SuperMatrix Out-of-Order Scheduling of Matrix Operations for SMP and Multi-Core Architectures

FLAME Working Note #23

Ernie Chan* Enrique S. Quintana-Ortí† Gregorio Quintana-Ortí†
Robert van de Geijn*

Abstract

We discuss the high-performance parallel implementation and execution of dense linear algebra matrix operations on SMP architectures, with an eye towards multi-core processors with many cores. We argue that traditional implementations, as those incorporated in LAPACK, cannot be easily modified to render high performance as well as scalability on these architectures. The solution we propose is to arrange the data structures and algorithms so that matrix blocks become the fundamental units of data, and operations on these blocks become the fundamental units of computation, resulting in algorithms-by-blocks as opposed to the more traditional blocked algorithms. We show that this facilitates the adoption of techniques akin to dynamic scheduling and out-of-order execution usual in superscalar processors, which we name *SuperMatrix Out-of-Order scheduling*. Performance results on a 16 CPU Itanium2-based server are used to highlight opportunities and issues related to this new approach.

1 Introduction

This paper explores the benefits of storing and indexing matrices by blocks when exploiting shared-memory parallelism on SMP and/or multi-core architectures. For dense linear algebra matrix operations, the observation is made that if blocks are taken to be the units of data, and operations on blocks the units of computation (tasks), then techniques for dynamic scheduling and out-of-order (OO) execution in superscalar processors can be extended, in software, to the systematic management of independent and dependent tasks. A system that facilitates this in a transparent manner both to the library developer and user is discussed, and its potential performance benefits are illustrated with experiments that are specific to the parallelization of the Cholesky factorization.

It has been observed that the storage of matrices by blocks, possibly recursively, has a number of advantages, including better data locality when exploiting one or more levels of memory [8, 13, 23] and compact storage of symmetric/triangular matrices [3]. In [21], it was shown how the FLASH extension of the FLAME Application Programming Interface (API) for the C programming language [6] greatly reduces the complexity of code by viewing matrices stored by blocks as a tree structure of matrices of matrices where the leaf nodes are blocks that can be stored as convenient. The idea of indexing matrices by blocks, recursively, was also successfully explored in [2, 15, 18, 19]. [10, 11, 17]

The main contributions of the present paper include the following:

- Storage by blocks allows submatrices to replace scalars as the basic units of data and operations on blocks as the basic units of computation (tasks). This greatly reduces the complexity of managing data dependencies between, and scheduling of, tasks in a multithreaded environment. This yields algorithms-by-blocks rather than the more customary blocked algorithms.

*Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712, {echan,rvdg}@cs.utexas.edu.
†Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume I, Campus Riu Sec, 12.071, Castellón, Spain, {quintana,gquintan}@icc.uji.es.

- Dynamic scheduling and OO execution techniques, implemented in hardware on superscalar architectures, can be applied to operations on blocks and be implemented in software for matrix operations.
- A concrete example is given on how the FLASH API facilitates the development of algorithms-by-blocks for computing the Cholesky factorization and allows analysis of data dependencies and management of dynamic scheduling to be separated from the coding issues.
- An implementation of various statically-scheduled implementations of the algorithm-by-blocks for the Cholesky factorization shows the potential performance benefits that can be attained.
- A long list of potential research issues that remain to be resolved is provided.

The paper is structured as follows. Section 2 provides a motivating example. A discussion of formulating algorithms that view matrices as a collection of submatrices is in Section 3. Our approach for dynamic scheduling and OO execution of matrix operations is explained in Section 4. Performance results are provided in Section 5. The paper is concluded in Section 6 with directions for future work.

2 A Motivating Example: The Cholesky Factorization

The Cholesky factorization of a symmetric positive definite (SPD) matrix A is given by $A \rightarrow LL^T$ where L is lower triangular. (Alternatively, one could factorize the matrix as $A \rightarrow U^T U$, where U is upper triangular.) We will follow this example to illustrate the main contributions of the paper.

2.1 A typical algorithm

In Figure 1 we give unblocked and blocked algorithms, in FLAME notation [7], for overwriting an SPD matrix A, of which only the lower triangular part is stored, with its Cholesky factor L. 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 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 can achieve high performance since most computation is cast in terms of the *symmetric rank-k update* (SYRK), $A_{22} := A_{22} - \text{TRIL}(A_{21}A_{21}^T)$, which performs $O(b)$ flops for every memop. An implementation of the blocked algorithm coded in a style similar to that employed by LAPACK, via calls to an unblocked Cholesky factorization routines (`DPOFT2`) and level-3 BLAS (`DTRSM` and `DSYRK`), is given in Figure 2 [4, 12]. Using the FLAME C API [6], the equivalent blocked algorithm can be represented in code as presented in Figure 4 (left) [5].

2.2 The trouble with the SMP-style parallelization of blocked algorithms

In Figure 3 we depict the first two iterations of the algorithm. These pictures allow us to describe the problems with *current techniques for parallelizing traditional codes* in a multithreaded environment.

Multithreaded BLAS The first technique pushes the parallelism into multithreaded (SMP-parallel) implementations of the TRSM and SYRK operations, $A_{21} := A_{21}\text{TRIL}(A_{11})^{-T}$ and $A_{22} := A_{22} - \text{TRIL}(A_{21}A_{21}^T)$, respectively. This works well when the matrix is large, and there are relatively few processors.

However, when the ratio of the matrix dimension to the number of processors is low, there is a natural bottleneck. The block size (variables `JB` and b in Figures 2 and 4 (left), respectively) has to be relatively large (in practice, around 128) in order for the bulk of the computation, performed by SYRK (routines `DSYRK` or `FLA_Syrk`), to deliver high performance. As a result, the computation of the Cholesky factorization of A_{11}, performed by only a single processor, keeps high performance from being achieved.

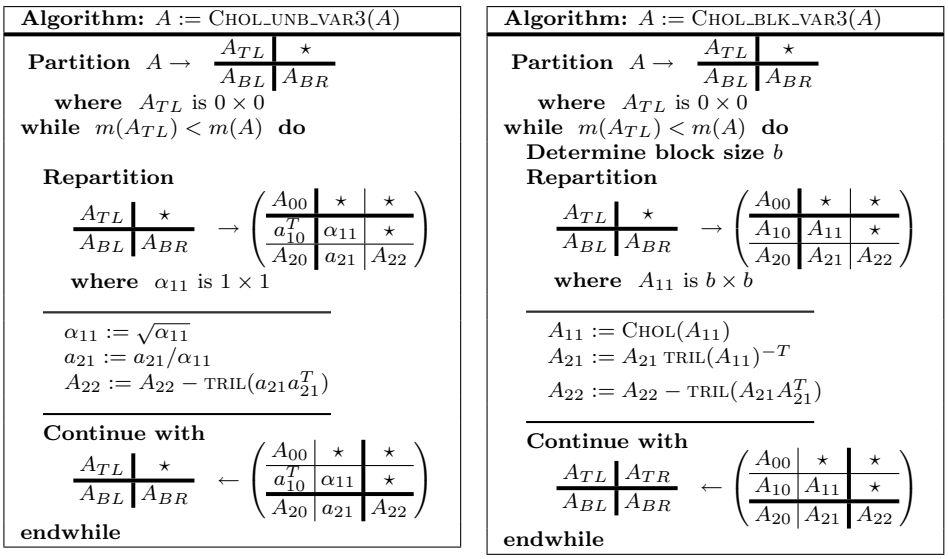

Figure 1: Unblocked and blocked algorithms (left and right, respectively) for computing the Cholesky factorization. There, $m(B)$ stands for the number of rows of B while $\text{TRIL}(B)$ indicates the lower triangular part of B. The '$\star$' symbol denotes entries that are not referenced.

```
DO J = 1, N, NB
  JB = MIN( NB, N-J+1 )
  CALL DPOTF2( 'Lower', JB, A( J, J ), LDA, INFO )
  CALL DTRSM( 'Right', 'Lower', 'Transpose', 'Non-unit',
$               N-J-JB+1, JB, ONE, A( J, J ), LDA,
$               A( J+JB, J ), LDA )
  CALL DSYRK( 'Lower', 'No transpose', N-J-JB+1, JB, -ONE,
$               A( J+JB, J ), LDA, ONE, A( J+JB, J+JB ), LDA )

ENDDO
```

Figure 2: LAPACK-style implementation of the blocked algorithm in Figure 1 (right).

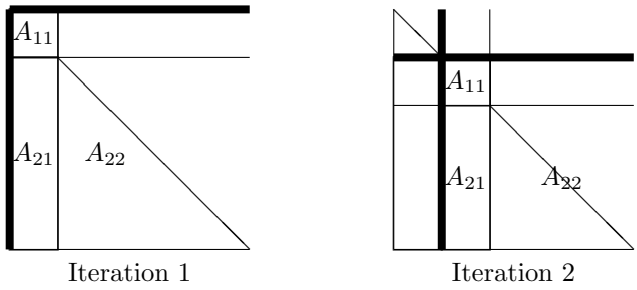

Figure 3: First two iterations of the blocked algorithm in Figure 1 (right).

```
FLA_Error FLA_Chol_blk_var3( 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_length(ATL) < FLA_Obj_length(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_Chol_unb_var3( A11 );
    FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
              FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
              FLA_ONE, A11, A21 );
    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
              FLA_MINUS_ONE, A21, 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_Chol_by_blocks_var3( FLA_Obj A, int nb_alg )
{
  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_length(ATL) < FLA_Obj_length(A) ) {

    FLA_Repart_2x2_to_3x3(
       ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
    /* ************* */   /* ******************** */
                           &A10, /**/ &A11, &A12,
       ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
       1, 1, FLA_BR );
    /*-----------------------------------------------*/
    FLA_Chol_unb_var3( FLASH_MATRIX_AT( A11 ) );
    FLASH_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
                FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
                FLA_ONE, A11, A21 );
    FLASH_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
                FLA_MINUS_ONE, A21, 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 4: Left: FLAME C implementation of the blocked algorithm in Figure 1 (right). Right: FLASH implementation of the algorithm-by-blocks.

Compute-ahead and pipelining A technique that attempts to overcome such a bottleneck is to "compute ahead." In the first iteration, the update of A_{22} 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 will compute ahead several iterations in a similar manner.

The problem with this idea is that it greatly complicates the code that implements the algorithm [1, 20, 24]. While doable for a single, relatively simple algorithm like the Cholesky factorization, reimplementation of an entirely library like LAPACK becomes a daunting task when these techniques are employed.

3 Algorithms-By-Blocks

Fred Gustavson (IBM) has long advocated an alternative to the blocked algorithms in LAPACK: *algorithms-by-blocks*, which are algorithms that view matrices as a collection of submatrices and compute with these submatrices.

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 starting point should be to improve the granularity of the data by replacing each element in the matrix by a submatrix (block). Algorithms should then be as before, except with operations on scalars substituted by operations on the blocks that replaced them. Thus, while a simple triple-nested loop for computing the Cholesky factorization on a matrix of scalars is given in Figure 5 (left), the corresponding algorithm-by-blocks is given in Figure 5 (right).

The motivation for Dr. Gustavson's work is to match an algorithm to an architecture in order to gain high performance [2, 13, 15, 19]. He supports algorithms-by-blocks because of the easy adaptation from standard to blocks algorithms by replacing scalar operations with corresponding blocked operations. Matrices are stored and referenced hierarchically where blocks are aligned to fit within each level of memory.

$$A = \begin{pmatrix} \alpha_{00} & \alpha_{01} & \cdots & \alpha_{0,n-1} \\ \alpha_{10} & \alpha_{11} & \cdots & \alpha_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{n-1,0} & \alpha_{n-1,0} & \cdots & \alpha_{n-1,n-1} \end{pmatrix}$$

for $j = 0, \ldots, n-1$
  $\alpha_{j,j} = \sqrt{\alpha_{j,j}}$
  for $i = j+1, \ldots, n-1$
    $\alpha_{i,j} = \alpha_{i,j} / \alpha_{j,j}$
  endfor
  for $k = j+1, \ldots, n-1$
    for $i = k, \ldots, n-1$
      $\alpha_{i,k} = \alpha_{i,k} - \alpha_{i,j}\alpha_{k,j}$
    endfor
  endfor
endfor

$$A = \begin{pmatrix} A_{00} & A_{01} & \cdots & A_{0,N-1} \\ A_{10} & A_{11} & \cdots & A_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{N-1,0} & A_{N-1,0} & \cdots & A_{N-1,N-1} \end{pmatrix}$$
where $A_{ij} \in \mathbb{R}^{b \times b}$

for $j = 0, \ldots, N-1$
  $A_{j,j} = \text{CHOL}(A_{j,j})$
  for $i = j+1, \ldots, N-1$
    $A_{i,j} = A_{i,j}\text{TRIL}(A_{j,j})^{-T}$
  endfor
  for $k = j+1, \ldots, N-1$
    $A_{k,k} = A_{k,k} - \text{TRIL}(A_{k,j}A_{k,j}^T)$
    for $i = k+1, \ldots, N-1$
      $A_{i,k} = A_{i,k} - A_{i,j}A_{k,j}^T$
    endfor
  endfor
endfor

Figure 5: Left: Algorithm that computes the Cholesky factorization as a triple-nested loop, one element at a time. Right: Algorithm that computes the Cholesky factorization as a triple-nested loop, by blocks.

Our motivation is similar with the high-performance implementation and parallel execution of matrix operations on SMP and multi-core architectures.

3.2 Obstacles

We believe that a major obstacle to algorithms-by-blocks lies with the complexity that is introduced into the code (see examples in Section 2.2). A number of solutions have been proposed to solve this problem, ranging from explicitly exposing intricate indexing into the individual elements to template programming using C++ to compiler solutions [27]. None of these have yielded a consistent methodology that allows the development of high-performance libraries with functionality that rivals LAPACK or FLAME.

3.3 The FLASH API for algorithms-by-blocks

Recent efforts [13, 18, 21] follow an approach different from those mentioned above. They view the matrix as a matrix of matrices, just as it is conceptually described. The FLASH API [21], which is an extension of the FLAME API used in Figure 4 (left), exploits the fact that FLAME encapsulates a matrix in an object, making it easy to allow elements of a matrices to themselves be descriptions of matrices, thus yielding matrices of matrices.

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, then 1) $\sqrt{\alpha_{11}}$ becomes the Cholesky factorization of the matrix at element α_{11}; 2) a_{21} represents a vector of blocks so that a_{21}/α_{11} becomes a triangular solve with multiple right-hand sides with the triangular matrix stored at element α_{11} and each of the blocks in vector a_{21}; and 3) each element of A_{22} describes a block that needs to be updated using blocks from the vector a_{21}. The FLASH code for this algorithm-by-blocks is given in Figure 4 (right).

4 Towards SuperMatrix OO Scheduling

In this section we discuss how techniques used in superscalar processors can be adopted to systematically expose parallelism in algorithms-by-blocks.

	Original table			After first operation			After third operation			After sixth operation		
Operation	In		In/out	In		In/out	In		In/out	In		In/out
$\sqrt{\alpha_{0,0}}$			$\alpha_{0,0}\surd$									
$\alpha_{1,0}/\alpha_{0,0}$	$\alpha_{0,0}$		$\alpha_{1,0}\surd$	$\alpha_{0,0}\surd$		$\alpha_{1,0}\surd$						
$\alpha_{2,0}/\alpha_{0,0}$	$\alpha_{0,0}$		$\alpha_{2,0}\surd$	$\alpha_{0,0}\surd$		$\alpha_{2,0}\surd$						
$\alpha_{1,1}-\alpha_{1,0}\alpha_{1,0}$	$\alpha_{1,0}$		$\alpha_{1,1}\surd$	$\alpha_{1,0}$		$\alpha_{1,1}\surd$	$\alpha_{1,0}\surd$		$\alpha_{1,1}\surd$			
$\alpha_{2,1}-\alpha_{2,0}\alpha_{1,0}$	$\alpha_{1,0}$	$\alpha_{2,0}$	$\alpha_{2,1}\surd$	$\alpha_{1,0}$	$\alpha_{2,0}$	$\alpha_{2,1}\surd$	$\alpha_{1,0}\surd$	$\alpha_{2,0}\surd$	$\alpha_{2,1}\surd$			
$\alpha_{2,2}-\alpha_{2,0}\alpha_{2,0}$	$\alpha_{2,0}$		$\alpha_{2,2}\surd$	$\alpha_{2,0}$		$\alpha_{2,2}\surd$	$\alpha_{2,0}\surd$		$\alpha_{2,2}\surd$			
$\sqrt{\alpha_{1,1}}$			$\alpha_{1,1}$			$\alpha_{1,1}$			$\alpha_{1,1}$			$\alpha_{1,1}\surd$
$\alpha_{2,1}/\alpha_{1,1}$	$\alpha_{1,1}$		$\alpha_{2,1}$	$\alpha_{1,1}$		$\alpha_{2,1}$	$\alpha_{1,1}$		$\alpha_{2,1}$	$\alpha_{1,1}$		$\alpha_{2,1}\surd$
$\alpha_{2,2}-\alpha_{2,1}\alpha_{2,1}$	$\alpha_{2,1}$		$\alpha_{2,2}$	$\alpha_{2,1}$		$\alpha_{2,2}$	$\alpha_{2,1}$		$\alpha_{2,2}$	$\alpha_{2,1}$		$\alpha_{2,2}\surd$
$\sqrt{\alpha_{2,2}}$			$\alpha_{2,2}$			$\alpha_{2,2}$			$\alpha_{2,2}$			$\alpha_{2,2}$

Figure 6: An illustration of Tomasulo's algorithm for the Cholesky factorization of a 3×3 matrix of scalars. A table is built of all operations and their input and output variables. A '$\surd$' check tag indicates the value is available. When all parameters are checked, the operation is scheduled for execution. Upon completing the operation, the output variable are checked everywhere in the table till it appears again as an output variable. Naturally, the scheduling of operations that are ready to be performed affects the exact order in which subsequent operations are identified as ready. In particular, one would expect some level of pipelining to occur.

4.1 Superscalar dynamic scheduling and OO execution

Consider the Cholesky factorization of

$$A = \begin{pmatrix} \alpha_{0,0} & \star & \star \\ \alpha_{1,0} & \alpha_{1,1} & \star \\ \alpha_{2,0} & \alpha_{2,1} & \alpha_{2,2} \end{pmatrix}.$$

The algorithm in Figure 5 (left) more explicitly exposes the operations in the left-most column of Figure 6. A superscalar processor [16] allows operations to be scheduled dynamically, as operands become available, while keeping control of data dependencies. For example, Tomasulo's algorithm [16, 26] keeps track of the availability of input operands (as indicated by '$\surd$' tags in Figure 6) and schedules operations for execution as their operands become available during the computation.

What is important to realize is that such tables can be systematically built regardless of whether the algorithm was expressed in terms of scalar operations, as in in Figure 5 (left), or in terms higher level expressions, as in Figure 1 (left). In both cases the operations to be performed and the data on which it must be performed can be systematically recognized.

4.2 SuperMatrix dynamic scheduling and OO execution

Let us examine what it would take to achieve the same benefits for the algorithm-by-blocks. Consider the partition

$$A \rightarrow \begin{pmatrix} A_{0,0} & \star & \star \\ A_{1,0} & A_{1,1} & \star \\ A_{2,0} & A_{2,1} & A_{2,2} \end{pmatrix}.$$

The code in Figure 4 (right) can be used to identify all the operations to be performed on the blocks, generating the table in Figure 7 during a preliminary stage of the execution. Rather than having the hardware generate the table, this can be done in software, with every call to the operations "enqueuing" the appropriate entries in the table. For example,

```
FLA_Chol_unb_var3( FLASH_MATRIX_AT( A11 ) )
```

inserts the Cholesky factorization of block A_{11} in the table, while operations encountered during this initial stage inside routines `FLASH_Trsm` or `FLASH_Syrk` also enqueue their corresponding entries. The table contains the data dependencies for the matrix operation to be executed and reflects the evolution during the execution of tasks.

Operation	Original table In		In/out	After first operation In		In/out	After third operation In		In/out	After sixth operation In		In/out
$\text{CHOL}(A_{0,0})$			$A_{0,0}\surd$									
$A_{1,0}\text{TRIL}(A_{0,0})^{-T}$	$A_{0,0}$		$A_{1,0}\surd$	$A_{0,0}\surd$		$A_{1,0}\surd$						
$A_{2,0}\text{TRIL}(A_{0,0})^{-T}$	$A_{0,0}$		$A_{2,0}\surd$	$A_{0,0}\surd$		$A_{2,0}\surd$						
$A_{1,1} - \text{TRIL}(A_{1,0}A_{1,0}^T)$	$A_{1,0}$		$A_{1,1}\surd$	$A_{1,0}$		$A_{1,1}\surd$	$A_{1,0}\surd$		$A_{1,1}\surd$			
$A_{2,1} - A_{2,0}A_{1,0}$	$A_{1,0}$	$A_{2,0}$	$A_{2,1}\surd$	$A_{1,0}$	$A_{2,0}$	$A_{2,1}\surd$	$A_{1,0}\surd$	$A_{2,0}\surd$	$A_{2,1}\surd$			
$A_{2,2} - A_{2,0}A_{2,0}$	$A_{2,0}$		$A_{2,2}\surd$	$A_{2,0}$		$A_{2,2}\surd$	$A_{2,0}\surd$		$A_{2,2}\surd$			
$\text{CHOL}(A_{1,1})$			$A_{1,1}$			$A_{1,1}$			$A_{1,1}$			$A_{1,1}\surd$
$A_{2,1}\text{TRIL}(A_{1,1})^{-T}$	$A_{1,1}$		$A_{2,1}$	$A_{1,1}$		$A_{2,1}$	$A_{1,1}$		$A_{2,1}$	$A_{1,1}$		$A_{2,1}\surd$
$A_{2,2} - \text{TRIL}(A_{2,1}A_{2,1}^T)$	$A_{2,1}$		$A_{2,2}$	$A_{2,1}$		$A_{2,2}$	$A_{2,1}$		$A_{2,2}$	$A_{2,1}$		$A_{2,2}\surd$
$\text{CHOL}(A_{2,2})$			$A_{2,2}$			$A_{2,2}$			$A_{2,2}$			$A_{2,2}$

Figure 7: An illustration of Tomasulo's algorithm for the Cholesky factorization of a 3×3 matrix of blocks using the algorithm-by-blocks.

Thus, executing the algorithm in Figure 4 (right) initially enters operations on blocks to be executed in a table. We call this the *analyzer* stage of the algorithm. Next, the operations on the table are dynamically scheduled and executed in a manner similar to Tomasulo's algorithm. We call this the *scheduler/dispatcher* stage. The overhead of the mechanism, now in software, is amortized over a large amount of computation, and therefore its cost can be expected to be within reason. Also, the entire dependence graph of the scheduled computations can be examined since the cost of doing so is again amortized over a lot of computation.

Thus, we propose to combine dynamic scheduling and OO execution while controlling data dependencies in a manner that is transparent to library developers and users. This approach is similar to the inspector–executor paradigm for parallelization [22, 28]. The new 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 resources.

5 Performance

The purpose of the discussion so far has been to show that when algorithms are cast as algorithms-by-blocks and an API is used that allows one to code such algorithms conveniently, superscalar techniques can be borrowed to achieve systematic scheduling of operations on blocks to multiple threads.

In this section, we examine implementations that more directly schedule execution by blocks, specifically for the Cholesky factorization, so that potential performance benefits that will result from SuperMatrix OO scheduling can be assessed.

5.1 Target architecture

Experiments were performed on a 16 CPU Itanium2 server. This NUMA architecture consists of eight nodes with two Intel Itanium2 (1.5 GHz) processors in each. The total RAM is 32 Gbytes, and the nodes are connected via an SGI NUMAlink connection ring.

We used OpenMP as the threading mechanism within the Intel Compiler 9.0. Performance was measured by linking to two different high-performance implementations of the BLAS: the GotoBLAS 1.06 [14] and Intel MKL 8.1 libraries.

5.2 Implementations

We report the performance (in Gigaflops/sec.) of six different parallelizations of the Cholesky factorization. Two extract parallelism from multithreaded BLAS implementations while the other four explicitly deal with the creation and management of tasks, which themselves call sequential BLAS.

LAPACK `dpotrf` (Parallel BLAS) LAPACK 3.0.0 routine `DPOTRF` (Cholesky factorization) was linked to multithreaded BLAS.

FLAME V3 (Parallel BLAS) The blocked code in Figure 4 (left) linked to multithreaded BLAS.

Pipelined algorithm (Serial BLAS) Our implementation of the first algorithm in [1] which includes compute-ahead and pipelining. We made a best-effort attempt to incorporate optimizations similar to those for other implementations.

Data-flow + NO data affinity (Serial BLAS) This implementation views the threads as forming a pool of resources. There is a single queue of ready tasks that all threads access to acquire work. No attempt is made to schedule tasks that are on the critical path earlier.

For this implementation, the matrices are stored in the traditional column-major order, and blocks are references into these matrices. Thus, blocks are not contiguous in memory.

Data-flow + 2D data affinity (Serial BLAS) Same as the previous implementation, except blocks are logically assigned to threads in a two-dimensional block-cyclic manner, much like ScaLAPACK [9] does on distributed-memory architectures. A thread performs all tasks that write to a particular block [25] to improve locality of data to processors.

This concept of *data affinity* is fundamentally different than CPU affinity where threads are bound to specific processors. CPU affinity is done implicitly by each of our implementations.

Data-flow + 2D data affinity + contiguous blocks (Serial BLAS) Same as the previous implementation, except that now blocks are stored contiguously.

When hand-tuning block sizes, a best-effort was made to determine the best block size for all combinations of parallel implementations and BLAS.

A number of implementations that used 1D (both row-wise or column-wise) cyclic assignment of blocks to threads (e.g., A_{ij} is assigned to thread $j \ mod \ p$ where p equals the number of threads) with various storage options were also examined. The described 2D cyclic assignment yielded the best performance.

5.3 Results

Performance results when linking to the GotoBLAS and MKL libraries are reported in Figure 8. A few comments are due:

- The LAPACK implementation, even when the block size is hand-tuned, performs poorly. This is due to the fact that the algorithm chosen by LAPACK is the so-called left-looking variant, which is rich in calls to `DGEMM` with a small "m" dimension (in $C := C - AB$ matrix B consists of a relatively small number of columns). This shape of matrix-matrix multiplication does not parallelize well. We note that when this same algorithm is coded with the FLAME API, performance is virtually identical to that of the LAPACK implementation.
- The GotoBLAS parallel BLAS are tuned for large matrices. For this reason, asymptotically it is FLAME V3 that performs best in Figure 8 (top). The multithreaded matrix-matrix multiply provided by MKL performs much worse for this algorithm.
- Data affinity and contiguous storage by blocks are clear winners relative to the same algorithms that do not employ both of these optimizations.
- The pipelined algorithm from [1] does not perform nearly as well as the data-flow algorithms proposed in this paper.
- The level-3 BLAS provided by the MKL library perform much better for small matrices than their counterparts from the GotoBLAS.

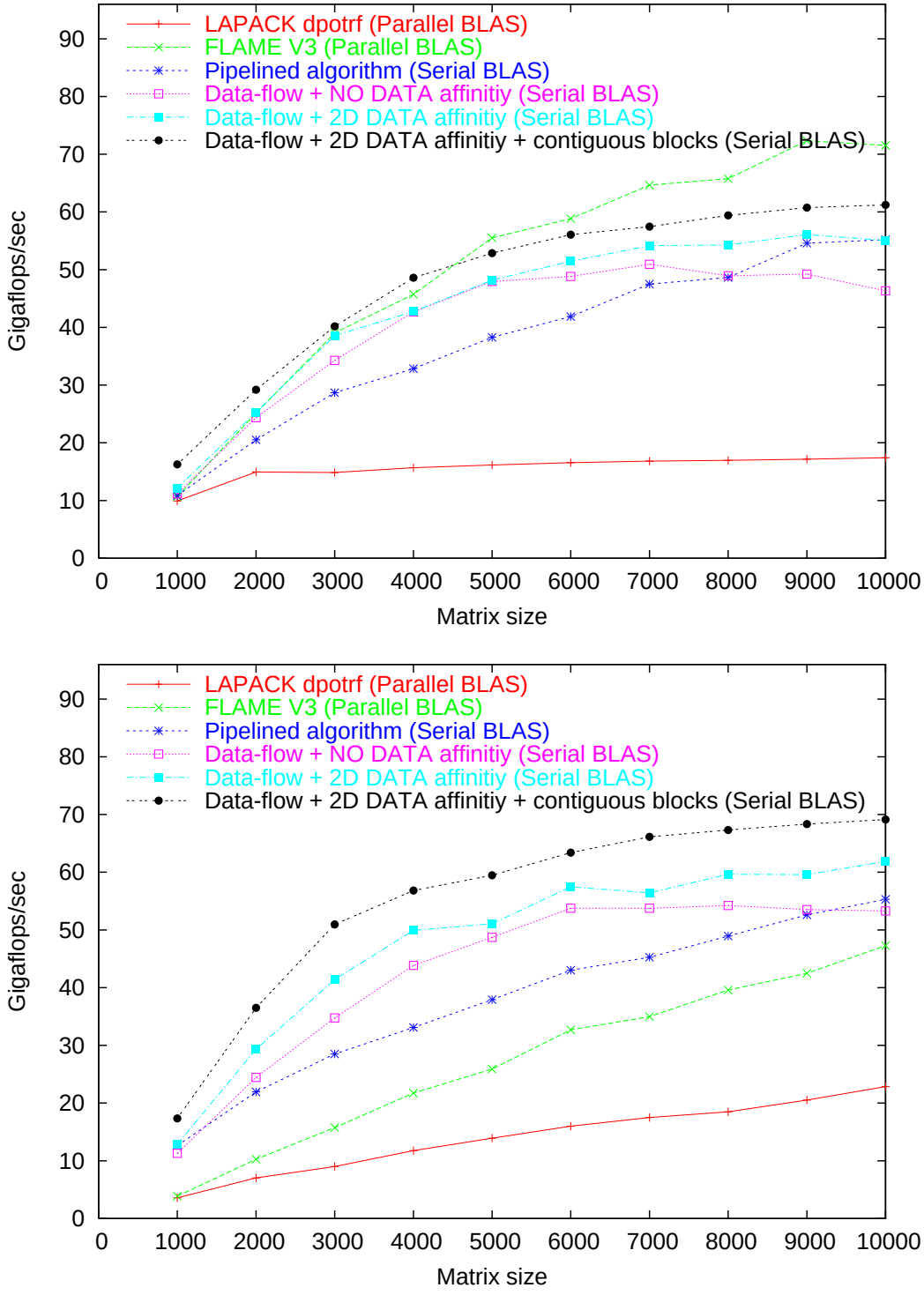

Figure 8: Top: Performance on 16 CPUs when linking to GotoBLAS 1.06. Bottom: Performance on 16 CPUs when linking to MKL 8.1.

6 Conclusion

The results in Figure 8 clearly demonstrate the opportunities that arise from computing the Cholesky factorization by blocks. When linked to a BLAS library that performs well on small submatrices, not only very good asymptotic performance is demonstrated, but more importantly performance ramps up quickly since parallelism is exposed at a relatively small granularity.

The implementations that yield the better performance decompose the Cholesky factorization into its component blocked operations and manage the dependencies among those operations explicitly. Parallelism is exploited through the inherent data flow of those matrix operations instead of being derived from the control flow specified by the program order of operations. While this is quite doable for an individual operation, it becomes cumbersome when entire libraries with functionality similar to the BLAS or LAPACK are to be parallelized for SMP or multi-core architectures. For this reason we introduce the FLASH API and the analyzer–scheduler/dispatcher mechanism for scheduling operations on submatrices. This methodology presents high-level abstractions that shield the library developer from the details of scheduling while generating the operations and dependencies. This yields a clean separation of concern between the high-level algorithm on one hand and the scheduling/execution on the other.

Experience with the FLAME library for sequential architectures tells us that the resulting methodology will allow libraries with functionality similar to the BLAS and LAPACK to be quickly developed. We believe the same cannot be said when code is developed in the tradition of LAPACK (Figure 2). Clearly operations like LU with partial pivoting and Householder QR factorization will still pose special challenges to be studied in future work.

Since the authors interact closely with Kazushige Goto, author of the GotoBLAS, there is a further opportunity to develop high-performance matrix computation kernels specifically in support of the operations that are now performed on submatrices that are stored contiguously. This is part of future research, as is the completion of the analyzer–scheduler/dispatcher and a prototype library with broad functionality.

Additional information

For additional information on FLAME visit

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

Acknowledgements

This research was partially sponsored by NSF grant CCF–0540926. We thank the other members of the FLAME team for their support.

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] Bjarne Stig Andersen, Jerzy Waśniewski, and Fred G. Gustavson. A recursive formulation of Cholesky factorization of a matrix in packed storage. *ACM Trans. Math. Soft.*, 27(2):214–244, 2001.

[4] 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.

[5] Paolo Bientinesi, Brian Gunter, and Robert van de Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. FLAME Working Note #19 TR-2006-20, The University of Texas at Austin, Department of Computer Sciences, 2006.

[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] Paolo Bientinesi and Robert A. van de Geijn. Representing dense linear algebra algorithms: A farewell to indices. FLAME Working Note #17 TR-2006-10, The University of Texas at Austin, Department of Computer Sciences, 2006.

[8] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi. Recursive array layouts and fast matrix multiplication. *IEEE Trans. on Parallel and Distributed Systems*, 13(11):1105–1123, 2002.

[9] 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.

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

[11] Timothy Scott Collins. *Efficient Matrix Computations through Hierarchical Type Specifications.* PhD thesis, The University of Texas at Austin, 1996.

[12] 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.

[13] 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.

[14] K. Goto. `http://www.tacc.utexas.edu/resources/software`.

[15] F. G. Gustavson, L. Karlsson, and B. Kagstrom. Three algorithms on distributed memory using packed storage. *Computational Science – Para 2006.* Bo Kagstrom, E. Elmroth, eds., accepted for Lecture Notes in Computer Science. Springer-Verlag, 2007.

[16] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Pub., San Francisco, 2003.

[17] Greg Henry. BLAS based on block data structures. Theory Center Technical Report CTC92TR89, Cornell University, Feb. 1992.

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

[19] IBM. IBM Engineering and Scientific Subroutine Library for AIX Version 3, Release 3. *IBM Pub. No. SA22–7272–04*, December 2001.

[20] 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.

[21] 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.

[22] 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.

[23] N. Park, B. Hong, and V. K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Trans. on Parallel and Distributed Systems*, 14(7):640–654, 2003.

[24] 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.

[25] Radhika Thekkath and Susan J. Eggers. Impact of sharing-based thread placement on multithreaded architecture. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 176–186, 1994.

[26] R. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. of Research and Development*, 11(1), 1967.

[27] 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.

[28] 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.