However, these routines are so small that they are amenable to both standard proof-of-correctness techniques and to truly "exhaustive" testing. In a sense, these routines are analogous to FLAME's "assembly language" and their reliability is comparable to that of a robust compiler.

Because our method of derivation leads to a class of algorithms whose proof of correctness is straightforward and since the language we use to express the produced algorithms should not lead to any (unintentional) mistranslation from algorithm to code, we feel that the *coupled* system leads to programs whose correctness follows from a mathematical derivation of the algorithm.

6. LU FACTORIZATION WITH PARTIAL PIVOTING

We now demonstrate that the techniques that we introduced using the example of LU factorization without pivoting are also applicable to the case of LU factorization with partial pivoting. The latter algorithm is the one commonly implemented, but involves complications that have traditionally made its derivation coding a more intricate and time-consuming procedure.

6.1 Notation

Let I_m denote the $m \times m$ identity matrix and $\tilde{P}_m(i)$ be the $m \times m$ permutation matrix such that $\tilde{P}_m(i)A$ only swaps the first and ith rows of A. Here, we consider an $m \times n$ matrix, A, where $m \geq n$ and define

$$P_m(p_0, p_1, \cdots, p_{k-1}) = \begin{pmatrix} I_{k-1} & 0 \\ 0 & \tilde{P}_{m-k+1}(p_{k-1}) \end{pmatrix} \cdots \begin{pmatrix} I_1 & 0 \\ 0 & \tilde{P}_{m-1}(p_1) \end{pmatrix} \tilde{P}_m(p_0)$$

and $P_{m;i:j} = P_m(p_i, \ldots, p_j)$. Here p_k equals the index, relative to the top row of the currently active matrix (A_{BR} in previous discussions), of the row that is swapped at the kth step of LU factorization with partial pivoting. Thus $P_m(p_0, p_1, \cdots, p_{k-1})A$ equals the matrix that results after swapping rows 0 and p_0 followed by swapping rows 1 and $p_1 + 1$, etc., in that order. Also, $P_{m;i:j}A$ equals the matrix that results after swapping rows i and p_i followed by $i + 1$ and $p_{i+1} + 1$, etc., in that order.

It is well-known that LU factorization with partial pivoting produces the LU factorization

$$P_{m;0:n-1}A = LU \tag{8}$$

6.2 Derivation of the invariants

Now, let us examine the possible contents of matrix $\tilde{A}_k = \mathcal{P}A$, where $\mathcal{P} = P_{m;0:k-1}$, the matrix as it has been overwritten partially into the LU factorization with partial pivoting. Eqn. 8 is equivalent to

$$\left(\begin{array}{c||c} I_k & 0 \\ \hline\hline 0 & P_{m-k;k:n-1} \end{array}\right) \tilde{A}_k = LU$$

or

$$\tilde{A}_k = \left(\begin{array}{c||c} I_k & 0 \\ \hline\hline 0 & Q^T \end{array}\right) LU$$

where

$$Q = P_{m-k;k:n-1}$$

Partitioning

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

we find that

$$\begin{aligned} \left(\begin{array}{c||c} \tilde{A}_{TL}^{(k)} & \tilde{A}_{TR}^{(k)} \\ \hline\hline \tilde{A}_{BL}^{(k)} & \tilde{A}_{BR}^{(k)} \end{array}\right) &= \left(\begin{array}{c||c} I_k & 0 \\ \hline\hline 0 & Q^T \end{array}\right) \left(\begin{array}{c||c} L_{TL}^{(k)} & 0 \\ \hline\hline L_{BL}^{(k)} & L_{BR}^{(k)} \end{array}\right) \left(\begin{array}{c||c} U_{TL}^{(k)} & U_{TR}^{(k)} \\ \hline\hline 0 & U_{BR}^{(k)} \end{array}\right) \\ &= \left(\begin{array}{c||c} L_{TL}^{(k)} U_{TL}^{(k)} & L_{TL}^{(k)} U_{TR}^{(k)} \\ \hline\hline \tilde{L}_{BL}^{(k)} U_{TL}^{(k)} & \tilde{L}_{BL}^{(k)} U_{TR}^{(k)} + \tilde{L}_{BR}^{(k)} U_{BR}^{(k)} \end{array}\right) \end{aligned}$$

where $L_{BL} = Q\tilde{L}_{BL}^{(k)}$ and $L_{BR} = Q\tilde{L}_{BR}^{(k)}$. Thus, for $0 \leq k < n$, the equalities in Eqns. 1–4 must again hold, except that $L_{BL}^{(k)}$, $L_{BR}^{(k)}$, and $A^{(k)}$, are now replaced by $\tilde{L}_{BL}^{(k)}$, $\tilde{L}_{BR}^{(k)}$, and $\tilde{A}^{(k)}$, respectively. We mention, as before, that unaccented submatrices of L and U denote final values. As for LU factorization without pivoting, different conditions on the contents of $\hat{A}_k$ logically dictate different variants for computing the LU factorization with partial pivoting. These are given in Table I, with the provisos mentioned above. Notice that in addition, a necessary condition is that $p_0, \ldots, p_{k-1}$ have been computed.

The second and third conditions listed in Table I are impractical since the computation of $p_0, \ldots, p_{k-1}$ requires that the entries of $L_{BL}^{(k)}$ be computed. By taking entries 4 through 6, listed in Table I, together with the requirement that $p_0, \ldots, p_{k-1}$ have been computed, and using them as part of predicate P, three different variants for LU factorization with partial pivoting can be derived. These conditions again lead to column-lazy (left-looking), row-column-lazy (Crout), and eager (right-looking) variants, respectively, this time with partial pivoting incorporated.

6.3 Derivation of the eager algorithm

Let us concentrate on the eager algorithm. Notice, our assumption is that $\hat{A}_k$ holds

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

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

$$\hat{A}_{k+b} = \left(\begin{array}{c||c} L\backslash U_{TL}^{(k+b)} & U_{TR}^{(k+b)} \\ \hline\hline \bar{L}_{BL}^{(k+b)} & \hat{A}_{BR}^{(k+b)} \end{array}\right)$$

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

where, $Q_1 = P_{m-k;k:k+b-1}$, $\bar{A}_{BR}^{(k)} = Q_1 \tilde{A}_{BR}^{(k)}$, and $\left(\frac{\bar{L}_{10}^{(k)}}{\bar{L}_{20}^{(k)}}\right) \leftarrow Q_1 \left(\frac{\tilde{L}_{10}^{(k)}}{\tilde{L}_{20}^{(k)}}\right)$. Note that $L\backslash U_{11}^{(k)}$ and $L_{21}^{(k)}$ are defined by Eqn. 9, below, and $L_{10}^{(k)} = \bar{L}_{10}^{(k)}$.

With some effort it can be verified that the following updates have the desired effect:

—Compute Q_1, given by $\{p_k, \ldots, p_{k+b-1}\}$, $L_{11}^{(k)}$, $U_{11}^{(k)}$, and $\bar{L}_{21}^{(k)}$ such that

$$\left(\frac{\hat{A}_{11}^{(k)}}{\hat{A}_{21}^{(k)}}\right) = \left(\frac{L_{11}^{(k)}}{\bar{L}_{21}^{(k)}}\right) U_{11}^{(k)} \tag{9}$$

overwriting

$$\left(\frac{\hat{A}_{11}^{(k)}}{\hat{A}_{21}^{(k)}}\right) \leftarrow \left(\frac{L\backslash U_{11}^{(k)}}{\bar{L}_{21}^{(k)}}\right)$$

—Permute and overwrite: $\left(\frac{\hat{A}_{10}^{(k)}}{\hat{A}_{20}^{(k)}}\right) \leftarrow Q_1 \left(\frac{\tilde{L}_{10}^{(k)}}{\tilde{L}_{20}^{(k)}}\right)$.

—Permute and overwrite: $\left(\frac{\hat{A}_{12}^{(k)}}{\hat{A}_{22}^{(k)}}\right) \leftarrow Q_1 \left(\frac{\hat{A}_{12}^{(k)}}{\hat{A}_{22}^{(k)}}\right)$.

—Update $\hat{A}_{12}^{(k)} \leftarrow U_{12}^{(k)} = L_{11}^{-1(k)} \hat{A}_{12}^{(k)}$ and $\hat{A}_{22}^{(k)} \leftarrow \hat{A}_{22}^{(k)} - \bar{L}_{21}^{(k)} U_{12}^{(k)}$.

In Fig. 8 we show how an eager blocked LU factorization with partial pivoting can be expressed in our algorithmic format. In this algorithm, the operation $\mathrm{LU}_{\mathrm{piv}}(B)$ returns the result of an LU factorization with partial pivoting of matrix B, as well as the pivot indices. In that figure, p_1 is a vector of pivot indices and $P(p_1)$ takes the place of $P_{m-k;k:k+b-1}$.

An unblocked algorithm results when the block size, b, is always chosen to equal unity. In this case, the operation

$$\left[A_{BR}^{(1)}, p_1\right] \leftarrow \left[\left(\frac{L\backslash U_{11}}{L_{21}}\right), p_1\right] = \mathrm{LU}_{\mathrm{piv}}(A_{BR}^{(1)}) \tag{10}$$

is replaced by a determination of the index of the element in vector $A_{BR}^{(1)}$, followed by a swap of that element with the first element of that vector, and finally a scaling of the elements of A_{21} by $1/A_{11}$. (Notice that now A_{21} is a vector and A_{11} is a scalar.) In other words, the operation in Eqn. 10 is replaced by

$$\begin{array}{l} \text{Choose } p_1 \text{ s.t. } |\left[A_{BR}^{(1)}\right]_{p_1}| = \max_i |\left[A_{BR}^{(1)}\right]_i| \\ \text{Swap } \left[A_{BR}^{(1)}\right]_1 \leftrightarrow \left[A_{BR}^{(1)}\right]_{p_1} \\ A_{21} \leftarrow L_{21} = A_{21}/A_{11} \end{array}$$

Partition $A = \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)$ and $p = \left(\begin{array}{c} p_T \\ \hline\hline p_B \end{array}\right)$
where A_{TL} is 0×0 and p_T has 0 elements
do until A_{BR} is 0×0
Determine block size b
Partition
$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c||c|c} A_{00} & A_{01} & A_{02} \\ \hline\hline A_{10} & A_{11} & A_{12} \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right)$
where A_{11} is $b \times b$
Partition
$\left(\begin{array}{c} p_T \\ \hline\hline p_B \end{array}\right) = \left(\begin{array}{c} p_0 \\ \hline\hline p_1 \\ \hline\hline p_2 \end{array}\right)$
where p_1 has b elements
Partition
$A_{BR} = \left(\begin{array}{c|c} A_{BR}^{(1)} & A_{BR}^{(2)} \end{array}\right)$
where $A_{BR}^{(1)}$ has width b.

$\left[A_{BR}^{(1)}, p_1\right] \leftarrow \left[\left(\begin{array}{c} L\backslash U_{11} \\ \hline\hline L_{21} \end{array}\right), p_1\right] = \mathrm{LU}_{\mathrm{piv}}(A_{BR}^{(1)})$
$A_{BL} \leftarrow P(p_1)A_{BL}$
$A_{BR}^{(2)} \leftarrow P(p_1)A_{BR}^{(2)}$
$A_{12} \leftarrow U_{12} = L_{11}^{-1}A_{12}$
$A_{22} \leftarrow A_{22} - L_{21}U_{12}$

Continue with
$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c||c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right)$
$\left(\begin{array}{c} p_T \\ \hline\hline p_B \end{array}\right) = \left(\begin{array}{c} p_0 \\ \hline\hline p_1 \\ \hline\hline p_2 \end{array}\right)$
enddo

Fig. 8. Eager blocked LU factorization with partial pivoting.

Here $[x]_i$ indicates the ith element of vector x. It is important to realize that multiple partitionings of the same matrix reference the same data. Thus after swapping the elements of $A_{BR}^{(1)}$, A_{11} contains what was $\left[A_{BR}^{(1)}\right]_{p_1}$ before the swap.

6.4 Implementation

A FLAME implementation of the blocked algorithm in Fig. 8 is given in Fig. 9. Notice that a FLAME implementation of the unblocked algorithm would look similar. Let us assume that the latter is correctly implemented in the FLAME routine

```
void FLA_LU_level2( FLA_Obj A, FLA_Obj ipiv )
```

Now, the correctness of algorithm in Fig. 8 depends only on the correctness of the LU factorization with partial pivoting of $A_{BR}^{(1)}$ and the other operation. Thus, there

```
1  void FLA_LU( FLA_Obj A, FLA_Obj ipiv, int nb_alg )
2  {
3    < declarations >
4
5    FLA_Part_2x2( A,  &ATL, /**/ &ATR,
6                  /* ************** */
7                     &ABL, /**/ &ABR,
8                 /* with */ 0, /* by */ 0, /* submatrix */ FLA_TL );
9    FLA_Part_2x1( ipiv, &ipivT,
10                    /* ****** */
11                       &ipivB,
12                /* with */ 0, /* length submatrix */ FLA_TOP );
13
14   while (b = min( min( FLA_Obj_length( ABR ), FLA_Obj_width( ABR ) ), nb_alg ))
15   {
16     FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,         &A00, /**/ &A01, &A02,
17                         /* ************* */    /* ********************* */
18                                  /**/              &A10, /**/ &A11, &A12,
19                            ABL, /**/ ABR,         &A20, /**/ &A21, &A22,
20                 /* with */ b, /* by */ b, /* A11 split from */ FLA_BR );
21     FLA_Repart_2x1_to_3x1( ipivT,                 &ipiv0,
22                         /* ***** */             /* ***** */
23                                                   &ipiv1,
24                            ipivB,                 &ipiv2,
25                 /* with */ b, /* length ipiv1 split from */ FLA_BOTTOM );
26     FLA_Part_1x2( ABR,   &ABR_1, &ABR_2,
27                 /* with */ b, /* width submatrix */ FLA_LEFT );
28     /* ********************************************************************* */
29
30     if ( nb_alg <= 4 ) FLA_LU_level2(ABR_1, ipiv1);
31     else               FLA_LU       (ABR_1, ipiv1, nb_alg/2);
32
33     FLA_Apply_pivots(FLA_SIDE_LEFT, FLA_NO_TRANSPOSE, ipiv1, ABL);
34     FLA_Apply_pivots(FLA_SIDE_LEFT, FLA_NO_TRANSPOSE, ipiv1, ABR_2);
35     FLA_Trsm(FLA_SIDE_LEFT, FLA_LOWER_TRIANGULAR,
36              FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
37              ONE, A11, A12);
38     FLA_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A21, A12, ONE, A22);
39
40     /* ********************************************************************* */
41     FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,         A00, A01, /**/ A02,
42                                     /**/               A10, A11, /**/ A12,
43                               /* ************** */    /* ****************** */
44                                 &ABL, /**/ &ABR,         A20, A21, /**/ A22,
45                 /* with A11 added to submatrix */ FLA_TL );
46     FLA_Cont_with_3x1_to_2x1( &ipivT,                  ipiv0,
47                                                        ipiv1,
48                               /* ***** */             /* ***** */
49                                 &ipivB,                  ipiv2,
50                 /* with ipiv1 added to */ FLA_TOP );
51   }
52 }
```

Fig. 9. FLAME recursive LU factorization with partial pivoting.

SYMM	$C \leftarrow \alpha(L + \hat{L}^T)B + \beta C$		$C \leftarrow \alpha(U + \hat{U}^T)B + \beta C$	
	$C \leftarrow \alpha B(L + \hat{L}^T) + \beta C$		$C \leftarrow \alpha B(U + \hat{U}^T) + \beta C$	
SYRK	$\mathrm{lo}(C) \leftarrow \alpha\mathrm{lo}(AA^T) + \beta\mathrm{lo}(C)$		$\mathrm{up}(C) \leftarrow \alpha\mathrm{up}(AA^T) + \beta\mathrm{up}(C)$	
	$\mathrm{lo}(C) \leftarrow \alpha\mathrm{lo}(A^TA) + \beta\mathrm{lo}(C)$		$\mathrm{up}(C) \leftarrow \alpha\mathrm{up}(A^TA) + \beta\mathrm{up}(C)$	
SYR2K	$\mathrm{lo}(C) \leftarrow \alpha\mathrm{lo}(AB^T + BA^T) + \beta\mathrm{lo}(C)$		$\mathrm{up}(C) \leftarrow \alpha\mathrm{up}(AB^T + BA^T) + \beta\mathrm{up}(C)$	
	$\mathrm{lo}(C) \leftarrow \alpha\mathrm{lo}(A^TB + B^TA) + \beta\mathrm{lo}(C)$		$\mathrm{up}(C) \leftarrow \alpha\mathrm{up}(A^TB + B^TA) + \beta\mathrm{up}(C)$	
TRMM	$B \leftarrow \alpha LB$	$B \leftarrow \alpha L^TB$	$B \leftarrow \alpha UB$	$B \leftarrow \alpha U^TB$
	$B \leftarrow \alpha BL$	$B \leftarrow \alpha BL^T$	$B \leftarrow \alpha BU$	$B \leftarrow \alpha BU^T$
TRSM	$B \leftarrow \alpha L^{-1}B$	$B \leftarrow \alpha L^{-T}B$	$B \leftarrow \alpha U^{-1}B$	$B \leftarrow \alpha U^{-T}B$
	$B \leftarrow \alpha BL^{-1}$	$B \leftarrow \alpha BL^{-T}$	$B \leftarrow \alpha BU^{-1}$	$B \leftarrow \alpha BU^{-T}$

Fig. 10. Level-3 BLAS operations implemented as part of the productivity experiment.

is the option of implementing the LU factorization of the panel $A_{BR}^{(1)}$ as a recursive call to the given routine (line 31). Only when the panel becomes very small is a routine that uses level-2 BLAS (matrix-vector computations) called (line 30).

Notice that the implementation is very flexible in that it is neither purely recursive nor purely iterative. By playing with the algorithmic block size b (`nb_alg`), one can attain a purely recursive algorithm (when $b = n/2$ for an $m \times n$ input matrix A), purely iterative (by always calling `FLA_LU_level2` for the subproblem) or an iterative algorithm that recursively calls itself. An induction on the level of the recursion would establish the correctness of the given code. A more detailed discussion on the correctness of recursively formulated linear algebra algorithms can be found in [Gustavson and Jonsson 2000; Elmroth and Gustavson 2000].

7. EXPERIMENTS

In this section, we report the results of three different experiment. The first measures the impact that the FLAME approach has on productivity. The second experiment demonstrates FLAME make the implementation of high-performance linear algebra algorithms more accessible to novices. In the final experiment we demonstrate that the attained performance is superb.

7.1 Productivity experiment

As an experiment to measure, albeit roughly, the degree to which FLAME reduces code development time, one of the authors implemented all level-3 BLAS operations given in Fig. 10 in terms of matrix-matrix multiplication. This exercise can easily require months to complete, even by a programmer who is experienced in the implementation of such operations. This includes time spent on extensive testing of correctness of the implementations. The entire library of operations was completed using FLAME in a matter of about ten hours, including testing. As of this writing, we have used the resulting library for about nine months without encountering a bug in the implementations. The resulting code is included on the FLAME webpage given at the end of this paper. The prototype implementation of FLAME required to support the implementations of the level-3 BLAS took approximately one man-week.

It should be noted that the number of lines of code required for the implementation is not necessarily less than that required for a more conventional implementa-

tion. This is already evident when considering Figs. 5 and 7. However, the effort is greatly reduced by the fact that the subroutines for the different operations use similar code skeletons. Moreover, we believe that the resulting code is substantially more readable.

7.2 Accessibility experiment

It is our claim that the FLAME approach to the derivation and implementation of linear algebra algorithms greatly simplifies the development of linear algebra libraries. To demonstrate this, we handed a recipe for deriving algorithms, similar to the one in Section 4, to a class of computer science undergraduates at UT-Austin. These students had a limited background in linear algebra and essentially no background in high-performance computing. Using the FLAME approach they implemented blocked algorithms for linear algebra operations that are part of the level-3 BLAS. The results of this experiments can be found in [Gunnels and van de Geijn 2001a].

7.3 Performance experiment

To illustrate that correctness, simplicity and modularity does not necessarily come at the expense of performance, we measured the performance of the LU factorization with pivoting given in Fig. 9 followed by forward and backward substitution, i.e., essentially the LINPACK benchmark. For comparison, we also measured the performance of the equivalent operations provided by ATLAS R3.2 [Whaley and Dongarra 1998].

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

In Fig. 11 we report performance for four different implementations, indicated by the curves marked

`ATLAS:`. This curve reports performance for the LU factorization provided by ATLAS R3.2, using the BLAS provided by ATLAS R3.2.

`ATL-FLAME:`. This curve reports the performance of our LU factorization coded using FLAME with BLAS provided by ATLAS R3.2. 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 provided our own optimized matrix-matrix multiply (ITXGEMM). Details of this optimization are the subject of another paper [Gunnels et al. 2001]. 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 ITX-FLAME 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 high-level 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 R3.2 `DTRSV` routine.

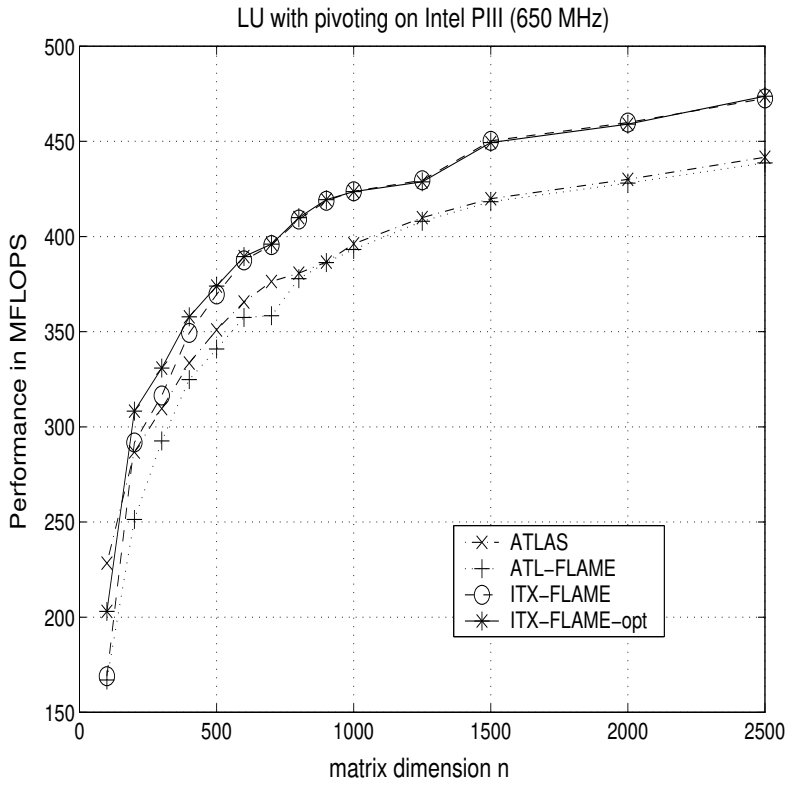


Fig. 11. Performance of LU factorization with pivoting followed by forward and backward substitution.

Notice that for small matrices the performance of `ATL-FLAME` is somewhat inferior to that of ATLAS, due to the overhead for manipulating the objects that encode the information about the matrices. This is due to the fact that this manipulation of objects introduces an $O(n)$ overhead which is amortized over a computational cost that is $O(n^3)$. When the level-2 BLAS based LU factorization is coded without this overhead, the performance is comparable for small matrices. The performance boost witnessed when the ITXGEMM matrix-matrix multiply kernel is used is entirely due to the superior performance of that kernel, relative to the ATLAS `DGEMM` implementation.

It is important to realize that the performance difference between the implementation offered as part of ATLAS R3.2 and our own implementation is not the point of this performance comparison or, more generally, of this paper. With some effort either implementation can be improved to match the performance of the other. Our primary point is that FLAME enables one to expend markedly less time to implement these algorithms in a provably correct manner. At the same time, the resulting implementation attains performance comparable to that of, what are widely considered to be, standard 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 [Gunnels 2001]. The goal is to create a mechanism that will automatically choose between different algorithms based on architectural 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 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 by the library. Thus, the programmer should be allowed to express their algorithms at a higher level of abstraction, e.g. in terms of equations that can be automatically translated to (library) function calls.

9. CONCLUSION

A colleague of ours, Dr. Timothy Mattson of 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 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 from the one that resulted 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 produced 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 variants for a given operation can now be more easily developed we often observe a performance benefit from the approach.

Throughout the paper we concentrate on the correctness of the algorithm. This is not the same as proving that the algorithm is numerically stable. While we do not claim that our methodology will automatically generate stable algorithms, we do claim that the skeleton used to express the algorithm, and to implement the code, can be used to implement variants of several algorithms. These variants include those whose numerical stability properties are known. The methodology and framework also facilitate the discovery and implementation of new algorithms for which numerical properties can then be subsequently established and, thus,

to select the variant whose properties most closely match the requirements of the application under consideration.

ACKNOWLEDGMENTS

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.

Additional Information

For additional information on FLAME, ITXGEMM, PLAPACK, and POOCLAPACK, please visit

http://www.cs.utexas.edu/users/flame/

REFERENCES

Agarwal, R., Gustavson, F., and Zubair, M. 1994. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM Journal of Research and Development 38,* 5 (Sept.), 563–576.

Andersen, B. S., Gustavson, F. G., and Wasniewski, J. 2000. A recursive formalation of Cholesky factorization of a matrix in packed storage. LAPACK Working Note 146 CS-00-441, University of Tennessee, Knoxville. May.

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

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

Crout, P. D. 1941. A short method for evaluating determinants and solving systmes of linear equations with real or complex coefficients. *Trans AIEE 60,* 1235–1240.

Dijkstra, E. W. 2000. Under the spell of Leibniz's dream. Tech. Rep. EWD1298, The University of Texas at Austin. April. `http://www.cs.utexas.edu/users/EWD/`.

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

Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft. 16,* 1 (March), 1–17.

Dongarra, J. J., Du Croz, J., Hammarling, S., and Hanson, R. J. 1988. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft. 14,* 1 (March), 1–17.

Dongarra, J. J., Duff, I. S., Sorensen, D. C., and van der Vorst, H. A. 1991. *Solving Linear Systems on Vector and Shared Memory Computers.* SIAM, Philadelphia, PA.

Dongarra, J. J., Gustavson, F. G., and Karp, A. 1984. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review 26,* 1 (Jan.), 91–112.

Elmroth, E. and Gustavson, F. 2000. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM J. Res. Develop. 44,* 4, 605–624.

Golub, G. H. and Loan, C. F. V. 1989. *Matrix Computations*, 2nd ed. The Johns Hopkins University Press, Baltimore.

Gries, D. and Schneider, F. B. 1992. *A Logical Approach to Discrete Math.* Texts and Monographs in Computer Science. Springer Verlag.

Gunnels, J. 2001. A systematic approach to the design and analysis of parallel dense linear algebra algorithms. Ph.D. thesis, Department of Computer Sciences, The University of Texas.

Gunnels, J., Lin, C., Morrow, G., and van de Geijn, R. 1998. 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)*. 110–116.

Gunnels, J. A., Henry, G. M., and van de Geijn, R. A. 2001. A family of high-performance matrix multiplication algorithms. In *Computational Science - ICCS 2001, Part I*, V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner, and C. K. Tan, Eds. Lecture Notes in Computer Science 2073. Springer-Verlag, 51–60.

Gunnels, J. A. and van de Geijn, R. A. 2001a. Developing linear algebra algorithms: A collection of class projects. Tech. Rep. CS-TR-01-19, Department of Computer Sciences, The University of Texas at Austin. May. `http://www.cs.utexas.edu/users/flame/pubs.html`.

Gunnels, J. A. and van de Geijn, R. A. 2001b. Formal methods for high-performance linear algebra libraries. In *The Architecture of Scientific Software*, R. F. Boisvert and P. T. P. Tang, Eds. Kluwer Academic Press, 193–210.

Gunter, B. C., Reiley, W. C., and van de Geijn, R. A. 2001. Parallel out-of-core cholesky and qr factorizations with pooclapack. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society.

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

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

Gustavson, F. and Jonsson, I. 2000. Minimal storage high-performance Cholesky factorization via blocking and recursion. *IBM J. Res. Develop. 44,* 6 (November), 823–850.

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

Gustavson, F. G. 2001. New generalized matrix data structures lead to a variety of high-performance algorithms. In *The Architecture of Scientific Software*, R. F. Boisvert and P. T. P. Tang, Eds. Kluwer Academic Press.

Kågström, B., Ling, P., and Loan, C. V. 1998. GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. *TOMS 24,* 3, 268–302.

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

Reiley, W. C. 1999. Efficient parallel out-of-core implementation of the Cholesky factorization. Tech. Rep. CS-TR-99-33, Department of Computer Sciences, The University of Texas at Austin. Dec. Undergraduate Honors Thesis.

Reiley, W. C. and van de Geijn, R. A. 1999. POOCLAPACK: Parallel Out-of-Core Linear Algebra Package. Tech. Rep. CS-TR-99-33, Department of Computer Sciences, The University of Texas at Austin. Nov.

Smith, B. T. et al. 1976. *Matrix Eigensystem Routines – EISPACK Guide*, Second ed. Lecture Notes in Computer Science 6. Springer-Verlag, New York.

Snir, M., Otto, S. W., Huss-Lederman, S., Walker, D. W., and Dongarra, J. 1996. *MPI: The Complete Reference*. The MIT Press.

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

van de Geijn, R. A. 1997. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press.

Whaley, R. C. and Dongarra, J. J. 1998. Automatically tuned linear algebra software. In *Proceedings of SC'98*.

Received Month Year; revised Month Year; accepted Month Year