Representing Linear Algebra Algorithms in Code: The FLAME Application Program Interfaces

Paolo Bientinesi
The University of Texas at Austin
and
Enrique S. Quintana-Ortí
Universidad Jaume I
and
Robert A. van de Geijn
The University of Texas at Austin

In this paper, we present a number of Application Program Interfaces (APIs) for coding linear algebra algorithms. On the surface, these APIs for the MATLAB M-script and C programming languages appear to be simple, almost trivial, extensions of those languages. Yet with them, the task of programming and maintaining families of algorithms for a broad spectrum of linear algebra operations is greatly simplified. In combination with our Formal Linear Algebra Methods Environment (FLAME) approach to deriving such families of algorithms, dozens of algorithms for a single linear algebra operation can be derived, verified to be correct, implemented, and tested often in a matter of minutes per algorithm. Since the algorithms are expressed in code much like they are explained in a classroom setting, these APIs become not just a tool for implementing libraries, but also a valuable tool for teaching the algorithms that are incorporated in the libraries. In combination with an extension of the Parallel Linear Algebra Package (PLAPACK) API, the approach presents a migratory path from algorithm to MATLAB implementation to high-performance sequential implementation to parallel implementation. Finally, the APIs are being used to create a repository of algorithms and implementations for linear algebra operations, the FLAME Interface REpository (FIRE), which already features hundreds of algorithms for dozens of commonly encountered linear algebra operations.

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

Additional Key Words and Phrases: Application program interfaces, formal derivation, linear algebra, high-performance libraries

---

Authors' addresses: Paolo Bientinesi, Robert A. van de Geijn, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, {pauldj,rvdg}@cs.utexas.edu. Enrique S. Quintana-Ortí, Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón, Spain, quintana@icc.uji.es.

1. INTRODUCTION

The Formal Linear Algebra Methods Environment (FLAME) encompasses a methodology for deriving provably correct algorithms for dense linear algebra operations as well as an approach to representing (coding) the resulting algorithms [Gunnels et al. 2001; Quintana-Ortí and van de Geijn 2003; Bientinesi et al.]. Central to the philosophy underlying FLAME are the observations that it is at a high level of abstraction that one best reasons about the correctness of algorithms, that therefore algorithms should themselves be expressed at a high level of abstraction, and that codes that implement such algorithms should themselves use an API that captures this high level of abstraction. A key observation is that in reasoning about algorithms intricate indexing is typically avoided and it is with the introduction of complex indexing that programming errors are often introduced and confidence in code is diminished. Thus a carefully designed API should avoid explicit indexing whenever possible. In this paper we give such APIs for the MATLAB M-script and C programming languages [Moler et al. 1987; Kernighan and Ritchie 1978]. We also show the resulting MATLAB and C implementations to be part of a natural migratory path towards high-performance parallel implementation.

Our FLAME@lab, FLAME/C and FLAME/PLAPACK interfaces strive to allow algorithms to be presented in code so that the knowledge expressed in the algorithms is also expressed in the code. In particular, this knowledge is not obscured by intricate indexing. In a typical development, an initial FLAME@lab implementation gives the user the flexibility of MATLAB to test the algorithms designed using FLAME before going to a high-performance sequential implementation using the FLAME/C API, and the subsequent parallel implementation using the Parallel Linear Algebra Package (PLAPACK) [van de Geijn 1997; Baker et al. 1998; Alpatov et al. 1997]. In our experience, an inexperienced user can use these different interfaces to develop and test MATLAB and high-performance C implementations of an algorithm in less than an hour. An experienced user can perform this task in a matter of minutes, and can in addition implement a scalable parallel implementation in less than a day. This represents a significant reduction in effort relative to more traditional approaches to such library development [Anderson et al. 1992; Choi et al. 1992].

The FLAME approach to deriving algorithms often yields a large number of algorithms for a given linear algebra operation. Since the APIs given in this paper allow these algorithms to be easily captured in code, they enable the systematic creation of a repository of algorithms and their implementations. As part of our work, we have started to assemble such a repository, the FLAME Interface Repository (FIREsite).

This paper is organized as follows: In Section 2, we present an example of how we represent a broad class of linear algebra algorithms in our previous papers. The most important components of the FLAME@lab API are presented in Section 3. The FLAME/C API is given in Section 4. A discussion of how the developed algorithms, coded using the FLAME/C API, can be migrated to parallel code written in C is discussed in Section 5. Performance issues are discussed in Section 6. We discuss productivity issues and FIREsite in section 7. A few concluding remarks are given in Section 8. In the appendix, a listing of the most commonly used

Partition $B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$ and $L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$
where B_T has 0 rows and L_{TL} is 0×0
while $\mathrm{m}(L_{TL}) \neq \mathrm{m}(L)$ **do**
Repartition

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline b_1^T \\ \hline B_2 \end{array}\right) \text{ and } \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right)$$

where b_1^T is a row and λ_{11} is a scalar

$b_1^T := b_1^T - l_{10}^T B_0$
$b_1^T := \lambda_{11}^{-1} b_1^T$

Continue with

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline b_1^T \\ \hline\hline B_2 \end{array}\right) \text{ and } \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline\hline L_{20} & l_{21} & L_{22} \end{array}\right)$$

enddo

Fig. 1. Unblocked algorithm for the TRSM example.

FLAME/C routines is given, as is a discussion regarding how to interface more traditional code with FLAME/C.

Notice that there are some repeated comments in Sections 3 and 4. Thus a reader can choose to skip the discussion of the FLAME@lab API in Section 3 or the FLAME/C API in Section 4 while fully benefiting from the insights in those sections.

2. A TYPICAL DENSE LINEAR ALGEBRA ALGORITHM

In [Bientinesi et al.] we introduced a methodology for the systematic derivation of provably correct algorithms for dense linear algebra algorithms. It is highly recommended that the reader become familiar with that paper before proceeding with the remainder of this paper. This section gives the minimal background in an attempt to make the present paper self-contained.

The algorithms that result from the derivation process present themselves in a very rigid format. We illustrate this format in Fig. 1, which gives an (unblocked) algorithm for the computation of $B := L^{-1}B$, where B is an $m \times n$ matrix and L is an $m \times m$ lower triangular matrix. This operation is often referred to as a triangular solve with multiple right-hand sides (TRSM). Notice that the presented algorithm was derived in [Bientinesi et al.].

At the top of the loop-body, it is assumed that different regions of the operands L and B have been used and/or updated in a consistent fashion. These regions are initialized by

Partition $B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$ and $L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$
where B_T has 0 rows and L_{TL} is 0×0

Here T, B, L, and R stand for Top, Bottom, Left, and Right, respectively.

NOTE 1. *Of particular importance in the algorithm are the single and double lines used to partition and repartition the matrices. Double lines are used to demark regions in the matrices that have been used and/or updated in a consistent fashion. Another way of interpreting double lines is that they keep track of how far into the matrices the computation has progressed.*

Let $\hat{B}$ equal the original contents of B and assume that $\hat{B}$ is partitioned like B. At the top of the loop it will be assumed that B_B contains the original contents $\hat{B}_B$ while B_T has been updated with the contents $L_{TL}^{-1}\hat{B}_T$. As part of the loop, the boundaries between these regions are moved one row and/or column at a time so that progress towards completion is made. This is accomplished by

Repartition

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline b_1^T \\ \hline B_2 \end{array}\right) \text{ and } \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right)$$

where b_1^T is a row and λ_{11} is a scalar

$\vdots$

Continue with

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline b_1^T \\ \hline\hline B_2 \end{array}\right) \text{ and } \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline\hline L_{20} & l_{21} & L_{22} \end{array}\right)$$

NOTE 2. *Single lines are introduced in addition to the double lines to demark regions that are involved in the update or used in the current step of the algorithm. Upon completion of the update, the regions defined by the double lines are updated to reflect that the computation has moved forward.*

NOTE 3. *We adopt the often-used convention where matrices, vectors, and scalars are denoted by upper-case, lower-case, and Greek letters, respectively [Stewart 1973].*

NOTE 4. *A row vector is indicated by adding a transpose to a vector, e.g.,* b_1^T *and* l_{10}^T.

The repartitioning exposes submatrices that must be updated before the boundaries can be moved. That update is given by

$$\begin{array}{l} b_1^T := b_1^T - l_{10}^T B_0 \\ b_1^T := \lambda_{11}^{-1} b_1^T \end{array}$$

Finally, the desired result has been computed when L_{TL} encompasses all of L so that the loop continues until $\mathrm{m}(L_{TL}) \neq \mathrm{m}(L)$ becomes *false*. Here $m(X)$ returns the row dimension of matrix X.

NOTE 5. *We would like to claim that the algorithm in Fig. 1 captures how one might naturally explain a particular algorithmic variant for computing the solution of a triangular linear system with multiple right-hand sides.*

Partition $L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$ and $B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$
  where B_T has 0 rows and L_{TL} is 0×0
while $\mathrm{m}(L_{TL}) \neq \mathrm{m}(L)$ **do**
  Determine block size b
  Repartition

$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right) \text{ and } \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline B_1 \\ \hline B_2 \end{array}\right)$$

  where $\mathrm{m}(B_1) = b$ and $\mathrm{m}(L_{11}) = \mathrm{n}(L_{11}) = b$

$B_1 := B_1 - L_{10} B_0$
$B_1 := L_{11}^{-1} B_1$

  Continue with

$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline\hline L_{20} & L_{21} & L_{22} \end{array}\right) \text{ and } \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline\hline B_2 \end{array}\right)$$

enddo

Fig. 2. Blocked algorithm for the TRSM example.

```
[ m, n ] = size( B );
for i=1:mb:m
  b = min( mb, m-i+1 );
  B( i:i+b-1, : ) = B( i:i+b-1, : ) - ...
     L( i:i+b-1, 1:i-1 ) * B( 1:i-1, : );
  B( i:i+b-1, : ) = ...
     inv( tril( L( i:i+b-1, i:i+b-1 ) ) ) * B( i:i+b-1, : );
end
```

Fig. 3. MATLAB implementation of blocked TRSM algorithm in Fig. 2.

NOTE 6. *The presented algorithm only requires one to use indices from the sets* $\{T, B\}$, $\{L, R\}$, *and* $\{0, 1, 2\}$.

For performance reasons, it is often necessary to formulate the algorithm as a *blocked* algorithm as illustrated in Fig. 2. The performance benefit comes from the fact that the algorithm is rich in matrix multiplication which allows processors with multi-level memories to achieve high performance [Dongarra et al. 1991; Anderson et al. 1992; Gunnels et al. 2001; Dongarra et al. 1990].

NOTE 7. *The algorithm in Fig. 2 is implemented by the more traditional MATLAB code given in Fig. 3*[1]. *We claim that the introduction of indices to explicitly indicate the regions involved in the update complicates readability and reduces confidence in the correctness of the MATLAB implementation. Indeed, an explanation*

[1] We realize that the use of `inv( tril( L( i:i+mb-1, i:i+mb-1 ) ) )` can introduce numerical instability and that therefore one in practice would code this as the solution of a triangular system with multiple right-hand sides.

of the code will inherently require the drawing of a picture that captures the repartitioned matrices in Fig. 2. In other words, someone experienced with MATLAB can easily translate the algorithm in Fig. 2 into the implementation in Fig. 3. The converse is considerably more difficult.

3. THE FLAME@LAB INTERFACE FOR MATLAB

In this section we introduce a set of MATLAB M-script functions that will allow us to capture in code the linear algebra algorithms presented in the format illustrated in the previous section. The idea is that by making the code look like the algorithms in Figs. 1 and 2 the opportunity for the introduction of coding errors is reduced while simultaneously making the code more readable.

3.1 Bidimensional partitionings

As illustrated in Figs. 1 and 2, in stating a linear algebra algorithm one may wish to partition a matrix like

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

In the FLAME@lab API, we hide complicated indexing by using MATLAB matrices. Given a MATLAB matrix A, the following call creates one matrix for each of the four quadrants:

```
[ ATL, ATR,...
  ABL, ABR     ] = FLA_Part_2x2( A,...
                                 mb, nb, quadrant )
```

Purpose: Partition matrix A into four quadrants where the quadrant indicated by `quadrant` is `mb` × `nb`.

Here `quadrant` is a MATLAB string that can take on the values `'FLA_TL'`, `'FLA_TR'`, `'FLA_BL'`, and `'FLA_BR'` to indicate that `mb` and `nb` are the dimensions of the Top-Left, Top-Right, Bottom-Left, or Bottom-Right quadrant, respectively.

NOTE 8. *Invocation of the operation*

```
[ ATL, ATR,...
  ABL, ABR     ] = FLA_Part_2x2( A,...
                                 mb, nb, 'FLA_TL' )
```

in MATLAB creates four new matrices, one for each quadrant. Subsequent modifications of the contents of a quadrant do not affect therefore the original contents of the matrix. This is an important difference to consider with respect to the FLAME/C API introduced in Section 4, where the quadrants are views (references) into the original matrix, not copies of it!

As an example of the use of this routine, the translation of the algorithm fragment on the left results in the code on the right

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 $m_b \times n_b$

```
[ ATL, ATR,...
  ABL, ABR     ] = FLA_Part_2x2( A,...
                                 mb, nb, ...
                                 'FLA_TL' )
```

where the parameters `mb` and `nb` have values m_b and n_b, respectively. Examples of the use of this routine can also be found in Figs. 4 and 5.

NOTE 9. *The above example stresses the fact that the formatting of the code can be used to help capture the algorithm in code. Clearly, some of the benefit of the API would be lost if in the example the code appeared as*

```
[ ATL, ATR, ABL, ABR  ] = FLA_Part_2x2( A, mb, nb, 'FLA_TL' )
```

Also from Figs. 1 and 2, we notice that it is useful to be able to take a 2×2 partitioning of a given matrix A and repartition that into a 3×3 partitioning so that the submatrices that need to be updated and/or used for computation can be identified. To support this, we introduce the call

```
[ A00, A01, A02,...
  A10, A11, A12,...
  A20, A21, A22     ] =  FLA_Repart_2x2_to_3x3( ATL, ATR,...
                                                ABL, ABR,...
                                                mb, nb, quadrant )
```

Purpose: Repartition a 2×2 partitioning of matrix `A` into a 3×3 partitioning where the `mb` $\times$ `nb` submatrix `A11` is split from the quadrant indicated by `quadrant`.

Here `quadrant` can again take on the values `'FLA_TL'`, `'FLA_TR'`, `'FLA_BL'`, and `'FLA_BR'` to indicate that the `mb` $\times$ `nb` submatrix `A11` is split from submatrix `ATL`, `ATR`, `ABL`, or `ABR`, respectively.

Thus,

Repartition

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

where A_{11} is $m_b \times n_b$

translates to the code

```
[ A00, A01, A02,...
  A10, A11, A12,...
  A20, A21, A22     ] =  FLA_Repart_2x2_to_3x3( ATL, ATR,...
                                                ABL, ABR,...
                                                mb, nb, 'FLA_BR' )
```

where the parameters `mb` and `nb` have values m_b and n_b, respectively. Other examples of the use of this routine can also be found in Figs. 4 and 5.

NOTE 10. *Similarly to what is expressed in Note 8, the invocation of the operation*

```
[ A00, A01, A02,...
  A10, A11, A12,...
  A20, A21, A22     ] =  FLA_Repart_2x2_to_3x3( ... )
```

creates nine new matrices and any modification of the contents of `A00, A01, A02, ...`, *does not affect the original matrix* `A` *nor the four quadrants* `ATL, ATR, ABL, ABR`.

NOTE 11. *Choosing variable names can further relate the code to the algorithm, as is illustrated by comparing*

$$\left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right) \quad \textit{and} \quad \begin{array}{lll} \texttt{L00,} & \texttt{l01,} & \texttt{L02,} \\ \texttt{l10t,} & \texttt{lambda11,} & \texttt{l12t,} \\ \texttt{L20,} & \texttt{l21,} & \texttt{L22,} \end{array}$$

in Figs. 1 and 4. Notice here that although in the algorithm certain regions are identified as containing only zeroes, variables are needed to store those regions in the partitioning.

Once the contents of the so-identified submatrices have been updated, the contents of `ATL`, `ATR`, `ABL`, and `ABR` must be updated to reflect that progress is being made, in terms of the regions indicated by the double-lines. This moving of the double-lines is accomplished by a call to

```
[ ATL, ATR,...
  ABL, ABR     ] = FLA_Cont_with_3x3_to_2x2( A00, A01, A02,...
                                             A10, A11, A12,...
                                             A20, A21, A22,...
                                             quadrant )
```

Purpose: Update the 2×2 partitioning of matrix `A` by moving the boundaries so that `A11` is added to the quadrant indicated by `quadrant`.

This time the value of `quadrant` (`'FLA_TL'`, `'FLA_TR'`, `'FLA_BL'`, or `'FLA_BR'`) indicates to which quadrant the submatrix `A11` is to be added.

For example,

Continue with

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

translates to the code

```
[ ATL, ATR,...
  ABL, ABR     ] = FLA_Cont_with_3x3_to_2x2( A00, A01, A02,...
                                             A10, A11, A12,...
                                             A20, A21, A22,...
                                             'FLA_TL' )
```

Further examples of the use of this routine can again be found in Figs. 4 and 5.

```
function [ X ] = Trsm_llnn_unb_var1( L, B )

  [ LTL, LTR,...
    LBL, LBR     ] = FLA_Part_2x2( L,...
                                   0, 0, 'FLA_TL' );
  [ BT,...
    BB     ] = FLA_Part_2x1( B,...
                             0, 'FLA_TOP' );

  while( size( LTL, 1 ) ~= size( L, 1 ) )
    [ L00,  l01,      L02,...
      l10t, lambda11, l12t,...
      L20,  l21,      L22      ] = FLA_Repart_2x2_to_3x3( LTL, LTR,...
                                                          LBL, LBR,...
                                                          1, 1, 'FLA_BR' );
    [ B0,...
      b1t,...
      B2      ] = FLA_Repart_2x1_to_3x1( BT,...
                                         BB,...
                                         1, 'FLA_BOTTOM' );
%* ------------------------------------------------------------------------ */
    b1t = b1t - l10t * B0;
    b1t = inv( lambda11 ) * b1t;
%* ------------------------------------------------------------------------ */
    [ LTL, LTR,...
      LBL, LBR     ] = FLA_Cont_with_3x3_to_2x2( L00,  l01,      L02,...
                                                 l10t, lambda11, l12t,...
                                                 L20,  l21,      L22,...
                                                 'FLA_TL' );
    [ BT,...
      BB     ] = FLA_Cont_with_3x1_to_2x1( B0,...
                                           b1t,...
                                           B2,...
                                           'FLA_TOP' );
  end

  X = BT;
  return;
```

Fig. 4. FLAME implementation of unblocked TRSM algorithm in Fig. 1 using the FLAME@lab interface.

3.2 Horizontal partitionings

Similar to the partitioning into quadrants discussed above, and as illustrated in Figs. 1 and 2, in stating a linear algebra algorithm one may wish to partition a matrix like

$$\textbf{Partition} \quad A \rightarrow \left(\frac{A_T}{\overline{A_B}} \right) \quad \textbf{where} \quad A_T \text{ has } k \text{ rows}$$

For this, we introduce the call

```
function [ X ] = Trsm_llnn_blk_var1( L, B, mb )

  [ LTL, LTR,...
    LBL, LBR     ] = FLA_Part_2x2( L,...
                                   0, 0, 'FLA_TL' );
  [ BT,...
    BB     ] = FLA_Part_2x1( B,...
                             0, 'FLA_TOP' );

  while( size( LTL, 1 ) ~= size( L, 1 ) )
    b = min( mb, size( LBR, 1 ) );

    [ L00, L01, L02,...
      L10, L11, L12,...
      L20, L21, L22     ] = FLA_Repart_2x2_to_3x3( LTL, LTR,...
                                                   LBL, LBR,...
                                                   b, b, 'FLA_BR' );
    [ B0,...
      B1,...
      B2     ] = FLA_Repart_2x1_to_3x1( BT,...
                                        BB,...
                                        b, 'FLA_BOTTOM');
%* ---------------------------------------------------------------------- */
    B1 = B1 - L10 * B0;
    B1 = Trsm_llnn_unb_var1( L11, B1 );
%* ---------------------------------------------------------------------- */
    [ LTL, LTR,...
      LBL, LBR     ] = FLA_Cont_with_3x3_to_2x2( L00, L01, L02,...
                                                 L10, L11, L12,...
                                                 L20, L21, L22,...
                                                 'FLA_TL' );
    [ BT,...
      BB     ] = FLA_Cont_with_3x1_to_2x1( B0,...
                                           B1,...
                                           B2,...
                                           'FLA_TOP' );
  end

  X = BT;
  return;
```

Fig. 5. FLAME implementation of blocked TRSM algorithm in Fig. 2 using the FLAME@lab interface.

```
[ AT,...
  AB     ] = FLA_Part_2x1( A,...
                           mb, side )
```

Purpose: Partition matrix `A` into a top and a bottom side where the side indicated by `side` has `mb` rows.

Here `side` can take on the values `'FLA_TOP'` or `'FLA_BOTTOM'` to indicate that `mb` is the row dimension of `AT` or `AB`, respectively.

Given that matrix `A` is already partitioned horizontally it can be repartitioned into three submatrices with the call

```
[ A0,...
  A1,...
  A2     ] = FLA_Repart_2x1_to_3x1( AT,...
                                    AB,...
                                    mb, side )
```

Purpose: Repartition a 2×1 partitioning of matrix `A` into a 3×1 partitioning where submatrix `A1` with `mb` rows is split from the side indicated by `side`.

Here `side` can take on the values `'FLA_TOP'` or `'FLA_BOTTOM'` to indicate that submatrix `A1`, with `mb` rows, is partitioned from `AT` or `AB`, respectively.

Given a 3×1 partitioning of a given matrix `A`, the middle submatrix can be appended to either the first or last submatrix with the call

```
[ AT,...
  AB     ] = FLA_Cont_with_3x1_to_2x1( A0,...
                                       A1,...
                                       A2,...
                                       side )
```

Purpose: Update the 2×1 partitioning of matrix `A` by moving the boundaries so that `A1` is added to the side indicated by `side`.

Examples of the use of the routines that deals with the horizontal partitioning of matrices can be found in Figs. 4 and 5.

3.3 Vertical partitionings

Finally, in stating a linear algebra algorithm one may wish to partition a matrix like

Partition $A \rightarrow \left(A_L \,\middle\|\, A_R \right)$
where A_L has k columns

For this we introduce the call

```
 [ AL, AR ] = FLA_Part_1x2( A,...
                            int nb, int side )
```

Purpose: Partition matrix `A` into a left and a right side where the side indicated by `side` has `nb` columns.

and

```
[ A0, A1, A2 ] = FLA_Repart_1x2_to_1x3( AL, AR,...
                                        nb, side )
```

Purpose: Repartition a 1×2 partitioning of matrix `A` into a 1×3 partitioning where submatrix `A1` with `nb` columns is split from the side indicated by `side`.

Here `side` can take on the values `'FLA_LEFT'` or `'FLA_RIGHT'`. Adding the middle submatrix to the first or last submatrix is now accomplished by a call to

```
[ AL, AR ] = FLA_Cont_with_1x3_to_1x2( A0, A1, A2,...
                                        side )
```

Purpose: Update the 1×2 partitioning of matrix `A` by moving the boundaries so that `A1` is added to the side indicated by `side`.

3.4 Additional routines

NOTE 12. *Interestingly enough, the routines described in this section for the MATLAB M-script language suffice to implement a broad range of algorithms encountered in dense linear algebra. So far, we have yet to encounter algorithms that cannot be elegantly described by partitioning into regions than can be indexed by the sets* $\{T, B\}$, $\{L, R\}$, $\{0, 1, 2\}$, $\{T, B\} \times \{L, R\}$, *and* $\{0, 1, 2\} \times \{0, 1, 2\}$. *Also, MATLAB provides a rich set of operations on matrices and vectors, which are needed to implement the updates to the exposed submatrices.*

4. THE FLAME/C INTERFACE FOR THE C PROGRAMMING LANGUAGE

It is easily recognized that the FLAME@lab codes given in the previous section will likely fall short of attaining peak performance. In particular, the copying that inherently occurs when submatrices are created and manipulated represents pure overhead. But then, generally people do not use MATLAB if they insist on attaining high performance. For that, they tend to code in C and link to high-performance libraries like the Basic Linear Algebra Subprograms (BLAS) and the Linear Algebra Package (LAPACK) [Anderson et al. 1992; Lawson et al. 1979; Dongarra et al. 1988; Dongarra et al. 1990]. In this section we introduce a set of library routines that will allow us to capture in C code linear algebra algorithms presented in the format given in Section 2.

Again, the idea is that by making C code look like the algorithms in Figs. 1 and 2 the opportunity for the introduction of coding errors is reduced. Readers familiar with MPI [Gropp et al. 1994; Snir et al. 1996], PETSc [Balay et al. 1996], and/or our own PLAPACK will recognize the programming style, *object-based programming*, as being very similar to that used by those (and other) interfaces. It is this style of programming that will allow us to hide the indexing details much like MATLAB does. However, as we will see, a more substantial infrastructure must be provided in addition to the routines that partition and repartition matrix objects.

4.1 Initializing and finalizing FLAME/C

Before using the FLAME/C environment one must initialize with a call to

```
void FLA_Init( )
```

Purpose: Initialize FLAME/C.

If no more FLAME/C calls are to be made, the environment is exited by calling

```
void FLA_Finalize( )
```

Purpose: Finalize FLAME/C.

4.2 Linear algebra objects

Notice that the following attributes describe a matrix as it is stored in the memory of a computer:

—the datatype of the entries in the matrix, e.g., `double` or `float`,
—m and n, the row and column dimensions of the matrix,
—the address where the data is stored, and
—the mapping that describes how the two dimensional array is mapped to one dimensional memory.

The following call creates an object (*descriptor* or *handle*) of type `FLA_Obj` for a matrix and creates space to store the entries in the matrix:

```
void FLA_Obj_create( int datatype, int m, int n, FLA_Obj *matrix )
```

Purpose: Create an object that describes an `m` × `n` matrix and create the associated storage array.

Valid datatype values include

`FLA_INT`, `FLA_DOUBLE`, `FLA_FLOAT`, `FLA_DOUBLE_COMPLEX`, and `FLA_COMPLEX`

for the obvious datatypes that are commonly encountered. The leading dimension of the array that is used to store the matrix is itself determined inside of this call.

NOTE 13. *For simplicity, we chose to limit the storage of matrices to use column-major storage. The leading dimension of a matrix can be thought of as the dimension of the array in which the matrix is embedded (which is often larger than the row-dimension of the matrix) or as the increment (in elements) required to address consecutive elements in a row of the matrix. Column-major storage is chosen to be consistent with Fortran which is often still the choice of language for linear algebra applications.*

FLAME/C treats vectors as special cases of matrices: an $n \times 1$ matrix or a $1 \times n$ matrix. Thus, to create an object for a vector x of n double-precision real numbers either of the following calls will suffice:

```
FLA_Obj_create( FLA_DOUBLE, n, 1, &x );
FLA_Obj_create( FLA_DOUBLE, 1, n, &x );
```

Here `n` is an integer variable with value n and `x` is an object of type `FLA_Obj`.

Similarly, FLAME/C treats scalars as a 1×1 matrix. Thus, to create a object for a scalar α the following call is made:

```
FLA_Obj_create( FLA_DOUBLE, 1, 1, &alpha );
```

where `alpha` is an object of type `FLA_Obj`. A number of scalars occur frequently and are therefore predefined by FLAME/C:

`MINUS_ONE`, `ZERO`, and `ONE`.

If an object was created with `FLA_Obj_create` (or `FLA_Obj_create_conf_to`, given in the Appendix), a call to `FLA_Obj_free` is required to ensure that all space associated with the object is properly released:

```
void FLA_Obj_free( FLA_Obj *matrix )
```

Purpose: Free all space allocated to store data associated with `matrix`.

4.3 Inquiry routines

In order to be able to work with the raw data, a number of inquiry routines can be used to access information about a matrix (or vector or scalar). The datatype and row and column dimensions of the matrix can be extracted by calling

```
int FLA_Obj_datatype( FLA_Obj matrix )
int FLA_Obj_length  ( FLA_Obj matrix )
int FLA_Obj_width   ( FLA_Obj matrix )
```

Purpose: Extract datatype, row, or column dimension of `matrix`, respectively.

The address of the array that stores the matrix and its leading dimension can be retrieved by calling

```
void *FLA_Obj_buffer( FLA_Obj matrix )
int FLA_Obj_ldim    ( FLA_Obj matrix )
```

Purpose: Extract address and leading dimension of `matrix`, respectively.

4.4 A most useful utility routine

Our approach to the implementation of algorithms for linear algebra operations starts with the careful derivation of provably correct algorithms. The stated philosophy is that if the algorithms are correct, and the API allows the algorithms to be coded so that the code reflects the algorithms, then the code will be correct as well.

Nonetheless, we single out one of the more useful routines in the FLAME/C library, which is particularly helpful for testing:

```
void FLA_Obj_show( char *string1, FLA_Obj A, char *format,
                   char *string2 )
```

Purpose: Print the contents of `A`.

In particular, the result of

```
FLA_Obj_show( "A =[", A, "%lf", "];" );
```

is something like

```
A = [
< entries_of_A >
];
```

which can then be fed to MATLAB. This becomes useful when checking results against a MATLAB implementation of an operation.

4.5 An example: matrix-vector multiplication

We now give an example of how to use the calls introduced so far to write a simple driver routine that calls a routine that performs the matrix-vector multiplication $y = Ax$.

```
1   #include "FLAME.h"
2
3   main()
4   {
5     FLA_Obj
6       A, x, y;
7     int
8       m, n;
9
10    FLA_Init( );
11
12    printf( "enter matrix dimensions m and n:" );
13    scanf( "%d%d", &m, &n );
14
15    FLA_Obj_create( FLA_DOUBLE, m, n, &A );
16    FLA_Obj_create( FLA_DOUBLE, m, 1, &y );
17    FLA_Obj_create( FLA_DOUBLE, n, 1, &x );
18
19    fill_matrix( A );
20    fill_matrix( x );
21
22    mv_mult( A, x, y );
23
24    FLA_Obj_show( "A = [", A, "%lf", "]" );
25    FLA_Obj_show( "x = [", x, "%lf", "]" );
26    FLA_Obj_show( "y = [", y, "%lf", "]" );
27
28    FLA_Obj_free( &A );
29    FLA_Obj_free( &y );
30    FLA_Obj_free( &x );
31
32    FLA_Finalize( );
33  }
```

Fig. 6. A simple C driver for matrix-vector multiplication.

In Fig. 6 we give the driver routine.

—**line 1:** FLAME/C program files start by including the `FLAME.h` header file.

—**line 5–6:** FLAME/C objects `A`, `x`, and `y`, which will hold matrix A and vectors x and y, are declared to be of type `FLA_Obj`.

—**line 10:** Before any calls to FLAME/C routines can be made, the environment must be initialized by a call to `FLA_Init`.

—**line 12–13:** In our example, the user inputs the row and column dimension of matrix A.

—**line 15–17:** Descriptors are created for A, x, and y.

—**line 19–20:** The routine in Fig. 7, described below, is used to fill A and x with values.

—**line 22:** Compute $y = Ax$ using the routine for performing that operation given in Fig. 8.

—**line 24–26:** Print out the contents of A, x, and y.

```
1  #include "FLAME.h"
2
3  #define BUFFER( i, j ) buff[ (j)*lda + (i) ]
4
5  void fill_matrix( FLA_Obj A )
6  {
7    int
8      datatype, m, n, lda;
9
10   datatype = FLA_Obj_datatype( A );
11   m        = FLA_Obj_length( A );
12   n        = FLA_Obj_width ( A );
13   lda      = FLA_Obj_ldim  ( A );
14
15   if ( datatype == FLA_DOUBLE ){
16     double *buff;
17     int    i, j;
18
19     buff = ( double * ) FLA_Obj_buffer( A );
20
21     for ( j=0; j<n; j++ )
22       for ( i=0; i<m; i++ )
23         BUFFER( i, j ) = i+j*0.01;
24   }
25   else FLA_Abort( "Datatype not yet supported", __LINE__, __FILE__ );
26 }
```

Fig. 7. A simple routine for filling a matrix.

—**line 28–30:** Free the created objects.

—**line 32:** Finalize FLAME/C.

A sample routine for filling `A` and `x` with data is given in Fig. 7. Notice that the macro definition in line 3 is used to access the matrix A stored in array `A` using column-major ordering.

The routine in Fig. 8 is itself a wrapper to the level 2 BLAS routine `cblas_dgemv`, a commonly available kernel for computing a matrix-vector multiplication. Notice that in order to call this routine, which requires parameters describing the matrix, vectors, and scalars to be explicitly passed, all of the inquiry routines are required.

4.6 Views

Figs. 1 and 2 illustrate the need for partitionings like

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

In C we avoid complicate indexing by introducing the notion of a *view*, which is a **reference** into an existing matrix or vector. Given a descriptor `A` of a matrix A, the following call creates descriptors of the four quadrants:

```
#include "FLAME.h"
#include "cblas.h"

void mv_mult( FLA_Obj A, FLA_Obj x, FLA_Obj y )
{
  int
    datatype_A,    m_A, n_A, ldim_A,    m_x, n_y, inc_x,    m_y, n_y, inc_y;

  datatype_A = FLA_Obj_datatype( A );
  m_A        = FLA_Obj_length( A );
  n_A        = FLA_Obj_width ( A );
  ldim_A     = FLA_Obj_ldim  ( A );

  m_x        = FLA_Obj_length( x );       m_y        = FLA_Obj_length( y );
  n_x        = FLA_Obj_width ( x );       n_y        = FLA_Obj_width ( y );


  if ( m_x == 1 ) {
    m_x  = n_x;
    inc_x = FLA_Obj_ldim( x );
  }
  else inc_x = 1;

  if ( m_y == 1 ) {
    m_y  = n_y;
    inc_y = FLA_Obj_ldim( y );
  }
  else inc_y = 1;

  if ( datatype_A == FLA_DOUBLE ){
    double *buff_A, *buff_x, *buff_y;

    buff_A = ( double * ) FLA_Obj_buffer( A );
    buff_x = ( double * ) FLA_Obj_buffer( x );
    buff_y = ( double * ) FLA_Obj_buffer( y );

    cblas_dgemv( CblasColMaj, CblasNoTrans,
                 1.0, buff_A, ldim_A, buff_x, inc_x,
                 1.0, buff_y, inc_y );
  }
  else FLA_Abort( "Datatype not yet supported", __LINE__, __FILE__ );
}
```

Fig. 8. A simple matrix-vector multiplication routine. This routine is implemented as a wrapper to the BLAS routine `cblas_dgemv` for matrix-vector multiplications.

```
void FLA_Part_2x2( FLA_Obj A, FLA_Obj *ATL, FLA_Obj *ATR,
                              FLA_Obj *ABL, FLA_Obj *ABR,
                   int mb, int nb, int quadrant )
```

Purpose: Partition matrix `A` into four quadrants where the quadrant indicated by `quadrant` is `mb` $\times$ `nb`.

Here `quadrant` can take on the values `FLA_TL`, `FLA_TR`, `FLA_BL`, and `FLA_BR` (defined in `FLAME.h`) to indicate that `mb` and `nb` specify the dimensions of the Top-Left, Top-

Right, Bottom-Left, or Bottom-Right quadrant, respectively. Thus, the algorithm fragment on the left is translated into the code on the right

$$\textbf{Partition } A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \quad \textbf{where } A_{TL} \text{ is } m_b \times n_b$$

```
FLA_Part_2x2( A, &ATL, /**/ &ATR,
                 /* ************** */
                   &ABL, /**/&ABR,
                 mb, nb, FLA_TL );
```

where parameters `mb` and `nb` have values m_b and n_b, respectively. Examples of the use of this routine can also be found in Figs. 9 and 10.

NOTE 14. *Note that invocation of the operation*

```
FLA_Part_2x2( A, &ATL, /**/ &ATR,
                 /* ************** */
                   &ABL, /**/ &ABR,
                 mb, nb, FLA_TL );
```

in C creates four views, one for each quadrant. Subsequent modifications of the contents of a view affect therefore the original contents of the matrix. This is an important difference to consider with respect to the FLAME@lab API introduced in Section 3, where the quadrants are copies of the original matrix!

NOTE 15. *The above example remarks that formatting the code as well as the careful introduction of comments helps in capturing the algorithm in code. Clearly, much of the benefit of the API would be lost if in the example the code appeared as*

```
FLA_Part_2x2( A, &ATL, &ATR, &ABL, &ABR, mb, nb, FLA_TL );
```

From Figs. 1 and 2, we also realize the need for an operation that takes a 2×2 partitioning of a given matrix A and repartitions this into a 3×3 partitioning so that submatrices that need to be updated and/or used for computation can be identified. To support this, we introduce the call

```
void FLA_Repart_from_2x2_to_3x3

     ( FLA_Obj ATL, FLA_Obj ATR,   FLA_Obj *A00, FLA_Obj *A01, FLA_Obj *A02,
                                   FLA_Obj *A10, FLA_Obj *A11, FLA_Obj *A12,
       FLA_Obj ABL, FLA_Obj ABR,   FLA_Obj *A20, FLA_Obj *A21, FLA_Obj *A22,
       int mb, int nb, int quadrant )
```

Purpose: Repartition a 2×2 partitioning of matrix `A` into a 3×3 partitioning where `mb` $\times$ `nb` submatrix `A11` is split from the quadrant indicated by `quadrant`.

Here `quadrant` can again take on the values `FLA_TL`, `FLA_TR`, `FLA_BL`, and `FLA_BR` to indicate that `mb` and `nb` submatrix `A11` is split from submatrix `ATL`, `ATR`, `ABL`, or `ABR`, respectively.

Thus,

Repartition

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

$$\textbf{where } A_{11} \text{ is } m_b \times n_b$$

is captured in the code

```
FLA_Repart_from_2x2_to_3x3( ATL, ATR,     &A00, /**/ &A01, &A02,
                                        /* ******************** */
                                          &A10, /**/ &A11, &A12,
                            ABL, ABR,     &A20, /**/ &A21, &A22,
                            mb, nb, FLA_BR );
```

where parameters mb and nb have values m_b and n_b, respectively. Others examples of the use of this routine can also be found in Figs. 9 and 10.

NOTE 16. *The calling sequence of* FLA_Repart_from_2x2_to_3x3 *and related calls is a testimony to throwing out the convention that input parameters should be listed before output parameters or vice versa, as well as to careful formating. Notice that is it specifically by mixing input and output parameters in what on the surface may appear to be complete chaos that the repartitioning in the algorithm can be elegantly captured in code.*

NOTE 17. *Chosing variable names can further relate the code to the algorithm, as is illustrated by comparing*

$$\left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right) \quad \textit{and}$$

```
   L00,  /**/ l01,      L02,
/* ************************ */
   l10t, /**/ lambda11, l12t,
   L20,  /**/ l21,      L22, ...
```

in Figs. 1 and 9.

Once the contents of the corresponding views have been updated, the descriptions of A_{TL}, A_{TR}, A_{BL}, and A_{BR} must be updated to reflect that progress is being made, in terms of the regions identified by the double-lines. Moving the double-lines is achieved by a call to

```
void FLA_Cont_with_3x3_to_2x2

      ( FLA_Obj *ATL, FLA_Obj *ATR,   FLA_Obj A00, FLA_Obj A01, FLA_Obj A02,
                                      FLA_Obj A10, FLA_Obj A11, FLA_Obj A12,
        FLA_Obj *ABL, FLA_Obj *ABR,   FLA_Obj A20, FLA_Obj A21, FLA_Obj A22,
        int quadrant )
```

Purpose: Update the 2×2 partitioning of matrix A by moving the boundaries so that A11 is added to the quadrant indicated by quadrant.

Here the value of quadrant (FLA_TL, FLA_TR, FLA_BL, or FLA_BR) specifies the quadrant submatrix A11 is to be added.

For example,

Continue with

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

translates to the code

```
FLA_Cont_with_3x3_to_2x2( &ATL, &ATR,       A00, A01, /**/ A02,
                                            A10, A11, /**/ A12,
```

```
                                /* ****************** */
                          &ABL, &ABR,      A20, A21, /**/ A22,
                          FLA_TL );
```

Further examples of the use of this routine can again be found in Figs. 9 and 10.

Similarly, a matrix can be partitioned horizontally into two submatrices with the call

```
void FLA_Part_2x1( FLA_Obj A, FLA_Obj *AT,
                              FLA_Obj *AB,
                   int mb, int side )
```

Purpose: Partition matrix `A` into a top and bottom side where the side indicated by `side` has `mb` rows.

Here `side` can take on the values `FLA_TOP` or `FLA_BOTTOM` to indicate that `mb` indicates the row dimension of `AT` or `AB`, respectively.

Given that matrix A is already partitioned horizontally it can be repartitioned into three submatrices with the call

```
void FLA_Repart_from_2x1_to_3x1( FLA_Obj AT,   FLA_Obj *A0,
                                               FLA_Obj *A1,
                                 FLA_Obj AB,   FLA_Obj *A2,
                                 int mb, int side )
```

Purpose: Repartition a 2×1 partitioning of matrix `A` into a 3×1 partitioning where submatrix `A1` with `mb` rows is split from the side indicated by `side`.

Here `side` can take on the values `FLA_TOP` or `FLA_BOTTOM` to indicate that `mb` submatrix `A1` is partitioned from `AT` or `AB`, respectively.

Given a 3×1 partitioning of a given matrix A, the middle submatrix can be appended to either the first or last submatrix with the call

```
void FLA_Cont_with_3x1_to_2x1( FLA_Obj *AT,  FLA_Obj A0,
                                             FLA_Obj A1,
                               FLA_Obj *AB,  FLA_Obj A2,
                               int side )
```

Purpose: Update the 2×1 partitioning of matrix `A` by moving the boundaries so that `A1` is added to the side indicated by `side`.

Examples of the use of the routine that deals with the horizontal partitioning of matrices can be found in Figs. 9 and 10.

Finally, a matrix can be partitioned and repartitioned vertically with the calls

```
void FLA_Part_1x2( FLA_Obj A, FLA_Obj *AL, FLA_Obj *AR,
                   int nb, int side )
```

Purpose: Partition matrix `A` into a left and right side where the side indicated by `side` has `nb` columns.

and

```
void FLA_Repart_from_1x2_to_1x3
       ( FLA_Obj AL,              FLA_Obj AR,
         FLA_Obj *A0, FLA_Obj *A1, FLA_Obj *A2,
         int nb, int side )
```

Purpose: Repartition a 1×2 partitioning of matrix `A` into a 1×3 partitioning where submatrix `A1` with `nb` columns is split from the side indicated by `side`.

Here `side` can take on the values `FLA_LEFT` or `FLA_RIGHT`. Adding the middle submatrix to the first or last is now accomplished by a call to

```
void FLA_Cont_with_1x3_to_1x2
       ( FLA_Obj *AL,            FLA_Obj *AR,
         FLA_Obj A0, FLA_Obj A1, FLA_Obj A2,
         int side )
```

Purpose: Update the 1×2 partitioning of matrix `A` by moving the boundaries so that `A1` is added to the side indicated by `side`.

4.7 Computational kernels

All operations described in the last subsection hide the details of indexing in the linear algebra objects. To compute with and/or update data associated with a linear algebra object, one calls subroutines that perform the desired operations.

Such subroutines will typically take one of three forms:

—subroutines coded using the FLAME/C interface (including, possibly, a recursive call),

—subroutines coded using a more traditional coding style, or

—wrappers to highly optimized kernels.

Naturally these are actually three points on a spectrum of possibilities, since one can mix these techniques.

A subset of currently supported operations is given in Appendix A.6. Here, we discuss how to create subroutines that compute these operations. For additional information on supported functionality, please visit the webpage given at the end of this paper or [Gunnels and van de Geijn 2001a].

4.7.1 *Subroutines coded using the FLAME/C interface.* The subroutine itself could be coded using the FLAME approach to deriving algorithms [Bientinesi et al.] and the FLAME/C described in this section.

For example, the implementation in Fig. 10 of the blocked algorithm given in Fig. 2 requires the update $B_1 := L_{11}^{-1} B_1$ which can be implemented by a call to the unblocked algorithm in Fig. 9.

4.7.2 *Subroutine coded using a more traditional coding style.* Notice that there is an overhead for the abstractions that we introduce to hide indexing. For implementations of blocked algorithms, this overhead is amortized over a sufficient amount of computation that it is typically not of much consequence. (In the case of the algorithm in Fig. 2 when B is $m \times n$ the indexing overhead is $O(m/b)$ while

```
#include "FLAME.h"

void Trsm_llnn_unb_var1( FLA_Obj L, FLA_Obj B )
{
  FLA_Obj     LTL, LTR,     L00,  l01,      L02,    BT,           B0,
              LBL, LBR,     l10t, lambda11, l12t,   BB,           b1t,
                            L20,  l21,      L22,                  B2;

  FLA_Part_2x2( L,  &LTL, /**/ &LTR,
                  /* ************** */
                    &LBL, /**/ &LBR,   0, 0,     /* submatrix */ FLA_TL );
  FLA_Part_2x1( B,  &BT,
                   /***/
                    &BB,               0, /* length submatrix */ FLA_TOP );

  while ( FLA_Obj_length( LTL ) != FLA_Obj_length( L ) ){
    FLA_Repart_2x2_to_3x3( LTL, /**/ LTR,         &L00,  /**/ &l01,      &L02,
                        /* ************* */    /* *************************** */
                                /**/              &l10t, /**/ &lambda11, &l12t,
                           LBL, /**/ LBR,         &L20,  /**/ &l21,      &L22,
                           1, 1, /* lambda11 from */ FLA_BR );
    FLA_Repart_2x1_to_3x1( BT,                   &B0,
                          /**/                   /**/
                                                 &b1t,
                           BB,                   &B2,
                           1, /* length b1t from */ FLA_BOTTOM );
    /* ------------------------------------------------------------------ */

    FLA_Gemv( FLA_TRANSPOSE, MINUS_ONE, B0, l10t, ONE, b1t );
    FLA_Inv_scal( lambda11, b1t );

    /* ------------------------------------------------------------------ */
    FLA_Cont_with_3x3_to_2x2( &LTL, /**/ &LTR,      L00,  l01,      /**/ L02,
                                    /**/            l10t, lambda11, /**/ l12t,
                            /* ************** */ /* ************************ */
                              &LBL, /**/ &LBR,      L20,  l21,      /**/ L22,
                            /* lambda11 added to */ FLA_TL );
    FLA_Cont_with_3x1_to_2x1( &BT,                     B0,
                                                       b1t,
                             /***/                    /**/
                              &BB,                     B2,
                             /* b1t  added to */ FLA_TOP );
  }
}
```

Fig. 9. FLAME/C implementation of unblocked TRSM algorithm in Fig. 1.

the useful computation is $O(m^2n)$.) However, for unblocked algorithms or algorithms that operate on vectors, the relative cost is more substantial. In this case, it may become beneficial to code the subroutine using a more traditional style that exposes indices. For example, the operation

```
FLA_Inv_scal( lambda11, b1t );
```

```
#include "FLAME.h"

void Trsm_llnn_var1_blk( FLA_Obj L, FLA_Obj B, int nb_alg )
{
  FLA_Obj     LTL, LTR,     L00, L01, L02,    BT,           B0,
              LBL, LBR,     L10, L11, L12,    BB,           B1,
                            L20, L21, L22,                  B2;
  int         b;

  FLA_Part_2x2( L,  &LTL, /**/ &LTR,
                  /* ************** */
                    &LBL, /**/ &LBR,   0, 0,     /* submatrix */ FLA_TL );
  FLA_Part_2x1( B,  &BT,
                   /***/
                    &BB,               0, /* length submatrix */ FLA_TOP );

  while ( FLA_Obj_length( LTL ) != FLA_Obj_length( L ) ){
    b = min( FLA_Obj_length( LBR ), nb_alg );

    FLA_Repart_2x2_to_3x3( LTL, /**/ LTR,         &L00, /**/ &L01, &L02,
                        /* ************* */    /* ******************** */
                                /**/              &L10, /**/ &L11, &L12,
                           LBL, /**/ LBR,         &L20, /**/ &L21, &L22,
                           b, b, /* L11 from */ FLA_BR );
    FLA_Repart_2x1_to_3x1( BT,                   &B0,
                          /**/                   /**/
                                                 &B1,
                           BB,                   &B2,
                           b, /* length B1 from */ FLA_BOTTOM );
    /* ------------------------------------------------------------------ */

    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, L10, B0, ONE, B1 );
    Trsm_llnn_var1_unb( L11, B1 );

    /* ------------------------------------------------------------------ */
    FLA_Cont_with_3x3_to_2x2( &LTL, /**/ &LTR,         L00, L01, /**/ L02,
                                    /**/               L10, L11, /**/ L12,
                            /* ************** */   /* ****************** */
                              &LBL, /**/ &LBR,         L20, L21, /**/ L22,
                              /* L11 added to */ FLA_TL );
    FLA_Cont_with_3x1_to_2x1( &BT,                     B0,
                                                       B1,
                             /***/                   /**/
                              &BB,                     B2,
                              /* B1  added to */ FLA_TOP );
  }
}
```

Fig. 10. FLAME/C implementation of blocked TRSM algorithm in Fig. 2.

```
#include "FLAME.h"

void FLA_Inv_scal( FLA_Obj alpha, FLA_Obj x )
{
  int datatype_alpha, datatype_x, n_x, inc_x, i;
  double *buffer_alpha, *buffer_x, inv_alpha;

  datatype_alpha = FLA_Obj_datatype( alpha );
  datatype_x     = FLA_Obj_datatype( x );

  if (( datatype_alpha == FLA_DOUBLE ) &&
      ( datatype_x     == FLA_DOUBLE )) {

    n_x   = FLA_Obj_length( x );

    if ( n_x == 1 ){
      n_x = FLA_Obj_width( x );
      inc_x = FLA_Obj_ldim( x );
    }
    else inc_x = 1;

    buffer_alpha = ( double * ) FLA_Obj_buffer( alpha );
    buffer_x     = ( double * ) FLA_Obj_buffer( x );

    inv_alpha = 1.0 / *buffer_alpha;

    for ( i=0; i<n_x; i++ )
      *buffer_x++ *= inv_alpha;

    /* For BLAS based implementation, comment out above loop
       and uncomment the call to cblas_dscal below */

    /* cblas_dscal( n_x, inv_alpha, buffer_x, inc_x ); */
  }
  else FLA_Abort( "datatype not yet supported", __LINE__, __FILE__ );
}
```

Fig. 11. Sample implementation of `FLA_Inv_scal`.

can be implemented by the subroutine in Fig. 11. (Note that it is probably more efficient to instead implement it by calling `cblas_dscal` or the equivalent BLAS routine for the appropriate datatype.)

NOTE 18. *Even when a routine is ultimately implemented using more traditional code, it is beneficial to incorporate the FLAME/C code as comments for clarification.*

4.7.3 *Wrappers to highly optimized kernels.* A number of matrix and/or vector operations have been identified to be frequently used by the linear algebra community. Many of these are part of the BLAS. Since highly optimized implementations of these operations are supported by widely available library implementations, it makes sense to provide a set of subroutines that are simply wrappers to the BLAS. An example of this is given in Fig. 8.

5. FROM FLAME@LAB TO FLAME/C TO PLAPACK

As mentioned, we view the FLAME@lab and FLAME/C interfaces as tools on a migratory path that starts with the specification of the operation to be performed, after which the FLAME derivation process can be used to systematically derive a family of algorithms for computing the operation, followed by an initial implementation with FLAME@lab, a high-performance implementation with FLAME/C, and finally a parallel implementation with a FLAME/C-like extension of the PLAPACK interface.

5.1 Cases where FLAME@lab is particularly useful

Since clearly algorithms can be directly translated to FLAME/C, the question of the necessity for the FLAME@lab API arises. As is well known, MATLAB-like environments are extremely powerful interactive tools for manipulating matrices and investigating algorithms; interactivity is probably the key feature, allowing the user to speed up dramatically the design of procedures such as input generation and output analysis.

The authors have had the chance to exploit the FLAME@lab API in a number of research topics:

—In [Quintana-Ortí and van de Geijn 2003], the interface was used to investigate the numerical stability properties of algorithms derived for the solution of the triangular Sylvester equation.

—In an ongoing study, we are similarly using it for the analysis of the stability of different algorithms for inverting a triangular matrix. Several algorithms exist for this operation. We derived them by using the FLAME methodology and implemented them with FLAME@lab. For each variant measurements of different forms of residuals and forward errors had to be made [Higham 2002]. As part of the study, the input matrices needed to be chosen with extreme care and often they are the result from some other operation, like the `lu` function in MATLAB.

For these kinds of investigative studies high performance is not required. It is the interactive nature of tools like MATLAB that is especially useful.

5.2 Moving on to FLAME/C

Once derived algorithms have been implemented and tested with FLAME@lab, the transition to a high-performance implementation using the FLAME/C API is trivial, requiring only the translation for the operations in the loop-body to calls to subroutines with the functionality of the BLAS).

The most significant difference between the FLAME/C and FLAME@lab APIs is that for the FLAME/C interface, the partitioning routines return views (i.e., references) into the matrix. Thus, any subsequent modification of the view results in a modification of the original contents of the matrix. The use of views on the FLAME/C API avoids much of the unnecessary data copying that occurs in the FLAME@lab API, leading to a high-performance implementation.

5.3 And finally the parallel implementation

While the PLAPACK API already hid details of indexing by using objects and to a large degree inspired the FLAME/C API, the notion of tracking all submatrices of

```
#include "PLA.h"

void PLA_Trsm_llnn_var1_blk( PLA_Obj L, PLA_Obj B, int nb_alg )
{
  PLA_Obj
    LTL=NULL, LTR=NULL,  L00=NULL, L01=NULL, L02=NULL,  BT=NULL, B0=NULL,
    LBL=NULL, LBR=NULL,  L10=NULL, L11=NULL, L12=NULL,  BB=NULL, B1=NULL,
                         L20=NULL, L21=NULL, L22=NULL,           B2=NULL,
    MINUS_ONE=NULL, ZERO=NULL, ONE=NULL;
  int         b;

  PLA_Create_constants_conf_to( L, &MINUS_ONE, &ZERO, &ONE );

  PLA_Part_2x2( L,  &LTL, /**/ &LTR,
                /* ************** */
                  &LBL, /**. &LBR,   0, 0,     /* submatrix */ PLA_TL );
  PLA_Part_2x1( B,  &BT,
                 /***/
                  &BB,               0, /* length submatrix */ PLA_TOP );

  while ( PLA_Obj_length( LTL ) != PLA_Obj_length( L ) ){
    b = min( PLA_Obj_length( LBR ), nb_alg );
    PLA_Repart_2x2_to_3x3( LTL, /**/ LTR,         &L00, /**/ &L01, &L02,
                      /* ************* */    /* ********************* */
                              /**/              &L10, /**/ &L11, &L12,
                           LBL, /**/ LBR,         &L20, /**/ &L21, &L22,
                           b, b, /* L11 from */ PLA_BR );
    PLA_Repart_2x1_to_3x1( BT,                   &B0,
                          /**/                   /**/
                                                 &B1,
                           BB,                   &B2,
                           b, /* length B1 from */ PLA_BOTTOM );
    /* ------------------------------------------------------------------ */

    PLA_Gemm( PLA_NO_TRANSPOSE, PLA_NO_TRANSPOSE, MINUS_ONE, L10, B0, ONE, B1 );
    PLA_Trsm_llnn_var1_unb( L11, B1 );

    /* ------------------------------------------------------------------ */
    PLA_Cont_with_3x3_to_2x2( &LTL, /**/ &LTR,         L00, L01, /**/ L02,
                                    /**/               L10, L11, /**/ L12,
                             /* ************** */   /* ****************** */
                              &LBL, /**/ &LBR,         L20, L21, /**/ L22,
                              /* L11 added to */ PLA_TL );
    PLA_Cont_with_3x1_to_2x1( &BT,                     B0,
                                                       B1,
                             /***/                    /**/
                              &BB,                     B2,
                              /* B1  added to */ PLA_TOP );
  }
  PLA_Obj_free( &LTL );
    [ : ]
  PLA_Obj_free( &ONE );
}
```

Fig. 12. FLAME/PLAPACK implementation of the blocked TRSM implementation in Fig. 10.

the matrices involved in the computation like FLAME/C does is new. Specifically, the routines `FLA_Repart_...` and `FLA_Cont_with_...` were not part of the original PLAPACK API. As part of our project, we have now added similar routines to the PLAPACK API. An implementation using PLAPACK for TRSM is given in Fig. 12. In essence, a parallel implementation can be created by replacing `FLAME.h` with `PLA.h` and all prefixes `FLA_` with `PLA_`. Notice that in PLAPACK, objects are defined as pointers to structures that are dynamically allocated. As a result, the declarations are somewhat different when compared to the FLAME/C code in Fig. 10. Furthermore, these so allocated objects must be freed at the end of the routine. Finally, the constants `MINUS_ONE`, `ZERO`, and `ONE` must be created in each new routine. These idiosyncrasies suggest that it is time to update the PLAPACK API to become closer to the FLAME API.

In addition to attaining performance by casting computation as much as possible in terms of matrix-matrix operations (blocked algorithms), a parallel implementation requires careful assignment of data and work to individual processors. Clearly, the FLAME/C interface does not capture this, nor does the most trivial translation of FLAME/C to FLAME/PLAPACK. It is here where the full PLAPACK API allows the user to carefully manipulate the data and the operations, while still coding at a high level of abstraction. This manipulation is relatively systematic. Indeed, the Broadway compiler can to some degree automate this process [Guyer and Lin 1999; 2000b; 2000a; Guyer et al. 2001]. Also, an automated system for directly translating algorithms like those given in Section 2 to optimized PLAPACK code have been prototyped [Gunnels 2001].

Further details regarding parallel implementations go beyond the scope of this paper.

5.4 MATLAB to parallel implementations

Notice that in some sense, our work answers the question of how to generate parallel implementations from algorithms coded in MATLAB M-script [Moler et al. 1987]: For the class of problems to which the presented approach applies, the answer is to start with the algorithm, and to create APIs that can target MATLAB, C, or parallel architectures.

6. PERFORMANCE

In a number of papers that were already mentioned in the introduction we have shown that the FLAME/C API can be used to attain high performance for implementations of a broad range of linear algebra operations. Thus, we do not include a traditional performance section. Instead, we discuss some of the issues.

Conventional wisdom used to dictate that raising the level of abstraction at which one codes will adversely impact the performance of the implementation. We, like others, disagree for a number of reasons:

—By raising the level of abstraction, more ambitious algorithms can be implemented which can achieve higher performance [Gunnels et al. 2001; Quintana-Ortí and van de Geijn 2003; Gunnels and van de Geijn 2001b; Bientinesi et al. 2002; Alpatov et al. 1997; van de Geijn 1997].

One can, of course, argue that these same algorithms can also be implemented at a lower level of abstraction. While this is true for individual operations, implementing entire libraries at a low level of abstraction greatly increases the effort required to implement, maintain, and verify correctness.

—Once implementations are implemented with an API at a high level of abstraction, components can be selectively optimized at a low level of abstraction. We learn from this that the API must be designed to easily accommodate this kind of optimization, as is also discussed in Section 4.7.

—Recent compiler technology (e.g., [Guyer and Lin 1999; 2000b; 2000a; Guyer et al. 2001]) allows library developers to specify dependencies between routines at a high level of abstraction which allows compilers to optimize between layers of libraries, automatically achieving the kinds of optimizations that would otherwise be performed by hand.

—Other situations in which abstraction offers the opportunity for higher performance include several mathematical libraries and C++ optimization techniques as well. For example, PMLP [Birov et al. 1998] uses C++ templates to support many different storage formats, thereby decoupling storage format from algorithmic correctness in classes of sparse linear algebra, thus allowing this degree of freedom to be explored for optimizing performance. Also, PMLP features operation sequences and non-blocking operations in order to allow scheduling of mathematical operations asynchronously from user threads. Template metaprogramming and expression templates support concepts including compile-time optimizations involving loop fusion, expression simplification, and removal of unnecessary temporaries; these allow C++ to utilize fast kernels while removing abstraction barriers between kernels, and further abstraction barriers between sequences of user operations (systems include Blitz++ [Veldhuizen et al.] and PETE [PET]). These techniques, in conjunction with an appropriate FLAME-like API for C++, should allow our algorithms to be expressed at a high level of abstraction without compromising performance.

NOTE 19. *The lesson to be learned is that by raising the level of abstraction, a high degree of confidence in the correctness of the implementation can be achieved while more aggressive optimizations, by hand or by a compiler, can simultaneously be facilitated.*

7. PRODUCTIVITY AND THE FLAME INTERFACE REPOSITORY (FIRE)

In the abstract and introduction of this paper, we make claims regarding the impact of the presented approach on productivity. In this section, we narrate a few experiences.

7.1 Sequential implementation of algorithms for the triangular Sylvester equation

A clear demonstration that the FLAME derivation process, in conjunction with the FLAME APIs, can be used to quickly generate new algorithms and implementations for non-trivial operations came in form of a family of algorithms for the triangular Sylvester equations. Numerous previously unknown high-performance algorithms were derived in a matter of hours, and implemented using the FLAME/C API in less than a day. In response to the submitted related paper, referees requested that the numerical properties of the resulting implementations be investigated. In an effort to oblidge, the FLAME@lab interface was created, and numerical experiments were performed with the aid of the MATLAB environment. The resulting paper has now appeared [Quintana-Ortí and van de Geijn 2003].

7.2 Parallel implementation of the reduction to tridiagonal form

As part of an effort to parallelize the Algorithm of Multiple Relatively Robust Representations (MRRR) for dense symmetric eigenproblems, a parallel implementation of the reduction to tridiagonal form via Householder transformations was developed using PLAPACK [Bientinesi et al. 2003]. First, a careful description of the algorithms was created, in the format presented in Section 2. Next, a FLAME@lab implementation was created, followed by a FLAME/C implementation. Finally, the sequential code was ported to PLAPACK. The entire development of this implementation from start to finish took about a day of the time of two of the authors.

7.3 Undergraduate and graduate education

As part of undergraduate and graduate courses at UT-Austin on high-performance parallel computing, a typical early exercise included the sequential implementation of a blocked linear algebra algorithm. As the classes become more popular and populated, this simple exercise exposed that invariably small mistakes in indexing required excessive time to debug on the part of the student and/or on the part of the instructor. This greatly reduced the breadth of material that could be covered in the course. With the introduction of FLAME@lab and FLAME/C, this problem has completely dissipated. Furthermore, much more ambitious semester projects can be undertaken by the students.

7.4 Assembling the FLAME Interface REpository (FIRE)

As part of the above-mentioned courses, students have been generating algorithms and implementations for a broad spectrum of linear algebra operations. An undergraduate in one of these classes, Minhaz Khan, took it upon himself to systematically assemble many of these implementations in the FLAME Interface REpository (FIRE). To date, hundreds of implementations of dozens of algorithms have been cataloged, about half single-handedly by this student. After some experience was gained, he reported being able to derive, prove correct, implement, and test algorithms at a rate of about seven minutes per algorithm for BLAS-like operations involving several triangular matrices[2].

8. CONCLUSION

In this paper, we have presented simple APIs for implementing linear algebra algorithms using the MATLAB M-script and C programming languages. In isolation, these interfaces illustrate how raising the level of abstraction at which one codes allows one to avoid intricate indexing in the code, which reduces the opportunity for the introduction of errors and raises the confidence in the correctness of the code. In combination with our formal derivation methodology, the APIs can be used to implement algorithms derived using that methodology so that the proven

[2]These operations are similar to those supported by the LAPACK auxiliary routines DLAUUM andDLAUU2.

correctness of those algorithms translates to a high degree of confidence in the implementation.

We want to emphasize that the presented APIs are merely meant to illustrate the issues. Similar interfaces for the Fortran, C++, and other languages are easily defined, allowing special features of those languages to be used to even further raise the level of abstraction at which one codes.

Finally, an increasing number of linear algebra operations have been captured with our formal derivation methodology. This set of operations includes, to name but a few, the complete levels 1, 2, and 3 BLAS, factorization operations such as the LU and QR (with and without pivoting), reduction to condensed forms, and linear matrix equations arising in control. An ever-growing collection of linear algebra operations written using the FLAME@lab and FLAME/C interfaces can be found at the URI given below.

Further Information

For further information on the FLAME project and to download the FLAME@lab or FLAME/C interface, visit

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

The FIREsite repository is being maintained at

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

Acknowledgments

Support for this research was provided by NSF grant ACI-0203685 and ACI-0305163. Additional support for this work came from the Visiting Researcher program of the Institute for Computational Engineering and Sciences (ICES).

An ever-growing number of people have contributed to date to the methodology that underlies the Formal Linear Algebra Methods Environment. These include

—UT-Austin: Brian Gunter, Mark Hinga, Minhaz Khan, Dr. Margaret Myers, Vinod Valsalam, and Thierry Joffrain.

—IBM's T.J. Watson Research Center: Dr. John Gunnels and Dr. Fred Gustavson.

—Intel: Dr. Greg Henry.

—Mississippi State University: Prof. Anthony Skjellum and Wenhao Wu.

In addition, numerous students in undergraduate and graduate courses on high-performance computing at UT-Austin have provided valuable feedback.

REFERENCES

PETE, the portable expression template engine. http://www.acl.lanl.gov/pete Date of access: Oct 24th, 2002.

Alpatov, P., Baker, G., Edwards, C., Gunnels, J., Morrow, G., Overfelt, J., van de Geijn, R., and Wu, Y.-J. J. 1997. PLAPACK: Parallel linear algebra package – design overview. In *Proceedings of SC97*.

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.

BAKER, G., GUNNELS, J., MORROW, G., RIVIERE, B., AND VAN DE GEIJN, R. 1998. PLAPACK: High performance through high level abstraction. In *Proceedings of ICCP98*.

BALAY, S., GROPP, W., MCINNES, L. C., AND SMITH, B. 1996. PETSc 2.0 users manual. Tech. Rep. ANL-95/11, Argonne National Laboratory. Oct.

BIENTINESI, P., DHILLON, I., AND VAN DE GEIJN, R. 2003. A parallel eigensolver for dense symmetric matrices based on multiple relatively robust representations. *SIAM J. Sci. Comput.*. to be submitted.

BIENTINESI, P., GUNNELS, J. A., GUSTAVSON, F. G., HENRY, G. M., MYERS, M. E., QUINTANA-ORTI, E. S., AND VAN DE GEIJN, R. A. 2002. The science of programming high-performance linear algebra libraries. In *Proceedings of Performance Optimization for High-Level Languages and Libraries (POHLL-02) , a workshop in conjunction with the 16th Annual ACM International Conference on Supercomputing (ICS'02)*.

BIENTINESI, P., GUNNELS, J. A., MYERS, M. E., QUINTANA-ORTÍ, E. S., AND VAN DE GEIJN, R. A. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Soft.*. submitted.

BIROV, L., PURKAYASTHA, A., SKJELLUM, A., DANDASS, Y., AND BANGALORE, P. V. 1998. PMLP home page. http://www.erc.msstate.edu/labs/hpcl/pmlp.

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.

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.

GROPP, W., LUSK, E., AND SKJELLUM, A. 1994. *Using MPI*. The MIT Press.

GUNNELS, J. A. 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. A., GUSTAVSON, F. G., HENRY, G. M., AND VAN DE GEIJN, R. A. 2001. FLAME: Formal linear algebra methods environment. *ACM Trans. Math. Soft. 27,* 4 (December), 422–455.

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/`.

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.

GUYER, S. Z., BERGER, E., AND LIN, C. 2001. Customizing software libraries for performance portability. In *10^{th} SIAM Conference on Parallel Processing for Scientific Computing*.

GUYER, S. Z. AND LIN, C. 1999. An annotation language for optimizing software libraries. In *Second Conference on Domain Specific Languages*. 39–52.

GUYER, S. Z. AND LIN, C. 2000a. *Broadway: A Software Architecture for Scientific Computing*. Kluwer Academic Press, 175–192.

GUYER, S. Z. AND LIN, C. 2000b. Optimizing the use of high performance software libraries. In *Languages and Compilers for Parallel Computing*. 221–238.

HIGHAM, N. J. 2002. *Accuracy and Stability of Numerical Algorithms*, Second ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

KERNIGHAN, B. AND RITCHIE, D. 1978. *The C programming language*. Prentice-Hall, Englewood Cliffs, NJ.

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.

MOLER, C., LITTLE, J., AND BANGERT, S. 1987. *Pro-Matlab, User's Guide*. The Mathworks, Inc.

Quintana-Ortí, E. S. and van de Geijn, R. A. 2003. Formal derivation of algorithms for the triangular Sylvester equation. *ACM Trans. Math. Soft. 29,* 2, 218–243.

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. 1973. *Introduction to Matrix Computations.* Academic Press, Orlando, Florida.

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

Veldhuizen, T. et al. Blitz++. URL:http://monet.uwaterloo.ca/blitz/.

A. SUMMARY OF FLAME/C ROUTINES

In this appendix, we list a number of routines supported as part of the current implementation of the FLAME/C interface. Those experienced with linear algebra libraries will recognize most routines as implementing the functionality of the BLAS.

A.1 Notation

In the descriptions of the below discussed routines, we use the following conventions:

—**Matrices, vectors, and scalar** are denoted by upper-case, lower-case, and lower-case Greek letters, respectively.

—**Transposition:**

$$\mathrm{op}_{\texttt{trans}}(X) = \begin{cases} X & \text{if } \texttt{trans == FLA\_NO\_TRANSPOSE} \\ X^T & \text{if } \texttt{trans == FLA\_TRANSPOSE} \\ \bar{X}^T & \text{if } \texttt{trans == FLA\_CONJ\_TRANSPOSE} \\ \bar{X} & \text{if } \texttt{trans == FLA\_CONJUGATE} \end{cases}.$$

—**Triangular matrices:** Parameter `uplo` is used to indicate whether a triangular matrix is stored in the lower or upper triangular part of array `A`. This parameter can take on the values `FLA_LOWER_TRIANGULAR` and `FLA_UPPER_TRIANGULAR`. Parameter `diag` is used to indicate the values of the diagonal elements of matrix A:

`diag`	Values on the diagonal of A
`FLA_NONUNIT_DIAG`	As stored in `A`.
`FLA_UNIT_DIAG`	Implicitly take all diagonal elements to equal one.
`FLA_ZERO_DIAG`	Implicitly take all diagonal elements to equal zero.

—**Symmetric matrices:** Only the upper or lower triangular part of a symmetric matrix is stored, as indicated by parameter `uplo`:

`uplo`	Storage of matrix
`FLA_LOWER_TRIANGULAR`	Only lower triangular part of matrix is stored.
`FLA_UPPER_TRIANGULAR`	Only upper triangular part of matrix is stored.

Notice that the values in the other part of the matrix cannot be disturbed and/or used.

—Unless specified otherwise, all routines are of type `void`.

A.2 Initializing and finalizing FLAME

`FLA_Init()`
Initialize FLAME.

`FLA_Finalize()`
Finalize FLAME.

A.3 Linear algebra objects

`FLA_Obj_create(int datatype, int m, int n, FLA_Obj *matrix)`
Create an object that describes an $m \times n$ matrix and create the associated storage array.
`FLA_Obj_create_without_buffer(int datatype, int m, int n, FLA_Obj *matrix)`
Create an object that describes an $m \times n$ matrix without creating the associated storage array.
`FLA_Obj_attach_buffer(*buff, int ldim, FLA_Obj *matrix)`
Attach an existing buffer that holds a matrix stored in column-major order with leading dimension `ldim` to the object `matrix`.
`FLA_Obj_create_conf_to(int trans, FLA_Obj old_matrix, FLA_Obj *new_matrix)`
Like `FLA_Obj_create` except that it creates an object with same datatype and dimensions as `old`, transposing if desired.
`FLA_Obj_free(FLA_Obj *matrix)`
Free all space allocated to store data associated with `obj`.
`int FLA_Obj_datatype(FLA_Obj matrix)`
Extract datatype of matrix.
`int FLA_Obj_length(FLA_Obj matrix)`
Extract row dimension of matrix.
`int FLA_Obj_width(FLA_Obj matrix)`
Extract column dimension of matrix.
`void *FLA_Obj_buffer(FLA_Obj matrix)`
Extract the address where the matrix is stored.
`int FLA_Obj_ldim(FLA_Obj matrix)`
Extract the leading dimension for the array in which the matrix is stored.

A.4 Views

```
FLA_Part_2x2(FLA_Obj A, FLA_Obj *ATL, FLA_Obj *ATR,
                        FLA_Obj *ABL, FLA_Obj *ABR,
             int mb, int nb, int quadrant)
```

Partition matrix A into four quadrants where the quadrant indicated by `quadrant` is `mb` $\times$ `nb`.

```
FLA_Repart_from_2x2_to_3x3
     (FLA_Obj ATL, FLA_Obj ATR,   FLA_Obj *A00, FLA_Obj *A01, FLA_Obj *A02,
                                  FLA_Obj *A10, FLA_Obj *A11, FLA_Obj *A12,
      FLA_Obj ABL, FLA_Obj ABR,   FLA_Obj *A20, FLA_Obj *A21, FLA_Obj *A22,
      int mb, int nb, int quadrant)
```

Repartition a 2×2 partitioning of matrix A into a 3×3 partitioning where `mb` $\times$ `nb` submatrix A_{11} is split from the quadrant indicated by `quadrant`.

```
FLA_Cont_with_3x3_to_2x2
      (FLA_Obj *ATL, FLA_Obj *ATR,   FLA_Obj A00, FLA_Obj A01, FLA_Obj A02,
                                     FLA_Obj A10, FLA_Obj A11, FLA_Obj A12,
       FLA_Obj *ABL, FLA_Obj *ABR,   FLA_Obj A20, FLA_Obj A21, FLA_Obj A22,
       int quadrant)
```

Update the 2×2 partitioning of matrix A by moving the boundaries so that A_{11} is added to the quadrant indicated by `quadrant`.

```
FLA_Part_2x1(FLA_Obj A, FLA_Obj *AT,
                        FLA_Obj *AB,
             int mb, int side)
```
Partition matrix A into a top and bottom side where the side indicated by `side` has `mb` rows.

```
FLA_Repart_from_2x1_to_3x1(FLA_Obj AT,   FLA_Obj *A0,
                                         FLA_Obj *A1,
                           FLA_Obj AB,   FLA_Obj *A2,
                           int mb, int side)
```
Repartition a 2×1 partitioning of matrix A into a 3×1 partitioning where submatrix A_1 with `mb` rows is split from the side indicated by `side`.

```
FLA_Cont_with_3x1_to_2x1(FLA_Obj *AT,  FLA_Obj A0,
                                       FLA_Obj A1,
                         FLA_Obj *AB,  FLA_Obj A2, int side)
```
Update the 2×1 partitioning of matrix A by moving the boundaries so that A_1 is added to the side indicated by `side`.

```
FLA_Part_1x2(FLA_Obj A, FLA_Obj *AL, FLA_Obj *AR,
             int nb, int side)
```
Partition matrix A into a left and right side where the side indicated by `side` has `nb` columns

```
FLA_Repart_from_1x2_to_1x3(FLA_Obj AL,               FLA_Obj AR,
                           FLA_Obj *A0, FLA_Obj *A1, FLA_Obj *A2,
                           int nb, int side)
```
Repartition a 1×2 partitioning of matrix A into a 1×3 partitioning where submatrix A_1 with `nb` columns is split from the side indicated by `side`.

```
FLA_Cont_with_1x3_to_1x2(FLA_Obj *AL,            FLA_Obj *AR,
                         FLA_Obj A0, FLA_Obj A1, FLA_Obj A2, int side)
```
Update the 1×2 partitioning of matrix A by moving the boundaries so that A_1 is added to the side indicated by `side`.

A.5 Printing the contents of an object

```
FLA_Obj_show(char *string1, FLA_Obj A, char *format, char *string2)
```
Print the contents of A.

A.6 Subset of supported operations

```
FLA_Axpy(FLA_Obj alpha, FLA_Obj A, FLA_Obj B)
```
$B := \alpha A + B$.

```
FLA_Axpy_x(int trans, FLA_Obj alpha, FLA_Obj A, FLA_Obj B)
```
$B := \alpha \mathrm{op}_{\texttt{trans}}(A) + B$.

```
FLA_Copy(FLA_Obj A, FLA_Obj B)
```
$B := A$.

```
FLA_Copy_x(int trans, FLA_Obj A, FLA_Obj B)
```
$B := \mathrm{op}_{\texttt{trans}}(A)$.

```
FLA_Dot(FLA_Obj x, FLA_Obj y, FLA_Obj rho)
```
$\rho := x^T y$.

```
FLA_Dot_x(FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj beta, FLA_Obj rho)
```
$\rho := \alpha x^T y + \beta \rho$.

```
FLA_Gemm(int transa, int transb, FLA_Obj alpha, FLA_Obj A, FLA_Obj B,
                                 FLA_Obj beta,  FLA_Obj C)
```
$C := \alpha \mathrm{op}_{\texttt{transa}}(A)\mathrm{op}_{\texttt{transb}}(B) + \beta C$.

```
FLA_Gemv(int trans, FLA_Obj alpha, FLA_Obj A, FLA_Obj x, FLA_Obj beta,  FLA_Obj y)
```
$y := \alpha \mathrm{op}_{\texttt{trans}}(A)x + \beta y$.

```
FLA_Ger(FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj A)
```
$A := \alpha x y^T + A$.

```
FLA_Iamax(FLA_Obj x, FLA_Obj k)
```
Compute index k such that $|\chi_k| = \|x\|_\infty$. **Note:** This operation only works when x has unit row or column dimension.

```
FLA_Invert(FLA_Obj alpha)
```
$\alpha := 1/\alpha$.

```
FLA_Inv_scal(FLA_Obj alpha, FLA_Obj A)
```
$A := \frac{1}{\alpha} A$.

```
FLA_Negate(FLA_Obj A)
```
$A := -A$.

```
FLA_Nrm1(FLA_Obj A, FLA_Obj alpha)
```
$\alpha := \|A\|_1$.

```
FLA_Nrm2(FLA_Obj x, FLA_Obj alpha)
```
$\alpha := \|x\|_2$. **Note:** This operation only works when x has unit row or column dimension.

```
FLA_Nrm_inf(FLA_Obj A, FLA_Obj alpha)
```
$\alpha := \|A\|_\infty$.

```
FLA_Obj_set_to_one(FLA_Obj A)
```
Set all elements of A to one.

```
FLA_Obj_set_to_zero(FLA_Obj A)
```
Set all elements of A to zero.

```
FLA_Random_matrix(FLA_Obj A)
```
Fill A with random values in the range $(-1, 1)$.

```
FLA_Scal(FLA_Obj alpha, FLA_Obj A)
```
$A := \alpha A$.

```
FLA_Set_diagonal(FLA_Obj sigma, FLA_Obj A)
```
Set the diagonal of A to σI. All other values in A are unaffected.

```
FLA_Shift_spectrum(FLA_Obj alpha, FLA_Obj sigma, FLA_Obj A)
```
$A := A + \alpha\sigma I$.

```
FLA_Sqrt(FLA_Obj alpha)
```
$\alpha := \sqrt{\alpha}$. **Note:** `A` must describe a scalar.

```
FLA_Swap(FLA_Obj A, FLA_Obj B)
```
$A, B := B, A$.

```
FLA_Swap_x(int trans, FLA_Obj A, FLA_Obj B)
```
$A, B := \mathrm{op}_{trans}(A), \mathrm{op}_{trans}(B)$.

`FLA_Symm(int side, int uplo, FLA_Obj alpha, FLA_Obj A, FLA_Obj B,`
`FLA_Obj beta, FLA_Obj C)`
$C := \alpha AB + \beta C$ or $C := \alpha BA + \beta C$, where A is symmetric, `side` indicates the side from which A multiplies B, `uplo` indicates whether A is stored in the upper or lower triangular part of `A`.

`FLA_Symmetrize(int uplo, int conj, FLA_Obj A)` $A := \text{symm}(A)$ or $A := \text{herm}(A)$, where `uplo` indicates whether A is originally stored only in the upper or lower triangular part of `A`.
`FLA_Symv(int uplo, FLA_Obj alpha, FLA_Obj A, FLA_Obj x, FLA_Obj beta, FLA_Obj y)` $y := \alpha Ax + \beta y$, where A is symmetric and stored in the upper or lower triangular part of `A`, as indicated by `uplo`.
`FLA_Syr(int uplo, FLA_Obj alpha, FLA_Obj x, FLA_Obj A)` $A := \alpha xx^T + A$, where A is symmetric and stored in the upper or lower triangular part of `A`, as indicated by `uplo`.
`FLA_Syr2(int uplo, FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj A)` $A := \alpha xy^T + \alpha yx^T + A$, where A is symmetric and stored in the upper or lower triangular part of `A`, as indicated by `uplo`.

`FLA_Syr2k(int uplo, int trans, FLA_Obj alpha, FLA_Obj A, FLA_Obj B,`
`FLA_Obj beta, FLA_Obj C)`
$C := \alpha(\text{op}_{\texttt{trans}}(A)\text{op}_{\texttt{trans}}(B)^T + \text{op}_{\texttt{trans}}(B)\text{op}_{\texttt{trans}}(A)^T + \beta C$, where C is symmetric and stored in the upper or lower triangular part of `C`, as indicated by `uplo`.

`FLA_Syrk(int uplo, int trans, FLA_Obj alpha, FLA_Obj A, FLA_Obj beta, FLA_Obj C)` $C := \alpha\text{op}_{\texttt{trans}}(A)\text{op}_{\texttt{trans}}(A)^T + \beta C$, where C is symmetric and stored in the upper or lower triangular part of `C`, as indicated by `uplo`.
`FLA_Triangularize(int uplo, int diag, FLA_Obj A)` $A := \text{lower}(A)$ or $A := \text{upper}(A)$.

`FLA_Trmm(int side, int uplo, int trans, int diag,`
`FLA_Obj alpha, FLA_Obj A, FLA_Obj B)`
$B := \alpha\text{op}_{\texttt{trans}}(A)B$ (`side == FLA_LEFT`) or $B := \alpha B\text{op}_{\texttt{trans}}(A)$ (`side == FLA_RIGHT`). where A is upper or lower triangular, as indicated by `uplo`.

`FLA_Trmm_x(int side, int uplo, int transa, int diag,`
`FLA_Obj alpha, FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C)`
$C := \alpha\text{op}_{\texttt{transa}}(A)(B) + \beta C$ (`side == FLA_LEFT`) or $C := \alpha(B)\text{op}_{\texttt{transa}}(A) + \beta C$ (`side == FLA_RIGHT`) where A is upper or lower triangular, as indicated by `uplo`.

`FLA_Trmv(int uplo, int trans, int diag, FLA_Obj A, FLA_Obj x)` $x := \text{op}_{\texttt{trans}}(A)x$, where A is upper or lower triangular, as indicated by `uplo`.

`FLA_Trmv_x(int uplo, int trans, int diag, FLA_Obj alpha,`
`FLA_Obj A, FLA_Obj x, FLA_Obj beta, FLA_Obj y)`
Update $y := \alpha\text{op}_{\texttt{trans}}(A)x + \beta y$, where A is upper or lower triangular, as indicated by `uplo`.

`FLA_Trsm(int side, int uplo, int trans, int diag,`
`FLA_Obj alpha, FLA_Obj A, FLA_Obj B)`
$B := \alpha\text{op}_{\texttt{trans}}(A)^{-1}B$ (`SIDE == FLA_LEFT`) or $B := \alpha B\text{op}_{\texttt{trans}}(A)^{-1}$ (`SIDE == FLA_RIGHT`) where A is upper or lower triangular, as indicated by `uplo`.

`FLA_Trsm_x(int side, int uplo, int transa, int diag,`
`FLA_Obj alpha, FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C)`
$C := \alpha\text{op}_{\texttt{transa}}(A)^{-1}(B) + \beta C$ (`SIDE == FLA_LEFT`) or $C := \alpha(B)\text{op}_{\texttt{transa}}(A)^{-1} + \beta C$ (`SIDE == FLA_RIGHT`) where A is upper or lower triangular, as indicated by `uplo`.

```
FLA_Trsv(int uplo, int trans, int diag, FLA_Obj A, FLA_Obj x)
```

$x := \mathrm{op}_{\texttt{trans}}(A)^{-1}x$, where A is upper or lower triangular, as indicated by `uplo`.

```
FLA_Trsv_x(int uplo, int trans, int diag, FLA_Obj alpha,
           FLA_Obj A, FLA_Obj x, FLA_Obj beta,  FLA_Obj y)
```

$y := \alpha\mathrm{op}_{\texttt{trans}}(A)^{-1}x + \beta y$, where A is upper or lower triangular, as indicated by `uplo`.

B. INTERFACING TRADITIONAL CODE WITH FLAME/C

In this appendix, we describe briefly how an application that previously interfaced with a library like LAPACK can interface with FLAME/C.

Consider a call to the CBLAS routine (C interface to the BLAS) for computing the triangular solve with multiple right-hand-sides:

```
cblas_dtrsm( CblasColMaj, CblasLeft, CblasLower,
                          CblasNoTrans, CblasNonUnit,
             m, n, 1.0D00, A, ldA, B, ldB );
```

In this call, arrays `A` and `B` are assumed to have leading dimension stored in variables `ldA` and `ldB`, respectively, and matrix B, stored in array `B` is assumed to be of dimension $m \times n$, with m and n stored in variables `m` and `n`, respectively. This call can be replaced by the code segment

```
FLA_Obj_create_without_buffer( FLA_DOUBLE, m, m, &Aobj );
FLA_Obj_attach_buffer( ( void *) A, ldA, &Aobj );
FLA_Obj_create_without_buffer( FLA_DOUBLE, m, n, &Bobj );
FLA_Obj_attach_buffer( ( void *) B, ldB, &Bobj );

FLA_Trsm( FLA_LEFT, FLA_LOWER_TRIANGULAR,
          FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
          ONE, Aobj, Bobj );
```

The FLAME/C calls above are explained next.

Sometimes it will be useful to create a descriptor without storage for the array. This allows a matrix that has already been stored in a conventional two-dimensional array to be attached to an object. The following call creates such a descriptor:

```
void FLA_Obj_create_without_buffer
          ( int datatype, int m, int n, FLA_Obj *matrix )
```

Purpose: Create an object that describes an $m \times n$ matrix without creating the associated storage array.

Once an object has been created without an attached storage array, an existing two-dimensional array can be attached by calling

```
void FLA_Obj_attach_buffer( void *buff, int ldim, FLA_Obj *matrix )
```

Purpose: Attach an existing buffer that holds a matrix stored in column-major order with leading dimension `ldim` to the object `matrix`.

Received Month Year; revised Month Year; accepted Month Year