Programming Algorithms-by-Blocks for Matrix Computations on Multithreaded Architectures FLAME Working Note #29

Gregorio Quintana-Ortí* Enrique S. Quintana-Ortí* Robert van de Geijn †
Field G. Van Zee † Ernie Chan †

Abstract

With the emergence of thread-level parallelism as the primary means for continued improvement of performance, the programmability issue has reemerged as an obstacle to the use of architectural advances. We argue that evolving legacy libraries for dense and banded linear algebra is not a viable solution due to constraints imposed by early design decisions. We propose a philosophy of abstraction and separation of concerns that provides a promising solution in this problem domain. The first abstraction, FLASH, allows algorithms to express computation with matrices consisting of blocks, facilitating algorithms-by-blocks. Transparent to the library implementor, operand descriptions are registered for a particular operation *a priori*. A runtime system, SuperMatrix, uses this information to identify data dependencies between sub-operations, allowing them to be scheduled to threads out-of-order and executed in parallel. But not all classical algorithms in linear algebra lend themselves to conversion to algorithms-by-blocks. We show how our recently proposed LU factorization with incremental pivoting and closely related algorithm-by-blocks for the QR factorization, both originally designed for out-of-core computation, overcome this difficulty. Anecdotal evidence regarding the development of routines with a core functionality demonstrates how the methodology supports high productivity while experimental results suggest that high performance is abundantly achievable.

1 Introduction

For the first time in history, architectures are approaching physical and technological barriers which make increasing the speed of a single core exceedingly difficult and economically infeasible. As a result, hardware architects have begun to design microprocessors with multiple processing cores that operate independently and share the same address space. It appears that the advent of these multi-core architectures will finally force a radical change in how applications are programmed. Specifically, developers must consider how to direct many concurrent threads of execution to collaborate and solve a single problem. In this paper, we present what we believe to be a promising solution to what is widely perceived to be a very difficult problem, targeting the domain of dense and banded linear algebra libraries.

Experience gained from parallel computing architectures with complex hierarchical memories has shown that an intricate and interdependent set of requirements must be met in order to achieve the level of performance that scientific applications demand of linear algebra libraries. These requirements include data locality, load balance, and careful attention to the critical path of computation, to name a few. While it may be possible to satisfy this set of constraints when implementing a single operation on a single architecture, addressing them for an entire library of commonly used operations requires one to face the additional challenge of programmability.

*Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume I, Campus Riu Sec, 12.071, Castellón, Spain, {gquintan,quintana}@icc.uji.es.

†Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712, {rvdg,field,echan}@cs.utexas.edu.

We propose abandoning essentially all conventions that were adopted when widely-used libraries like LINPACK [13], LAPACK [3], and ScaLAPACK [12], denoted by LIN(Sca)LAPACK hereafter, were designed in the last quarter of the 20th century. To us, nothing associated with these libraries is sacred: not the algorithms, not the notation used to express algorithms, not the data structures used to store matrices, not the APIs used to code the algorithms, and not the runtime system that executes the implementations (if one ever existed). Instead we build on the notation, APIs, and tools developed as part of the FLAME project, which provide modern object-based abstractions to increase developer productivity and user-friendliness alike. Perhaps the most influential of these abstractions is one that provides advanced shared-memory parallelism by borrowing out-of-order scheduling techniques from sequential superscalar architectures.

From the outset, we have been focused on developing a prototype library that encompasses much of the functionality of the core of LAPACK, including LU, QR and Cholesky factorizations, and related solvers. The LU factorization with pivoting and Householder QR factorizations are key since algorithms-by-blocks for these operations require new algorithms. Fortunately, we had already developed high-performance algorithms for out-of-core computation that were algorithm-by-blocks, except that the block size targeted the movement of data from disk to main memory rather than thread-level parallelism [20, 24]. Thus, we knew from the start that algorithms-by-blocks were achievable for all of these important operations.

We focus on the programmability issue that developers face given that parallelism will be required in order to exploit the performance potential of multi-core and many-core systems. We point to the fact that the parallelization effort described in this paper was conceived in early September 2006. Remarkably, all results presented in this paper were already achieved by September 2007. The key has been a clear separation of concerns between the code that implements the algorithms and the runtime system that schedules tasks.

We are not alone in recognizing the utility of a runtime system that dynamically schedules subproblems for parallel execution. The PLASMA project, independent of our efforts but with LU and QR factorization algorithms that are similarly based on our earlier work on out-of-core algorithms, has developed a similar mechanism in the context of the QR factorization. Like the SuperMatrix system described here and in [9, 10, 11], the PLASMA system enqueues operation subproblems as "tasks" on a queue, builds a directed acyclic graph (DAG) to encode dependencies, and then executes the subproblems as task dependencies become satisfied [8]. However, as part of [8] they do not provide any source code—neither example code implementing the runtime-aware algorithm nor code implementing the runtime system itself. The same authors expand on their work in [7] to include remarks and results for the Cholesky factorization and LU factorization with incremental pivoting based on our out-of-core algorithm [24], which we encourage the reader to study. In that paper they once again elect to omit code. Thus, it is impossible for the reader to evaluate or verify the merits of the PLASMA scheduling mechanism with respect to portability, clarity, and flexibility to other operations.

In this paper, we bring together results from a number of conference papers [9, 10, 11, 31, 32, 33] into a comprehensive treatment of the methodology. The paper is structured as follows. Section 2 provides a motivating example in the form of the LU factorization (without pivoting) and makes the case that the LIN(Sca)LAPACK design philosophies are conducive neither to ease-of-programming nor efficient parallelism. Section 3 discusses algorithms-by-blocks and the challenges they present, and provides an overview of the FLASH API, including an interface for filling matrices that use storage-by-blocks. Section 4 gives an overview of the SuperMatrix runtime parallel execution mechanism in the context of the LU factorization. Section 5 recounts the authors' work in parallelizing the level-3 BLAS [14] operations using SuperMatrix. Section 6 expands the discussion of operations to the LU factorization with partial and incremental pivoting. Performance results are reported in Section 7. Finally, Section 8 contains concluding remarks.

2 A Motivating Example: The LU Factorization without Pivoting

The LU factorization (without pivoting) of an $n \times n$ matrix A is given by $A = LU$ where L is $n \times n$ unit lower triangular and U is $n \times n$ upper triangular. In traditional algorithms for this factorization, the triangular factors overwrite A, with the strictly lower triangular part of L stored on the subdiagonal elements of A and the upper triangular part of U stored on those elements of A on and above the diagonal. We denote this as $A := \{L \backslash U\}$.

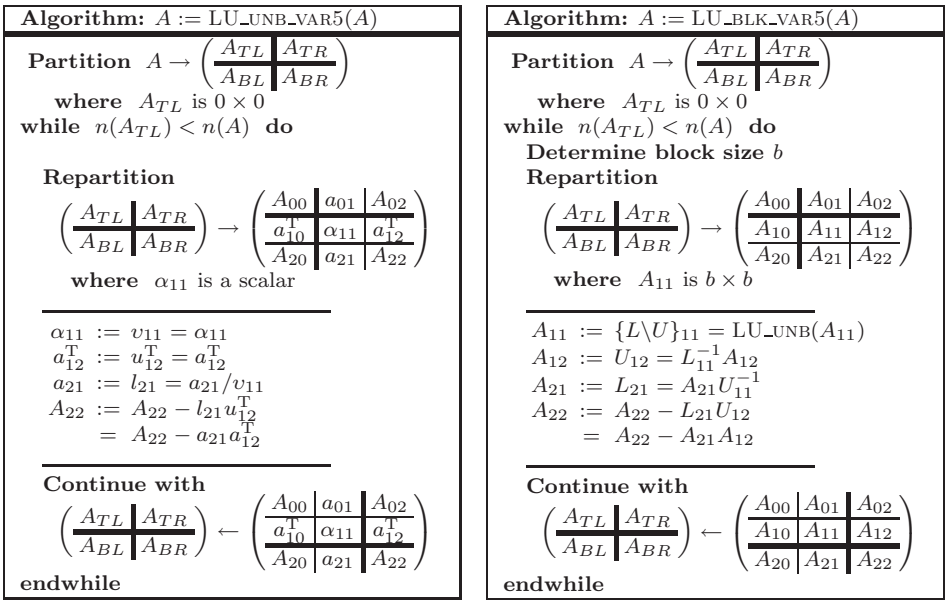


Figure 1: Unblocked and blocked algorithms (left and right, respectively) for computing the LU factorization (without pivoting). Here, $n(B)$ stands for the number of columns of B.

```
FLA_Error FLA_LU_blk_var5( FLA_Obj A, int nb_alg )
{
  FLA_Obj ATL, ATR,      A00, A01, A02,
          ABL, ABR,      A10, A11, A12,
                         A20, A21, A22;
  int b;

  FLA_Part_2x2( A,    &ATL, &ATR,
                      &ABL, &ABR,    0, 0, FLA_TL );

  while ( FLA_Obj_width( ATL ) < FLA_Obj_width( A ) ) {
    b = min( FLA_Obj_length(ABR), nb_alg );
    FLA_Repart_2x2_to_3x3(
       ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
    /* ************* */   /* ******************** */
                           &A10, /**/ &A11, &A12,
       ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
       b, b, FLA_BR );
    /*-----------------------------------------------*/
    FLA_LU_unb_var5( A11 );
    FLA_Trsm( FLA_LEFT, FLA_LOWER_TRIANGULAR,
              FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
              FLA_ONE, A11,
                       A12 );
    FLA_Trsm( FLA_RIGHT, FLA_UPPER_TRIANGULAR,
              FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
              FLA_ONE, A11,
                       A21 );
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
              FLA_MINUS_ONE, A21,
                             A12,
              FLA_ONE,       A22 );
    /*-----------------------------------------------*/
    FLA_Cont_with_3x3_to_2x2(
       &ATL, /**/ &ATR,        A00, A01, /**/ A02,
                               A10, A11, /**/ A12,
    /* *************** */   /* ****************** */
       &ABL, /**/ &ABR,        A20, A21, /**/ A22,
       FLA_TL );
  }
  return FLA_SUCCESS;
}
```

```
FLA_Error FLASH_LU_by_blocks_var5( FLA_Obj A )
{
  FLA_Obj ATL, ATR,      A00, A01, A02,
          ABL, ABR,      A10, A11, A12,
                         A20, A21, A22;


  FLA_Part_2x2( A,    &ATL, &ATR,
                      &ABL, &ABR,    0, 0, FLA_TL );

  while ( FLA_Obj_width( ATL ) < FLA_Obj_width( A ) ) {

    FLA_Repart_2x2_to_3x3(
       ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
    /* ************* */   /* ******************** */
                           &A10, /**/ &A11, &A12,
       ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
       1, 1, FLA_BR );
    /*-----------------------------------------------*/
    FLA_LU_unb_var5( FLASH_MATRIX_AT( A11 ) );
    FLASH_Trsm( FLA_LEFT, FLA_LOWER_TRIANGULAR,
                FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
                FLA_ONE, A11,
                         A12 );
    FLASH_Trsm( FLA_RIGHT, FLA_UPPER_TRIANGULAR,
                FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
                FLA_ONE, A11,
                         A21 );
    FLASH_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
                FLA_MINUS_ONE, A21,
                               A12,
                FLA_ONE,       A22 );
    /*-----------------------------------------------*/
    FLA_Cont_with_3x3_to_2x2(
       &ATL, /**/ &ATR,        A00, A01, /**/ A02,
                               A10, A11, /**/ A12,
    /* *************** */   /* ****************** */
       &ABL, /**/ &ABR,        A20, A21, /**/ A22,
       FLA_TL );
  }
  return FLA_SUCCESS;
}
```

Figure 2: Left: FLAME/C implementation of the blocked algorithm in Figure 1 (right). Right: FLASH implementation of the algorithm-by-blocks.

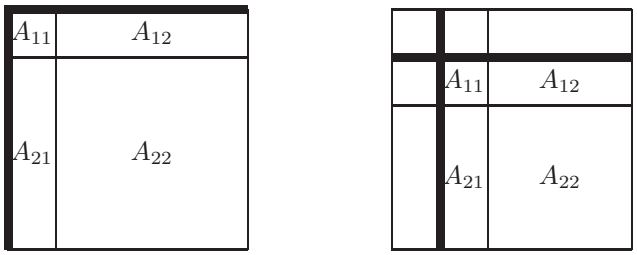


Figure 3: First two iterations of the blocked algorithm in Figure 1 (right).

2.1 A typical algorithm

In Figure 1 we give unblocked and blocked algorithms, in FLAME notation [19], for overwriting a matrix A with the triangular factors L and U. The unblocked algorithm on the left involves vector-vector and matrix-vector operations, which perform $O(1)$ floating-point arithmetic operations (flops) for every memory operation (memop). This ratio renders low performance on current cache-based processors as memops are considerably slower than flops in these architectures. The blocked algorithm on the right of that figure is likely to attain high performance since most computation is cast in terms of the *matrix-matrix product* (GEMM) $A_{22} := A_{22} - A_{21}A_{12}$, which performs $O(b)$ flops for every memop.

Using the FLAME/C API [6], an equivalent blocked algorithm can be represented in code as presented in Figure 2 (left). Comparing and contrasting Figures 1 and 2 (left) shows how the FLAME notation, which departs from the commonly encountered loop-based algorithms, translates more naturally into code when an appropriate API is defined for the target programming language. And thus we abandon conventional algorithm notation and the LIN(Sca)LAPACK style of programming.

2.2 The trouble with evolving legacy code to multithreaded architectures

The conventional approach to parallelize dense linear algebra operations on multithreaded architectures has been to push parallelism into multithreaded versions of the BLAS. The rationale behind this is to require minimal changes to existing codes.

In the case of the LU factorization, this means parallelism is attained only within the two TRSM and the GEMM operations:

$$\begin{array}{rcl} A_{12} & := & L_{11}^{-1}A_{12}, \\ A_{21} & := & A_{21}U_{11}^{-1}, \\ A_{22} & := & A_{22} - L_{21}U_{12}. \end{array}$$

While we will see that this works well when the matrix is large and there are relatively few processors, a bottleneck forms when the ratio of the matrix dimension to the number of processors is low. In particular, the block size (variable b in Figures 1 (right) and 2 (left), respectively) must be relatively large (in practice, in the $128 - 256$ range) so that the GEMM subproblems, which form the bulk of the LU computation, deliver high performance [18]. As a result, the LU factorization of A_{11}, typically computed by only a single processor, leaves other threads idle and therefore hinders parallel efficiency. Thus, this approach to extracting parallelism traditionally advocated by LAPACK/ScaLAPACK is inherently limiting.

One technique that attempts to overcome such a bottleneck is to "compute ahead." Consider the illustration in Figure 3 of the partitionings of A at the beginning of the first two iterations of the blocked algorithm for the LU factorization. In this technique, the update of A_{22} during the first iteration is broken down into the update of the part of A_{22} that will become A_{11} in the next iteration (see Figure 3), followed by the update of the rest of A_{22}. This then allows the factorization of the next A_{11} to be scheduled before the update of the remaining parts of the current A_{22}, thus overcoming the bottleneck. Extensions of this idea compute ahead several iterations in a similar manner.

The problem with this idea is that it greatly complicates the code that implements the algorithm if coded in a traditional style [1, 25, 34]. While feasible for a single, relatively simple algorithm like the LU factorization without pivoting or the Cholesky factorization, re-implementing a linear algebra library like LAPACK would become a daunting task if this strategy were employed.

3 Algorithms-By-Blocks

Fred Gustavson (IBM) has long advocated an alternative to the blocked algorithms in LAPACK [2, 16, 22]. The solution, *algorithms-by-blocks*, proposes algorithms that view matrices as collections of submatrices and express their computation in terms of these submatrix blocks.

3.1 Basic idea

The idea is simple. When moving from algorithms that cast most computation in terms of matrix-vector operations to algorithms that mainly operate in terms of matrix-matrix computations, rather than improving performance by aggregating the computation into matrix-matrix computations, the developer should raise the granularity of the data by replacing each element in the matrix by a submatrix (block). Algorithms are then written as before, except with scalar operations replaced by operations on the blocks.

For example, consider the LU factorization of the partitioned matrix:

$$A \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{ccc|c|ccc} \bar{\alpha}_{00} & \cdots & \bar{\alpha}_{0,k-1} & \bar{\alpha}_{0k} & \bar{\alpha}_{0,k+1} & \cdots & \bar{\alpha}_{0,n-1} \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \bar{\alpha}_{k-1,0} & \cdots & \bar{\alpha}_{k-1,k-1} & \bar{\alpha}_{k-1,k} & \bar{\alpha}_{k-1,k+1} & \cdots & \bar{\alpha}_{k-1,n-1} \\ \hline \bar{\alpha}_{k0} & \cdots & \bar{\alpha}_{k,k-1} & \bar{\alpha}_{kk} & \bar{\alpha}_{k,k+1} & \cdots & \bar{\alpha}_{k,n-1} \\ \hline \bar{\alpha}_{k+1,0} & \cdots & \bar{\alpha}_{k+1,k-1} & \bar{\alpha}_{k+1,k} & \bar{\alpha}_{k+1,k+1} & \cdots & \bar{\alpha}_{k+1,n-1} \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \bar{\alpha}_{n-1,0} & \cdots & \bar{\alpha}_{n-1,k-1} & \bar{\alpha}_{n-1,k} & \bar{\alpha}_{n-1,k+1} & \cdots & \bar{\alpha}_{n-1,n-1} \end{array}\right)$$

where α_{11}, $\bar{\alpha}_{ij}$, $0 \leq i, j < n$, are all scalars. The unblocked algorithm in Figure 1 (left) can be turned into an algorithm-by-blocks by recognizing that if each element in the matrix is itself a matrix, as in

$$A \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{ccc|c|ccc} \bar{A}_{00} & \cdots & \bar{A}_{0,K-1} & \bar{A}_{0K} & \bar{A}_{0,K+1} & \cdots & \bar{A}_{0,N-1} \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \bar{A}_{K-1,0} & \cdots & \bar{A}_{K-1,K-1} & \bar{A}_{K-1,K} & \bar{A}_{K-1,K+1} & \cdots & \bar{A}_{K-1,N-1} \\ \hline \bar{A}_{K0} & \cdots & \bar{A}_{K,K-1} & \bar{A}_{KK} & \bar{A}_{K,K+1} & \cdots & \bar{A}_{K,N-1} \\ \hline \bar{A}_{K+1,0} & \cdots & \bar{A}_{K+1,K-1} & \bar{A}_{K+1,K} & \bar{A}_{K+1,K+1} & \cdots & \bar{A}_{K+1,N-1} \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \bar{A}_{N-1,0} & \cdots & \bar{A}_{N-1,K-1} & \bar{A}_{N-1,K} & \bar{A}_{N-1,K+1} & \cdots & \bar{A}_{N-1,N-1} \end{array}\right)$$

with A_{11} and $\bar{A}_{ij}$, $0 \leq i, j < N$, are all $b \times b$ blocks, then:

1. $\alpha_{11} := \upsilon_{11}$ $(= \lambda_{11}\upsilon_{11}$, with $\lambda_{11} = 1)$ becomes the LU factorization of the matrix element α_{11}:
$$A_{11} := \{L \backslash U\}_{11} = \{\bar{L} \backslash \bar{U}\}_{KK}.$$

2. a_{21} represents a column vector of blocks so that $a_{21} := a_{21}/\upsilon_{11}$ becomes a triangular solve with multiple right-hand sides with the updated lower triangular matrix in element α_{11} and each of the blocks in vector a_{21}:
$$A_{21} := A_{21}U_{11}^{-1} = \left(\begin{array}{c} \bar{A}_{K+1,K} \\ \vdots \\ \bar{A}_{N-1,K} \end{array}\right) \bar{U}_{KK}^{-1} = \left(\begin{array}{c} \bar{A}_{K+1,K}\bar{U}_{KK}^{-1} \\ \vdots \\ \bar{A}_{N-1,K}\bar{U}_{KK}^{-1} \end{array}\right).$$

3. a_{12} represents a row vector of blocks so that $a_{12} := u_{12}^T$ $(= \lambda_{11}^{-1}u_{12}^T)$ becomes a triangular solve with multiple right-hand sides with the updated upper triangular matrix in element α_{11} and each of the blocks in vector a_{12}:
$$\begin{array}{rcl} A_{12} := L_{11}^{-1}A_{12} & = & \bar{L}_{KK}^{-1} \left(\begin{array}{ccc} \bar{A}_{K,K+1} & \cdots & \bar{A}_{K,N-1} \end{array}\right) \\ & = & \left(\begin{array}{ccc} \bar{L}_{KK}^{-1}\bar{A}_{K,K+1} & \cdots & \bar{L}_{KK}^{-1}\bar{A}_{K,N-1} \end{array}\right). \end{array}$$

4. Each element in A_{22} describes a block that needs to be updated via a matrix-matrix product using blocks from the updated vectors a_{21} and a_{12}:

$$\begin{array}{rcl} A_{22} & := & A_{22} - A_{21}A_{12} \\ & = & \begin{pmatrix} \bar{A}_{K+1,K+1} & \cdots & \bar{A}_{K+1,N-1} \\ \vdots & \ddots & \vdots \\ \bar{A}_{N-1,K+1} & \cdots & \bar{A}_{N-1,N-1} \end{pmatrix} - \begin{pmatrix} \bar{A}_{K+1,K} \\ \vdots \\ \bar{A}_{N-1,K} \end{pmatrix} \begin{pmatrix} \bar{A}_{K,K+1} & \cdots & \bar{A}_{K,N-1} \end{pmatrix} \\ & = & \begin{pmatrix} \bar{A}_{K+1,K+1} - \bar{A}_{K+1,K}\bar{A}_{K,K+1} & \cdots & \bar{A}_{K+1,N-1} - \bar{A}_{K+1,K}\bar{A}_{K,N-1} \\ \vdots & \ddots & \vdots \\ \bar{A}_{N-1,K+1} - \bar{A}_{N-1,K}\bar{A}_{K,K+1} & \cdots & \bar{A}_{N-1,N-1} - \bar{A}_{N-1,K}\bar{A}_{K,N-1} \end{pmatrix}. \end{array}$$

Below we will show that the algorithm-by-blocks approach also facilitates the high-performance implementation and parallel execution of matrix operations on SMP and multi-core architectures.

3.2 Obstacles

A major obstacle to algorithms-by-blocks lies with the complexity that is introduced into the code when matrices are manipulated and stored by blocks. A number of solutions have been proposed to solve this problem, ranging from storing the matrix in arrays with four or more dimensions and explicitly exposing intricate indexing into the individual elements [21], to template programming using C++ [36], and to compiler-based solutions [39]. None of these have yielded a consistent methodology that allows the development of high-performance libraries with functionality that rivals LAPACK or FLAME. The problem is programmability.

3.3 The FLASH API for algorithms-by-blocks

Several recent efforts [16, 23, 26] follow an approach different from those mentioned above. They view the matrix as a matrix of smaller matrices, just as it is conceptually described. Among these, the FLASH API [26], which is an extension of the FLAME API, exploits the fact that FLAME encapsulates matrix information in objects by allowing elements of a matrix to themselves be descriptions of matrices. This view yields a matrix hierarchy, potentially with multiple levels.

Using the FLASH API, code for an algorithm-by-blocks for the LU factorization is given in Figure 2 (right). Note the remarkable similarity between that implementation and the blocked implementation for the LU factorization on the left of the same figure. Note also how that code maintains the traditional layering of subroutine calls that implement linear algebra operations, which illustrates that we are willing to preserve conventions from the BLAS/LIN(Sca)LAPACK efforts that continue to benefit programmability. However, it is worth pointing out that we actually advocate a deeper layering, one that allows the programmer to invoke routines that assume certain matrix shapes. This design yields potential performance benefits when the underlying BLAS implementation provides interfaces to low-level kernels [18, 28]. Such an extended matrix multiplication interface can be seen in our use and implementation of `FLASH_Gebp_nn` in Figure 4, which assumes a matrix-matrix product where A is a block and B is a row-panel.

It may seem that complexity is merely hidden in the routines `FLASH_Trsm`, and `FLASH_Gemm`. The abbreviated implementations of these operations shown in Figure 4 demonstrate how the FLASH API is used in the implementation of those routines as well. The reader can see here that many of the details of the FLASH implementation have been buried within the FLASH-aware FLAME object definition. The fact that these algorithms operate on hierarchical matrices (which use storage-by-blocks) manifests only through the unit block size, the use of alternative `FLASH_` routines to further break subproblems into tasks with block operands, and an additional `FLASH_MATRIX_AT` macro to extract the appropriate submatrix when wrappers to external level-3 BLAS are invoked.

As a result, transforming blocked algorithms into algorithms-by-blocks and/or developing algorithms-by-blocks from scratch using the FLASH API is straightforward.

```
void FLASH_Trsm_llnu( FLA_Obj alpha, FLA_Obj L,
                                     FLA_Obj B )
/* Special case with mode parameters
     FLASH_Trsm( FLA_LEFT, FLA_LOWER_TRIANGULAR,
                 FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
                 ...                                  )
   Assumption: L consists of one block and
               B consists of a row of blocks        */
{
  FLA_Obj BL, BR,      B0, B1, B2;



  FLA_Part_1x2( B,    &BL, &BR,    0, FLA_LEFT );


  while ( FLA_Obj_width( BL ) < FLA_Obj_width( B ) ) {
    FLA_Repart_1x2_to_1x3( BL, /**/ BR,   &B0, /**/ &B1, &B2,
                           1, FLA_RIGHT );



    /*--------------------------------------------------*/
    FLA_Trsm( FLA_LEFT, FLA_LOWER_TRIANGULAR,
              FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
              alpha, FLASH_MATRIX_AT( L ),
                     FLASH_MATRIX_AT( B1 ) );
    /*--------------------------------------------------*/
    FLA_Cont_with_1x3_to_1x2( &BL, /**/ &BR,   B0, B1, /**/ B2,
                              FLA_LEFT );


  }
}
```

```
void FLASH_Trsm_runn( FLA_Obj alpha, FLA_Obj U,
                                     FLA_Obj B )
/* Special case with mode parameters
     FLASH_Trsm( FLA_RIGHT, FLA_UPPER_TRIANGULAR,
                 FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
                 ...                                  )
   Assumption: U consists of one block and
               B consists of a column of blocks     */
{
  FLA_Obj BT,      B0,
          BB,      B1,
                   B2;


  FLA_Part_2x1( B,    &BT,
                      &BB,    0, FLA_TOP );


  while ( FLA_Obj_length( BT ) < FLA_Obj_length( B ) ) {
    FLA_Repart_2x1_to_3x1( BT,       &B0,
                         /* ** */   /* ** */
                                    &B1,
                           BB,       &B2,    1, FLA_BOTTOM );
    /*--------------------------------------------------*/
    FLA_Trsm( FLA_RIGHT, FLA_UPPER_TRIANGULAR,
              FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
              alpha, FLASH_MATRIX_AT( U ),
                     FLASH_MATRIX_AT( B1 ) );
    /*--------------------------------------------------*/
    FLA_Cont_with_3x1_to_2x1( &BT,        B0,
                                          B1,
                            /* ** */   /* ** */
                              &BB,        B2,    FLA_TOP );
  }
}
```

```
void FLASH_Gemm_nn( FLA_Obj alpha, FLA_Obj A,
                                   FLA_Obj B,
                    FLA_Obj beta,  FLA_Obj C )
/* Special case with mode parameters
     FLASH_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
                 ...                                  )
   Assumption: A is a column of blocks (column panel) and
               B is a row of blocks (row panel)     */
{
  FLA_Obj AT,      A0,      CT,      C0,
          AB,      A1,      CB,      C1,
                   A2,               C2;

  FLA_Part_2x1( A,    &AT,
                      &AB,    0, FLA_LEFT );
  FLA_Part_2x1( C,    &CT,
                      &CB,    0, FLA_TOP );

  while ( FLA_Obj_length( AL ) < FLA_Obj_length( A ) ) {
    FLA_Repart_2x1_to_3x1( AT,       &A0,
                         /* ** */   /* ** */
                                    &A1,
                           AB,       &A2,    1, FLA_BOTTOM );
    FLA_Repart_2x1_to_3x1( CT,       &C0,
                         /* ** */   /* ** */
                                    &C1,
                           CB,       &C2,    1, FLA_BOTTOM );
    /*--------------------------------------------------*/
    FLASH_Gebp_nn( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
                   alpha, A1,
                          B,
                   beta,  C1 );
    /*--------------------------------------------------*/
    FLA_Cont_with_3x1_to_2x1( &AT,        A0,
                                          A1,
                            /* ** */   /* ** */
                              &AB,        A2,    FLA_TOP );
    FLA_Cont_with_3x1_to_2x1( &CT,        C0,
                                          C1,
                            /* ** */   /* ** */
                              &CB,        C2,    FLA_TOP );
  }
}
```

```
void FLASH_Gebp_nn( FLA_Obj alpha, FLA_Obj A,
                                   FLA_Obj B,
                    FLA_Obj beta,  FLA_Obj C )
/* Special case with mode parameters
     FLASH_Gebp( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
                 ...                                  )
   Assumption: A is a block and
               B, C are rows of blocks (row panels) */
{
  FLA_Obj BL, BR,      B0, B1, B2,
          CL, CR,      C0, C1, C2;


  FLA_Part_1x2( B,    &BL, &BR,    0, FLA_LEFT );

  FLA_Part_1x2( C,    &CL, &CR,    0, FLA_LEFT );


  while ( FLA_Obj_width( BL ) < FLA_Obj_width( B ) ) {
    FLA_Repart_1x2_to_1x3( BL, /**/ BR,   &B0, /**/ &B1, &B2,
                           1, FLA_RIGHT );


    FLA_Repart_1x2_to_1x3( CL, /**/ CR,   &C0, /**/ &C1, &C2,
                           1, FLA_RIGHT );


    /*--------------------------------------------------*/
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
              alpha, FLASH_MATRIX_AT( A ),
                     FLASH_MATRIX_AT( B1 ),
              beta,  FLASH_MATRIX_AT( C1 ) );
    /*--------------------------------------------------*/
    FLA_Cont_with_1x3_to_1x2( &BL, /**/ &BR,   B0, B1, /**/ B2,
                              FLA_LEFT );


    FLA_Cont_with_1x3_to_1x2( &CL, /**/ &CR,   C0, C1, /**/ C2,
                              FLA_LEFT );


  }
}
```

Figure 4: Top: FLASH implementations of triangular system solve with multiple right-hand sides. Bottom: FLASH implementations of matrix-matrix product.

3.4 Filling the matrix

A nontrivial matter that has prevented acceptance of alternative data structures for storing matrices has been the interface to the application. Historically, the LIN(Sca)LAPACK approach has granted application direct access to the data. This requires the application programmer to understand how data is stored, which greatly increases the programming burden on the user [15].

Our approach currently supports three alternative solutions.

Referencing conventional arrays

Recall that the FLASH API allows matrix elements to contain submatrices. Leaf objects in this hierarchy encapsulate the actual numerical matrix data. Given a matrix stored in conventional column-major order, a FLASH matrix object can be constructed such that the leaf matrices simply refer to submatrices of the user-supplied matrix. Notice that this means the user can access the elements of the matrix as one would when interfacing with a conventional library like LAPACK. The main disadvantage is that the leaf matrices are not stored contiguously.

Contributions to a matrix object

We will see that there is a distinct performance benefit to storing leaf matrices contiguously. Also, applications often naturally generate matrices by computing submatrices which are contributed to a larger overall matrix, possibly by adding to a partial result [15].

For this scenario we provide routines for contribution to a FLASH matrix object. For example, FLASH provides a function whose signature is given by:

```
void FLASH_Axpy_submatrix_to_global( FLA_Obj alpha,
                                     int m, int n,
                                     void* B, int ldb,
                                     int i, int j, FLA_Obj H );
```

This call accepts an `m` $\times$ `n` matrix B, stored at address `B` with leading dimension `ldb`, scales it by scalar α and adds the result to the submatrix of H that has as its top-left element the (`i`, `j`) element of H. Note that matrix descriptor `H` refers to a hierarchically stored matrix object while B is a matrix created and stored in conventional column-major order. In (MATLAB) Mscript notation, this operation is given by

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

A complementary routine allows submatrices to be extracted. Notice that given such an interface the user needs not know how matrix H is actually stored.

This approach has been highly successful for interface applications to our Parallel Linear Algebra Package (PLAPACK) library for distributed-memory architectures [15, 37] where filling distributed matrices poses a similar challenge. Analogous interfaces are also used by the Global Array abstraction [29] and PETSc [4]. We strongy recommend this departure from the LIN/(Sca)LAPACK interface to applicationts.

Converting whole matrices

It is also possible to allow the user to construct whole matrices in column-major order, which then may be used to build hierarchical matrices that contain the equivalent data. The submission process described above can be used for this conversion.

4 SuperMatrix Out-of-Order Scheduling

In this section we discuss how techniques used in superscalar processors can be adopted to systematically expose parallelism in algorithms-by-blocks without obfuscating the coded algorithms with further complexity.

Operation	Result	Operands		
		In		In/out
1. FLA_LU_unb_var5($\bar{A}_{00}$)	$\bar{A}_{00} := \{\bar{L}_{00} \backslash \bar{U}_{00}\} = LU(\bar{A}_{00})$			$\bar{A}_{00}\surd$
2. FLA_Trsm(..., $\bar{A}_{00}, \bar{A}_{01}$)	$\bar{A}_{01} := \bar{L}_{00}^{-1}\bar{A}_{01}$	$\bar{A}_{00}$		$\bar{A}_{01}\surd$
3. FLA_Trsm(..., $\bar{A}_{00}, \bar{A}_{02}$)	$\bar{A}_{02} := \bar{L}_{00}^{-1}\bar{A}_{02}$	$\bar{A}_{00}$		$\bar{A}_{02}\surd$
4. FLA_Trsm(..., $\bar{A}_{00}, \bar{A}_{10}$)	$\bar{A}_{10} := \bar{A}_{10}\bar{U}_{00}^{-1}$	$\bar{A}_{00}$		$\bar{A}_{10}\surd$
5. FLA_Trsm(..., $\bar{A}_{00}, \bar{A}_{20}$)	$\bar{A}_{20} := \bar{A}_{20}\bar{U}_{00}^{-1}$	$\bar{A}_{00}$		$\bar{A}_{20}\surd$
6. FLA_Gemm(..., $\bar{A}_{10}, \bar{A}_{01}, \ldots, \bar{A}_{11}$)	$\bar{A}_{11} := \bar{A}_{11} - \bar{A}_{10}\bar{A}_{01}$	$\bar{A}_{10}$	$\bar{A}_{01}$	$\bar{A}_{11}\surd$
7. FLA_Gemm(..., $\bar{A}_{10}, \bar{A}_{02}, \ldots, \bar{A}_{12}$)	$\bar{A}_{12} := \bar{A}_{12} - \bar{A}_{10}\bar{A}_{02}$	$\bar{A}_{10}$	$\bar{A}_{02}$	$\bar{A}_{12}\surd$
8. FLA_Gemm(..., $\bar{A}_{20}, \bar{A}_{01}, \ldots, \bar{A}_{21}$)	$\bar{A}_{21} := \bar{A}_{21} - \bar{A}_{20}\bar{A}_{01}$	$\bar{A}_{20}$	$\bar{A}_{01}$	$\bar{A}_{21}\surd$
9. FLA_Gemm(..., $\bar{A}_{20}, \bar{A}_{02}, \ldots, \bar{A}_{22}$)	$\bar{A}_{22} := \bar{A}_{22} - \bar{A}_{20}\bar{A}_{02}$	$\bar{A}_{20}$	$\bar{A}_{02}$	$\bar{A}_{22}\surd$
10. FLA_LU_unb_var5($\bar{A}_{11}$)	$\bar{A}_{11} := \{\bar{L}_{11} \backslash \bar{U}_{11}\} = LU(\bar{A}_{11})$			$\bar{A}_{11}$
11. FLA_Trsm(..., $\bar{A}_{11}, \bar{A}_{12}$)	$\bar{A}_{12} := \bar{L}_{11}^{-1}\bar{A}_{12}$	$\bar{A}_{11}$		$\bar{A}_{12}$
12. FLA_Trsm(..., $\bar{A}_{11}, \bar{A}_{21}$)	$\bar{A}_{21} := \bar{A}_{21}\bar{U}_{11}^{-1}$	$\bar{A}_{11}$		$\bar{A}_{21}$
13. FLA_Gemm(..., $\bar{A}_{21}, \bar{A}_{12}, \ldots, \bar{A}_{22}$)	$\bar{A}_{22} := \bar{A}_{22} - \bar{A}_{21}\bar{A}_{12}$	$\bar{A}_{21}$	$\bar{A}_{12}$	$\bar{A}_{22}$
14. FLA_LU_unb_var5($\bar{A}_{22}$)	$\bar{A}_{22} := \{\bar{L}_{22} \backslash \bar{U}_{22}\} = LU(\bar{A}_{22})$			$\bar{A}_{22}$

Figure 5: Complete list of operations to be performed on blocks for the LU factorization (without pivoting) of a 3×3 matrix of blocks using algorithm-by-blocks. The "$\surd$"-marks denote those operands that are available immediately at the beginning of the algorithm (ie: those operands that are not dependent upon other operations).

4.1 SuperMatrix dynamic scheduling and out-of-order execution

In order to illustrate the scheduling mechanism during this subsection, we consider the matrix of 3×3 blocks

$$A \rightarrow \begin{pmatrix} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} \\ \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} \end{pmatrix}$$

where all blocks are $b \times b$. First, the code in Figure 2 (right) is linked to the SuperMatrix runtime library and executed sequentially. As suboperations are encountered, the information associated with each suboperation is encapsulated and placed onto an internal queue. Once all operations are enqueued, the initial *analyzer stage* of execution is complete. Figure 5 provides a human-readable list corresponding to the full queue generated for a 3×3 matrix of blocks.

For example, during the first iteration of the code, the call

```
FLA_LU_unb_var5( FLASH_MATRIX_AT( A11 ) );
```

inserts the LU factorization of block $\bar{A}_{00}$ into the list. During the same iteration, suboperations encountered inside `FLASH_Trsm` or `FLASH_Syrk` also enqueue their corresponding task entries. The order of the operations in the list, together with the operands that are read (input operands) and written (output operands) in the operation, determine the dependencies among matrix operations. Thus, the second operation in the list, which has $\bar{A}_{00}$ as an input operand and $\bar{A}_{01}$ as both an input and an output operand, requires the first operation to be completed before it may begin. The list denotes available operands with a "$\surd$"-mark; these operands are not dependent upon the completion of any other operations. They also happen to represent the operands that are available at the beginning of the algorithm-by-blocks since the list captures the state of the queue before execution.

During the *scheduler/dispatcher stage*, operations that have all operands available are scheduled for execution. As computation progresses, dependencies are satisfied and new operands become available, allowing more operations to be dequeued and executed (see Figure 6). The overhead of this runtime mechanism is amortized over a large amount of computation, and therefore its overall cost is minor.

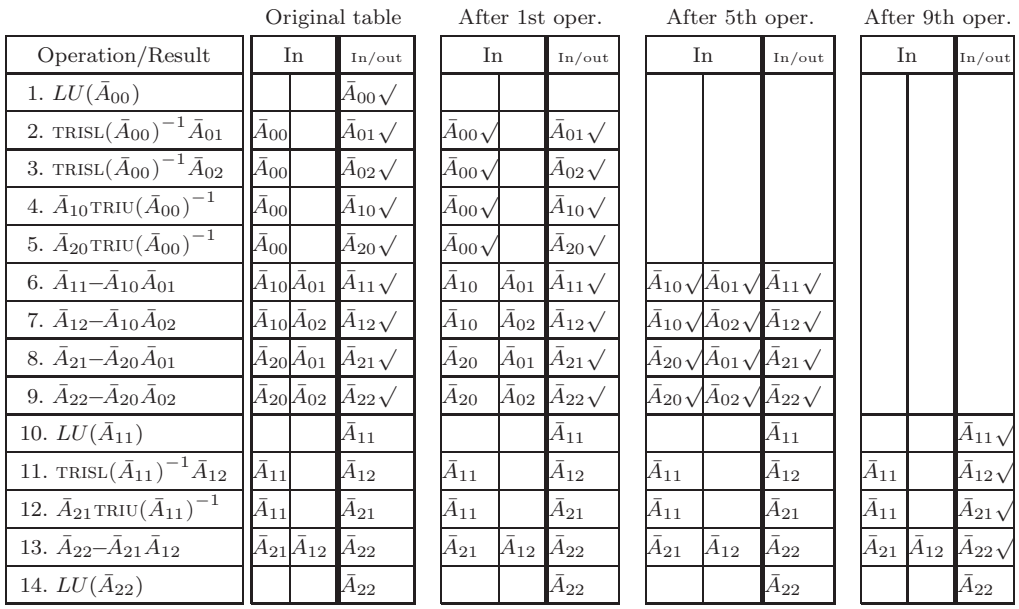


Figure 6: An illustration of the scheduling of operations for the LU factorization (without pivoting) of a 3×3 matrix of blocks using algorithm-by-blocks. Here, $\text{TRIU}(B)$ stands for the upper triangular part of B while $\text{TRISL}(B)$ denotes the matrix consisting of the lower triangular part of B with the diagonal entries replaced by ones.

Thus, we combine two techniques from superscalar processors, dynamic scheduling, and out-of-order execution while hiding the management of data dependencies from both library developers and users. This approach is similar in philosophy to the inspector–executor paradigm for parallelization [27, 38], but that work solves a very different problem. This approach also reflects a shift from control-level parallelism, specified strictly by the order in which operations appear in the code, to data-flow parallelism, restricted only by true data dependencies and availability of compute resources.

5 An Experiment in Programmability: the Level-3 BLAS

In [9] we report on the implementation of the level-3 BLAS using FLASH and SuperMatrix. In this section we briefly summarize the insights from that paper with a primary focus on what it tells us about how the approach addresses the programmability issue.

When we commensed parallelizing the level-3 BLAS, we had a full set of sequential level-3 BLAS implemented using the FLAME API. It is important to realize that a "full set" entails all datatypes[1] and all unblocked and blocked algorithm variants[2] for all operations that constitute the level-3 BLAS. We also had an implementation of the FLASH extension to FLAME and the SuperMatrix runtime system for scheduling the tasks used for the execution of a SuperMatrix-enabled algorithm. This implementation had previously been used only for the Cholesky factorization.

Two of the authors, Ernie Chan and Field Van Zee, spent a weekend implementing and testing the parallelization, yielding a full set of multithreaded level-3 BLAS using FLASH and SuperMatrix. This productivity attests as to how effectively the methodology addresses the programmability issue. Impressive performance is reported in [11] despite the absence of dependencies in many of the reported operations, the lack of which reduces much of the SuperMatrix system to overhead.

[1] This includes single-precision and double-precision for real and complex operations.

[2] The FLAME methodology often yields half a dozen or more algorithms for each operation.

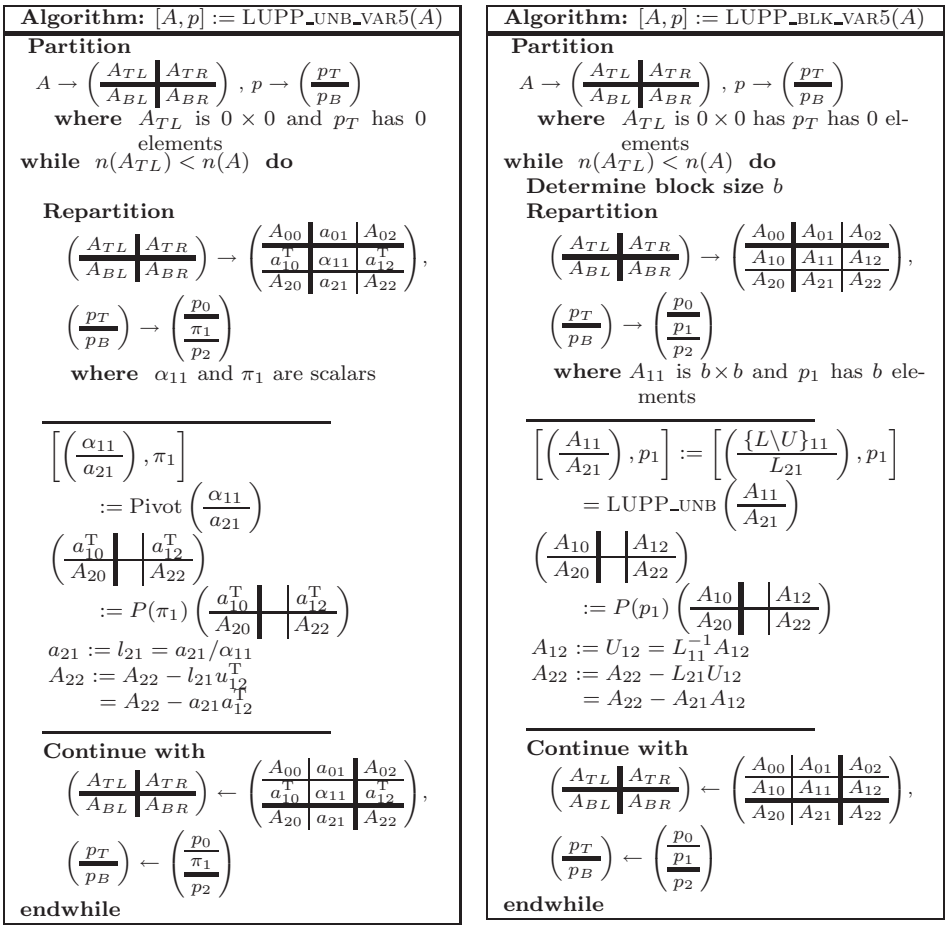


Figure 7: Unblocked and blocked algorithms (left and right, respectively) for computing the LU factorization with partial pivoting. Pivot(v) refers to a function that returns the index of the entry of largest magnitude of a vector v and interchanges that element with the first entry of v. $P(\pi_1)$ and $P(p_1)$ denote permutation matrices.

6 LAPACK-Level Operations: Dense Factorizations

The LAPACK library provides functionality one level above the level-3 BLAS. The subset with which we will primarily concern ourselves in this section includes the LU with pivoting, QR, and Cholesky factorizations.

6.1 An algorithm-by-blocks for the LU factorization with pivoting

It becomes immediately obvious that algorithms-by-blocks for the LU factorization with partial pivoting and the QR factorization based on Householder transformations require us to abandon the tried-and-trusted algorithms incorporated in LAPACK since pivoting information for the former and the computation of Householder transformations for the latter require access to columns that span multiple blocks. Unblocked and blocked algorithms for the LU factorization with partial pivoting, in FLAME notation, are given in Figure 7.

Thus, a second major obstacle to algorithms-by-blocks is that not all operations lend themselves nicely to this class of algorithms, with a clear example being the LU factorization when pivoting for stability enters the picture. We next describe our solution to this problem, inspired by out-of-core tiled algorithms for the QR and LU factorizations [20, 24, 30].

Traditional algorithms for the LU factorization with partial pivoting exhibit the property that, periodically, an updated column is required for a critical computation; in order to compute which row to pivot during the k-th iteration, the k-th column must have been updated with respect to all previous computation. This greatly restricts the order in which the computations can be performed. The problem is compounded by the fact that the column needed for computing which row to pivot, as well as the row to be pivoted, likely span multiple blocks. This need for viewing and/or storing matrices by blocks was also observed for out-of-core dense linear algebra computations [35] and the implementation of dense linear algebra operations on distributed-memory architectures [12, 37].

We will describe how partial pivoting can be modified to facilitate an algorithm-by-blocks. We do so by first reviewing results from [24, 30] that show how an LU factorization can be updated while incorporating pivoting. Afterward, we generalize the insights to the desired algorithm-by-blocks.

6.1.1 Updating an LU factorization

We briefly review how to compute the LU factorization of a matrix A of the form

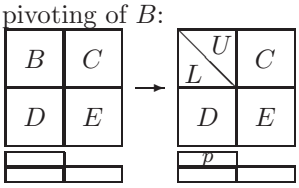
$$A = \left(\begin{array}{c|c} B & C \\ \hline D & E \end{array}\right) \tag{1}$$

in such a way that the LU factorization with partial pivoting of B can be reused if D, C, and E change. In our description, we assume that both B and E are square matrices.

The following procedure [24, 30], consisting of 5 steps, computes an *LU factorization with incremental pivoting* of the matrix in (1):

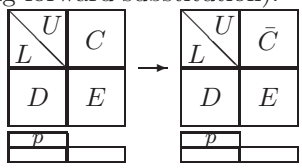
Step 1: Factor B. Compute the LU factorization with partial pivoting of B:

$$[B, p] := [\{L\backslash U\}, p] = \text{LUPP\_BLK}(B).$$



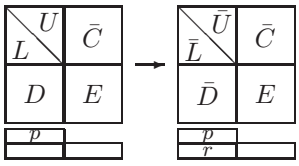
Step 2: Update C consistent with the factorization of B (using forward substitution):

$$\bar{C} := L^{-1}P(p)C = \text{TRSM\_LLNU}(L, P(p)C).$$



Step 3: Factor $\left(\frac{U}{D}\right)$. Compute the LU factorization with partial pivoting:

$$\begin{aligned}\left[\left(\frac{U}{D}\right), r\right] &:= \left[\left(\frac{\{\bar{L}\backslash\bar{U}\}}{D}\right), r\right] \\ &= \text{LUPP\_SA\_BLK}\left(\left(\frac{U}{D}\right)\right).\end{aligned}$$



Here, $\bar{U}$ overwrites the upper triangular part of B (where U was stored before this operation). The lower triangular matrix $\bar{L}$ that results needs to be stored separately since both L, computed in Step 1 and used at Step 2, and $\bar{L}$ are needed during the forward substitution stage when solving a linear system.

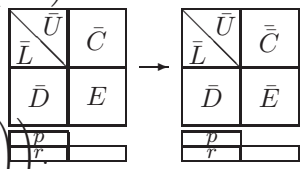
Care must be taken in this step not to completely fill the zeroes below U, which would greatly increase the computational cost of the next step and the storage costs of both this and the next step. The procedure

computes a "structure-aware" (SA) LU factorization with partial pivoting, employing a blocked algorithm that combines the LINPACK and LAPACK styles of pivoting. For details, see algorithm $\mathrm{LU}_{\mathrm{BLK}}^{\mathrm{SA-LIN}}$ in [30].

Step 4: Update $\left(\frac{\bar{C}}{E}\right)$ consistent with the factorization of $\left(\frac{U}{D}\right)$:

$$\left(\frac{\bar{\bar{C}}}{\bar{E}}\right) := \left(\begin{array}{c|c} \bar{L} & 0 \\ \hline \check{L} & I \end{array}\right)^{-1} P(r) \left(\frac{\bar{C}}{E}\right)$$

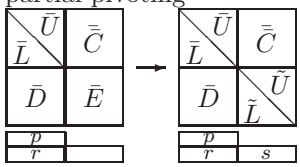
$$= \textsc{Trsm\_SA\_llnu}\left(\left(\begin{array}{c|c} \bar{L} & 0 \\ \hline \check{L} & I \end{array}\right), P(r)\left(\frac{\bar{C}}{E}\right)\right)$$



Again, care must be taken in this step to exploit the zeroes below the diagonal of the upper triangular matrix produced in the previous step. This structure-aware procedure, though not equivalent to a clean triangular system solve (plus the application of the corresponding permutations), can be performed in terms of level-3 BLAS and presents essentially the same computational cost, modulo a lower order term. For details, see algorithm $\mathrm{FS}_{\mathrm{BLK}}^{\mathrm{SA-LIN}}$ in [30].

Step 5: Factor E**.** Finally, compute the LU factorization with partial pivoting

$$\left[\bar{\bar{E}}, s\right] := \left[\{\tilde{L}\backslash\tilde{U}\}, s\right] = \textsc{lupp\_blk}(\bar{E}).$$



Overall, the five steps of the procedure apply Gauss transforms and permutations to reduce A to the upper triangular matrix

$$\left(\begin{array}{c|c} \bar{U} & \bar{C} \\ \hline 0 & \tilde{U} \end{array}\right).$$

6.1.2 An Algorithm-By-Blocks

The insights from the previous section naturally extend to an algorithm-by-blocks for the LU factorization with incremental pivoting [24, 31]. Consider the partitioning by *blocks*

$$A = \begin{pmatrix} A_{00} & A_{01} & \cdots & A_{0,N-1} \\ A_{10} & A_{11} & \cdots & A_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{N-1,0} & A_{N-1,0} & \cdots & A_{N-1,N-1} \end{pmatrix} \tag{2}$$

where, for simplicity, A_{ij}, $0 \leq i, j < N$, are considered to be of size $b \times b$. Then the algorithm in Figure 8 is a generalization of the algorithm described in Section 6.1.1.

While there is some flexibility in the order in which the loops are arranged, the SuperMatrix runtime system, described in Section 4, rearranges the operations, and therefore the exact order of the loops is not important.

6.1.3 Stability

Strictly speaking, the LU factorization with partial pivoting is not numerically stable; theory predicts that so-called element growth proportional to 2^n may occur. It is *practice* that taught us to rely on this method. In [30] we discuss how the stability of incremental pivoting relates to that of partial pivoting. However, as

for $k = 0 : N-1$
  $[A_{kk}, p_{kk}] := \text{LUPP\_BLK}(A_{kk})$
  for $j = k+1 : N-1$
    $A_{kj} := \text{TRSM\_LLNU}(A_{kk}, P(p_{kk})A_{kj})$
  endfor
  for $i = k+1 : N-1$
    $\left[\left(\frac{A_{kk}}{A_{ik}}\right), L_{ik}, p_{ik}\right] := \text{LUPP\_SA\_BLK}\left(\frac{A_{kk}}{A_{ik}}\right)$
    for $j = k+1 : N-1$
      $\left[\left(\frac{A_{kj}}{A_{ij}}\right)\right] := \text{TRSM\_SA\_LLNU}\left(\text{TRIL}\left(\left(\frac{L_{ik} \mid 0}{A_{ik} \mid I}\right)\right),\ P(p_{ik})\left(\frac{A_{kj}}{A_{ij}}\right)\right)$
    endfor
  endfor
endfor

Figure 8: Algorithm-by-blocks for the LU factorization with incremental pivoting. Here, TRIL(B) stands for the lower triangular part of B. The actual implementation is similar to those in Figures 2 (right) and 4, but for conciseness we present it as loops.

was the case for partial pivoting, practical experience will be needed to establish incremental pivoting as being a numerically stable method.

6.2 An algorithm-by-blocks for the QR factorization

The QR factorization of an $m \times n$ matrix is given by $A = QR$, where Q is an $m \times m$ orthogonal matrix and R is an $m \times n$ upper triangular matrix. Although there exist several approaches to compute this factorization, the algorithm based on Householder reflectors [17] is usually chosen when seeking high performance.

The QR factorization based on Householder reflectors and the LU factorization with partial pivoting share the property that, periodically, an updated column is required for a critical computation; in the case of the QR factorization, the k-th column must have been updated with respect to all previous computation before the Householder reflectors that annihilate subdiagonal entries in this column can be computed. This greatly restricts the order in which the computations can be performed.

An algorithm-by-blocks for the QR factorization can be obtained following the out-of-core algorithm in [20].

The algorithm-by-blocks for the QR factorization incurs a certain extra cost when compared with the traditional implementation of the QR factorization via Householder reflectors. This overhead is negligible for matrices of medium and large size. The use of orthogonal transformations ensures that the algorithm-by-blocks and the traditional QR factorization are numerically stable.

For details, see [32].

6.3 An algorithm-by-blocks for the Cholesky factorization

Given a symmetric positive-definite (SPD) matrix A, its Cholesky factorization is given by $A = LL^T$ (or $A = U^T U$), where L is lower triangular (or U is upper triangular). The construction of an algorithm-by-blocks to obtain the Cholesky factorization is straightforward. The algorithm is illustrated in Figure 9 for an SPD matrix partitioned as in (2). On completion, the lower triangular part of A is overwritten by the Cholesky factor L while the strictly upper triangular part of the matrix is not modified.

This algorithm incurs in the same flop count as the traditional implementation for the Cholesky factorization and the two exhibit the same numerical stability properties.

For details, see [9].

```
for k = 0 : N − 1
    [A_kk] := {L_kk \ A_kk} = CHOL_BLK(A_kk)
    for i = k + 1 : N − 1
        A_ik := L_ik = A_ik L_kk^{−T}
        for j = k + 1 : i
            A_ij := A_ij − A_ik A_jk^T
        endfor
    endfor
endfor
```

Figure 9: Algorithm-by-blocks for the Cholesky factorization. CHOL_BLK(B) refers to a blocked algorithm to compute the Cholesky factorization of B. On completion, this algorithm overwrites the lower triangular part of B with its Cholesky factor. The actual implementation is similar to those in Figures 2 (right) and 4, but for conciseness we present it as loops.

7 Advanced LAPACK-Level Operations: Banded Factorization and Inversion of Matrices

In this section, we offer a few comments on the development of algorithms-by-blocks for slightly more complex operations: factorization of a banded matrix and matrix inversion.

7.1 Cholesky factorization of banded matrices

Consider a banded SPD matrix with bandwidth k_d, partitioned into $b \times b$ blocks as in (2) so that nonzero entries only appear in the diagonal blocks A_{kk}, the subdiagonal blocks $A_{k+1,k}, \ldots, A_{\min(N-1,k+D),k}$ and, given the symmetry of the matrix, $A_{k,k+1}, \ldots, A_{k,\min(N-1,k+D)}$. (If we assume for simplicity that $k_d + 1$ is an exact multiple of b then $D = (k_d + 1)/b - 1$.) An algorithm-by-blocks for the Cholesky factorization of this matrix is easily obtained from the algorithm in Figure 9 by changing the upper limit of the second loop to $\min(N - 1, k + D)$.

The ideas extend to provide algorithm-by-blocks for the LU factorization with incremental pivoting and the QR factorization of a banded matrix.

One of the advantages of using FLASH in the implementation of banded algorithm-by-blocks is that storage of a band matrix does not differ from that of a dense matrix. We can still view the matrix as a matrix of matrices but store only those blocks that contain nonzero entries into the structure. Thus, FLASH easily provides compact storage schemes for banded matrices.

For further details, see [33].

7.2 Inversion of SPD matrices

Traditionally, the inversion of an SPD matrix A is performed as a sequence of three stages: compute the Cholesky factorization of the matrix $A = LL^T$; invert the Cholesky factor $L \rightarrow L^{-1}$; and form $A^{-1} := L^{-T}L^{-1}$. An algorithm-by-blocks has been given above for the first stage and two more algorithms-by-blocks can be easily formulated for the second and third stages. The result is an alternative algorithm-by-blocks that yields much higher performance than one which synchronizes all computation after each stage. For further details, see [10].

The same approach provides an algorithm-by-blocks for the inversion of a general matrix via the LU factorization with incremental pivoting.

The authors in [5] show it is possible to compute these three stages concurrently, and that doing so enhances load-balance on distributed memory architectures. Since the run-time system performance the operations on blocks out-of-order, there no benefit results from a one-sweep algorithm.

8 Experimental Results

In this section, we examine various multithreaded codes in order to assess the potential performance benefits offered by algorithms-by-blocks. All experiments were performed using double-precision floating-point arithmetic on two architectures:

- A ccNUMA SGI Altix 350 server consisting of eight nodes, each with two 1.5 GHz Intel Itanium2 processors, providing a total of 16 CPUs and a peak performance of 96 GFLOPs (96×10^9 floating point operations per second). The nodes are connected via an SGI NUMAlink connection ring and collectively access 32 GBytes of general-purpose physical RAM, with 2 GBytes local to each node. Performance was measured by linking to the BLAS in Intel's Math Kernel Library (MKL) version 8.1.

- An SMP server with eight AMD Opteron processors, each one with two cores clocked at 2.2 GHz, providing a total of 16 cores and a peak performance of 70.4 GFLOPs. The cores in this platform share 64 GBytes of general-purpose physical RAM. Performance was measured by linking to the BLAS in Intel Math Kernel Library (MKL) version 9.1.

We report the performance of the following three parallel implementations in Figure 10:

- LAPACK: Routines `dpotrf` (Cholesky factorization), `dgeqrf` (QR factorization), `dgetrf` (LU factorization with partial pivoting), and `dpbtrf` (Cholesky factorization of band matrices) in LAPACK 3.0 linked to multithreaded BLAS in MKL.

- MKL: Multithreaded implementation of routines `dpotrf`, `dgeqrf`, and `dgetrf` in MKL.

- AB: Our implementation of algorithm-by-blocks, with matrices stored hierarchically using the FLASH API, scheduled with the SuperMatrix runtime system and linked to serial BLAS in MKL. The OpenMP implementation provided by the Intel C Compiler served as the underlying threading mechanism used by SuperMatrix on both platforms.

We consider the usual flop counts for the factorizations: $n^3/3$, $4n^3/3$ and $2n^3/3$, respectively, for the Cholesky, QR and LU factorizations of a square matrix of order n. The cost of the Cholesky factorization of a matrix with bandwidth k_d is computed as $n(k_d^2 + 3k_d)$ flops. Note that the algorithms-by-blocks for the QR and LU factorization actually perform a slightly higher number of flops.

When hand-tuning block sizes, an effort was made to determine the best values for all combinations of parallel implementations and BLAS. In the evaluation of the band factorization case, the dimension of the matrix was set to 5000. In this case, we report those results corresponding to the most favorable number of processors/cores for each implementation as using a lower number of resources resulted in some cases in a lower execution time.

The results show that algorithms-by-blocks clearly outperform the codes in LAPACK and are competitive with highly tuned implementations provided by libraries such as MKL.

9 Conclusion

While architectural advances promise to deliver a high level of parallelism in the form of many-core platforms, we argue that it is programmability that will determine the success of these architectures. In this paper we have illustrated how the notation, APIs, and tools that are part of the FLAME project provide modern object-based abstractions to increase developer productivity and user-friendliness alike in the context of dense and banded linear algebra libraries. One of these abstractions targets multi-core architectures by borrowing dynamic out-of-order scheduling techniques from sequential superscalar architectures. Results for the most significative (dense) matrix factorizations on two shared-memory parallel platforms consisting of a relatively large number of processors/cores illustrate the benefits of our approach.

The FLAME project strives to remain forward-looking. By maintaining a clean API design and clear separation of concerns, we streamline the process of taking a new algorithm from whiteboard concept to high-performance parallel implementation. The base FLAME/C API, the FLASH hierarchical matrix extension,

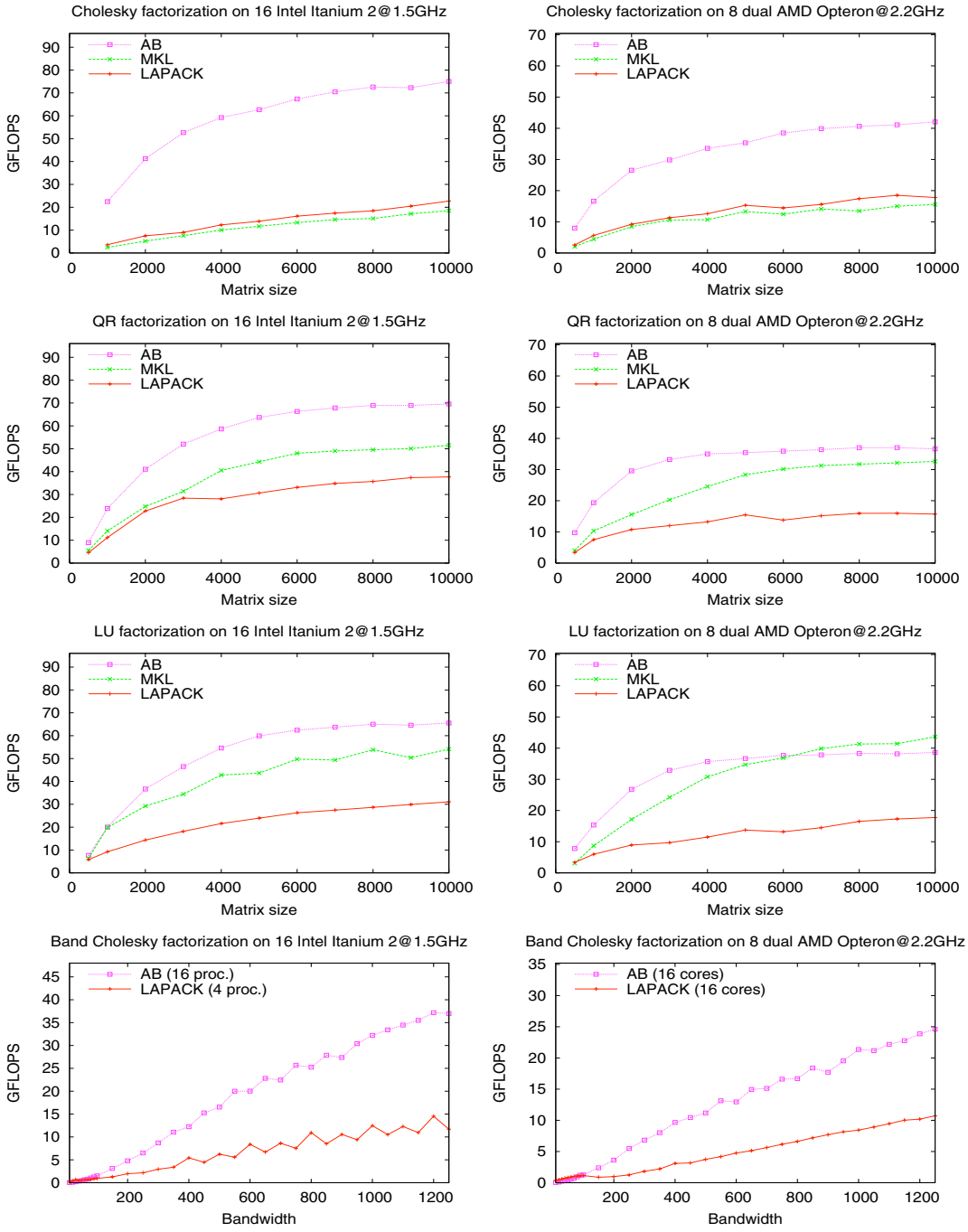


Figure 10: Performance of the multithreaded factorization algorithms.

and the SuperMatrix runtime scheduling and execution mechanism compliment each other through friendly abstractions that facilitate a striking increase in developer-level productivity as well as uncompromising end-user performance.[3]

[3]It is worth pointing out that the FLAME research group's ability to provide a complete production-level alternative to LAPACK has been hampered by its funding mandate. Almost all grants awarded to date have originated from the National

From the beginning, we have separated the SuperMatrix heuristic used for scheduling tasks from the library that implements the linear algebra operations. In [9] we demonstrated the benefits of using different heuristics to schedule suboperations to threads. As part of ongoing efforts, we continue to investigate the effects of different scheduling strategies on overall performance. We do not discuss this topic in the present paper because our desire to focus the present paper squarely on the issue of programmability.

Additional information

For additional information on FLAME visit

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

Acknowledgements

This research was partially sponsored by NSF grants CCF–0540926 and CCF–0702714. Additional support came from the *J. Tinsley Oden Faculty Fellowship Research Program* of the Institute for Computational Engineering and Sciences (ICES) at UT-Austin. Gregorio Quintana-Ortí and Enrique S. Quintana-Ortí were supported by the CICYT project TIN2005-09037-C02-02 and FEDER.

We thank the other members of the FLAME team for their support. We thank John Gilbert and Vikram Aggarwal from the University of California at Santa Barbara for granting access to the NEUMANN platform.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

References

[1] C. Addison, Y. Ren, and M. van Waveren. OpenMP issues arising in the development of parallel blas and lapack libraries. *Scientific Programming*, 11(2), 2003.

[2] R. C. Agarwal and F. G. Gustavson. Vector and parallel algorithms for cholesky factorization on ibm 3090. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 225–233, New York, NY, USA, 1989. ACM Press.

[3] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

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

[5] Paolo Bientinesi, Brian Gunter, and Robert A. Van de Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Soft.*, 2009. to appear.

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

[7] Alfredo Buttari, Julien Langou, Jakub Kurzak, , and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. LAPACK Working Note 190 UT-CS-07-600, University of Tennessee, September 2007.

[8] Alfredo Buttari, Julien Langou, Jakub Kurzak, , and Jack Dongarra. Parallel tiled QR factorization for multicore architectures. LAPACK Working Note 190 UT-CS-07-598, University of Tennessee, July 2007.

Science Foundation, which funds scientific research and proof-of-concept prototypes. While we constantly strive for our studies to result in ready-to-use algorithms and code, we understand the basis for these limitations as they remind us that the fundamental science should always preceed any new tool or product.

[9] Ernie Chan, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *SPAA '07: Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures*, pages 116–125, San Diego, CA, USA, June 9-11 2007a. ACM.

[10] Ernie Chan, Field G. Van Zee, Paolo Bientinesi, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. SuperMatrix: A multithreaded runtime scheduling system for algorithms-by-blocks. In *ACM SIGPLAN 2008 symposium on Principles and practices of parallel programming (PPoPP'08)*, 2008. to appear.

[11] Ernie Chan, Field G. Van Zee, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. Satisfying your dependencies with SuperMatrix. In *Proceedings of IEEE Cluster Computing 2007*, pages 91–99, September 2007b.

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

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

[14] 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.

[15] H. Carter Edwards and Robert A. van de Geijn. On application interfaces to parallel dense matrix libraries: Just let me solve my problem! FLAME Working Note #18 TR-2006-15, The University of Texas at Austin, Department of Computer Sciences, 2006.

[16] 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.

[17] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 3rd edition, 1996.

[18] Kazushige Goto and Robert A. van de Geijn. Anatomy of a high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3), 2008.

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

[20] Brian C. Gunter and Robert A. van de Geijn. Parallel out-of-core computation and updating the QR factorization. *ACM Transactions on Mathematical Software*, 31(1):60–78, March 2005.

[21] Jia Guo, Ganesh Bikshandi, Basilio Fraguela, Maria Garzaran, and David Padua. Programming with tiles. In *PPoPP '08: The 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, USA, 2008.

[22] Fred G. Gustavson, Lars Karlsson, and Bo Kagstrom. Three algorithms for Cholesky factorization on distributed memory using packed storage. In *Applied Parallel Computing. State of the Art in Scientific Computing*, pages 550–559. Springer Berlin / Heidelberg, 2007.

[23] José Ramón Herrero. *A framework for efficient execution of matrix computations*. PhD thesis, Polytechnic University of Catalonia, Spain, 2006.

[24] Thierry Joffrain, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Rapid development of high-performance out-of-core solvers. In *Proceedings of PARA 2004*, number 3732 in LNCS, pages 413–422. Springer-Verlag Berlin Heidelberg, 2005.

[25] Jakub Kurzak and Jack Dongarra. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. LAPACK Working Note 178 UT-CS-06-581, University of Tennessee, September 2006.

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

[27] Honghui Lu, Alan L. Cox, Sandhya Dwarkadas, Ramakrishnan Rajamony, and Willy Zwaenepoel. Compiler and software distributed shared memory support for irregular applications. In *PPOPP '97: Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–56, New York, NY, USA, 1997. ACM Press.

[28] Bryan A. Marker, Field G. Van Zee, Kazushige Goto, Gregorio Quintana-Ortí, and Robert A. van de Geijn. Toward scalable matrix multiply on multithreaded architectures. In *European Conference on Parallel and Distributed Computing*, pages 748–757, February 2007.

[29] J. Nieplocha, R.J. Harrison, and RJ Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10:197–220, 1996.

[30] Enrique S. Quintana-Ortí and Robert van de Geijn. Updating an LU factorization with pivoting. *ACM Trans. Math. Soft.*, 2009. to appear.

[31] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Ernie Chan, Robert van de Geijn, and Field G. Van Zee. Design and scheduling of an algorithm-by-blocks for LU factorization on multithreaded architectures. FLAME Working Note #26 TR-07-50, The University of Texas at Austin, Department of Computer Sciences, September 2007a.

[32] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Ernie Chan, Field G. Van Zee, and Robert A. van de Geijn. Scheduling of QR factorization algorithms on SMP and multi-core architectures. In *16th Euromicro International Conference on Parallel, Distributed and network-based Processing*, 2008. to appear.

[33] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Alfredo Remón, and Robert van de Geijn. Supermatrix for the factorization of band matrices. FLAME Working Note #27 TR-07-51, The University of Texas at Austin, Department of Computer Sciences, September 2007b.

[34] Peter Strazdins. A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. Technical Report TR-CS-98-07, Department of Computer Science, The Australian National University, Canberra 0200 ACT, Australia, 1998.

[35] Sivan Toledo. A survey of out-of-core algorithms in numerical linear algebra. In James Abello and Jeffrey Scott Vitter, editors, *External Memory Algorithms and Visualization*, pages 161–180. American Mathematical Society Press, Providence, RI, 1999.

[36] 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*, 14(10):805–840, 2002.

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

[38] Reinhard von Hanxleden, Ken Kennedy, Charles H. Koelbel, Raja Das, and Joel H. Saltz. Compiler analysis for irregular problems in Fortran D. In *1992 Workshop on Languages and Compilers for Parallel Computing*, number 757, pages 97–111, New Haven, Conn., 1992. Berlin: Springer Verlag.

[39] David S. Wise, Jeremy D. Frens, Yuhong Gu, and Gregory A. Alexander. Language support for morton-order matrices. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 24–33, New York, NY, USA, 2001. ACM Press.