FLAME@lab: A Farewell to Indices*

FLAME Working Note #11

Paolo Bientinesi
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
pauldj@cs.utexas.edu

Enrique S. Quintana-Ortí
Depto. de Ingeniería y Ciencia de Computadores
Universidad Jaume I
12.071–Castellón (Spain)
quintana@icc.uji.es

Robert A. van de Geijn
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
rvdg@cs.utexas.edu

April 9, 2003

Abstract

MATLAB-like environments have been essential tools for the development of linear algebra libraries for almost three decades. The benefits include ease of implementation and maintenance of code, functionality, and interactivity. In this paper, we make the seemingly outrageous claim that the script language used for such environment is unnecessarily complex and stands in the way of the rapid development of robust, readable, and maintainable code. To correct this problem, we propose the introduction of nine almost trivial functions, the FLAME@lab API, that hide complex index manipulation. In isolation, the FLAME@lab interface illustrates how raising the level of abstraction at which one codes allows one to avoid intricate indexing in the code, thereby reducing the opportunity for the introduction of errors and raising the confidence in the correctness of the code. In combination with our Formal Linear Algebra Methods Environment (FLAME) approach to deriving linear algebra algorithms, FLAME@lab becomes an API for implementing proven correct algorithms. Finally, in combination with a similar API for C and for distributed memory parallel architectures (our PLAPACK environment), FLAME@lab becomes a natural step in the development of high-performance and parallel linear algebra libraries.

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. 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

*This work was supported in part by NSF grant ACR-0203685 and the Visiting Researcher program of the Institute for Computational Engineering and Sciences (ICES).

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

Figure 1: Unblocked algorithm for the TRSM example.

of complex indexing that programming errors are often encountered and confidence in code is diminished. Thus a carefully designed API should avoid explicit indexing whenever possible.

We have illustrated to the high-performance linear algebra library community the benefits of the formal derivation of algorithms in a series of previous papers [8, 12, 4]. While there we alluded at an API that allows code to reflect algorithms that have been derived to be correct, in this paper we explicitly give this API for the MATLAB [11] M-script programming language. Notice that in [14] we present a similar API for the C programming language.

Our FLAME@lab interface comes to fill a gap in the development cycle of linear algebra algorithms, giving the user the flexibility of MATLAB to test the algorithms designed using FLAME before going to a high-performance sequential implementation using our FLAME/C API, and the subsequent parallel implementation using, e.g., the Parallel Linear Algebra Package (PLAPACK) [15, 3, 1].

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. A discussion of how the developed algorithms, coded using the M-script language, can be migrated to sequential and parallel code written in C is discussed in Section 4. A few concluding remarks are given in Section 5.

2 A Typical Dense Linear Algebra Algorithm

In [4] 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 [4].

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

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

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

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

$$\vdots$$

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 to be updated and/or used in the next 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 [13].*

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 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 $B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$ 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**

Determine block size b

Repartition

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

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

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

enddo

Figure 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
```

Figure 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* $\{0, 1, 2\}$.

It is Note 6 that is captured in the title of this paper.

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 [7, 2, 8, 5].

Note 7 *The algorithm in Fig. 2 is implemented by the more traditional MATLAB code given in Fig. 3. 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 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*[1]*. The converse is considerably more difficult.*

[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.

3 The FLAME@lab Interface for Linear Algebra Algorithms

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.

3.1 Bidimensional partitionings

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

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

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` $\times$ `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 *Note that 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 other FLAME APIs [14] 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

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

```
[ 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' )
```

```
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;
```

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

```
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;
```

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

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 A_{11} 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` and `nb` submatrix `A11` is split from submatrix `ATL`, `ATR`, `ABL`, or `ABR`, respectively.

Thus

Repartition

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

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 identify those regions when partitioning.

Once the contents of the so-identified submatrices 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. 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 A_{11} 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 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.

3.2 Horizontal partitionings

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

```
[ 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` indicates the row dimension of A_T or A_B, 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 A_1 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 A_1, with `mb` rows, is partitioned from A_T or A_B, 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 A_1 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. 4 and 5.

3.3 Vertical partitionings

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

```
 [ 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 A_1 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 A_1 is added to the side indicated by `side`.

4 From FLAME@lab to FLAME/C to PLAPACK

Those familiar with our paper on the FLAME/C API [14] will have noticed the similarity between that paper and the present paper. This was, of course, a conscious decision since it emphasizes the fact that the FLAME@lab interface is part of a path for the development of high-performance sequential and parallel linear algebra libraries.

In Fig. 6 we further illustrate how the algorithms implemented using FLAME@lab can be subsequently translated into C using the FLAME/C API. It shows an implementation of the unblocked TRSM algorithm in Fig. 1 using the FLAME/C API. Notice the similarity between the algorithm that employs the FLAME@lab API in Fig. 4 and the one that uses the FLAME/C API.

Since clearly algorithms can be directly translated to 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 [12], 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 study 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 [9]. 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.

Once derived algorithms have been implemented and investigated 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 Basic Linear Algebra Subprograms (BLAS) [10, 6, 5]. Parallel implementations using PLAPACK can subsequently be created, since a similar API has been created for that environment.

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 Conclusion

In this paper, we have presented a simple API for implementing linear algebra algorithms using MATLAB. In isolation, the FLAME@lab interface illustrates 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 API can be used to implement algorithms derived using that methodology so that the proven correctness of those algorithms translates to a high degree of confidence in the implementation.

We want to emphasize that the presented API is merely a very simple one that illustrates 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 interface can be found at the URI given below.

Further Information

For further information, visit `http://www.cs.utexas.edu/users/flame/FLAME@lab/`. At that site, we also give a number of examples.

Acknowledgments

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: Mark Hinga, Dr. Margaret Myers, Vinod Valsalam, and Thierry Joffrain.

```
#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 );
  }
}
```

Figure 6: FLAME implementation of unblocked TRSM algorithm in Fig. 1 using the FLAME/C API.

- 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

[1] Philip Alpatov, Greg Baker, Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, Robert van de Geijn, and Yuan-Jye J. Wu. PLAPACK: Parallel linear algebra package – design overview. In *Proceedings of SC97*, 1997.

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

[3] Greg Baker, John Gunnels, Greg Morrow, Beatrice Riviere, and Robert van de Geijn. PLAPACK: High performance through high level abstraction. In *Proceedings of ICCP98*, 1998.

[4] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, and R. A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Soft.* Submitted.

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

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

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

[8] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Trans. Math. Soft.*, 27(4):422–455, December 2001.

[9] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

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

[11] C. Moler, J. Little, and S. Bangert. *Pro-Matlab, User's Guide.* The Mathworks, Inc., 1987.

[12] E. S. Quintana-Ortí and R. A. van de Geijn. Formal derivation of algorithms for the triangular Sylvester equation. *ACM Trans. Math. Soft.* To appear.

[13] G. W. Stewart. *Introduction to Matrix Computations.* Academic Press, Orlando, Florida, 1973.

[14] R. A. van de Geijn. Representing Linear Algebra algorithms in code: The FLAME API. *ACM Trans. Math. Soft.* Submitted.

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