An API for Manipulating Matrices Stored by Blocks *

Tze Meng Low
Robert A. van de Geijn

Department of Computer Sciences
The University of Texas at Austin
1 University Station, C0500
Austin, TX 78712
{ltm,rvdg}@cs.utexas.edu

FLAME Working Note #12

May 11, 2004

Abstract

We discuss an API that simplifies the specification of a data structure for storing hierarchical matrices (matrices stored recursively by blocks) and the manipulation of such matrices. This work addresses a recent demand for libraries that support such storage of matrices for performance reasons. We believe a move towards such libraries has been slow largely because of the difficulty that has been encountered when implementing them using more traditional coding styles. The impact on ease of coding and performance is demonstrated in examples and experiments. The applicability of the approach for sparse matrices is also discussed.

1 Introduction

We propose an Application Programming Interface (API) for expressing and manipulating hierarchical matrices (matrices that have been stored recursively, by blocks). Specifically, this API simplifies the difficult index manipulation that is required when these more complex schemes are used to store matrices. We show that not only can code be written in a relatively straightforward manner, but the approach supports the higher performance that is the driving force behind the interest in these storage schemes.

With the advent of processors with multiple layers of cache, a number of projects have started to re-examine how matrices should be stored in memory (a thorough review of these projects can be found in [10]). The primary goal is to improve performance of basic linear algebra kernels like the level-3 Basic Linear Algebra Subprograms (BLAS) [9], a set of matrix-matrix operations that perform $O(n^3)$ computations on $O(n^2)$ data, as well as higher level linear algebra libraries like LAPACK [2]. The idea is that by storing blocks at different levels of granularity packed in memory, costly memory-to-memory copies and/or transpositions that are required to provide contiguous access when accessing memory and/or reduce cache and TLB misses can be avoided. While conceptually the proposed solutions are simple and often elegant, complex indexing has so far prevented general acceptance. An additional complication comes from the fact that the filling of such data structures tends to put a considerable indexing burden on the application. Since the literature in this area was recently reviewed in great detail [10], we will not go into great detail here.

*This work was supported in part by NSF contracts ACI-0305163 and CCF-0342369 and an equipment donation from Hewlett-Packard.

The simple observation that underlies our approach is that storage by resursive blocks is typically explained as a tree structure with matrix blocks that are stored contiguously as leaves, and inductively blocks of matrix blocks at each other level of the tree. Thus, a data structure that reflects this tree and an API that obeys this tree is perhaps the more natural way of expressing what we will refer to as hierarchical matrices, and of manipulating such matrices. Similarly, algorithms over these trees are expressed as recursive algorithms. Thus, an API for implementing recursive algorithms that obey this tree seems a natural solution.

The primary contribution of this paper lies with the demonstration that by raising the level of abstraction at which we think about hierarchical matrices and the way they are stored, we simplify the code which implements algorithms that perform operations on such matrices. Notice that we are *not* proposing a standard for the API and resulting library. Much will need to be learned about this new approach to coding before standardization is in order. Neither are we claiming optimal performance. Rather, we are illustrating with a concrete example the issues and benefits of abandoning traditional approaches to coding linear algebra libraries and the possible performance benefits that motivate this new approach.

The remainder of this paper is organized as follows: Section 2 reviews the notation and background related to hierarchical matrices. An API for describing hierarchical matrices and coding algorithms is given in Section 3. Experiments are described in Section 4. Conclusions are given in Section 5. In that section, we also discuss future directions. Performance results are given in Appendix A. An implementation of algorithms for the solution of the triangular Sylvester equation is given in Appendix B.

2 Notation and Background

We first introduce some basic notation and review fundamental insights that motivate the recent shift towards storage by blocks.

2.1 Notation

In this paper, we use the common convention that matrices are denoted by capital letters, vectors by lower case letters, and scalars by Greek letters. Given a matrix X, the functions $m(X)$ and $n(X)$ return the row and column dimensions of X, respectively.

Consider the operation $C = AB + C$. It must be the case that $m(C) = m(A)$, $n(C) = n(B)$, and $n(A) = m(B)$. The elements of C, $\{\gamma_{ij}\}$, are updated with the elements of A, $\{\alpha_{ip}\}$, and B, $\{\beta_{pj}\}$ by the algorithm

$$
\begin{array}{l}
\text{for } i = 1 : m(C) \\
\quad \text{for } j = 1 : n(C) \\
\quad\quad \text{for } p = 1 : n(A) \\
\quad\quad\quad \gamma_{ij} = \gamma_{ij} + \alpha_{ip}\beta_{pj} \\
\quad\quad \text{endfor} \\
\quad \text{endfor} \\
\text{endfor}
\end{array}
$$

Notice that the order of the loops can be arranged in $3! = 6$ different ways.

It is common in the description of high-performance algorithms to partition matrices into blocks of matrices:

$$
C = \left(\begin{array}{c|c|c|c}
C_{11} & C_{12} & \cdots & C_{1N} \\ \hline
C_{21} & C_{22} & \cdots & C_{2N} \\ \hline
\vdots & \vdots & & \vdots \\ \hline
C_{M1} & C_{M2} & \cdots & C_{MN}
\end{array} \right), \quad
A = \left(\begin{array}{c|c|c|c}
A_{11} & A_{12} & \cdots & A_{1K} \\ \hline
A_{21} & A_{22} & \cdots & A_{2K} \\ \hline
\vdots & \vdots & & \vdots \\ \hline
A_{M1} & A_{M2} & \cdots & A_{MK}
\end{array} \right), \quad \text{and} \quad
B = \left(\begin{array}{c|c|c|c}
B_{11} & B_{11} & \cdots & B_{1N} \\ \hline
B_{21} & B_{21} & \cdots & B_{2N} \\ \hline
\vdots & \vdots & & \vdots \\ \hline
B_{K1} & B_{K1} & \cdots & B_{KN}
\end{array} \right),
$$

where $m(C_{ij}) = m(A_{ip})$, $n(C_{ij}) = n(B_{pj})$, $n(A_{ip}) = m(B_{pj})$. Then matrix-matrix multiplication, in terms of these blocks, can be implemented by the algorithm

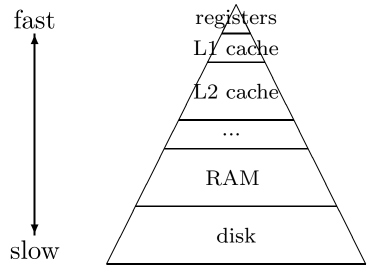

Figure 1: The hierarchical memories viewed as a pyramid.

$$
\begin{array}{l}
\text{for } i = 1 : M \\
\quad \text{for } j = 1 : N \\
\quad\quad \text{for } p = 1 : K \\
\quad\quad\quad C_{ij} = C_{ij} + A_{ip} B_{pj} \\
\quad\quad \text{endfor} \\
\quad \text{endfor} \\
\text{endfor}
\end{array}
$$

Again, the order of the loops can be arranged in $3! = 6$ different ways.

Multiple layers of blocking can be imposed upon matrices by further partitioning the blocks C_{ij}, A_{ip}, and B_{pj} into finer blocks.

2.2 Architectures with multi-level memory

With the advent of cache-based processors, a major concern became how to amortize the cost of moving data between memory layers. Let us consider a typical multi-level memory as depicted in Fig. 1. Inherently, data must be moved from memory layers lower in this pyramid into, ultimately, the registers in order for the CPU to perform floating point operations on those data. Since such movement represents unproductive overhead, it becomes important to reuse data that is up the memory pyramid many times before allowing it to slide back down.

One of the most ideal operations that allows such amortization of data movement is the matrix-matrix multiplication. In the most simple terms, this operation performs $O(n^3)$ operations on $O(n^2)$ data. Since this operation can be decomposed into smaller blocks that fit a given target memory layer, there is the clear opportunity of amortizing the cost of the data movement over many operations. It has been shown that most dense linear algebra operations can be cast in terms of matrix-matrix multiplication, which then allows portable high performance to be achieved.

2.3 High-performance implementation of matrix-matrix multiplication

Let us briefly review the anatomy of a typical high-performance implementation of matrix-matrix multiplication.

The fastest implementations [1, 12, 15, 23, 6] of matrix-matrix multiplication $C = AB + C$ use algorithms similar to:

$$
\begin{array}{l}
\text{for } p = 1, \ldots, K \\
\quad \text{for } i = 1, \ldots, M \\
\quad\quad T = A_{ip}^T \\
\quad\quad \text{for } j = 1, \ldots, M \\
\quad\quad\quad C_{ij} = T^T B_{pj} + C_{ij} \\
\quad\quad \text{endfor} \\
\quad \text{endfor} \\
\text{endfor}
\end{array}
$$

We recognize this as a simple permutation of the loops. Details of why the loops are structured in this specific order go beyond the scope of this paper. Also, when matrices are stored by columns (column-major order) as in Fortran, in order to reduce TLB misses and other unwanted cache behavior, A_{ip} is typically transposed and copied into a contiguous buffer, T. This reduces the number of TLB entries required to address this block and allows inner-products of columns of T and B_{pj} to be used, which access memory mostly contiguously, for the computation of elements of C_{ij}. The block size of A_{ip} is chosen so that one such submatrix fills most of the L2 cache (unless the amount memory addressible by the TLB is less that the size of the L2 cache, in which case this becomes the limiting factor)[1]. Since often $C = A^T B + C$, $C = AB^T + C$, and/or $C = A^T B^T + C$ are also encountered in applications, blocks of B and/or C may also need to be packed and/or transposed.

The simple observation now is that *if* matrices are stored by blocks, then this packing is no longer required. Morever, within this storage scheme, transposition can be made cheaper, for example by transposing in-place. A second observation is that with the introduction of more and more memory layers, multiple levels of blocking may become required in order to optimally implement matrix-matrix multiplication [15].

2.4 Notation related to hierarchical matrices

In order to facilitate discussion, we introduce the following definitions to describe hierarchically partitioned matrices:

- A *flat matrix* is a logical representation of how a matrix would look if we were to write it out on a piece of paper. It is a storage and partition independent idea of a matrix, a mathematical description.
- The *depth* of a matrix, X, is the maximum number of times the flat matrix of X has been partitioned into smaller sub-matrices. A non-partitioned matrix has a depth of 0. Matrices that have been partitioned into sub-matrices would have a depth of 1 plus the maximum of the depths of its sub-matrices. We denote the depth of X as $d(X)$.
- *Leaf matrices* are matrices whose elements are scalars and are not further partitioned into smaller sub-matrices. The depth of a leaf matrix is 0.
- A *hierarchical matrix*, X, is a either a leaf matrix or a matrix such that all its elements are all hierarchical matrices of depth at most one less than the depth of X.

2.5 Necessary conditions for operations on hierarchical matrices

Consider the operation $C = AB + C$ where all three matrices are hierarchical matrices. In order for the multiplication to be conveniently implemented as a recursive algorithms in a level-by-level fashion, certain conditions must be imposed on the dimensions of level matrices that must be multiplied as subproblems. We do not here give a full discussion of this (for space reasons). Rather, we will restrict ourselves to the case where matrices are hierarchically partitioned in a conformal manner, without formally specifying what this means. Clearly, in future papers on the subject, this will need to be more precisely treated.

3 An API for Storing and Manipulating Hierarchical Matrices

Many papers describe the concept of hierarchically stored matrices, as mentioned in the introduction. What is absent from these papers tends to be any example of how to instantiate such data structures in code and how to code operations over such data structures. One exception is those efforts that utilize the ability of C++ to hide indexing details [21]. Even there, much complexity is required under the covers in order to index into the data structures. In this section, we show how an API that mirrors the way hierarchical matrices are naturally explained allows many of these indexing intricacies to disappear.

[1] In some implementations, A_{ip} is chosen to fill most of the L1 cache instead.

We will next show that a very simple approach allows the specifics of the mapping of data memory locations to be hidden, keeping the indexing problems to a minimum. We do so by discussing a simple extension to the FLAME API for the C programming language, FLAME/C, as a concrete example of an instantiation of this idea [14, 4, 5].

3.1 Describing matrices in FLAME/C

The FLAME/C API hides details like storage method and matrix dimensions from the user by utilizing *object based programming* techniques much like the Message-Passing Interface (MPI) and other software packages [13, 20, 3, 22]. These details are stored in a descriptor and hidden from the programmer. They can be accessed through inquiry routines. In its current implementation, it is assumed that all matrices are stored using column-major storage as prescribed by Fortran.

FLAME/C provides the following call to create a descriptor and storage for an $m \times n$ matrix of type `double`, where integers `m` and `n` indicate the matrix row and column dimensions, respectively:

```
FLA_Obj C;

FLA_Obj_create( FLA_DOUBLE, m, n, &C );
```

Given that now `C` is a descriptor for a FLAME object (declared to be of type `FLA_Obj`), the following calls then extract the various attributes:

```
datatype_C = FLA_Obj_datatype( C );
m_C        = FLA_Obj_length( C );
n_C        = FLA_Obj_width ( C );
ldim_C     = FLA_Obj_ldim  ( C );
buf_C      = ( double * ) FLA_Obj_buffer ( C );
```

Notice that the last call will set `buf_C` to the address of where the elements of the matrix are stored, so that the (i, j) element can be set by the assignment

```
buf_C[ j*ldim_C + i ] = < value of (i,j) element >
```

3.2 Supporting hierarchical matrices in FLAME/C

The logical extension to the above API call that will allow the support of hierarchical matrices is to allow each element of a matrix created by `FLA_Obj_create` to itself be a matrix. Thus the call

```
FLA_Obj_create( FLA_MATRIX, m, n, &C );
```

creates an $m \times n$ matrix where each element of the C is a descriptor for a (not yet created) matrix C_{ij}. We will refer to the FLAME/C API augmented with this as FLASH: Formal Linear Algebra Scalable Hierarchical API.

Given arrays `ms` and `ns` where each element, C_{ij}, is to be of size `ms[i]` by `ns[j]`, the following loop then creates the descriptors for the elements.

```
ldim_C = FLA_Obj_ldim( C );
buf_C = ( FLA_Obj * ) FLA_Obj_buffer( C );

for (j=0; j<n; j++ )
  for ( i=0; i<m; i++ )
    /* Create a matrix C_ij of size m[ i ] x n[ j ] */
    FLA_Obj_create( FLA_DOUBLE, ms[ i ], ns[ j ],
                    &buf_C[ j*lda_C + i ] );
```

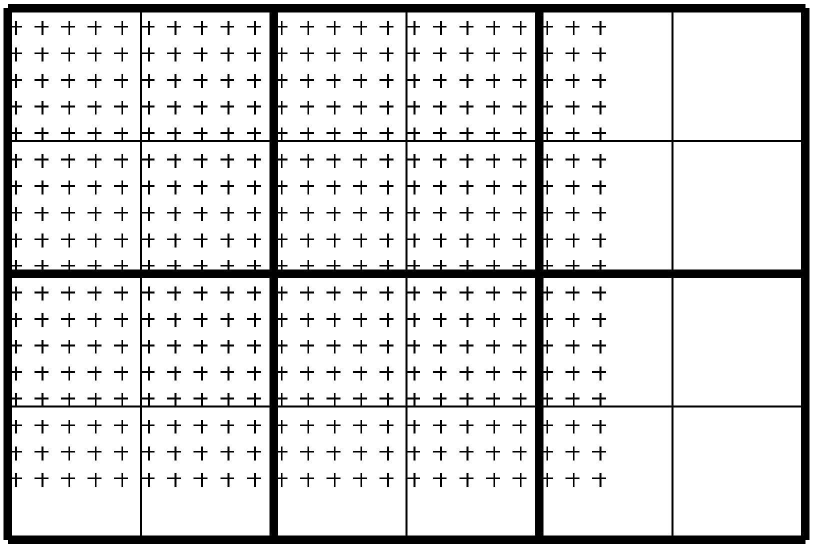

Figure 2: Graphical representation of hierachy using "seives".

Naturally, by taking some or all submatrices C_{ij} of type FLA_MATRIX, further layers of a hierarchical matrix can be created.

It is beneficial to have one subroutine call that creates all the layers of the hierarchy. More importantly, having a single subroutine call to create the hierarchy allows us to ensure that the hierarchy created will possess the properties related to conformity to which we allude in Section 2.5. One such hierarchy is created by the call

```
FLASH_Obj_create( int datatype, int order, int m, int n, int depth,
                  int *Blksizes, FLA_Obj &C );
```

This call requires considerable explanation: The datatype of the elements in the leaf matrices is given by datatype. The order in which blocks are ordered is given by order. The flat matrix to be created is to be of size $m \times n$, where m and n are passed as arguments m and n, respectively. The argument depth takes the value of $d(C)$ where C is the hierarchical matrix to be created. Blksizes takes as input an array of integers that indicates the size of the matrix at each level.

A more intuitive way of thinking about creating the hierarchy may be to think of partitioning the matrix with increasingly finer seives while maintaining the constraint that the size of the current seive must be a multiple of the size of the next smaller seive. Figure 2 is a diagram of how the hierarchy is created with "seives. That figure illustrates a hierarchical matrix created with the call

```
FLASH_Obj_create( datatype, FLASH_GENERALIZED_MORTON, 18, 23, 2, Blksizes, &C )
```

where

```
int Blksizes[ 2 ] = { 2, 5 };
```

In this particular example, the top matrix is a $\lceil 18/(2 \times 5) \rceil \times \lceil 23/(2 \times 5) \rceil = 2 \times 3$ matrix of matrices. Each of those matrices is a 2×2 matrix, except the last column. At the leaves are matrices of dimension (at most) 5×5. This description is captured in a possible implementation for FLASH_Obj_create given in Fig. 4.

Note: In our above explanation we do not "pad" the matrix to become of a dimension that is a multiple of the largest seive. However, the interface doesn't necessarily prescribe the implementation. Additional memory can always be used to pad the matrices if there is some advantage to doing so.

3.3 Storage schemes

Having described how to build the hierarchy, we, next, need to consider how to store the elements of the matrix. Here what really matters is the order in which the data in the leaf matrices is stored and the order

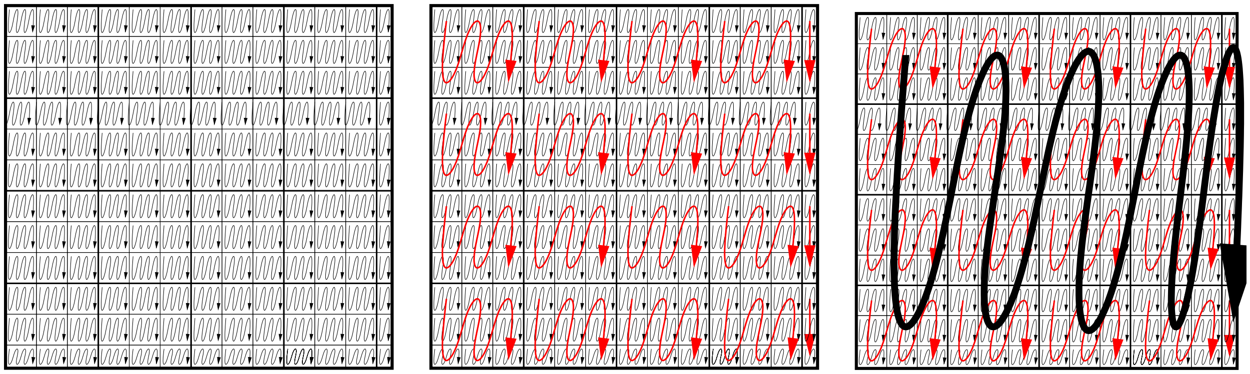

Figure 3: Generalized Morton order: The lowest level blocks are 5×5 matrices stored in column-major order. The next level consists of 3×3 blocked matrices, stored in column-major order. Finally, those blocks are stored in column-major order themselves. Notice that the traditional Morton order can be achieved by picking the blocking at each level to be 2×2 for $\log_2(n)$ levels, storing each level in row-major order.

in which these blocks are stored relative to each other. Street wisdom is that the better the locality of the data at the various levels, the better different layers of the memory hierarchy can be used.

3.3.1 Recursive Block Storage - Morton(Z) Ordering

A Morton (Z) ordering [11, 21] (with blocks at the leaves) can be achieved as follows:

- Store the $b \times b$ leave matrices in some format, where the b^2 elements are stored in contiguous memory.
- Form 2×2 matrices of such blocks at the next level, storing these blocks in contiguous memory ordering them in row-major order.
- Continue like this, building 2×2 matrices at each level.

The approach illustrated in Fig. 3 is a variant of this. Given an $n \times n$ matrix, it can be described as follows:

- An $n \times n$ item buffer of continuous memory is allocated.
- Given a set of blocking sizes $\{b_0, b_1, \ldots, b_{L-1}\}$, the flat matrix is partitioned into $B = \prod b_i$ blocks.
- Assuming for simplicity that n is an integer multiple of B, these top-level blocks are assigned $B \times B$ items of contiguous memory, in a column-major order.
- Each such $B \times B$ block is now further subdivided using blocking sizes $\{b_0, b_1, \ldots, b_{L-2}\}$.
- Leave matrices, of size $b_0 \times b_0$, are stored in column-major order.

It is not hard to see that this is modification of the Morton (Z) ordering, in that it uses column-major ordering and at each level the blocked matrices need not to be 2×2.

3.3.2 Canonical Storage

In order to compare with current existing means of storing matrices in row or column major order, we have also included the following calls which allow us to attach a conventional column major ordered matrix to a existing hierarchy. This creates a hierarchy in which the leaf matrices act as views into different parts of the original matrices.

```
FLASH_Obj_create_without_buffer( datatype, m, n, depth, *Blksizes,  &C );
FLASH_Obj_attach_buffer( *buf, ldim, &C );
```

```
#include "FLAME.h"
#define Cij(i,j)  C[ (j)*ldim_C + (i) ]

void FLASH_Obj_create( int datatype, int order, int m, int n,
                       int depth, int *Blksizes, FLA_Obj *C )
{
  char *buffer;     int typesize;

  typesize = FLA_Type_size( datatype );
  buffer = ( char * ) malloc( m * n * typesize );

  switch( order ){
  case FLASH_GENERALIZED_MORTON:
    FLA_Obj_create_gen_Morton( datatype, m, n, depth, Blksizes, C, buffer );
    break;
  default:
    FLA_Abort( "Order not yet implemented", __LINE__, __FILE__ );
  }
}

void FLASH_Obj_create_gen_Morton( int datatype, int m, int n,
                   int depth, int *Blksizes, FLA_Obj *C, char *buffer )
{
  int b, i, ii, j, jj, m_cur, n_cur, mb, nb, typesize;

  if ( depth == 0 ){    /* leaf matrix */
    FLA_Obj_create_without_buffer( datatype, m, n, C );
    FLA_Obj_attach_buffer( buffer, C );
  }
  else{
    typesize = FLA_Type_size( datatype );
                     /* Extract current seive size = Blksizes[0] * Blksizes[1] * ...  */
    b = 1;
    for ( i=0; i<depth; i++ ) b *= Blksizes[ i ];
                              /* Determine row and column dimension of level matrix  */
    m_cur = m / b + ( m % b == 0 ? 0 : 1 );  /* m_cur = ceil( m / b ) */
    n_cur = n / b + ( n % b == 0 ? 0 : 1 );  /* n_cur = ceil( n / b ) */
                              /* Create a level matrix and extract leading dimension */
    FLA_Obj_create( FLA_MATRIX, m_cur, n_cur, C );
    ldim_C = FLA_Obj_ldim( *C );
                              /* Loop over the columns of the level matrix           */
    j = 0;
    for ( jj=0; jj<n; jj+=b ){
                              /* nb = column size of current hierarchical submatrix  */
      nb = min( n-jj, b );
                              /* Loop over the rows of the level matrix              */
      i = 0;
      for ( ii=0; ii<m; ii+=b ){
        mb = min( m-ii, b );        /* column size of current hierarchical submatrix */
                              /* create (hierarchical) matrix i,j                    */
        FLASH_Obj_create_gen_Morton( datatype, mb, nb,
                       depth-1, &Blksizes[ 1 ], &Cij( i,j ), buffer );
        buffer += mb * nb * typesize;
        i++;
      }
      j++;
    }
  }
}
```

Figure 4: Sample implementation of FLASH_Obj_create.

More precisely, the flat matrix is stored in column major order. At each level, the submatrices are simply submatrices of this matrix. The net effect is that the leaf matrices are stored in column-major order, with a leading dimension that equals the leading dimension of the flat matrix.

3.4 Filling the matrix

An easy argument for *not* accepting alternative data structures is to point out that filling such matrices with data is simply too complex for applications. This would appear to favor the use of C++ since it would allow indexing to be hidden from the user. We propose an alternative solution.

Notice that for small matrices, it is always possible to create the matrix using a more traditional storage, and to then rearrange the data into the our alternative approach. For larger matrices, this is not acceptable, since it would require a considerable amount of additional space (unless it can be done in-place, which would become quite complex.) Fortunately, very large dense matrices do not appear often in applications. More importantly, frequently matrices are conveniently created as smaller submatrices that represent contributions to the total (global) matrix. Thus, we propose the following call:

```
FLASH_Axpy_matrix_to_global( m, n, alpha, B, ldb, A, i, j );
```

This call will add $m \times n$ matrix B, scaled by α, to the submatrix of A that has as its top-left element the (i, j) element of A. In (MATLAB) Mscript notation this operation is given by

```
A( i:i+m-1, j:j+n-1 ) = alpha * B  + A( i:i+m-1, j:j+n-1 );
```

A similar routine allows submatrices to be extracted. Notice that now the user needs not know how the matrix A is actually stored.

This approach has be highly successful for interface applications to our Parallel Linear Algebra Package (PLAPACK) library for distributed memory architectures [22] where filling distributed matrices poses a similar challenge.

3.5 FLAME Algorithms for Hierarchical Matrices

An example of how simple code becomes when matrices are stored hierarchical using the proposed API, consider the matrix-matrix multiplicaiton presented in Fig. 5. Notice that apart from providing an API that reduces the complexity arising from indexing, FLAME provides a methodology for deriving dense linear algebra algorithms. Using the same methodology, algorithms that operate on hierarchical matrices can also be systematically derived. In addition, we advocate a coding style that avoids intricate indexing, unlike the code shown in Fig. 5 (which we present as nested loops for those not familiar with FLAME, and for conciseness). Unfortunately further details go beyond the scope of this paper.

4 Experiments

In this section, we concentrate on the ease of programming available when an API like the one we propose is adopted. We also *briefly* discuss performance. However, we note that *no conclusions* should yet be drawn from the performance experiments. So as not to distract from the main focus of the paper, we report performance in Appendix A.

A nontrivial operation for which high-performance implementation that has been studied over the last few years is the triangular Sylvester equation

$$UX + XR = C$$

where U and R are (square) uppertriangular matrices and X and C are rectangular matrices. Notice that if U is $m \times m$ and R is $n \times n$, then C and X are both $m \times n$. In this equation, U, R, and C are inputs, and X is an output. It is customary that the solution X overwrites input matrix C.

A few recent papers discuss algorithms for computing the solution of this equation, as well as their implementation. In [18], we show how a family of algorithms can be derived using the FLAME approach to deriving algorithms. Also in that paper, we show how the FLAME/C API can be used to implement them. While the derived algorithms are loop-based, the paper also points out how recursion and loops (iteration) can be combined. In a more recent paper [10], a recursive algorithm (also discussed in [16]) is discussed. In that paper, the authors also discuss the benefits of storage by blocks, but do not show implementations using this alternative mapping of matrices to memory.

To evaluate how easily the proposed API can facilitate the implementation of these kinds of algorithms, we coded both the recursive algorithm in [10] and the best algorithm (which combines iteration and recursion) in [18] (Algorithm C3 in that paper). In a matter of less than an hour, both these algorithms were implemented and tested for correctness. We use this as an indication that the API facilitates rapid prototyping of recursive algorithms for matrices that are stored as hierarchical matrices.

It is worth noting that in addition to the implementations of matrix-matrix multiplication and the solution of the triangular Sylvester equation, the proposed API has also been used to implement a number of level-3 BLAS operations. It is our belief that most of the functionality of the BLAS and LAPACK can be supported with the proposed techniques and API.

5 Conclusion and Future Work

In this paper, we have shown that by storing hierarchical matrices and utilizing an API that can express them as matrices of matrices greatly simplifies the more complex indexing. We believe this overcomes a major obstacle to widespread acceptance of these alternative approaches to storage.

Much work remains. Most research in this area has focused on the efficient implementation of the kernels that allow high performance to be attained. **Our work now needs to be combined with the results of that research in order to demonstrate the performance benefits** that will come from abandoning

```
#include "FLAME.h"

#define AA(i,j) buf_A[ (j)*ldim_A+(i) ]
#define BB(i,j) buf_B[ (j)*ldim_B+(i) ]
#define CC(i,j) buf_C[ (j)*ldim_C+(i) ]

void FLASH_AB_plus_C( FLA_Obj A, FLA_Obj B, FLA_Obj C )
/* Compute C = A * B + C
   Simplifying assumption: all matrices have the same depth and are stored conformally. */
{
  int m, n, k, ldim_A, ldim_B, ldim_C, i, j, p, datatype;
  FLA_Obj *buf_A, *buf_B, *buf_C;

  datatype = FLA_Obj_datatype( A );

  if ( datatype != FLA_MATRIX )  /* A, B, and C, are all leaf matrices */
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, ONE, A, B, ONE, C );
  else {
    buf_C = ( FLA_Obj * ) FLA_Obj_buffer( C ); ldim_C = FLA_Obj_ldim( C ); m = FLA_Obj_length( C );
    buf_A = ( FLA_Obj * ) FLA_Obj_buffer( A ); ldim_A = FLA_Obj_ldim( A ); n = FLA_Obj_length( A );
    buf_B = ( FLA_Obj * ) FLA_Obj_buffer( B ); ldim_B = FLA_Obj_ldim( B ); k = FLA_Obj_length( B );

    for ( p=0; p<k; p++ )
      for ( j=0; j<n; j++ )
        for ( i=0; i<m; i++ )
          FLASH_AB_plus_C( AA( p,i ), BB( p,j ), CC( i,j ) );
  }
}
```

Figure 5: Matrix-matrix multiplication implemented using FLAME/C with hierarchical matrices.

traditional storage methods for matrices like column-major storage used by linear algebra packages like LAPACK [2].

Perhaps an even greater impact will come from exploiting the API for problems that give rise to sparse linear systems. For most engineering applications, the degrees of freedom are inherently hierarchically organized as domains and subdomains, which is often referred to as successive substructuring [17]. The matrices that capture the coupling between these hierarchically organized components are themselves hierarchical. Our API allows these kinds of matrices to be naturally captured as hierarchical matrices that have pruned branches (where submatrices at some level are tagged as containing only zeroes rather than explicitly stored). By writing so-called sparse direct methods as dense direct methods, with operations on pruned branch matrices being detected and pruned at the time of the computation, dense linear algebra libraries could operate efficiently on sparse problems. Whenever fill-in occurs, branches can be added to the hierarchy. Other structural and numerical properties of branches could similarly be expressed and exploited. This vision can be made reality with minor extensions to our proposed API.

A simple application of this last observation lies with the compact storage of triangular, symmetric, and/or banded matrices, a topic of recent research [10]. While others propose complex storage schemes that take, for example, triangular matrices and store them in a single dense array by partitioning and folding, our approach would allow blocks that are known to contain only zeroes to be marked and not stored. While the storage is not as compact as other approaches, it attains almost all of the benefits with while simplifying the coding effort tremendously. This idea was already explored for block tridiagonal matrices in a parallel implementation of the Cholesky factorization [7]. That paper stored the blocks on the diagonal and subdiagonal as vectors, without an API to elegantly support the idea, but illustrates the benefits.

We conclude by pointing out that the idea of storing matrices as matrices of matrices was already explored in the early 1970s [19], and later revisited for distribution of matrices to distributed memory architectures [8]. It is an old idea that perhaps should have become the norm long ago.

Additional Information

Please visit `www.cs.utexas.edu/users/flame/`.

Acknowledgments

We would like to thank Drs. John Gunnels and Enrique Quintana for comments on early drafts of this paper.

References

[1] R. C. Agarwal, F. Gustavson, and M. Zubair. Exploiting functional parallelism on Power2 to design high-performance numerical algorithms. *IBM Journal of Research and Development*, 38(5):563–576, 1994.

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

[4] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Soft.* to appear.

[5] Paolo Bientinesi, Enrique S. Quintana-Orti, and Robert A. van de Geijn. Representing linear algebra algorithms in code: The FLAME APIs. *ACM Trans. Math. Soft.* To appear.

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

[7] Thuan D. Cao, John F. Hall, and Robert A. van de Geijn. Parallel cholesky factorization of a block tridiagonal matrix. In *In Proceedings of the International Conference on Parallel Processing 2002 (ICPP-02)*, August 2002.

[8] Timothy Collins and James C. Browne. Matrix++: An objectoriented environment for parallel high-perfomance matrix computations. In *Proc. of the Hawaii Intl. Conf. on Systems and Software*, 1995.

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

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

[11] J.D. Frens and D.S. Wise. Auto-blocking matrix-multiplication or tracking blas3 performance from source code. In *Proceedings of 6th ACM Symp. on Principle and Practice of Parallel Programming, Las Vegas*, pages pp 206–216, 1997.

[12] Kazushige Goto and Robert A. van de Geijn. On reducing TLB misses in matrix multiplication. Technical Report CS-TR-02-55, Department of Computer Sciences, The University of Texas at Austin, 2002.

[13] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. The MIT Press, 1994.

[14] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Trans. Math. Soft.*, 27(4):422–455, December 2001.

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

[16] I. Jonsson and B. Kågström. Recursive blocked algorithms for solving triangular matrix equations - part I: One-sided and coupled Sylvester equations. Department of Computing Science and HPC2N Report UMINF-01.05, Umeå University, 2001.

[17] A. Patra and J. T. Oden. Computational techniques for adaptive *hp* finite element methods. *Finite Elements in Analysis and Design*, 25:27–39, 1997.

[18] Enrique S. Quintana-Ortí and Robert A. van de Geijn. Formal derivation of algorithms: The triangular Sylvester equation. *ACM Trans. Math. Soft.*, 29(2):218–243, June 2003.

[19] G.M. Skagestein. *Rekursiv Unterteilte Matrizen Sowie Methoden Zur Erstellung Von Rechnerprogrammen Fur Ihre Verarbeitung*. PhD thesis, Universitat Stuttgart, 1972.

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

[21] Vinod Valsalam and Anthony Skjellum. A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. Concurrency and Computation: Practice and Experience, 2002.

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

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

Intel Pentium 4 (2.4 GHz)	Intel Itanium2 (900 MHz)

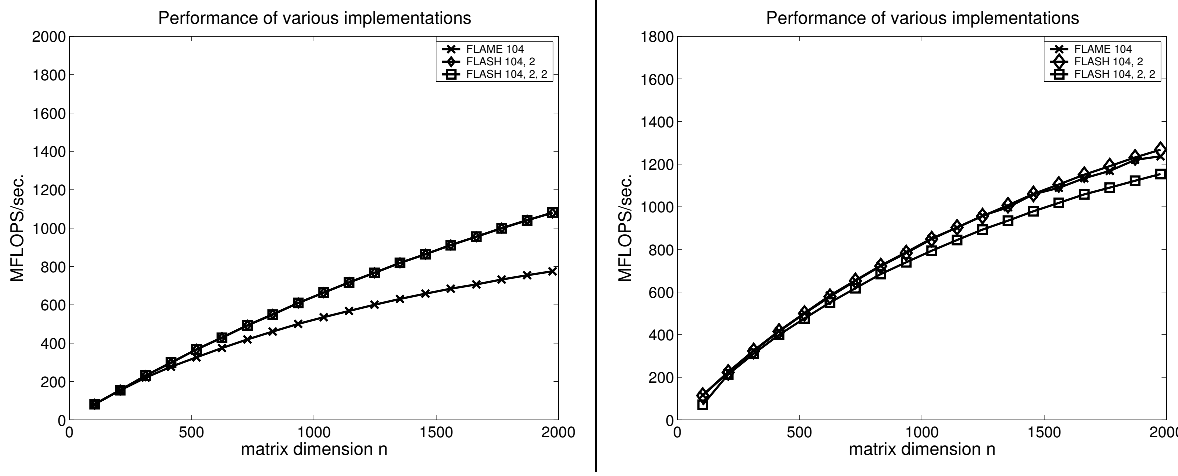

Figure 6: Performance of the various implementations. Notice that the scale of the y-axis is such that the top of the graph corresponds to **50%** of peak performance on the respective machines. Unlike my other linear algebra operations, near-peak performance is difficult to attain for the solution of the triangular Sylvester equation since not as much of the computation can be cast in terms of large matrix-matrix multiplications [18]. We, again, warn the reader not to attach too much value to the qualitative or quantitative differences (or, more importantly, the lack thereof) that are observed: until inner kernels are rewritten to take advantage of the new mappings, performance results are not indicative of the benefits of the new mappings.

A Preliminary Performance Results

We re-emphasize that this is *not* a paper about performance: Much work is still required before conclusions can be drawn regarding the performance impact of storing matrices hierarchically by blocks. In particular, when matrices are stored by blocks, the kernels that perform matrix-matrix multiplication once data reaches the level 1 and 2 caches will likely need to be rewritten. The conclusion that we believe *can* be drawn from the results of the experiments that we report is that the API, *when combined with research on how to implement low level kernels (not discussed in this paper)*, can be used to evaluate the (possible) benefits of storing matrices as hierchical matrices.

We report performance results on two different platforms: workstations based on the Intel Pentium4 (R) processor (2.4 GHz) and the Intel Itanium2 (R) processor (900 MHz). On both systems, implementations were linked to the BLAS library of Kazushige Goto [12], which are among the fastest available for these architectures.

We report performance for what was found to be the best algorithm in [18] (Algorithm C3). In all our implementations, subproblems of dimension 104 or less were solved with the same unblocked algorithm. Performance for four versions are reported:

- FLAME 104: The algorithm as reported in [18] with matrices stored in the traditional column-major order. Only one level of blocking, with a block size of 104, is employed.
- FLASH 104, 2: This algorithm employs two levels of blocking: 104×104 for the leaf matrices and 208×208 for the next level. The hierarchical matrix is stored using the generalized Morton order with these same dimensions. Thus, the leaf matrices are contiguous in memory, as is the next level of matrices.
- FLASH 104, 2, 2: This algorithm employs three levels of blocking: 104×104 for the leaf matrices and 208×208 for the next level, and 416×416 for the top level. The hierarchical matrix is stored

using the generalized Morton order with these same dimensions.

Performance is reported in Fig. 6. On the Pentium 4 platform, a substantial improvement in performance is observed when the leaf matrices are stored as contigous blocks. Additional coarser levels of blocking do not appear to help. The results are quite different on the Itanium2 platform. There, the benefits from storing the leaf matrices contiguously is not as dramatic, and additional coarser levels appear to adversely affect performance.

One conclusion that perhaps can be drawn from the graphs is that there appears to be merrit to storing leaf matrices contiguous in memory, but managing the storage of those blocks at higher levels does not seem to impact performance substantially. In other words, complex storage schemes that enforce a Morton order at many levels may not warrant the additional complexity.

B Code

In Figs. 7–9 we show the actual FLASH code for the solution of the triangular Sylvester equation.

```
#include "FLAME.h"

int FLASH_TriSylv_colC3(FLA_Obj A, FLA_Obj B, FLA_Obj C){

  FLA_Obj A11, A12,    C1, C2;
          A21, A22;

  FLA_Obj

  FLA_Part_2x2( A,    &A11, &A12,
                      &A21, &A22,     1, 1, FLA_TL );

  FLA_Part_2x1(C, &C1,
                  &C2, 1, FLA_TOP);

  FLASH_TriSylv_C3(A22, B, C2);
  FLASH_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
             MINUS_ONE, A12, C2, ONE, C1);
  FLASH_TriSylv_C3(A11, B, C1);


  return FLA_SUCCESS;
}

int FLASH_TriSylv_rowC3(FLA_Obj A, FLA_Obj B, FLA_Obj C){

  FLA_Obj B11, B12,   C1, C2,
          B21, B22;

  FLA_Part_2x2( B,    &B11, &B12,
                      &B21, &B22,     1, 1, FLA_TL );

  FLA_Part_1x2( C, &C1, &C2, 1, FLA_LEFT);

  FLASH_TriSylv_C3(A, B11, C1);
  FLASH_Gemm(FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
             MINUS_ONE, C1, B12, ONE, C2);
  FLASH_TriSylv_C3(A, B22, C2);

  return FLA_SUCCESS;
}
```

Figure 7: FLASH code for the solution of the triangular Sylvester equation.

```
int FLASH_TriSylv_C3( FLA_Obj A, FLA_Obj B, FLA_Obj C)
{
  FLA_Obj
    ATL, ATR,   A00,  A01,     A02,
    ABL, ABR,   A10,  A11,     A12,
                A20,  A21,     A22,
    BTL, BTR,   B00,  B01,     B02,
    BBL, BBR,   B10,  B11,     B12,
                B20,  B21,     B22,
    CTL, CTR,   C00,  C01,     C02,
    CBL, CBR,   C10,  C11,     C12,
                C20,  C21,     C22;

  if( ( FLA_Obj_width( C ) ==  0 )||( FLA_Obj_length( C ) == 0 ))
    return FLA_SUCCESS;

  if ( FLA_Obj_datatype(C) != FLA_MATRIX ){
    FLA_TriSylv(A, B, C);
    return FLA_SUCCESS;
  }

  if ( FLA_Obj_length(C) == 1 ){
    if (FLA_Obj_width(C) == 1 ) {
      FLASH_TriSylv_C3(*((FLA_Obj*)FLA_Obj_buffer(A)),
                    *((FLA_Obj*)FLA_Obj_buffer(B)),
                    *((FLA_Obj*)FLA_Obj_buffer(C)));
      return FLA_SUCCESS;
    }
  }

  if (FLA_Obj_width(C) > FLA_Obj_length(C)){
    FLASH_TriSylv_rowC3(A, B, C);
    return FLA_SUCCESS;
  }

  if (FLA_Obj_width(C) < FLA_Obj_length(C)){
    FLASH_TriSylv_colC3(A, B, C);
    return FLA_SUCCESS;
  }

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

  FLA_Part_2x2( B,  &BTL, /**/ &BTR,
                 /* ************** */
                   &BBL, /**/ &BBR,
           /* with */ 0, /* by */ 0, /* submatrix */ FLA_TL );

  FLA_Part_2x2( C,  &CTL, /**/ &CTR,
                 /* ************** */
                   &CBL, /**/ &CBR,
           /* with */ 0, /* by */ 0, /* submatrix */ FLA_BL );
```

Figure 8: FLASH code for the solution of the triangular Sylvester equation (continued).

```
  while ( ( FLA_Obj_length( BBR ) != 0 )||( FLA_Obj_length( ATL ) != 0 ) ){
    FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,       &A00,  &A01,    /**/  &A02,
                                /**/            &A10,  &A11,    /**/  &A12,
                        /* ************* */   /* ************************* */
                           ABL, /**/ ABR,       &A20,  &A21,    /**/  &A22,
             /* with */ 1, /* by */ 1, /* A11 split from */ FLA_TL );
    FLA_Repart_2x2_to_3x3( BTL, /**/ BTR,       &B00,  /**/ &B01,     &B02,
                        /* ************* */   /* ************************* */
                                /**/            &B10,  /**/ &B11,     &B12,
                           BBL, /**/ BBR,       &B20,  /**/ &B21,     &B22,
             /* with */ 1, /* by */ 1, /* B11 split from */ FLA_BR );
    FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,       &C00,  /**/ &C01,     &C02,
                                /**/            &C10,  /**/ &C11,     &C12,
                        /* ************* */   /* ************************* */
                           CBL, /**/ CBR,       &C20,  /**/ &C21,     &C22,
             /* with */ 1, /* by */ 1, /* B11 split from */ FLA_TR );
    /* ********************************************************************* */
    /* C10 <- X10 where X10 solves A11 X10 + X10 B00 = C10 */
    FLASH_TriSylv_C3(A11, B00, C10);
    /* C21 <- X21 where X21 solves A22 X21 + X21 B11 = C21 */
    FLASH_TriSylv_C3(A22, B11, C21);
    /* C11 <- - A12 X21 + C11 */
    FLASH_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A12, C21, ONE, C11 );
    /* C11 <- - X10 B01 + C11 */
    FLASH_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, C10, B01, ONE, C11 );
    /* C11 <- X11 where X11 solves A11 X11 + X11 B11 = C11 */
    FLASH_TriSylv_C3(A11, B11, C11);
    /* C00 <- - A01 X10 + C00 */
    FLASH_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A01, C10, ONE, C00 );
    /* C01 <- - A01 X11 + C01 */
    FLASH_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A01, C11, ONE, C01 );
    /* C01 <- - A02 X21 + C01 */
    FLASH_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, A02, C21, ONE, C01 );
    /* C22 <- - X21 B12 + C22 */
    FLASH_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, C21, B12, ONE, C22 );
    /* C12 <- - X10 B02 + C12 */
    FLASH_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, C10, B02, ONE, C12 );
    /* C12 <- - X11 B12 + C12 */
    FLASH_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, MINUS_ONE, C11, B12, ONE, C12 );
    /* ********************************************************************* */
    FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,      A00, /**/ A01,      A02,
                            /* ************** */ /* ************************ */
                                    /**/            A10, /**/ A11,      A12,
                              &ABL, /**/ &ABR,      A20, /**/ A21,      A22,
             /* with A11 added to submatrix */ FLA_BR );
    FLA_Cont_with_3x3_to_2x2( &BTL, /**/ &BTR,      B00,  B01,     /**/ B02,
                                    /**/            B10,  B11,     /**/ B12,
                            /* ************** */ /* ************************ */
                              &BBL, /**/ &BBR,      B20,  B21,     /**/ B22,
             /* with B11 added to submatrix */ FLA_TL );
    FLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR,      C00,  C01,     /**/ C02,
                            /* ************** */ /* ************************ */
                                    /**/            C10,  C11,     /**/ C12,
                              &CBL, /**/ &CBR,      C20,  C21,     /**/ C22,
             /* with C11 added to submatrix */ FLA_BL );
  }
  return FLA_SUCCESS;
}
```

Figure 9: FLASH code for the solution of the triangular Sylvester equation (continued).