Towards the Final Generation of Dense Linear Algebra Libraries

Paolo Bientinesi* Kazushige Goto† Tze Meng Low*
Enrique S. Quintana-Ortí‡ Robert van de Geijn* Field Van Zee *

April 20, 2005

Abstract

What if one set out to develop the final dense linear algebra library? For such a library backward compatibility to existing libraries would be less important than forward compatibility to future architectures, languages, and functionality. Invariably it would have to be able to generate routines from specification rather than taking the form of the static libraries that have evolved from LINPACK. We believe that the software architecture of such a final generation library would be very different from LAPACK and ScaLAPACK.

We discuss results that suggests that such mechanical derivation of libraries is achievable, as is mechanical analysis (cost and stability). The input should be the mathematical specifications of operations, rewrite rules for translating algorithms to code, and models of target architectures. A final system should be able to mechanically generate algorithms and implementations, packaged as libraries tuned based on mechanically generated performance analyses, with mechanically generated stability analyses.

1 Introduction

At any given time some project is always pursuing the *next* dense linear algebra library. Such development tends to be justified by the needs of computational scientists who wish to use such libraries as black boxes called from application codes. The contribution to science of such packages lies primarily with the science it enables. Often there are contributions to numerical analysis from advances for individual operations and algorithms as well. Typically, the development is evolutionary: functionality is added to an existing library.

An alternative question that can be asked is how to develop the *final* dense linear algebra library. The pursuit of that question is likely to *also* yield contributions to fundamental computer science since it requires the process of developing libraries to be examined and made systematic[1]. The question is meant to focus attention on the research questions in software engineering and software architecture rather than on the actual library that results. The Formal Linear Algebra Methods Environment (FLAME) project at UT-Austin pursues this topic.

What features should one expect from a final library? The answer is that it must have the functionality, high performance, portability, and accuracy of current libraries. It also must be forward compatible to

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

†Texas Advance Computing Center, The University of Texas at Austin, Austin, TX 78712, kgoto@tacc.utexas.edu.

‡Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071 – Castellón, Spain, quintana@icc.uji.es.

[1]One definition of *science* is *knowledge that has been reduced to a system.*

future computer architectures and programming languages. More challenging is that it must be forward compatible to operations *yet to be identified by the community.* A static library (as is traditionally implemented) cannot achieve this. What is needed is a *system* that *mechanically* develops libraries.

How can a system mechanically develop high-performance, accurate libraries? We observe that a linear algebra operation is posed as a mathematical specification, an architecture can be described by a model, and a new language can be accommodated by rewrite rules that translate algorithms to code. Our conclusion is that the system needs to take mathematical specifications and architectural models as input and must produce algorithms, cost analyses, and stability analyses as output. A second (possibly separate) system can then translate algorithms to code given rewrite rules for a specified language.

The current doctrine. Let us review some of the assumptions that underlie the current doctrine by examining this following quote from a recently proposal [7], funded by NSF[2]:

> "By exploiting features of modern programming languages and making Sca/LAPACK easier to use we will be in a position to capture a new generation of users who are not interested in using Fortran 77, the current implementation language. Balanced against this are the cost in performance, memory usage or even reliability of some of these features, and the difficulty of building and maintaining one version of these very large libraries, let alone several versions in different languages. Since we do not believe that we can simultaneously maximize performance, memory efficiency, ease of use, reliability, and ease of maintenance, we have decided on the following strategy: Maintain one core version in Fortran 77, and provide wrappers in other languages just for the driver routines. Based on current user demand, these other languages will include Fortran 95 and C, as well as selected higher level languages such as Matlab, Python and Mathematica (where ultimate ease of use is possible, such as typing $x = A\backslash b$ to solve Ax = b no matter what type, mathematical properties or data structure A has). Users of the Fortran 77 version will get maximum performance and memory efficiency, but worst ease-of-use. Users of the wrappers will have better ease of use and reliability, but worse performance and memory efficiency in some cases. We, the developers, will have a tractable amount of code to maintain."

This quote makes clear that the software architecture of the next (Sca)LAPACK library will be identical to that of the original LAPACK, the software architecture of which is essentially identical to that of the 1970s package LINPACK [1, 6, 8]. While following this doctrine will satisfy the needs of the scientific computing community in the short run, it does not provide a solution to the perennial problem of having to extend and modify this library for new architectures, languages, and functionality. Furthermore, it places a heavy burden on the library developer. We believe the current doctrine cannot evolve into a final library.

This paper. In this paper we review preliminary research [12, 3, 5, 16, 18, 4] that facilitates a mechanical system that targets most of the operations supported by the BLAS, LAPACK, ScaLAPACK, as well as many operations encountered in control theory [9, 1, 6, 15, 14]. We will describe different components and insights by focusing on a concrete example, the symmetric rank-k update operation (SYRK), which is a level-3 BLAS operations [9]: $C := AA^T + C$ where C is symmetric and hence only the lower triangular part of C is stored and updated.

In Section 2, we present a methodology for systematically deriving correct algorithms. We reason that the methodology is sufficiently systematic that is can be automated, a claim supported by a prototype mechanical system coded in Mathematica [24]. In Section 3, we show how algorithms can be easily mapped to code via the introduction of appropriate Application Programming Interfaces (APIs) [5]. Examples of how new architectural features and/or language extensions can be accommodated are given in Section 4 [16, 18]. Analysis of the resulting implementations, regarding cost and numerical stability, is briefly discussed in Section 5. Related topics are mentioned in Section 6. Section 7 deals with backward compatibility to legacy libraries. Conclusions are given in the final section.

[2] In subsequent discussion we will refer to the new LAPACK project as LAPACK07.

2 Enabling science: formal derivation of algorithms

The high performance requirement inherently means that a loop-based algorithm is desired possibly combined with recursion [13]. (Details related to this claim go beyond the scope of this discussion.) For architectures with complex multi-level memories, subproblems that arise as the problem is blocked for different memory layers must often be computed with different algorithms [13, 19]. Thus, **the system must to be able to mechanically develop a family of loop-based algorithms for computing the operation from the mathematical specification.** In this section we review a systematic approach to deriving loop-based algorithms. The methodology is sufficiently systematic that it can, and has been, automated.

The FLAME approach starts by systematically deriving algorithms via an eight-step process that we next reproduce for the symmetric rank-k update operation [16, 18, 3]. We instantiate that process in the "worksheet" in Fig. 1 for a specific algorithmic variant for computing SYRK. The column marked "Step" indicates the order in which the worksheet is filled out.

Step 1: Determine the precondition and postcondition. We will let $\hat{C}$ denote the original contents of C so that upon completion C should contain $C = AA^T + \hat{C}$, which is called the *postcondition*. It describes the state of the variables upon completion of the computation. The precondition $C = \hat{C}$ and the postcondition appear in Steps 1a and 1b in Fig.1.

Step 2: Determine loop-invariants. Next, matrices are partitioned into regions:

$$C \rightarrow \left(\begin{array}{c|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array}\right) \quad \text{and} \quad A \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \tag{1}$$

where the thick lines indicate how far into the matrices the computation has reached. For SYRK it is assumed that C_{TL} is square so that both C_{TL} and C_{BR} are symmetric. Here the $\star$ indicates the symmetric part of C that is not referenced. For different operations and/or algorithmic variants operands may be partitioned differently.

Substituting the partitioned matrices into the postcondition yields

$$\left(\begin{array}{c|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array}\right) = \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right)^T + \left(\begin{array}{c|c} \hat{C}_{TL} & \star \\ \hline \hat{C}_{BL} & \hat{C}_{BR} \end{array}\right) = \left(\begin{array}{c|c} A_T A_T^T + \hat{C}_{TL} & \star \\ \hline A_B A_T^T + \hat{C}_{BL} & A_B A_B^T + \hat{C}_{BR} \end{array}\right). \tag{2}$$

This shows that $m(C_{TL})$ should equal $m(A_T)$, where $m(X)$ denotes the row dimension of matrix X, and that $\hat{C}$ should be partitioned as C.

The idea now is that (2) tells us *all* computation that must be performed in terms of the different submatrices of C and A. We wish to determine the state of matrix C at the top of a loop-body that computes the result $C = AA^T + \hat{C}$. This state is referred to as the *loop-invariant*. If the loop computes the result, not all required computation has already been performed at the top of the loop-body. This suggests that the states given in Fig. 2 can be maintained as loop-invariants: they are partial results towards the final result.

An important is that each loop-invariant has a corresponding algorithmic variant. Let us pick Loop-invariant 1 in Fig. 2:

$$\left(\begin{array}{c|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array}\right) = \left(\begin{array}{c|c} A_T A_T^T + \hat{C}_{TL} & \star \\ \hline \hat{C}_{BL} & \hat{C}_{BR} \end{array}\right).$$

The state described by the loop-invariant must hold before and after execution of an iteration of the loop. Thus, it must hold at the top and bottom of the loop-body, but also before and after the loop. This is indicated in Fig. 1 everywhere where Step 2 appears.

<table>
<tr><th>Step</th><th>Annotated Algorithm: $C := AA^T + C$</th></tr>
<tr><td>1a</td><td>$\left\{ C = \hat{C} \right\}$</td></tr>
<tr><td>4</td><td>Partition $C \rightarrow \left(\begin{array}{c|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array} \right)$, $\hat{C} \rightarrow \left(\begin{array}{c|c} \hat{C}_{TL} & \star \\ \hline \hat{C}_{BL} & \hat{C}_{BR} \end{array} \right)$,
$A \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array} \right)$
where C_{TL} is 0×0, $\hat{C}_{TL}$ is 0×0, A_T has 0 rows</td></tr>
<tr><td>2</td><td>$\left\{ \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array} \right) = \left(\begin{array}{c|c} A_T A_T^T + \hat{C}_{TL} & \star \\ \hline \hat{C}_{BL} & \hat{C}_{BR} \end{array} \right) \right\}$</td></tr>
<tr><td>3</td><td>while $m(C_{TL}) < m(C)$ do</td></tr>
<tr><td>2,3</td><td>$\left\{ \left(\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array} \right) = \left(\begin{array}{c|c} A_T A_T^T + \hat{C}_{TL} & \star \\ \hline \hat{C}_{BL} & \hat{C}_{BR} \end{array} \right) \right) \wedge (m(C_{TL}) < m(C)) \right\}$</td></tr>
<tr><td>5a</td><td>Determine block size b
Repartition
$\left(\begin{array}{c|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} C_{00} & \star & \star \\ \hline C_{10} & C_{11} & \star \\ \hline C_{20} & C_{21} & C_{22} \end{array} \right)$, $\left(\begin{array}{c} A_T \\ \hline A_B \end{array} \right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array} \right)$,
$\left(\begin{array}{c|c} \hat{C}_{TL} & \star \\ \hline \hat{C}_{BL} & \hat{C}_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} \hat{C}_{00} & \star & \star \\ \hline \hat{C}_{10} & \hat{C}_{11} & \star \\ \hline \hat{C}_{20} & \hat{C}_{21} & \hat{C}_{22} \end{array} \right)$
where C_{11} is $b \times b$, $\hat{C}_{11}$ is $b \times b$, A_1 has b rows</td></tr>
<tr><td>6</td><td>$\left\{ \left(\begin{array}{c|c|c} C_{00} & \star & \star \\ \hline C_{10} & C_{11} & \star \\ \hline C_{20} & C_{21} & C_{22} \end{array} \right) = \left(\begin{array}{c|c|c} A_0 A_0^T + \hat{C}_{00} & \star & \star \\ \hline \hat{C}_{10} & \hat{C}_{11} & \star \\ \hline \hat{C}_{20} & \hat{C}_{21} & \hat{C}_{22} \end{array} \right) \right\}$</td></tr>
<tr><td>8</td><td>$C_{10} := A_1 A_0^T + C_{10}$
$C_{11} := A_1 A_1^T + C_{11}$</td></tr>
<tr><td>5b</td><td>Continue with
$\left(\begin{array}{c|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c|c|c} C_{00} & \star & \star \\ \hline C_{10} & C_{11} & \star \\ \hline C_{20} & C_{21} & C_{22} \end{array} \right)$, $\left(\begin{array}{c} A_T \\ \hline A_B \end{array} \right) \leftarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array} \right)$
$\left(\begin{array}{c|c} \hat{C}_{TL} & \star \\ \hline \hat{C}_{BL} & \hat{C}_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c|c|c} \hat{C}_{00} & \star & \star \\ \hline \hat{C}_{10} & \hat{C}_{11} & \star \\ \hline \hat{C}_{20} & \hat{C}_{21} & \hat{C}_{22} \end{array} \right)$</td></tr>
<tr><td>7</td><td>$\left\{ \left(\begin{array}{c|c|c} C_{00} & \star & \star \\ \hline C_{10} & C_{11} & \star \\ \hline C_{20} & C_{21} & C_{22} \end{array} \right) = \left(\begin{array}{c|c|c} A_0 A_0^T + \hat{C}_{00} & \hat{C}_{01} & \hat{C}_{02} \\ \hline A_1 A_0^T + \hat{C}_{10} & A_1 A_1^T + \hat{C}_{11} & \hat{C}_{12} \\ \hline \hat{C}_{20} & \hat{C}_{21} & \hat{C}_{22} \end{array} \right) \right\}$</td></tr>
<tr><td>2</td><td>$\left\{ \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array} \right) = \left(\begin{array}{c|c} A_T A_T^T + \hat{C}_{TL} & \star \\ \hline \hat{C}_{BL} & \hat{C}_{BR} \end{array} \right) \right\}$</td></tr>
<tr><td></td><td>endwhile</td></tr>
<tr><td>2,3</td><td>$\left\{ \left(\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array} \right) = \left(\begin{array}{c|c} A_T A_T^T + \hat{C}_{TL} & \star \\ \hline \hat{C}_{BL} & \hat{C}_{BR} \end{array} \right) \right) \wedge \neg (m(C_{TL}) < m(C)) \right\}$</td></tr>
<tr><td>1b</td><td>$\left\{ C = AA^T + \hat{C} \right\}$</td></tr>
</table>

Figure 1: Worksheet for deriving the blocked algorithm for $C := AA^T + C$ (Variant 1). Grey-shaded boxes state assertions that must hold at the indicated point in the algorithm. The remaining commands are chosen to make those assertions *true*. The thick and thin lines are used to indicate movement through the matrices.

Step 3: Determine the loop-guard G. Upon completion

$$\left(\left(\begin{array}{c|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array} \right) = \left(\begin{array}{c|c} A_T A_T^T + \hat{C}_{TL} & \star \\ \hline \hat{C}_{BL} & \hat{C}_{BR} \end{array} \right) \right) \wedge \neg G$$

$\left(\begin{array}{c\|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array}\right) = \left(\begin{array}{c\|c} A_T A_T^T + \hat{C}_{TL} & \star \\ \hline \hat{C}_{BL} & \hat{C}_{BR} \end{array}\right)$ Invariant 1	$\left(\begin{array}{c\|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array}\right) = \left(\begin{array}{c\|c} A_T A_T^T + \hat{C}_{TL} & \star \\ \hline A_B A_T^T + \hat{C}_{BL} & \hat{C}_{BR} \end{array}\right)$ Invariant 2
$\left(\begin{array}{c\|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \hat{C}_{TL} & \star \\ \hline \hat{C}_{BL} & A_B A_B^T + \hat{C}_{BR} \end{array}\right)$ Invariant 3	$\left(\begin{array}{c\|c} C_{TL} & \star \\ \hline C_{BL} & C_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \hat{C}_{TL} & \star \\ \hline A_B A_T^T + \hat{C}_{BL} & A_B A_B^T + \hat{C}_{BR} \end{array}\right)$ Invariant 4

Figure 2: Loop-invariants for computing SYRK.

must imply that $C = AA^T + \hat{C}$. This suggests $G : m(C_{TL}) < m(C)$, which is then placed in Step 3.
Step 4: Determine the initialization. Before commencement of the loop, the loop-invariant must be true. This prescribes the initialization in Step 4 in Fig. 1.
Step 5: Indicate progress through the matrices. Progress is indicated by exposing submatrices to be updated and/or used, in Step 5a, and to then move the thick lines, in Step 5b. The initialization and the loop-guard prescribe the direction of the movement. The range of the movement of the thick lines determines whether the resulting algorithm is an unblocked algorithm or a blocked algorithm.
Step 6: Determine the state before the update. Exposing submatrices is an indexing operation. The state of those exposed submatrices is prescribed by the loop-invariant, and can be obtained via *textual substitution* of the exposed submatrices into the loop-invariant. This is indicated in Step 6.
Step 7: Determine the state after moving the thick lines. Similarly, moving the thick lines is an indexing operation. The state of the exposed submatrices must updated so that the loop-invariant again becomes true. Determining that state requires again textual substitution of the exposed submatrices into the loop-invariant. This is indicated in Step 7.
Step 8: Determine the update. Finally, by comparing the known state in Step 6 with the desired state in Step 7, one can deduce the update to be performed in Step 8.
The algorithm. All states in the shaded boxes in Fig. 1 only help derive, constructively, a correct algorithm. Deleting these, and temporary variables as $\hat{C}$ that are only required to prove correctness, leaves the algorithm. In Fig. 3 we show all four algorithmic variants for SYRK under the partitioning given in (1).
Note: The specification of the operation together with the partitioning of the operands (the systematic way in which the algorithm accesses the arrays) dictates reasonable loop-invariants, which in turn dictate all other steps of the derivation of the algorithm.

Remark 1 *In [4] we discuss a prototype mechanical system, coded in Mathematica, that implements the above steps. We note that the scope of this prototype system has been shown to include all of the BLAS, most of LAPACK, and it has also applied to* much *more complex problems, including the solution of the triangular Sylvester equation [19] and the generalized triangular Sylvester equation [14].* **This suggests that our approach should be able to generate algorithms for operations that have not yet been identified.**

3 Rewriting algorithms as code

Having the ability to derive correct algorithms solves only part of the problem since translating those algorithms to code traditionally requires delicate indexing into arrays, which exposes opportunities for the introduction of errors. As part of the FLAME project, we have defined APIs for Matlab's M-script language, for the C and Fortran programming languages, and for PLAPACK [5, 23]. These APIs hide intricate indices

Algorithm: $C = \text{SYRK\_BLK\_VAR1\_2}(\ A,\ C\ )$

Partition $C \rightarrow \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right)$, $A \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right)$

where C_{TL} is 0×0, A_T has 0 rows

while $m(C_{TL}) < m(C)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right), \quad \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right)$$

where C_{11} is $b \times b$, A_1 has b rows

Variant 1:	**Variant 2:**
$C_{10} := A_1 A_0^T + C_{10}$	$C_{21} := A_2 A_1^T + C_{21}$
$C_{11} := A_1 A_1^T + C_{11}$	$C_{11} := A_1 A_1^T + C_{11}$

Continue with

$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right), \quad \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \leftarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right)$$

endwhile

Algorithm: $C = \text{SYRK\_BLK\_VAR3\_4}(\ A,\ C\ )$

Partition $C \rightarrow \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right)$, $A \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right)$

where C_{BR} is 0×0, A_B has 0 rows

while $m(C_{BR}) < m(C)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right), \quad \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right)$$

where C_{11} is $b \times b$, A_1 has b rows

Variant 3:	**Variant 4:**
$C_{21} := A_2 A_1^T + C_{21}$	$C_{10} := A_1 A_0^T + C_{10}$
$C_{11} := A_1 A_1^T + C_{11}$	$C_{11} := A_1 A_1^T + C_{11}$

Continue with

$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right), \quad \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \leftarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right)$$

endwhile

Figure 3: Blocked algorithms for computing $C := AA^T + C$. The left algorithm implements Variants 1 and 2, which correspond to Loop-invariants 1 and 2 in Fig. 2. The right algorithm implements Variants 3 and 4, which correspond to Loop-invariants 3 and 4 in Fig. 2. The top algorithm sweeps through C from the top-left to the bottom-right, while the bottom algorithm accesses the matrix in the opposite direction.

using techniques similar to those used by MPI [20] and PETSc [2], allowing code to closely reflect the algorithm. In Fig. 4(a), we show an example of FLAME/C code corresponding to algorithmic Variant 2, which is shown in Fig. 4(d). To understand the code, it suffices to know that `C` and `A` are descriptors for the matrices C and A, respectively. The various routines facilitate the creation of *views* (references) into the data described by C and A. Think of a variable as `CTL` as a fancy pointer into the array C. Furthermore, the calls to `FLA_Gemm` and `FLA_Syrk` are wrappers to the BLAS calls DGEMM and DSYRK. *What is most striking about this code is the absence of intricate indexing, just as in the algorithm.*

Remark 2 *If the algorithm is stored in some meta language, rewrite rules can be defined to map the algorithm to any language. This overcomes the problem identified in LAPACK07 of having to maintain the library in multiple languages leading that project to propose to use Fortran77 for the core code base.* **This makes our approach forward compatible to languages that have not yet been conceived.**

4 Portability to different architectures

The APIs discussed above can be easily extended to target different features of current and next generation architectures. To illustrate this, we now consider SMP systems (which are often programmed by adding OpenMP directives to code) and distributed memory systems.

```c
int Syrk_blk_var2( FLA_Obj C, FLA_Obj A, int nb_alg )
{
  FLA_Obj CTL,   CTR,      C00, C01, C02,
          CBL,   CBR,      C10, C11, C12,
                           C20, C21, C22;
  FLA_Obj AT,              A0,
          AB,              A1,
                           A2;
  int b;

  FLA_Part_2x2( C,    &CTL, &CTR,
                      &CBL, &CBR,     0, 0, FLA_TL );
  FLA_Part_2x1( A,    &AT,
                      &AB,            0, FLA_TOP );

  while ( FLA_Obj_length( CTL ) < FLA_Obj_length( C ) ){
    b = min( FLA_Obj_length( CBR ), nb_alg );
    FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,   &C00, /**/ &C01, &C02,
                         /*************/  /*********************/
                                            &C10, /**/ &C11, &C12,
                           CBL, /**/ CBR,   &C20, /**/ &C21, &C22,
                           b, b, FLA_BR );
    FLA_Repart_2x1_to_3x1( AT,               &A0,
                         /* ** */           /* ** */
                                             &A1,
                           AB,               &A2,   b, FLA_BOTTOM );
    /*------------------------------------------------------------*/

    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
      ONE, A2, A1, ONE, C21 );
    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
      ONE, A1, ONE, C11 );

    /*------------------------------------------------------------*/
    FLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR,  C00, C01, /**/ C02,
                                                C10, C11, /**/ C12,
                            /**************/ /*****************/
                              &CBL, /**/ &CBR,  C20, C21, /**/ C22,
                              FLA_TL );
    FLA_Cont_with_3x1_to_2x1( &AT,                A0,
                                                  A1,
                            /* ** */            /* ** */
                              &AB,                A2,     FLA_TOP );
  }

}
```

(a) FLAME/C

```c
int OpenFLA_Syrk_blk_var2( FLA_Obj C, FLA_Obj A, int nb_alg )
{
  FLA_Obj CTL,   CTR,      C00, C01, C02,
          CBL,   CBR,      C10, C11, C12,
                           C20, C21, C22;
  FLA_Obj AT,              A0,
          AB,              A1,
                           A2;
  int b;

  FLA_Part_2x2( C,    &CTL, &CTR,
                      &CBL, &CBR,     0, 0, FLA_TL );
  FLA_Part_2x1( A,    &AT,
                      &AB,            0, FLA_TOP );
#pragma intel omp parallel taskq {
  while ( FLA_Obj_length( CTL ) < FLA_Obj_length( C ) ){
    b = min( FLA_Obj_length( CBR ), nb_alg );
    FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,   &C00, /**/ &C01, &C02,
                         /*************/  /*********************/
                                            &C10, /**/ &C11, &C12,
                           CBL, /**/ CBR,   &C20, /**/ &C21, &C22,
                           b, b, FLA_BR );
    FLA_Repart_2x1_to_3x1( AT,               &A0,
                         /* ** */           /* ** */
                                             &A1,
                           AB,               &A2,   b, FLA_BOTTOM );
    /*------------------------------------------------------------*/
#pragma intel omp task captureprivate( A0, A1, C10, C11 ){
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
      ONE, A2, A1, ONE, C21 );
    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
      ONE, A1, ONE, C11 );
} /* end task */
    /*------------------------------------------------------------*/
    FLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR,  C00, C01, /**/ C02,
                                                C10, C11, /**/ C12,
                            /**************/ /*****************/
                              &CBL, /**/ &CBR,  C20, C21, /**/ C22,
                              FLA_TL );
    FLA_Cont_with_3x1_to_2x1( &AT,                A0,
                                                  A1,
                            /* ** */            /* ** */
                              &AB,                A2,     FLA_TOP );
  }
}   /* end of taskq */
}
```

(b) OpenFLAME

```c
int PLA_Syrk_blk_var2( FLA_Obj C, FLA_Obj A, int nb_alg )
{
  PLA_Obj CTL,   CTR,      C00, C01, C02,
          CBL,   CBR,      C10, C11, C12,
                           C20, C21, C22;
  PLA_Obj AT,              A0,
          AB,              A1,
                           A2;
  int b;

  PLA_Part_2x2( C,    &CTL, &CTR,
                      &CBL, &CBR,     0, 0, PLA_TL );
  PLA_Part_2x1( A,    &AT,
                      &AB,            0, PLA_TOP );

  while ( PLA_Obj_length( CTL ) < PLA_Obj_length( C ) ){
    b = min( PLA_Obj_length( CBR ), nb_alg );

    PLA_Repart_2x2_to_3x3( CTL, /**/ CTR,   &C00, /**/ &C01, &C02,
                         /*************/  /*********************/
                                            &C10, /**/ &C11, &C12,
                           CBL, /**/ CBR,   &C20, /**/ &C21, &C22,
                           b, b, PLA_BR );
    PLA_Repart_2x1_to_3x1( AT,               &A0,
                         /* ** */           /* ** */
                                             &A1,
                           AB,               &A2,   b, PLA_BOTTOM );
    /*------------------------------------------------------------*/
    PLA_Gemm( PLA_NO_TRANSPOSE, PLA_TRANSPOSE,
      ONE, A2, A1, ONE, C21 );
    PLA_Syrk( PLA_LOWER_TRIANGULAR, PLA_NO_TRANSPOSE,
      ONE, A1, ONE, C11 );
    /*------------------------------------------------------------*/
    PLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR,  C00, C01, /**/ C02,
                                                C10, C11, /**/ C12,
                            /**************/ /*****************/
                              &CBL, /**/ &CBR,  C20, C21, /**/ C22,
                              PLA_TL );
    PLA_Cont_with_3x1_to_2x1( &AT,                A0,
                                                  A1,
                            /* ** */            /* ** */
                              &AB,                A2,     PLA_TOP );
  }
}
```

(c) PLAPACK

Algorithm: $C = \textsc{Syrk\_blk\_var2}(\ A,\ C\ )$

Partition $C \rightarrow \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right)$, $A \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right)$

where C_{TL} is 0×0, A_T has 0 rows

while $m(C_{TL}) < m(C)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right), \quad \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right)$$

where C_{11} is $b \times b$, A_1 has b rows

$C_{21} := A_2 A_1^T + C_{21}$
$C_{11} := A_1 A_1^T + C_{11}$

Continue with

$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right), \quad \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \leftarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right)$$

endwhile

(d) Algorithm

Figure 4: Various implementations of blocked Variant 2.

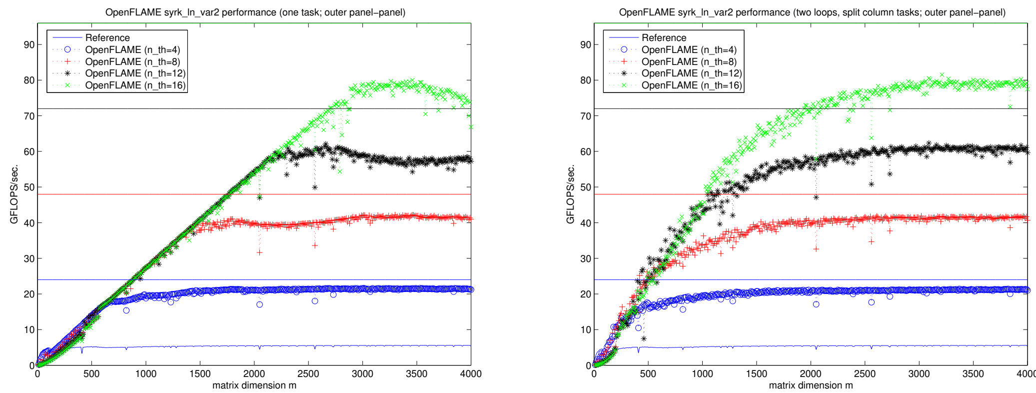

Figure 5: Performance of SMP implementations of SYRK (Variant 2). Here `n_th` indicates the number of threads used in the experiment. Left: 1D work decomposition. Right: 2D work decomposition.

SMP architectures. In Fig. 4(b) we show how the loop in Fig. 4(a) can be annotated with OpenMP directives, in this case task queue directives which have been proposed for the next OpenMP standard [21]. In that implementation, the update in the loop is taken as a single task. (In [16, 18] we discuss additional optimizations that can be equally easily coded. In that paper we also show the importance of choosing the appropriate algorithmic variant.)

Performance attained by the code in Fig. 4(b) on a 16CPU 1.5GHz Itanium2 system is reported in Fig. 5 (left). The top of each graph represents the theoretical peak of the system: 96 GFLOPS. The **`FLA_Gemm`** and **`FLA_Syrk`** calls are wrappers to sequential BLAS implementations by Kazushige Goto [11]. Impressive speedup is observed. In the graph to its right, we show how performance can be improved further for smaller problem sizes by an additional level of parallelism in the call to **`FLA_Gemm`**, creating a two-dimensional partitioning of the work, similar to the two-dimensional distributions used on distributed memory architectures.

Distributed memory architectures. In order to target distributed memory architectures the C language PLAPACK API can be extended so that PLAPACK code looks essentially identical to FLAME/C code, as illustrated in Fig. 4(c). Naturally, such code can be combined with OpenFLAME calls to perform the local computations on cluster architectures with SMP nodes.

In numerous papers we have demonstrated that PLAPACK code is competitive with ScaLAPACK implementations.

Remark 3 *The extensions that support SMP and distributed memory computing can be made part of the rewrite rules that transform algorithms into a library for a specific architecture.* **This demonstrates how the FLAME approach, in some sense, is forward compatible to next generation architectures.**

5 Mechanical analysis

We noted that the derivation process yields many algorithmic variants for a given operation. Experience and theory indicates that different algorithms are best suited for different architectures, problem sizes, and/or subproblems [19]. If new algorithms are derived (e.g., for new operations), the stability of those algorithms needs to be determined. We now briefly discuss these topics.

Cost analysis. The Mathematica systems for deriving and implementing algorithms can be retooled to also analyze their cost. By replacing operations in the body of the loop with their cost, and the loop construct by a summation, an expression for the cost of the algorithm can be derived. Mathematica can then be used to massage those summations into closed-form expressions for the cost of the algorithm. Architectural details can be added to the model to improve the accuracy since complex expressions will manipulated by Mathematica rather than by hand. We believe that the resulting analyses will be sufficiently accurate that it will be possible to mechanically resolve tradeoffs between different algorithms and implementations in order to optimize the library.

Stability analysis. As part of a dissertation by Paolo Bientinesi, a member of the FLAME team, the possibility of systematically and/or mechanically analyzing the stability of algorithms is being pursued. The ability to mechanically generate stability analyses is of great importance to our approach, since it often generates new algorithms with unknown stability properties. Early results look promising.

6 Related topics

Attaining high performance. The LAPACK07 proposal argues that the best performance is attained by codes written in Fortran77 in combination with ATLAS-like tuning. Our experience has been quite different.

The strength of the FLAME approach is that it can identify multiple algorithms for a single operation. As a result, the best algorithm can be chosen for a given situation and often different algorithms are chosen as subproblems become small enough to fit in cache memory. The reason why one can code at the level of abstraction demonstrated in Fig. 4 is that the somewhat more costly way of indexing employed by FLAME is amortized over enough computation that it does not adversely impact performance. However, for small subproblems, e.g. matrix-matrix products where one of the matrices fits in the L2 cache, it becomes necessary to call highly optimized kernels.

Our approach identifies the smallest unit of computation is for which such kernels must be called, the smallest set of such kernels is, the functionality they must support, and the interface to such low-level kernels in order to accommodate current and future architectures. This provides a clean separation of concern between algorithmic development at a high level of abstraction and kernels that support performance. It also enhances portability to new architectures. Finally, it facilitates analytic performance analysis since the performance of this small set of kernels can be reasonably modeled with high accuracy.

Alternative storage schemes. With the advent of processors with multiple layers of cache, a number of projects have started to re-examine how matrices should be stored in memory (a thorough review of these projects can be found in a recent SIAM Review paper by Elmroth et al. [10]). The primary goal is to improve performance of basic linear algebra kernels as the level-3 BLAS, a set of matrix-matrix operations that perform $O(n^3)$ computations on $O(n^2)$ data, as well as higher level linear algebra libraries such as LAPACK. The idea is that by storing blocks at different levels of granularity packed in memory, costly memory-to-memory copies and/or transpositions can be avoided. These copies are currently required to provide contiguous access to memory and/or to reduce cache and TLB misses. While conceptually the proposed solutions are simple and often elegant, complex indexing has so far prevented general acceptance. An additional complication comes from the fact that the filling of such data structures tends to put a considerable indexing burden on the application.

A simple observation underlies our approach to this: Storage by recursive blocks is typically explained as a tree structure with submatrices that are stored contiguously as leaves, and inductively as blocks of submatrices at each other level of the tree. Thus, a data structure that reflects this tree and an API that obeys this tree is the most natural way of expressing hierarchical matrices, and of manipulating such matrices.

Similarly, algorithms over these trees are expressed as recursive algorithms. An API for implementing recursive algorithms that obey this tree seems a natural solution.

The observation is that if one allows elements in a FLAME object (matrix) themselves to be objects describing matrices, then hierarchically stored matrices are naturally supported. This then allows algorithms over hierarchically stored matrices to be coded over submatrices in a style like that shown in Fig. 4. Prototype implementations of this approach show promise [17].

Interfaces to applications. A key weakness of ScaLAPACK has been that it does not have an interface that allows applications to build distributed matrices without the application programmer having to know intricate details about ScaLAPACK matrix distribution. By contrast, PLAPACK from the beginning has had an interface that allows applications to submit submatrices to a global matrix without having to know the distribution of that global matrix [23, 22]. We have demonstrated the effectiveness of a similar (prototype) interface for hierarchically stored matrices [17].

7 Backward compatibility with legacy libraries

While backward compatibility is not necessary to achieve our goal of a final generation library, it is necessary if we are to impact the computational science community. It is easy to provide an interface from the old LAPACK subroutine specifications to a library that is generated using the proposed methodology. Backward compatibility, however, comes at a price: often higher performance can be achieved simply by changing the interface slightly from the one supported by LAPACK[3].

While the same could be achieved for ScaLAPACK, we observe that only very sophisticated users employ ScaLAPACK. The effort to recode applications to call a PLAPACK-like library is minimal. Thus, being backward compatible with ScaLAPACK interface is less critical.

8 Conclusion

We have summarized a number of research projects that together provide initial evidence that large portions of commonly used dense linear algebra libraries can be mechanically derived, coded, and analyzed. It is in striving towards the ultimate goal of making the entire process mechanical that useful practical tools have been, and/or will be, developed:

- The formal derivation process makes the discovery of algorithms sufficiently systematic that it can be, and has been, applied manually by novices. This greatly reduced the time required to identify algorithms for operations while providing confidence in the correctness of the algorithms.
- The mechanical system that implements the process can be, and has been, applied to complex operations, for which the algebraic manipulations of expressions invites errors when applied by hand [4]. This further reduces the cost of developing correct algorithms.
- The APIs for representing algorithms in code can be, and have been, used to implement algorithms independently of the derivation process. Even without the formal derivation, they greatly reducing the opportunity for the introduction of indexing errors, improve readability, and reduce the cost development and maintainance of the resulting libraries.

[3]An example is the QR factorization. By allowing certain intermediate results to be passed from the QR factorization routine to the routine that subsequently solves a linear least-squares system, recomputation of those intermediate results can be avoided.

- The mechanical derivation of cost analyses will facilitate new levels of detail of performance analyses and will facilitate the investigation of strengths and weaknesses of proposed architectures. It will also facilitate the tuning of libraries.
- The mechanical derivation of stability analyses is required to provide confidence in solutions computed by the resulting libraries.

A major reason why legacy libraries are evolved into "new" libraries is that approach this preserves the investment that has been made to ensure that they are robust. The presented techniques reduce the validity of this reason, since now correct, highly efficient, and numerically stable algorithms and implementations can be systematically and/or mechanically developed.

Acknowledgments This research was partially sponsored by NSF grants ACI-0305163 and CCF-0342369. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

We gratefully acknowledge the donation of several multiple-CPU Itanium2 (1.5 GHz) servers by the Hewlett-Packard and their administration by UT-Austin's Texas Advanced Computing Center. Access to the 16 CPU Itanium2 used for the results reported in this paper was arranged by NEC Solutions (America), Inc. Additional support came from a donation by Dr. James Truchard, President, CEO, and Co-Founder of National Instruments.

References

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

[2] Satish Balay, William Gropp, Lois Curfman McInnes, and Barry Smith. PETSc 2.0 users manual. Technical Report ANL-95/11, Argonne National Laboratory, Oct. 1996.

[3] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Soft.*, 31(1):1–26, March 2005.

[4] Paolo Bientinesi, Sergey Kolos, and Robert A. van de Geijn. Automatic derivation of linear algebra algorithms with application to control theory. In *PARA04*. to appear.

[5] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Representing linear algebra algorithms in code: The FLAME APIs. *ACM Trans. Math. Soft.*, 31(1):27–59, March 2005.

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

[7] Jim Demmel and Jack Dongarra. LAPACK 2005 prospectus: Reliable and scalable software for linear algebra computations on high end computers. LAPACK Working Note 164 UT-CS-05-546, University of Tennessee, February 2005.

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

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

[10] Erik Elmroth, Fred Gustavson, Isak Jonsson, and Bo Kågström. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 46(1):3–45, 2004.

[11] Kazushige Goto. `http://www.cs.utexas.edu/users/kgoto`, 2004.

[12] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. Flame: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001.

[13] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. A family of high-performance matrix multiplication algorithms. In Vassil N. Alexandrov, Jack J. Dongarra, Benjoe A. Juliano, René S. Renner, and C.J. Kenneth Tan, editors, *Computational Science - ICCS 2001, Part I*, Lecture Notes in Computer Science 2073, pages 51–60. Springer-Verlag, 2001.

[14] Isak Jonsson and Bo Kågström. Recursive blocked algorithms for solving triangular systems: Part II: Two-sided and generalized Sylvester and Lyapunov matrix equations. *ACM Transactions on Mathematical Software*, 28(4):416–435, December 2002.

[15] Isak Jonsson and Bo Kågström. Recursive blocked algorithms for solving triangular systems u2014part i: one-sided and coupled sylvester-type matrix equations. *ACM Trans. Math. Softw.*, 28(4):392–415, 2002.

[16] Tze Meng Low, Kent Milfeld, Robert van de Geijn, and Field Van Zee. Parallelizing flame code with OpenMP task queues. *ACM Trans. Math. Soft.*, submitted.

[17] Tze Meng Low and Robert van de Geijn. An API for manipulating matrices stored by blocks. FLAPACK Working Note #12 TR-2004-15, The University of Texas at Austin, Department of Computer Sciences, May 2004.

[18] Tze Meng Low, Robert van de Geijn, and Field Van Zee. Extracting SMP parallelism for dense linear algebra algorithms from high-level specifications. In *PPoPP'05*, 2005.

[19] Enrique S. Quintana-Ortí and Robert A. van de Geijn. Formal derivation of algorithms: The triangular Sylvester equation. *ACM Transactions on Mathematical Software*, 29(2):218–243, June 2003.

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

[21] Ernesto Su, Xinmin Tian, Milind Girkar, Grant Haab, Sanjiv Shah, and Paul Peterson. Compiler support of the workqueuing execution model for Intel SMP architectures. In *EWOMP*, 2002.

[22] Robert van de Geijn. Zen and the art of high performance parallel computing. PLAPACK Tutorial available from `http://www.cs.utexas.edu/users/plapack`.

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

[24] Stephen Wolfram. *The Mathematica Book: 3rd Edition*. Cambridge University Press, 1996.